NEWLY AVAILABLE SUPPLEMENT
TO THE CLASSIC WORK

The MMIX Supplement

Supplement to
The Art of Computer Programming
Volumes 1, 2, 3
by Donald E. Knuth

MARTIN RUCKERT

NEWLY AVAILABLE SUPPLEMENT
TO THE CLASSIC WORK

The MMIX Supplement

Supplement to
The Art of Computer Programming
Volumes 1, 2, 3
by Donald E. Knuth

MARTIN RUCKERT

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

In this eBook, the limitations of the ePUB format have caused us to render some equations as text and others as images, depending on the complexity of the equation. This can result in an odd juxtaposition in cases where the same variables appear as part of both a text presentation and an image presentation. However, the author's intent is clear and in both cases the equations are legible.

THE MMIX SUPPLEMENT
Supplement to The Art of Computer Programming Volumes 1, 2, 3
by Donald E. Knuth MARTIN RUCKERT

Munich University of Applied Sciences
ADDISON–WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco New York • Toronto • Montréal • London • Munich • Paris • Madrid Capetown • Sydney • Tokyo • Singapore • Mexico City

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries,
please contact governmentsales@pearsoned.com.

For questions about sales outside the United States,
please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

```
Ruckert, Martin.
  The MMIX supplement : supplement to The art of computer
programming,
volumes 1, 2, 3 by Donald E. Knuth / Martin Ruckert, Munich
University
of Applied Sciences.
      pages    cm
  Includes index.
  ISBN 978-0-13-399231-1 (pbk. : alk. paper) -- ISBN 0-13-399231-4
(pbk.
: alk. paper)
  1. MMIX (Computer architecture) 2. Assembly languages (Electronic
computers) 3. Microcomputers--Programming. I. Knuth, Donald Ervin,
1938-. Art of computer programming. II. Title.
QA76.6 .K64 2005 Suppl. 1
005.1--
dc23                                                        2014045485
```

Internet page http://mmix.cs.hm.edu/supplement.html contains current information about this book, downloadable software, and general news about MMIX. See also http://www-cs-faculty.stanford.edu/~knuth/taocp.html for information about *The Art of Computer Programming* by Donald E. Knuth.

ISBN-13: 978-0-13-399231-1
ISBN-10: 0-13-399231-4

Text printed in the United States on recycled paper at Courier in Kendallville, Indiana.

First printing, February 2015

FOREWORD

WHY ARE SOME programmers so much better than others? What is the magical ingredient that makes it possible for some people to resonate with computers so well, and to reach new heights of performance? Many different skills are clearly involved. But after decades of observation I've come to believe that one particular talent stands out among the world-class programmers I've known—namely, an ability to move effortlessly between different levels of abstraction.

That may sound like a scary and complex thing, inherently abstract in itself, but I think it's not really too hard to explain. A programmer must deal with high-level concepts related to a problem area, with low-level concepts related to basic steps of computation, and with numerous levels in between. We represent reality by creating structures that are composed of progressively simpler and simpler parts. We don't only need to understand how those parts fit together; we also need to be able somehow to envision the whole show—to see everything in the large while seeing it simultaneously in the small and in the middle. Without blinking an eye, we need to understand why a major goal can be accomplished if we begin by increasing the contents of a lowly computer register by 1.

The best way to enhance our level-jumping skills is to exercise them frequently. And I believe the most effective strategy for that is to repeatedly examine the details of what goes on at the hardware level when a sophisticated algorithm is being implemented at a conceptual level. In the preface to Volume 1 of *The Art of Computer Programming*, I listed six reasons for choosing to discuss machine-oriented details together with high-level abstractions, integrating both aspects as I was presenting fundamental paradigms and algorithms of computer science. I still like those six reasons. But in retrospect I see now that I was actually blind to the most important reason—that is, the pedagogical reason: I know of no better way to teach a student to think like a top computer scientist than to ground everything in a firm knowledge of how a computing machine actually works. This bottom-up approach seems to be the best way to help nurture an ability to navigate fluently between levels. Indeed, Tony Hoare once told me that I should never even think of condensing these books by removing the machine-language parts, because of their educational value.

I am thrilled to see the present book by Martin Ruckert: It is jam-packed with goodies from which an extraordinary amount can be learned. Martin has not merely transcribed my early programs for `MIX` and recast them in a modern idiom. He has penetrated to their essence and rendered them anew with elegance and good taste. His carefully checked codes represent a significant contribution to the art of pedagogy as well as to the art of programming. Although I myself rarely write machine-level instructions nowadays, my experiences of doing so in the past have provided an indispensable boost to the quality of everything that I now am undertaking. So I encourage serious programmers everywhere to sharpen their skills by devouring this book.

D. E. K.

Stanford, California
December 2014

PREFACE

TRANSLATIONS are made to bring important works of literature closer to those reading—and thinking—in a different language. The challenge of translating is finding new words, phrases, or modes of expression without changing what was said before. An easy task, you may think, when the translation asks only for replacing one programming language with another. Wouldn't a simple compiler suffice to do the job? The answer is Yes, as long as the translated programs are intended to be executed by a machine; the answer is No, if the translated programs are intended to explain concepts, ideas, limitations, tricks, and techniques to a human reader. *The Art of Computer Programming* by Donald E. Knuth starts out by describing the "process of preparing programs for a digital computer" as "an aesthetic experience much like composing poetry or music." That raises the level of expectation to a point where a translation becomes a formidable challenge.

In 1990, the mythical `MIX` computer used for the exposition of implementation details in *The Art of Computer Programming* was so outdated that Knuth decided to replace it. The design of the new `MMIX` computer was finally published as a fascicle, comprising a replacement for the description of `MIX` in Chapter 1 of that series of books. It made the translation of all the `MIX` programs to `MMIX` programs in Volumes 1, 2, and 3 inevitable; but Knuth decided that it would be more important to complete Volumes 4 and 5 first before starting to rewrite Volumes 1–3. Volume 4 meanwhile has grown and by now is to be delivered in (at least) three installments, Volumes 4A, 4B, and 4C, of which the first has already appeared in print. Still it means we have to exercise patience until the new edition of Volume 1 will be published.

With the introduction of the new `MMIX`, Knuth asked programmers who would like to help with the conversion process to join the `MMIX`masters, a loose group of volunteers organized and coordinated by Vladimir Ivanović. However, progress was slow, so in the fall of 2011, when I took over the maintenance of the `MMIX` home page, I decided to take on the task of translating all the remaining programs and update them to a readable form. The result of that effort is the present book, which is intended to be a bridge into the future although not the future itself. It is supplementing Volumes 1, 2, and 3 for those who do not want to wait several more years until that future arrives.

This book is not written for independent reading; it is a *supplement*, supplementing the reading of another book. You should read it side by side with *The Art of Computer Programming* (let's call that "the original" for short). Therefore it is sprinkled with page references such as "[*123*]" pointing the reader to the exact page (in the third edition of Volumes 1 and 2, and in the second edition of Volume 3) where the `MIX` version can be found in the original. References are also included in the headings to simplify searching for a translation given the page number in the original. Further, I tried to pick up a sentence or two unchanged from the original before switching to `MMIX` mode. I also tried to preserve, even in `MMIX` mode, the wording of the original as closely as possible, changing as little as possible and as much as needed. Of course, all section names and their numbering, as well as the numbers of tables, figures, or equations are taken unchanged from the original. It should help you find the point where the translation should be spliced in with the original.

When I assume that you are reading this book in parallel with the original, strictly speaking, I assume that you are reading the original as augmented by the above-mentioned Fascicle 1. A basic knowledge of the `MMIX` computer and its assembly language as explained there is indispensable for an understanding of the material presented here. If you want to know every detail, you should consult `MMIX`*ware* [*Lecture Notes in Computer Science* **1750**, Springer Verlag, as updated in 2014].

Also online you can find plenty of documentation; the `MMIX` home page at http://mmix.cs.hm.edu provides full documentation and current sources of the `MMIX`ware package. Further, the tools mentioned below, other useful `MMIX`-related software, and all the programs presented in this book, including test cases, are available for download. The best companion of `MMIX` theory is `MMIX` practice—so download the software, run the programs, and see for yourself.

This book is written using the TeX typesetting system. To display `MMIX` code in print, it is therefore needed in TeX format; however, to assemble and test `MMIX` code, it is needed in `MMIX` assembly language. An automatic converter, `mmstotex`, was used to produce (almost all) TeX code in the book from the same file that was submitted to the `MMIX` assembler. Another tool, `testgen`, was written just for the production of this book: It combines a set of source files, containing program fragments and test case descriptions, with library code to produce a sequence of complete, ready-to-run test programs.

Great care was taken to complement the programs shown in this book with appropriate test cases. Every line of code you see on the following pages was checked by the `MMIX` assembler for syntactic correctness and executed at least once in a test case. While I am sure that no errors could creep in by manual preparation of TeX sources, that by no means implies that the `MMIX` code shown is error free. Of course, it is not only possible but most likely that several bugs are still hidden in the about 15,000 lines of code written for this book. So please help in finding them!

Thanks to Donald Knuth, I have several boxes of nice `MMIX` T-shirts (sizes L and XL) sitting on the shelf in my office, and I will gladly send one to each first finder of a bug—technical, typographical, orthographical, grammatical, or otherwise—as long as supplies last (T-shirts, not bugs). Known bugs will be listed on the `MMIX` home page, so check there first before sending me an email.

Aside from tracking down bugs, the test cases helped me a lot while conducting experiments with the code because I could see immediately how changes affected correctness and running time. Think of the public test cases as an invitation to do your own experiments. Let me know about your findings, be it an improvement to the code or a new test case to uncover a hidden bug.

Speaking about experiments: Of course it was tempting to experiment with the pipeline meta simulator `mmmix`. The temptation was irresistible, especially since it is so easy to take the existing programs, run them on the pipeline simulator, and investigate the influence of configuration parameters on the running time. But in the end, I had to stop any work on this wide-open field of research and decided to postpone a discussion of pipelined execution. It would have made this booklet into a book and delayed its publication for years.

I am extremely grateful to Donald Knuth, who supported me in every aspect of preparing this book. The draft version, which I sent to him at Stanford, came back three months later with dozens of handwritten remarks on nearly every page, ranging from typographic details such as: "*Here I would put a* `\hair` *between* `SIZE` *and ;*" to questions of exposition: "*No, you've got to leave this tag bit 0. Other exercises depend on it (and so does illustration (*10*))*", wrong instruction counts: "*Should be* $A + 1_{[A]}$", suggestions: "*Did you consider keeping* 2^b *instead of b in a register?*", and bug fixes: "`SRU` *or you'll be propagating a 'minus' sign.*" Without him, this book would not have been written in the first place, and without him, it would also not have reached its present form. For the remaining shortcomings, errors, and omissions, I take full responsibility. I hope that there are not too many left, and that you will enjoy the book all the same.

Martin Ruckert

München
December 2014

STYLE GUIDE

1. NAMES

Choosing good names is one of the most important and most difficult tasks when writing programs, especially if the programs are intended for publication. Good names need to be consistent and so this section starts with some simple rules that guided how names in this book were chosen.

Small named constants, for instance, have all uppercase names such as `FACEUP`. Special cases of this rule are the offsets of fields inside records such as `NEXT` or `TAG` (see 2.1–(1) and 2.1–(5)). Addresses are associated with names that always start with an uppercase letter and continue with uppercase or lowercase letters. Examples are '`TOP OCTA 1F`' and '`Main SET i,0`'. In contrast, names for registers use only lowercase letters, as in `x`, `t`, or `new`.

As a short example illustrating these rules, consider the solution to exercise 2.1–9 on page 123. The address where the printing subroutine starts has the name `:PrintPile` (an explanation for the colon follows below), and the location where the string is stored is named `String`. The constant `#0a`, the ASCII newline character, is named `NL`; every node has a `CARD` field at offset 8, and when the value of this field is loaded into a register, this register has the name `card`.

Often the statement of algorithms has a more mathematical nature. In mathematical language most variables have single-letter names that are set in italic font, such as x, y, Q, or even Q', f_0, or α. In the actual program these variables might look like `x`, `y`, `Q`, `Qp`, `f0`, or `alpha`. The single-letter style of mathematics leads to rather terse programs. This style is appropriate if the exposition is mostly mathematical and the implementation has to convince the reader that it embodies the right mathematics. If the program describes the manipulation of "real-world objects," a more verbose style using descriptive names such as `card` or `title` will improve readability.

In this book, the ultimate aim of choosing a specific name for an address, register, or constant is to make the transition from the algorithms and `MIX` programs, as given in *The Art of Computer Programming*, to their implementations as `MMIX` programs as painless as possible.

One difficulty arises from the fact that the `MIX` assembly language did not provide named registers but only named memory locations; further, names consisted of uppercase letters only. So when an algorithm mentions the variable `X`, there is the silent assumption that if the corresponding `MIX` program uses `X`, it names a memory location where the value of the variable `X` is stored. In `MMIX` programs, names for memory locations are quite rare, because all load and store instructions require registers to compute the target address. Therefore, it is most likely that you will not find `X` in the corresponding `MMIX` program; instead you will find a register, named `x`, that contains the address of the memory location where the variable `X` resides. Taking this one step further, often there is no need to store the value of variable `X` in memory at all; instead, it is completely sufficient to keep the value of `X` in register `x` for the entire program or subroutine. As an example, consider again the solution to exercise 2.1–9. The line that read

```
LD2    0,2(NEXT)    Set X ← NEXT(X).
```

in the `MIX` program on page [*535*] now reads as follows in the `MMIX` program:

```
LDOU   x,x,NEXT     Set X ← NEXT(X).
```

2. TEMPORARIES

There is one special variable, named `t`, which is used as a temporary variable (hence `t`). It is used to carry intermediate values from one instruction to the next and there is no benefit in giving it an individual name. In a few cases, where the name `t` is used already in the exposition of the algorithm, `x` is used to name the temporary variable.

The specific register number used for one of the named registers is typically not relevant; in connection with the `PUSHJ` instruction, however, all named local registers will have register numbers smaller than `t` such that the subroutine call '`PUSHJ t,...`' will not clobber any of them—except `t`, which might hold the return value.

3. INDEX VARIABLES

The variables used to index arrays fall into a special class. If the exposition of an algorithm refers to x_i for $1 \le i \le n$, we might expect a register `xi` (the value of x_i), a register `x` (the address of the array), and a register `i` (the index) to show up somewhere in the implementation. Often, however, the implementation will find it more convenient to maintain in register `i` the value of $8 \times i$ (the offset of x_i relative to `LOC(`x_0`)`), or $8 \times (i - 1)$ (the offset of x_i relative to `LOC(`x_1`)`), or even $8 \times (i - n)$ (the offset of x_i relative to `LOC(`x_n`)`). In the latter case (see below), it is also more convenient to change the value of `x` to `x` $+ 8n$. In all these cases, the use of `x` (not `X`) and `i` (not i) will remind the reader that the registers `x` and `i` are not exactly the variables `X` and i. For a short example see the solution to exercise 4.3.1–25 on page 157.

4. REGISTER NUMBERS

Typically, it is best to avoid the use of register numbers, but instead use register names. There are, though, a few exceptions.

When using `TRIP` and `TRAP` instructions, register `$255` has a special purpose: It serves as parameter register. For the reader of a program, there is some useful information in the fact that a value is stored in `$255`: It will serve as a parameter to the next `TRAP` or `TRIP`. This information should not be hidden by using an alias for `$255`. Similarly, using the return value from a `TRAP` or `TRIP` can be made explicit by using `$255`. For an example see Program 1.3.3A on page 1.

Further, the return value of a function must be in register `$0` just before the final `POP`. Identifying the register by its number makes the assignment of a return value visible. For an example see again the solution to exercise 4.3.1–25.

The program in Section 2.2.5 is special, however: Due to the restrictions imposed by its very simple implementation of coroutines, this program can use local registers only for temporary variables. Consequently, there is no need to give them names.

5. LOCAL NAME SPACES

If programs have multiple subroutines, name conflicts will be inevitable—unless the pseudo-instruction `PREFIX` is used. In this book, every subroutine is given its own name space by starting it with '`PREFIX :`*name*`:`', where *name* repeats the name of the subroutine itself. (See, for example, the solution to exercise 5–7 on page 162.)

The use of two colons, one before and one after '*name*', begs for an explanation. Without the first colon, '*name*`:`' would just be added to the current prefix, leading to longer and longer prefixes unless the prefix is reset regularly by '`PREFIX :`'. Adding a colon before '*name*' is the safer and more convenient alternative. To explain the second colon, imagine using the label '`Put`'—without defining it—after '`PREFIX :Out`'; then `MMIXAL` will complain about an undefined symbol '`OutPut`'. In a long program, this error might be hard to diagnose. Could it be a misspelling of '`Output`'? It becomes really hard to track down such an error if your program contains an unrelated global symbol '`OutPut`'; `MMIXAL` will use it without notice. The colon after '*name*' will prevent `MMIXAL` from confusing global and local names and will make error messages, like a complaint about '`Out:Put`', more readable.

In order to avoid a clumsy `:`*name*`:`*name* in the calling code, the entry point into the subroutine is marked by `:`*name*, making it global. A short example is the

`ShiftLeft` subroutine shown in the solution to exercise 4.3.1–25. The entry point is usually the only global name defined by a subroutine. However, the subroutine might use quite a few global names, defined elsewhere, to reference other subroutines, global registers, or special registers such as `:rJ`. In these cases, the extra colon in front of the name is a useful hint that the name belongs to a global entity; as an added benefit, it allows us to say '`rJ IS $0`' and use `rJ` to keep a local copy of `:rJ`.

Not typical, but occasionally useful, is a joint name space for multiple subroutines. For example, in the simulation program of Section 2.2.5, the routines `Insert` and `Delete` (lines 059–072 on page 30) share the same name space.

To leave the local name space and return to the global name space, a simple '`PREFIX :`' is sufficient.

Because name spaces are merely a technicality, in most of the program listings in this book, the `PREFIX` instructions are not shown.

6. INSTRUCTION COUNTS

For the analysis of algorithms, a column of instruction counts is added to the program display. (Actually, line counts are shown. In the rare cases where several instructions share a single line of code, the instruction counts are easier to read if multiple instructions are treated as one single-but-complex instruction that is counted once.) Instruction counts are shown rather than cycle counts because the former are easier to read and because there is no simple way to determine the latter. For a superscalar pipeline processor such as `MMIX`, the number of cycles per instruction depends on many, many factors. To further complicate the issue, `MMIX` can be configured to mimic a wide variety of processors. Therefore, the running time is approximated by counting υ and μ, where 1υ is approximately one cycle and 1μ is one access to main memory. Most `MMIX` instructions require 1υ; the most important exceptions are load and store instructions ($1\upsilon + 1\mu$), multiplication (10υ), division (60υ), most floating point instructions (4υ), `POP` (3υ), `TRIP` (5υ), and `TRAP` (5υ).

For branch instructions, the number of bad guesses is given in square brackets. So $m_{[n]}$ will label a branch that is executed m times with n bad guesses (and $m - n$ good guesses). It will contribute $(m + 2n)\upsilon$ to the total running time.

Often the code is presented as a subroutine. In this case, the "call overhead"—the assignment of parameters, the `PUSHJ`, and the final `POP`—is not included in the computation of the total running time. In situations where the call overhead would be a significant percentage of the running time, the subroutine code can be expanded inline (see, for example, the `FindTag` subroutine in the solution to exercise 2.5–27 on page 143).

If, however, the subroutine under examination is itself the caller of a subroutine, the called subroutine, including its call overhead, will be included in the total count. A special case arises for recursive routines. There, the `PUSHJ` and `POP` instructions cannot be eliminated and must be counted. Further, it would be confusing not to include the final `POP` in the total count since this would violate Kirchhoff's law. The initial `PUSHJ` is, however, not shown—and not counted.

PROGRAMMING TECHNIQUES

1. INDEX VARIABLES

Many algorithms traverse information structures that are sequentially allocated in memory. Let us assume that a sequence of n data items a_0, a_1, . . . , a_{n-1} is stored sequentially. Further assume that each data item occupies 8 bytes, and the first element a_0 is stored at address `A`; the address of a_i is then `A` + $8i$. To load a_i with $0 \le i < n$ from memory into register `ai`, we need a suitable base address and so we assume that we have `A` = `LOC(`a_0`)` in register `a`. Then we can write '`8ADDU t,i,a; LDO ai,t,0`' or alternatively '`SL t,i,3; LDO ai,a,t`'. If this operation is necessary for all i, it is more efficient to maintain a register `i` containing $8i$ as follows:

```
    SET   i,0          i ← 0.
    LDO   ai,a,i       Load a_i.
    ADD   i,i,8        Advance to next element: i ← i + 1.  ■
    . . .
```

Note how `i` advances by 8 when i advances by 1.

The branch instructions of `MMIX`, like most computer architectures, directly support a test against zero; therefore a loop becomes more efficient if the index variable runs toward 0 instead of toward n. The loop may then take the form:

```
      SL    i,n,3     i ← n.
0H    SUB   i,i,8     Advance to next element: i ← i − 1.
      LDO   ai,a,i    Load a_i.
      . . .
      PBP   i,0B      Continue while i > 0.  ■
```

In the above form, the items are traversed in decreasing order. If the algorithm requires traversal in ascending order, it is more efficient to keep `A` + $8n$, the address of a_n, as new base address in a register `an`, and to run the index register `i` from $-8n$ toward -8 as in the following code:

```
      8ADDU  an,n,a    an ← A + 8n.
      SUBU   i,a,an    i ← 0 (or i ← −8n).
0H    LDO    ai,an,i   ai ← a_i.
      . . .
      ADD    i,i,8     Advance to next element: i ← i + 1.
```

```
     PBN     i,0B        Continue while i < n. ■
```

If `a` is used only to compute `A`+$8n$, it is possible to write '`8ADDU a,n,a`' and reuse register `a` to hold `A` + $8n$. Loading a_i then resumes the nice form '`LDO ai,a,i`', without any need for `an`. For an example, see Program 4.3.1S on page 63.

When computer scientists enumerate n elements, they say "a_0, a_1, a_2, . . .", starting with index zero. When mathematicians (and most other people) enumerate n elements, they say "a_1, a_2, a_3, . . ." and start with index 1. Nevertheless when such a sequence of elements is passed as a parameter to a subroutine, it is customary to pass the address of its first element `LOC(`a_1`)`. If this address is in register `a`, the address of a_i is now `a` + $8(i - 1)$. To load a_i efficiently into register `ai`, we have two choices: Either we adjust register `a`, saying '`SUBU a,a,8`' for `a` ← `LOC(`a_0`)`, or we maintain in register `i` the value of $8(i - 1)$, saying for example '`SET i,0`' for i ← 1. In both cases, we can write '`LDO ai,a,i`' to load `ai` ← a_i.

Many variations of these techniques are possible; a nice and important example is Program 5.2.1S on page 76.

2. FIELDS

Let us assume that the data elements a_i, just considered, are further structured by having three fields, two `WYDE`s and one `TETRA`, like this:

LEFT	RIGHT	KEY

It is then convenient to define offsets for the fields reusing the field names as follows:

```
 LEFT    IS      0          Offset for field LEFT
 RIGHT   IS      2          Offset for field RIGHT
 KEY     IS      4          Offset for field KEY
```

There is very little information in these lines, so these definitions are usually suppressed in a program's display.

Computing the address of, say, the `KEY` field of a_i requires two additions, `A` + $8i$ + `KEY`, of which only one must be done *inside* a loop over i. The quantity `A` + `KEY` can be precomputed and kept in a register named `key`. This simplifies loading of `KEY(`a_i`)` as follows:

```
   ADDU    key,a,KEY     key ← A + KEY.
```

```
    ...                     Loop on i with i = 8i.
    LDT     k,key,i         k ← KEY(a_i).  ▮
```

3. RELATIVE ADDRESSES

In a more general setting, this technique can be applied to relative addresses. Assume that one of the data items a_i is given by its relative address P = `LOC(`a_i`)` − `BASE` relative to some base address `BASE`.

Then again `KEY(`a_i`)` can be loaded by a single instruction '`LDT k,key,p`', if P is in register `p`, and `BASE` + `KEY` is in register `key`.

While an absolute address always requires eight bytes in `MMIX`'s memory, relative addresses can be stored using only four bytes, two bytes, or one byte, which allows tighter packing of information structures and reduces the memory footprint of applications that handle large numbers of links. Using this technique, the use of relative addresses can be as efficient as the use of absolute addresses.

4. USING THE LOW ORDER BITS OF POINTERS ("BIT STUFFING")

Modern computers impose alignment restrictions on the possible addresses of primitive data types. In the case of `MMIX`, an `OCTA` may start only at an address that is a multiple of 8, a `TETRA` requires a multiple of 4, and a `WYDE` needs an even address. As a result, data structures are typically octabyte-aligned, because they contain one or more `OCTA`-fields—for example, to hold an absolute address in a link field. Those link fields, in turn, are multiples of eight as well. Put differently, their three low-order bits are all zero. Such precious bits can be put to use as tag bits, marking the pointer to indicate that either the pointer itself or the data item it points to has some special property. `MMIX` further simplifies the use of these bits as tags by ignoring the low-order bits of an address in load and store instructions. That convention is not the case for all CPU architectures. Still, these bits are usable as tags; they just need to be masked to zero on such computers before using link fields as addresses.

Three different uses need to be distinguished. First, a tag bit in a link may contain some additional information about the data item it links to. Second, it may tell about the data item that contains the link. Third, it may disclose information about the link itself.

An example of the first type of use is the implementation of two-dimensional sparse arrays in Section 2.2.6. There, the nonzero elements of each row (or column) form a circular linked list anchored in a special list head node. It would have been possible to mark each head node using one of the bits in one of its link fields, but it is more convenient to put this information into the links pointing to a head node. Once the link to the next node in the row is known, a single instruction is sufficient to test for a head node, as for example in the implementation of Program 2.2.6S on page 132:

```
S3   LDOU  q0,q0,UP    S3. Find new row.  Q0 ← UP(Q0).
     BOD   q0,9F       Exit if Q0 is odd. ▮
```

If a head node would be marked by using a tag bit in its own UP link, the code would require an extra load instruction:

```
S3   LDOU  q0,q0,UP    S3. Find new row.  Q0 ← UP(Q0).
     LDOU  t,q0,UP     t ← UP(Q0).
     BOD   t,9F        Exit if TAG(Q0) = 1. ▮
```

The great disadvantage of this method, so it seems, is the need to maintain all the tag bits in all of the links that point to a head node during the running time of the program. A closer look at the operations a program like Algorithm 2.2.6S performs will reveal, however, that it inserts and deletes matrix elements but never deletes or creates head nodes. Inserting or deleting matrix elements will just copy existing link values; hence no special coding is required to maintain the tag bits in the links to head nodes.

The second, more common, type of use of a tag field is illustrated by the solution to exercise 2.3.5–4 on page 139. The least significant bit of the ALINK field is used to mark accessible nodes, and the least significant bit of the BLINK field is used to distinguish between atomic and non-atomic nodes. The following snippet taken from this code is typical for testing and setting of these tag bits:

```
E2   LDOU  x,p,ALINK     E2. Mark P.
     OR    x,x,1
     STOU  x,p,ALINK     MARK(P) ← 1.
E3   LDOU  x,p,BLINK     E3. Atom?
     PBEV  x,E4          Jump if ATOM(P) = 0. ▮
```

An interesting variation of this use of a tag bit can be seen in exercise 2.2.3–26 on page 23. There, the data structure asks for a variable-length list of links allocated sequentially in memory. Instead of encoding the length of the list somewhere as part of the data structure, the last link of the structure is marked by setting a tag bit. This arrangement leads to very simple code for the traversal of the list.

As a final example, consider the use of tag bits in the implementation of threaded binary trees in Section 2.3.1. There, the `RIGHT` and `LEFT` fields of a node might contain "down" links to a left or right subtree, or they might contain "thread" or "up" links to a parent node (see, for example, 2.3.1–(10), page *324*). Within a tree, there are typically both "up" and "down" links for the same node. Hence, the tag is clearly a property of the link, not the node. Searching down the left branch of a threaded binary tree, as required by step S2 of Algorithm 2.3.1S, which reads "If `LTAG(Q)` = 0, set `Q` ← `LLINK(Q)` and repeat this step," may take the following simple form:

```
0H   SET    q,p          Set Q ← LLINK(Q) and repeat step S2.
S2   LDOU   p,q,LLINK    S2. Search to left. p ← LLINK(Q).
     PBEV   p,0B         Jump if LTAG(Q) = 0. ▮
```

5. LOOP UNROLLING

The loop shown at the end of the last section has a `SET` operation that has no computational value; it just reorganizes the data when the code advances from one iteration to the next. A small loop may benefit significantly from eliminating such code by unrolling it or, in the simplest case, doubling it. Doubling the loop adds a second copy of the loop where the registers `p` and `q` exchange roles. This leads to

```
S2   LDOU  p,q,LLINK     S2. Search to left. P ← LLINK(Q).
     BOD   p,1F          If LTAG(Q) ≠ 0, exit the loop.
     LDOU  q,p,LLINK     S2. Search to left. Q ← LLINK(P).
     PBEV  q,S2          If LTAG(P) = 0, repeat step S2.
     SET   q,p           At this point p and q have exchanged roles.
1H   IS    @             ▮
```

The new loop requires 2υ per iteration instead of 3υ. For another example, see the solution to exercise 5.2.1–33 on page 167. Further, Program 6.1Q′ on page 98 illustrates how loop unrolling can benefit loops maintaining a counter variable, and the solution to exercise 6.2.1–10 on page 184 shows how to completely unroll a loop with a small, fixed number of iterations.

6. SUBROUTINES

The code of a subroutine usually starts with the definition of its stack frame, the storage area containing parameters and local variables. Using the `MMIX` register stack, it is sufficient for most subroutines to list and name the appropriate local registers. Once the stack frame is defined, the instructions that make up the body of the subroutine follow. The first instruction is labeled with the name of the subroutine—typically preceded by a colon to make it global; the last instruction is a `POP`. For a simple example see the solution to exercise 2.2.3–2 on page 124 or the solution to exercise 5–7 on page 162.

Subroutine Invocation. Calling a subroutine requires three steps: passing of parameters, transfer of control, and handling of return values. In the simplest case, with no parameters and no return values, the transfer of control is accomplished with a single '`PUSHJ $X,YZ`' instruction and a matching `POP` instruction. The problem remains choosing a register `$X` such that the subroutine call will preserve the values of registers belonging to the caller's stack frame. For this purpose, the subroutines in this book will define a local register, named `t`, such that all other named local registers have register numbers smaller than `t`. Aside from its role in calling subroutines, `t` is used as temporary variable. The typical form of a subroutine call is then '`PUSHJ t,YZ`'.

If the subroutine has $n > 0$ parameters, the registers for the parameter values can be referenced as `t+1`, `t+2`, . . . , `t+`n. A simple example is Program 2.3.1T, where the two functions `Inorder` and `Visit` are called like this:

```
T3   LDOU    t+1,p,LLINK      T3. Stack ⇐ P.
     SET     t+2,visit
     PUSHJ   t,:Inorder       Call Inorder(LLINK(P),Visit).
T5   SET     t+1,p            T5. Visit P.
     PUSHGO  t,visit,0        Call Visit(P).
```

After the subroutine has transferred control back to the caller, it may use the return values. If the subroutine has no return values, register `t` (and all registers with higher register numbers) will be marginal and a reference to it will yield zero; otherwise, `t` will hold the principal return value and further return values will be in registers `t+1`, `t+2`, ... The function `FindTag` in the solution to exercise 2.5–27 on page 143 is an example of a function with three return values.

Nested Calls. If the return value of one function serves as a parameter for the next function, the schema just described needs some modification. It is better to place the return value of the first function not in register `t` but directly in the parameter register for the second function; therefore we have to adjust the first function call. For example, the `Mul` function in Section 2.3.2, page 42, needs to compute `Q1` ← `Mult(Q1,Copy(P2))`, and that is done like this:

```
SET     t+1,q1       t+1 ← Q1.
SET     t+3,p2
PUSHJ   t+2,:Copy    t+2 ← Copy(P2).
```

```
PUSHJ   t,:Mult
SET     q1,t        Q1 ← Mult(Q1,Copy(P2)).
```

The `Div` function of exercise 2.3.2–15, which computes the slightly more complex formula

Q ← Tree2(Mult(Copy(P1),Q),Tree2(Copy(P2),Allocate(),"↑"),"/"),

contains more examples of nested function calls (see also the `Pwr` function of exercise 2.3.2–16).

Nested Subroutines. If one subroutine calls another subroutine, we have a situation known as nested subroutines. The most common error when programming `MMIX` is failing to save and restore the rJ register. At the start of a subroutine, the special register rJ contains the return address for the `POP` instruction. It will be rewritten by the next `PUSHJ` instruction and therefore must be saved if the next `PUSHJ` occurs before the `POP`.

There are two preferred places to save and restore rJ: Either start the subroutine with a `GET` instruction, saving rJ in a local register, and end the subroutine with a `PUT` instruction, restoring rJ, immediately before the terminating `POP` instruction; or, if the subroutine contains only a single `PUSHJ` instruction, save rJ immediately before the `PUSHJ` and restore it immediately after the `PUSHJ`. An example of the first method is the `Mult` function in Section 2.3.2; the second method is illustrated by the `Tree2` function in the same section. If subroutines use the `PREFIX` instruction to create local namespaces, the local copy of '`:rJ`' can simply be called '`rJ`'; that is the naming convention used in this book.

Tail Call Optimization. The `Mult` function of Section 2.3.2 is an interesting example for another reason: It uses an optimization called "tail call optimization." If a subroutine ends with a subroutine call in such a way that the return values of the inner subroutine are already the return values of the outer subroutine, the stack frame of the outer subroutine can be reused for the inner subroutine because it is no longer needed after the call to the inner routine. Technically, this is achieved by moving the parameters into the right place inside the existing stack frame and then using a jump or branch instruction to transfer control to the inner subroutine. The `POP` instruction of the inner subroutine will then return directly to the caller of the outer subroutine. So, when the function `Mult(u,v)` wants to return `Tree2(u,v,"×")`, `u` and `v` are already in place and '`GETA v+1,:Mul`' initializes the third parameter; then '`BNZ t,:Tree2`' transfers control to the `Tree2` function, which will return its result directly to the caller of `Mult`.

A special case of this optimization is the "tail recursion optimization." Here, the last call of the subroutine is a recursive call to the subroutine itself. Applying the optimization will remove the overhead associated with recursion, turning a recursive call into a simple loop. For an example, see Program 5.2.2Q on page 82, which uses `PUSHJ` as well as `JMP` to call the recursive part `Q2`.

7. REPORTING ERRORS

There is no good program without good error handling. The standard situation is the discovery of an error while executing a subroutine. If the error is serious enough, it might be best to issue an error message and terminate the program immediately. In most cases, however, the error should be reported to the calling program for further processing.

The most common form of error reporting is the specification of special return values. Most UNIX system calls, for example, return negative values on error and nonnegative values on success. This schema has the advantage that the test for a negative value can be accomplished with a single instruction, not only by `MMIX` but by most CPUs. Another popular error return value, which can be tested equally well, is zero. For example, functions that return addresses often use zero as an error return, because addresses are usually considered unsigned and the valid addresses span the entire range of possible return values. In most circumstances, it is, furthermore, simple to arrange things in a way that excludes zero from the range of valid addresses.

`MMIX` offers two ways to return zero from a subroutine: The two instructions '`SET $0,0; POP 1,0`' will do the job, but just '`POP 0,0`' is sufficient as well. The second form will turn the register that is expected to contain the return value into a marginal register, and reading a marginal register yields zero (see the solution to exercise 2.2.3–4 on page 125 for an example).

The `POP` instruction of `MMIX` makes another form of error reporting very attractive: the use of separate subroutine exits for regular return and for error return (see exercise 2.2.3–3 and its solution on page 125 for an example). The subroutine will end with '`POP 0,0`' in case of error and with '`POP 1,1`' in case of success, returning control to the instruction immediately following the `PUSHJ` in case of error and to the second instruction after the `PUSHJ` otherwise. The calling sequence must then insert a jump to the error handler just after the `PUSHJ` while the normal control flow continues with the instruction after the jump instruction. The advantages of this method are twofold. First, the execution of the normal control path is faster because it no longer contains a branch instruction to test the return value. Second, this programming style forces the calling program to provide explicit error handling; simply skipping the test for an error return will no longer work.

CONTENTS

CHAPTER ONE

BASIC CONCEPTS

1.3.3. Applications to Permutations

In this section, we shall give several more examples of `MMIX` programs, and at the same time introduce some important properties of permutations. These investigations will also bring out some interesting aspects of computer programming in general.

[167]

An `MMIX` program. To implement this algorithm for `MMIX`, the "tagging" can be done by using the sign bit of a `BYTE`. Suppose our input is an ASCII text file, with characters in the range 0 to #`7F`, where each character is either (a) '`(`', representing the left parenthesis beginning a cycle; (b) `')'`, representing the right parenthesis ending a cycle; (c) an ignorable formatting character in the range 0 to #`20`; or (d) anything else, representing an element to be permuted. For example, (6) might be represented in two lines as follows:

```
(ACFG) (BCD)
(AED) (FADE) (BGFAE)
```

The output of our program will be the product in essentially the same format.

Program A (*Multiply permutations in cycle form*). This program implements Algorithm A, and it also includes provision for input, output, and the removing of singleton cycles. But it doesn't catch errors in the input.

```
        LOC   Data_Segment
        GREG  @
MAXP    IS    #2000          Maximum number of permutations
InArg   OCTA  Buffer,MAXP    The arguments for Fread
Buffer  BYTE  0              Place for input and output
left    GREG  '('
right   GREG  ')'
        LOC   #100
base    IS    $0             Base address of permutations
k       IS    $1             Index into input
j       IS    $2             Index into output
x       IS    $4             Some permutation
current IS    $5
```

```
start    IS     $6
size     IS     $7
t        IS     $8
Main     LDA    $255,InArg                      Prepare for input.
         TRAP   0,Fread,StdIn                   Read input.
         SET    size,$255
         INCL   size,MAXP                       size ← $255 + MAXP.
         BNP    size,Fail                       Check if input was OK.
         LDA    base,Buffer
         ADDU   base,base,size                  base ← Buffer + size.
         NEG    k,size               1          A1. First pass.
2H       LDBU   current,k,base       A          Get next element of input.
         CMP    t,current,#20        A
         CSNP   current,t,0          A          Set format characters to zero.
         STB    current,k,base       A
         CMP    t,current,'('        A          Is it '('?
         PBNZ   t,1F                 A[B]
         ORL    current,#80          B          If so, tag it.
         STBU   current,k,base       B
0H       ADD    k,k,1                B
         LDBU   start,k,base         B          Put the next nonformat
         BZ     start,0B             B[0]         input symbol in START.
1H       CMP    t,current,')'        C          Is it ')'?
         PBNZ   t,0F                 C[D]
         ORL    start,#80            D
         STBU   start,k,base         D          Replace ')' by tagged START.
0H       ADD    k,k,1                C
         PBN    k,2B                 C[1]       Have all elements been processed?
         SET    j,0                  1
Open     NEG    k,size               E          A2. Open.
1H       LDB    x,k,base             F          Look for untagged element.
         PBP    x,Go                 F[G]
         ADD    k,k,1                G
         PBN    k,1B                 G[1]
Done     BNZ    j,0F                            Is answer the identity permutation?
         STB    left,base,0                     If so, change to '()'.
```

```
        STB    right,base,1
        SET    j,2
0H      SET    t,#0a                    Add a newline.
        STB    t,base,j
        ADD    j,j,1
        SET    t,0                      Terminate the string.
        STB    t,base,j
        SET    $255,base
        TRAP   0,Fputs,StdOut           Print the answer.
        SET    $255,0
Fail    TRAP   0,Halt,0                 Halt program.
Go      STB    left,base,j       H      Output '('.
        ADD    j,j,l             H
        STBU   x,base,j          H      Output X.
        ADD    j,j,l             H
        SET    start,x           H
Succ    ORL    x,#80             J
        STBU   x,k,base          J      TagX.
3H      ADD    k,k,l             J      A3. Set CURRENT.
        LDBU   current,k,base    J
        ANDNL  current,#80       J      Untag.
        PBNZ   current,1F        J[0]   Skip past blanks.
        JMP    3B                0
5H      STBU   current,base,j    Q      Output CURRENT.
        ADD    j,j,l             Q
        NEG    k,size            Q      Scan formula again.
4H      LDBU   x,k,base          K      A4. Scan formula.
        ANDNL  x,#80             K      Untag.
        CMP    t,x,current       K
        BZ     t,Succ            K[K+J-L]
IH      ADD    k,k,l             L      Move to right.
        PBN    k,4B              L[P]   End of formula?
        CMP    t,start,current   P      A5. CURRENT ≠ START.
        PBNZ   t,5B              P[R]
        STBU   right,base,j      R      A6.Close.
        SUB    j,j,2             R      Suppress singleton cycles.
```

```
86        LDB    t,base,j          R
87        CMP    t,t,'('           R
88        BZ     t,Open            R[R-S]
89        ADD    j,j,3             S
90        JMP    Open              S      ▮
```

This program of approximately 74 instructions is quite a bit longer than the programs of the previous section, and indeed it is longer than most of the programs we will meet in this book. Its length is not formidable, however, since it divides into several small parts that are fairly independent. Lines 17–23 read the input file; lines 24–41 accomplish step A1 of the algorithm, the preconditioning of the input; lines 42–47 and 61–90 do the main business of Algorithm A; and lines 48–60 output the answer.

. . .

Timing. The parts of Program A that are not concerned with input-output have been decorated with frequency counts as we did for Program 1.3.2´M; thus, line 34 is supposedly executed B times. For convenience it has been assumed that no formatting characters appear in the input; under this assumption, line 72 is never executed and the branch in line 35 is never taken.

By simple addition the total time to execute the program is

$$(6+6A+7B+4C+4D+E+2F+4G+5H+8J+3Q+6K+4P+9R)v, \quad (7)$$

plus the time for input and output. In order to understand the meaning of formula (7), we need to examine the thirteen unknowns A, B, C, D, E, F, G, H, J, K, P, Q, R (the running time does not depend on S or L) and we must relate them to pertinent characteristics of the input. We will now illustrate the general principles of attack for problems of this kind.

First we apply "Kirchhoff's first law" of electrical circuit theory: The number of times an instruction is executed must equal the number of times we transfer to that instruction. This seemingly obvious rule often relates several quantities in a nonobvious way. Analyzing the flow of Program A, we get the following equations.

From lines	*We deduce*
24, 25, 41	$A = 1 + (C - 1)$
30, 35, 36	$C = B + (A - B)$
42, 43, 88, 90	$E = 1 + R$
43, 44, 47	$F = E + (G - 1)$

45, 60, 61	$H = F - G$
65, 66, 79	$J = H + (K - L + J)$
75, 76, 81	$K = Q + (L - P)$
72, 73, 83	$R = P - Q$

[171]

The next step is to try to match up variables with important characteristics of the data. We find from lines 24, 33, and 40 that

$$B + C = \text{size of the input file} = X. \tag{9}$$

From line 31,

$$B = \text{number of '(' in input} = \text{number of cycles in input.} \tag{10}$$

Similarly, from line 38,

$$D = \text{number of ')' in input} = \text{number of cycles in input.} \tag{11}$$

Now (10) and (11) give us a fact that could not be deduced by Kirchhoff's law:

$$B = D. \tag{12}$$

From line 61,

$$H = \text{number of cycles in output (including singletons).} \tag{13}$$

Line 84 says R is equal to this same quantity; the fact that $H = R$ *was* in this case deducible from Kirchhoff's law, since it already appears in (8).

Using the fact that each nonformatting character is ultimately tagged, and lines 32, 39, and 67, we find that

$$J = Y - 2B, \tag{14}$$

where Y is the number of nonformatting characters appearing in the input. From the fact that every *distinct* element appearing in the input permutation is written into the output just once, either at line 63 or line 73, we have

$$P = H + Q = \text{number of distinct elements in input.} \tag{15}$$

(See Eqs. (8).) A moment's reflection makes this clear from line 82 as well.

Clearly the quantities B, C, H, J, and P that we have now interpreted are essentially independent parameters that may be expected to enter into the timing of Program A.

The results we have obtained so far leave us with only the unknowns G and L to be analyzed. For these we must use a little more ingenuity. The scans of the input that start at lines 43 and 75 always terminate either at line 48 (the last time) or at line 82. During each one of these $P + 1$ loops, the instruction '`ADD k,k,1`' is performed $B + C$ times; this takes place only at lines 46, 68, and 80, so we get the nontrivial relation

$$G + J + L = (B + C)(P + 1) \tag{17}$$

concerning our unknowns G and L. Fortunately, the running time (7) is a function of $G + L$ (it involves $\cdots + 2F + 4G + 6K + \cdots = \cdots + 6G + \cdots + 6L + \cdots$), so we need not try to analyze the individual quantities G and L any further.

Summing up all these results, we find that the total time exclusive of input-output comes to

$$(6NX + 16X - 3M + 2Y + 2U + 13N + 7)v; \tag{18}$$

in this formula, new names for the data characteristics have been used as follows:

$$\begin{aligned} X &= \text{number of characters in input,} \\ Y &= \text{number of nonformatting characters in input,} \\ M &= \text{number of cycles in input,} \\ N &= \text{number of distinct element names in input,} \\ U &= \text{number of cycles in output (including singletons).} \end{aligned} \tag{19}$$

In this way we have found that analysis of a program like Program A is in many respects like solving an amusing puzzle.

We will show below that, if the output permutation is assumed to be random, the quantity U will be H_N on the average.

[174]

Let us now write an `MMIX` program based on the new algorithm. . . . A simple way to solve this problem is to make table T large enough so that we can use the elements x_i directly as indices. In our case the range of possible elements is `#21` to `#7F`, which makes a moderate-sized table.

Program B (*Same effect as Program A*).

```
01          LOC    Data_Segment
02  T       GREG   @-#21               T ← LOC(T[0]).
03          BYTE   0                   Now make a table
04          LOC    @+#5F                 for all valid names.
05  Z       IS     $9
                    ⋮                  Same as lines 02–22 of Program A.
27          SET    k,#21         1     B1. Initialize. Set k to first valid name.
28  0H      STB    k,T,k         A     T[k] ← k.
29          ADD    k,k,1         A     k ← k + 1.
30          CMP    t,k,#80       A     Loop until k = #7F.
31          PBN    t,0B
```

```
                                         A[1]
         SET    k,size                   1
         JMP    9F                       1
2H       LDB    X,base,k                 B       B2. Next element.
         CMP    t,X,#20                  B       Skip formatting characters.
         BNP    t,9F                     B[0]
         CMP    t,X,')'                  B
         BZ     t,0F                     B[B-C]
         CMP    t,X,'('                  C
         CSZ    X,t,j                    C       B4. Change T[i].
         CSZ    j,Z,X                    C       B3. Change T[j].
         LDB    t,T,X                    C
         STB    Z,T,X                    C
0H       SET    Z,t                      D       If t = 0, set Z ← 0.
9H       SUB    k,k,1                    E
         PBNN   k,2B                     E[1]    Input exhausted.
Output   ADDU   base,base,size           1       base ← Buffer + size.
         SET    j,0                      1
         SET    k,#21                    1       Traverse table T.
0H       LDB    X,T,k                    F
         CMP    t,X,k                    F
         PBZ    t,2F                     F[G]    Skip singleton.
         PBN    X,2F                     G[H]    Skip tagged element.
         STB    left,base,j              H       Output '('.
         ADD    j,j,1                    H
         SET    Z,k                      H       Loop invariant: X = T[Z].
1H       STB    Z,base,j                 J       Output z.
         ADD    j,j,1                    J
         OR     t,X,#80                  J
         STBU   t,T,Z                    J       Tag T[Z].
         SET    Z,X                      J       Advance Z.
         LDB    X,T,Z                    J       Get successor element
         PBNN   X,1B                     J[H]      and continue, if untagged.
         STB    right,base,j             H       Otherwise, output ')'.
         ADD    j,j,1                    H
2H       ADD    k,k,1                    K       Advance in table T.
```

```
67          CMP    t,k,#80         K
68          PBN    t,0B            K[1]
            ⋮                                  Same as lines 48–60 of Program A. ▮
```

Notice how lines 38–44 accomplish most of Algorithm B with just a few instructions.

. . .

Making the table T large enough to enable the use of the elements as indices is not feasible if arbitrary strings are allowed as element names. Algorithms for searching and building dictionaries of names, called *symbol table algorithms*, are of great importance in computer applications. Chapter 6 contains a thorough discussion of efficient symbol table algorithms.

[177]

Program I (*Inverse in place*). We assume that the permutation is stored as an array of BYTEs and that x ≡ LOC(X[1]).

```
:Invert  SUBU  x,x,1   1        x ← LOC(X[0]).
         SET   m,n     1        I1. Initialize.
         NEG   j,1     1        j ← −1.
2H       LDB   i,x,m   N        I2. Next element. i ← X[m].
         BN    i,5F    N[N−C]   To I5 if i < 0.
3H       STB   j,x,m   N        I3. Invert one. X[m] ← j.
         NEG   j,m     N        j ← −m.
         SET   m,i     N        m ← i.
         LDB   i,x,m   N        i ← X[m].
4H       PBP   i,3B    N[C]     I4. End of cycle?. To I3 if i > 0.
         SET   i,j     C        Otherwise set i ← j.
5H       NEG   i,i     N        I5. Store final value. i ← −i.
         STB   i,x,m   N        X[m] ← i.
6H       SUB   m,m,1   N        I6. Loop on m.
         BP    m,2B    N[1]     To I2 if m > 0. ▮
```

The timing for this program is easily worked out in the manner shown earlier; every element $X[m]$ is set first to a negative value in step I3 and later to a positive value in step I5. The total time comes to $(13N + C + 5)\upsilon$, where N is the size of the array and C is the total number of cycles. The behavior of C in a random permutation is analyzed below.

[178]

Program J (*Analogous to Program I*).

```
:Invert  SUBU  x,x,1    1       x ← LOC(X[0]).
         SET   k,n      1       J1. Negate all.
0H       LDB   i,x,k    N       i ← X[k].
         NEG   i,i      N       i ← −i.
         STB   i,x,k    N       X[k] ← i.
         SUB   k,k,1    N       Continue
         PBP   k,0B     N[1]      while k > 0.
         SET   m,n      1       m ← n.
2H       SET   i,m      N       J2. Initialize. i ← m.
0H       SET   j,i      A       j ← i.
         LDB   i,x,j    A       J3. Find negative entry. i ← X[j].
         PBP   i,0B     A[N]    i ¿ 0?
         NEG   i,i      N       J4. Invert. i ← −i.
         LDB   k,x,i    N       k ← X[i].
         STB   k,x,j    N       X[j] ← k.
         STB   m,x,i    N       X[i] ← m.
         SUB   m,m,1    N       J5. Loop on m.
         BP    m,2B     N[1]    To J2 if m ¿ 0.  ▮
```

1.4.4. Input and Output

[215]

A brief digression about terminology is perhaps appropriate here. . . . This completes today's English lessons.

The old `MIX` machine that is featured in Volumes 1, 2, and 3 of *The Art of Computer Programming* has old-fashioned conventions for input and output, now called "non-blocking I/O." That is, a `MIX` programmer said, "Please start inputting (or outputting) now, but let me continue executing more code." The machine would block further computation only if it hadn't yet finished the previous I/O instruction on the same device. The programmer could also test, if desired, whether or not that previous command was complete, again without blocking.

By contrast, input and output are specified in `MMIX` programs by the primitive operations `Fopen`, `Fclose`, `Fread`, . . ., which are supplied by an underlying operating system. Modern operating systems and programming languages tend to discourage the use of more primitive, low-level operations, because such instructions are deemed to be too dangerous. Thus it is impossible to give `MMIX` programs that correspond closely with the `MIX` programs in the original text.

At the same time, the rise of modern multicore processors has made it necessary for every serious programmer to understand *threads*. A thread is a kind of coroutine that enjoys special support from the operating systems. The system might assign separate physical processors to individual threads, executing them in parallel; or it might allow a pool of threads to share a pool of processors, periodically switching processors from one thread to another, so as to create the illusion of truly parallel execution. Like coroutines, multiple threads share a joint memory space; in contrast to coroutines, each thread has its own register file and stack space. In such an environment, the techniques used with non-blocking I/O reappear when one thread is responsible for asking the operating system to do input or output while another thread is concurrently doing the computation. The computing thread can send data to the I/O thread for output, or the I/O thread can send input data to the computing thread for processing.

There's a nice symmetry between these two threads, because both are doing "computation" in some sense. The I/O thread is blocked while waiting for the operating system to finish reading or writing; the other thread is "blocked" while waiting for its instructions to be performed. In the following, we will call one of the threads the *producer* and the other one the *consumer*—but it really won't matter which one is doing the I/O because of the symmetry.

The main interesting point is the sharing of a common resource. Inside an operating system kernel, the available physical devices (disks, screens, network connections, etc.) are shared resources; within user-space, the shared resource is usually just a set of locations in main memory. In general, many threads can share a complex data structure, but two threads might actually need to share only one octabyte.

Let us therefore consider the problem of an I/O thread and a computing thread, which exchange data using a shared area of memory called a "buffer." The simplest way to do this is probably to make the producer and the consumer alternate in their use of the buffer: While the producer fills the buffer, the consumer will wait; and while the consumer uses the buffer data, the producer will wait. To synchronize both threads, we use a shared octabyte `S`, called a semaphore. The octabyte will have the value 0 if the producer is allowed to access the buffer (and the semaphore); it will have the value 1 if the consumer has access to both. The code granting mutual exclusive access to the buffer may look like this:

```
Consumer:                                   Producer:
0H    LDO    t,S    Acquire.                0H    LDO    t,S    Acquire.
      BZ     t,0B   Wait.                         BNZ    t,0B   Wait.
      SYNC   2      Synchronize.                  ⋮             Use buffer.        (1)
      ⋮             Use buffer.                   SYNC   1      Synchronize.
      STCO   0,S    Release.                      STCO   1,S    Release.
```

Note the 'SYNC 2' and 'SYNC 1' instructions in the consumer and the producer, respectively. Here we assume that the producer is writing to the buffer and the consumer is reading from it. Without the 'SYNC 2', the consumer might guess that the 'BZ' will not be taken and it might load data from the buffer even before the 'LDO t,S' instruction loads S. By the time S is known to be zero, the data loaded from the buffer might already be outdated.

The reason for the 'SYNC 1' instruction in the producer is similar. Modern processors will not usually guarantee "sequential consistency"; in other words, we cannot rely on the machine to make the effect of store instructions visible to another thread in exactly the same order in which the instructions are issued. The 'SYNC 1' instruction is there to ensure that the consumer will see all changes made to the buffer once it has seen the change of S.

Programming concurrent threads on the instruction level is a demanding task. Here we can only touch on some of the problems and assure the reader that this book is mostly about sequential programs.

The method of (1) is generally wasteful of computer time, however, because a very large amount of potentially useful calculating time is spent in the waiting loop. The program's running speed can be as much as doubled if this additional time is used for calculation (see exercises 4 and 5, page *225*).

One way to avoid such a "busy wait" is to use two buffers to exchange data between producer and consumer: The producer can fill one buffer while the consumer is using the data in the other. The code for the consumer could change to the following:

```
Consumer:
0H    LDO    t,S    Acquire.
      BZ     t,0B   Wait.
      SYNC   2      Synchronize.                          (3)
      ⋮             Copy buffer one to buffer two.
      STCO   0,S    Release.
      ⋮             Use buffer two.
```

This has the same overall effect as (1), but it keeps the producer busy while the consumer works on the data in buffer two.

[217]

The sequence (3) is not always superior to (1), although the exceptions are rare. Let us compare the execution times: Suppose P is the time required by the producer to input one page containing 256 octabytes, and suppose C is the computation time that intervenes between two input requests by the consumer. Method (1) requires a time of essentially $P + C$ per input page, while method (3) takes essentially $\max(P,\ C) + 256\upsilon$. (The quantity 256υ is an estimate for the time needed for the copy operation assuming that a pipelined processor can complete one `LDO` and one `STO` instruction simultaneously per cycle.) One way to look at this running time is to consider "critical path time"—in this case, the amount of time the I/O unit is idle between uses. Method (1) keeps the unit idle for C units of time, while method (3) keeps it idle for 256υ (assuming that $C < P$).

The relatively slow copying of buffers in (3) is undesirable, particularly because it takes up critical path time. An almost obvious improvement of the method allows us to avoid the copying: Producer and consumer can be revised so that they refer alternately to two buffers. While one buffer is filled by the producer, the consumer can perform computations using the other; then the producer can fill the second buffer while the consumer continues with the information in the first. This is the important technique known as *buffer swapping.* The location of the current buffer of interest will be kept in memory together with the semaphore protecting it and a link to the next semaphore.

As an example of buffer swapping, suppose we have two buffers at locations `Buffer1` and `Buffer2`, each `SIZE` bytes long. Then we define two semaphores, `S1` and `S2`, and combine each one with a link to the respective buffer and a link to the other semaphore. We assume that the consumer has set up three global registers: `buffer`, pointing to one of the buffers; `i`, an index into this buffer; and `s`, pointing to the corresponding semaphore. Then the following subroutine `GetByte` gets the next byte from the buffer, switching to a new buffer if the end of the current buffer (marked by a zero byte) is reached.

```
S1      OCTA  1,Buffer1,S2      Consumers buffer linked to S2.
S2      OCTA  0,Buffer2,S1      Producers buffer linked to S1.
              ⋮
1H      STCO  0,s,0             Release.
        LDO   s,s,16            Switch to next buffer.
0H      LDO   t,s,0             Acquire.
        BZ                      Wait.                          (4)
```

```
          BZ    t,0B          Wait.                         (4)
          SYNC  2             Synchronize.
          LDO   buffer,s,8    Update buffer.
          NEG   i,1           Initialize i ← −1.
:GetByte  ADD   i,i,1         Advance to next byte.
          LDBU  $0,buffer,i   Load one byte.
          BZ    $0,1B         Jump if end of buffer.
          POP   1,0           Otherwise return a byte. ▮
```

The subroutine used by the producer to fill the buffer is quite symmetric (see exercise 2).

It is easy to see that the same subroutine would also work for multiple buffers provided that they are set up with multiple semaphores linked to form a ring.

Some more programming is required to make the subroutine work for multiple concurrent consumers. If used as written above, the second consumer could acquire the same buffer that the first consumer is working on and process it a second time. The obvious way to prevent this from happening is to use a three-valued semaphore: The value 0 implies that the producer owns it, 1 marks it for consumer one, and 2 marks it for consumer two. The producer could then schedule the buffers alternating between both consumers.

In general, the flow of buffers through a system with many buffers and many threads can be organized in the manner outlined above as long as every thread releasing a buffer knows in advance which thread should acquire this buffer for further processing. But this assumption is unrealistic in many situations. Just think of a web server with one producer that turns incoming network traffic into page requests and a variable number of consumers (depending on the current workload) that take one page request at a time and assemble a reply. Since the producer will not know in advance which consumer will finish next, it cannot possibly assign the right consumer to a new page request.

We solve this problem in three steps. First, we separate acquisition and release of buffers into separate subroutines; second, we color each buffer with `Red` ≡ 0 if it is empty, with `Green` ≡ 1 if it is full, and with `Yellow` ≡ 2 if it is assigned to a consumer; and third, we maintain two pointers, `NEXTG` and `NEXTR`, pointing respectively to the red and green buffer that is to be processed next. These pointers will split the ring of buffers in two sections: `NEXTG` points to the sequence of all the green buffers, after which `NEXTR` points to the sequence of all the red and yellow buffers. Of course, any of these sequences might be empty. As long as these pointers are used by a single thread, we can keep them in global registers; if multiple threads need to share them, they need to be stored in main memory and concurrent access to them must be protected by two semaphores `SG` and `SR`, respectively.

The producer will fill the first red buffer, then turn it green, and advance to the next red buffer, waiting, if necessary, for a yellow (or even green) buffer to turn red. Multiple consumers will be working, each one on its own yellow buffer. When a consumer has finished work with its buffer, it will release the buffer and color it red. Then the consumer will advance to the first green buffer, acquire the corresponding semaphore, then wait if necessary for the buffer to turn green, before finally coloring it yellow and releasing the semaphore.

Program A (*Acquire for multiple consumers*).

```
01 s          GREG  0             Pointer to current color, buffer, and link
02 t          IS    $0            Temporary variable
03 :Acquire   PUT   :rP,0         Expect GS = 0.
04            SET   t,1           Intend to set GS ← 1.
05            CSWAP t,:GS         Acquire green semaphore.
06            BZ    t,:Acquire    Start over if swap failed.
07            SYNC  2             Synchronize.
08            LDOU  s,:NEXTG      Load address of next green buffer.
09 0H         LDO   t,s,0         Load buffer color.
10            CMP   t,t,:Green    Is it green?
11            BNZ   t,0B          Jump if it's not green.
12            STCO  :Yellow,s,0   Color buffer yellow.
13            LDOU  t,s,16        Load link.
14            STOU  t,:NEXTG      Advance NEXTG.
15            LDO   $0,s,8        Load buffer address.
16            SYNC  1             Synchronize.
17            STCO  0,:GS         Release green semaphore.
18            POP   1,0           Return buffer address. ▮
```

The most interesting part of this routine is the loop in lines 03–06 where the consumer waits until it can acquire the green semaphore. The loop culminates in the instruction '`CSWAP t,:GS`'. This instruction will—in one atomic operation—load the content of the octabyte at location `GS`, compare it with the content of the special prediction register rP, and, if both values are equal, will store the content of register `t` at location `GS` and set register `t` to 1. The important word here is "atomic." The same sequence of operations could be achieved with a sequence of ordinary load, compare, branch, and store instructions, but it would not be atomic. In the context of multiple threads that execute in parallel, it would easily be possible that one thread loads the value zero from location `GS`, and while it is busy with comparing and branching, a second thread also loads the value zero from location `GS`, long before the first thread can execute its store instruction. Then both threads would proceed and both would start working with the same buffer. The `CSWAP` instruction, in contrast, will do the load, compare, and store as one uninterruptable (that is, atomic) operation. Once a `CSWAP` instruction has started, it will prevent any other `CSWAP` instruction from loading or storing at the same memory location in parallel. Multiple `CSWAP` instructions will always execute one after the other.

Therefore, if multiple consumer threads enter the above subroutine concurrently, one lucky thread gets its `CSWAP` instruction executed first and successfully. The `CSWAP` instructions executing later will find that the value at location `GS` no longer matches the content of the prediction register and so will fail. In case of failure, the instruction '`CSWAP t,:GS`' will change the prediction register to reflect the new value at location `GS`, leave the memory at location `GS` unchanged, and set the register `t` to zero to indicate failure.

In this way, the `CSWAP` instruction protects the code sequence from line 08 up to line 17 where `GS` is reset to zero. If multiple consumers need a green buffer, `CSWAP` and the semaphore guarantee that at any time only one consumer can set `GS` to one and enter the protected code sequence; all the others will have to wait. Once inside the protected code sequence, the thread has earned the right to modify `NEXTG`, the buffer it points to, its color, and the semaphore `GS` (see also exercise 15). First, the loop in lines 09–11 ensures that the buffer at `NEXTG` is indeed green. Since `NEXTG` points to the next green buffer, we can reasonably expect that the loop is executed only once. Then the color of the buffer is changed to yellow and the `NEXTG` pointer is advanced. The final '`SYNC 1`' ensures that these changes become visible to other threads before they can see the change in `GS` from 1 back to 0.

Compared to this, releasing a buffer is extremely simple.

Program R (*Release for multiple consumers*).

```
:Release    STCO   :Red,s,0   Turn buffer red. ▮
```

EXERCISES

[225]

1. [*20*] *New*: In (1), the memory at location `S` is shared between two concurrent threads that both alter it. Why is no `CSWAP` instruction required?

2. [*20*] *New*: Write a program for a producer collaborating with a consumer that uses (4). The producer should use `Fgets` to fill each buffer with one line from `StdIn`.

3. [*25*] *New*: Write an improved version of (4). The current subroutine will delay the release of the buffer unnecessarily until the first byte of the next buffer is requested. The improved version should release the buffer as soon as the last byte of the current buffer is taken out.

6. [*20*] *New*: How should the global registers `s`, `i`, and `buffer` as well as the content of `Buffer1` and `Buffer2` be initialized so that the `GetByte` subroutine in (4) gets off to the right start?

7. [*17*] *New*: Which changes are required in Programs A and R in order to obtain `Acquire` and `Release` subroutines for use by a *single* producer?

12. [*12*] *New*: Modify Program A and R to work with *multiple* producers. *Hint:* Add the color `Purple`; a buffer should have the color `Purple` if it is currently owned by a producer.

13. [*20*] *New*: Discuss why a ring of buffers is not always the best data structure for sharing buffers between multiple consumers and multiple producers.

15. [*20*] *New*: Mr. B. C. Dull (an `MMIX` programmer) thought that `CSWAP` is an expensive instruction and he could improve Program A by first waiting until the buffer at `NEXTG` turns green and only then start an attempt to acquire the green semaphore. After all, the waiting loop does not modify any memory locations, therefore setting the semaphore should not be necessary for this part of the program. So he used the following code instead of Program A:

```
01  s          GREG   0             Pointer to current color, buffer, and link
02  t          IS     $0            Temporary variable
03  :Acquire   LDOU   s,:NEXTG      Load address of next green buffer.
04             LDO    t,s,0         Load buffer color.
05             CMP    t,t,:Green    Is it green?
06             BNZ    t,:Acquire    Jump if it's not green.
07             PUT    :rP,0         Expect GS = 0.
08             SET    t,1           Intend to set GS ← 1.
09             CSWAP  t,:GS         Acquire green semaphore.
10             BZ     t,:Acquire    Start over if swap failed.
11             STCO   :Yellow,s,0   Color buffer yellow.
12             ...                  (Lines 12–18 remain as before.)   ▮
```

```
12                            (Lines 12–18 remain as before.)
```

What serious mistake did he make, and what should he have done instead?

18. [*35*] *New*: Inside an operating system, I/O typically uses the interrupt facilities of the processor. Write a forced trap handler that implements '`TRAP 0,Fgets,StdIn`' and a matching dynamic trap handler, which takes care of the keyboard interrupt. Both handlers should communicate by using a shared buffer.

To keep things simple, assume that each keystroke causes an interrupt which will set the `KBDINT` bit of rQ to 1; and that after such an interrupt, the character code just typed can be read as a single byte value at physical address `KBDCHAR`. An invocation of '`TRAP 0,Fgets,StdIn`' should return immediately, if the necessary data is already available in the buffer; otherwise, it should wait until sufficient data has accumulated. Besides the buffer, both handlers may share additional data to do the "bookkeeping."

CHAPTER TWO

INFORMATION STRUCTURES

2.1. INTRODUCTION

[233]

We will illustrate methods of dealing with information structures in terms of the MMIX computer. A reader who does not care to look through detailed MMIX programs should at least study the ways in which structural information is represented in MMIX's memory.

. . .

As a more interesting example, suppose the elements of our table are intended to represent playing cards; we might have two-octabyte nodes broken into five fields, TAG, SUIT, RANK, TITLE, and NEXT:

NEXT			
TAG	SUIT	RANK	TITLE

(1)

(This format reflects the content of two octabytes; see Section 1.3.1´.)

[234]

TAG is stored as one BYTE; TAG = #80 means that the card is face down, TAG = #00 means that it is face up. A single bit would be enough to store this information; it is, however, convenient to use an entire byte, because this is the smallest unit of memory that can be loaded or stored individually. Using the most significant bit has the further advantage that it is the "sign" bit; it can be tested directly—for instance, with a BN (branch if negative) instruction. SUIT is another byte, with SUIT = 1, 2, 3, or 4 for clubs, diamonds, hearts, or spades, respectively. The next byte holds the RANK; RANK = 1, 2, . . . , 13 for ace, deuce, . . . , king. TITLE is a five-character alphabetic name of this card, for use in printouts. NEXT is a *link* to the card below this one in the pile. A typical pile might look like this:

Computer representation

#20...100:	Λ							
#20...108:	#80	1	10	␣	1	0	␣	C
#20...388:	#2000000000000100							
#20...390:	#00	4	3	␣	␣	3	␣	S
#20...240:	#2000000000000388							
#20...248:	#00	2	2	␣	␣	2	␣	D

(2)

It is easy to transform this notation into MMIXAL assembly language code. The values of link variables are put into registers; field-offsets, defined as appropriate constants, are used in load and store instructions. For example, Algorithm A above could be written thus:

```
        LOC   Data_Segment
        GREG  @
TOP     OCTA  1F              Link variable; points to top card on pile.
NEWCARD OCTA  2F              Link variable; points to a new card.
NEXT    IS    0               Definition of NEXT
TAG     IS    8                 and TAG offsets for the assembler
FACEUP  IS    0
top     IS    $0              Register for TOP
new     IS    $1              Register for NEWCARD
t       IS    $2              Temporary variable
        ...                                                        (5)
        LOC   #100
Main    ...
        LDOU  new,NEWCARD     A1. new ← NEWCARD.
        LDOU  top,TOP         top ← TOP.
        STOU  top,new,NEXT    NEXT(NEWCARD) ← TOP.
        STOU  new,TOP         A2. TOP ← NEWCARD.
        SET   t,FACEUP        A3.
        STBU  t,new,TAG       TAG(TOP) ← FACEUP.
        ...                   ▮
```

. . .

There is an important distinction between assembly language and the notation used in algorithms. Since assembly language is close to the machine's internal language, the symbols used in MMIXAL programs mostly stand for addresses and registers instead of values. Thus, in the left-hand column of (5), the symbol TOP actually is bound to the *address* where the pointer to the top card appears in memory; but in (6) and (7) and in the remarks at the right of (5), it denotes the *value* of TOP—namely, the address of the top card node. To complicate things even further, before MMIX can work with the address of the top card, it needs to load this address into a register. For this purpose, (5) introduces the symbol top and binds it to register $0. After MMIX loads the content of TOP, the octabyte in memory, into top, the register, both will contain the same value. Occasionally, however, a symbol in an MMIXAL program is indeed bound to a plain value; in (5), the name FACEUP was introduced just to illustrate this case.

EXERCISES

[237]

7. [*07*] In the text's example MMIX program (5), the link variable TOP is stored in an OCTA labeled TOP in MMIXAL assembly language. Given the field structure (1), which of the following sequences of code brings the quantity SUIT(TOP) into register t? Explain why the other sequences are incorrect.

a)
```
LDA t,TOP
LDB t,t,SUIT
```

b)
```
LDA t,TOP+SUIT
LDB t,t,0
```

c)
```
LDOU t,TOP
LDB t,t,SUIT
```

8. [*18*] Write an MMIX program corresponding to steps B1–B3.

9. [*23*] Write an MMIX subroutine that prints out the alphabetic names of the cards in the card pile, starting with card X, passed as a parameter, with one card per line, and with parentheses around cards that are face down.

2.2.2. Sequential Allocation

[246]

In the case of MMIX, given an index in register i, the coding to bring the ith one-octabyte node into register a is changed from

LDA base,L_0		LDOU base,BASE	
SL ii,i,3	to, for example,	SL ii,i,3	(8)
LDO a,base,ii		LDO a,base,ii	

where `ii` is an auxiliary register and `BASE` contains the address of $\mathtt{L}_0$. Such relative addressing may take longer than fixed-base addressing, because the `LDOU` executes an additional load from memory after an address calculation, which by itself is equivalent to the `LDA` instruction. If, however, the base address is kept in a global register instead of in a memory location, relative addressing can be as fast as fixed-base addressing.

EXERCISES

[251]

3. [*21*] *New*: Suppose that `MMIX` is extended as follows: The value of the `Z` field of the `LDOUI` instruction is to have the form $Z = 8Z_1 + 4Z_2 + Z_3$, where $0 \le Z_1 < 32$, $0 \le Z_2 < 2$, and $0 \le Z_3 < 4$. If $Z_2 = 0$, the meaning is that the instruction will load $\mathrm{u}(\$X) \leftarrow M_8[\$Y + Z_1 \times 8]$ if $Z_3 = 0$, and it will load $\mathrm{u}(\$X) \leftarrow M_8[\$Y + (Z_1 + \$Z_3) \times 8]$ if $0 < Z_3 < 4$. If, however, $Z_2 = 1$ instead of loading $\$X$, the instruction will first load a new value of Z according to the rules above and then repeat the load instruction using the new value for Z (with $0 \le Z_1 < 2^{61}$) and zero instead of $\$Y$. The execution time of the instruction will be $1\upsilon + 1\mu$ plus an extra $\upsilon + \mu$ for each time where $Z_2 = 1$.

The instruction `LDOU` will work the same, but will take the value of Z from register `$Z`; the instructions `LDO` and `LDOI` will work like their unsigned counterparts. As a nontrivial example, suppose that the octabyte at location `#1020` contains `#2002`, register `$0` holds the value `#1000`, and register `$2` holds the value 7.

Then the instruction `LDOUI $X,$0,#24` will first compute `$0` + `#20` = `#1020`, then load $Z \leftarrow M_8$[`#1020`] = `#2002`, and start over, now computing the address `#2000` + `$2` × 8 = `#2038`, and finally load $\mathrm{u}(\$X) \leftarrow M_8$[`#2038`].

Using this new addressing feature, show how to simplify the coding of (8). How much faster is your code than (8)?

4. [*20*] *New*: Given the extension of exercise 3, suppose there are several tables whose base addresses are stored as octabytes in locations `X`, `X` + 8, `X` + 16, ... How can the new addressing feature be used to bring the `i`th element of the `j`th table into register `a`?

5. [*20*] *New*: Discuss the merits of the extension proposed in exercise 3.

2.2.3. Linked Allocation

[256]

7) Simple operations, like proceeding sequentially through a list, are slightly faster for sequential lists on many computers. For `MMIX`, the comparison is between '`INCL i,`*c*' and '`LDOU p,p,LINK`', which both are done in one cycle but with the difference of an additional memory access. If the elements of a linked list belong to different cache lines, or even to different pages in bulk memory, the memory accesses might take significantly longer.

. . .

In the next few examples we will assume for convenience that a node has two octabytes—first one octabyte for the `LINK` and then one octabyte for the `INFO`:

LINK
INFO

. (3)

[258]

Before looking at the case of queues, let us see how the stack operations can be expressed conveniently in programs for `MMIX`. Assuming that `AVAIL` is kept in a global register `avail`, we can write a program for insertion, with parameter `y` (the `INFO`) as follows, using two auxiliary local registers `p` and `t`:

```
LINK   IS     0                 Offset of the LINK field
INFO   IS     8                 Offset of the INFO field
       SET    p,:avail          P ← AVAIL.
       BZ     p,:Overflow       Is AVAIL = Λ?
       LDOU   :avail,p,LINK     AVAIL ← LINK(P).                    (10)
       STO    y,p,INFO          INFO(P) ← Y.
       LDOU   t,:T
       STOU   t,p,LINK          LINK(P) ← T.
       STOU   p,:T              T ← P. ▮
```

This takes $7\upsilon + 5\mu$, compared to $3\upsilon + 1\mu$ for a comparable operation with a sequential table (although `Overflow` in the sequential case would in many cases take considerably longer). In this program, as in others to follow in this chapter, `Overflow` denotes an ending routine.

A program for deletion is equally simple:

```
LDOU  p,:T              P ← T.
BZ    p,:Underflow      Is T = Λ?
```

```
LDOU  t,p,LINK
STOU  t,:T               T ← LINK(P).
LDO   y,p,INFO           Y ← INFO(P).
STOU  :avail,p,LINK      LINK(P) ← AVAIL.
SET   :avail,p           AVAIL ← P.  ▮
```
(11)

[263]

Therefore we will assume that the objects to be sorted are numbered from 1 to n in any order. The input of the program will be in a `Buffer` as a sequential list of 256 pairs of `TETRA`s, where the pair $(j,\ k)$ means that object j precedes object k. The first pair, however, is $(0,\ n)$, where n is the number of objects. The pair (0, 0) terminates the input. We shall assume that $n + 1$ table entries plus the number of relation pairs will fit comfortably in memory; that the next input buffer can be obtained with '`LDA $255,InArgs; TRAP 0,Fread,Fin`' from a binary file; and that it is not necessary to check the input for validity. The output is to be the numbers of the objects in sorted order, followed by the number 0. Up to 512 of these numbers can be stored as `TETRA`s in the `Buffer`, before the buffer needs to be written to disk using the instructions '`LDA $255,OutArgs; TRAP 0,Fwrite,Fout`'.

[264]

The algorithm that follows uses a sequential table `X[0]`, `X[1]`, . . . , `X[`n`]`, and each node `X[`k`]` has the form

`COUNT[`k`]`	`TOP[`k`]`

.

Here `COUNT[`k`]` is the *number of direct predecessors* of object k (the number of relations $j \prec k$ that have appeared in the input), and `TOP[`k`]` is a link to the beginning of the *list of direct successors* of object k. The latter list contains entries in the format

`SUC`	`NEXT`

,

where `SUC` is a direct successor of k and `NEXT` is the next item of the list. To make the links in the `TOP[`k`]` and `NEXT` fields fit into one tetrabyte, we use relative addresses: All addresses are relative to a fixed global address `Base` and can be converted into an absolute address by adding `Base`.

. . .

The coding of Algorithm T in `MMIX` assembly language has a few additional points of interest. Since no deletion from tables is made in the algorithm (because no storage must be freed for later use), the operation `P ⇐ AVAIL` can be done in an extremely simple way, as shown in lines 11, 12, 24, and 25 below; we need not keep any linked pool of memory, and we can choose new nodes consecutively. The program includes complete input and output using `Fopen`, `Fread`, `Fwrite`, and `Fclose` system calls; the details of the data structures containing the parameters are omitted for the sake of simplicity. Right after the input buffer, we assume a `Sentinel`, the pair (0, 0), in memory. It allows us to assert in step T4 simultaneously that neither the end of the input nor the end of the buffer has been reached. The reader should not find it very difficult to follow the details of the coding in this program, since it corresponds directly to Algorithm T but with slight changes for efficiency. The efficient use of base addresses, which is an important aspect of linked memory processing, is illustrated here. We combine the conversion of relative addresses to absolute addresses and the addition of an appropriate offset to access a field by precomputing two base addresses (see line 13): `count ← Base+COUNT` and `top ← Base + TOP`. Using these bases, `COUNT[`*j*`]` and `TOP[`*j*`]` can be loaded or stored with a single instruction. The same applies to the `SUC` and `NEXT` fields; because they use the same base and incidentally the same offsets, we merely define `suc` as an alias for `count` and `next` for `top`. Again, `qlink` is just an alias for `count`. The code is further simplified by scaling object numbers by 8. This turns the object number k into the relative address of `X[`k`]`. Similarly, we define suitable base address `left` and `right` for loading pairs from the `Buffer`.

Program T (*Topological Sort*).

01	`:TSort`	`LDA`	`$255,InOpen`	1	<u>*T1. Initialize.*</u>
02		`TRAP`	`0,:Fopen,Fin`	1	Open input file.
03		`LDA`	`$255,IOArgs`	1	
04		`TRAP`	`0,:Fread,Fin`	1	Read first input buffer.
05		`SET`	`size,SIZE`	1	Load buffer size.
06		`LDA`	`left,Buffer+SIZE`	1	Point `left` to the buffer end.
07		`ADDU`	`right,left,4`	1	Point `right` to next `TETRA`.
08		`NEG`	`i,size`	1	$i \leftarrow 0$.
09		`LDT`	`n,right,i`	1	First pair is $(0, n)$, `n` ← n.
10		`ADD`	`i,i,8`	1	$i \leftarrow i + 1$.
11		`SET`	`:avail,8`	1	Allocate `QLINK[0]`.
12		`8ADDU`	`:avail,n,:avail`	1	Allocate n `COUNT` and `TOP` fields.
13		`LDA`	`count,Base+COUNT`	1	`count` ← `LOC(COUNT[0])`.
14		`LDA`	`top,Base+TOP`	1	`top` ← `LOC(TOP[0])`.

	`SL`	`k,n,3`	1	$k \leftarrow n$.
`1H`	`STCO`	`0,k,count`	$n+1$	Set (`COUNT[`k`]`, `TOP[`k`]`) $\leftarrow (0,0)$,
	`SUB`	`k,k,8`	$n+1$	for $0 \le k \le n$.
	`PBNN`	`k,1B`	$n+1_{[1]}$	Anticipate `QLINK[0]` $\leftarrow 0$ (step T4).
	`JMP`	`T2`	1	
`T3`	`SL`	`k,k,3`	m	<u>*T3. Record the relation.*</u>
	`LDT`	`t,k,count`	m	Increase `COUNT[`k`]` by one.
	`ADD`	`t,t,1`	m	
	`STT`	`t,k,count`	m	
	`SET`	`p,:avail`	m	`P` $\Leftarrow$ `AVAIL`.
	`ADD`	`:avail,:avail,8`	m	
	`STT`	`k,suc,p`	m	`SUC(P)` $\leftarrow k$.
	`SL`	`j,j,3`	m	
	`LDTU`	`t,top,j`	m	`NEXT(P)` $\leftarrow$ `TOP[`j`]`.
	`STTU`	`t,next,p`	m	
	`STTU`	`p,top,j`	m	`TOP[`j`]` $\leftarrow$ `P`.
`T2`	`LDT`	`j,left,i`	$m+b$	<u>*T2. Next relation.*</u>
	`LDT`	`k,right,i`	$m+b$	
	`ADD`	`i,i,8`	$m+b$	$i \leftarrow i+1$.
	`PBNZ`	`j,T3`	$m+b_{[b]}$	End of input or buffer?
`1H`	`BNP`	`i,T4`	$b_{[1]}$	End of input?
	`TRAP`	`0,:Fread,Fin`	$b-1$	Read next buffer.
	`NEG`	`i,size`	$b-1$	$i \leftarrow 0$.
	`JMP`	`T2`	$b-1$	
`T4`	`TRAP`	`0,:Fclose,Fin`	1	<u>*T4. Scan for zeros.*</u>
	`SET`	`r,0`	1	`R` $\leftarrow 0$.
	`SL`	`k,n,3`	1	$k \leftarrow n$.
`1H`	`LDT`	`t,k,count`	n	Examine `COUNT[`k`]`,
	`PBNZ`	`t,0F`	$n_{[a]}$	and if it is zero,
	`STT`	`k,qlink,r`	a	set `QLINK[R]` $\leftarrow k$,
	`SET`	`r,k`	a	and `R` $\leftarrow k$.
`0H`	`SUB`	`k,k,8`	n	
	`PBP`	`k,1B`	$n_{[1]}$	For $n \ge k > 0$.
	`LDT`	`f,qlink,0`	1	`F` $\leftarrow$ `QLINK[0]`.
	`LDA`	`$255,OutOpen`	1	Open output file.
	`TRAP`	`0,:Fopen,Fout`	1	

	NEG	i,size	1	Point i to the buffer start.
	JMP	T5	1	
T5B	PBN	i,0F	$n_{[c-1]}$	Jump if buffer is not full.
	LDA	$255,IOArgs	$c-1$	
	TRAP	0,:Fwrite,Fout	$c-1$	Flush output buffer.
	NEG	i,size	$c-1$	Point i to the buffer start.
0H	SUB	n,n,1	n	`n` ← `n` − 1.
	LDTU	p,top,f	n	`P` ← `TOP[F]`.
	BZ	p,T7	$n_{[d]}$	If `P` = Λ go to T7.
T6	LDT	s,suc,p	m	*T6. Erase relations.*
	LDT	t,s,count	m	Decrease `COUNT[SUC(P)]`.
	SUB	t,t,1	m	
	STT	t,s,count	m	
	PBNZ	t,0F	$m_{[n-a]}$	If zero,
	STT	s,qlink,r	$n-a$	set `QLINK[R]` ← `SUC(P)`,
	SET	r,s	$n-a$	and `R` ← `SUC(P)`.
0H	LDT	p,next,p	m	`P` ← `NEXT(P)`.
	PBNZ	p,T6	$m_{[n-d]}$	If `P` = Λ go to T7.
T7	LDT	f,qlink,f	n	*T7. Remove from queue.*
T5	SR	t,f,3	$n+1$	*T5. Output front of queue.*
	STT	t,left,i	$n+1$	Output the value of `F`.
	ADD	i,i,4	$n+1$	
	PBNZ	f,T5B	$n+1_{[1]}$	If `F` = 0 go to T8.
T8	LDA	$255,IOArgs	1	*T8. End of process.*
	TRAP	0,:Fwrite,Fout	1	Flush output buffer.
	TRAP	0,:Fclose,Fout	1	Close output file.
	POP	1,0		Return `n`. ▮

The analysis of Algorithm T is quite simple with the aid of Kirchhoff's law; the execution time has the approximate form $c_1 m + c_2 n$, where m is the number of input relations, n is the number of objects, and c_1 and c_2 are constants. It is hard to imagine a faster algorithm for this problem! The exact quantities in the analysis are given with Program T above, where a = number of objects with no predecessor, b = number of disk blocks in the input file = $\lceil (m+2)/256 \rceil$, c = number of disk blocks in the output file = $\lceil (n+2)/512 \rceil$, and d = number of objects with no successor (needed only for the analysis of bad guesses at the end of T4 and T6). Exclusive of input-output operations, with each `TRAP` instruction contributing only 5υ, the total running time in this case is only $(22m + 22n + 14b + 9c + 50)\upsilon + (12m + 6n + 2b + 4)\mu$.

EXERCISES

[269]

2. [*22*] Write a "general purpose" `MMIX` subroutine to do the insertion operation, (10). This subroutine should have the following specifications:

Calling sequence:	`PUSHJ $X,Insert`
Entry conditions:	`$0` ≡ `LOC(T)` and `$1` ≡ `Y`.
	`AVAIL` is kept in the global register `avail`.
Exit conditions:	The information `Y` is inserted just before the node that was pointed to by link variable `T`.

3. [*22*] Write a "general purpose" `MMIX` subroutine to do the deletion operation, (11). This subroutine should have the following specifications:

Calling sequence:	`PUSHJ $X,Delete`
	JMP Underflow
Entry conditions:	`$0` ≡ `LOC(T)`.
	`AVAIL` is kept in the global register `avail`.
Exit conditions:	If the stack whose pointer is the link variable `T` is empty, the first exit is taken. Otherwise, the top node of that stack is deleted, exit is made to the second instruction following '`PUSHJ`', and the return value in `$X` is the contents of the `INFO` field of the deleted node.

4. [*22*] The exercise for the `MIX` computer used the fact that the conditional jump to `OVERFLOW` could also be thought of as a subroutine call with a return to the instruction immediately preceding the call. The `MMIX` computer, as most computers now do, uses different instructions for subroutine calls and for conditional jumps, which are oneway streets with no return path. A comparable exercise for `MMIX` could use a calling convention as in exercise 3 and replace the '`JMP Underflow`' after the call to `Delete` with '`PUSHJ $255,Underflow`'. Another approach, used by many code libraries, is to provide a subroutine that combines the operation `P` ⇐ `AVAIL` with memory repacking and/or garbage collection. If in spite of all efforts sufficient memory is not available, these subroutines return Λ and leave it to the calling program to attempt a programspecific recovery. The following new exercise follows this second approach.

Show how to write an `MMIX` memory allocation subroutine `Allocate` following

(7). This subroutine should have the following specifications:

Calling sequence:	`PUSHJ $X,Allocate`
Entry conditions:	`AVAIL`, `POOLMAX`, and `SEQMIN` are kept in global registers.
Exit conditions:	If memory is available, return the address of a newly allocated node. Otherwise, the subroutine returns zero.

8. [*24*] Write an `MMIX` subroutine for the problem of exercise 7, taking the address of `FIRST` as a parameter. Try to design your program to operate as fast as possible.

. . .

22. [*23*] Program T assumes that its input file contains valid information, but a program that is intended for general use should always make careful tests on its input so that clerical errors can be detected, and so that the program cannot "destroy itself." For example, if one of the input relations for k were negative, Program T may erroneously change memory locations preceding array `X` when storing into `X[`k`]`. Suggest ways to modify Program T so that it is suitable for general use.

24. [*24*] Incorporate the extensions of Algorithm T made in exercise 23 into Program T.

26. [*29*] (*Subroutine allocation.*) Suppose that we have a large file containing the main subroutine library in relocatable form. The loading routine wants to determine the amount of relocation for each subroutine used, so that it can make one pass through the file to load the necessary routines.

. . .

One way to tackle this problem is to have a "file directory" that fits in memory. The loading routine has access to two tables:

a) The file directory. This table is composed of variable-length nodes that consist of two or more tetrabytes each. The first tetrabyte of a node contains the `SPACE` field, and the following tetrabytes contain one or more `LINK` fields.

Dir:		
	SPACE	$LINK_0$
	$LINK_1$	$LINK_2 \mid 1$
	SPACE	$LINK_0$
	$LINK_1 \mid 1$	SPACE
	$LINK_0$	...

In each node, SPACE is the number of tetrabytes required by the subroutine in the range $0 <$ SPACE $< 2^{31}$; $\texttt{LINK}_0$, the first LINK field, is a link to the directory entry for the subroutine that follows this subroutine in the linked list of entries or zero if this subroutine is the last. We implement links as relative addresses and ensure, by a suitable choice of the base address, that zero will not occur as link to a valid directory entry. The remaining LINK fields, $\texttt{LINK}_1$, $\texttt{LINK}_2$, . . . , $\texttt{LINK}_k$ $(k \geq 0)$, are links to the directory entries for any other subroutines required by this one. The LINK fields are normally even, because nodes are TETRA aligned. However, the last LINK field of a node has its least significant bit set to 1 to indicate the end of the node; this bit is ignored when using a LINK field as an address in load or store instructions. The relative address of the directory entry for the first subroutine of the library file is specified by the link variable FIRST.

b) The list of subroutines directly referred to by the program to be loaded. This is stored in consecutive octabytes X[0], X[1], . . . , X[N − 1], where N ≥ 0 is a variable known to the loading routine. Each octabyte in this list has the following form:

BASE	SUB

.

Initially, only the SUB field is used, for the offsets of the directory entries for the subroutines desired; the BASE field is unused.

The loading routine also knows MLOC, the amount of relocation to be used for the first subroutine loaded.

As a small example, consider the following configuration:

File directory

#1000:	20	#1024
#1008:	#100D	30
#1010:	#1049	200
#1018:	#1034	#1000
#1020:	#102D	100
#1028:	#1015	60
#1030:	#1001	200
#1038:	#0000	#1024
#1040:	#100C	#102D
#1048:	20	#102D

List of subroutines needed

X[0]:		#1014
X[1]:		#1048

with N $= 2$, FIRST = #100C, and MLOC = 2400.

The file directory in this case shows that the subroutines on file are #100C, #1048, #102C, #1000, #1024, #1014, and #1034 in that order. Subroutine #1034 takes 200 TETRAs and implies the use of subroutines #1024, #100C, and #102C; etc. The program to be loaded requires #1014 and #1048, which are to be placed into locations ≥ 2400. These subroutines in turn imply that #1000, #102C, and #100C must also be loaded.

The subroutine allocator is to change the X-table so that in each entry X[0], X[1], . . . , the SUB field is a subroutine to be loaded and the BASE field is the amount of its relocation. These entries are to be in the order in which the subroutines appear in the file directory. The last entry contains the first unused memory address and a zero link field.

One possible answer for the example above would be:

X[0]:	2400	#100C
X[1]:	2430	#1048
X[2]:	2450	#102C
X[3]:	2510	#1000
X[4]:	2530	#1014
X[5]:	2730	#0000

The problem in this exercise is to design an algorithm for the stated task.

27. [*25*] Write an MMIX program for the subroutine allocation algorithm of exercise 26.

2.2.4. Circular Lists

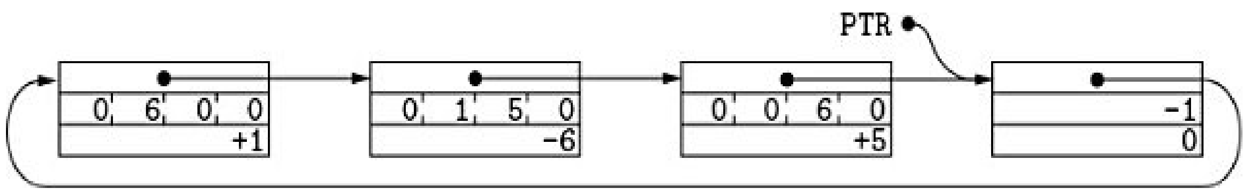
[275]

We will consider here the two operations of addition and multiplication. Let us suppose that a polynomial is represented as a list in which each node stands for one nonzero term, and has the three-octabyte form

LINK			
SIGN	A	B	C
COEF			

. (5)

Here `COEF` contains the (signed) coefficient of the term $x^A y^B z^C$. We will assume that the coefficients and exponents will always lie in the range allowed by this format, and that it is not necessary to check the ranges during our calculations. The notation `ABC` will be used to stand for the `SIGN A B C` fields of the node (5), treated as a single octabyte. The `SIGN` field will always be zero, except that there is a *special node* at the end of every polynomial that has `ABC` $= -1$ and `COEF` $= 0$. This special node is a great convenience, analogous to our discussion of a list head above, because it provides a convenient sentinel and it avoids the problem of an empty list (corresponding to the polynomial 0). Actually, only the sign bit of the `SIGN` field is necessary to tag the sentinel node; if required, the remaining 15 bits could be used to accommodate a fourth exponent. The nodes of the list always appear in *decreasing order* of the `ABC` field, if we follow the direction of the links, except that the last node (which has `ABC` $= -1$) links to the largest value of `ABC`. For example, the polynomial $x^6 - 6xy^5 + 5y^6$ would be represented thus:



. . .

The programming of Algorithm A in `MMIXAL` language shows again the ease with which linked lists are manipulated in a computer. In the following code, we assume that the global register `avail` points to a sufficiently large stack of available nodes.

Program A (*Addition of polynomials*). The subroutine expects two parameters, `p` ≡ polynomial(`P`) and `q` ≡ polynomial(`Q`). It will replace polynomial(`Q`) by polynomial(`Q`) + polynomial(`P`).

:Add	SET	q1,q	$1 + m''$	A1. Initialize. `Q1` ← `Q`.
	LDOU	q,q,LINK	$1 + m''$	`Q` ← `LINK(Q)`.
0H	LDOU	p,p,LINK	$1 + p$	`P` ← `LINK(P)`.
	LDO	coefp,p,COEF	$1 + p$	`coefp` ← `COEF(P)`.
	LDO	abcp,p,ABC	$1 + p$	A2. `ABC(P)`: `ABC(Q)`.
2H	LDO	t,q,ABC	x	`t` ← `ABC(Q)`.
	CMP	t,abcp,t	x	Compare `ABC(P)` and `ABC(Q)`.
	BZ	t,A3	$x_{[m+1]}$	If equal, go to A3.
	BP	t,A5	$p' + q'_{[p']}$	If greater, go to A5.

```
      SET    q1,q                  q'          If less, set Q1 ← Q.
      LDOU   q,q,LINK              q'          Q ← LINK(Q).
      JMP    2B                    q'          Repeat.
A3    BN     abcp,6F               m + 1[1]    A3. Add coefficients.
      LDO    coefq,q,COEF          m           coefq ← COEF(Q).
      ADD    coefq,coefq,coefp     m           coefq ← coefq + coefp.
      STO    coefq,q,COEF          m           COEF(Q) ← COEF(Q) + COEF(P).
      PBNZ   coefq,:Add            m[m']       Jump if nonzero.
      SET    q2,q                  m'          A4. Delete zero term. Q2 ← Q.
      LDOU   q,q,LINK              m'          Q ← LINK(Q).
      STOU   q,q1,LINK             m'          LINK(Q1) ← Q.
      STOU   :avail,q2,LINK        m'
      SET    :avail,q2             m'          AVAIL ⇐ Q2.
      JMP    0B                    m'          Go to advance P.
A5    SET    q2,:avail             p'          A5. Insert new term.
      LDOU   :avail,:avail,LINK    p'          Q2 ⇐ AVAIL.
      STO    coefp,q2,COEF         p'          COEF(Q2) ← COEF(P).
      STOU   abcp,q2,ABC           p'          ABC(Q2) ← ABC(P).
      STOU   q,q2,LINK             p'          LINK(Q2) ← Q.
      STOU   q2,q1,LINK            p'          LINK(Q1) ← Q2.
      SET    q1,q2                 p'          Q1 ← Q2.
      JMP    0B                    p'          Go to advance P.
6H    POP    0,0                               Return from subroutine. ▮
```

. . .

The analysis given with Program A uses the abbreviations

$$m = m' + m'', \quad p = m + p', \quad q = m + q', \quad x = 1 + m + p' + q';$$

the running time for MMIX is $(21m' + 15m'' + 17p' + 7q' + 13)\upsilon + (9m' + 7m'' + 9p' + 2q' + 5)\mu$.

EXERCISES

[279]

11. [*24*] . . . Write an MMIX subroutine with the following specifications:

Calling sequence: `PUSHJ $X,Copy`

Entry conditions:	`$0` ≡ polynomial`(P)`.
Exit conditions:	Returns a pointer to a newly created polynomial equal to polynomial`(P)`.

12. [*21*] Compare the running time of the program in exercise 11 with that of Algorithm A when the polynomial`(Q)` = 0.

13. [*20*] Write an `MMIX` subroutine with the following specifications:

Calling sequence:	`PUSHJ $X,Erase`
Entry conditions:	`$0` ≡ polynomial`(P)`.
Exit conditions:	polynomial`(P)` has been added to the `AVAIL` list.

[*Note:* This subroutine can be used in conjunction with the subroutine of exercise 11 in the sequence '`LDOU t+1,Q; PUSHJ t,Erase; LDOU t+1,P; PUSHJ t,Copy; STOU t,Q`' to achieve the effect "polynomial`(Q)` ← polynomial`(P)`".]

14. [*22*] Write an `MMIX` subroutine with the following specifications:

Calling sequence:	`PUSHJ $X,Zero`
Entry conditions:	None
Exit conditions:	Returns a newly created polynomial equal to 0.

15. [*24*] Write an `MMIX` subroutine to perform Algorithm M, having the following specifications:

Calling sequence:	`PUSHJ $X,Mult`
Entry conditions:	`$0` ≡ polynomial`(Q)`, `$1` ≡ polynomial`(M)`, `$2` ≡ polynomial`(P)`.
Exit conditions:	polynomial`(Q)` ← polynomial`(Q)` + polynomial`(M)` × polynomial`(P)`.

[*Note:* Modify Program A by adding an outer loop on `M` and a multiplication by one term of `M` in the inner loop.]

16. [*M28*] Estimate the running time of the subroutine in exercise 15 in terms of some relevant parameters.

2.2.5. Doubly Linked Lists

[282]

. . . Corresponding to these buttons, there are two variables `CALLUP` and `CALLDOWN`, in which the five least significant bits each represent one button. There is also a variable `CALLCAR` representing with its bits the buttons within the elevator car, which direct it to the destination floor. The individual bits are denoted by `CALLUP[`j`]`, `CALLDOWN[`j`]`, and `CALLCAR[`j`]` in the following algorithms, for $0 \leq j \leq 4$. When a person presses a button, the appropriate bit in one of these variables is set to 1; the elevator clears the bit to 0 after the request has been fulfilled.

So far we have described the elevator from the user's point of view; the situation is more interesting as viewed by the elevator. The elevator is in one of three states: `GOINGUP` (`STATE` > 0), `GOINGDOWN` (`STATE` < 0), or `NEUTRAL` (`STATE` $= 0$).

[288]

Each node representing an activity (whether a user or an elevator action) has the form

<table>
<tr><td colspan="8">LLINK1</td></tr>
<tr><td colspan="8">RLINK1</td></tr>
<tr><td colspan="8">NEXTINST</td></tr>
<tr><td colspan="4">NEXTTIME</td><td></td><td></td><td>IN</td><td>OUT</td></tr>
<tr><td colspan="8">LLINK2</td></tr>
<tr><td colspan="8">RLINK2</td></tr>
</table>

. (6)

[289]

First comes a number of lines of code that just serve to define the initial contents of the tables. There are several points of interest here: We have list heads for the `WAIT` list (line 010), the `QUEUE` lists (lines 020–024), and the `ELEVATOR` list (line 026). Each of them is a node of the form (6), but with unimportant words deleted; the `WAIT` list head contains only the first four octabytes of a node, while the `QUEUE` and `ELEVATOR` list heads require only the last two octabytes of a node. For convenience, we set up global registers `wait`, `queue`, and `elevator` pointing to these list heads. We have also four nodes that are always present in the system (lines 011–015): `USER1`, a node that is always positioned at step U1 ready to enter a new user into the system; `ELEV1`, a node that governs the main actions of the elevator at steps E1, E2, E3, E4, E6, E7, and E8; and `ELEV2` and `ELEV3`, nodes that are used for the elevator actions E5 and E9, which take place independently of other elevator actions with respect to simulated time. Each of these four nodes contains only four octabytes, since they never appear in the `QUEUE` or `ELEVATOR` lists. The nodes representing each actual user in the system will appear in the pool segment.

```
001 LLINK1      IS   0            Definition of fields
002 RLINK1      IS   8
003 NEXTINST    IS   16
004 NEXTTIME    IS   24
005 IN          IS   30
```

```
IN          IS     30
OUT         IS     31
LLINK2      IS     32
RLINK2      IS     40
            LOC    Data_Segment
WAIT        OCTA   USER1,USER1,0,0   List head for WAIT list
USER1       OCTA   WAIT,WAIT,U1,0    User action U1
wait        GREG   WAIT              Pointer to WAIT list head
ELEV1       OCTA   0,0,E1,0          Elevator actions except E5 and E9
ELEV2       OCTA   0,0,E5,0          Elevator action E5
ELEV3       OCTA   0,0,E9,0          Elevator action E9
time        GREG   0                 Current simulated time
c           GREG   0                 Current node
c0          GREG   0                 Backup for current node
queue       GREG   @-4*8             Pointer to QUEUE[0] list head
            OCTA   @-4*8,@-4*8       List head for QUEUE[0]
            OCTA   @-4*8,@-4*8       List head for QUEUE[1]
            OCTA   @-4*8,@-4*8       (All queues are
            OCTA   @-4*8,@-4*8         initially empty.)
            OCTA   @-4*8,@-4*8       List head for QUEUE[4]
elevator    GREG   @-4*8             Pointer to ELEVATOR list head
            OCTA   @-4*8,@-4*8       List head for ELEVATOR
callup      GREG   0
calldown    GREG   0
callcar     GREG   0
off         IS     0
on          GREG   1
floor       GREG   0
d1          GREG   0                 Indicates doors open, activity
d2          GREG   0                 Indicates no prolonged standstill
d3          GREG   0                 Indicates doors open, inactivity
state       GREG   0                 -1 going down, 0 neutral, +1 going up
dt          GREG   0                 Hold time
fi          GREG   0                 Floor IN
fo          GREG   0                 Floor OUT
tg          GREG   0                 Give-up time
```

The next part of the program coding contains basic subroutines and the main control routines for the simulation process. Subroutines `Insert` and `Delete` perform typical manipulations on doubly linked lists; they put the current node `C` into or take it out of a `QUEUE` or `ELEVATOR` list. There are also subroutines for the `WAIT` list: Subroutine `SortIn` adds the current node to the `WAIT` list, sorting it into the right place based on its `NEXTTIME` field. Subroutine `Immed` inserts the current node at the front of the `WAIT` list. Subroutine `Hold` puts the current node into the `WAIT` list, with `NEXTTIME` equal to the current time plus the amount in register `dt`. Subroutine `DeleteW` deletes the current node from the `WAIT` list.

The heart of the simulation control is the scheduling of the coroutines. The following program implements these subroutines as `TRIP` handlers, and we will see that `TRIP`s are very flexible and convenient for this kind of "system programming." `TRIP Cycle,0` decides which activity is to be performed next (namely, the first element of the `WAIT` list, which we know is nonempty) and jumps to it. There are three special entrances to `Cycle`: `Cycle1` first sets `NEXTINST` in the current node; `HoldC` is the same with an additional call on the `Hold` subroutine using the global register `dt` to specify the hold time; and `HoldCI` is like `HoldC` but with the hold time given as an immediate value in the `Z` field of the `TRIP` instruction. Thus, the effect of the instruction '`TRIP HoldC,0`' with amount t in register `dt` or of '`TRIP HoldCI,`t' is to suspend activity for t units of simulated time and then return to the following location.

The implementation that follows will not save and restore the complete context of each coroutine; in particular, it will not save the contents of local registers. Consequently, it is not possible to use `TRIP`s inside of subroutines because the register stack would be corrupted. This is a small inconvenience but it simplifies the code.

```
          LOC    0               TRIP entry point
          GET    $0,rX           $0 ← TRIP X,Y,Z.
          GET    $1,rW           $1 ← rW (the return address).
          SR     $2,$0,16        Extract X field
          AND    $2,$2,#FF         and
          GO     $2,$2,0           dispatch depending on X.
Cycle1    STOU   $1,c,NEXTINST   Set NEXTINST(C) ← rW.
          JMP    Cycle
HoldCI    AND    dt,$0,#FF       Set dt ← Z.
HoldC     STOU   $1,c,NEXTINST   Set NEXTINST(C) ← rW.
          PUSHJ  $0,Hold         Insert NODE(C) in WAIT with delay dt.
```

```
Cycle    LDOU    c,wait,RLINK1      Set C ← RLINK1(LOC(WAIT)).
         LDTU    time,c,NEXTTIME    TIME ← NEXTTIME(C).
         PUSHJ   $0,DeleteW         Remove NODE(C) from WAIT list.
         LDOU    $0,c,NEXTINST
         PUT     rW,$0              rW ← NEXTINST(C).
         RESUME  0                  Return.
         LOC     #100
         PREFIX  :queue:
p        IS      $0                 Parameter for Insert
q        IS      $1                 Local variable
:Insert  LDOU    q,p,:LLINK2        Insert NODE(C) to left of NODE(P).
         STOU    q,:c,:LLINK2
         STOU    :c,p,:LLINK2
         STOU    :c,q,:RLINK2
         STOU    p,:c,:RLINK2
         POP     0,0
:Delete  LDOU    p,:c,:LLINK2       Delete NODE(C) from its list.
         LDOU    q,:c,:RLINK2
         STOU    p,q,:LLINK2
         STOU    q,p,:RLINK2
         POP     0,0
         PREFIX  :wait:
tc       IS      $0                 Parameter for SortIn
q        IS      $1                 Local variables
p        IS      $2
tp       IS      $3
t        IS      $4
:Immed   SET     tc,:time           Insert NODE(C) first in WAIT list.
         STTU    tc,:c,:NEXTTIME
         SET     p,:wait
         JMP     2F
:Hold    ADDU    tc,:time,:dt       Add delay dt to current TIME.
:SortIn  STTU    tc,:c,:NEXTTIME    Sort NODE(C) into WAIT list.
         SET     p,:wait            P ← wait.
1H       LDOU    p,p,:LLINK1        P ← LLINK1(P).
         LDTU    tp,p,:NEXTTIME     tp ← NEXTTIME(P).
```

```
088             CMP     t,tp,tc            Compare NEXTTIME fields, right to left.
089             BP      t,1B               Repeat until tp ≤ tc.
090  2H         LDOU    q,p,:RLINK1        Insert NODE(C) right of NODE(P).
091             STOU    q,:c,:RLINK1
092             STOU    p,:c,:LLINK1
093             STOU    :c,p,:RLINK1
094             STOU    :c,q,:LLINK1
095             POP     0,0
096  :DeleteW   LDOU    p,:c,:LLINK1       Delete NODE(C) from WAIT list.
097             LDOU    q,:c,:RLINK1       (This is the same as lines 068–071
098             STOU    p,q,:LLINK1          except LLINK1, RLINK1 are used
099             STOU    q,p,:RLINK1          instead of LLINK2, RLINK2.)
100             POP     0,0                ▮
```

Now comes the program for Coroutine U. At the beginning of step U1, the function `Values` will initialize `fi`, `fo`, `tg`, and `dt` by generating new values for `IN`, `OUT`, `GIVEUPTIME`, and `INTERTIME`. After these quantities have been computed, line 103 of the program causes the current node `C`, which is `USER1` (see line 011 above) to be reinserted into the `WAIT` list so that the next user will be generated after `dt` = `INTERTIME` units of simulated time. The following lines 104–106 create a new node using the function `Allocate` and record the values of `fi` and `fo` in this node. The give-up time `tg` is used in line 139 when the new node enters the `WAIT` list. The node is returned to free storage in step U6 by calling the subroutine `Free` (line 146).

```
101        PREFIX  :
102  U1    PUSHJ   $0,Values          U1. Enter, prepare for successor.
103        PUSHJ   $0,Hold            Put NODE(C) in WAIT list.
104        PUSHJ   $0,Allocate        Allocate new NODE(C).
105        STB     fi,c,IN
106        STB     fo,c,OUT
107  U2    SET     c0,c               U2. Signal and wait. Save value of C.
108        CMP     $0,fi,floor
109        BNZ     $0,2F              Jump if FLOOR ≠ fi.
110        LDA     c,ELEV1            Set current coroutine to ELEV1.
111        LDOU    $0,c,NEXTINST
112        GETA    $1,E6
113        CMPU    $0,$0,$1           Is elevator positioned at E6?
```

```
        BNZ     $0,3F
        GETA    $0,E3
        STOU    $0,c,NEXTINST           If so, reposition at E3.
        PUSHJ   $0,DeleteW              Remove it from WAIT list
        JMP     4F                        and reinsert it at front of WAIT.
3H      BZ      d3,2F                   Jump if not waiting;
        SET     d3,off                    otherwise, make it not waiting,
        SET     d1,on                     but loading.
4H      PUSHJ   $0,Immed                Schedule ELEV1 for
        JMP     U3                        immediate execution.
2H      SL      $1,on,fi                Elevator is not on floor fi.
        CMP     $0,fo,fi
        ZSP     $2,$0,$1
        OR      callup,callup,$2
        ZSN     $2,$0,$1
        OR      calldown,calldown,$2    Press buttons.
        BZ      d2,0F                   If not busy, make a decision.
        LDOU    $0,ELEV1+NEXTINST
        GETA    $1,E1
        CMP     $0,$0,$1                Elevator at E1?
        BNZ     $0,U3                   If yes,
0H      PUSHJ   $0,Decision               make a decision.
U3      SET     c,c0                    U3. Enter queue. Restore C.
        16ADDU  $1,fi,queue
        PUSHJ   $0,Insert               Insert NODE(C) at right end of QUEUE[IN].
U4A     SET     dt,tg
        TRIP    HoldC,0                 Wait GIVEUPTIME units.
U4      LDB     fi,c,IN                 U4. Give up.
        CMP     $0,fi,floor
        BNZ     $0,U6                   Give up if fi ≠ FLOOR.
        BNZ     d1,U4A                  See exercise 7.
U6      PUSHJ   $0,Delete               U6. Get out.
        PUSHJ   $0,Free                 AVAIL ⇐ C.
        TRIP    Cycle,0                 Continue simulation.
U5      PUSHJ   $0,Delete               U5. Get in. Delete C from QUEUE.
        SET     $1,elevator
```

```
          PUSHJ  $0,Insert           Insert it at right of ELEVATOR.
          LDB    fo,c,OUT
          SL     $0,on,fo
          OR     callcar,callcar,$0  Set bit CALLCAR[OUT(C)] ← 1.
          BZ     state,1F
          TRIP   Cycle,0
1H        CMP    state,fo,floor      STATE ← 1, 0, or −1.
          LDA    c,ELEV2
          PUSHJ  $0,DeleteW          Remove E5 action from WAIT list.
          TRIP   HoldCI,25
          JMP    E5                  Restart E5 action 25 units from now. ▮
```

The function `Allocate` and `Free` perform the actions ‘`C` ⇐ `AVAIL`’ and ‘`AVAIL` ⇐ `C`’ using the `POOLMAX` technique; no test for `Overflow` is necessary here, since the total size of the storage pool (the number of users in the system at any one time) rarely exceeds 10 nodes (480 bytes).

```
avail     GREG  0                List of available nodes
poolmax   GREG  0                Location of pool memory
Allocate  PBNZ  avail,1F         C ⇐ AVAIL using 2.2.3–(7).
          SET   c,poolmax
          ADDU  poolmax,c,6*8
          POP   1,0
1H        SET   c,avail
          LDOU  avail,c,LLINK1
          POP   1,0
Free      STOU  avail,c,LLINK1   AVAIL ⇐ C using 2.2.3–(5).
          SET   avail,c
          POP   0,0              ▮
```

The program for Coroutine E is a rather straightforward rendition of the semi-formal description given earlier. Perhaps the most interesting portion is the preparation for the elevator’s independent actions in step E3, and the searching of the `ELEVATOR` and `QUEUE` lists in step E4.

```
E1A       TRIP   Cycle1,0            Set NEXTINST ← E1, go to Cycle.
E1        IS     @                   E1. Wait for call. (no action)
E2A       TRIP   HoldC,0             Decelerate.
E2        OR     $0,callup,calldown  E2. Change of state?
```

```
        OR     $0,$0,callcar
        BN     state,1F                Jump if going down.
        ADD    $1,floor,1              State is GOINGUP.
        SR     $2,$0,$1
        BNZ    $2,E3                   Are there calls for higher floors?
        NEG    $1,64,floor             If not, have passengers in the
        SL     $2,callcar,$1             elevator called for lower floors?
        JMP    2F
1H      NEG    $1,64,floor             State is GOINGDOWN.
        SL     $2,$0,$1
        BNZ    $2,E3                   Are there calls for lower floors?
        ADD    $1,floor,1              If not, have passengers in the
        SR     $2,callcar,$1             elevator called for upper floors?
2H      NEG    state,state             Reverse direction of STATE.
        CSZ    state,$2,0              STATE ← NEUTRAL or reversed.
        SL     $0,on,floor
        ANDN   callup,callup,$0        Set all CALL bits to zero.
        ANDN   calldown,calldown,$0
        ANDN   callcar,callcar,$0
E3      LDA    c,ELEV3                 E3. Open doors.
        LDO    $0,c,LLINK1
        BZ     $0,1F                   If activity E9 is already scheduled,
        PUSHJ  $0,DeleteW                remove it from the WAIT list.
1H      SET    dt,300
        PUSHJ  $0,Hold                 Schedule activity E9 after 300 units.
        LDA    c,ELEV2
        SET    dt,76
        PUSHJ  $0,Hold                 Schedule activity E5 after 76 units.
        SET    d2,on
        SET    d1,on
        SET    dt,20
E4A     LDA    c,ELEV1
        TRIP   HoldC,0
E4      LDA    $0,elevator             E4. Let people out, in.
        LDA    c,elevator              C ← LOC(ELEVATOR).
1H      LDOU   c,c,LLINK2              C ← LLINK2(C).
```

```
         CMP     $1,c,$0               Search ELEVATOR list, right to left.
         BZ      $1,1F                 If C = LOC(ELEVATOR), search is complete.
         LDB     $1,c,OUT
         CMP     $1,$1,floor           Compare OUT(C) with FLOOR.
         BNZ     $1,1B                 If not equal, continue searching;
         GETA    $0,U6                   otherwise, send user to U6.
         JMP     2F
1H       16ADDU  $0,floor,queue
         LDOU    c,$0,RLINK2           Set C ← RLINK2(LOC(QUEUE[FLOOR])).
         LDOU    $1,c,RLINK2
         CMP     $1,$1,c               Is C = RLINK2(C)?
         BZ      $1,1F                 If so, the queue is empty.
         PUSHJ   $0,DeleteW            If not, cancel action U4 for this user.
         GETA    $0,U5                 Prepare to replace U4 by U5.
2H       STOU    $0,c,NEXTINST         Set NEXTINST(C).
         PUSHJ   $0,Immed              Put user at the front of the WAIT list.
         SET     dt,25
         JMP     E4A                   Wait 25 units and repeat E4.
1H       SET     d1,off
         SET     d3,on
         TRIP    Cycle,0               Return to simulate other events.
E5       BZ      d1,0F                 E5. Close doors.
         TRIP    HoldCI,40             If people are still getting in or out,
         JMP     E5                      wait 40 units and repeat E5.
0H       SET     d3,off                If not loading, stop waiting.
         LDA     c,ELEV1
         TRIP    HoldCI,20             Wait 20 units, then go to E6.
E6       SL      $0,on,floor           E6. Prepare to move.
         ANDN    callcar,callcar,$0    Reset CALLCAR on this floor.
         ZSNN    $1,state,$0           If not going down,
         ANDN    callup,callup,$1        reset CALLUP on this floor.
         ZSNP    $1,state,$0           If not going up,
         ANDN    calldown,calldown,$1    reset CALLDOWN on this floor.
         PUSHJ   $0,Decision
E6B      BZ      state,E1A             If STATE = NEUTRAL, go to E1 and wait.
         BZ      d2,0F
```

```
249        LDA    c,ELEV3              If busy,
250        PUSHJ  $0,DeleteW             cancel activity E9
251        STCO   0,c,LLINK1             (see line 197).
252  0H    LDA    c,ELEV1
253        TRIP   HoldCI,15            Wait 15 units of time.
254        BN     state,E8             If STATE = GOINGDOWN, go to E8.
255  E7    ADD    floor,floor,1        E7. Go up a floor.
256        TRIP   HoldCI,51            Wait 51 units.
257        SL     $0,on,floor
258        OR     $1,callcar,callup
259        AND    $2,$1,$0             Is CALLCAR[FLOOR] ≠ 0
260        BNZ    $2,1F                  or CALLUP[FLOOR] ≠ 0?
261        CMP    $2,floor,2
262        BZ     $2,2F                If not, is FLOOR = 2?
263        AND    $2,calldown,$0       If not, is CALLDOWN[FLOOR] ≠ 0?
264        BZ     $2,E7                If not, repeat step E7.
265  2H    OR     $1,$1,calldown
266        ADD    $2,floor,1
267        SR     $1,$1,$2
268        BNZ    $1,E7                Are there calls for higher floors?
269  1H    SET    dt,14                It is time to stop the elevator.
270        JMP    E2A                  Wait 14 units and go to E2.
           ⋮                           (See exercise 8.)
287  E9    STCO   0,c,LLINK1           E9. Set inaction indicator. (See line 197.)
288        SET    d2,off
289        PUSHJ  $0,Decision
290        TRIP   Cycle,0              Return to simulation of other events. ▮
```

We will not consider here the `Decision` subroutine (see exercise 9), nor the `Values` subroutine that is used to specify the demands on the elevator. At the very end of the program comes the code

```
Main  SET   floor,2                      Start with FLOOR = 2,
      SET   state,0                        STATE = NEUTRAL,
      SETH  poolmax,Pool_Segment>>48       and no extra nodes.
      TRIP  Cycle,0,0                    Begin simulation. ▮
```

. . .

. . . The author made such an experiment with the elevator program above, running it for 10000 units of simulated elevator time; 26 users entered the simulated system. The instructions in the `SortIn` loop, lines 086–089, were executed by far the most often, 1432 times, while the `SortIn` subroutine itself was called 437 times. The `Cycle` routine was performed 407 times; so we could gain a little speed by not calling the `DeleteW` subroutine at line 054: The four lines of that subroutine could be written out in full (to save 4υ each time `Cycle` is used). The profiler also showed that the `Decision` subroutine was called only 32 times and the loop in E4 (lines 212–217) was executed only 142 times.

EXERCISES

[297]

7. [*25*] . . .

Assume that line 144 said ‘`BZ D1,U6; TRIP Cycle,0`’ instead of ‘`BNZ D1,U4A`’.

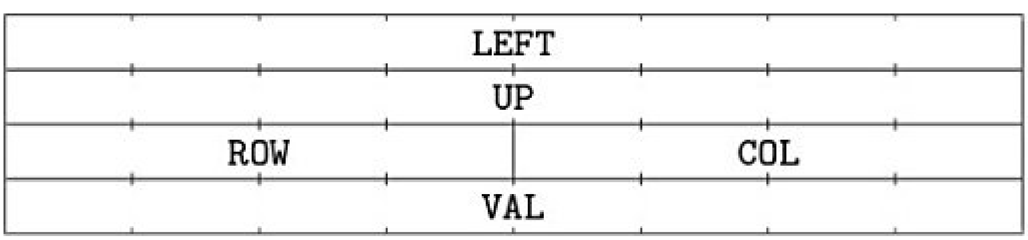
8. [*21*] Write the code for step E8, lines 271–286, which has been omitted from the program in the text.

9. [*23*] Write the code for the `Decision` subroutine, which has been omitted from the program in the text.

2.2.6. Arrays and Orthogonal Lists

[302]

The representation we will discuss consists of circularly linked lists for each row and column. Every node of the matrix contains four octabytes and five fields:



(11)

. . .

There are special list head nodes, `BASEROW[`i`]` and `BASECOL[`j`]`, for every row and column. These nodes are identified by odd links pointing to them. So `UP(P)` is odd if and only if `UP(P)` = `LOC(BASECOL[`j`])` | 1, and `LEFT(P)` is odd if and only if `LEFT(P)` = `LOC(BASEROW[`i`])` | 1.

. . .

Using sequential allocation of storage, a 400 × 400 matrix would fill more than 1 MByte, and this is more memory than used to fit in the cache of many computers; but a suitable sparse 400 × 400 matrix can be represented even in a small 64 KByte level 1 cache.

[305]

The programming of this algorithm is left as a very instructive exercise for the reader (see exercise 15). It is worth pointing out here that it is necessary to allocate only one octabyte to each of the nodes `BASEROW[`i`]`, `BASECOL[`j`]`, since most of their fields are irrelevant. (See the shaded areas in Fig. 14, and see the program of Section 2.2.5.) Furthermore there is one additional octabyte required for each `PTR[`j`]`.

EXERCISES

[306]

5. [*20*] Show that it is possible to bring the value of `A[J,K]` into register `a` in one `MMIX` instruction, using the indirect addressing feature of exercise 2.2.2–3, even when `A` is a *triangular* matrix as in (9). (Assume that the values of `J` and `K` are in registers `$1` and `$2`, respectively.)

11. [*11*] Suppose that we have a 400 × 400 matrix in which there are at most four nonzero entries per row. How much storage is required to represent this matrix as in Fig. 14, if we use four octabytes per node except for list heads, which will use one octabyte?

15. [*29*] Write an `MMIX` program for Algorithm S. Assume that the `VAL` field is a floating point number.

2.3.1. Traversing Binary Trees

[324]

For threaded trees, it turns out that things will work nicely if `NODE(LOC(T))` is made into a "list head" for the tree, with

$$\texttt{LLINK(HEAD)} = \texttt{T}, \qquad \texttt{LTAG(HEAD)} = 0,$$
$$\texttt{RLINK(HEAD)} = \texttt{HEAD}, \qquad \texttt{RTAG(HEAD)} = 0. \qquad (8)$$

(Here `HEAD` denotes `LOC(T)`, the address of the list head.) An empty threaded tree will satisfy the conditions

`LLINK(HEAD) = HEAD`, `LTAG(HEAD)` $= 1.$ (9)

. . .

With these preliminaries out of the way, we are now ready to consider `MMIX` versions of Algorithms S and T. The following programs assume that binary tree nodes have the three-word form

RLINK	RTAG
LLINK	LTAG
INFO	

.

The two `TAG`s are stored in the least significant bit of the link fields. In an unthreaded tree, both `TAG`s will always be zero and terminal links will be represented by zero. In a threaded tree, the least significant bits of the link fields come “for free,” because pointer values will generally be even, and `MMIX` ignores the low-order bits when addressing memory.

The following two subroutines traverse a binary tree in symmetric order (that is, inorder), calling the subroutine `Visit` periodically; that subroutine is given a pointer to the node that is currently of interest.

Program T (*Traverse binary tree inorder*). In this implementation of Algorithm T, the stack is kept conveniently on the register stack. While this might appear to be less memory efficient—the register stack stores three octabytes per nesting level instead of only one—it is just making good use of the available hardware. After all, if the tree is well balanced, the 256 registers in the register ring will go a long way. The subroutine expects two parameters: `p` ≡ `LOC(HEAD)`, the address of the root node of the tree; and `visit` ≡ `LOC(Visit)`, the address of a subroutine to be called for every node in the tree.

```
:Inorder  PBZ     p,T4           n + 1[a]   T2. P = Λ?
          GET     rJ,:rJ         a
T3        LDOU    t+1,p,LLINK    n          T3. Stack ⇐ P.
          SET     t+2,visit      n
          PUSHJ   t,:Inorder     n          Call Inorder(LLINK(P),Visit).
T5        SET     t+1,p          n          T5. Visit P.
          PUSHGO  t,visit,0      n          Call Visit(P).
          LDOU    p,p,RLINK      n          P ← RLINK(P).
          BNZ     p,T3           n[n−a]     T2. P = Λ?
          PUT     :rJ,rJ         a
```

```
T4    POP     0,0             n+1       T4. P ⇐ Stack. ▮
```

Program S (*Symmetric successor in a threaded binary tree*). Algorithm S has been augmented to form a complete subroutine comparable to Program T.

```
:Inorder  GET     rJ,:rJ        1          S0. Initialize.
          SET     head,p        1          Remember HEAD.
          JMP     S2            1          Skip step S1.
S3        PUSHGO  t,visit,0     n          S3. Visit P.
S1        LDOU    p,p,RLINK     n          S1. RLINK(P) a thread?
          BOD     p,1F          n[a]       If RTAG(P) = 1, visit P.
S2        LDOU    t,p,LLINK     n+1        S2. Search to left.
          CSEV    p,t,t         n+1        If LTAG(P) = 0, set P ← LLINK(P)
          BEV     t,S2          n+1[a]       and repeat this step.
1H        ANDN    t+1,p,1       n+1        Untag P and prepare to visit P.
          CMP     t,t+1,head    n+1        Unless P = HEAD,
          PBNZ    t,S3          n+1[1]     visit P.
9H        PUT     :rJ,rJ        1
          POP     0,0                      ▮
```

[326]

The analysis tells us Program T takes $(15n + 2a + 4)\upsilon + 2n\mu$, and Program S takes $(11n + 4a + 12)\upsilon + (2n + 1)\mu$, where n is the number of nodes in the tree and a is the number of terminal right links (nodes with no right subtree).

EXERCISES

[332]

20. [*20*] Modify Program T so that it maintains an explicit stack, instead of using the implicit register stack provided by `PUSHJ`. The stack can be kept in consecutive memory locations or in a linked list.

22. [*25*] Write an `MMIX` program for the algorithm given in exercise 21 and compare its execution time to Programs S and T.

[334]

37. [*24*] (D. Ferguson) If three computer words (octabytes) are necessary to contain two link fields and an `INFO` field, representation (2) requires $3n$ words of memory for a tree with n nodes. Design a representation scheme for binary trees that uses less space, assuming that *one* `LINK` and an `INFO` field will fit in two computer words.

2.3.2. Binary Tree Representation of Trees

[338]

We shall assume that tree structures for the algebraic formulas with which we will be dealing have nodes of the following form in `MMIX` programs:

RLINK		RTAG
LLINK		
INFO	DIFF	

(10)

Here `RLINK` and `LLINK` have the usual significance, and `RTAG` is 1 for thread links. The `INFO` field and the `DIFF` field share the third octabyte as shown. Instead of storing a `TYPE` field to distinguish different kinds of nodes, we store `DIFF[TYPE(P)]` (see Algorithm D) directly as `DIFF(P)`, thereby avoiding an extra level of indirection. Using object-oriented terminology, the `DIFF` field contains the differentiation *method*; in terms of `MMIX` machine language, it contains the address of the code needed to differentiate the current node. In order to squeeze the address into a single `WYDE`, the address is given relative to `DIFF[0]`, the code used to differentiate a constant. As a consequence, constants have conveniently a `DIFF` value of zero. Constants use the high tetrabyte of the `INFO` field to store the value of the constant, and variables use the `INFO` field to store the variable name padded with zeros to the right; otherwise, the `INFO` field is zero.

[342]

Program D (*Differentiation*). The following `MMIX` subroutine performs Algorithm D. It expects two parameters: Register `y` points to the list head of a tree representing an algebraic formula and register `x` contains the `INFO` and `DIFF` fields of the dependent variable. The return value is a pointer to the list head of a tree representing the analytic derivative of `y` with respect to the variable given by `x`. The order of computations has been rearranged a little, for convenience.

```
:D     GET    rJ,:rJ
       LDOU   p1,y,:LLINK      D1. Initialize. P1 ← LLINK(Y), prepare to find Y$.
1H     SET    p,p1             P ← P1.
2H     LDOU   p1,p,:LLINK      P1 ← LLINK(P).
       BNZ    p1,1B            If P1 ≠ Λ, repeat.
D2     LDWU   diff,p,:DIFF     D2. Differentiate.
       GETA   t,:Const
```

```
        GO      t,t,diff          Jump to the differentiation method.
D3      STOU    p2,p1,:RLINK      D3. Restore link. RLINK(P1) ← P2.
D4      SET     p2,p              D4. Advance to P$. P2 ← P.
        LDOU    p,p,:RLINK        P ← RLINK(P).
        BOD     p,1F              Jump if RTAG(P) = 1;
        STOU    q,p2,:RLINK         otherwise, set RLINK(P2) ← Q.
        JMP     2B                Note that Node(P$) will be terminal.
1H      ANDN    p,p,1             Remove tag from P.
D5      CMP     t,p,y             D5. Done?
        LDOU    p1,p,:LLINK       P1 ← LLINK(P), prepare for step D2.
        LDOU    q1,p1,:RLINK      Q1 ← RLINK(P1).
        BNZ     t,D2              Jump to D2 if P ≠ Y;
        PUSHJ   dy,:Allocate        otherwise, allocate DY.
        STOU    q,dy,:LLINK       LLINK(DY) ← Q.
        STOU    dy,dy,:RLINK      RLINK(DY) ← DY.
        OR      t,dy,1
        STOU    t,q,:RLINK        RLINK(Q) ← DY, RTAG(Q) ← 1.
        PUT     :rJ,rJ
        SET     $0,dy             Return DY.
        POP     1,0               Exit from differentiation subroutine. ▮
```

The next part of the program contains the basic subroutines `Tree1` and `Tree2`. They create nodes for unary and binary operations, respectively. `Tree2` expects three parameters: first `u` and `v`, the pointers to the operands; and then `diff`, the absolute address of the differentiation method of the operation in question. `Tree2` returns a tree that represents the two operands connected by the given operation.

For convenience, `Tree1` uses the same calling convention; the second parameter `v` is, however, ignored.

```
:Tree1    SET     v,u             Set V ← U in the unary case.
          JMP     1F
:Tree2    STOU    v,u,:RLINK      RLINK(U) ← V.
1H        GET     rJ,:rJ
          PUSHJ   r,:Allocate     R ⇐ AVAIL.
          PUT     :rJ,rJ
          STOU    u,r,:LLINK      LLINK(R) ← U.
          GETA    t,:Const
```

```
          SUBU   diff,diff,t        Convert diff to relative address.
          STOU   diff,r,:INFO       INFO(R) ← 0, DIFF(R) ← diff.
          OR     t,r,1              Set tag bit.
          STOU   t,v,:RLINK         RLINK(V) ← R, RTAG(V) ← 1.
          SET    $0,r               Return R.
          POP    1,0                ■
```

Next is the `Copy` subroutine, which appears as exercise 13.

`Allocate` returns a zero-initialized node representing the constant "0"; `Free` puts a node back to free storage.

```
avail       GREG   0
pool        GREG   0
:Allocate   BNZ    avail,1F           AVAIL stack empty?
            SETH   $0,#4000           If so, get 24 bytes
            ADDU   $0,$0,pool           from the Pool_Segment.
            ADDU   pool,pool,24
            JMP    0F
1H          SET    $0,avail           Else, get the next node
            LDOU   avail,avail,:LLINK   from the AVAIL stack.
0H          STCO   0,$0,:RLINK        Zero out the node.
            STCO   0,$0,:LLINK
            STCO   0,$0,:INFO
            POP    1,0
:Free       STOU   avail,$0,:LLINK    Add node to the AVAIL stack.
            SET    avail,$0
            POP    0,0                ■
```

The remaining portion of the program corresponds to the differentiation routines. These routines are written to return control to step D3 after processing a binary operator; otherwise they return to step D4. Note that all named registers (except `t`) have register numbers smaller than register `q`, so that '`PUSHJ q,:Allocate`' will not clobber them.

```
:Const    PUSHJ  q,:Allocate        Q ← "0".
          JMP    D4
:Var      PUSHJ  q,:Allocate        Q ← "0".
          LDOU   t,p,:INFO
          CMP    t,t,x              Is INFO(P) = x?
```

```
         BNZ    t,D4          If not, it's a constant;
         SET    t,1             else Q ← "1".
         STT    t,q,:INFO
         JMP    D4
:Ln      LDOU   t,q,:INFO
         BZ     t,D4          Return to control routine if INFO(Q) = 0.
         SET    q+1,q
         SET    q+3,p1
         PUSHJ  q+2,:Copy
         GETA   q+3,:Div
         PUSHJ  q,:Tree2      Q ← Tree2(Q,Copy(P1),"/").
         JMP    D4
:Neg     LDOU   t,q,:INFO
         BZ     t,D4          Return to control routine if INFO(Q) = 0.
         SET    q+1,q
         GETA   q+3,:Neg
         PUSHJ  q,:Tree1      Q ← Tree1(Q, · ,"−").
         JMP    D4
:Add     LDOU   t,q1,:INFO
         PBNZ   t,1F          Jump unless INFO(Q1) = 0.
         SET    t+1,q1
         PUSHJ  t,:Free       AVAIL ⇐ Q1.
         JMP    D3
1H       LDOU   t,q,:INFO
         PBNZ   t,1F          Jump unless INFO(Q) = 0.
2H       SET    t+1,q
         PUSHJ  t,:Free       AVAIL ⇐ Q.
         SET    q,q1          Q ← Q1
         JMP    D3
1H       GETA   q+3,:Add
3H       SET    q+1,q1
         SET    q+2,q
         PUSHJ  q,:Tree2      Q ← Tree2(Q1,Q,"+").
         JMP    D3
:Sub     LDOU   t,q,:INFO
         BZ     t,2B          If INFO(Q) = 0, then −Q = +Q.
```

```
        GETA    q+3,:Sub       Prepare for Q ← Tree2(Q1,Q,"−").
        LDOU    t,q1,:INFO
        PBNZ    t,3B
        SET     t+1,q1
        PUSHJ   t,:Free        AVAIL ⇐ Q1.
        SET     q+1,q
        GETA    q+3,:Neg
        PUSHJ   q,:Tree1       Q ← Tree1(Q, · ,"−").
        JMP     D3
:Mul    LDOU    t,q1,:INFO
        BZ      t,1F           Jump if INFO(Q1) = 0.
        SET     t+1,q1
        SET     t+3,p2
        PUSHJ   t+2,:Copy
        PUSHJ   t,:Mult
        SET     q1,t           Q1 ← Mult(Q1,Copy(P2)).
1H      LDOU    t,q,:INFO
        BZ      t,:Add         Jump if INFO(Q) = 0.
        SET     q+2,p1
        PUSHJ   q+1,:Copy
        SET     q+2,q
        PUSHJ   q,:Mult        Q ← Mult(Copy(P1),Q).
        JMP     :Add           ▮
```

`Mult` expects two parameters `u` and `v`; it returns an optimized representation of `u` × `v`.

```
:Mult   GET     rJ,:rJ
        SETMH   info,1         The constant "1" has INFO = 1 and DIFF = 0.
        LDO     t,u,:INFO
        CMP     t,info,t       Test if U is the constant "1".
        BZ      t,1F           Jump if so.
        LDO     t,v,:INFO      Otherwise,
        CMP     t,info,t         test if V is the constant "1",
        GETA    v+1,:Mul         prepare third parameter,
        BNZ     t,:Tree2         and if not so, return Tree2(U,V,"×");
        SET     t+1,v            else, pass V to Free.
        JMP     2F
```

```
162  1H   SET    t+1,u1       Pass U to Free.
163       SET    u,v          U ← V.
164  2H   PUSHJ  t,:Free      Free one parameter
165       PUT    :rJ,rJ         and return U.
166       POP    1,0          ▮
```

The last two routines Div and Pwr are similar and they have been left as exercises (see exercises 15 and 16).

EXERCISES

[347]

13. [*26*] Write an MMIX program for the Copy subroutine. [*Hint:* Adapt Algorithm 2.3.1C to the case of a right-threaded binary tree, with suitable initial conditions.]

14. [*M21*] How long does it take the program of exercise 13 to copy a tree with n nodes?

15. [*23*] Write an MMIX program for the Div routine, corresponding to DIFF[7] as specified in the text. (This program should be added to the program in the text after line 166.)

16. [*24*] Write an MMIX program for the Pwr routine, corresponding to DIFF[8] as specified in exercise 12. (This program should be added to the program in the text after the solution to exercise 15.)

2.3.3. Other Representations of Trees

[357]

Nodes have six fields, which in the case of MMIX might fit in three octabytes. A compact representation may use the fact that either the VALUE field is used to represent a constant or the NAME and DOWN fields are used to represent a polynomial g_j. So two kinds of nodes are possible:

RIGHT	LEFT
UP	EXP
VALUE	

or

RIGHT	LEFT
UP	EXP
NAME	DOWN

. (17)

Here RIGHT, LEFT, UP, and DOWN are relative links; EXP is an integer representing an exponent; VALUE contains a 64-bit floating point constant; and the NAME field contains the variable name. To distinguish between the two types of nodes, the low-order bit in a link field can be used. There are two essentially different choices: Either one of the link fields within the node is used or all the links that point to the node are marked. The first choice makes it easy to change a node from one type to the other (as is possible in step A9); the second choice makes searching for a constant (as in step A1) simpler.

2.3.5. Lists and Garbage Collection

[411]

1) . . . Therefore each node generally contains tag bits that tell what kind of information the node represents. The tag bits can occupy a separate `TYPE` field that can also be used to distinguish between various types of atoms (for example, between alphabetic, integer, or floating point quantities, for use when manipulating or displaying the data), or the tag bits can be placed in the low-order bits of the link fields, where they are ignored when using link fields as addresses of other `OCTA`-aligned nodes.

2) The format of nodes for general List manipulation with the `MMIX` computer might be designed in many different ways. For example, consider the following two ways.

a) Compact one-word format, assuming that all `INFO` appears in atoms:

REF	RLINK	HMA

(9)

This format uses 32-bit relative addresses to nodes from a common storage pool; the short addresses imply a limit of 4GByte on its maximum size. `RLINK` is such a pointer for straight or circular linkage as in (8). Limiting addresses to `OCTA`-aligned data, the three least significant bits `H`, `M`, and `A` are freely available as tag bits.

The `M` bit, normally zero, is used as a mark bit in garbage collection (see below).

The `A` bit indicates an atomic node. If `A` = 1, all the bits of the node, except `A` and `M`, can be used to represent the atom. If `A` = 0, the `H` bit can be used to distinguish between List heads and List elements. If `H` = 1, the node is a List head, and `REF` is a reference count (see below); otherwise, `REF` points to the List head of the sub-List in question.

b) Simple three-word format: A straightforward modification of (9) yields three-word nodes using absolute addresses. For example:

RLINK	HMA
LLINK	
INFO	

(10)

The H, M, and A bits are as in (9). RLINK and LLINK are the usual pointers for double linkage as in (8). INFO is a full word of information associated with this node; for a header node this may include a reference count, a running pointer to the interior of the List to facilitate linear traversal, an alphabetic name, and so on. If H $= 0$, this field contains the DLINK.

[420]

Of all the marking algorithms we have discussed, only Algorithm D is directly applicable if atomic nodes must use all the node bits except a single bit, the mark bit. For example, Lists could be represented as in (9) using only the least significant bit for M. The other algorithms all test whether or not a given node P is an atom; they will need the A bit. However, each of the other algorithms can be modified so that they will work when atomic data is distinguished from pointer data in the word that links to it instead of by looking at the word itself. . . . The adaptation of Algorithm E is almost as simple; both ALINK and BLINK can even accommodate two more tag bits in addition to the mark bit.

EXERCISES

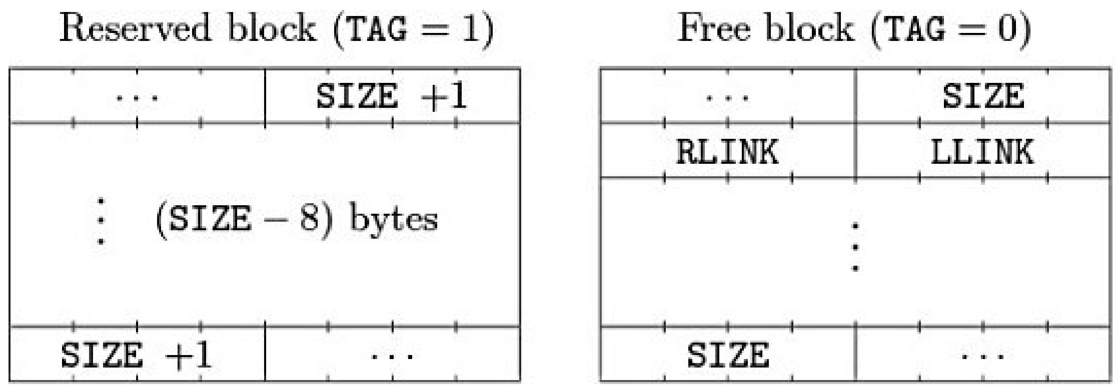
[422]

4. [*28*] Write an MMIX program for Algorithm E, assuming that the nodes are represented as two octabytes, with ALINK the first octabyte and BLINK the second octabyte. The least significant bits of ALINK and BLINK can be used for MARK and ATOM. Also determine the execution time of your program in terms of relevant parameters.

2.5. DYNAMIC STORAGE ALLOCATION

[440]

The method we will describe assumes that each block has the following form:



(7)

Note that the SIZE − 8 bytes reserved for use by an application are OCTA-aligned, while the node itself starts and ends with a SIZE field that is only TETRA-aligned.

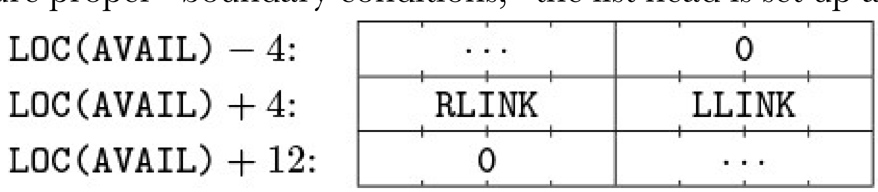
The idea in the following algorithm is to maintain a doubly linked AVAIL list, so that entries may conveniently be deleted from random parts of the list. The TAG bit at either end of a block—the least significant bit in the SIZE field—can be used to control the collapsing process, since we can tell easily whether or not both adjacent blocks are available.

To save space, links are stored as relative addresses in a TETRA. As base address, we use LOC(AVAIL), the address of the list head, which conveniently makes the relative address of the list head zero.

Unfortunately, a notation such as 'LINK(P + 1)' does not work well in the world of MMIX, where addresses refer to bytes and links are stored as tetrabytes or octabytes. Therefore, we use the familiar RLINK and LLINK instead of 'LINK(P)' and 'LINK(P + 1)', but we do not rephrase Algorithm C. Double linking is achieved in a familiar way—by letting RLINK point to the next free block in the list, and letting LLINK point back to the previous block; thus, if P is the address of an available block, we always have

$$\texttt{LLINK(RLINK(P))} = \texttt{P} = \texttt{RLINK(LLINK(P))}. \qquad (8)$$

To ensure proper "boundary conditions," the list head is set up as follows:



(9)

Here RLINK points to the first block and LLINK to the last block in the available space list. Further, a tagged tetrabyte should occur before and after the memory area used to limit the activities of Algorithm C.

[449]

Here are the approximate results:

Here are the approximate results.

	Time for reservation	Time for liberation
Boundary tag system:	$24 + 5A$	18, 22, 27, or 28
Buddy system:	$26 + 26R$	$36.5 + 24S$

. . .

This shows that both methods are quite fast, with the buddy system reservation faster and liberation slower by a factor of approximately 1.5 in MMIX's case. Remember that the buddy system requires about 44 percent more space when block sizes are not constrained to be powers of 2.

A corresponding time estimate for the garbage collection and compacting algorithm of exercise 33 is about 98υ to locate a free node, assuming that garbage collection occurs when the memory is approximately half full, and assuming that nodes have an average length of 5 octabytes with two links per node.

EXERCISES

[453]

4. [*22*] Write an MMIX program for Algorithm A, paying special attention to making the inner loop fast. Assume that the SIZE and the LINK fields are stored in the high and low TETRA of an octabyte. To make links fit in a tetrabyte, use addresses relative to the base address in the global register base. If successful, return an *absolute* address. Use $\Lambda = -1$ if dealing with relative addresses, but for absolute addresses (the return value) use $\Lambda = 0$.

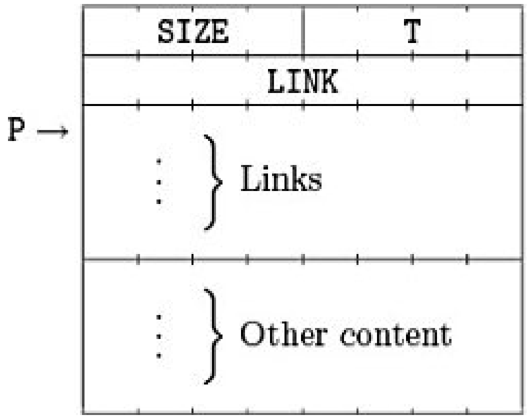
13. [*21*] Write an MMIX subroutine using the algorithm of exercise 12. Assume that the only parameter N is the size of the requested memory in bytes and that the return value is an OCTA-aligned absolute address where these N bytes are available. In case of overflow, the return value should be zero.

16. [*24*] Write an MMIX subroutine for Algorithm C that complements the program of exercise 13, incorporating the ideas of exercise 15.

27. [*24*] Write an MMIX program for Algorithm R, and determine its running time.

28. [*25*] Write an MMIX program for Algorithm S, and determine its running time.

33. [*28*] (*Garbage collection and compacting.*) Assume a storage pool of nodes of varying sizes, each one having the following form:



SIZE(P) = number of bytes in NODE(P);
T(P) = number of bytes used for links, T(P) < SIZE(P);
LINK(P) = special link field for use only during garbage collection.

The node at address `P` starts with two octabytes *preceding* the address `P`; these contain special data for use during garbage collection only. The node immediately following `NODE(P)` in memory is the node at address `P` + `SIZE(P)`. The nodes populate a memory area starting at `BASE` − 16 up to `AVAIL` − 16. Assume that the only fields in `NODE(P)` that are used as links to other nodes are the octabytes `LINK(P)` + 8, `LINK(P)` + 16, . . . , `LINK(P)` + `T(P)`, and that each of these link fields is either Λ or the absolute address of another node. Finally, assume that there is one further link variable in the program, called `USE`, and it points to one of the nodes.

34. [*29*] Write an `MMIX` program for the algorithm of exercise 33, and determine its running time.

CHAPTER THREE

RANDOM NUMBERS

3.2.1.1. Choice of modulus

[12]

Consider `MMIX` as an example. We can compute $y \bmod m$ by putting y and m in registers and dividing y by m using the instruction '`DIV t,y,m`'; $y \bmod m$ will then appear in register rR. But division is a comparatively slow operation, and it can be avoided if we take m to be a value that is especially convenient, such as the *word size* of our computer.

Let w be the computer's word size, namely 2^e on an e-bit binary computer. The result of an addition and multiplication is usually given modulo w. Thus, the following program computes the quantity $(aX + c) \bmod w$ efficiently:

```
MULU   x,x,a    X ← aX mod w.                 (1)
ADDU   x,x,c    X ← (X + c) mod w.  ▮
```

The result appears in register `x`. The code uses arithmetic on unsigned numbers, which never causes overflow. If c is less than 2^{16}, the instruction '`ADDU x,x,c`' can be replaced by '`INCL x,c`', using an immediate value `c` instead of a register `c`. The same is possible for the constant a; however, satisfactory values for a are typically large and the `MULU` instruction allows only a one-byte immediate constant.

A clever technique that is less commonly known can be used to perform computations modulo $w + 1$. For reasons to be explained later, we will generally want $c = 0$ when $m = w + 1$, so we merely need to compute $(aX) \bmod (w + 1)$. With $w = 2^{64}$, the following program does this:

```
MULU   r,x,a;  GET q,rH    Compute q, r with aX = qw + r.
SUBU   x,r,q               X ← r − q mod w.
CMPU   t,r,q                                               (2)
ZSN    t,t,1               Set t ← [r < q].
ADDU   x,x,t               X ← X + t mod w.  ▮
```

The register `x` now contains the value $(aX) \bmod (w+1)$. Of course, this value might lie anywhere between 0 and w, inclusive, so the reader may legitimately wonder how we can represent so many values in one register! (The register obviously cannot hold a number larger than $w - 1$.) The answer is that X will be 0 and t will be 1 after program (2) if and only if the result equals w. We could represent w by 0, since (2) will not normally be used when $X = 0$; but it is most convenient simply to reject the value w (and 0) if it appears in the congruential sequence modulo $w + 1$. We do this by appending the instructions '`NEGU t,1,a; CSZ x,x,t`'.

To prove that code (2) actually does determine $(aX) \bmod (w + 1)$, note that in line 02 we are subtracting the lower half of the product from the upper half; and if $aX = qw + r$, with $0 \le r < w$, we will have the quantity $r - q$ in register `x` after line 02. Now

$$aX = q(w + 1) + (r - q),$$

and we have $-w < r - q < w$ since $q < w$; hence $(aX) \bmod (w + 1)$ equals either $r - q$ or $r - q + (w + 1)$, depending on whether $r - q \ge 0$ or $r - q < 0$.

EXERCISES

[15]

1. [*M12*] In exercise 3.2.1–3 we concluded that the best congruential generators will have a multiplier a relatively prime to m. Show that in such a case there is a constant c' such that $(aX + c) \bmod m = a(X + c') \bmod m$.

2. [*16*] Write an `MMIX` subroutine having the following characteristics:

Calling sequence:	`PUSHJ t,Random`
Entry conditions:	The global registers `x` $\equiv X$, `a` $\equiv a$, and `c` $\equiv c$ are initialized.
Exit conditions:	Set $X \leftarrow (aX + c) \bmod 2^{64}$ and return X.

(Thus a call on this subroutine will produce the next random number of a linear congruential sequence.)

5. [*20*] Given that m is less than the word size, that x and y are nonnegative integers less than m, and assuming that the values x, y, and m are already loaded into registers, show that the difference $(x - y) \bmod m$ may be computed in just four `MMIX` instructions without requiring any division. What is the best code for the sum $(x + y) \bmod m$? What is the best code if m is less than 2^{e-1}?

8. [*20*] Write an `MMIX` program analogous to (2) that computes $(aX) \bmod (w - 1)$. The values 0 and $w - 1$ are to be treated as equivalent in the input and output of your program.

3.2.1.3. Potency

[24]

For example, suppose that we choose $a = 2^k + 1$, where $k \geq 2$ is a constant. With a temporary register `t`, the code

$$\texttt{SLU t,x,}k\texttt{; ADDU x,t,x; ADDU x,x,1} \qquad (3)$$

can be used in place of the instructions given in Section 3.2.1.1, and the execution time decreases from 11υ to 3υ. Even faster code is possible for $k = 2$, 3, or 4. For example, the code '`16ADDU x,x,x; ADDU x,x,1`' has a running time of only 2υ.

EXERCISES

[25]

1. [*M10*] Show that, for all $k \geq 2$, the code in (3) yields a random number generator of maximum period.

2. [*10*] What is the potency of the generator represented by the `MMIX` code (3)?

3.2.2. Other Methods

[28]

This algorithm in `MMIX` is simply the following:

Program A (*Additive number generator*). Using global registers `j` $\equiv 8j$, `k` $\equiv 8k$, and `y` $\equiv$ `LOC(`$Y[1]$`)` $-$ 8, the following `MMIX` code is a step-by-step implementation of Algorithm A.

```
:Random  LDOU  $0,y,j     A1. Add. $0 ← Y[j].
         LDOU  t,y,k      t ← Y[k].
         ADDU  $0,$0,t    $0 ← Y[j] + Y[k].
         STOU  $0,y,k     Y[k] ← Y[j] + Y[k].
         SUB   j,j,8      A2. Advance. j ← j − 1.
         SUB   k,k,8      k ← k − 1.
         SET   t,55*8
         CSNP  j,j,t      If j ≤ 0, set j ← 55.
         CSNP  k,k,t      If k ≤ 0, set k ← 55.
         POP   1,0        Return $0. ▮
```

One disadvantage of the code above is its use of three possibly precious global registers. An improved version of this program is discussed in exercise 25.

There is a simple way to generate a highly random bit sequence on a binary computer, manipulating k-bit words: Start with an arbitrary binary word X in register `x`. To get the next random bit of the sequence, do the following operations, shown in `MMIX`'s language (see exercise 16):

```
ZSN   t,x,a        Adjust by a if the high bit of x is 1, else by zero.
SLU   x,x,1        Shift left one bit.                                    (10)
XOR   x,x,t        Apply the adjustment with "exclusive or."  ▮
```

The value of the global register `a` is the k-bit binary constant `a` $= (a_1 \ldots a_k)_2$, shifted left by $64 - k$ bits, where $x^k - a_1x^{k-1} - \cdots - a_k$ is a primitive polynomial modulo 2 as above. After the code (10) has been executed, the next bit of the generated sequence may be taken as the kth bit from the left of register `x`. Alternatively, we could consistently use the most significant bit (the sign bit) of `x`; that gives the same sequence, but each bit is seen one step earlier.

. . .

On `MMIX` we may implement Algorithm B by taking $k = 256$, obtaining the following simple generation scheme once the initialization has been done:

```
SRU   j,y,53                 j ← ⌊256Y/w⌋, j ← 8j + {0, . . . , 7}.
MULU  x,x,a; ADD x,x,c       X_{n+1} ← (aX_n + c) mod w.              (14)
LDOU  y,V,j                  Y ← V[j].
STOU  x,V,j                  V[j] ← X_{n+1}.  ▮
```

The output appears in register `y`. Notice that Algorithm B requires only $3\upsilon + 2\mu$ of additional overhead per generated number.

EXERCISES

7. [*20*] Show that a complete sequence of length 2^e (that is, a sequence in which each of the 2^e possible sets of e consecutive sign bits occurs just once in the period) may be obtained if program (10) is changed to the following:

```
ZSN  t,x,a
SLU  x,x,1
ZSZ  s,x,a
XOR  x,x,t
```

```
XOR   x,x,s
```

[39]

25. [*26*] Discuss an alternative to Program A: a subroutine `Random55` that changes all 55 entries of the Y table every 55th time a random number is required. Try to get by with just one global register.

3.4.1. Numerical Distributions

[119]

In general, to get a random integer X between 0 and $k-1$, we can *multiply* by k, and let $X = \lfloor kU \rfloor$. On `MMIX`, we would write

```
MULU    t,k,u       (rH, t) ← kU
GET     x,rH        X ← ⌊kU/m⌋                        (1)
```

and after these two instructions have been executed the desired integer will appear in register `x`. If a random number between 1 and k is desired, we add one to this result. (The instruction '`INCL x,1`' would follow (1).)

EXERCISES

[138]

3. [*14*] Discuss treating U as an integer and computing its *remainder* mod k to get a random integer between 0 and $k-1$, instead of multiplying as suggested in the text. Thus (1) would be changed to

```
DIV   t,u,k          t ← ⌊U/k⌋
GET   x,rR           X ← U mod k
```

with the result again appearing in register `x`. The new method might be especially tempting if $k = 2^i$ (for a small constant i) because

```
AND   x,u,(2^i - 1)        X ← U mod 2^i
```

will do the job in a single `MMIX` cycle. Is this a good method?

3.6. SUMMARY

EXERCISES

[189]

1. [*21*] Write an `MMIX` subroutine `RandInt` using method (1) according to the following specification:

Calling sequence:	`PUSHJ t,RandInt`
Entry conditions:	Global register `x` $\equiv X$ initialized.
	`$0` $\equiv k$, a positive integer.
Return value:	A random integer Y, $1 \le Y \le k$, with each integer about equally probable.
Exit conditions:	Global register `x` modified.

CHAPTER FOUR

ARITHMETIC

4.1. POSITIONAL NUMBER SYSTEMS

[203]

The `MIX` computer, as used in Chapter 4 of *The Art of Computer Programming*, deals only with signed magnitude arithmetic, whereas the `MMIX` computer, used here, deals only with two's complement binary arithmetic. However, alternative procedures for complement notations are discussed in Chapter 4 when it is important to do so.

EXERCISES

[209]

4. [*20*] Assume that we have an `MMIX` program in which register `a` contains a nonnegative number for which the radix point lies between bytes 3 and 4, while register `b` contains a nonnegative number whose radix point lies between bytes 5 and 6. (The leftmost byte is number 1.) Where will the radix point be in registers `x`, rH, and rR after the following instructions (assuming that the instructions do not raise an arithmetic exception)?

(a) `MUL x,a,b`

(b) `DIV x,a,b`

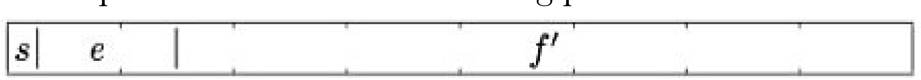
(c) `MULU x,a,b`

(d) `PUT rD,0; DIVU x,a,b`

4.2.1. Single-Precision Calculations

[215]

The `MMIX` computer assumes that its floating point numbers have the form



(4)

Here we have base $b = 2$, excess $q = 1023$, floating point notation with $p = 53$ bits of precision. The sign bit is stored in the leftmost bit; it is 1 for negative numbers and 0 otherwise. The exponent e is stored in the next 11 bits; it is an integer in the range $0 < e < 2047$. The fraction part f is stored as a 52-bit binary value f' in the range $0 \le f' < 2^{52}$ with $f = 1 + f'/2^{52}$. Since $b = 2$, the most significant digit of a normalized fraction part is always 1, and there is no need to store this bit. With this hidden bit added to the left of f', the precision is 53.

B. Normalized calculations. The floating point arithmetic of `MMIX` follows IEEE/ANSI Standard 754, which is implemented by most modern computers. Following this standard and contrary to the definitions used in the current edition of *The Art of Computer Programming*, Volume 2, the radix point is placed just between the hidden bit and the stored part f' of f. A floating point number (s, e, f) is *normalized* if $0 < e < 2047$ and the most significant digit of the representation of f is nonzero, so that

$$1 \le f < 2. \tag{5}$$

The floating point number represents ± 0.0 if $f = e = 0$.

[218]

The following `MMIX` subroutines, for addition and subtraction of numbers having the form (4), show how Algorithms A and N can be expressed as computer programs. The subroutines below do not handle all the complications of the IEEE Standard 754. They are designed to take two parameters u and v and return a normalized result w. A simple `JMP Error` is used whenever this is not possible.

Program A (*Addition, subtraction, and normalization*). The following program is an implementation of Algorithm A, and it is also designed so that the trailing implementation of Algorithm N can be used by other programs that appear later in this section.

The variables are named to match Algorithms A and N. Where the variable names differ in Algorithms A and N, we gave preference to Algorithm N. So instead of f_w we use f in Algorithm A, and similarly we use e instead of e_w. The registers `s`, `su`, and `sv` are used for the sign bits of w, u, and v. To ensure proper rounding, the next lower 64 bits of f are stored in register `fl`. The register `carry` is used as a shuttle between `f` and `fl`. Another register, `d`, is needed in step A4 and A5 to hold the difference $e_u - e_v$.

`:Fsub`	`SETH`	`t,#8000; XOR v,v,t`	Change sign of operand.
`:Fadd`	`SLU`	`eu,u,1; SLU ev,v,1`	Remove sign bit.
	`CMPU`	`t,eu,ev`	*A2. Assume e_u dominates e_v.*
	`BNN`	`t,A1`	Jump if $(e_u, f_u) \ge (e_v, f_v)$;
	`SET`	`t,u; SET u,v; SET v,t`	else swap u with v
	`SLU`	`eu,u,1; SLU ev,v,1`	and remove sign bits again.
`A1`	`SRU`	`eu,eu,53; SRU ev,ev,53`	*A1. Unpack.*

	`SETH`	`t,#FFF0`	Get sign and exponent mask.
	`ANDN`	`fu,u,t; ANDN fv,v,t`	Remove sign and exponent.
	`INCH`	`fu,#10; INCH fv,#10`	Add hidden bit.
	`SRU`	`su,u,63; SRU sv,v,63`	Get sign bit.
	`SET`	`e,eu; SET s,su`	*A3. Set $e_w \leftarrow e_u$.*
	`SUB`	`d,eu,ev`	Step A4 unnecessary.
`A5`	`NEG`	`t,64,d`	*A5. Scale right.*
	`SLU`	`fl,fv,t`	Shift (f_v, f_l) to the right
	`SRU`	`fv,fv,d`	$e_u - e_v$ places.
	`CMP`	`t,su,sv; BNZ t,0F`	Signs s_u and s_v differ.
	`ADDU`	`f,fu,fv`	*A6. Add.*
	`JMP`	`:Normalize`	
`0H`	`NEGU`	`fl,fl; ZSNZ carry,fl,1`	*A6. Subtract.*
	`SUBU`	`f,fu,fv`	
	`SUBU`	`f,f,carry`	
`:Normalize`	`OR`	`t,f,fl; BZ t,:Zero`	Assume $u + v \neq 0$.
	`SRU`	`t,f,53`	*N1. Test f.*
	`BP`	`t,N4`	If $f \geq 2$, scale right.
`N2`	`SRU`	`t,f,52; BP t,N5`	*N2. Is f normalized?*
	`SRU`	`carry,fl,63`	*N3. Scale left.*
	`SLU`	`fl,fl,1`	
	`SLU`	`f,f,1`	
	`ADDU`	`f,f,carry`	
	`SUB`	`e,e,1`	
	`JMP`	`N2`	
`N4`	`SLU`	`carry,f,63`	*N4. Scale right.*
	`SRU`	`f,f,1`	
	`SRU`	`fl,fl,1`	
	`ADDU`	`fl,fl,carry`	
	`ADD`	`e,e,1`	
`N5`	`SETH`	`t,#8000`	*N5. Round.*
	`CMPU`	`t,fl,t`	Compare f_l to $\frac{1}{2}$.
	`CSOD`	`carry,f,1`	f is odd. Round up if $f_l \geq \frac{1}{2}$.
	`CSEV`	`carry,f,t`	f is even. Round up if $f_l > \frac{1}{2}$.
	`ZSNN`	`carry,t,carry`	

```
          ZSNN   carry,t,carry           Round down if f_l < 1/2.
          ADDU   f,f,carry
          SET    fl,0
          SRU    t,f,53; BP t,N4         Rounding overflow.
          SET    t,#7FE; CMP t,e,t       N6. Check e.
          BP     t,:Error                Overflow.
          BNP    e,:Error                Underflow.
          SLU    w,s,63                  N7. Pack.
          SLU    t,e,52; OR w,w,t
          ANDNH  f,#FFF0                 Remove hidden bit.
          OR     $0,w,f
          POP    1,0                     Return w.
:Zero     POP    0,0                     Return zero. ▮
```

Using a second register `fl` for the lower 64 bits of fraction f and extending adding, subtracting, and shifting to it is not strictly necessary. Exercise 5 shows how to get by with $p + 2 = 55$ digits, which fit nicely into one of MMIX's registers. This optimization, however, will make the code neither significantly shorter nor faster; there are just too many special cases to consider. On the other hand, MMIX is well suited to do multi-precision arithmetic.

[220]

The following MMIX subroutines, intended to be used in connection with Program A, illustrate the machine considerations that arise in Algorithm M.

Program M (*Floating point multiplication and division*).

```
:Fmul  SLU   eu,u,1; SRU eu,eu,53    M1. Unpack.
       SLU   ev,v,1; SRU ev,ev,53
       SETH  t,#FFF0                 Get sign and exponent mask.
       ANDN  fu,u,t; ANDN fv,v,t     Remove sign and exponent bits.
       INCH  fu,#10; INCH fv,#10     Add hidden bit.
       XOR   s,u,v; SRU s,s,63       s ← s_u × s_v.
       SLU   fv,fv,6; SLU fu,fu,6    M2. Operate.
       MULU  fl,fu,fv; GET f,:rH     (f, f_l) ← 2^(52+6) f_u · 2^(52+6) f_v = 2^(52+64) f_u f_v.
       ADD   e,eu,ev
       SET   t,1023; SUB e,e,t       e ← e_u + e_v − q.
       JMP   :Normalize              M3. Normalize.
:Fdiv  SLU   eu,u,1; SRU eu,eu,53    M1. Unpack.
```

```
13    SLU   ev,v,1; SRU ev,ev,53
14    SETH  t,#FFF0                  Get sign and exponent mask.
15    ANDN  fu,u,t; ANDN fv,v,t      Remove sign and exponent bits.
16    INCH  fu,#10; INCH fv,#10      Add hidden bit.
17    XOR   s,u,v; SRU s,s,63        s ← s_u × s_v.
18    SLU   fv,fv,11                 M2. Operate. f_v ← 2^11 f_v.
19    PUT   :rD,fu; SET t,0
20    DIVU  f,t,fv                   (f, f_l) ← 2^(52+64) f_u/(2^(52+11) f_v) = 2^53 f_u/f_v.
21    GET   t,:rR; PUT :rD,t
22    SET   t,0; DIVU fl,t,fv
23    SUB   e,eu,ev
24    INCL  e,1023-1                 e ← e_u − e_v + q − 1.
25    JMP   :Normalize               M3. Normalize. ▮
```

The most noteworthy feature of this program is the use of double-precision multiplication in line 08 and division in lines 19–22 in order to ensure enough accuracy to round the answer.

The numbers f_u and f_v are represented by the unsigned integers $2^{52}f_u$ and $2^{52}f_v$, respectively. Using the `MULU` directly would yield $2^{52+52}f_uf_v$; applying an extra factor of 2^6 to both f_u and f_v prior to the multiplication yields $2^{52+64}f_uf_v$, which moves the radix point in rH just to the right place after bit 52. Applying an extra factor of 2^{12} to only one operand would cause overflow.

The division works differently since extra factors applied to f_u and f_v shift the radix point of the result in opposite directions. Shifting f_u (the high 64 bits of the dividend) right would be possible if the bits are shifted into the low 64 bits of the dividend. Fortunately, the limit for shifting f_v left is 11 bits, which is just what we need. Dividing by $2^{11}f_v$ gives $2^{1+52+64}f_u/f_v$. With the imagined radix point just left of bit 52 in $(f,\ f_l)$, we have $(f,\ f_l) \leftarrow 2f_u/fv$. We compensate for the extra factor 2 by reducing e by 1. If f_u and f_v are normalized, we have $1 \le f_u < 2$ and $1 \le f_v < 2$ so that $1 \le 2f_u/f_v < 4$; step N4 of the normalization will then adjust f if needed.

We occasionally need to convert values between fixed and floating point representations. A "fix-to-float" routine is easily obtained with the help of the normalization algorithm above; for example, in `MMIX`, the following subroutine converts a nonzero integer u to floating point form:

```
:Flot  ZSN   s,u,1                          Set sign.
       SET   f,0; NEG fl,u; CSNN fl,u,u     (f, fl) ← |u| /2^64.
       SET   e,64+52+1023                   Set raw exponent.              (10)
       JMP   :Normalize                     Normalize, round, and exit. ▮
```

A "float-to-fix" subroutine is the subject of exercise 14.

[223]

The MMIX computer, which is being used as an example of a "typical" machine in this supplement, has a full set of floating point instructions conforming to IEEE/ANSI Standard 754.

EXERCISES

[228]

14. [*25*] Write an MMIX subroutine, to be used in connection with the other subroutines in this section, that takes as a parameter a normalized floating point number and returns the nearest signed 64 bit two's complement integer (or determines that the number is too large in absolute value to make such a conversion possible).

15. [*28*] Write an MMIX subroutine, to be used in connection with the other subroutines in this section, that takes a nonzero normalized floating point number u as a parameter and returns u mod 1, namely $u - \lfloor u \rfloor$ rounded to the nearest floating point number. Notice that when u is a very small negative number, u mod 1 should be rounded so that the result is unity (even though u mod 1 has been defined to be always *less* than unity, as a real number).

19. [*24*] What is the running time for the Fadd subroutine in Program A, in terms of relevant characteristics of the data? What is the maximum running time, over all inputs that do not cause exponent overflow or underflow?

20. [*28*] *New*: Given a nonzero octabyte in register f, find a fast way to compute the number of its leading zero bits and use the result to eliminate the loop in steps N2 and N3 of Algorithm N. How will this change affect the average running time?

21. [*40*] *New*: Imagine a low-cost version of MMIX with no hardware support for floating point numbers (used in the CEO's office, where floating point calculations are routinely delegated to the research department). In such an MMIX

CPU, floating point instructions will trap with the operands in registers rYY and rZZ. The operating system should then compute the result, store it back to register rZZ, and set exception flags in the upper half of rXX in preparation for a final `RESUME 1`. Write a subroutine library, emulating the standard `MMIX` floating point hardware, to be used in such an operating system.

4.2.2. Accuracy of Floating Point Arithmetic

EXERCISES

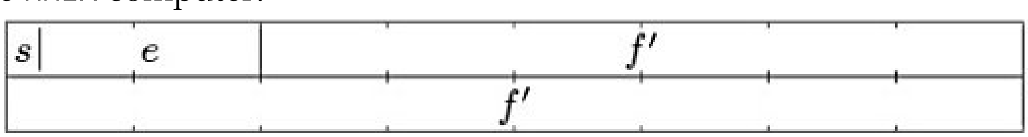
[244]

17. [*28*] Assume that `MMIX` needs to emulate its `FCMPE` (floating compare with respect to epsilon) instruction in software. Write an `MMIX` subroutine, `Fcmpe`, that compares two nonzero normalized floating point numbers u and v with respect to a positive normalized floating point number ϵ stored in register rE. Under the conditions just stated, the subroutine should be equivalent to '`Fcmpe FCMPE $0,$0,$1; POP 1,0`'.

4.2.3. Double-Precision Calculations

[246]

Double precision is quite frequently desired not only to extend the precision of the fraction parts of floating point numbers, but also to increase the range of the exponent part. The IEEE/ANSI standard specifies a lower bound on the precision and a minimum exponent range only for what it calls "extended precision." It requires $p \geq 64$ and $e_{\min} \leq -16382$ and $e_{\max} > 16382$. One way to satisfy these requirements could be to take one `OCTA` for the fraction part and another `OCTA` to provide very generous room for the sign and exponent. A more common compromise between precision and exponent range is to use 15 bits for the exponent, just enough to satisfy the range requirement. With one bit for the sign, that leaves 112 bits for the fraction part. Thus we shall deal in this section with the following 128-bit format for double-precision floating point numbers in the `MMIX` computer:



. (1)

Here two bytes are used for the sign bit and the exponent and 14 bytes for the fraction part. We have base $b = 2$, excess $q = 2^{14} - 1 = 16383$, and because of the hidden bit added to the left of f', a precision of $p = 113$.

For a double-precision floating point number u, we will use the notation s_u for the sign field and e_u for the exponent field of u as before; u_m is used to denote the most significant fraction part from the first octabyte with the radix point just after the hidden bit, and u_l is used to denote the least significant fraction part stored in the second octabyte with the radix point just to the left of its 64 bits. With that notation and $\epsilon = 2^{-48}$, we can write $f = 1 + f' = u_m + \epsilon\, u_l$. To do computations on u_m and u_l, the programs that follow will use registers named `um` and `ul` to perform unsigned integer arithmetic on the values $2^{48}u_m$ and $2^{64}u_l$, respectively.

. . .

Program A (*Double-precision addition*). The subroutine `DFadd` adds a double-precision floating point number v, having the form (1), in registers `vm` and `vl` to a double-precision floating point number u in registers `um` and `ul`, storing the answer w in registers `wm` and `wl`. The subroutine `DFsub` subtracts v from u under the same conventions.

Both input operands are assumed to be nonzero and normalized; the answer is normalized. The last portion of this program is a double-precision normalization procedure that is used by other subroutines of this section. Step 5 of Algorithm N is omitted; exercise 5 shows how to get better rounding.

```
:DFsub   SETH   t,#8000; XOR vm,vm,t          Change sign of operand.
:DFadd   SLU    eu,um,1; SLU ev,vm,1          Remove sign bit.
         CMPU   t,eu,ev                       A2. Assume
         BP     t,A1                              e_u dominates e_v.
         PBN    t,0F
         CMPU   t,ul,vl; BNN t,A1             If (e_u, u_m, u_l) < (e_v, v_m, v_l),
0H       SET    t,um; SET um,vm; SET vm,t       swap u with v
         SET    t,ul; SET ul,vl; SET vl,t       and
         SLU    eu,um,1; SLU ev,vm,1            remove sign bit again.
A1       SRU    eu,eu,49; SRU ev,ev,49        A1. Unpack.
         SRU    su,um,63; SRU sv,vm,63        Get sign bit.
         ANDNH  um,#FFFF; ANDNH vm,#FFFF      Remove s and e bits.
         ORH    um,#0001; ORH vm,#0001        Add hidden bit.
```

	SET	e,eu; SET s,su	*A3. Set $e_w \leftarrow e_u$.*
	SUB	d,eu,ev	*A4. Test $e_u - e_v$.*
	CMP	t,d,113+2; PBN t,A5	$e_u - e_v \geq p + 2$?
	SET	wm,um; SET wl,ul	$w \leftarrow u$.
	JMP	:DNormalize	
A5	CMP	t,d,64; PBN t,0F	*A5. Scale right.*
	SET	vl,vm; SET vm,0	Scale right by 64 bits.
	SUB	d,d,64	
0H	NEG	t,64,d	
	SRU	vl,vl,d	
	SLU	carry,vm,t; OR vl,vl,carry	Shift (v_m, v_l) right by
	SRU	vm,vm,d	$e_u - e_v$ places.
	CMP	t,su,sv; BNZ t,0F	Signs s_u and s_v differ.
	ADDU	wl,ul,vl	*A6. Add.*
	CMPU	t,wl,ul; ZSN carry,t,1	
	ADDU	wm,um,vm	
	ADDU	wm,wm,carry	
	JMP	:DNormalize	
0H	SUBU	wl,ul,vl	*A6. Subtract.*
	CMPU	t,wl,ul; ZSP carry,t,1	
	SUBU	wm,um,vm	
	SUBU	wm,wm,carry	
:DNormalize	SRU	t,wm,49	*N1. Test f.*
	BOD	t,N4	If $w \geq 2$, scale right.
	OR	t,wm,wl; BZ t,:Zero	
N2	SRU	t,wm,48; PBOD t,6F	*N2. Is w normalized?*
	ZSN	carry,wl,1; SLU wl,wl,1	*N3. Scale left.*
	SLU	wm,wm,1	
	ADDU	wm,wm,carry	
	SUB	e,e,1	
	JMP	N2	
N4	SLU	carry,wm,63	*N4. Scale right.*
	SRU	wl,wl,1	
	ADDU	wl,wl,carry	
	SRU	wm,wm,1	

```
        ADD    e,e,1
6H      SET    t,#7FFE; CMP t,e,t     N6. Check e.
        BP     t,:Error               Overflow.
        BNP    e,:Error               Underflow.
        SLU    s,s,63                 N7. Pack.
        SLU    e,e,48
        ANDNH  wm,#FFFF               Remove hidden bit.
        OR     wm,wm,s; OR wm,wm,e
        SET    $0,wl
        SET    $1,wm
        POP    2,0                    Return w.
:Zero   POP    0,0                    Return zero. ▮
```

. . .

Now let us consider double-precision multiplication. The product has four components, shown schematically in Fig. 4. If the limited precision ($p = 96$) of the leftmost six WYDEs is sufficient, we can ignore the digits to the right of the vertical line in the diagram; in particular, we need not even compute the product of the two least-significant halves.

		1.uuu	uuuu	$= u_m + \epsilon\, u_l$
		1.vvv	vvvv	$= v_m + \epsilon\, v_l$
		xxxx	xxxx	$= \epsilon^2 u_l \times v_l$
	1.xxx	xxxx		$= \epsilon\, u_m \times v_l$
	1.xxx	xxxx		$= \epsilon\, u_l \times v_m$
3.xx	xxxx			$= u_m \times v_m$
3.ww	wwww	wwww	wwww	$=$

Fig. 4. Double-precision multiplication of seven-WYDE fraction parts.

Program M (*Double-precision multiplication*). The input and output conventions for this subroutine are the same as for Program A.

```
:DFmul  SLU    eu,um,1; SLU ev,vm,1     M1. Unpack.
        SRU    eu,eu,49; SRU ev,ev,49
        XOR    s,um,vm; SRU s,s,63      s ← s_u × s_v.
        ANDNH  um,#FFFF; ORH um,#0001
        ANDNH  vm,#FFFF; ORH vm,#0001
        MULU   t,um,vl                  M2. Operate.
```

07		`GET`	`wl,:rH`	$\mathtt{w_l} \leftarrow 2^{48}u_m \times 2^{64}v_l \times 2^{-64}$.
08		`MULU`	`t,ul,vm`	
09		`GET`	`t,:rH; ADDU wl,wl,t`	$\mathtt{w_l} \leftarrow \mathtt{w_l} + 2^{48}u_l v_m$.
10		`MULU`	`t,um,vm; GET wm,:rH`	$\mathtt{w_m} \leftarrow \lfloor 2^{32}u_m \times v_m \rfloor$.
11		`ADDU`	`wl,wl,t`	$\mathtt{w_l} \leftarrow \mathtt{wl} + \mathtt{um} \times \mathtt{vm} \bmod 2^{64}$.
12		`CMPU`	`t,wl,t; ZSN carry,t,1`	$\mathtt{carry} \leftarrow 1$ if $\mathtt{wl} + \mathtt{t} < \mathtt{t}$.
13		`ADDU`	`wm,wm,carry`	
14		`SLU`	`wm,wm,16`	$\mathtt{wm} \leftarrow 2^{16}\mathtt{wm} = 2^{16}\lfloor 2^{32}u_m \times v_m \rfloor$.
15		`SRU`	`carry,wl,64-16`	
16		`ADDU`	`wm,wm,carry`	
17		`SLU`	`wl,wl,16`	$\mathtt{wl} \leftarrow 2^{16}\mathtt{wl}$.
18		`ADD`	`e,eu,ev`	
19		`SET`	`t,#3FFF; SUB e,e,t`	$e \leftarrow e_u + e_v - q$.
20		`JMP`	`:DNormalize`	M3. Normalize. ▮

Notice that there is no carry into `wm` from the addition in line 09 because `um` and `vm` are smaller than 2^{49}; in line 11, however, we add the low 64 bits of `um` × `vm`, which can be any value less than 2^{64}, so that we need to consider a carry. Program M is perhaps too slipshod in accuracy, since it uses only 49-bit operands when computing the most significant digits of the result in line 10, and it adds 16 zero bits in line 17. More accuracy can be achieved as discussed in exercise 4.

[251]

Program D (*Double-precision division*). This program adheres to the same conventions as Programs A and M. For the `DIVU` (divide unsigned) instruction in line 11 to work properly, we need `um` < `vm`. Since u and v are normalized, shifting (`vm`, `vl`) to the left by one bit would be sufficient. We shift by 15 bits, the maximum amount possible, instead and compute `wm` ← $(2^{64+48}u_m + 2^{64}u_l)/(2^{15+48}v_m) = 2^{48+1}(u_m + \epsilon\, u_l)/v_m$. This moves the radix point in w_m one bit too far to the left. We compensate for this by adjusting the exponent e by -1 in line 09; the "Scale Right" step in the normalization routine will shift `wm` back if necessary. Some more precision could be gained if we shifted v only by one bit, but then the normalization routine would need a "Scale Right" step that is not restricted to shifting a single bit.

01	`:DFdiv`	`SLU`	`eu,um,1; SLU ev,vm,1`	D1. Unpack.
02		`SRU`	`eu,eu,49; SRU ev,ev,49`	

`XOR`	`s,um,vm; SRU s,s,63`	$s_w \leftarrow s_u \cdot s_v$.
`ANDNH`	`um,#FFFF; ORH um,#0001`	
`SLU`	`vm,vm,15; ORH vm,#8000`	$v_m \leftarrow v_m 2^{15}$.
`SRU`	`carry,vl,64-15`	
`ADDU`	`vm,vm,carry`	
`SLU`	`vl,vl,15`	$(v_m, v_l) \leftarrow (v_m, v_l)2^{15}$.
`SUB`	`e,eu,ev; INCL e,#3FFF-1`	$e \leftarrow e_u - e_v + q - 1$.
`PUT`	`:rD,um`	<u>D2. Operate.</u>
`DIVU`	`wm,ul,vm`	`wm` $\leftarrow \lfloor 2^{48+1}(u_m + E\, u_l)/v_m \rfloor$.
`GET`	`r,:rR`	Get remainder r.
`PUT`	`:rD,r; SET t,0`	
`DIVU`	`wl,t,vm`	`wl` $\leftarrow 2^{64} r / v_m$.
`MULU`	`pl,wm,vl; GET pm,:rH`	$(p_m, p_l) \leftarrow w_m \times v_l$.
`PUT`	`:rD,pm`	
`DIVU`	`ql,pl,vm`	$q_l \leftarrow (p_m + E\, p_l)/v_m$.
`CMPU`	`t,wl,ql; ZSN carry,t,1`	`carry` $\leftarrow [w_l < q_l]$.
`SUBU`	`wl,wl,ql; SUBU wm,wm,carry`	$w \leftarrow w - E\, q_l$.
`JMP`	`:DNormalize`	<u>M3. Normalize.</u> ▮

Here is a table of the approximate average computation times for these double-precision subroutines, compared to the single-precision subroutines that appear in Section 4.2.1:

	Single precision	Double precision
Addition	62.3υ	64.4υ
Subtraction	64.3υ	66.4υ
Multiplication	55.8υ	75.6υ
Division	167.5υ	235.5υ

EXERCISES

2. [*20*] Is it strictly necessary to clear the hi-wyde of `um` in line 12 of Program A? After all, these bits get cleared later in line 55 during normalization.

3. [*M20*] Explain why overflow cannot occur during Program M.

4. [*22*] Program M should be changed so that extra accuracy is achieved, essentially by making a better use of the `MULU` instruction. Investigate these alternatives:

(a) Use the low 64 bits now wasted in lines 06 and 08.

(b) Shift the fraction parts left by up to 15 bits when unpacking.

Specify all changes that are required, and determine the difference in execution time caused by these changes.

5. [*24*] How should Program A be changed so that extra accuracy is achieved, essentially by keeping the lowest bits of v in a separate register `vll` and using it to achieve proper rounding in the normalization procedure? Specify all changes that are required, and determine the difference in execution time caused by these changes.

4.3.1. The Classical Algorithms

[266]

For the following `MMIX` subroutines, we assume that u, v, and w are stored in arrays and the addresses of the three arrays are in registers `u`, `v`, and `w`. In principle, the arrays can be in big-endian or little-endian order; that is, if `LOC(`u`)` is the starting address of the array holding $u = (u_{n-1} \ldots u_1 u_0)_b$, then at address `LOC(`$u$`)` we might have either u_{n-1} or u_0. Here, we assume little-endian ordering; thus `LOC(`u`)` is the address of u_0, `LOC(`u`)` + 8 is the address of u_1, and so on. Further, we take $b = 2^{64}$, so that each digit u_j fits in one octabyte.

Program A (*Addition of nonnegative integers*). This subroutine expects four parameters: the addresses of u, v, w, in registers `u`, `v`, and `w` and the value of n in register `n`. To make traversal of the arrays from $j = 0$ to $j = n - 1$ as efficient as possible, we keep the value $8(j - n)$ in register `j` and change the values of `u`, `v`, and `w` to `LOC(`u`)` + $8n$, `LOC(`v`)` + $8n$, and `LOC(`w`)` + $8n$. After these changes, adding the values of `u` and `j` will yield `LOC(`u`)` + $8n + 8(j - n)$ = `LOC(`u`)` + $8j$, which is exactly the address of the digit u_j.

```
:Add   8ADDU  u,n,u                  1   A1. Initialize. u ← u + 8n.
       8ADDU  v,n,v                  1   v ← v + 8n.
       8ADDU  w,n,w                  1   w ← w + 8n.
       SL     j,n,3; NEG j,j         1   j ← 0.
       SET    k,0                    1   k ← 0.
A2     LDOU   t,u,j; ADDU wj,t,k     N   A2. Add digits. wj ← uj + k.
```

```
       ZSZ    k,wj,k                  N       Carry?
       LDOU   t,v,j; ADDU wj,wj,t     N       w_j ← w_j + v_j.
       CMPU   t,wj,t; CSN k,t,1       N       Carry?
       STOU   wj,w,j                  N
       ADD    j,j,8                   N       A3. Loop on j. j ← j + 1.
       PBN    j,A2                    N[1]    Probably j < n.
       STOU   k,w,j                   1       w_n ← k.
       POP    0,0                             ▮
```

The running time for this program is $9\upsilon + 1\mu + N(10\upsilon + 3\mu)$.

[267]

Program S (*Subtraction of nonnegative integers*). The program uses the same conventions as Program A and is very similar to it. It changes the `ADDU` instruction into a `SUBU` instruction as expected and the carry is now a borrow. The `CSN` instruction in line 10 will not work with negative constants, so we set `k` to 1 (not −1) if a subtraction does not make the number smaller.

```
01 :Sub  8ADDU  u,n,u                   1       S1. Initialize.
02       8ADDU  v,n,v                   1
03       8ADDU  w,n,w                   1
04       SL     j,n,3; NEG j,j          1       j ← 0.
05       SET    k,0                     1       k ← 0.
06 S2    LDOU   uj,u,j                  N       S2. Subtract digits.
07       SUBU   wj,uj,k                 N       w_j ← u_j − k.
08       CSNZ   k,uj,0                  N       Carry?
09       LDOU   vj,v,j                  N
10       CMPU   t,wj,vj; CSN k,t,1      N       Carry?
11       SUBU   wj,wj,vj                N       w_j ← w_j − v_j.
12       STOU   wj,w,j                  N
13       ADD    j,j,8                   N       S3. Loop on j. j ← j + 1.
14       PBN    j,S2                    N[1]    Probably j < n.
15       BNZ    k,:Error                1[0]    k ≠ 0 only if u < v.
16       POP    0,0                             ▮
```

The running time of Program S is $9\upsilon + N(10\upsilon + 3\mu)$, just one μ shorter than the corresponding amount for Program A, because it finally tests k but does not store it.

The following MMIX program shows the considerations that are necessary when Algorithm M is implemented on a computer. Fortunately, MMIX has the MULU operation, which delivers a 128-bit result.

Program M (*Multiplication of nonnegative integers*). To make the inner loop as fast as possible, we scale i by 8 and run register `i` from $-8m$ toward zero. Further, we maintain in register `wj` the address (namely `LOC(`w_j`)` $+ 8m$) needed to make `wj` + `i` the address of w_{j+i}. Thanks to the MULU instruction, the value of $\lfloor t/b \rfloor$ needed in step M4 can be found in the rH register (we just need to add the possible carry of the two ADDU instructions).

:Mul	8ADDU	u,m,u; 8ADDU v,n,v	1	M1. Initialize.
	SL	j,n,3; NEG j,j	1	$j \leftarrow 0$.
	8ADDU	wj,m,w	1	`wj` $\leftarrow$ `LOC(`w_j`)` $+ 8m$.
	SL	i,m,3; NEG i,i	1	$i \leftarrow 0$.
0H	STCO	0,wj,i	M	$(w_{m-1} \ldots w_0) \leftarrow (0 \ldots 0)$.
	ADD	i,i,8	M	$i \leftarrow i + 1$.
	PBN	i,0B	$M_{[1]}$	Loop for $0 \le i < m$.
M2	SET	k,0	N	M2. Zero multiplier?
	LDOU	vj,v,j	N	
	BZ	vj,6F	$N_{[Z]}$	If $v_j = 0$, set $w_{j+m} \leftarrow 0$.
	SL	i,m,3; NEG i,i	$N - Z$	M3. Initialize i. $i \leftarrow 0$.
M4	LDOU	t,u,i	$(N - Z)M$	M4. Multiply and add.
	MULU	t,t,vj	$(N - Z)M$	$t \leftarrow u_i \times v_i$.
	ADDU	t,t,k	$(N - Z)M$	$t \leftarrow u_i \times v_i + k$.
	CMPU	c,t,k; ZSN k,c,1	$(N - Z)M$	Carry?
	LDOU	wij,wj,i	$(N - Z)M$	
	ADDU	t,t,wij	$(N - Z)M$	$t \leftarrow u_i \times v_i + k + w_{i+j}$.
	CMPU	c,t,wij; CSN k,c,1	$(N - Z)M$	Carry?
	STOU	t,wj,i	$(N - Z)M$	$w_{i+j} \leftarrow t \bmod b$.
	GET	t,:rH; ADDU k,k,t	$(N - Z)M$	$k \leftarrow \lfloor t/b \rfloor$.
	ADD	i,i,8	$(N - Z)M$	M5. Loop on i. $i \leftarrow i + 1$.
	PBN	i,M4	$(N - Z)M_{[N-Z]}$	
6H	STOU	k,wj,0	$N - Z$	$w_{j+m} \leftarrow k$.

```
24          ADD    wj,wj,8            N        M6. Loop on j.
25          ADD    j,j,8              N        j ← j + 1.
26          PBN    j,M2               N[1]
27          POP    0,0                         ▮
```

The execution time of Program M depends on the number of places, M, in the multiplicand u; the number of places, N, in the multiplier v; and the number of zeros, Z, in the multiplier. We find that the total running time comes to $(23MN + 3M + 11N + 11 - Z(23M + 3))\upsilon + (3MN + M + 2N - Z(3M + 1))\mu$. If step M2 were deleted, the running time would be $(23MN + 3M + 10N + 11)\upsilon + (3MN + M + 2N)\mu$, so that step is advantageous only if the density of zero positions within the multiplier is $Z/N > 1/(23M + 3)$. If the multiplier is chosen completely at random, the ratio Z/N is expected to be only about $1/b$, which is extremely small. Unless the `PBNZ` instruction on line 10 can be done in parallel on a superscalar pipeline processor (such as `MMIX`) with proper branch prediction, causing zero delay if the branch is not taken, we conclude that step M2 is usually *not* worthwhile.

[273]

Program D (*Division of nonnegative integers*). The conventions of this subroutine are analogous to Program A. It expects five parameters: First, `u`, `v`, and `q` hold the addresses of $u = (u_{m+n-1} \ldots u_0)_b$, $v = (v_{n-1} \ldots v_0)_b$ where $v_{n-1} \neq 0$, and $q = (q_m \ldots q_0)_b$; then follow `nu` and `nv`, which hold the number of digits of u and v (we compute m, needed in Algorithm D, as $m =$ `nu` $-$ `nv`). The array u is used as the algorithm's working area. It will contain the remainder r after the program has finished. Similar to Program M, we maintain registers `uj` and `i` such that `uj` + `i` = `LOC(`u_{j+i}`)`. The `DIVU` instruction will not compute quotient $\hat{q}$ and remainder $\hat{r}$ unless rD $= u_{j+n} < v_{n-1}$. So we test for it before attempting the division. In step D1, instead of d, we compute the number of leading zeros in v_{n-1} because shifting is more efficient than multiplication. The variables p_m and p_l in registers `pm` and `pl`, respectively, are used for the most and least significant 64 bits of the product $\hat{q} \times v_{n-2}$. Registers `vn1`, `vn2`, `uji`, and `ujn` are used to hold the values of v_{n-1}, v_{n-2}, u_{j+i}, and u_{j+n}, respectively.

```
:Div    GET    rJ,:rJ                  1
        SL     nv,nv,3; SL nu,nu,3     1
        SUB    t,nv,8                  1        D1. Normalize.
        LDOU   ld+1,v,t                1
```

	PUSHJ	ld,:LeadingZeros	1	See new exercise 4.2.1–20.
	SET	t+1,v; SR t+2,nv,3	1	
	SET	t+3,ld	1	
	PUSHJ	t,:ShiftLeft	1	See new exercise 25.
	SET	t+1,u; SR t+2,nu,3	1	
	SET	t+3,ld	1	
	PUSHJ	t,:ShiftLeft	1	See new exercise 25.
	SET	ujn,t	1	$u_{j+n} \leftarrow$ carry.
	SUB	m,nu,nv	1	$m \leftarrow n_u - n_v$.
	SET	j,m	1	*D2. Initialize j.* $j \leftarrow m$.
	ADDU	v,nv,v	1	v $\leftarrow$ LOC$(v) + 8n$.
	NEG	t,8; LDOU vn1,v,t	1	vn1 $\leftarrow v_{n-1}$.
	NEG	t,16; LDOU vn2,v,t	1	vn2 $\leftarrow v_{n-2}$.
	ADDU	uj,j,u	1	
	ADDU	uj,nv,uj	1	uj $\leftarrow$ LOC$(u) + 8(j+n)$.
	JMP	0F	1	Avoid loading u_{m+n}.
D3	LDOU	ujn,uj,0	M	*D3. Calculate* $\hat{q}$.
0H	CMPU	t,ujn,vn1	$M+1$	
	BNN	t,1F	$M+1_{[0]}$	Jump if $\hat{q}$ would be b.
	NEG	i,8	$M+1$	$i \leftarrow n-1$.
	LDOU	uji,uj,i	$M+1$	Get u_{j+n-1}.
	PUT	:rD,ujn	$M+1$	rD $\leftarrow u_{j+n}$.
	DIVU	qh,uji,vn1	$M+1$	$\hat{q} \leftarrow \lfloor (u_{j+n}b + u_{j+n-1})/v_{n-1} \rfloor$.
	GET	rh,:rR	$M+1$	$\hat{r} \leftarrow \cdots \bmod v_{n-1}$.
	JMP	2F	$M+1$	
1H	SET	qh,0		$\hat{q} \leftarrow b$.
	SET	rh,uji		$\hat{r} \leftarrow u_{j+n} = v_{n-1}$.
3H	SUBU	qh,qh,1	E	Decrease $\hat{q}$ by one.
	ADDU	rh,rh,vn1	E	$\hat{r} \leftarrow \hat{r} + v_{n-1}$.
	CMPU	t,rh,vn1	E	Check if overflow.
	BN	t,D4	$E_{[E-F]}$	If yes, continue the test.
2H	MULU	pl,qh,vn2	$M+F+1$	$p_m b + p_l \leftarrow \hat{q} v_{n-2}$.
	GET	pm,:rH	$M+F+1$	
	CMPU	t,pm,rh	$M+F+1$	Compare high 64 bits.
	PBN	t,D4		

	PBN	t,D4	$M + F + 1_{[E]}$	
	PBP	t,3B	$E_{[0]}$	
	NEG	i,16		$i \leftarrow n - 2$.
	LDOU	uji,uj,i		Get u_{j+n-2}.
	CMPU	t,pl,uji		Compare low 64 bits.
	BP	t,3B		
D4	SET	k,0	$M + 1$	D4. Multiply and subtract.
	NEG	i,nv	$M + 1$	$i \leftarrow 0$.
0H	LDOU	uji,uj,i	$N(M + 1)$	Load u_{j+i}.
	LDOU	t,v,i	$N(M + 1)$	t $\leftarrow v_i$.
	MULU	pl,t,qh	$N(M + 1)$	$(p_m, p_l) \leftarrow v_i \times \hat{q}$.
	GET	pm,:rH	$N(M + 1)$	
	ADDU	pl,pl,k	$N(M + 1)$	$(p_m, p_l) \leftarrow (p_m, p_l) + k$.
	CMPU	t,pl,k; ZSN k,t,1	$N(M + 1)$	Carry from p_l to p_m?
	ADDU	pm,pm,k	$N(M + 1)$	
	CMPU	t,uji,pl; ZSN k,t,1 $N(M + 1)$	$N(M + 1)$	Carry from $u_{i+j} - p_l$?
	SUBU	uji,uji,pl	$N(M + 1)$	$u_{j+\mathrm{i}} \leftarrow u_{j+\mathrm{i}} - v_\mathrm{i} \times \hat{q}$.
	STOU	uji,uj,i	$N(M + 1)$	Store u_{j+i}.
	ADDU	k,pm,k	$N(M + 1)$	Add p_m to carry.
	ADD	i,i,8	$N(M + 1)$	$i \leftarrow i + 1$.
	PBN	i,0B	$N(M + 1)_{[M+1]}$	Repeat for $0 \le i < n$.
	SUBU	uji,ujn,k	$M + 1$	Complete D4 for $i = n$.
	CMPU	t,ujn,k	$M + 1$	
	ZSN	k,t,1	$M + 1$	Borrow to the left?
	CMP	t,j,m; BNN t,D5	$M + 1_{[1]}$	Store unless $j = m$.
	STOU	uji,uj,i	M	$u_{j+\mathrm{n}} \leftarrow u_{j+n} +$ carry.
D5	PBZ	k,1F	$M + 1_{[0]}$	D5. Test remainder.
	SUBU	qh,qh,1		D6. Add back.
	NEG	i,nv		$i \leftarrow -8n$.
	SET	k,0		Carry $\leftarrow 0$.
0H	LDOU	uji,uj,i		
	ADDU	uji,uji,k		$u_{j+\mathrm{i}} \leftarrow u_{j+i} +$ carry.
	ZSZ	k,uji,k		Carry?
	LDOU	t,v,i		t $\leftarrow v_i$.

```
73          ADDU   uji,uji,t                          u_{j+i} ← u_{j+i} + v_i.
74          CMPU   t,uji,t
75          CSN    k,t,1                              Carry?
76          STOU   uji,uj,i
77          ADD    i,i,8
78          PBN    i,0B                               Probably j < 0.
79          LDOU   uji,uj,i
80          ADDU   uji,uji,k
81          STOU   uji,uj,i                           u_{j+n} ← u_{j+i} + carry.
82   1H     STOU   qh,q,j               M + 1         q_j ← q̂.
83   D7     SUB    uj,uj,8              M + 1         D7. Loop on j.
84          SUBU   j,j,8                M + 1         j ← j − 1.
85          PBNN   j,D3                 M + 1[1]
86          SET    t+1,u                  1           D8. Unnormalize.
87          SR     t+2,nv,3               1
88          SET    t+3,ld                 1
89          PUSHJ  t,:ShiftRight          1           See exercise 26.
90          PUT    :rJ,rJ                 1
91          POP    0,0                                ▮
```

The running time for Program D can be estimated by considering the quantities M, N, E, and F shown in the program. (These quantities ignore several situations that occur only with very low probability; for example, we may assume that lines 30 and 31, lines 41–44, and step D6 are never executed.) Here $M + 1$ is the number of words in the quotient; N is the number of words in the divisor; E is the number of times $\hat{q}$ is adjusted downward in step D3; and F is the number of times the full test of $\hat{q}$ in step D3 is required. If we assume that F is approximately $0.5E$ and E is approximately $0.5M$, we get a total running time of approximately $(24MN + 45N + 110.25M + 169)v$. When M and N are large, this is only about five percent longer than the time needed by Program M to multiply the quotient and the divisor.

EXERCISES

[281]

3. [*21*] Write an `MMIX` program for the algorithm of exercise 2, and estimate its running time as a function of m and n.

8. [*M26*] Write an `MMIX` program for the algorithm of exercise 5, and estimate its running time based on the expected number of carries as computed in the text.

10. [*18*] Would Program S work properly if the three instructions on lines 10 and 11 were replaced by '`SUBU wj,wj,vj; CSN k,wj,1`'?

13. [*21*] Write an `MMIX` subroutine that multiplies $(u_{n-1} \ldots u_1 u_0)_b$ by v, where v is a a single-precision number (that is, $0 \le v < b$), producing the answer $(w_n \ldots w_1 w_0)_b$. Assume that $b = 2^{32}$ and numbers are stored as `TETRA` arrays in little-endian order. How much running time is required?

25. [*26*] Write an `MMIX` subroutine `ShiftLeft`, which is needed to complete Program D. `ShiftLeft` accepts three parameters: `LOC(`x`)`, the address of an array of octabytes; n, the size of the array; and p, the number of bits to shift x to the left. If x is considered an n-digit number to the base 2^{64} stored in little-endian order, the routine will transform x to $2^p x$. The bits shifted out of the most significant "digit" of x comprise the return value of the subroutine.

26. [*21*] Write an `MMIX` routine `ShiftRight`, which is needed to complete Program D. `ShiftRight` uses the same conventions as `ShiftLeft` in exercise 25, but shifts it in the other direction.

4.4. RADIX CONVERSION

[320]

B. Single-precision conversion. To illustrate these four methods, suppose that we want to store the decimal representation of a nonnegative (binary) integer u in register `u` as an array U of `BYTE`s in little-endian order at address `u10` ≡ `LOC(`U`)`. With $b = 2$ and $B = 10$, Method 1a could be programmed as follows:

```
      SET   j,0                 Set j ← 0.
      PUT   rD,0                Prepare for DIVU.
1H    DIVU  u,u,10              u ← ⌊u/10⌋ and rR ← u mod 10.
      GET   t,rR; STBU t,u10,j  U_j ← u mod 10.                (1)
      ADD   j,j,1               j ← j + 1.
      PBP   u,1B                Repeat until result is zero. ▮
```

This requires $(64\upsilon + \mu)M + 4\upsilon$ to obtain M digits. The expensive instruction here is the division, which costs 60υ each time.

[321]

For the corresponding MMIX program, we choose $n = 19$, the largest n with $10^n < 2^{64} = w$, and assume that the global register ten19 contains the constant 10^{19}. If $u < 10^n$, we can implement Method 2a as follows:

```
      PUT     rD,u
      DIVU    x,ten19,ten19         x ← ⌊(wu + 10^n)/10^n⌋.
      SET     j,n-1                 j ← n − 1.
0H    MULU    x,x,10                (rH, x) ← 10x.                      (4)
      GET     t,rH; STB t,u10,j     U_j ← ⌊10x⌋.
      SUB     j,j,1                 j ← j − 1.
      PBNN    j,0B                  Repeat for n > j ≥ 0.  ▮
```

This slightly longer routine requires $(14\upsilon + \mu)n + 64\upsilon$, so it is faster than program (1) if no leading zeros are present and $n = M \geq 2$; if leading zeros are present, (1) will be faster if $n = 19$ and $M \leq 5$. The most expensive instruction of the previous program is the MULU inside the loop, which contributes 190υ. If we choose w sufficiently smaller than 2^{64}, we can avoid this multiplication. For example, with 32-bit integers, we choose $w = 2^{32}$ and $n = 9$. We can then write

```
      SLU     u,u,32
      ADD     u,u,ten9
      DIV     x,u,ten9      x ← ⌊(wu + 10^n)/10^n⌋.
      SET     j,n-1         j ← n − 1.
0H    4ADDU   x,x,x
      SLU     x,x,1         x ← 10x.                         (4′)
      SRU     t,x,32
      STBU    t,u10,j       U_j ← ⌊10x⌋.
      ANDNMH  x,#FFFF       x ← x mod w.
      SUB     j,j,1         j ← j − 1.
      PBNN    j,0B          Repeat for n > j ≥ 0.  ▮
```

This routine requires $(7\upsilon + \mu)n + 65\upsilon$; it has a loop twice as fast as before. For $n = 9$, it requires 128υ, which is close to the 131υ required by Method 1a for a two-digit number. With more than two digits, Method 1a is significantly slower.

. . .

An MMIX program for conversion using (5) appears in exercise 8; it requires about 19υ per digit.

Method 1b is the most practical method for decimal-to-binary conversion in the great majority of cases. The following MMIX code assumes that there are at least two digits in the number $(u_m \ldots u_1 u_0)_{10}$ being converted, and that $10^{m+1} < w$ so that overflow is not an issue:

```
     SET    j,m-1                   j ← m − 1.
     LDBU   u,u10,m                 U ← u_m.
1H   MULU   u,u,10                                                  (6)
     LDBU   t,u10,j; ADDU u,u,t     U ← 10U + u_j.
     SUB    j,j,1                   j ← j − 1.
     PBNN   j,1B                    Repeat for m > j ≥ 0. ▮
```

The running time is $(14\upsilon + \mu)m - 10\upsilon$.

The multiplication by 10 can be done in 2υ by '`4ADDU u,u,u; SL u,u,1`', which brings the running time down to $(6\upsilon + \mu)m - 4\upsilon$.

EXERCISES

[328]

5. [*M20*] Show that program (4) would still work if the `DIVU` instruction were replaced by `DIVU x,`c`,`c for certain other constants c.

8. [*24*] Write an `MMIX` program analogous to (1) that uses (5) and includes no division instructions.

13. [*25*] Assume that u is a multiple-precision fraction $u = (.u_{-1}u_{-2} \ldots u_{-m})_b$, where $b = 2^{32}$, and that u is stored as an array of tetrabytes in little-endian order. Write an `MMIX` subroutine with parameters `LOC(`u`)`, m, and `LOC(Buffer)` that converts the fraction u to decimal notation, truncating it to 126 decimal digits. The answer should be stored in the given `Buffer` as an ASCII string, such that the two instructions '`LDA $255,Buffer; TRAP 0,Fputs,StdOut`' print the answer on two lines, with the digits grouped into 14 blocks of nine each separated by blanks.

19. [*M23*] Let the decimal number $u = (u_7 \ldots u_1 u_0)_{10}$ be represented in register `u` as a sequence of eight ASCII characters u_7 + `'0'`, . . . , u_1 + `'0'`, u_0 + `'0'`. Convert the ASCII code representation first to a sequence of eight binary coded numbers $u_7, \ldots, u_1, u_0$. Then find appropriate constants c_i and masks m_i so that the operation `u` ← `u` − c_i(`u` & m_i) repeated for $i = 1, 2, 3$, will convert `u` to the binary representation of u. Write an `MMIX` routine to do the conversion.

4.5.2. The Greatest Common Divisor

[337]

The following MMIX program illustrates the fact that Algorithm A can easily be implemented on a computer.

Program A (*Euclid's algorithm*). Assume that u and v are nonnegative integers. This subroutine expects u and v as parameters and returns gcd(u, v).

```
0H   DIV   t,u,v    A2. Take u mod v.
     SET   u,v      u ← v.
     GET   v,rR     v ← u mod v.
Gcd  PBNZ  v,0B     A1. v = 0? Done if v = 0.
     POP   1,0      Return u. ▮
```

The running time for this program is $(63T + 3)v$, where T is the number of divisions performed.

[339]

An MMIX program for Algorithm B requires a bit more code than for Algorithm A, but the steps are elementary.

Program B (*Binary gcd algorithm*). Assume that u and v are positive integers. This subroutine expects u and v as parameters, uses Algorithm B, and returns gcd(u, v).

```
Gcd  SET   k,0                 1              B1. Find powers of 2.
0H   OR    t,u,v               A + 1
     PBOD  t,B2                A + 1[A]       Both even?
     SR    u,u,1; SR v,v,1     A              u ← u/2 and v ← v/2.
     ADD   k,k,1               A              k ← k + 1.
     JMP   0B                  A
B2   NEG   t,v                 1              B2. Initialize.
     PBOD  u,B4                1[B]
     SET   t,u                 B
B3   SR    t,t,1               D              B3. Halve t.
B4   PBEV  t,B3                1 − B + D[C]   B4. Is t even?
     CSP   u,t,t               C              B5. Reset max(u,v).
     NEG   t,t; CSNN v,t,t     C
     SUB   t,u,v               C              B6. Subtract.
     PBNZ  t,B3                C[1]
```

```
16         SL     u,u,k                        1
17         POP    1,0                                    Return 2^k · u. ▮
```

The running time of this program is

$$(8A + 2B + 7C + 2D + 9)\upsilon,$$

where $A = k$, $B = 1$ if $t \leftarrow u$ in step B2 (otherwise $B = 0$), C is the number of subtraction steps, and D is the number of halving steps in step B3. Calculations discussed later in this section imply that we may take $A = \frac{1}{3}$, $B = \frac{1}{3}$, $C = 0.71\mathrm{N} - 0.5$, and $D = 1.41N - 2.7$ as average values for these quantities, assuming random inputs u and v in the range $1 \le u,\ v < 2^N$. The total running time is therefore about $7.8N + 3.4$ cycles, compared to about $36.8N + 6.8$ cycles for Program A under the same assumptions. The worst possible running time for u and v in this range occurs when $A = 0$, $C = N$, $D = 2N - 2$; this amounts to $11N + 5.7$ cycles. (The corresponding value for Program A is $90.7N + 45.4$ cycles.)

Thus the greater speed of the iterations in Program B, due to the simplicity of the operations, compensates for the greater number of operations required. We have found that the binary algorithm is almost 5 times faster than Euclid's algorithm on the MMIX computer.

EXERCISES

[356]

43. [*20*] *New*: It is possible to compute k in Step B1 of Algorithm B with just three MMIX instructions, because lines 01–06 can be replaced by

```
01  :Gcd   OR     t,u,v                  B1. Find powers of 2.
02         SUBU   k,t,1; SADD k,k,t
03         SR     u,u,k; SR v,v,k        u ← u/2^k and v ← v/2^k. ▮
```

Will this make Program B more efficient?

4.5.3. Analysis of Euclid's Algorithm

EXERCISES

[373]

1. [*20*] Since the quotient $\lfloor u/v \rfloor$ is equal to unity more than 40 percent of the time in Algorithm 4.5.2A, it may be advantageous on some computers to make a test for this case and to avoid the division when the quotient is unity. Is the following MMIX program for Euclid's algorithm more efficient than Program 4.5.2A?

`0H`	`SUB`	`r,u,v`	$r \leftarrow u - v$.
	`SET`	`u,v`	$u \leftarrow v$.
	`NEG`	`v,r; CSN v,v,r`	$v \leftarrow \lvert r \rvert$.
	`CMP`	`t,r,u`	
	`BN`	`t,Gcd`	$r < u$?
	`DIV`	`t,v,u; GET v,:rR`	$v \leftarrow u \bmod v$.
`Gcd`	`PBNZ`	`v,0B`	
	`POP`	`1,0`	▮

4.5.4. Factoring into Primes

[389]

An even more important method of speeding up Algorithm D is to use Boolean operations. For example, MMIX has 64 bits per word. The tables $S[i, k_i]$ can be kept in memory with one bit per entry; thus 64 values can be stored in a single word and the AND instruction can be used to process 64 values of x at once! For convenience, we can make several copies S_i of the tables $S[i, j]$ so that the table entries for m_i involve $\operatorname{lcm}(m_i, 64)$ bits; then the sieve tables for each modulus fill an integral number of words. Under these assumptions, 64 executions of the main loop in Algorithm D are equivalent to code of the following form:

`D2`	`LDOU`	`sieve,S1,k1`	`sieve` $\leftarrow S'[1, k_1']$.
	`CSZ`	`k1,k1,m1*8; SUB k1,k1,8`	$k_1 \leftarrow (k_1 - 64) \bmod \operatorname{lcm}(m_1, 64)$.
	`LDOU`	`t,S2,k2; AND sieve,sieve,t`	`sieve` $\leftarrow$ `sieve` $\&\ S'[2, k_2']$.
	`CSZ`	`k2,k2,m2*8; SUB k2,k2,8`	$k_2 \leftarrow (k_2 - 64) \bmod \operatorname{lcm}(m_2, 64)$.
	⋮		(m_3 through m_r are like m_2)
	`LDOU`	`t,Sr,kr; AND sieve,sieve,t`	`sieve` $\leftarrow$ `sieve` $\&\ S'[r, k'_r]$.
	`CSZ`	`kr,kr,mr*8; SUB kr,kr,8`	$k_r \leftarrow (k_r - 64) \bmod \operatorname{lcm}(m_r, 64)$.
	`ADD`	`x,x,64`	$x \leftarrow x + 64$.
	`PBZ`	`sieve,D2`	Repeat if all sieved out. ▮

The number of cycles for 64 iterations is essentially $(1 + 4r)v$; if $r < 16$, this means that less than one v is being used on each iteration, compared to $3v$ to $5v$

in Algorithm C, and Algorithm C involves $y = \frac{1}{2}(v-u)$ more iterations. The savings in the loop are partially offset by the extra time needed to initialize all the registers and tables.

4.6.3. Evaluation of Powers

EXERCISES

[481]

2. [*22*] Write an `MMIX` subroutine for Algorithm A, with parameters x and $n > 0$, returning x^n mod w (where w is the word size).

Write another `MMIX` subroutine that computes x^n mod w in a serial manner (multiplying repeatedly by x), and compare the running times of these subroutines.

4.6.4. Evaluation of Polynomials

EXERCISES

[516]

20. [*10*] Write an `MMIX` program that evaluates a fifth-degree polynomial according to scheme (11). Use `MMIX`'s floating point instructions.

CHAPTER FIVE

SORTING

EXERCISES

[6]

6. [*15*] Mr. B. C. Dull (an `MMIX` programmer) wanted to know if the number stored in location `A` is greater than, less than, or equal to the number stored in location `B`. So he wrote '`LDO $0,A; LDO $1,B; SUB $2,$0,$1`' and tested whether register `$2` was positive, negative, or zero. What serious mistake did he make, and what should he have done instead?

7. [*17*] Write an `MMIX` subroutine `MCmp` for multiprecision comparison of n-byte keys $(a_{n-1}, \ldots, a_0)$ and $(b_{n-1}, \ldots, b_0)$, where a_i and b_i are unsigned bytes stored in order of increasing index i. Use the following specification:

Calling sequence:	PUSHJ t,MCmp
Entry conditions:	`$0` $\equiv n$; `$1` $\equiv$ `LOC`(a_0); and `$2` $\equiv$ `LOC`(b_0)
Return value:	1, if $(a_{n-1}, \ldots, a_0) < (b_{n-1}, \ldots, b_0)$;
	0, if $(a_{n-1}, \ldots, a_0) \equiv (b_{n-1}, \ldots, b_0)$;
	–1, if $(a_{n-1}, \ldots, a_0) < (b_{n-1}, \ldots, b_0)$.

Here the relation $(a_{n-1}, \ldots, a_0) < (b_{n-1}, \ldots, b_0)$ denotes lexicographic ordering from left to right; that is, there is an index j such that $a_k = b_k$ for $n > k > j$, but $a_j < b_j$.

8. [*20*] Registers `a` and `b` contain two nonnegative numbers a and b, respectively. Find the most efficient `MMIX` program that computes min(a, b) and max(a, b) and assigns these values to registers `min` and `max`. *Hint:* 3υ are sufficient for this task.

5.2. INTERNAL SORTING

[76]

Program C (*Comparison counting*). The following `MMIX` implementation of Algorithm C assumes that keys and counts are stored as arrays of consecutive octabytes. Furthermore, the registers `k`, `count`, and `n` are initialized to contain `LOC(`K_1`)`, `LOC(COUNT[1])`, and N, respectively. To allow a more efficient use of the counts later on (see exercises 4 and 5), we scale the counts by 8.

`:Sort`	`SL`	`i,n,3`	1	*C1. Clear* `COUNT`*s.*
	`JMP`	`0F`	1	
`1H`	`STCO`	`0,count,i`	N	`COUNT`$[i] \leftarrow 0$.
`0H`	`SUB`	`i,i,8`	$N+1$	
	`PBNN`	`i,1B`	$N+1_{[1]}$	$N > i \geq 0$.
	`SL`	`i,n,3`	1	*C2. Loop on i.*
	`JMP`	`1F`	1	
`2H`	`LDO`	`ci,count,i`	$N-1$	
	`LDO`	`ki,k,i`	$N-1$	
`3H`	`LDO`	`kj,k,j`	A	
	`CMP`	`t,ki,kj`	A	*C4. Compare* $K_i : K_j$
	`PBNN`	`t,4F`	$A_{[B]}$	Jump if $K_i \geq K_j$.
	`LDO`	`cj,count,j`	B	`COUNT`$[j]$
	`ADD`	`cj,cj,8`	B	$+1$
	`STO`	`cj,count,j`	B	$\rightarrow$`COUNT`$[j]$.
	`JMP`	`5F`	B	
`4H`	`ADD`	`ci,ci,8`	$A-B$	`COUNT`$[i] \leftarrow$ `COUNT`$[i]+1$.
`5H`	`SUB`	`j,j,8`	A	*C3. Loop on j.*
	`PBNN`	`j,3B`	$A_{[N-1]}$	
	`STO`	`ci,count,i`	$N-1$	
`1H`	`SUB`	`i,i,8`	N	
	`SUB`	`j,i,8`	N	$N > i > j \geq 0$.
	`PBNN`	`j,2B`	$N_{[1]}$	▮

The running time of this program is $(11N+6A+5B+5)\upsilon+(4N+A+2B-3)\mu$. [78]

Hence Program C requires between $(3N^2+8N+5)\upsilon+(0.5N^2+3.5N-3)\mu$ and $(5.5N^2+5.5N+5)\upsilon+(1.5N^2+2.5N-3)\mu$; the average running time lies halfway between these two extremes. For example, the data in Table 1 has $N=16$, $A=120$, $B=41$, so Program C will sort it in $1106\upsilon+263\mu$.

EXERCISES

[79]

4. [*16*] Write an `MMIX` program that "finishes" the sorting begun by Program C; your program should transfer the records $R_1, \ldots, R_N$ to an output area $S_1, \ldots, S_N$ in the desired order. How much time does your program require?

5. [*22*] Does the following set of changes improve Program C?

New line 08a:	`ADD`	`ci,ci,i`
Change line 12:	`PBNN`	`t,5F`
Change line 16:	`SUB`	`ci,ci,8`
Delete line 17.		

9. [*23*] Write an `MMIX` program for Algorithm D, analogous to Program C and exercise 4. What is the execution time of your program, as a function of N and $(v - u)$?

11. [*M27*] Write an `MMIX` program for the algorithm of exercise 10, and analyze its efficiency.

12. [*25*] Design an efficient algorithm suitable for rearranging the records $R_1, \ldots, R_N$ into sorted order, after a list sort (Fig. 7) has been completed. Try to avoid using excess memory space. Write an `MMIX` program for this algorithm.

5.2.1. Sorting by Insertion

[81]

Program S (*Straight insertion sort*). For simplicity, we assume that the records consist just of the keys, which are 64-bit signed integers. This subroutine expects two parameters: `key` ≡ `LOC(`R_1`)` = `LOC(`K_1`)`, the address where the items to be sorted are located; and `n` ≡ N, the number of items. We use the register `i` ≡ $8i$ together with the base addresses `key` and `key1` ≡ `key` + 8 (for the computation of `key` + $8(i + 1)$); register `j` ≡ $8(N - j)$ is used with the base address `keyn` ≡ `key` + $8N$. To convert between the bases, we keep the difference in register `d` ≡ `keyn` − `key1`.

`:Sort`	`ADD`	`key1,key,8`	1	
	`8ADDU`	`keyn,n,key`	1	
	`SUBU`	`d,keyn,key1`	1	
	`SUBU`	`j,key1,keyn`	1	$j \leftarrow 1$.
	`JMP`	`S1`	1	
`S2`	`LDO`	`k,keyn,j`	$N - 1$	S2. Set up j, K, R.
	`ADD`	`i,d,j`	$N - 1$	$i \leftarrow j - 1$.

08	S3	LDO	ki,key,i	$N-1+B-A$	*S3. Compare $K : K_i$.*
09		CMP	c,k,ki	$N-1+B-\mathrm{A}$	
10		BNN	c,S5	$N-1+B-A_{[N\text{-}1\text{-}A]}$	To S5 if $K \geq K_i$.
11		STO	ki,key1,i	B	*S4. Move R_i, decrease i.*
12		SUB	i,i,8	B	$i \leftarrow i-1$.
13		PBNN	i,S3	$B_{[A]}$	To S3 if $i \geq 0$.
14	S5	STO	k,key1,i	$N-1$	*S5. R into R_{i+1}.*
15		ADD	j,j,8	$N-1$	$j \leftarrow j+1$.
16	S1	PBN	j,S2	$\mathrm{N}_{[1]}$	*S1. Loop on j.* $1 \leq j \leq N$.
17		POP	0,0		▮

The running time of this program is $(10N-3A+6B-2)\upsilon + (3N-1A+2B-3)\mu$, where N is the number of records sorted, A is the number of times i decreases to zero in step S4, and B is the number of moves.

The branch in line 10 is optimized for large values of B compared to $N-A$. For an array that is expected to be almost sorted, B might be small compared to $N-A$; in this case the branch should be replaced by a probable branch.

[82]

The average running time of Program S, assuming that the input keys are distinct and randomly ordered, is $(1.5N^2 + 8.5N - 3H_N - 2)\upsilon$. Exercise 33 shows how to improve this slightly.

The example data in Table 1 involves 16 items; there are two changes to the left-to-right minimum, namely 087 and 061; and there are 41 inversions, as we have seen in the previous section. Hence $N = 16$, $A = 2$, $B = 41$, and the total sorting time is 398υ.

. . .

Program D (*Shellsort*). We assume that the increments are stored in an auxiliary table, with h_s in location `H` + $8s$; all increments are less than N. The parameters of the following subroutine are: `key` ≡ `LOC(`K_1`)`, the address of an array of octabytes to be sorted; `n` ≡ N, the number of elements in the array; `inc` ≡ `LOC(H)`, the address of a suitable array of increments; and `t` ≡ t, the number of increments to be used. The use of other registers is similar to Program S; the constant `d`, used to set i to $j-h$ in line 10, is computed once for each h.

01	:Sort	8ADDU	keyn,n,key	1	keyn ← LOC(K_{N+1}).

	`SL`	`s,t,3`	1	$s \leftarrow t - 1$.
	`JMP`	`D1`	1	
`D2`	`LDO`	`h,inc,s`	T	*D2. Loop on j.* $h \leftarrow h_s$.
	`SL`	`h,h,3`	T	
	`ADDU`	`keyh,key,h`	T	`keyh` $\leftarrow$ `LOC`(K_{h+1}).
	`SUBU`	`d,keyn,keyh`	T	`d` $\leftarrow N - h$.
	`SUBU`	`j,keyh,keyn`	T	$j \leftarrow h + 1$.
	`JMP`	`0F`	T	
`D3`	`ADD`	`i,d,j`	$NT - S$	*D3. Set up j, K, R.* $i \leftarrow j - h$.
	`LDO`	`k,keyn,j`	$NT - S$	
`D4`	`LDO`	`ki,key,i`	$B + NT - S - A$	*D4. Compare K : K_i.*
	`CMP`	`c,k,ki`	$B + NT - S - A$	
	`BNN`	`c,D6`	$B + NT - S - A_{[NT-S-A]}$	To D6 if $K \geq K_i$.
	`STO`	`ki,keyh,i`	B	*D5. Move R_i, decrease i.*
	`SUB`	`i,i,h`	B	$i \leftarrow i - h$.
	`PBNN`	`i,D4`	$B_{[A]}$	To D4 if $i \geq 0$.
`D6`	`STO`	`k,keyh,i`	$NT - S$	*D6. R into R_{i+1}.*
	`ADD`	`j,j,8`	$NT - S$	$j \leftarrow j + 1$.
`0H`	`PBN`	`j,D3`	$NT - S + T_{[T]}$	To D3 if $j < N$.
`D1`	`SUB`	`s,s,8`	$T + 1$	*D1. Loop on s.*
	`PBNN`	`s,D2`	$T + 1_{[1]}$	$0 \leq s < t$. ▮

[92]

Let's consider practical sizes of N more carefully by looking at the *total* running time of Program D, namely $(6B + 10NT + 11T - 10S - 3A + 7)\upsilon + (2B + 3NT + T - 3S - A)\mu$. Table 5 shows the average running time for various sequences of increments when $N = 8$. For this small value of N, bookkeeping operations are the most significant part of the cost, and the best results are obtained when $t = 1$; hence for $N = 8$, we are better off using simple straight insertion. (The average running time of Program S when $N = 8$ is only 154υ.) Curiously, the best two-pass algorithm occurs with `MMIX` when $h_1 = 7$ (this was $h_1 = 6$ for the `MIX` computer), since a large value of S is more important here than a small value of B.

[94]

Increments	A_{ave}	B_{ave}	S	T	MMIX υ	MMIX μ
1	1.718	14.000	1	1	166.85	48.28
2 1	2.667	9.657	3	2	208.94	57.65
3 1	2.917	9.100	4	2	194.85	53.28
4 1	3.083	10.000	5	2	189.75	51.92
5 1	2.601	10.000	6	2	181.20	49.40
6 1	2.135	10.667	7	2	176.60	48.20
7 1	1.718	12.000	8	2	175.85	48.28
4 2 1	3.500	8.324	7	3	249.44	67.15
5 3 1	3.301	8.167	9	3	229.10	61.03
3 2 1	3.320	7.829	6	3	257.01	69.34

Table 5 Analysis of Algorithm D when $N = 8$

Since Program D takes $(6B + 10(NT - S) + \cdots)\upsilon$, we see that saving one pass is about as desirable as saving $\frac{10}{6}N$ moves; when $N = 1000$ we are willing to add 1666 moves if we can save one pass. (The first pass is very quick, however, if h_{t-1} is near N, because $NT - S = (N - h_{t-1}) + \cdots + (N - h_0)$.)

[97]

Program L (*List insertion*). We assume that K_j is stored in the octabyte at `LOC(`R_0`)` $+ 16j +$ `KEY` and L_j is stored in the octabyte at `LOC(`R_0`)` $+ 16j$. The subroutine has two parameters: `link` $\equiv$ `LOC(`R_0`)` = `LOC(LINK(`R_0`))` = `LOC(`L_0`)` and `n` $\equiv N$, the number of records. The registers `p` and `q`, as well as the link fields, contain relative addresses using `LOC(`R_0`)` as base address.

`:Sort`	`ADDU`	`key,link,KEY`	1	L1. Loop on j.
	`SL`	`j,n,4`	1	$j \leftarrow N$.
	`STOU`	`j,link,0`	1	$L_0 \leftarrow N$.
	`STCO`	`0,link,j`	1	$L_N \leftarrow 0$.
	`JMP`	`0F`	1	Go to decrease j.
`L2`	`LDOU`	`p,link,0`	$N-1$	L2. Set up p, q, K. $p \leftarrow L_0$.
	`SET`	`q,0`	$N-1$	$q \leftarrow 0$.
	`LDO`	`k,key,j`	$N-1$	$K \leftarrow K_j$.
`L3`	`LDO`	`kp,key,p`	$B+N-1-A$	L3. Compare $K : K_p$.
	`CMP`	`t,k,kp`	$B+N-1-A$	
	`BNP`	`t,L5`	$B+N-1-A_{[N-1-A]}$	To L5 if $K \le K_p$.
	`SET`	`q,p`	B	L4. Bump q, q. $q \leftarrow p$.
	`LDOU`	`p,link,q`	B	$p \leftarrow L_q$.

```
          PBNZ  p,L3             B[A]      To L3 if p ≠ 0.
    L5    STOU  j,link,q         N − 1     L5. Insert into list. Lq ← j.
          STOU  p,link,j         N − 1     Lj ← p.
    0H    SUB   j,j,16           N         j ← j − 1.
          PBP   j,L2             N[1]      N > j ≥ 1.  ▮
```

The running time of this program is $(6B + 12N - 3A - 3)\upsilon + (2B + 5N - A - 3)\mu$, where N is the length of the file, $A + 1$ is the number of right-to-left maxima, and B is the number of inversions in the original permutation. (See the analysis of Program S. Note that Program L does not rearrange the records in memory; this can be done as in exercise 5.2–12, at a cost of about $17N$ additional units of time.) Program S requires $(6B + 10N - 3A - 2)\upsilon$, and we can see that the extra memory space used for the link fields has not bought us any extra speed. If, however, the records contain other data besides the key and the link field, the copy operation of Program S will require one `LDO` and one `STO` for each additional memory word. So for each additional octabyte, the running time of Program S will increase by $2B\upsilon + 2B\mu$, which is about 33 percent of the running time. The running time of Program L can be reduced by 33 percent by careful programming (see exercise 33), but the running time remains proportional to N^2.

[99]

To illustrate this approach, suppose that the 16 keys used in our examples are divided into the $M = 4$ ranges 0–255, 256–511, 512–767, 768–1023. We obtain the following configurations as the keys $K_1, K_2, \ldots, K_{16}$ are successively inserted:

	After 4 items:	After 8 items:	After 12 items:	Final state:
List 1:	061, 087	061, 087, 170	061, 087, 154, 170	061, 087, 154, 170
List 2:	503	275, 503	275, 426, 503, 509	275, 426, 503, 509
List 3:	512	512	512, 653	512, 612, 653, 677, 703, 765
List 4:		897, 908	897, 908	897, 908

(Program M below actually inserts the keys in reverse order, $K_{16}, \ldots, K_2, K_1$, but the final result is the same.) Because linked memory is used, the varying-length lists cause no storage allocation problem. All lists can be combined into a single list at the end, if desired (see exercise 35).

Program M (*Multiple list insertion*). In this program we make the same assumptions as in Program L, except that the keys must be *nonnegative* in the range

$$0 \le K_j < 2^e$$

for some suitable value of $e \le 64$. The program divides this range into M equal parts by multiplying each key by a suitable constant. As before, `p`, `q`, and the link fields contain relative addresses using the address of the artificial record `LOC(`R_0`)` as base address. The lists heads H_1 to H_M are allocated as M consecutive octabytes with nonzero relative addresses. Besides `link` ≡ `LOC(`R_0`)` and `n` ≡ N, the subroutine takes `head` ≡ `LOC(`H_1`)` and `m` ≡ M as parameters; $e \le 64$ is assumed to be constant.

`:Sort`	`SL`	`i,m,3`	1	i ← M.
	`JMP`	`1F`	1	
`0H`	`STCO`	`0,head,i`	M	Clear heads.
`1H`	`SUB`	`i,i,8`	$M+1$	$i \leftarrow i - 1$.
	`PBNN`	`i,0B`	$M+1_{[1]}$	
	`SUBU`	`head,head,link`	1	Make `head` a relative address.
	`ADDU`	`key,link,KEY`	1	<u>*M1. Loop on j.*</u>
	`SL`	`j,n,4`	1	$j \leftarrow N$.
	`JMP`	`0F`	1	
`M2`	`LDO`	`k,key,j`	N	<u>*M2. Set up p, q, K.*</u> $K \leftarrow K_j$.
	`MUL`	`i,m,k`	N	$i \leftarrow M \cdot K_j$.
	`SRU`	`i,i,e-3`	N	$i \leftarrow \lfloor M \cdot K_j/2^e \rfloor$.
	`ADDU`	`q,head,i`	N	$q \leftarrow$ relative address of H_i.
	`JMP`	`4F`	N	Jump to load and test p.
`M3`	`LDO`	`kp,key,p`	$B+N-A$	<u>*M3. Compare $K : K_p$.*</u>
	`CMP`	`t,k,kp`	$B+N-A$	
	`BNP`	`t,M5`	$B+N-A_{[N-A]}$	To L5 if $K \le K_p$.
	`SET`	`q,p`	B	<u>*M4. Bump p, q.*</u> $q \leftarrow p$.
`4H`	`LDOU`	`p,link,q`	$B+N$	$p \leftarrow L_q$.
	`PBNZ`	`p,M3`	$B+N_{[A]}$	To L3 if $p \ne 0$.
`M5`	`STOU`	`j,link,q`	N	<u>*M5. Insert into list.*</u> $L_q \leftarrow j$.
	`STOU`	`p,link,j`	N	$L_j \leftarrow p$.
	`SUB`	`j,j,16`		

N

24	`0H`	`PBP`	`j,M2`	$N+1_{[1]}$	$N > j \geq 1$. ▮

This program is written for general M, but it would be much better to fix M at some convenient value; for example, if the range of keys is $0 \leq K_j < 2^e$, we can choose $d < e$ and $M = 2^d$, so that the multiplication sequence of lines 11–12 could be replaced by the single instruction `SRU i,k,e-3-d`, reducing the total running time by $10N\upsilon$. In the following discussion, we shall consider this improved version of Program M, unless otherwise noted.

The most notable contrast between Program L and Program M is the fact that Program M must consider the case of an empty list, when no comparisons are to be made.

How much time do we save by having M lists? The total running time of (the improved) Program M is $(6B + 15N - 3A + 3M + 13)\upsilon + (2B + 5N - A + M)\mu$, . . .

[101]

By combining (17) and (18) we can deduce the total running time of Program M, for fixed M as $N \to \infty$:

$$\begin{array}{ll} \text{min} & 12N + 3M + 13, \\ \text{ave} & 1.5N^2/M + 15N - 3MH_N + 3M\ln M + 3M - 3\delta - 1.5N/M + 13, \\ \text{max} & 3N^2 + 12N + 3M + 10, \end{array} \qquad (19)$$

. . .

If we set $M = N$, the average running time of Program M is approximately $(17.11N + 11.5)\upsilon + (5.70N - 0.5)\mu$; when $M = \frac{1}{2}N$ it is approximately $(16.02N + 11.5)\upsilon + (5.34N - 0.5)\mu$; and when $M = \frac{1}{10}N$ it is approximately $(15.94N + 11.5)\upsilon + (5.31N - 2.5)\mu$. The additional cost of the supplementary program in exercise 35, which links all M lists together in a single list, raises these times respectively to $(28.00N + 8.5)\upsilon + (8.34N - 1.5)\mu$, $(23.32N + 5.5)\upsilon + (7.27N - 2.5)\mu$, $(19.84N - 18.5)\upsilon + (6.51N - 10.5)\mu$. (Note that an extra $10N\upsilon$ is necessary if the multiplication by M cannot be avoided.)

EXERCISES

[102]

3. [*30*] Is Program S the shortest possible sorting program that can be written for MMIX, or is there a shorter program that achieves the same effect?

10. [*22*] If $K_j \geq K_{j-h}$ when we begin step D3, Algorithm D specifies a lot of actions that accomplish nothing. Show how to modify Program D so that this redundant computation can be avoided, and discuss the merits of such a modification.

[104]

31. [*25*] Write an MMIX program for Pratt's sorting algorithm (exercise 30). Express its running time in terms of quantities A, B, S, T, N analogous to those in Program D.

33. [*25*] Find a way to improve an Program L so that its running time is dominated by $4B$ instead of $6B$, where B is the number of inversions. Discuss corresponding improvements to Program S.

35. [*21*] Write an MMIX program to follow Program M, so that all lists are combined into a single list. Your program should set the LINK fields exactly as they would have been set by Program L.

36. [*18*] The sixteen example keys in Table 8 fit nicely into the range $0 \leq K_j < 2^{10}$. Determine the running time of Programs L and M on this data, when $M = 4$.

5.2.2. Sorting by Exchanging

[107]

Program B (*Bubble sort*). As in previous MMIX programs of this chapter, the Sort subroutine expects two parameters: key $\equiv$ LOC(K_1), the address where the items to be sorted are located; and n $\equiv$ N, the number of items. For simplicity, we assume that the records consist of just the key, which is a 64-bit signed integer. Instead of the index BOUND, we maintain the address of K_{BOUND} in register keyb.

:Sort	SUB	n,n,1	1	*B1. Initialize* BOUND.
	8ADDU	keyb,n,key	1	BOUND ← N.
	JMP	B2	1	
B3	LDO	kj,keyb,j	A	*B3. Compare/exchange* $R_j : R_{j+1}$.
B3A	ADD	j,j,8	C	$j \leftarrow j + 1$.
	LDO	kjj,keyb,j	C	kjj ← K_{j+1}.
	CMP	c,kj,kjj	C	$K_j > K_{j+1}$?
	BNP	c,0F	$C_{[C-B]}$	If $K_j > K_{j+1}$,

	`STO`	`kj,keyb,j`	B	interchange $R_j \leftrightarrow R_{j+1}$.
	`SUB`	`t,j,8`	B	$t \leftarrow j$.
	`STO`	`kjj,keyb,t`	B	$K_j \leftarrow K_{j+1}$.
	`PBN`	`j,B3A`	$B_{[D]}$	
	`JMP`	`1F`	D	To B4 (but skip test for termination).
`0H`	`SET`	`kj,kjj`	$C - B$	`kj` $\leftarrow K_j$.
	`PBN`	`j,B3A`	$C - B_{[A-D]}$	
`B4`	`BZ`	`t,9F`	$A - D_{[1]}$	<u>*B4. Any exchanges?*</u>
`1H`	`ADD`	`keyb,keyb,t`	$A - 1$	`BOUND` $\leftarrow t$.
`B2`	`SET`	`t,0`	A	<u>*B2. Loop on j.*</u> t $\leftarrow 0$.
	`SUB`	`j,key,keyb`	A	$j \leftarrow 1$.
	`PBN`	`j,B3`	$A_{[0]}$	$1 \le j <$ `BOUND`.
`9H`	`POP`	`0,0`		▮

Analysis of the bubble sort. It is quite instructive to analyze the running time of Algorithm B. Four quantities are involved in the timing: the number of passes, A; the number of exchanges, B; the number of comparisons, C; and the number of times that a pass ends with an exchange, D. The running time of Program B (not counting the final `POP`) is $(4 + 8A + 8C)\upsilon + (A + 2B + C)\mu$; fortunately, it does not depend on D (which appears subtle to analyze).

. . .

In example (1) we therefore have $A = 9$, $B = 41$, $C = 15 + 14 + 13 + 12 + 7 + 5 + 4 + 3 + 2 = 75$. The total `MMIX` sorting time for Fig. 14 is $676\upsilon + 166\mu$.

[109]

In each case the minimum occurs when the input is already in order, and the maximum occurs when it is in reverse order; so the `MMIX` running time is $(4 + 8A + 8C)\upsilon + (A + 2B + C)\mu = (\text{min } (6N + 6)\upsilon + N\mu, \text{ ave } (4N^2 + O(N \ln N))\upsilon + (N^2 + O(N \ln N))\mu, \text{ max } (5N^2 + 4N + 4)\upsilon + (1.5N^2 - 0.5N)\mu)$.

[117]

The corresponding `MMIX` program is rather long, but not complicated; in fact, a large part of the coding is devoted to step Q7, which uses recursion to make use of the `MMIX` register stack.

Program Q (*Quicksort*). Records to be sorted are octabyte values. Assume that the extra records R_0 and R_{N+1} contain, respectively, the smallest and largest 64-

bit signed number.

Instead of the index l, we maintain the register `left` ≡ `LOC(`R_{l-1}`)`; it serves as base address for the registers `i`, `j`, and `r`, which are scaled to make `LOC(`K_i`)` = `left` + `i` and similarly for `j` and `r`. The stack is kept on the register stack of `MMIX`. The recursive part from steps Q2 to Q8 is called with two parameters, the address `$0` ≡ `left` and the offset `$1` ≡ `LOC(`R_{r+1}`)` − `LOC(`R_{l-1}`)`, such that the addresses of all records to be sorted are then strictly between `$0` and `$0` + `$1`. Instead of using `$1` to hold `r`, we use it to hold `j`. This is very convenient for the recursive calls, since in step Q7, the left partition simply has the parameters `left` and `j`, and the right partition has the parameters `left` + `j` and `r` − `j`.

To keep the stack frame for each invocation as small as possible, we choose `key` ≡ `left` ≡ `$0`, `n` ≡ `j` ≡ `$1`, `rJ` ≡ `$2`, and `t` ≡ `$3`; all other local registers have register numbers greater than 3.

`:Sort`	`CMP`	`t,n,M`	1	*Q1. Initialize.*
	`BNP`	`t,Q9`	$1_{[0]}$	To Q9 if $N \le M$.
	`GET`	`rJ,:rJ`	1	
	`SUBU`	`t+1,key,8`	1	$l \leftarrow 0$.
	`8ADDU`	`t+2,n,8`	1	$r \leftarrow N+1$.
	`PUSHJ`	`t,Q2`	1	To Q2.
	`PUT`	`:rJ,rJ`	1	
	`JMP`	`Q9`	1	
`Q2`	`SET`	`i,16`	A	*Q2. Begin new stage.* $i \leftarrow l+1$.
	`LDO`	`k,left,8`	A	`k` $\leftarrow K_l$.
	`SET`	`r,j`	A	$r \leftarrow j$.
	`JMP`	`0F`	A	
`Q6`	`STO`	`ki,left,j`	B	*Q6. Exchange.* $K_j \leftarrow K_i$.
	`STO`	`kj,left,i`	B	$K_i \leftarrow K_j$.
`Q3`	`ADD`	`i,i,8`	$C' - A$	*Q3. Compare $K_i : K$.* $i \leftarrow i+1$.
`0H`	`LDO`	`ki,left,i`	C'	`ki` $\leftarrow K_i$.
	`CMP`	`t,ki,k`	C'	If $K_i < K$,
	`PBN`	`t,Q3`	$C'_{[A]}$	repeat this step.
`Q4`	`SUB`	`j,j,8`	$C - C'$	*Q4. Compare $K : K_j$.* $j \leftarrow j-1$.
	`LDO`	`kj,left,j`	$C - C'$	`kj` $\leftarrow K_j$.
	`CMP`	`t,k,kj`	$C - C'$	If $K < K_j$,

	`PBN`	`t,Q4`	$C - C'_{[B+A]}$	repeat this step.
	`CMP`	`t,i,j`	$B + A$	<u>*Q5. Test i : j.*</u>
	`PBN`	`t,Q6`	$B + A_{[A]}$	If $i < j$ go to Q6.
	`STO`	`kj,left,8`	A	Interchange $R_l \leftrightarrow R_j$.
	`STO`	`k,left,j`	A	
	`SUB`	`d,r,j`	A	<u>*Q7. Put on stack.*</u> `d` $\leftarrow r - j$.
	`CMP`	`t,d,j`	A	
	`BNN`	`t,0F`	$A_{[A-A']}$	Put smaller subfile on stack.
	`CMP`	`t,j,8*M+8`	A'	Is left subfile too small?
	`BNP`	`t,Q8`	$A'[A'-S'-A'']$	To Q8 if $M + 1 \geq j > r - j$.
	`CMP`	`t,d,8*M+8`	$S' + A''$	If right subfile is too small,
	`PBNP`	`t,Q2`	$S' + A''_{[S']}$	go to Q2 with l and j.
	`GET`	`rJ,:rJ`	S'	Now $j > r - j > M + 1$.
	`ADDU`	`t+1,left,j`	S'	To Q2 with $l + j$.
	`SET`	`t+2,d`	S'	and $r - j$.
	`PUSHJ`	`t,Q2`	S'	$(l, j) \Rightarrow$ stack.
	`PUT`	`:rJ,rJ`	S'	
	`JMP`	`Q2`	S'	To Q2 with l and j.
`0H`	`CMP`	`t,d,8*M+8`	$A - A'$	Is right subfile too small?
	`BNP`	`t,Q8`	$A - A'_{[A-A'-S+S'-A''']}$	To Q8 if $M + 1 \geq r - j \geq j$.
	`CMP`	`t,j,8*M+8`	$S - S' + A'''$	Is left subfile too small?
	`PBNP`	`t,0F`	$S - S' + A'''_{[S-S']}$	Jump if $r - j > M + 1 \geq j$
	`GET`	`rJ,:rJ`	$S - S'$	Now $r - j \geq j > M + 1$.
	`SET`	`t+1,left`	$S - S'$	Continue with l
	`SET`	`t+2,j`	$S - S'$	and j.
	`ADD`	`left,left,j`	$S - S'$	$l \leftarrow l + j$.
	`SET`	`j,d`	$S - S'$	$j \leftarrow r - j$.
	`PUSHJ`	`t,Q2`	$S - S'$	$(l + j, r - j) \Rightarrow$ stack.
	`PUT`	`:rJ,rJ`	$S - S'$	
	`JMP`	`Q2`	$S - S'$	To Q2 with $l + j$ and $r - j$.
`0H`	`ADD`	`left,left,j`	A'''	Now $r - j > M \geq j - 0$.
	`SET`	`j,d`	A'''	
	`JMP`	`Q2`	A'''	To Q2 with $l + j$ and $r - j$.
`Q8`	`POP`	`0,0`	S	<u>*Q8. Take off stack.*</u>
`Q9`	`SL`	`j,n,3`	1	<u>*Q9. Straight insertion sort.*</u>

	`SUB`	`j,j,8`	1	$j \leftarrow N-1$.
	`SUBU`	`key0,key,8`	1	`key0` $\leftarrow$ `LOC`(K_0).
	`JMP`	`S1`	1	
`S2`	`LDO`	`ki,key,j`	$N-1$	S2. Set up j, K, R.
	`SUB`	`j,j,8`	$N-1$	
	`LDO`	`kj,key,j`	$N-1$	
	`CMP`	`t,kj,ki`	$N-1$	S3. Compare $K : K_i$.
	`PBNP`	`t,S1`	$N-1_{[D]}$	
	`ADD`	`i,j,8`	D	
`S4`	`STO`	`ki,key0,i`	E	S4. Move Ri, increase i.
	`ADD`	`i,i,8`	E	
	`LDO`	`ki,key,i`	E	S3. Compare $K : K_i$.
	`CMP`	`t,kj,ki`	E	
	`PBP`	`t,S4`	$E_{[D]}$	
	`STO`	`kj,key0,i`	D	$R_{i+1} \leftarrow R_j$.
`S1`	`PBP`	`j,S2`	$N_{[1]}$	S1. Loop on j. ▮

Analysis of quicksort. The timing information shown with Program Q is not hard to derive using Kirchhoff's conservation law (Section 1.3.3) and the fact that everything put onto the stack is eventually removed again. Kirchhoff's law applied at Q2 also shows that

$$A = 1 + A'' + 2S' + 2(S - S') + A''' = 2S + 1 + A'' + A''', \qquad (15)$$

hence the total running time comes to

$$(25A - 2A' - 3A'' + 6B + 4C + 6D + 5E + 6N + 7S - 2S' + 6)\upsilon + (3A + 2B + C + D + 2E + 2N - 2)\mu$$

where

$$\begin{aligned} S' &= \text{number of times } j > r - j > M+1;\\ A' &= \text{number of times } r - j < j;\\ A'' &= \text{number of times } j > M+1 \geq r-j;\\ A''' &= \text{number of times } r - j > M+1 \geq j;\\ A &= \text{number of partitioning stages;}\\ B &= \text{number of exchanges in step Q7;}\\ C &= \text{number of comparisons made while partitioning;}\\ D &= \text{number of times } K_{j-1} > K_j \text{ during straight insertion (step Q9);}\\ E &= \text{number of inversions removed by straight insertion;}\\ S &= \text{number of times an entry is put on the stack.} \end{aligned} \qquad (16)$$

Because of symmetry, we may assume $A'' = A'''$, $A' = A - A'$, and $S' = S - S'$. This simplifies the total running time to

$$(22.5A+6B+4C+6D+5E+6N+9S+7.5)\upsilon+(3A+2B+C+D+2E+2N-2)\mu.$$

[121]

Formulas (24) and (25) can be used to determine the best value of M on a particular computer. In `MMIX`'s case, Program Q requires $10(N+1)H_{N+1} + \frac{1}{6}(N+1)f(M) - 27$ cycles on the average, for $N > 2M + 1$, where

$$f(M) = 5M - 60H_{M+2} + 87 - 72\frac{H_{M+1}}{M+2} + \frac{264}{M+2} + \frac{54}{2M+3}. \qquad (26)$$

We want to choose M so that $f(M)$ is a minimum, and a simple computer calculation shows that $M = 12$ is best. The average running time of Program Q is approximately $10(N+1)\ln N - 7.27N - 34.27$ cycles when $M = 12$, for large N.

So Program Q is quite fast, on average, considering that it requires very little memory space. Its speed is primarily due to the fact that the inner loops, in steps Q3 and Q4, are extremely short—only four `MMIX` instructions each (see lines 15–18 and 19–22).

[125]

Program R (*Radix exchange sort*). The following `MMIX` code expects the parameters `key` ≡ `LOC(`K_1`)`, `n` ≡ N, and `b` ≡ 2^{m-1}. It keeps the addresses of K_l, K_r, and K_j in registers `left`, `right`, and `j`; instead of i and j, the main loop, maintains the difference `d` ≡ $8(i - j)$. The code uses recursion, keeping the return address rJ and the values of `right` and `b` on the register stack. The function returns the final value of `right` so that processing can continue with `left` = `right` + 8. Steps R2 to R10 form the body of this recursive procedure. Its parameters are `$0` ≡ `right` ≡ `LOC(`K_r`)`, `$2` ≡ `b`, and `$3` ≡ `left` ≡ `LOC(`K_l`)`; the second parameter is ignored and the corresponding register `$1` is used later to save the return address. Passing the address of K_j as the new value for `right` does not need an instruction, since `j` ≡ `$4` is the same register as `left+1`.

`:Sort`	`SET`	`left,key`	1	R1. Initialize. $l \leftarrow 1$.
	`8ADDU`	`right,n,left`	1	
	`SUBU`	`right,right,8`	1	$r \leftarrow N$.
`R2`	`SET`	`j,right`	A	R2. Begin new stage. $j \leftarrow r$.
	`SUB`	`d,left,j`	A	$i \leftarrow l$.
`R3`	`LDO`	`ki,j,d`	C'	R3. Inspect K_i for 1.
	`AND`	`t,ki,b`	C'	
	`PBZ`	`t,R4`	$C'_{[B+X]}$	To R4 if it is 0.
`R6`	`SUBU`	`j,j,8`	$C'' + X$	R6. Decrease j. $j \leftarrow j - 1$.
	`BNN`	`d,R8`	$C' + X_{[X]}$	To R8 if $j < i$.
	`ADD`	`d,d,8`	C''	$j \leftarrow j - 1$.
	`LDO`	`kj,j,8`	C''	R5. Inspect K_{j+1} for 0.
	`AND`	`t,kj,b`	C''	
	`BNZ`	`t,R6`	$C''_{[C'' - B]}$	To R6 if it is 1.
	`STO`	`ki,j,8`	B	R7. Exchange R_i, R_{j+1}.
	`STO`	`kj,j,d`	B	
`R4`	`ADD`	`d,d,8`	$C' - X$	R4. Increase i. $i \leftarrow i + 1$.
	`PBNP`	`d,R3`	$C' - X_{[A - X]}$	To R3 if $i \le j$.
`R8`	`BOD`	`b,R10`	$A_{[G]}$	R8. Test special cases.
	`SRU`	`b,b,1`	$A - G$	$b \leftarrow b + 1$.
	`CMPU`	`t,j,right`	$A - G$	
	`BNN`	`t,R2`	$A - G_{[R]}$	To R2 if $j = r$.
	`CMPU`	`t,j,left`	$A - G - R$	
	`BN`	`t,R2`	$A - G - R_{[L]}$	To R2 if $j < l$.
	`BZ`	`t,0F`	$A - G - R - L_{[K+1]}$	To R9 if $j \ne l$.
	`SET`	`left+3,b`	S	R9. Put on stack.
	`SET`	`left+4,left`	S	
	`GET`	`rJ,:rJ`	S	
	`PUSHJ`	`left,R2`	S	Call R2 with $(K_j, \cdot, b, K_l)$.
	`PUT`	`:rJ,rJ`	S	

```
OH     ADDU   left,left,8       S + K + 1            l ← return value + 1.
       CMP    t,left,right      S + K + 1            R2. Begin new stage.
       BN     t,R2              S + K + 1[K+G]       To R2 if l < r.
R10    POP    1,0               S + 1                R10. Take off stack.  ▮
```

[127]

By Kirchhoff's law, $S = A - G - R - L - K - 1$; so the total running time comes to $(22A + 2B + 5C' + 8C'' - 13G - 4K - 10L - 12R + 2X)\upsilon + (C' + C'' + 2B)\mu$. Assuming $C' = C'' = C/2$, this simplifies to $(22A + 2B + 6.5C - 13G - 4K - 10L - 12R + 2X)\upsilon + (2B + C)\mu$.

. . .

Here $\alpha = 1/\ln 2 \approx 1.4427$. Notice that the average number of exchanges, bit inspections, and stack accesses is essentially the same for both kinds of data, even though case (ii) takes about 44 percent more stages. Our MMIX program takes approximately $10.1N \ln N$ units of time, on the average, to sort N items in case (ii), and this could be cut to about $8.66N \ln N$ using the suggestion of exercise 34; the corresponding figure for Program Q is $10.0N \ln N$, which can be decreased to about $8.91N \ln N$ using Singleton's median-of-three suggestion.

Thus radix exchange sorting takes about as long as quicksort, on the average, when sorting uniformly distributed data. On some machines, such as MMIX, it is actually a little quicker than quicksort.

EXERCISES

[134]

12. [*24*] Write an MMIX program for Algorithm M. How much time does your program take to sort the sixteen records in Table 1?

34. [*20*] How can the bit-inspection loops of radix exchange (in steps R3 through R6) be speeded up?

55. [*22*] Show how to modify Program Q so that the partitioning element is the median of the three keys (28), assuming that $M > 1$.

56. [*M43*] Analyze the average behavior of the quantities that occur in the running time of Algorithm Q when the program has been modified to take the median of three elements as in exercise 55. (See exercise 29.)

5.2.3. Sorting by Selection

[140]

Program S (*Straight selection sort*). As in previous programs of this chapter, the parameters `key` ≡ `LOC(`K_1`)` and `n` ≡ N are passed to this subroutine to sort the records in place on a full octabyte key.

```
:Sort  SL    j,n,3       1          S1. Loop on j. j ← N.
       JMP   1F          1
2H     SET   k,j         N − 1      S2. Find max(K_1, . . . , K_j).
       SET   i,j         N − 1      i ← j.
       LDO   max,key,i   N − 1      max ← K_i.
3H     SUB   k,k,8       A          Loop on k.
       LDO   kk,key,k    A          kk ← K_k.
       CMP   t,max,kk    A          Compare max : K_k.
       PBNN  t,0F        A[B]       If max < K_k,
       SET   i,k         B          i ← k and
       SET   max,kk      B          max ← K_k.
0H     PBP   k,3B        A[N − 1]   Repeat if k > 0.
       LDO   t,key,j     N − 1      S3. Exchange with R_j.
       STO   max,key,j   N − 1
       STO   t,key,i     N − 1
1H     SUB   j,j,8       N          Decrement j.
       PBP   j,2B        N[1]       N > j > 0. ▮
```

. . .

Thus the average running time of Program S is $(2.5N^2 + 4(N + 1)H_N - 0.5N - 4)\upsilon + (0.5N^2 + 3.5N - 4)\mu$, noticeably slower than straight insertion (Program 5.2.1S).

[146]

Program H (*Heapsort*). The records K_1 through K_N are sorted by Algorithm H. The subroutine expects the parameters `key` ≡ `LOC(`K_1`)` and `n` ≡ N.

```
:Sort  SLU   r,n,3       1          H1. Initialize.
       SUB   r,r,8       1          r ← N.
       SRU   l,n,1       1
```

	`SLU`	`l,l,3`	1	$l \leftarrow \lfloor N/2 \rfloor$.
	`BNP`	`l,9F`	$1_{[0]}$	Terminate if $N < 2$.
`1H`	`SUB`	`l,l,8`	$\lfloor N/2 \rfloor$	$l \leftarrow l - 1$.
	`LDO`	`k,key,l`	$\lfloor N/2 \rfloor$	$K \leftarrow K_l$.
	`SET`	`j,l`	$\lfloor N/2 \rfloor$	*H3. Prepare for siftup.* $j \leftarrow l$.
	`JMP`	`H4`	$\lfloor N/2 \rfloor$	
`5H`	`LDO`	`kj,key,j`	$B + A$	`kj` $\leftarrow K_j$.
	`BZ`	`t,H6`	$B + A_{[D]}$	To H6 if $j = r$.
	`ADD`	`j1,j,8`	$B + A - D$	*H5. Find larger child.* $j1 \leftarrow j + 1$.
	`LDO`	`kj1,key,j1`	$B + A - D$	`kj1` $\leftarrow K_{j+1}$.
	`CMP`	`t,kj,kj1`	$B + A - D$	Compare $K_j : K_{j+1}$.
	`CSNP`	`j,t,j1`	$B + A - D$	If $K_j < K_{j+1}$, $j \leftarrow j + 1$.
	`CSNP`	`kj,t,kj1`	$B + A - D$	If $K_j < K_{j+1}$, `kj` ← `kj1`.
`H6`	`CMP`	`t,k,kj`	$B + A$	*H6. Larger than K?*
	`BNN`	`t,H8`	$B + A_{[A]}$	To H8 if $K \geq K_j$.
	`STO`	`kj,key,i`	B	*H7. Move it up.* $R_i \leftarrow R_j$.
`H4`	`SET`	`i,j`	$B + P$	*H4. Advance downward.* $i \leftarrow j$.
	`2ADDU`	`j,j,8`	$B + P$	$j \leftarrow 2j + 1$.
	`CMP`	`t,j,r`	$B + P$	Compare $j : r$.
	`PBNP`	`t,5B`	$B + P_{[P-A]}$	Jump if $j \leq r$.
`H8`	`STO`	`k,key,i`	P	*H8. Store R.* $K_i \leftarrow K$.
	`BP`	`l,1B`	$P_{[\lfloor N/2 \rfloor - 1]}$	*H2. Decrease l or r.*
`2H`	`LDO`	`k,key,r`	$N - 1$	If $l = 0$, set $K \leftarrow K_r$.
	`LDO`	`t,key,0`	$N - 1$	
	`STO`	`t,key,r`	$N - 1$	$K_r \leftarrow K_1$.
	`SUB`	`r,r,8`	$N - 1$	$r \leftarrow r - 1$.
	`SET`	`j,0`	$N - 1$	*H3. Prepare for siftup.* $j \leftarrow l$.
	`PBP`	`r,4B`	$N - 1_{[1]}$	To H3 if $r > 1$.
	`STO`	`k,key,0`	1	$K_1 \leftarrow K$.
`9H`	`POP`	`0,0`		▮

[148]

The total running time,

$(9A + 14B + 17N - 3D + 8\lfloor N/2\rfloor - 16)\upsilon + (2A + 3B + 4.5N - D + \lfloor N/2\rfloor - 4)\mu$, is therefore approximately $(14N \lg N - 2N - 3 \ln N - 16)\upsilon + (3N \lg N - \ln N - 4)\mu$ on the average.

A glance at Table 2 makes it hard to believe that heapsort is very efficient; large keys migrate to the left before we stash them at the right! It is indeed a strange way to sort, when N is small; the sorting time for the 16 keys in Table 2 is 898υ, while the simple method of straight insertion (Program 5.2.1S) takes only 393υ. Straight selection (Program S) takes 852υ.

For larger N, Program H is more efficient. It invites comparison with shellsort (Program 5.2.1D) and quicksort (Program 5.2.2Q), since all three programs sort by comparisons of keys and use little or no auxiliary storage. When $N = 1000$, the approximate average running times on `MMIX` are

$$140000\upsilon \text{ for heapsort,}$$
$$100000\upsilon \text{ for shellsort,}$$
$$70000\upsilon \text{ for quicksort.}$$

(`MMIX` is a typical computer, but particular machines will of course yield somewhat different relative values.) As N gets larger, heapsort will be superior to shellsort, but its asymptotic running time $14N \lg N \approx 20.2N \ln N$ will never beat quicksort's $10N \ln N$.

. . .

We always have

$$A \leq 1.5N, \qquad B \leq N\lfloor \lg N\rfloor, \qquad C \leq N\lfloor \lg N\rfloor, \tag{8}$$

so Program H will take no more than $14N\lfloor \lg N\rfloor + 34.5N - 16$ units of time, regardless of the distribution of the input data.

EXERCISES

[156]

8. [*24*] Show that if the search for $\max(K_1, \ldots, K_j)$ in step S2 is carried out by examining keys in left-to-right order $K_1, K_2, \ldots, K_j$, instead of going from right to left as in Program S, it is often possible to reduce the number of comparisons needed on the next iteration of step S2. Write an `MMIX` program based on this observation.

9. [*M25*] What is the average number of comparisons performed by the algorithm of exercise 8, for random input?

5.2.4. Sorting by Merging

[162]

We can sketch the time in the inner loop as follows, if we assume that there is low probability of equal keys:

	Step	Operations	Time
	N3	`CMP`	$1v$
Either	*N3*	`BP` (good guess), `CMP`, `BZ` (good guess)	$3v$
	N4	`STO`, `ADD`	$2v$
	N5	`ADD`, `SET`, `LDO`, `CMP`, `PBNP` (good guess)	$5v$
Or	*N3*	`BP` (bad guess)	$3v$
	N8	`STO`, `ADD`	$2v$
	N9	`SUB`, `SET`, `LDO`, `CMP`, `PBNP` (good guess)	$5v$

Thus about $11v$ is spent on each record in each pass, and the total running time will be asymptotically $11N\lg N$, for both the average case and the worst case. This is a little bit slower than quicksort's average time, and it may not be enough better than heapsort to justify taking twice as much memory space, since the asymptotic running time of Program 5.2.3H is never more than $14N\lg N$.

[163]

The former tests for stepdowns have been replaced by decrementing q or r and testing the results for zero; this reduces the asymptotic `MMIX` running time to $9N\lg N$ units, slightly faster than we were able to achieve with Algorithm N. (The implementation of exercise 9 reduces this further to $8N\lg N$ units.)

[164]

Algorithm L (*List merge sort*).

. . .

The use of signed links is well suited to `MIX`, but unfortunately not to `MMIX` and most other computers. Instead of the sign bit, we use the least significant bit of a link as a `TAG` field; `TAG(`L_s`)` $= 1$ denotes the end of an ordered sublist. `MMIX` ignores this tag bit when the link value is used as an address; the `TAG` bit can be tested with `BEV` (branch if even) or `BOD` (branch if odd) instructions. Inside the inner loop, it is too expensive to extract the tag bit from L_s and set it in p before storing p; instead, we keep track of the location of the initial link to an ordered sublist by setting $s_0 \leftarrow s$ each time we start a new sublist and set $\texttt{TAG}(L_{s_0}) \leftarrow 1$ after we finish the sublist. This method can be used on all computers that have "spare bits" in address values.

L1. [Prepare two lists.] Set $L_i \leftarrow i + 2$ and $\texttt{TAG}(L_i) = 1$ for $1 \le i \le N - 2$, $L_0 \leftarrow 1$, $\texttt{TAG}(L_0) = 1$, $L_{N+1} \leftarrow 2$, $\texttt{TAG}(L_{N+1}) = 1$, $L_N \leftarrow 0$, $\texttt{TAG}(L_N) = 1$, $L_{N-1} \leftarrow 0$, and $\texttt{TAG}(L_{N-1}) = 1$. (We have created two lists containing R_1, R_3, R_5, . . . and R_2, R_4, R_6, . . . , respectively; the `TAG` fields indicate that each ordered sublist consists of one element only. For another way to do this step, taking advantage of ordering that may be present in the initial data, see exercise 12.)

L2. [Begin new pass.] Set $s \leftarrow 0$, $S_0 \leftarrow s$, $t \leftarrow N + 1$, $p \leftarrow L_s$, $\texttt{TAG}(p) = 0$, $q \leftarrow L_t$, $\texttt{TAG}(q) = 0$. If $q = 0$, the algorithm terminates. (During each pass, p and q traverse the lists being merged; s_0 points to the location of the initial link to the current sublist; s usually points to the most recently processed record of the current sublist; while t points to the end of the previously output sublist.)

L3. [Compare $K_p : K_q$.] If $K_p > K_q$, go to L6.

L4. [Advance p.] Set $L_s \leftarrow p$, $s \leftarrow p$, $p \leftarrow L_p$. If $\texttt{TAG}(p) = 0$, return to L3.

L5. [Complete the sublist.] Set $L_s \leftarrow q$, $s \leftarrow t$. Then set $t \leftarrow q$ and $q \leftarrow L_q$, one or more times, until $\texttt{TAG}(q) = 1$. Finally go to L8.

L6. [Advance q.] (Steps L6 and L7 are dual to L4 and L5.) Set $L_s \leftarrow q$, $s \leftarrow q$, $q \leftarrow L_q$. If $\texttt{TAG}(q) = 0$, return to L3.

L7. [Complete the sublist.] Set $L_s \leftarrow p$, $s \leftarrow t$. Then set $t \leftarrow p$ and $p \leftarrow L_p$, one or more times, until $\texttt{TAG}(p) = 1$.

L8. [End of pass?] (At this point $\texttt{TAG}(p) = 1$ and $\texttt{TAG}(q) = 1$, since both pointers have moved to the end of their respective sublists.) Set $\texttt{TAG}(L_{s_0}) \leftarrow 1$, $s_0 \leftarrow s$, $\texttt{TAG}(p) \leftarrow 0$, $\texttt{TAG}(q) \leftarrow 0$. If $q = 0$, set $L_s \leftarrow p$, $L_t \leftarrow 0$ and return to L2. Otherwise return to L3. ▮

. . .

Let us now construct an `MMIX` program for Algorithm L, to see whether the list manipulation is advantageous from the standpoint of speed as well as space:

Program L (*List merge sort*). For convenience, we assume that records are one octabyte long, with L_j in the low `TETRA` and K_j in the high `TETRA`. The parameters are `key` ≡ `LOC(`R_0`)` = `LOC(`K_0`)`, the location of the first key, and `n` ≡ N, the number of records to be sorted.

`:Sort`	`SL`	`n,n,3`	1	<u>*L1. Prepare two lists.*</u>
	`ADDU`	`link,key,4`	1	`link` ← `LOC`(L_0).
	`SUB`	`p,n,16`	1	$p \leftarrow N-2$.
	`BN`	`p,9F`	$1_{[0]}$	Terminate if $N < 2$.
	`OR`	`q,n,1`	1	$q \leftarrow N$, `TAG`$(q) \leftarrow 1$.
`0H`	`STTU`	`q,link,p`	$N-2$	`LINK`(p) ← q.
	`SUB`	`q,q,8`	$N-2$	$q \leftarrow q-1$.
	`SUB`	`p,p,8`	$N-2$	$p \leftarrow p-1$.
	`PBP`	`p,0B`	$N-2_{[1]}$	Repeat until $p = 0$.
	`SET`	`c,8|1`	1	
	`STTU`	`c,link,0`	1	$L_0 \leftarrow 1$, `TAG`$(L_0) \leftarrow 1$.
	`SUB`	`c,n,8`	1	
	`ADDU`	`linkn1,link,c`	1	`linkn 1` ← `LOC`(L_{N-1}).
	`SET`	`c,16|1`	1	
	`STTU`	`c,linkn1,16`	1	$L_{N+1} \leftarrow 2$, `TAG`$(L_{N+1}) \leftarrow 1$.
	`SET`	`c,0|1`	1	
	`STTU`	`c,linkn1,8`	1	$L_N \leftarrow 0$, `TAG`$(L_N) \leftarrow 1$.
	`STTU`	`c,linkn1,0`	1	$L_{N-1} \leftarrow 0$, `TAG`$(L_{N-1}) \leftarrow 1$.
	`JMP`	`L2`	1	
`L3`	`CMP`	`c,kp,kq`	C	<u>*L3. Compare $K_p : K_q$.*</u>
	`BP`	`c,L6`	$C_{[C'']}$	If $K_p > K_q$, go to L6.
`L4`	`STTU`	`p,link,s`	C'	<u>*L4. Advance p.*</u> $L_s \leftarrow p$.
	`SET`	`s,p`	C'	$s \leftarrow p$.
	`LDTU`	`p,link,p`	C'	$p \leftarrow L_p$.
	`LDT`	`kp,key,p`	C'	`kp` ← K_p.
	`PBEV`	`p,L3`	$C'_{[B']}$	If `TAG`$(p) = 0$, return to L3.
`L5`	`STTU`	`q,link,s`	B'	<u>*L5. Complete the sublist.*</u> $L_s \leftarrow q$.
	`SET`	`s,t`	B'	$s \leftarrow t$.
`0H`	`SET`	`t,q`	D'	$t \leftarrow q$.

	LDTU	q,link,q	D'	$q \leftarrow L_q$.
	BEV	q,0B	$D'_{[D'-B']}$	Repeat until $\mathtt{TAG}(q) = 1$.
	LDT	kq,key,q	B'	$\mathtt{kq} \leftarrow K_q$.
	JMP	L8	B'	Go to L8.
L6	STTU	q,link,s	C''	*L6. Advance q.* $L_s \leftarrow q$.
	SET	s,q	C''	$s \leftarrow q$.
	LDTU	q,link,q	C''	$q \leftarrow L_q$.
	LDT	kq,key,q	C''	$\mathtt{kq} \leftarrow K_q$.
	PBEV	q,L3	$C''_{[B'']}$	If $\mathtt{TAG}(q) = 0$, return to L3.
L7	STTU	p,link,s	B''	*L7. Complete the sublist.* $L_s \leftarrow p$.
	SET	s,t	B''	$s \leftarrow t$.
0H	SET	t,p	D''	$t \leftarrow p$.
	LDTU	p,link,p	D''	$p \leftarrow L_p$.
	BEV	p,0B	$D''_{[D''-B'']}$	Repeat until $\mathtt{TAG}(p) = 1$.
	LDT	kp,key,p	B''	$\mathtt{kp} \leftarrow K_p$.
L8	LDTU	c,link,s0	B	*L8. End of pass?*
	OR	c,c,1	B	
	STTU	c,link,s0	B	$\mathtt{TAG}(L_{s_0}) \leftarrow 1$.
	SET	s0,s	B	$s_0 \leftarrow s$.
	ANDN	p,p,1	B	$\mathtt{TAG}(p) \leftarrow 0$.
	ANDN	q,q,1	B	$\mathtt{TAG}(q) \leftarrow 0$.
	PBNZ	q,L3	$B_{[A]}$	If $q \neq 0$, go to L3.
	OR	p,p,1	A	
	STTU	p,link,s	A	$L_s \leftarrow p$, $\mathtt{TAG}(L_s) \leftarrow 1$.
	SET	c,1	A	
	STTU	c,link,t	A	$L_t \leftarrow 0$, $\mathtt{TAG}(L_t) \leftarrow 1$.
L2	SET	s,0	$A+1$	*L2. Begin new pass.* $s \leftarrow 0$.
	SET	s0,s	$A+1$	$s_0 \leftarrow s$.
	ADDU	t,n,8	$A+1$	$t \leftarrow N+1$.
	LDTU	p,link,s	$A+1$	$p \leftarrow L_s$.
	ANDN	p,p,1	$A+1$	Clear TAG bit.
	LDTU	q,link,t	$A+1$	$q \leftarrow L_t$.
	ANDN	q,q,1	$A+1$	Clear TAG bit.
	LDT	kp,key,p	$A+1$	$\mathtt{kp} \leftarrow K_p$.

```
         LDT    kq,key,q       A + 1      kq ← K_q.
         PBNZ   q,L3           A + 1[1]   Terminate if q = 0.
   9H    POP    0,0                       ▮
```

The running time of this program can be deduced using techniques we have seen many times before (see exercises 13 and 14); it comes to approximately $(8N \lg N + 21.5N)\upsilon$ on the average, with a small standard deviation of order $\sqrt{N}$. Exercise 15 shows that the running time can be reduced to about $(6.5N \lg N)\upsilon$ at the expense of a somewhat longer program.

Thus we have a victory for linked-memory techniques over sequential allocation, when internal merging is being done: Typically, less memory space is required, and using all possible optimizations, the program runs about 10 percent faster. On the other hand, we haven't considered the effects of cache memory, which can be complicated.

EXERCISES

[167]

9. [*24*] Write an `MMIX` program for Algorithm S. Specify instruction frequencies in terms of quantities analogous to A, B', B'', C', . . . in Program L.

13. [*M34*] Give an analysis of the average running time of Program L, in the style of other analyses in this chapter: Interpret the quantities A, B, B', . . . , and explain how to compute their exact average values. How long does Program L take to sort the 16 numbers in Table 3?

15. [*20*] Hand simulation of Algorithm L reveals that it occasionally does redundant operations; the assignments $L_s \leftarrow p$, $L_s \leftarrow q$ in steps L4 and L6 are unnecessary about half of the time, since we have $L_s = p$ (or q) each time step L4 (or L6) returns to L3. How can Program L be improved so that this redundancy disappears?

5.2.5. Sorting by Distribution

[173]

Program R (*Radix list sort*). The given records are assumed to have a link field at offset `LINK` = 0 and a p-byte key field at offset `KEY` + 8 − p. We use $M = 256$ and extract the next a_k from the key with a simple `LDBU` (load byte unsigned) instruction. The following subroutine has four parameters: `key` ≡ `LOC(`R_1`)`, the location of the records; `n` ≡ N, the number of records; `p` ≡ p, the number of bytes in the key; and `bot` ≡ `LOC(BOTM[0])`, the location of the 256 bottom link fields followed by the 256 top link fields. We keep the variable named `P` in register `P` (using an uppercase name for a register) because we are already using `p` for p, the length of the key.

`:Sort`	`GET`	`rJ,:rJ`	1	First pass.
	`SET`	`t+1,bot`	1	
	`PUSHJ`	`t,:Empty`	1	_R2. Set piles empty._
	`SET`	`t,M`	1	
	`8ADDU`	`top,t,bot`	1	`top` ← `LOC(TOP[0])`.
	`16ADDU`	`P,n,key`	1	_R1. Loop on k._ `P` ← `LOC`(R_{N+1}).
	`SET`	`k,KEY+7`	1	$k \leftarrow 1$.
`0H`	`SUBU`	`P,P,16`	N	_R5. Step to next record._
	`LDBU`	`i,P,k`	N	_R3. Extract rst digit of key._
	`SL`	`i,i,3`	N	
	`LDOU`	`ti,top,i`	N	_R4. Adjust links._ `ti` ← `TOP[i]`.
	`STOU`	`P,ti,LINK`	N	`LINK(TOP[i])` ← `P`.
	`STOU`	`P,top,i`	N	`TOP[i]` ← `P`.
	`SUB`	`n,n,1`	N	
	`PBP`	`n,0B`	$N_{[1]}$	
	`JMP`	`R6`	1	Later passes.
`R2`	`SET`	`t+1,bot`	$P-1$	_R2. Set piles empty._
	`PUSHJ`	`t,:Empty`	$P-1$	
	`SUB`	`k,k,1`	$P-1$	
`R3`	`LDBU`	`i,P,k`	$N(P-1)$	_R3. Extract kth digit of key._
	`SL`	`i,i,3`	$N(P-1)$	
	`LDOU`	`ti,top,i`	$N(P-1)$	_R4. Adjust links._ `ti` ← `TOP[i]`.
	`STOU`	`P,ti,LINK`	$N(P-1)$	`LINK(TOP[i])` ← `P`.
	`STOU`	`P,top,i`	$N(P-1)$	`TOP[i]` ← `P`.
	`LDOU`	`P,P,LINK`	$N(P-1)$	_R5. Step to next record._
	`PBNZ`	`P,R3`	$N(P-1)_{[P-1]}$	To R3 if not end of pass.
`R6`	`SET`	`t+1,bot`	P	_R6. Do Algorithm H._
	`PUSHJ`	`t,:Hook`	P	
	`LDOU`	`P,bot,0`	P	`P` ← `BOTM[0]`.
	`SUB`	`p,p,1`	P	_R1. Loop on k._
	`PBP`	`p,R2`	$P_{[1]}$	
	`PUT`	`:rJ,rJ`	1	

```
          POP     0,0                                    ▮
```

The running time of Program R is $(7P + 1)N + 11PM + 26P + 8$ cycles, where N is the number of input records, M is the radix (the number of piles), and P is the number of passes. This includes the running time for the two auxiliary procedures: `Hook` and `Empty`. Both procedures are called P times.

After the `Hook` procedure, the first bottom link is pointing to the entire list.

```
:Hook     SET     i,M*8             1           H1. Initialize. i ← 0.
          ADDU    bot,bot,i         1           bot ← LOC(BOTM[M + 1]).
          ADDU    top,bot,i         1           top ← LOC(TOP[M + 1]).
          NEG     i,i               1           Now bot + i = LOC(BOTM[i])
          JMP     H2                1             and top + i = LOC(TOP[i]).
0H        LDOU    bi,bot,i          M − 1       bi ← BOTM[i].
          BZ      bi,H3             M − 1[E′]   H4. Is pile empty?
          STOU    bi,P,LINK         M − 1 − E′  H5. Tie piles together.
H2        LDOU    P,top,i           M − E′      H2. Point to top of pile.
H3        ADD     i,i,8             M           H3. Next pile.
          PBN     i,0B              M[1]
          STCO    0,P,LINK          1           Terminate list.
          POP     0,0               1           ▮
```

The total running time for the `Hook` procedure is $(6M + 8)\upsilon + (3M - 2E' - 1)\mu$, where E' is the number of occurrences of empty piles in each pass.

After the `Empty` procedure, all piles are empty.

```
:Empty    SET     i,M*8         1      i ← M.
          ADDU    top,bot,i     1      top ← LOC(TOP[0]).
          SUB     i,i,8         1      i ← i − 1.
0H        ADDU    bi,bot,i      M      bi ← LOC(BOTM[i]).
          STCO    0,bi,LINK     M      BOTM[i] ← Λ.
          STOU    bi,top,i      M      TOP[i] ← LOC(BOTM[i]).
          SUB     i,i,8         M      i ← i − 1.
          PBNN    i,0B          M[1]   0 ≤ i < M.
          POP     0,0           1      ▮
```

The `Empty` procedure takes $(5M + 8)\upsilon + 2M\mu$.

EXERCISES

[177]

5. [*20*] *New*: What changes are necessary to Program R so that it uses $M = 2^m$, sorting keys of length $Pm \le 64$ bits in P passes? What is the running time of the program, after these changes have been made?

5.3.1. Minimum-Comparison Sorting

EXERCISES

[196]

28. [*40*] Write an MMIX program that sorts five one-byte keys in the minimum possible amount of time, and halts. (See the beginning of Section 5.2 for ground rules.)

5.5. SUMMARY, HISTORY, AND BIBLIOGRAPHY

[381]

Table 1 summarizes the speed and space characteristics of many of these methods, when programmed for MMIX.

. . .

since MMIX is a fairly typical computer.

[383]

The case $N = 16$ refers to the sixteen keys that appear in so many of the examples of Section 5.2; the binary representation of the keys requires 10 bits. The case $N = 1000$ refers to the sequence $K_1, K_2, \ldots, K_{1000}$ of 32-bit keys defined by

$$X_0 = 0;\ X_{n+1} = (6364136223846793005X_n + 9754186451795953191) \bmod 2^{64};$$

$$K_n = \lfloor X_n/2^{32} \rfloor.$$

For the multiplier, see Section 3.3.4, page 108; 9754186451795953191 is some random increment value. An MMIX program of reasonably high quality has been used to represent each algorithm in the table, often incorporating improvements that have been suggested in the exercises.

EXERCISES

2. [*20*] Based on the information in Table 1, what is the best list-sorting method for 32-bit keys, for use on the `MMIX` computer?

Method	Reference	Stable?	Length of MMIX code	Space	Running Time: Average	Running Time: Maximum	$N = 16$	$N = 1000$	Notes
Comparison counting	Ex. 5.2–5	Yes	23	$N(1+\epsilon)$	$4N^2 + 8N$	$5.5N^2$	1042	4046134	c
Distribution counting	Ex. 5.2–9	Yes	35	$2N + 2^{16}\epsilon$	$15N + 8 \cdot 2^{16} + 29$	$15N$	7054	539310	a
Straight insertion	Ex. 5.2.1–33	Yes	15	$N+1$	$1.25N^2 + 9.75N$	$2.5N^2$	377	1291521	
Shellsort	Prog. 5.2.1D	No	22	$N + \epsilon \lg N$	$2.58N^{7/6} + 10N \lg N + 111N$	$cN^{3/4}$	443	103798	d, h
List insertion	Ex. 5.2.1–33	Yes	27	$N(1+\epsilon)$	$1N^2 + 11N$	$2N^2$	356	1036420	c
Multiple list insertion	Prog 5.2.1M	No	24	$N + \epsilon(N+128)$	$0.012N^2 + 15N$	$3N^2$	286	26092	c, f, i
Merge exchange	Ex. 5.2.2–12	No	39	N	$2.75N(\lg N)^2$	$3.5N(\lg N)^2$	819	258142	
Quicksort	Prog. 5.2.2Q	No	72	$N + 3\epsilon \lg N$	$10N \ln N - 7.27N$	$\geq 2N^2$	401	67587	
Median-of-3 quicksort	Ex. 5.2.2–55	No	91	$N + 3\epsilon \lg N$	$8.91N \ln N - 3.66N$	$\geq 2N^2$	413	67384	e
Radix exchange	Ex. 5.2.2–34	No	61	$N + 5 \cdot 20\epsilon$	$8.66N \ln N - 1.14N$	$291N$	400	63975	g, i
Straight selection	Prog. 5.2.3S	No	17	N	$2.5N^2 + 4N \ln N$	$3.5N^2$	852	2529124	j
Heapsort	Prog. 5.2.3H	No	33	N	$20.2N \ln N - 2N$	$20.2N \ln N$	898	137106	h, j
List merge	Prog. 5.2.4L	Yes	66	$N(1+\epsilon)$	$11.5N \ln N - 21.5N$	$11.5N \ln N$	757	90571	c, j
Radix list sort	Prog. 5.2.5R	Yes	33	$N + \epsilon(N+512)$	$29N + 11376$	$29N$	5932	40376	c

a: Sixteen-bit (that is, two-byte) keys only.

c: Output not rearranged; final sequence is specified implicitly by links or counters.

d: Increments chosen as in 5.2.1–(11); a slightly better sequence appears in exercise 5.2.1–29; $N^{7/6}$ is not rigorous.

e: $M = 11$.

f: $M = 2^2 = 4$ for $N = 16$; $M = 2^7 = 128$ for average, maximum, and $N = 1000$.

g: $M = 32$.

h: The average time is based on an empirical estimate, since the theory is incomplete.

i: The average time is based on the assumption of uniformly distributed keys.

j: Further refinements, mentioned in the text and exercises accompanying this program, would reduce the running time.

Table 1 A Comparison of Internal Sorting Methods Using the `MMIX` Computer

CHAPTER SIX

SEARCHING

6.1. SEQUENTIAL SEARCHING

[397]

Program S (*Sequential search*). Assume that the keys K_i are stored as an array of octabyte values.

The following subroutine has three parameters: `key` ≡ `LOC(`K_1`)`; `n` ≡ N, the number of keys; and `k` ≡ K, the key we want to find. After a successful search, the subroutine returns the location of the key found; otherwise, it returns zero. For efficiency, register `i` is scaled by 8, the size of the table entries. Further, we subtract $8N$, the table size, from `i` and add it to `key`. With this trick, we can replace the test $i \le N$ by $8(i - N) < 0$ and control the loop with a single `PBN` instruction.

`:Search`	`SL`	`i,n,3`	1	<u>*S1. Initialize.*</u>
	`NEG`	`i,i`	1	`i` ← $-8N$, $i \leftarrow 1$.
	`SUBU`	`key,key,i`	1	`key` ← `LOC`(K_{N+1}).
`S2`	`LDO`	`ki,key,i`	C	<u>*S2. Compare.*</u>
	`CMP`	`t,k,ki`	C	
	`BZ`	`t,Success`	$C_{[S]}$	Exit if $K = K_i$.
	`ADD`	`i,i,8`	$C - S$	<u>*S3. Advance.*</u>
	`PBN`	`i,S2`	$C - S_{[1-S]}$	<u>*S4. End of file?*</u>
	`POP`	`0,0`		Return zero if not in table.
`Success`	`ADDU`	`$0,key,i`	S	Return `LOC`(K_i).
	`POP`	`1,0`		▮

The analysis of this program is straightforward; it shows that the running time of Algorithm S depends on two things,

$$C = \text{the number of key comparisons;} \qquad S = 1 \text{ if successful, } 0 \text{ if unsuccessful.} \tag{1}$$

Program S takes $(5C - S + 5)\upsilon + C\mu$ units of time. If the search successfully finds $K = K_i$, we have $C = i$, $S = 1$; hence the total time is $(5i + 4)\upsilon + i\mu$. On the other hand if the search is unsuccessful, we have $C = N$, $S = 0$, for a total time of $(5N + 5)\upsilon + N\mu$.

. . .

Program Q (*Quick sequential search*). This algorithm is the same as Algorithm S, except that it assumes the presence of a dummy record R_{N+1} at the end of the file.

:Search	SL	i,n,3	1	<u>*Q1. Initialize.*</u>
	NEG	i,i	1	i $\leftarrow -8N$, $i \leftarrow 1$.
	SUBU	key,key,i	1	key $\leftarrow$ LOC(K_{N+1}).
	STO	k,key,0	1	$K_{N+1} \leftarrow K$.
	JMP	Q2	1	
Q3	ADD	i,i,8	$C - S$	<u>*Q3. Advance.*</u>
Q2	LDO	ki,key,i	$C - S + 1$	<u>*Q2. Compare.*</u>
	CMP	t,k,ki	$C - S + 1$	
	PBNZ	t,Q3	$C - S + 1_{[1]}$	To Q3 if $K \neq K_i$.
	PBN	i,Success	$1_{[1-S]}$	<u>*Q4. End of file?*</u>
	POP	0,0		Exit if not in table.
Success	ADDU	$0,key,i	S	Return LOC(K_i).
	POP	1,0		▮

In terms of the quantities C and S in the analysis of Program S, the running time has decreased to $(4C - 5S + 13)\upsilon + (C - S + 2)\mu$; this is an improvement whenever $i \geq 5$ in a successful search, and whenever $N \geq 9$ in an unsuccessful search.

The transition from Algorithm S to Algorithm Q makes use of an important speed-up principle: When an inner loop of a program tests two or more conditions, we should try to reduce the testing to just one condition.

Another technique will make Program Q *still* faster.

Program Q′ (*Quicker sequential search*).

:Search	SL	i,n,3	1	<u>*Q1. Initialize.*</u>
	NEG	i,i	1	i $\leftarrow -8N$, $i \leftarrow 1$.
	SUBU	key,key,i	1	key $\leftarrow$ LOC(K_{N+1}).
	ADDU	key1,key,8	1	key1 + i $\leftarrow$ LOC$(K_N + 2)$.
	STO	k,key,0	1	$K_{N+1} \leftarrow K$.
	JMP	Q2	1	
Q3	ADD	i,i,16	$\lfloor (C - S)/2 \rfloor$	<u>*Q3. Advance.*</u> (twice)

`Q2`	`LDO`	`ki,key,i`	$\lfloor (C-S)/2 \rfloor + 1$	*Q2. Compare.*
	`CMP`	`t,k,ki`	$\lfloor (C-S)/2 \rfloor + 1$	
	`BZ`	`t,Q4`	$\lfloor (C-S)/2 \rfloor + 1_{[1-F]}$	To Q4 if $K = K_i$.
	`LDO`	`ki,key1,i`	$\lfloor (C-S)/2 \rfloor + F$	*Q2. Compare.*
	`CMP`	`t,k,ki`	$\lfloor (C-S)/2 \rfloor + F$	
	`PBNZ`	`t,Q3`	$(C-S)/2 \rfloor + F_{[F]}$	To Q3 if $K \neq Ki$.
	`ADD`	`i,i,8`	F	
`Q4`	`PBN`	`i,Success`	$1_{[1-S]}$	*Q4. End of file?*
	`POP`	`0,0`		Exit if not in table.
`Success`	`ADDU`	`$0,key,i`	S	Return `LOC`(K_i).
	`POP`	`1,0`		▮

The inner loop has been duplicated; this avoids about half of the "$i \leftarrow i + 1$" instructions, so with $F = (C - S) \bmod 2$, it reduces the running time to

$$3.5C - 4.5S + 14 + \frac{(C-S) \bmod 2}{2}$$

units.

EXERCISES

[405]

3. [*16*] Write an `MMIX` program for the algorithm of exercise 2. What is the running time of your program, in terms of the quantities C and S in (1)?

5. [*20*] Program Q′ is, of course, noticeably faster than Program Q, when C is large. But are there any small values of C and S for which Program Q′ actually takes more time than Program Q?

6. [*20*] Add five more instructions to Program Q′, reducing its running time to about $(3.33C + \text{constant})\upsilon$.

6.2.1. Searching an Ordered Table

[411]

Program B (*Binary search*). As in the programs of Section 6.1, we assume here that the keys K_i are an array of octabyte values. The following subroutine expects three parameters: `key` ≡ `LOC(`K_1`)`, the location of K_1; `n` ≡ N, the number of keys; and `k` ≡ K, the given key. It returns the address of the key, if found, and zero otherwise.

:Search	SET	l,0	1	<u>*B1. Initialize.*</u> $l \leftarrow 1$.
	SUB	u,n,1	1	$u \leftarrow N$.
	JMP	B2	1	
B5	ADD	l,i,1	C_1	<u>*B5. Adjust l.*</u>
B2	CMP	t,u,l	$C + 1 - S$	<u>*B2. Get midpoint.*</u>
	BN	t,Failure	$C + 1 - S_{[1-S]}$	Jump if $u < l$.
	ADDU	i,u,l	C	
	SRU	i,i,1	C	$i \leftarrow \lfloor (u + l)/2 \rfloor$.
	SLU	t,i,3	C	<u>*B3. Compare.*</u>
	LDO	ki,key,t	C	`ki` $\leftarrow K_i$.
	CMP	t,k,ki	C	
	BP	t,B5	$C_{[C_1]}$	Jump if $K > K_i$.
	BZ	t,Success	$C_{2[S]}$	
	SUB	u,i,1	$C_2 - S$	<u>*B4. Adjust u.*</u> $u \leftarrow i - 1$.
	JMP	B2	$C_2 - S$	To B2.
Success	8ADDU	$0,i,key	S	
	POP	1,0		
Failure	POP	0,0		▮

The running time is $(11C - 3S + 7)\upsilon + C\mu$, where $C = C_1 + C_2$ is the number of comparisons made (the number of times step B3 is performed), and S = [outcome is successful].

[414]

The average running time of Program B is approximately

$$(11 \lg N - 7)\upsilon \qquad \text{for a successful search,} \qquad (5)$$
$$(11 \lg N + 7)\upsilon \qquad \text{for an unsuccessful search,}$$

if we assume that all outcomes of the search are equally likely.

. . .

Program C (*Uniform binary search*). This program does the same job as Program B, using Algorithm C. It adds a fourth parameter `j` ≡ `LOC(DELTA[1])`, the location of the auxiliary table. For convenience, this table contains offsets, scaled and decremented, ready to access the keys relative to the parameter `key`.

:Search	LDO	i,j,0	1

:Search	LDO	i,j,0	1	*C1. Initialize.* $j = 1$, $i \leftarrow$ DELTA[j].
	JMP	2F	1	
3H	BZ	t,Success	$C1_{[S]}$	Jump if $K = K_i$.
	BZ	dj,Failure	$C_1 - S_{[A]}$	Jump if DELTA[j] $= 0$.
	SUB	i,i,dj	$C_1 - S - A$	*C3. Decrease i.*
2H	ADDU	j,j,8	C	$j \leftarrow j + 1$.
	LDO	dj,j,0	C	*C2. Compare.*
	LDO	ki,key,i	C	
	CMP	t,k,ki	C	
	PBNP	t,3B	$C_{[C_2]}$	Jump if $K \le K_i$.
	ADD	i,i,dj	C_2	*C4. Increase i.*
	PBNZ	dj,2B	$C_{2[1-S-A]}$	Jump if DELTA[j] $\neq 0$.
Failure	POP	0,0		Exit if not in table.
Success	ADDU	$0,key,i	S	Return LOC(K_i).
	POP	1,0		▮

. . .

The total running time of Program C is not quite symmetrical between left and right branches, since C_2 is weighted more heavily than C_1, but exercise 11 shows that we have $K < K_i$ roughly as often as $K > K_i$; hence Program C takes approximately

$$\begin{aligned} &(8.5 \lg N - 6)\upsilon && \text{for a successful search,} \\ &(8.5\lfloor \lg N \rfloor + 12)\upsilon && \text{for an unsuccessful search.} \end{aligned} \qquad (7)$$

This is about 23 percent faster than Program B.

. . .

Program F (*Fibonaccian search*). We follow the previous conventions, with key $\equiv$ LOC(K_1) and k $\equiv K$. Instead of N, we have i $\equiv 8F_k - 8$, p $\equiv 8F_{k-1}$, and q $\equiv 8F_{k-2}$. As usual, the values are scaled by 8 and i is reduced by 8, so that it can be used directly as offset relative to key.

F4A	ADD	i,i,q	$C_2 - S - A$	*F4. Increase i.* $i \leftarrow i + q$.
	SUB	p,p,q	$C_2 - S - A$	$p \leftarrow p - q$.
	SUB	q,q,p	$C_2 - S - A$	$q \leftarrow q - p$.

04	:Search	LDO	ki,key,i	C	F2. Compare.
05		CMP	t,k,ki	C	
06		PBN	t,F3A	$C_{[C_2]}$	To F3 if $K < K_i$.
07		BZ	t,Success	$C_{2[S]}$	Exit if $K = K_i$.
08		CMP	t,p,8	$C_2 - S$	
09		PBNZ	t,F4A	$C_2 - S_{[A]}$	To F4 if $p \neq 1$.
10		POP	0,0		Exit if not in table.
11	F3A	SUB	i,i,q	C_1	F3. Decrease i. $i \leftarrow i - q$.
12		SUB	p,p,q	C_1	$p \leftarrow p - q$.
13		PBP	q,F2B	$C_{1[1-S-A]}$	Swap registers if $q > 0$.
14		POP	0,0		Exit if not in table.
15	F4B	ADD	i,i,p		(Lines 15–27 are parallel to 01–13.)
16		SUB	q,q,p		
17		SUB	p,p,q		
18	F2B	LDO	ki,key,i		
19		CMP	t,k,ki		
20		PBN	t,F3B		
21		BZ	t,Success		
22		CMP	t,q,8		
23		PBNZ	t,F4B		
24		POP	0,0		
25	F3B	SUB	i,i,p		
26		SUB	q,q,p		
27		PBP	p,:Search		
28		POP	0,0		
29	Success	ADDU	$0,key,i	S	Return LOC(K_i).
30		POP	1,0		∎

. . .

The total average running time of Program F therefore comes to approximately

$$\frac{\sqrt{5}}{5}(8\phi^{-1} + 3 + 3\phi)kv + 6v \approx (8.24 \lg N + 6)v \qquad (9)$$

for a successful search, and $(4 + 3\phi^{-1})v \approx 5.85v$ less for an unsuccessful search.

This is slightly faster than Program C, although the worst case running time (roughly 14.4 lg N) is slower.

EXERCISES

[423]

4. [*20*] If a search using Program 6.1S (sequential search) takes exactly 640 units of time, how long does it take with Program B (binary search)?

5. [*M24*] For what values of N is Program B actually *slower* than a sequential search (Program 6.1Q′) on the average, assuming that the search is successful?

10. [*21*] Explain how to write an `MMIX` program for Algorithm C containing approximately 6 lg N instructions and having a running time of about 6 lg N units.

6.2.2. Binary Tree Searching

[428]

This algorithm lends itself to a convenient machine language implementation. We may assume, for example, that the tree nodes have the form

RLINK
LLINK
KEY

(1)

followed perhaps by additional words of `INFO`. Using an `AVAIL` list for the free storage pool, as in Chapter 2, we can write the following `MMIX` program:

Program T (*Tree search and insertion*). This subroutine expects two parameters: `p`, a pointer to the root node, and `k` ≡ K, the given key. If the search is successful, it returns the location of the node found; otherwise, it returns zero. Note how the `ZSN` (zero or set if negative) instruction is used to compute the offset of the next link.

`0H`	`SET`	`p,q`	$C-1$	`P` ← `Q`.
`:Search`	`LDO`	`kp,p,KEY`	C	T2. Compare. `kp` ← `KEY(P)`.
	`CMP`	`t,k,kp`	C	
	`BZ`	`t,Success`	$C_{[S]}$	Exit if $K=$ `KEY(P)`.
	`ZSN`	`l,t,LLINK`	$C-S$	`l` ← ($K<$ `KEY(P)`)? `LLINK` : `RLINK`.
`T3`	`LDOU`	`q,p,l`	$C-S$	T3/4. Move left/right.
	`PBNZ`	`q,0B`	$C-S_{[1-S]}$	To T2 if `Q` $=\neq\Lambda$.

```
08          SET   q,:avail          1 − S    T5. Insert.
09          BZ    q,:Overflow       1 − S
10          LDOU  :avail,:avail,0   1 − S    Q ⇐ AVAIL.
11          STOU  q,p,l             1 − S    LINK(P) ← Q.
12          STCO  0,q,RLINK         1 − S    RLINK(Q) ← Λ.
13          STCO  0,q,LLINK         1 − S    LLINK(Q) ← Λ.
14          STO   k,q,KEY           1 − S    KEY(Q) ← K.
15          POP   0,0                        Exit after insertion.
16 Success  POP   1,0                        Return node address. ▮
```

The first 7 lines of this program do the search; the next 8 lines do the insertion. The running time for the searching phase is $(7C - 3S + 1)\upsilon + (2C - S)\mu$, where

C = number of comparisons made;

S = [search is successful].

This compares favorably with the binary search algorithms that use an implicit tree (see Program 6.2.1C). By duplicating the code we could effectively eliminate line 01 of Program T, reducing the running time to $(6C - 3S + 7)\upsilon$. If the search is unsuccessful, the insertion phase of the program costs an extra $7\upsilon + 5\mu$.

EXERCISES

[454]

1. [*15*] Algorithm T has been stated only for nonempty trees. What changes should be made so that it works properly for the empty tree too?

3. [*20*] In Section 6.1 we found that a slight change to the sequential search Algorithm 6.1S made it faster (Algorithm 6.1Q). Can a similar trick be used to speed up Algorithm T?

6.2.3. Balanced Trees

[464]

Program A (*Balanced tree search and insertion*). This program for Algorithm A uses tree nodes having the form

RLINK	|a
LLINK	|a .
KEY	

(4)

The balance factor a of a node is *not* stored as a field in the node itself; it is stored as a 2-bit value in two's complement format (a mod 4) in the low-order bits of the link field *pointing to* the node. (`MMIX` ignores these low-order bits of a register when using it to load or store an octabyte.) This saves one load instruction in the main loop (lines 05–12), because we can determine the balance factor of `NODE(P)` from `P` without loading `B(P)`. Further, we do not need to maintain the variable `S` inside the loop. Instead, we set `T` ← `LOC(LINK(`a`,T))`, computing a from K and `KEY(T)`, and set `S` ← `LINK(`a`,T)` right after the loop (lines 13–17). The new value of `T` is more convenient in step A7, where we modify `B(S)`, and in step A10, when we finish the new tree.

Extending the notation used in Algorithm A, we use the notation `LINK(`a`)` as a synonym for the offset `LLINK` if $a = -1$, and for `RLINK` if $a = +1$. These offsets are zero and eight, respectively, so that `MMIX` can compute `LINK(`a`)` from $a \neq 0$ with a single `ZSN` (zero or set if negative) instruction (see line 27).

The first parameter of the subroutine is `head` ≡ `LOC(HEAD)`. The second parameter, `k` ≡ K, is the key.

01	:Search	SET	t,head	1	<u>*A1. Initialize.*</u> `T` ← `HEAD`.
02		STO	k,t,KEY	1	(See line 13.)
03		LDOU	p,t,RLINK	1	`P` ← `RLINK(HEAD)`.
04		JMP	A2	1	
05	0H	CSOD	t,q,p	$C-1$	If `B(Q)` ≠ 0, T ← `LOC(LINK(`a`, P))`
06		SET	p,q	$C-1$	`P` − `Q`.
07	A2	LDO	kp,p,KEY	C	<u>*A2. Compare.*</u> `kp` ← `KEY(P)`.
08		CMP	a,k,kp	C	Compare K and `KEY(P)`; set a.
09		BZ	a,Success	$C_{[S]}$	Exit if $K =$ `KEY(P)`.
10		ZSN	la,a,LLINK	$C-S$	`la` ← `LINK(`a`)`.
11		LDOU	q,p,la	$C-S$	<u>*A3/4. Move left/right.*</u>
12		PBNZ	q,0B	$C-S_{[1-S]}$	Jump if `Q` = `LINK(`a`,P)` ≠ Λ.
13		LDOU	x,t,KEY	$1-S$	`x` ← `KEY(T)`.
14		CMP	a,k,x	$1-S$	Compare K and `KEY(T)`; set a.
15		ZSN	x,a,LLINK	$1-S$	`x` ← `LINK(`a`)`.
16		ADDU	t,t,x	$1-S$	`T` ← `LOC(LINK(`a`,T))`.

	LDOU	s,t	$1-S$	S ← LINK(a,T).
	SET	q,:avail	$1-S$	*A5. Insert.* B(Q) ← 0.
	BZ	q,:Overflow	$1-S$	
	LDOU	:avail,:avail	$1-S$	Q ⇐ AVAIL.
	STOU	q,p,la	$1-S$	LINK(a,P) ← Q.
	STCO	0,q,RLINK	$1-S$	RLINK(Q) ← Λ.
	STCO	0,q,LLINK	$1-S$	LLINK(Q) ← Λ.
	STO	k,q,KEY	$1-S$	KEY(Q) ← K.
	LDO	kp,s,KEY	$1-S$	*A6. Adjust balance factors.*
	CMP	a,k,kp	$1-S$	Compare K and KEY(S); set a.
	ZSN	la,a,LLINK	$1-S$	la ← LINK(a).
	ADDU	ll,s,la	$1-S$	ll ← LOC(LINK(a,S)).
	LDOU	p,ll	$1-S$	P ← LINK(a,S).
	JMP	0F	$1-S$	
1H	LDO	kp,p,KEY	F	kp ← KEY(P).
	CMP	c,k,kp	F	c ← K : KEY(P).
	AND	x,c,3	F	x ← c mod 4.
	OR	p,p,x	F	B(P) ← K : KEY(P).
	STOU	p,ll	F	LINK(c,P) ← P.
	ZSN	x,c,LLINK	F	x ← LINK(c).
	ADDU	ll,p,x	F	ll ← LOC(LINK(c,P)).
	LDOU	p,ll	F	P ← LINK(c,P).
0H	CMPU	x,p,q	$F+1-S$	P = Q?
	PBNZ	x,1B	$F+1-S_{[1-S]}$	Repeat until P = Q.
	AND	a,a,3	$1-S$	*A7. Balancing act.*
	AND	x,s,3	$1-S$	x ← B(S).
	BZ	x,i	$1-S_{[J]}$	If B(S) = 0, go to case (i).
	CMP	x,x,a	$1-S-J$	B(S) = a?
	BZ	x,iii	$1-S-J_{[G+H]}$	If B(S) = a, go to case (iii).
ii	ANDN	s,s,3	$1-S-J-G-H$	(ii)
	STOU	s,t	$1-S-J-G-H$	B(S) ← 0.
	POP	0,0		
i	LDO	x,head,LLINK	J	(i)
	ADD	x,x,1	J	The tree has grown higher.

	STO	x,head,LLINK	J	LLINK(HEAD) $\leftarrow$ LLINK(HEAD) $+ 1$.
	OR	s,s,a	J	
	STOU	s,t	J	B(S) $\leftarrow a$.
	POP	0,0		
iii	LDOU	r,s,la	$G + H$	(iii) R $\leftarrow$ LINK(a,S).
	NEG	lm,LLINK,la	$G + H$	lm $\leftarrow$ LINK$(-a)$.
	AND	x,r,3	$G + H$	x $\leftarrow$ B(R).
	CMP	x,a,x	$G + H$	$a =$ B(R)?
	BZ	x,A8	$G + H_{[G]}$	Go to A8 if B(R) $= 0$.
	LDOU	p,r,lm	H	_A9. Double Rotation._
	LDOU	x,p,la	H	x $\leftarrow$ LINK(a,P).
	STOU	x,r,lm	H	LINK($-a$,R) $\leftarrow$ LINK(a,P).
	AND	bp,p,3	H	bp $\leftarrow$ B(P).
	CMP	x,bp,a	H	B(P) $= a$?
	CSZ	a,x,#02	H	$a \leftarrow 1 \bmod 4$, if B(P) $= a$.
	XOR	s,s,a	H	B(S) $\leftarrow$ B(P) $= a? -$ B(S) : 0.
	CSZ	x,bp,0	H	x $\leftarrow 0$, if B(P) $= a$.
	AND	bp,r,3	H	bp $\leftarrow$ B(R).
	CSNZ	bp,x,#02	H	bp $\leftarrow -1$, if B(P) $= -a$.
	XOR	r,r,bp	H	B(R) $\leftarrow$ B(P) $= a$? $-$B(R) : 0.
	STOU	r,p,la	H	LINK(a,P) $\leftarrow$ R.
	LDOU	x,p,lm	H	x $\leftarrow$ LINK($-a$,P).
	STOU	x,s,la	H	LINK(a,S) $\leftarrow$ LINK($-a$,P).
	STOU	s,p,lm	H	LINK($-a$,P) $\leftarrow$ S.
	ANDN	p,p,3	H	B(P) $= 0$?
	STOU	p,t	H	_A10. Finishing touch._
	POP	0,0		
A8	ANDN	r,r,3	G	_A8. Single Rotation._ B(R) $\leftarrow 0$.
	ANDN	s,s,3	G	B(S) $\leftarrow 0$.
	SET	p,r	G	P $\leftarrow$ R.
	LDOU	x,r,lm	G	x $\leftarrow$ LINK($-a$,R).
	STOU	x,s,la	G	LINK(a,S) $\leftarrow$ LINK($-a$,R).
	STOU	s,r,lm	G	LINK($-a$,R) $\leftarrow$ S.
	STOU	p,t	G	_A10. Finishing touch._
	POP	0,0		
Success	SET	$0,p	S	

```
87              POP    1,0                                     ▮
```

[470]

The running time of the search phase of Program A (lines 01–12) is

$$8C - 3S + 4, \qquad (15)$$

where C and S are the same as in previous algorithms of this chapter. Empirical tests show that we may take $C + S \approx 1.01 \lg N + 0.1$, so the average search time is approximately $8.08 \lg N + 4.8 - 11S$ units. (If searching is done much more often than insertion, we could of course use a separate, faster program for searching, since it would be unnecessary to look at the balance factors. With `p` ≡ `LOC(HEAD)` and `k` ≡ K, we can write:

```
01  :Search   LDOU  p,p,RLINK    1            A1. Initialize. P ← RLINK(HEAD).
02            BZ    p,Failure    1[0]
03  A2        LDO   kp,p,KEY     C            A2. Compare. kp ← KEY(P).
04            CMP   a,k,kp       C            Compare K and KEY(P); set a.
05            BZ    a,Success    C[S]         Exit if K = KEY(P).
06            ZSN   la,a,LLINK   C − S        la ← LINK(a).
07            LDOU  p,p,la       C − S        A3/4. Move left/right. P ← LINK(a, P).
08            PBNZ  p,A2         C − S[1 − S]
09  Failure   POP   0,0                       Not found.
10  Success   POP   1,0                       ▮
```

The running time of the above code is only $(6C - 3S + 4)\upsilon + (2C - S + 1)\mu$; it reduces the average running time for a successful search to only about $(6.06 \lg N - 4.4)\upsilon$. Even the worst case running time would, in fact, be similar to the average running time obtained with Program 6.2.2T.

The running time of the insertion phase of Program A (lines 18–40) is $(10F + 22)\upsilon$, when the search is unsuccessful. The data of Table 1 indicate that $F \approx 1.8$ on the average. The rebalancing phase (lines 41–85) takes either 10, 7, 21, or 29 υ, depending on whether we increase the total height, or simply exit without rebalancing, or do a single or double rotation. The first case almost never occurs, and the others occur with the approximate probabilities .534, .233, .232, so the average running time of the combined insertion-rebalancing portion of Program A is about 55υ.

These figures indicate that maintenance of a balanced tree in memory is reasonably fast, even though the program is rather lengthy. If the input data are random, the simple tree insertion algorithm of Section 6.2.2 is roughly 48υ faster per insertion; but the balanced tree algorithm is guaranteed to be reliable even with nonrandom input data.

One way to compare Program A with Program 6.2.2T is to consider the worst case of the latter. If we study the amount of time necessary to insert N keys in increasing order into an initially empty tree, it turns out that Program A is slower for $N \le 27$ and faster for $N \ge 28$.

EXERCISES

[479]

12. [*24*] What is the maximum possible running time of Program A when the eighth node is inserted into a balanced tree? What is the minimum possible running time for this insertion?

28. [*41*] Prepare efficient implementations of 2-3 tree algorithms.

6.3. DIGITAL SEARCHING

[493]

Program T (*Trie search*). This program assumes that all keys consist of seven or less uppercase characters; keys are represented in one `OCTA`, left aligned and padded with zero bytes to the right; the rightmost byte is always zero. Since `MMIX` uses ASCII codes, each byte of the search argument is assumed to contain a value between 65 (ASCII 'A') and 90 (ASCII 'Z'). For simplicity, we use the five least significant bits of each character as index k. This allows 32 values instead of 26 and, therefore, uses more memory but simplifies the extraction of the index. Links are represented as absolute addresses, with the least significant bit set to 1 (this bit is ignored by `MMIX` when using the value to load `OCTA`s). The following subroutine expects two parameters: `p` ≡ `LOC(ROOT)`, the location of the root node, and `K` ≡ K, the given key. It returns the location of the key in the table if the search is successful and zero otherwise. To obtain successive characters from the key K, we copy it into a shift register `s`, from which we extract the leftmost character by shifting right and advance to the next character by shifting left.

```
:Start   SLU   s,K,3      1       T1. Initialize. s ← 8K.
         JMP   T2         1
T3       SET   p,x        C − 1   T3. Advance. P ← X.
         SLU   s,s,8      C − 1   Advance to next character of K.
```

```
T2       SRU   k,s,64-8     C          T2. Branch. Extract 8k.
         LDOU  x,p,k        C          X ← table entry number k in NODE(P).
         PBOD  x,T3         C[1]       If X is a link, go to T3.
         CMP   t,K,x        1          T4. Compare.
         BNZ   t,Failure    1[1 − S]   If X ≠ K, terminate unsuccessfully;
         ADDU  $0,p,k       S          else return LOC(X).
         POP   1,0
Failure  POP   0,0                     ▮
```

The running time of this program is $(5C - S + 6)\upsilon + C\mu$, where C is the number of characters examined. Since $C \le 7$, the search will never take more than 41υ.

If we now compare the efficiency of this program (using the trie of Table 1) to Program 6.2.2T (using the *optimum* binary search tree of Fig. 13), we can make the following observations.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
0		A				I					HE	
A	(2)				(10)				WAS			THAT
B	(3)											
C												
D										HAD		
E			BE		(11)							THE
F	(4)						OF					
G												
H	(5)							(12)	WHICH			
I	(6)				HIS				WITH			THIS
J												
K												
L												
M												
N	NOT	AND				IN	ON					
O	(7)			FOR				TO				
P												
Q												
R	ARE		FROM			OR				HER		
S		AS				IS						
T	(8)	AT				IT						
U			BUT									
V										HAVE		
W	(9)											
X												
Y	YOU		BY									
Z												

Table 1 A Trie for the 31 Most Common English Words

1. The trie takes much more memory space; we are using 384 octabytes just to represent 31 keys, while the binary search tree uses only 93 octabytes. (However, exercise 4 shows that, with some fiddling around, we can actually fit the trie of Table 1 into only 53 octabytes.)

2. A successful search takes about 16υ with trie search compared to 28υ with binary search. An unsuccessful search will go even faster in the trie and slower in the binary search tree. Hence, the trie is preferable from the standpoint of speed.

3. If we consider the KWIC indexing application of Fig. 15 (page *440*) instead of the 31 commonest English words, the trie loses its advantage because of the nature of the data. For example, a trie requires 12 iterations to distinguish between `COMPUTATION` and `COMPUTATIONS`. In this case it would be better to build the trie so that words are scanned from right to left instead of from left to right.

EXERCISES

[507]

4. [*21*] Most of the 384 entries in Table 1 are blank (null links). But we can compress the table into only 53 entries, by overlapping nonblank entries with blank ones as follows:

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Entry		THAT		WAS		THE	OF	HE	(12)	THIS	WHICH	WITH	(10)	BE	ON	TO	(11)	I	OR	FOR	HIS	HAD	FROM		A	HER

Position	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
Entry		(2)	(3)	BUT		IN	(4)	BY	(5)	(6)	IS	IT	AND	HAVE	NOT	(7)	ARE	AS	AT		(8)			(9)		YOU	

(Nodes (1), (2), . . . , (12) of Table 1 begin, respectively, at positions 26, 24, 8, 4, 11, 17, 0, 0, 2, 17, 7, 0 within this compressed table.)

Show that if the compressed table is substituted for Table 1, Program T will still work, but not quite as fast.

9. [*21*] Write an `MMIX` program for Algorithm D, and compare it to Program 6.2.2T. You may use the idea of exercise 8 if it helps.

6.4. HASHING

[513]

For example, let's consider again the set of 31 English words that we have subjected to various search strategies in Sections 6.2.2 and 6.3. Table 1 shows a short `MMIX` program that transforms each of the 31 keys into a unique number $f(K)$ between 0 and 39. If we compare this method to the `MMIX` programs for the other methods we have considered (for example, binary search, optimal tree search, trie memory, digital tree search), we find that it is superior from the standpoint of both space and speed, except that binary search uses slightly less space. In fact, the average time for a successful search, using the program of Table 1 with the frequency data of Fig. 12 on page *436*, is only about 13.4υ (not counting the final `POP`), and only 40 table locations are needed to store the 31 keys.

Unfortunately, such functions $f(K)$ aren't very easy to discover. There are $40^{31} \approx 10^{50}$ possible functions from a 31-element set into a 40-element set, and only $40 \cdot 39 \cdot \ldots \cdot 10 = 40!/9! \approx 10^{42}$ of them will give distinct values for each argument; thus only about one of every 100 million functions will be suitable.

[516]

For example, on the `MMIX` computer we could choose $M = 1009$ (unfortunately 2009 is not prime), computing $h(K)$ by the sequence

```
SET    m,1009
DIV    t,k,m                                   (3)
GET    h,rR       h(K) ← K mod 1009.
```

The multiplicative hashing scheme is equally easy to do, but it is slightly harder to describe because we must imagine ourselves working with fractions instead of with integers. Let w be the word size of the computer, so that w is usually 2^{32} or 2^{64} for `MMIX`; we can regard an integer A as the fraction A/w if we imagine the radix point to be at the left of the word. The method is to choose some integer constant A relatively prime to w, and to let

$$h(K) = \left\lfloor M\left(\left(\frac{A}{w}K\right) \bmod 1\right)\right\rfloor. \qquad (4)$$

In this case we usually let M be a power of 2, so that $h(K)$ consists of the leading bits of the least significant half of the product AK.

In `MMIX` code, if we let $M = 2^m$ for some small constant m and $w = 2^{64}$, the multiplicative hash function is

`MULU`	`t,a,k`	`t` $\leftarrow AK \bmod 2^{64}$.
`SRU`	`h,t,64-m`	Retain the m most significant bits. (5)

Now $h(K)$ appears in register `h`. Since `MMIX`, like many machines, has a multiplication instruction that is significantly faster than its division instruction, this sequence takes only 11 cycles to compute, compared to 62 cycles for (3).

[518]

The theory above suggests *Fibonacci hashing*, where we choose the constant A to be the nearest integer to $\phi^{-1}w$ that is relatively prime to w. For example with `MMIX`, a binary computer with $w = 2^{64}$, we would take

Contents of a after executing the instruction, given a particular key K

Instruction	A	AND	ARE	AS	AT	BE	BUT	BY	FOR	FROM	HAD	HAVE	HE	HER	HIS	I	IN	IS	IT	NOT	OF	ON	OR	THAT	THE	THIS	TO	WAS	WHICH	WITH	YOU
SET k,$0																															
LDOU a,k,0	65	65	65	65	65	66	66	66	70	70	72	72	72	72	72	73	73	73	73	78	79	79	79	84	84	84	84	87	87	87	89
SUB a,a,63	2	2	2	2	2	3	3	3	7	7	9	9	9	9	9	10	10	10	10	15	16	16	16	21	21	21	21	24	24	24	26
LDBU b,k,1	2	2	2	2	2	3	3	3	7	7	9	9	9	9	9	10	10	10	10	15	16	16	16	21	21	21	21	24	24	24	26
BZ b,9F	2	2	2	2	2	3	3	3	7	7	9	9	9	9	9	10	10	10	10	15	16	16	16	21	21	21	21	24	24	24	26
LDOU c,k,2	.	2	2	2	2	3	3	3	7	7	9	9	9	9	9	.	10	10	10	15	16	16	16	21	21	21	21	24	24	24	26
PBNZ c,1F	.	2	2	2	2	3	3	3	7	7	9	9	9	9	9	.	10	10	10	15	16	16	16	21	21	21	21	24	24	24	26
2ADDU a,a,a	.	.	.	6	6	9	.	9	.	.	.	.	27	.	.	.	30	30	30	.	48	48	48	.	.	.	63	.	.	.	.
ADD a,a,b	.	.	.	89	90	78	.	98	.	.	.	.	96	.	.	.	108	113	114	.	118	126	130	.	.	.	142	.	.	.	.
SUB a,a,75	.	.	.	14	15	3	.	23	.	.	.	.	21	.	.	.	33	38	39	.	43	51	55	.	.	.	67	.	.	.	.
SUB t,a,38	.	.	.	14	15	3	.	23	.	.	.	.	21	.	.	.	33	38	39	.	43	51	55	.	.	.	67	.	.	.	.
CSNN a,t,t	.	.	.	14	15	3	.	23	.	.	.	.	21	.	.	.	33	0	1	.	5	13	17	.	.	.	29	.	.	.	.
POP 1,0	.	.	.	14	15	3	.	23	.	.	.	.	21	.	.	.	33	0	1	.	5	13	17	.	.	.	29	.	.	.	.
1H LDBU d,k,3	.	2	2				3		7	7	9	9		9	9	.				15				21	21	21		24	24	24	26
BNZ d,1F	.	2	2				3		7	7	9	9		9	9	.				15				21	21	21		24	24	24	26
ADD a,a,c	.	70	71				87		89	.	77	.		91	92	.				99				.	90	.		107	.	.	111
SUB a,a,51	.	19	20				36		38	.	26	.		40	41	.				48				.	39	.		56	.	.	60
CMP t,a,37	.	19	20				36		38	.	26	.		40	41	.				48				.	39	.		56	.	.	60
BN t,9F	.	19	20				36		38	.	26	.		40	41	.				48				.	39	.		56	.	.	60
SUB a,a,32	.	.	.				.		6	.	.	.		8	9	.				16				.	7	.		24	.	.	28
POP 1,0	.	.	.				.		6	.	.	.		8	9	.				16				.	7	.		24	.	.	28
1H ADD a,a,d	.	.	.				.			84	.	78				.								105		104			91	96	
SUB a,a,68	.	.	.				.			18	.	12				.								39		38			25	30	
9H POP 1,0	2	19	20				36			18	26	12				10								39		38			25	30	

Table 1 Transforming a Set of Keys Into Unique Addresses

$$\begin{aligned} A &= 11400714819323198485 \\ &= (\mathtt{9E37\ 79B9\ 7F4A\ 7C15})_{16}\,. \end{aligned} \qquad (7)$$

. . .

Therefore we might do better with a multiplier like

$$A = (\texttt{9E 9E 9E 9E 9E 9E 9E 9E})_{16}$$

in place of (7); such a multiplier will separate out consecutive sequences of keys that differ in *any* character position.

[519]

A value of A can be found so that each of its bytes lies in a good range and is not too close to the values of the other bytes or their complements, for example

$$A = (\texttt{40 56 93 B4 62 46 5C 68})_{16}\,. \tag{8}$$

. . .

Program C (*Chained hash table search and insertion*). For convenience, the keys and links are assumed to be only four bytes long, and nodes are represented as follows:

KEY	LINK

. (13)

Empty nodes have a negative link field; occupied nodes have a nonnegative link field containing the offset of the next node in the chain. These offsets are all even; an odd offset is used to mark the end of the chain.

We assume a descriptor `D` for each hash table that contains the absolute address of the table and the values of M and R as follows:

TABLE	
M	R

.

The following subroutine is called with two parameters: `d` ≡ `LOC(D)`, the location of the descriptor for the hash table, and `k` ≡ K, the given key.

```
:Start  LDT   m,d,M           1          M ← M(D).
        LDOU  key,d,TABLE     1          key ← TABLE(D).
        ADDU  link,key,LINK   1          link ← TABLE(D) + LINK.
        DIV   t,k,m           1          C1. Hash.
        GET   i,:rR           1          i ← h(K) = K mod M.
        SL    i,i,3           1          Scale i. (Now 0 ≤ i < 8M.)
        LDT   t,link,i        1          C2. Is there a list?
        BN    t,C6            1[1 − A]   If TABLE[i] is empty, go to C6.
```

```
3H       LDT   t,key,i      C                t ← KEY[i].
         CMP   t,t,k        C                C3. Compare.
         PBZ   t,Success    C[C−S]           Exit if K = KEY[i].
         SET   p,i          C − S            Keep previous value of i.
         LDT   i,link,i     C − S            C4. Advance to next.
         PBEV  i,3B         C − S[A−S]       To C3 if LINK[i] is even.
         LDT   r,d,R        A − S            C5. Find empty node. R ← R(D).
5H       SUB   r,r,8        T                R ← R − 1.
         BN    r,Failure    T[0]             Exit if no empty nodes left.
         LDT   t,link,r     T                t ← LINK[R].
         BNN   t,5B         T[T−(A−S)]       Repeat until TABLE[R] empty.
         STT   r,d,R        A − S            R(D) ← R.
         STT   r,link,p     A − S            LINK[i] ← R.
         SET   i,r          A − S            i ← R.
C6       SET   t,1          1 − S            C6. Insert new key.
         STT   t,link,i     1 − S            LINK[i] ← 1. (End of chain.)
         STT   k,key,i      1 − S            KEY[i] ← K.
         POP   0,0
Success  ADDU  $0,key,i     S                Return LOC(TABLE[i]).
         POP   1,0
Failure  NEG   $0,1         0                Return −1.
         POP   1,0                           ▮
```

The running time of this program depends on

C = number of table entries probed while searching;

A = [initial probe found an occupied node];

S = [search was successful];

T = number of table entries probed while looking for an empty space.

The total running time for the searching phase of Program C is $(8C - 6S + 69)\upsilon + (2C - S + 3)\mu$ and the insertion of a new key when $S = 0$ takes an additional $(6T + 2A + 3)\upsilon + (T + 3A + 2)\mu$. The division to obtain $h(K)$ is the most expensive part of this subroutine.

Program L (*Linear probing and insertion*). This program deals with full octabyte keys; but a key of 0 is not allowed, since 0 is used to signal an empty position in the table. (Alternatively, we could require the keys to be nonnegative, letting empty positions contain -1.)

As in Program C, we assume a descriptor `D` for each hash table that contains the absolute address of the table, the value of M, and the number of vacancies, $M - 1 - N$, as follows:

TABLE	
M	VACANCIES

The following subroutine is called with two parameters: `d` ≡ `LOC(D)`, the location of the descriptor for the hash table, and `k` ≡ K, the given key.

The table size M is assumed to be prime, and `KEY[`i`]` is stored in location `TABLE(D)` + $8i$ for $0 \le i < M$. For speed in the inner loop, location `TABLE(D)` − 8 is assumed to contain 0, and the test "$i < 0$" has been removed from the loop so that only the essential parts of steps L2 and L3 remain. The total running time for the searching phase comes to $(7C + 6E + 2S + 62)\upsilon + (C + E + 2)\mu$, and the insertion after an unsuccessful search adds an extra $5\upsilon + 3\mu$.

:Start	LDO	m,d,M	1	$M \leftarrow$ `M(D)`.
	LDOU	key,d,TABLE	1	`key` $\leftarrow$ `TABLE(D)`.
	DIV	t,k,m	1	<u>*L1. Hash.*</u>
	GET	i,:rR	1	$i \leftarrow K \bmod M$.
	SL	i,i,3	1	$i \leftarrow 8i$.
	JMP	L2	1	
L3	SL	i,m,3	E	<u>*L3. Advance to next.*</u>
L3B	SUB	i,i,8	$C + E - 1$	$i \leftarrow i - 1$.
L2	LDO	ki,key,i	$C + E$	<u>*L2. Compare.*</u>
	CMP	t,ki,k	$C + E$	`KEY`$[i] = K$?
	BZ	t,Success	$C + E_{[S]}$	Exit if `KEY`$[i] = K$.
	BNZ	ki,L3B	$C + E - S_{[C-1]}$	To L3 if `TABLE`$[i]$ nonempty.
	BN	i,L3	$E + 1\ S_{[E]}$	To L3 with $i \leftarrow M$ if $i < 0$.
	LDO	t,d,VACANCIES	$1 - S$	<u>*L4. Insert.*</u> `t` $\leftarrow$ `VACANCIES(D)`.
	BZ	t,Failure	$1 - S_{[0]}$	Exit with overflow if $N = M - 1$.
	SUB	t,t,1	$1 - S$	Increase N by 1.
	STO	t,d,VACANCIES	$1 - S$	
	STO	k,key,i	$1 - S$	`KEY`$[i] \leftarrow K$.
	POP	0,0		
Success	ADDU	$0,key,i	S	Return `LOC(KEY`$[i]$`)`.
	POP	1,0		
Failure	NEG	$0,1	0	Return −1.

	POP	1,0		▮

[529]

If $M = 2^m$ and we are using multiplicative hashing, $h_2(K)$ can be computed simply by shifting left m more bits and “oring in“ a 1, so that the coding sequence in (5) would be followed by

SLU	h2,t,m	Shift $AK \bmod 2^{64}$ left m more bits.	
SRU	h2,h2,64-m	Retain the m most significant bits.	(24)
OR	h2,h2,1	$h_2 \leftarrow h_2 \mid 1$.	

This is faster than the division method.

[530]

Algorithms L and D are very similar, yet there are enough differences that it is instructive to compare the running time of the corresponding MMIX programs.

Program D (*Open addressing with double hashing*). This program is substantially like Program L, except that no zero value is assumed in location TABLE(D) − 8.

:Start	LDO	m,d,M	1	$M \leftarrow$ M(D).
	LDOU	key,d,TABLE	1	key ← TABLE(D).
	DIV	q,k,m	1	D1. First hash.
	GET	i,:rR	1	$i \leftarrow h_1(K) = K \bmod M$.
	SL	i,i,3	1	$i \leftarrow 8i$.
	LDO	ki,key,i	1	D2. First probe.
	CMP	t,ki,k	1	KEY$[i] = K$?
	PBZ	t,Success	$1_{[1-S_1]}$	Exit if KEY$[i] = K$.
	PBZ	ki,D6	$1-S_{1[A-S_1]}$	To D6 if TABLE $[i]$ is empty.
	SUB	t,m,2	$A-S_1$	D3. Second hash.
	DIV	t,k,t	$A-S_1$	
	GET	c,:rR	$A-S_1$	$c \leftarrow K \bmod (M-2)$.
	8ADDU	c,c,8	$A-S_1$	$c \leftarrow 1 + (K \bmod (M-2))$.
D4	SUB	i,i,c	$C-1$	D4. Advance to next. $i \leftarrow i - c$.
	8ADDU	t,m,i	$C-1$	t ← i + $8M$.
	CSN	i,i,t	$C-1$	If $i < 0$, then $i \leftarrow i + M$.
	LDO	ki,key,i	$C-1$	D5. Compare.
	CMP	t,ki,k	$C-1$	KEY$[i] = K$?

19		PBZ	t,Success	$C - 1_{[C-1-S_2]}$	Exit if KEY[i] $= K$.
20		BNZ	ki,D4	$C - 1 - S_{2[C-1-A+S_1]}$	To D4 if nonempty.
21	D6	LDO	t,d,VACANCIES	$1 - S$	<u>*D6. Insert.*</u> t ← VACANCIES(D).
22		BZ	t,Failure	$1 - S_{[0]}$	Overflow if $N = M - 1$.
23		SUB	t,t,1	$1 - S$	Increase N by 1.
24		STO	t,d,VACANCIES	$1 - S$	VACANCIES(D) ← $M - 1 - N$.
25		STO	k,key,i	$1 - S$	KEY[i] ← K.
26		POP	0,0		
27	Success	ADDU	$0,key,i	S	Return LOC(KEY[i]).
28		POP	1,0		
29	Failure	NEG	$0,1	0	Return −1.
30		POP	1,0		▮

The frequency counts A, C, and $S = S_1 + S_2$ in this program have a similar interpretation to those in Program C above.

[531]

Since each probe takes less time in Algorithm L, double hashing is advantageous only when the table gets full. Figure 42 compares the average running time of Program L, Program D, and a modified Program D that involves secondary clustering, replacing the rather slow calculation of $h_2(K)$ in lines 10–13 by the following three instructions:

SL	t,m,3	$t \leftarrow 8M$.	
SUB	c,t,i	$c \leftarrow M - i$.	(30)
CSZ	c,i,8	If $i = 0$, $c \leftarrow 1$.	

Program D takes a total of $11C + 63(A - S1) - 7S + 64$ units of time; modification (30) saves $60(A - S1) \approx 30\alpha$ of these in a successful search. In this case, secondary clustering is preferable to independent double hashing.

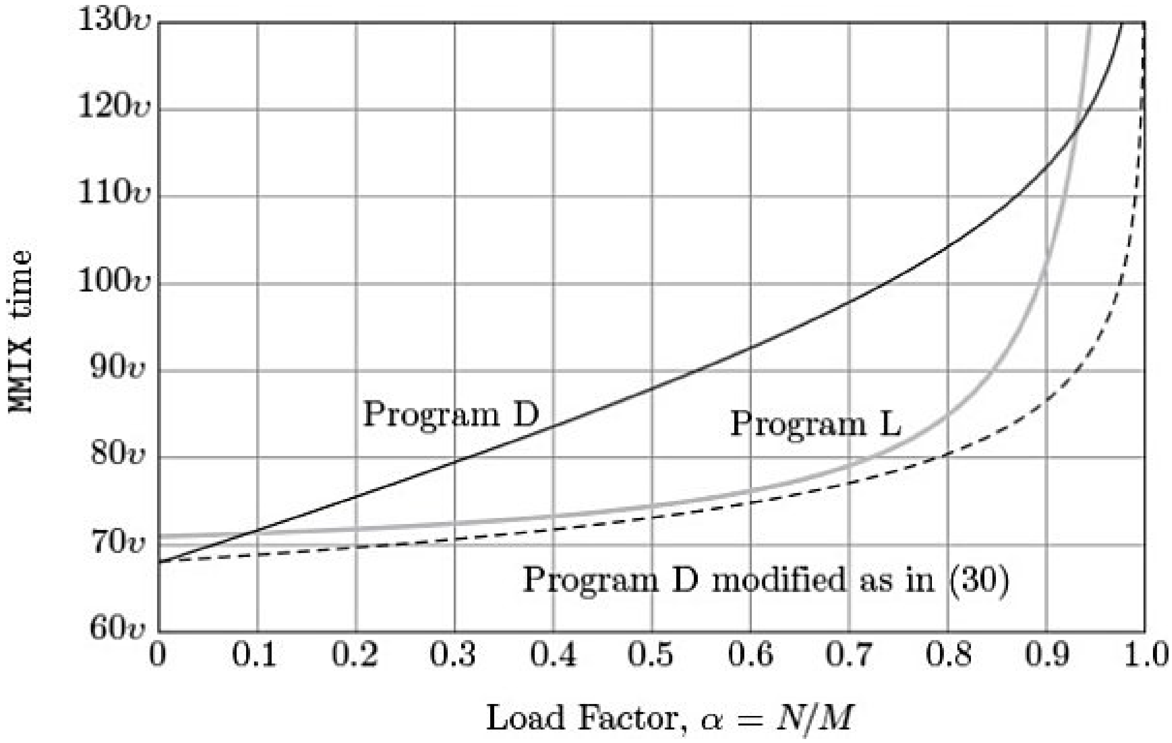


Fig. 42. The running time for successful searching by three open addressing schemes.

On a binary computer, we can speed up the computation of $h_2(K)$ in another way, if M is a prime greater than, say, 512, replacing lines 10–13 by

```
AND     t,q,511      t ← ⌊K/M⌋ mod 512.
8ADDU   c,t,8        c ← ⌊K/M⌋ mod 512 + 1 (scaled).        (31)
```

EXERCISES

[549]

1. [*20*] When one of the `POP 1,0` instructions in Table 1 is reached, how small and how large can the return value in `a ≡ $0` possibly be, assuming that bytes 1, 2, 3, and 4 of K each contain ASCII codes for uppercase alphabetic characters?

2. [*20*] Find a reasonably common English word not in Table 1 that could be added to that table without changing the program.

3. [*23*] Explain why no program beginning with the seven instructions

```
SET     k,$0
LDBU    a,k,0
ADD     a,a,x       or      SUB a,a,x
LDBU    b,k,1
```

```
ADD     a,a,b       or      SUB a,a,b
LDBU    c,k,2
BZ      c,9F
```

could be used in place of the more complicated program in Table 1, for any constant x, since unique addresses would not be produced for the given keys.

5. [*15*] Mr. B. C. Dull was writing a FORTRAN compiler using an `MMIX` computer, and he needed a symbol table to keep track of the names of variables in the FORTRAN program being compiled. These names were restricted to be at most 31 characters in length. He decided to use a hash table with $M = 256$, and to use the fast hash function $h(K)$ = leftmost byte of K. Was this a good idea?

6. [*15*] Would it be wise to change the second instruction of (3) from '`DIV t,k,m`' to '`PUT rD,k; SET z,0; DIVU t,z,m`'?

[551]

12. [*21*] Show that Program C can be rewritten so that there is only one conditional jump instruction in the inner loop. Compare the running time of the modified program with the original.

[557]

72. [*M28*] . . .

b) Suppose each h_j in (9) is a randomly chosen mapping from the set of all characters to the set $\{0, 1, \ldots, M-1\}$. Show that this corresponds to a universal family of hash functions.

Write an `MMIX` program to compute such a hash function. Assume that $K = x_1 x_2 \ldots x_8$ is a full octabyte key consisting of eight `BYTE` values and that M is a power of 2, so that you can avoid the division in (9) as suggested in the text. Compare the average running time to the running time of Program L, Program D, and the modified Program D as shown in Fig. 42.

ANSWERS TO EXERCISES

1.3.2. The MMIX Assembly Language

With three exceptions, the exercises of this section haven been revised in Fascicle 1. Here we give solutions to exercises 14, 18, and 22, which are numbered 32, 21, and 29 in Fascicle 1.

[516]

14. The following subroutine has one parameter, the year, and two return values, the day and the month. The printing is left to a driver that is not shown here. A basic implementation is easy to obtain. The following solution uses multiplication instead of division (see exercise 1.3.1´–19), cutting the running time from approximately 337υ down to 122υ. Further improvements are possible. The multiplication by 19 can be achieved in two cycles using `2ADDU` and `16ADDU`; similarly, multiplication by 7 can be done with `NEG` and `8ADDU`; and multiplication by 30 needs three cycles using `SL`, `NEG` and `2ADDU`.

```
1H       GREG   970881267037344822         2^64/19 + 2/19
:Easter  MULU   t,y,1B; GET t,:rH          E1. Golden number.
         MUL    t,t,19
         SUB    g,y,t
         ADD    g,g,1                      G ← Y mod 19 + 1.
1H       GREG   184467440737095517         2^64/100 + 84/100
         MULU   t,y,1B; GET t,:rH          E2. Century.
         ADD    c,t,1                      C ← ⌊Y/100⌋ + 1.
         2ADDU  x,c,c                      E3. Corrections.
         SRU    x,x,2
         SUB    x,x,12                     X ← ⌊3C/4⌋ − 12.
         8ADDU  z,c,5
1H       GREG   737869762948382065         2^64/25 − 9/25
         MULU   t,z,1B; GET z,:rH
         SUB    z,z,5                      z ← ⌊(8C + 5)/25⌋ − 5.
         4ADDU  d,y,y                      E4. Find Sunday.
         SRU    d,d,2
         SUB    d,d,x
         SUB    d,d,10                     D ← ⌊5Y/4⌋ − X − 10.
         2ADDU  e,g,g                      E5. Epact.
         8ADDU  e,g,e
```

```
            8ADDU  e,g,e
            ADD    e,e,20
            ADD    e,e,z
            SUB    e,e,x
1H          GREG   614891469123651721       2^64/30 − 14/30
            MULU   t,e,1B; GET t,:rH
            MUL    t,t,30
            SUB    e,e,t                    E ← (11G + 20 + Z − X) mod 30.
            CMP    t,e,25
            BNZ    t,1F
            CMP    t,g,11
            ZSP    t,t,1                    t ← G > 11.
            JMP    2F
1H          CMP    t,e,24
            ZSZ    t,t,1                    t ← E = 24.
2H          ADD    e,e,t                    Increase E if needed.
            NEG    n,44,e                   E6. Find full moon. N ← 44 − E.
            CMP    t,n,21
            ZSN    t,t,30
            ADD    n,n,t                    N ← N + 30 if N < 21.
            ADD    t,d,n                    E7. Advance to Sunday.
1H          GREG   2635249153387078803      2^64/7 + 5/7
            MULU   t+1,t,1B; GET t+1,:rH
            MUL    t+1,t+1,7
            SUB    t,t,t+1
            ADD    n,n,7
            SUB    n,n,t                    N ← N + 7 − (D + N) mod 7.
            CMP    t,n,31                   E8. Get month.
            BNP    t,1F                     If N > 31,
            SUB    $1,n,31                     return N − 31
            SET    $0,4                        and April.
            POP    2,0
1H          SET    $1,n                     Else return N
            SET    $0,3                        and March.
            POP    2,0                      ▮
```

18. For each value of $k \geq 1$, we maintain the three values x_{k-1}, x_k, and x_{k+1} in registers `xp` (previous), `xk`, and `xn` (next), respectively; we follow a similar pattern for the y-values. Advancing k therefore needs four `SET` instructions, which could be eliminated by unrolling the loop.

```
x       IS    $0        } Parameter
y       IS    $1        }
n       IS    $2        }
k       IS    $3        k scaled by 4
xn      IS    $4        x_{k+1}
yn      IS    $5        y_{k+1}
xk      IS    $6        x_k
yk      IS    $7        y_k
xp      IS    $8        x_{k-1}
yp      IS    $9        y_{k-1}
f       IS    $10       ⌊(y_{k-1} + n)/y_k⌋
t       IS    $11

:Farey  SET   k,4       k ← 1.
        SET   xp,0      x_{k-1} ← 0.
```

```
            SET   yp,1        y_{k-1} ← 1.
            STT   xp,x,0      Store x_{k-1}.
            STT   yp,y,0      Store y_{k-1}.
            SET   xk,1        x_k ← 1.
            SET   yk,n        y_k ← n.
            JMP   1F
Loop        ADD   t,yp,n
            DIV   f,t,yk      f ← ⌊(y_{k-1} + n)/y_k⌋.
            MUL   t,f,xk
            SUB   xn,t,xp     x_{k+1} ← f · x_k − x_{k-1}.
            MUL   t,f,yk
            SUB   yn,t,yp     y_{k+1} ← f · y_k − y_{k-1}.
            ADD   k,k,4       Advance k.
            SET   xp,xk       Advance xp.
            SET   xk,xn       Advance xk.
            SET   yp,yk       Advance yp.
            SET   yk,yn       Advance yk.
1H          STT   xk,x,k      Store x_k.
            STT   yk,y,k      Store y_k.
            CMP   t,xk,yk     Test if x_k < y_k.
            PBN   t,Loop      If so, continue.
            POP   0,0         Exit from subroutine. ▌
```

22. For $n = 24$ and $m = 11$, the last man is found after 913υ in position 15.

```
:Josephus   SET   i,n      1
            SET   t,0      1
            JMP   1F       1
0H          STBU  t,x,i    N                     Set each cell to the
            SET   t,i      N                        number of the next man
1H          SUB   i,i,1    N+1                      in the sequence.
            PBNN  i,0B     N+1[1]
            SET   e,1      1                     Set execution count.
            SET   p,0      1                     Start with the first man.
0H          SUB   i,m,3    N−1                   Count around the circle.
1H          LDBU  p,x,p    (M−3)(N−1)
            SUB   i,i,1    (M−3)(N−1)
            PBP   i,1B     (M−3)(N−1)[N−1]
            LDBU  l,x,p    N−1                   lucky man
```

```
        LDBU  d,x,l        N-1        doomed man
        LDBU  p,x,d        N-1        next man
        STBU  p,x,l        N-1        Take man from circle.
        STBU  e,x,d        N-1        Store execution count.
        ADD   e,e,1        N-1        Increment execution count.
        CMP   t,e,n        N-1        How many left?
        PBN   t,0B         N-1[1]
        STBU  e,x,l        1          One man left; he is clobbered too.
        POP   0,0                     ▮
```

The total running time is $(3(N-1)(M+2)+16)\upsilon + ((N-1)(M+3)+2)\mu$. An asymptotically faster method appears in exercise 5.1.1–5.

1.3.3. Applications to Permutations

[522]

7. With some formatting characters as shown on page 1, we have $X = 34$, $Y = 29$, $M = 5$, $N = 7$, $U = 3$. Total, by Eq. (18), 2095υ. Without any formatting characters, we have $X = 29$, $Y = 29$, $M = 5$, $N = 7$, $U = 3$, $V = 1$. Total, by Eq. (18), 1805υ.

9. No. For example, given (6) as input, Program A will produce '`(ADG)(CEB)`' as output, while Program B produces '`(ADG)(BCE)`'. The answers are equivalent but not identical, due to the nonuniqueness of cycle notation. The first element chosen for a cycle is the leftmost available name, in the case of Program A, and the first character in the order given by the ASCII code, in Program B.

10. (1) Kirchhoff's law yields $D = B$, $E = D+1$ (assuming that there are no formatting characters in the input), and $F = K$. (2) Interpretations: $F = A =$ `#80` $-$ `#21` $= 95$ is the size of table T; $B =$ number of characters in the input $= X$; $B - C =$ number of cycles in the input $= M$; $G =$ number of distinct elements in the output $= N$; $H = J =$ number of cycles in the output (not counting singletons) $= U - V$. (3) Summing up, we have $(10A + 13X + 10N - 3M + 9(U - V) + 14)\upsilon$, where A is the size of table T. This is better than Program A. Even for a table T that is far too large for the simple input (6), the time is still only 1439υ and without any formatting symbols 1404υ.

1.4.4. Input and Output

1. The code in (1) has two protected code sequences that allow access to the buffer. Each code sequence starts with a wait loop to acquire access rights and ends with a store instruction to release the access rights. Let's assume that the system is initially in a valid state: The consumer is using the buffer, the octabyte `S` has the value 1, and the producer is not using the buffer. There is only one instruction that can change the value of `S` from 1 to 0; to execute this instruction, the consumer has to exit the protected code segment. Using real hardware, it might take some time until the change in the value of `S` becomes visible to the producer, but the change will be immediately visible to the consumer itself. A load following a store on the same memory location and within the same thread will always return the value just stored. Therefore, the consumer will not be able to reenter the protected code segment but will get caught in the waiting loop. Eventually, the producer will notice the value 1 in octabyte `S` and can enter the protected code. The new situation is symmetric to the initial situation and the same reasoning applies. (See also 7.2.2.2–(43).) The '`SYNC 1`' instruction in the producer is not needed to protect `S`; it is needed to protect the buffer. Without it, the consumer could see the change in `S`, but still miss recent changes to the buffer made by the producer before changing `S`.

2.

```
:Producer  LDA   s,:S2            Initialize s ← LOC(S2).
0H         LDO   t,s,0            Acquire.
           BNZ   t,0B             Wait.
           LDO   buffer,s,16      Update buffer.
           LDA   $255,:InArgs     Load argument for Fgets.
           STOU  buffer,$255      Point InArgs to the buffer.
           TRAP  0,:Fgets,:StdIn  Read one line.
           BN    $255,EOF         Jump if error or end of le.
           SYNC  1                Synchronize.
           STCO  1,s,0            Release.
           LDO   s,s,16           Advance to next buffer.
           JMP   0B               Repeat.
```

3. In order to decide if the current character is the last character of the buffer, we need to look ahead to the next character in the buffer. For efficiency, we use an additional global register `c`, initially set to zero, to hold the look-ahead character.

```
1H        STCO  0,s,0        Release.
          LDO   s,s,16       Switch to next buffer.
2H        LDO   t,s,0        Acquire.
          BZ    t,2B         Wait.
          LDO   buffer,s,8   Update buffer.
          SET   i,0          Initialize i ← 0.
          SYNC  2            Synchronize.
          LDB   c,buffer,i   Load first byte.
          BZ    c,1B         If zero, advance to next buffer.
:GetByte  BZ    c,2B         Jump if look-ahead is zero.
          SET   $0,c         Prepare to return c.
          ADD   i,i,1        Advance to next byte.
          LDB   c,buffer,i   Load next byte.
          BNZ   c,0F         Jump if not end of buffer.
          STCO  0,s,0        Release.
0H        POP   1,0          Return byte. ▮
```

6.

```
Buffer1    OCTA    0             Empty buffer.
           LOC     Buffer1+SIZE
Buffer2    OCTA    0
           LOC     Buffer2+SIZE
           ...
           PREFIX  :Consumer:
buffer     GREG    0
i          GREG    0
s          GREG    0
t          IS      $0
:Consumer  LDA     s,:S1         Initialize s ← LOC(S1).
           LDOU    buffer,s,8    Initialize buffer.
           NEG     i,1           Initialize i ← -1.
           PUSHJ   t,:GetByte
           ...
```

7. With a single producer thread, there is no need for another semaphore. In Program A, delete the instructions of lines 03–07, 12, and 16–17; then replace `Green` by `Red` and `NEXTG` by `NEXTR`. For Program R, it is sufficient to insert ‘`SYNC 1`’ at the beginning and then replace `Red` by `Green`.

12. We define `Red` ≡ 0, `Purple` ≡ 1, `Green` ≡ 2, and `Yellow` ≡ 3. With these settings, no changes are necessary for the consumers. For the producers, in Program A replace `GS` by `RS`, `NEXTG` by `NEXTR`, `Green` by `Red`, and `Yellow` by `Purple`; and in Program R insert '`SYNC 1`' at the beginning and then replace `Red` by `Green`.

13. One invariant of the buffer ring is that all red or yellow buffers follow all green and purple buffers, and vice versa. This invariant ensures that all buffers are consumed in the same order as they are produced. So a single consumer that needs more time than usual can delay all producers, waiting for its yellow buffer to turn red, even if there are many red buffers following the yellow buffer. If the situation lasts long enough, the other consumers must also wait because no new red buffers have been produced. Because of symmetry, the same can happen with a slow producer. If the time a consumer or producer needs for a buffer varies greatly, it might be more efficient to process buffers out of order; in this case, maintaining separate linked lists for buffers of "different color" can be more efficient.

15. The thread that sets the semaphore to 1 does not only earn the right to modify the protected data: It earns the *exclusive* right to do so, preventing all other threads from making modifications. The thread executing the "improved" code loads `NEXTG` into register `s` before it sets the green semaphore to 1; so by the time the semaphore is 1, another thread might have modified `NEXTG`. In this case, `s` is pointing to the wrong buffer, which might not even be green any more. If Mr. Dull wants to wait for a green buffer first, he has to repeat the wait loop after setting the semaphore to 1, just as Program A does. It still might be an improvement. A `CSWAP` instruction might need to synchronize multiple distributed caches of multiple processors to gain exclusive and atomic access to the semaphore. So one processor executing a `CSWAP` instruction can reduce the performance of all other processors. But of course, it is much better to allocate sufficient buffers so that `NEXTG` almost always points to a green buffer.

2.1. INTRODUCTION

[535]

7. Sequence (a) loads the address of `TOP` to `t` and then the contents of `t` + `SUIT`; so we have `t` ← `SUIT(LOC(TOP))`. Sequence (b) loads the address of `TOP` + `SUIT` to `t` and then the contents of `t` + 0; so again we have `t` ← `SUIT(LOC(TOP))`. Sequence (c) is correct. There is no need for confusion; consider the analogous example when `x` is the `MMIXAL` label of a numeric variable x: To bring the value of x into register `t`, we write '`LDO t,x`', not '`LDA t,x`', since the latter brings

LOC(x) into the register (namely, the value of the label).

8. With registers x and n we can write:

```
         SET   n,0; LDOU x,TOP          B1. N ← 0, X ← TOP.
         JMP   B2
B3       ADD   n,n,1; LDOU x,x,NEXT     B3. N ← N + 1, X ← NEXT(X).
B2       PBNZ  x,B3                     B2. If X = Λ, stop. ▮
```

9. The following subroutine takes a pointer to the starting card in the pile as a parameter and prints the card names on StdOut.

```
         LOC    Data_Segment
         GREG   @
String   OCTA   0             8-byte string
         BYTE   0               with a terminating zero byte
         LOC    #100
         PREFIX :PrintPile:
x        IS     $0            The parameter
card     IS     $1            ⎫
down     IS     $2            ⎪
up       IS     $3            ⎬ Local variables
title    IS     $4            ⎪
t        IS     $5            ⎭
NL       IS     #0a           The ASCII newline character
NEXT     IS     0             Offset of NEXT
CARD     IS     8             Offset of TAG, SUIT, RANK, and TITLE
```

```
:PrintPile SETH   t,#FF00
           ORL    t,#FFFF            t ← #FF0000000000FFFF.
           PUT    :rM,t              Move t to the mask register.
           SETH   down,'('<<8
           ORL    down,(')'<<8)+NL   down ← '(',0,0,0,0,0,')',NL.
           SETH   up,' '<<8
           ORL    up,NL<<8           up ← ' ',0,0,0,0,0,NL,0.
           JMP    2F                 Start the loop.
1H         LDOU   card,x,CARD        Load TAG, SUIT, RANK, and TITLE.
           SLU    title,card,16      Position TITLE(X) after '(' or '␣'.
           SET    t,up               Assume face up.
           CSN    t,card,down        If sign bit in TAG is set, it's face down.
           MUX    title,t,title      Combine up or down with title.
           LDA    $255,:String       Get address of String.
           STOU   title,$255         Store title in String.
           TRAP   0,:Fputs,:StdOut   Print it.
           LDOU   x,x,NEXT           Set X ← NEXT(X).
2H         PBNZ   x,1B               Continue until reaching the end.
           POP    0,0                Return from subroutine. ▌
```

2.2.2. Sequential Allocation

[540]

3. Left side: The instruction `LDA base,L`$_0$ is assembled as `ADDUI base,b,`c for some suitable constant $0 \le c < 256$ with $c \bmod 8 = 0$ and base register `b` determined by the assembler. If register `i` is, for example, register `$2`, the instruction `LDOI a,b,`$c + 2$ will do the job.

Right side: Again the assembler will choose a constant c and base register `b` as before to assemble the instruction `LDOU base,BASE` as `LDOUI base,b,`c. Hence we can replace the three instructions in (8) by `LDOU a,b,`$c + 4$ provided the octabyte at location `BASE` (ordinarily a multiple of 8) is incremented by 2 to specify register `$2` as the index register to be used. The left side might take $1\upsilon + 1\mu$ instead of $3\upsilon + 1\mu$ as in (8), while the right side will take $2\upsilon + 2\mu$ instead of $3\upsilon + 2\mu$.

4. Assuming that register `j` is `$1`, register `i` is `$2`, `LOC(X)` $=$ `b` $+ c$, and the addresses stored in `X`, `X` $+ 8$, `X`$+16, \ldots$ are incremented by 2 to specify register `$2` as index register, we can simply write `LDO a,b,`$c + 4 + 1$.

5. A multiple-level `LDOU` instruction will cost as much μ and υ as the written-out sequence of ordinary `LDOU` instructions, except that the implicit scaling of the index registers might save some execution time. But a pipelined RISC machine, such as `MMIX`, can easily execute the scaling in parallel with the loading because there is no data dependency between index and loaded value. Further, as many implementations in this booklet attest, the shift instructions to scale index registers can be entirely eliminated at least from critical loops.

By comparison, automatic scaling or an extension as proposed in exercise 3 will make special use of these precious low-order bits, preventing their use as tag bits (as shown later in this chapter).

The whole concept is of limited use, because the available bits in an instruction are severely limited such that only 3 bits remain to specify an index register. If complex operations need to be specified for a RISC processor, we can use multiple short instructions instead of one long instruction. The concept of a pointer specifying an index register in its low-order bits moves information that is normally part of the code into the data. Again this goes against the concept of pipelined RISC processors, where data dependencies can prevent parallel and speculative execution of code.

In summary, such an extension violates the principles of RISC processor design, is of limited use, and does not offer true advantages on pipelined processors. There is no need to implement it.

2.2.3. Linked Allocation

[545]

2. As an example, we show the full code of the subroutine `Insert`.

```
        PREFIX  :Insert:
t       IS      $0            LOC(T)               } Parameters
y       IS      $1            The INFO             }
p       IS      $2            Pointer to node      } Local variables
x       IS      $3            Temporary variable   }
LINK    IS      0             Offset of the LINK field
INFO    IS      8             Offset of the INFO field
:Insert SET     p,:avail      P ← AVAIL.
        BZ      p,:Overflow   Is AVAIL = Λ?
        LDOU    :avail,p,LINK AVAIL ← LINK(P).
        STO     y,p,INFO      INFO(P) ← Y.
        LDOU    x,t
        STOU    x,p,LINK      LINK(P) ← T.
        STOU    p,t           T ← P.
        POP     0,0           Return. ▮
```

3. The `Delete` subroutine is similar. Notice that it has separate exits for success and failure.

```
        PREFIX  :Delete:
t       IS      $0            First parameter
p       IS      $1            Local variable
x       IS      $2            Temporary variable
LINK    IS      0             Offset of the LINK field
INFO    IS      8             Offset of the INFO field
:Delete LDOU    p,t           P ← T.
        BZ      p,1F          Is T = Λ?
        LDOU    x,p,LINK
        STOU    x,t           T ← LINK(P).
        LDO     $0,p,INFO     y ← INFO(P).
        STOU    :avail,p,LINK LINK(P) ← AVAIL.
        SET     :avail,p      AVAIL ← P.
        POP     1,1           Successful (second) exit
1H      POP     0,0           Unsuccessful (first) exit ▮
```

4. The `Allocate` subroutine uses a different way to signal errors. It “returns” zero using the instruction `POP 0,0`, making the return register marginal.

```
        PREFIX  :Allocate:
x       IS      $0            The return value
t       IS      $1            Local variable
```

```
c          IS    16                   The node size
LINK       IS    0                    Offset of the LINK field
:Allocate  SET   x,:avail             X ← AVAIL.
           BZ    x,1F                 Is AVAIL = Λ?
           LDOU  :avail,:avail,LINK   AVAIL ← LINK(AVAIL).
0H         POP   1,0                  Return X.
1H         SET   x,:poolmax           X ← POOLMAX.
           ADDU  :poolmax,:poolmax,c  POOLMAX ← X + c.
           CMPU  t,:poolmax,:seqmin   Is POOLMAX > SEQMIN?
           PBNP  t,0B                 If not, return X.
Overflow   ...                        Try to recover; if all fails,
           POP   0,0                      return zero. ▮
```

8. Here and in the following, we will not show the definition of register names, such as ‘`p IS $1`’, that are irrelevant for an understanding of the code.

```
:Revert  LDO   p,first     1       I1. P ← FIRST.
         BZ    p,2F        1[0]    I2. If the list is empty, jump.
         SET   q,0         1       Q ← Λ.
1H       SET   r,q         n       R ← Q.
         SET   q,p         n       Q ← P.
         LDOU  p,q,LINK    n       P ← LINK(Q).
         STOU  r,q,LINK    n       LINK(Q) ← R.
         PBNZ  p,1B        n[1]    Is P ≠ Λ?
         STOU  q,first     1       I3. FIRST ← Q.
2H       POP   0,0                 ▮
```

For a nonempty list, the time is $(5n+6)\upsilon+(2n+2)\mu$ (not counting the call overhead). Better speed ($3n\upsilon+2n\mu$ + constant) is attainable; see exercise 1.1–3.

22. To make the program “fail-safe” we should (a) check that $0 < n <$ some appropriate maximum; (b) check each relation $j \prec k$ for the conditions $0 < j,\ k \le n$ and check the initial zero in the first pair $(0, n)$ and the final zero in the last pair $(0, 0)$; (c) check that `avail` does not get too large.

24. Insert four lines in the program of the text:

```
51a   SL    k,n,3       Prepare for T9: k ← n.
58a   SET   t,0
58b   STTU  t,top,f     TOP[F] ← 0.
76a   BNZ   n,T9        Jump if N ≠ 0.
```

Add the following at the end of Program T:

```
T9    GET    rJ,:rJ
      GETA   $255,Msg
      TRAP   0,:Fputs,:StdErr    Print indication of loop.
      SET    t,0                 t ← 0.
1H    LDTU   p,top,k             P ← TOP[k].
      STT    t,top,k             TOP[k] ← 0.
T10   BZ     p,0F                Resume T9 if P = Λ.
      LDT    t,suc,p
      STT    k,qlink,t           QLINK[SUC(P)] ← k.
      LDT    p,next,p            P ← NEXT(P).
      BNZ    p,T10               Is P = Λ?
0H    SUB    k,k,8               k ← k + 1.
      BP     k,1B                Repeat while k > 0.
T11   ADD    k,k,8               k ← k + 1.
      LDT    t,qlink,k
      BZ     t,T11               Find k with QLINK[k] ≠ 0.
T12   STT    k,top,k             TOP[k] ← k.
      LDT    k,qlink,k           k ← QLINK[k].
      LDT    t,top,k
      BZ     t,T12               Repeat if TOP[k] = 0.
T13   SR     t+1,k,3             Scale back.
      PUSHJ  t,:Println          Assume this prints k on StdErr.
      LDT    t,top,k
      BZ     t,1F                Stop when TOP[k] = 0.
      SET    t,0
      STT    t,top,k             TOP[k] ← 0.
      LDT    k,qlink,k           k ← QLINK[k].
      JMP    T13
1H    PUT    :rJ,rJ
      POP    0,0                 Return.
Msg   BYTE   "Loop detected"
      BYTE   " in input:",#a,0   ■
```

Note: If the relations $9 \prec 1$ and $6 \prec 9$ are added to the data (18), this program will print "1, 9, 6, 4, 7, 3, 1" as the loop.

26. One solution is to proceed in two phases as follows:

Phase 1. (We use the `X`-table as a (sequential) stack as we mark each subroutine that needs to be used by setting `SPACE` ← −`SPACE`.)

`A0.` For $0 \le i <$ `N` set `SPACE(SUB[i])` ← −`SPACE(SUB[i])`.

`A1.` If `N` = `0`, go to phase 2; otherwise set $i \leftarrow 0$, decrease `N` by 1, and set `Q` ← $\texttt{LINK}_i$`(SUB[N])`.

`A2.` If `Q` is odd, go to A1.

`A3.` Set $i \leftarrow i + 1$ and `Q` ← $\texttt{LINK}_i$`(SUB[N])`. If `SPACE(Q)` ≥ 0, set `SPACE(Q)` ← −`SPACE(Q)`, `SUB[N]` ← `Q`, and set `N` ← `N` + 1. Now return to A2. ▮

Phase 2. (We go through the table and allocate memory.)

`B1.` Set `P` ← `FIRST`.

`B2.` If `P` = 0, set `BASE[N]` ← `MLOC`, `SUB[N]` ← `P`, and terminate the algorithm.

`B3.` If `SPACE(P)` < 0, set `BASE[N]` ← `MLOC`, `SUB[N]` ← `P`, `SPACE[(P)` ← − `SPACE(P)`, `MLOC` ← `MLOC` + `SPACE(P)`, and `N` ← `N` + 1.

`B4.` Set `P` ← `LINK(P)` and return to B2.Now return to A2 ▮

27. The following subroutine expects five parameters: `dir` ≡ `LOC(Dir)`, the address of the file directory; `x` ≡ `LOC(X[0])`, the address of the `X`-table; `n` ≡ `N`, the number of entries in the `X`-table; `mloc` ≡ `MLOC`, the amount of relocation for the first subroutine loaded; and `first` ≡ `FIRST`, the address of the directory entry for the first subroutine in the file. To access the `LINK` field in the file directory, register `link` is set to `dir` + `LINK`; to access the `SPACE` field, it suffices to define `space` as an alias for `dir` because the offset is zero. Similarly for the fields in the `X`-table, register `sub` is set to `x` + `SUB` and `base` is defined as an alias for `x`.

```
:Ex27   ADDU   link,dir,LINK
        ADDU   sub,x,SUB
        SL     n,n,3          Scale N.
        SET    i,n            A0. i ← N.
        BNP    i,A1           Loop on i for N > i ≥. 0.
0H      SUB    i,i,8          i ← i − 1.
        LDTU   p,sub,i        P ← SUB[i].
        LDT    s,space,p      s ← SPACE(P).
        NEG    s,s            Negate s.
        STT    s,space,p      SPACE(SUB[i]) ← −SPACE(SUB[i]).
        PBP    i,0B           Continue while i > 0.
        JMP    A1
A3      ADDU   p,p,4          A3. i ← i + 1.
```

```
        LDTU  q,link,p       Q ← LINK_i(SUB[N]).
        LDT   s,space,q
        BN    s,A2           If SPACE(Q) ≥ 0,
        NEG   s,s
        STT   s,space,q        SPACE(Q) ← −SPACE(Q),
        STT   q,sub,n          SUB[N] ← Q, and
        ADD   n,n,8          N ← N + 1.
A2      PBEV  q,A3           A2. If Q is odd, go to A1; else to A3.
A1      BZ    n,B1           A1. If N = 0, go to phase 2.
        SUB   n,n,8          N ← N − 1.
        LDTU  p,sub,n        P ← SUB[N], i ← 0.
        LDTU  q,link,p       Q ← LINK_i(SUB[N]).
        JMP   A2
B1      SET   p,first        B1. P ← FIRST.
        JMP   B2
B4      LDT   p,link,p       B4. P ← LINK(P).
B2      BZ    p,0F           B2.
        LDT   s,space,p      B3.
        PBNN  s,B4           To B4 if SPACE(P) ≥ 0.
0H      STT   mloc,base,n    B2/B3. BASE[N] ← MLOC.
        ANDN  p,p,1          Remove tag bit.
        STTU  p,sub,n        SUB[N] ← P.
        NEG   s,s
        STT   s,space,p      SPACE(P) ← − SPACE(P).
        ADD   mloc,mloc,s    MLOC ← MLOC + SPACE(P).
        ADD   n,n,8          N ← N + 1.
        PBNZ  p,B4           If P = 0, terminate.
        POP   0,0            Done.
```

2.2.4. Circular Lists

[552]

As stated before, we assume in the following code that the global register `avail` points to a sufficiently large stack of available nodes.

11.

```
:Copy   SET    q0,:avail            1      The future backlink
1H      SET    q,:avail             p      Q ← AVAIL.
        LDOU   :avail,:avail,LINK   p      AVAIL ← LINK(AVAIL).
        LDOU   p,p,LINK             p      Advance P.
        LDO    t,p,COEF             p
        STO    t,q,COEF             p      COEF(Q) ← COEF(P).
        LDOU   t,p,ABC              p
        STOU   t,q,ABC              p      ABC(Q) ← ABC(P).
        PBNN   t,1B                 p[1]   Was ABC ≠ 0?
        STOU   q0,q,LINK            1      Store backlink to LINK(Q).
        SET    $0,q                 1
        POP    1,0                         Return Q. ▮
```

Note that it is not necessary to set LINK(Q) (except for the last node) because the nodes on the AVAIL stack are already linked together.

12. Let the polynomial copied have p terms. Program A takes $(17p + 13)\upsilon + (9p + 5)\mu$. One can argue that a fair comparison should add the time to create a zero polynomial with exercise 14, which is $6\upsilon + 4\mu$ (not including the final POP). The program of exercise 11 takes $(8p + 5)\upsilon + (6p + 1)\mu$, about half as much time as Program A and for small p just a third as much time as the combination of Program A with exercise 14.

13.

```
:Erase  LDOU   t,p,LINK         Get first node.
        STOU   :avail,p,LINK    Link end of polynomial to the AVAIL list.
        SET    :avail,t         Point AVAIL to first node.
        POP    0,0              Done. ▮
```

14.

```
:Zero   SET    p,:avail             P ⇐ AVAIL.
        LDOU   :avail,:avail,LINK
        STCO   0,p,COEF             COEF(P) ← 0.
        NEG    t,1; STO t,p,ABC     ABC(P) ← − 1.
        STOU   p,p,LINK             LINK(P) ← P.
        POP    1,0                  Return P. ▮
```

15. This subroutine combines Algorithm M with Algorithm A. The parallel addition of the exponents is accomplished using the WDIF operation. In case of an overflow, this will produce the maximum exponent that can be represented as a two-byte unsigned integer; and as a special case of this, adding to ABC $= -1$ will always give -1.

```
:Mult  LDOU  m,m,LINK             r + 1            M1. Next multiplier.
       LDO   abcm,m,ABC           r + 1            abcm ← ABC(M).
       BN    abcm,9F              r + 1[1]         If ABC(M) < 0, terminate.
       LDO   coefm,m,COEF         r                coefm ← COEF(M).
A1     SET   q1,q                 Σm″              A1. Initialize. Q1 ← Q.
       LDOU  q,q,LINK             Σm″              Q ← LINK(Q).
0H     LDOU  p,p,LINK             Σp               P ← LINK(P).
       LDO   coefp,p,COEF         Σp               coefp ← COEF(P).
       MUL   coefp,coefm,coefp    Σp               coefp ← coefm · coefp.
       LDO   abcp,p,ABC           Σp               A2. ABC(P) : ABC(Q).
       NOR   abcp,abcp,0          Σp               abcp ← abcm + abcp by:
       WDIF  abcp,abcp,abcm       Σp                 invert, parallel subtract,
       NOR   abcp,abcp,0          Σp                 and invert.
2H     LDO   t,q,ABC              Σx               t ← ABC(Q).
       CMP   t,abcp,t             Σx               Compare abcp and ABC(Q).
       BZ    t,A3                 Σx[Σ m + 1]      If equal, go to A3.
       BP    t,A5                 Σ p′ + q′[Σ p′]  If greater, go to A5.
       SET   q1,q                 Σ q′             If less, set Q1 ← Q.
       LDOU  q,q,LINK             Σ q′             Q ← LINK(Q).
       JMP   2B                   Σ q′             Repeat.
A3     BN    abcp,:Mult           Σ m + 1[1]       A3. Add coefficients.
       LDO   coefq,q,COEF         Σ m
       ADD   coefq,coefq,coefp    Σ m              coefq ← coefq + coefp.
       STO   coefq,q,COEF         Σ m              COEF(Q) ← coefq.
       PBNZ  coefq,A1             Σ m[Σ m′]        If coefq ≠ 0, go to A1.
       SET   q2,q                 Σ m′             A4. Delete zero term.
       LDOU  q,q,LINK             Σ m′             Q ← LINK(Q).
       STOU  q,q1,LINK            Σ m′             LINK(Q1) ← Q.
       STOU  :avail,q2,LINK       Σ m′
       SET   :avail,q2            Σ m′             AVAIL ⇐ Q2.
       JMP   0B                   Σ m′             Go to advance P.
A5     SET   q2,:avail            Σ p′             A5. Insert new term.
       LDOU  :avail,:avail,LINK
```

```
33           LDOU  :avail,:avail,LINK       Σ p′     Q2 ⇐ AVAIL.
34           STO   coefp,q2,COEF            Σ p′     COEF(Q2) ← coefp.
35           STO   abcp,q2,ABC              Σ p′     ABC(Q2) ← abcp.
36           STOU  q,q2,LINK                Σ p′     LINK(Q2) ← Q.
37           STOU  q2,q1,LINK               Σ p′     LINK(Q1) ← Q2.
38           SET   q1,q2                    Σ p′     Q1 ← Q2.
39           JMP   0B                       Σ p′     Go to advance P.
40   9H      POP   0,0                               Return from subroutine. ▮
```

16. Let r be the number of terms in polynomial(`M`). The subroutine requires $13 + 4r + 34\,\Sigma m' + 28\,\Sigma m'' + 30\,\Sigma p' + 7\,\Sigma q'$ units of time, where the summations refer to the corresponding quantities during the r activations of the modified Program A. The number of terms in polynomial(`Q`) goes up by $p' - m'$ each activation of Program A. If we make the not unreasonable assumption that $m' = 0$ and $p' = \alpha p$ where $0 < \alpha < 1$, we get the respective sums equal to 0, $(1 - \alpha)pr$, αpr, and $rq_0' + \alpha p(r(r - 1)/2)$, where q_0' is the value of q' in the first iteration. The grand total is $3.5\alpha pr^2 + 28pr - 1.5\alpha pr + 7q_0'r + 4r + 13$. This analysis indicates that the multiplier ought to have fewer terms than the multiplicand, since we have to skip over unmatching terms in polynomial(`Q`) more often. (See exercise 5.2.3–29 on page *157* for a faster algorithm.)

2.2.5. Doubly Linked Lists

[554]

7. In line 225 this user is assumed to be in the `WAIT` list. . . .

8. This code implements step E8 of the elevator coroutine.

```
271  E8   SUB   floor,floor,1        E8. Go down a floor.
272       TRIP  HoldCI,61            Wait 61 units.
273       SL    $0,on,floor
274       OR    $1,callcar,calldown
275       AND   $2,$1,$0             Is CALLCAR[FLOOR] ≠ 0
276       BNZ   $2,1F                   or CALLDOWN[FLOOR] ≠ 0?
277       CMP   $2,floor,2
278       BZ    $2,2F                If not, is FLOOR = 2?
279       AND   $2,callup,$0         If not, is CALLUP[FLOOR] ≠ 0?
280       BZ    $2,E8                If not, repeat step E8.
```

```
2H      OR      $1,$1,callup
        NEG     $2,64,floor
        SL      $1,$1,$2                Ignore FLOOR and above.
        BNZ     $1,E8                   Are there calls for lower floors?
1H      SET     dt,23                   It is time to stop the elevator.
        JMP     E2A                     Wait 23 units and go to E2. ▮
```

9. This code implements the `Decision` subroutine.

```
            PREFIX  :Decision:
next        IS      $0                       NEXTINST(ELEV1)
e1          IS      $1                       Zero if next = E1
calls       IS      $2                       All buttons combined
j           IS      $3
c           IS      $4                       Local copy of :c
rJ          IS      $5
t           IS      $6
:Decision   BNZ     :state,9F                D1. Decision necessary?
            LDOU    next,:ELEV1+:NEXTINST    D2. Should doors open?
            GETA    t,:E1
            CMP     e1,next,t
            BNZ     e1,D3                    Jump if elevator not at E1.
            OR      calls,:callup,:calldown
            OR      calls,calls,:callcar
            GETA    next,:E3                 Prepare to schedule E3.
            AND     t,calls,1<<2
            BNZ     t,8F                     Jump if call set in 2.
D3          SL      t,:on,:floor             D3. Any calls?
            ANDN    calls,calls,t            Calls except in current floor
            SUB     t,calls,1
            SADD    j,t,calls                Smallest j with a call
            BNZ     calls,D4                 Jump if calls with j ≠ FLOOR.
            GET     rJ,:rJ
            GETA    t,:E6B
            CMPU    t,rJ,t                   Invoked by step E6?
            BNZ     t,9F                     If not, exit subroutine.
            SET     j,2
```

```
D4  CMP    :state,j,:floor        D4. Set STATE.
    BNZ    e1,9F                  D5. Elevator dormant?
    BZ     :state,9F              Exit if j = 2.
    GETA   next,:E6               Prepare to schedule E6.
8H  SET    c,:c                   Save current thread.
    LDA    :c,:ELEV1              Disguise as ELEV1.
    STOU   next,:c,:NEXTINST      Set NEXTINST to E3 or E6.
    SET    :dt,20                 Wait 20 units of time.
    GET    rJ,:rJ
    PUSHJ  t,:Hold                Schedule the activity.
    PUT    :rJ,rJ
    SET    :c,c                   Restore current thread.
9H  POP    0,0                    ▮
```

2.2.6. Arrays and Orthogonal Lists

[556]

5. With a secondary table `TA2` of base addresses for each row such that the octabyte at location `TA2` + $8j$ contains `LOC(A[`j`,0])` + 2, and assuming that there is a global base register `b` and small constant c with `b` + c = `LOC(TA2)` (such that the `MMIX` assembler could assemble the instruction '`LDA t,TA2`'), we can write '`LDO a,b,`$c + 4 + 1$'.

11. At most 400 + 400 + 4 · 4 · 400 = 7200 octabytes or approximately 56 KByte.

15. The following program expects four parameters: first `pivot`, the address of the pivot node; then `baserow` ≡ `LOC(BASEROW[0])`; next `basecol` ≡ `LOC(BASECOL[0])`; and finally `ptr` ≡ `LOC(PTR[0])`. Since only the `LEFT` field of the `BASEROW` nodes and the `UP` field of the `BASECOL` nodes is used, the nodes are assumed to overlap, such that only a single octabyte is used per header node. Further, the program assumes that pointers to the list heads have their least significant bit set to 1, making them odd. Within the program, no new pointers to the list heads are created, since inserting and deleting nodes will just copy existing links. The functions `Allocate` and `Free` are assumed to manage the allocation of nodes and their return to free storage. Note that line 54 requires register `x` to have a suitably large register number, and that the floating point comparison in line 67 assumes that register rE (epsilon register) has been set appropriately.

```
:PStep  GET    rJ,:rJ             S1. Initialize.
        LDO    v,pivot,VAL        v ← VAL(PIVOT).
        SETH   t,#3FF0            t ← 1.0.
        STO    t,pivot,VAL        VAL(PIVOT) ← 1.0.
        FDIV   alpha,t,v          ALPHA ← 1.0/VAL(P).
        SETH   t,#8000            The sign bit
        XOR    malpha,t,alpha     Precompute malpha ← −ALPHA.
        LDT    i0,pivot,ROW       I0 ← ROW(PIVOT).
        8ADDU  p0,i0,baserow      P0 ← LOC(BASEROW[I0]).
        LDT    J0,pivot,COL       J0 ← COL(PIVOT).
        8ADDU  q0,J0,basecol      Q0 ← LOC(BASECOL[J0]).
        JMP    S2
2H      LDT    J,p0,COL           J ← COL(P0).
        SL     j,J,3              Scale J.
        ADDU   t,basecol,j
        STOU   t,ptr,j            PTR[J] ← LOC(BASECOL[J]).
        LDO    t,p0,VAL
        FMUL   t,alpha,t
        STO    t,p0,VAL           VAL(P0) ← ALPHA × VAL(P0).
S2      LDOU   p0,p0,LEFT         S2. Process pivot row. P0 ← LEFT(P0).
        BEV    p0,2B              If P0 is even, process P0.
S3      LDOU   q0,q0,UP           S3. Find new row. Q0 ← UP(Q0).
        BOD    q0,9F              Exit if Q0 is odd.
        LDT    i,q0,ROW           I ← ROW(Q0).
        CMP    t,i,i0
        BZ     t,S3               If I = I0, repeat.
        8ADDU  p,i,baserow        P ← LOC(BASEROW[I]).
S4A     LDOU   p1,p,LEFT          P1 ← LEFT(P).
S4      LDOU   p0,p0,LEFT         S4. Find new column. P0 ← LEFT(P0).
        BOD    p0,1F
        LDT    J,p0,COL           J ← COL(P0).
        CMP    t,J,J0
        BNZ    t,S5               If J = J0,
        JMP    S4                   repeat step S4.
```

```
1H      LDO     t,q0,VAL        If P0 is odd,
        FMUL    t,malpha,t
        STO     t,q0,VAL          VAL(Q0) ← − ALPHA × VAL(Q0),
        JMP     S3                and return to S3.
1H      SET     p,p1            P ← P1.
        LDOU    p1,p,LEFT       P1 ← LEFT(P).
S5      BOD     p1,S6           S5. Find I, J element.
        LDT     t,p1,COL        t ← COL(P1).
        CMP     t,t,J
        BP      t,1B            Loop until COL(P1) ≤ J.
        BZ      t,S7            If COL(P1) = J, go right to S7.
S6      SL      t,J,3           S6. Insert I, J element.
        LDOU    pj,ptr,t        pj ← PTR[J].
2H      SET     qj,pj           qj ← pj.
        LDOU    pj,qj,UP        pj ← UP(PTR[J]).
        BOD     pj,0F           Jump if pj is odd.
        LDT     t,pj,ROW
        CMP     t,t,i
        BP      t,2B            Loop until ROW(UP(PTR[J])) ≤ I.
0H      PUSHJ   x,:Allocate     X ⇐ AVAIL.
        STCO    0,x,VAL         VAL(X) ← 0.0.
        STT     i,x,ROW         ROW(X) ← I.
        STT     J,x,COL         COL(X) ← J.
        STOU    p1,x,LEFT       LEFT(X) ← P1.
        STOU    pj,x,UP         UP(X) ← UP(PTR[J]).
        STOU    x,p,LEFT        LEFT(P) ← X.
        STOU    x,qj,UP         UP(PTR[J]) ← X.
        SET     p1,x            P1 ← X.
S7      LDO     v,q0,VAL        S7. Pivot. v ← VAL(Q0).
        LDO     t,p0,VAL        t ← VAL(P0).
        FMUL    v,v,t           v ← VAL(Q0) × VAL(P0).
        LDO     w,p1,VAL        w ← VAL(P1).
        FEQLE   t,w,v
        BNZ     t,S8            If w ≈ v (E), go to S8.
        FSUB    v,w,v
        STO     v,p1,VAL        VAL(P1) ← VAL(P1) − VAL(Q0) × VAL(P0).
```

```
        SL     t,J,3
        STOU   p1,ptr,t        PTR[J] ← P1.
        SET    p,p1            P ← P1.
        JMP    S4A
S8      SL     t,J,3           S8. Delete I, J element.
        LDOU   pj,ptr,t        pj ← PTR[J].
1H      SET    qj,pj           qj ← pj.
        LDOU   pj,qj,UP        pj ← UP(qj).
        CMP    t,pj,p1
        BNZ    t,1B            Repeat if UP(PTR[J]) ≠ P1.
        LDOU   t,p1,UP
        STOU   t,qj,UP         UP(PTR[J]) ← UP(P1).
        LDOU   t,p1,LEFT
        STOU   t,p,LEFT        LEFT(P) ← LEFT(P1).
        SET    t+1,p1
        PUSHJ  t,:Free         AVAIL ⇐ P1.
        JMP    S4A
9H      PUT    :rJ,rJ
        POP    0,0             ▮
```

2.3.1. Traversing Binary Trees

[567]

20. The following implementation of Program T uses a third parameter a, the address where it will store the stack in consecutive memory locations. The local register s is used as a stack pointer such that the stack consists of the octabyte values at a, a + 8, . . ., a + 8(s − 1).

```
:Inorder  BZ     p,1F       1[0]     T1. Initialize.
          GET    rJ,:rJ     1        Stop if P = Λ.
          SET    s,0        1        Set stack empty.
T3        STOU   p,a,s      n        T3. Stack ⇐ P.
          ADD    s,s,8      n
          LDOU   p,p,LLINK  n        P ← LLINK(P).
          BNZ    p,T3       n[a−1]   T2. P = Λ?
T4        SUB    s,s,8      n        T4. P ⇐ Stack.
          LDOU   p,a,s
```

				n	
10	T5	SET	t+1,p	n	*T5. Visit P.*
11		PUSHGO	t,visit,0	n	
12		LDOU	p,p,RLINK	n	P ← RLINK(P).
13		PBNZ	p,T3	$n_{[a]}$	*T2. P = Λ?*
14		PBP	s,T4	$a_{[1]}$	Test if the stack is empty.
15		PUT	:rJ,rJ	1	
16	1H	POP	0,0		▮

This version reduces the running time of Program T to $(12n + 5a + 4)\upsilon + 4n\mu$.

If `LLINK(P)` = Λ, the node `P` is pushed on the stack in step T3 and removed immediately again in step T4. Adding a test to step T3 like this

T3	LDOU	left,p,LLINK	n	
	PBZ	left,T5	$n_{[a-1]}$	To T5 if LLINK(P) = Λ.
	STOU	p,a,s	$a - 1$	*T3. Stack ⇐ P.*
	ADD	s,s,8	$a - 1$	
	SET	p,left	$a - 1$	P ← LLINK(P).
	JMP	T3	$a - 1$	

will eliminate the redundancy. The running time would then be $(8n + 11a - 2)\upsilon + (2n + 2a - 2)\mu$, which is a further improvement if we assume that $a = (n + 1)/2$.

For a linked stack, replace in the previous program

lines 04-05 by:

T3	STOU	p,a,INFO	n
	LDOU	t,a,LINK	n
	STOU	s,a,LINK	n
	SET	s,a	n
	SET	a,t	n

and lines 08-09 by:

T4	LDOU	t,s,LINK	n
	STOU	a,s,LINK	n
	SET	a,s	n
	SET	s,t	n
	LDOU	p,a,INFO	n

These replacements increase the running time for pushing and popping the stack from $4n\upsilon + 2n\mu$ to $10n\upsilon + 6n\mu$ to yield a total running time of $(18n + 5a + 4)\upsilon + 8n\mu$. Applying the optimization for nodes with `LLINK(P)` = Λ, we can reduce the total to $(10n + 13a - 8)\upsilon + (2n + 6a - 6)\mu$.

The same optimization applied to the recursive implementation of Program T yields the following program:

01	:Inorder	BZ	p,T4	$1_{[0]}$	*T2. P = Λ?*
02	0H	GET	rJ,:rJ	a	Entry for recursive calls.

T3	LDOU	t+1,p,LLINK	n	*T3. Stack ⇐ P.*
	PBZ	t+1,T5	$n_{[a-1]}$	*T2. P = Λ?*
	SET	t+2,visit	$a-1$	
	PUSHJ	t,0B	$a-1$	Call Inorder(LLINK(P),Visit).
T5	SET	t+1,p	n	*T5. Visit P.*
	PUSHGO	t,visit,0	n	Call Visit(P).
	LDOU	p,p,RLINK	n	P ← RLINK(P).
	BNZ	p,T3	$n_{[n-a]}$	*T2. P = Λ?*
	PUT	:rJ,rJ	a	
T4	POP	0,0	a	*T4. P ⇐ Stack.* ▮

Its running time is a remarkable $(10n + 7a - 3)\upsilon + 2n\mu$.

22. In the following implementation of algorithm U, the variable R has been eliminated (saving two instructions) by replacing the test R = Q with RLINK(Q) = P.

:Inorder	BZ	p,1F	$1_{[0]}$	*U2. Done?* Stop if P = Λ.
	GET	rJ,:rJ	1	
U3	LDOU	q,p,LLINK	$n+a-1$	*U3. Look left.* Q ← LLINK(P).
	PBZ	q,U6	$n+a-1_{[a-1]}$	To U6 if Q = Λ.
U4	LDOU	rq,q,RLINK	$2c$	*U4. Search for thread.*
	CMP	t,rq,p	$2c$	
	BZ	t,5F	$2c_{[a-1]}$	Branch if RLINK(Q) = P.
	CSNZ	q,rq,rq	d	Q ← RLINK(Q) if RLINK(Q) ≠ Λ.
	PBNZ	rq,U4	$d_{[a-1]}$	Continue with U4 if RLINK(Q) ≠ Λ.
	STOU	p,q,RLINK	$a-1$	*U5a. Insert thread.* RLINK(Q) ← P.
	LDOU	p,p,LLINK	$a-1$	*U9. Go to left.* P ← LLINK(P).
	JMP	U3	$a-1$	To U3.
5H	STCO	0,q,RLINK	$a-1$	*U5b. Remove thread.* RLINK(Q) = Λ.
U6	SET	t+1,p	n	*U6. Inorder visit P.*
	PUSHGO	t,visit,0	n	
	LDOU	p,p,RLINK	n	*U7. Go to right or up.*
	PBNZ	p,U3	$n_{[1]}$	*U2. Done?* To U3 if P ≠ Λ.
	PUT	:rJ,rJ	1	
1H	POP	0,0		▮

The total running time is $(18n + 10a - 10b - 5)\upsilon + (4n + 4a - 2b - 4)\mu$, where n

is the number of nodes, a is the number of null `RLINK`s (hence $a-1$ is the number of nonnull `LLINK`s), $c = n - b$, and $d = 2c - (a - 1)$, where b is the number of nodes of the tree's "right spine" `P`, `RLINK(P)`, `RLINK(RLINK(P))`, etc.

In summary, the approximate running times for inorder traversal are:

Program U	$(23\upsilon + 6\mu)n - O(\log n)$
Program T (with register stack)	$(16\upsilon + 2\mu)n + O(1)$
Program T (with stack in linked list)	$(16.5\upsilon + 5\mu)n + O(1)$
Program T (with stack in consecutive locations)	$(13.5\upsilon + 3\mu)n + O(1)$
Program T (optimized with register stack)	$(13.5\upsilon + 2\mu)n + O(1)$
Program S	$(13\upsilon + 2\mu)n + O(1)$

The optimized recursive version of Program T is simple and short, requires a minimum amount of memory access, and is among the fastest programs considered here. If a program needs a simple stack, recursion should be considered an option; it is hard to beat the efficiency of a hardware-supported register stack.

[571]

37. If `LLINK(P)` = `RLINK(P)` = Λ in the representation (2), let `LINK(P)` = Λ; otherwise let `LINK(P)` = `Q` where `NODE(Q)` corresponds to `NODE(LLINK(P))` and `NODE(Q + 16)` to `NODE(RLINK(P))`. The condition `LLINK(P)` or `RLINK(P)` = Λ is represented by a sentinel in `NODE(Q)` or `NODE(Q + 16)` respectively. This representation uses between $2n$ and $4n$–2 octabytes; under the stated assumptions, (2) would require 27 octabytes, compared to 22 in the present scheme. Insertion and deletion operations are approximately of equal efficiency in either representation. But this representation is not quite as versatile in combination with other structures.

2.3.2. Binary Tree Representation of Trees

[572]

13. The following subroutine implements Algorithm 2.3.1C after appropriate changes to the initialization and termination conditions. It expects one parameter `p` pointing to a node and returns a copy of this node and everything reachable through its `LLINK` pointer.

```
:Copy   BZ    p,9F        1[0]   C1. Initialize.
        GET   rJ,:rJ      1
```

```
      PUSHJ  u,:Allocate       1          Create NODE(U) with RLINK(U) = Λ.
      SET    q,u               1          Q ← U.
      JMP    C3                1          To C3, the rst time.
4H    PUSHJ  r,:Allocate       a          R ⇐ AVAIL.
      STOU   r,q,:LLINK        a          LLINK(Q) ← R.
      OR     t,q,1             a
      STOU   t,r,:RLINK        a          RLINK(R) ← Q, RTAG(R) ← 1.
      SET    q,r               a          C5a. Advance. Q ← LLINK(Q).
      LDOU   p,p,:LLINK        a          P ← LLINK(P).
C2    LDOU   t,p,:RLINK        n − 1      C2. Anything to right?
      BOD    t,C3              n − 1[a]   Jump if RTAG(P) = 1.
      PUSHJ  r,:Allocate       n − 1 − a  R ⇐ AVAIL.
      LDOU   t,q,:RLINK        n − 1 − a
      STOU   t,r,:RLINK        n − 1 − a  RLINK(R) ← RLINK(Q).
      STOU   r,q,:RLINK        n − 1 − a  RLINK(Q) ← R, RTAG(Q) ← 0.
C3    LDOU   t,p,:INFO         n          C3. Copy INFO.
      STOU   t,q,:INFO         n
      LDOU   t,p,:LLINK        n          C4. Anything to left?
      BNZ    t,4B              n[a]       Jump if LLINK(P) ≠ Λ.
C5B   LDOU   p,p,:RLINK        n          C5b. Advance. P ← RLINK(P).
      LDOU   q,q,:RLINK        n          Q ← RLINK(Q).
      BOD    q,C5B             n[a]       Jump RTAG(Q) = 1.
      PBNZ   q,C2              n − a[1]   C6. Test if complete.
      STOU   u,u,:RLINK        1          RLINK(U) ← U.
      PUT    :rJ,rJ            1
      SET    $0,u              1          Return U.
9H    POP    1,0                          ▮
```

Here n is the total number of nodes copied and a is the number of nonterminal (operator) nodes copied.

14. The total time (not counting the time spent in `Allocate`) is $(14n + 7a + 4)\upsilon + (9n - 3)\mu$. The time used to copy the `INFO` field is just $2n(\upsilon + \mu)$; for the `LLINK` fields, we need $a(\upsilon + \mu)$; and for the `RLINK` fields, we need $n(\upsilon + \mu)$. The total copy time of $(3n + a)(\upsilon + \mu)$ accounts for about 20% of the cycles and 40% of the memory access. The rest is spent on traversing the tree.

15. The following code is an exercise in nesting subroutines.

```
        PREFIX  :D:             This is part of subroutine D.
:Div    LDOU    t,q1,:INFO
        BZ      t,1F
        SET     t+1,q1
        SET     t+3,p2
        PUSHJ   t+2,:Copy
        GETA    t+3,:Div
        PUSHJ   t,:Tree2
        SET     q1,t            Q1 ← Tree2(Q1,Copy(P2),"/").
1H      LDOU    t,q,:INFO
        BZ      t,:Sub
        SET     q+3,p1
        PUSHJ   q+2,:Copy
        SET     q+3,q
        PUSHJ   q+1,:Mult       Q+1 ← Mult(Copy(P1),Q).
        SET     q+4,p2
        PUSHJ   q+3,:Copy
        PUSHJ   q+4,:Allocate
        SET     q+5,2
        STTU    q+5,q+4,:INFO
        GETA    q+5,:Pwr
        PUSHJ   q+2,:Tree2      Q+2 ← Tree2(Copy(P2),Allocate(),"↑").
        GETA    q+3,:Div
        PUSHJ   q,:Tree2        Q ← Tree2(Q+1,Q+2,"↑").
        JMP     :Sub            Q ← Q1 − Q.  ■
```

16. Even more nested subroutine calls! Note the unusual definition of register `r` serving as basis for the nested subroutine calls.

```
r       IS      t+1
:Pwr    LDOU    t,q1,:INFO
        BZ      t,2F            Jump if INFO(Q1) = 0.
        SET     r+1,p1
        PUSHJ   r,:Copy         R ← Copy(p1).
        LDWU    diff,p2,:DIFF
        BNZ     diff,1F         Jump if DIFF(P2) ≠ 0.
        LDT     info,p2,:INFO   Load value of constant P2.
        CMP     t,info,2        Is it 2?
```

```
        BZ     t,3F              If yes, jump.
        SET    r+1,r             1) R
        PUSHJ  r+2,:Allocate     2) New constant
        SUB    info,info,1             with value INFO(P2) − 1
        STT    info,r+2,:INFO
        GETA   r+3,:Pwr          3) “↑”
        PUSHJ  r,:Tree2          R ← Tree 2(R,INFO(P2) − 1,“↑”).
        JMP    3F
1H      SET    r+1,r             1) R
        SET    r+4,p2                              α) P2
        PUSHJ  r+3,:Copy            a) Copy(P2)
        PUSHJ  r+4,:Allocate        b) New constant
        SET    info,1                              with value 1
        STT    info,r+4,:INFO
        GETA   r+5,:Sub             c) “−”
        PUSHJ  r+2,:Tree2        2) Tree2(Copy(P2),1,“−”
        GETA   r+3,:Pwr          3) “↑”
        PUSHJ  r,:Tree2          R ← Tree2(R,Tree2(Copy(P2),1,“−”),“↑”
3H      SET    r+1,q1            1) Q1
        SET    r+4,p2                              α) P2
        PUSHJ  r+3,:Copy            a) Copy(P2)
        SET    r+4,r                b) R
        PUSHJ  r+2,:Mult         2) Mult(Copy(P2),R)
        PUSHJ  r,:Mult           R ← Mult(Q1,Mult(Copy(P2),R)).
        SET    q1,r              Q1 ← Mult(Q1,Mult(Copy(P2),R)).
2H      LDOU   t,q,:INFO
        BZ     t,:Add            If INFO(Q) = 0 go to Add.
        SET    q+4,p1                                        i) P1
        PUSHJ  q+3,:Copy              α) Copy(P1)
        GETA   q+5,:Ln                β) ignored, γ) “ln”
        PUSHJ  q+2,:Tree1           a) Tree1(Copy(P1), ·, “ln”)
        SET    q+3,q                b) Q
        PUSHJ  q+1,:Mult         1) Mult(Tree1(Copy(P1), ·, “ln”),Q)
        SET    q+4,p1                            α) P1
        PUSHJ  q+3,:Copy            a) Copy(P1)
```

```
        SET    q+5,p2                α) P2
        PUSHJ  q+4,:Copy          b) Copy(P2)
        GETA   q+5,:Pwr           c) "↑"
        PUSHJ  q+2,:Tree2      2) Tree2(Copy(P1),Copy(P2),"↑")
        GETA   q+3,:Mul        3) "×"
        PUSHJ  q,:Tree2        Q ← Tree2(Mult(Tree1(Copy(P1),·, "ln"),Q),
        JMP    :Add                 Tree2(Copy(P1),Copy(P2),"↑"), "×").  ∎
```

2.3.5. Lists and Garbage Collection

[601]

4. The program that follows incorporates the suggested improvements in the speed of processing atoms that appear in the text after the statement of Algorithm E. It follows closely the original `MIX` program. The least significant bit of `ALINK(P)` is used as mark bit `MARK(P)`, and the least significant bit of `BLINK(P)` is used as atom bit `ATOM(P)`. Note the use of the `MUX` (multiplex) instruction to selectively set or copy these bits.

```
:Mark   SET    t,0         1          E1. Initialize. T ← A.
        PUT    :rM,1       1          Prepare for MUXing the tag bits.
E2      LDOU   x,p,ALINK   1          E2. Mark P.
        OR     x,x,1       1
        STOU   x,p,ALINK   1          MARK(P) ← 1.
E3      LDOU   x,p,BLINK   1          E3. Atom?
        PBEV   x,E4        1[0]       Jump if ATOM(P) = 0.
E6      BZ     t,9F        n[1]       E6. Up.
        SET    q,t         n-1        Q ← T.
        LDOU   t,q,BLINK   n-1        T ← BLINK(Q).
        PBOD   t,1F        n-1[t2]    Jump if ATOM(T) = 1.
        STOU   p,q,BLINK   t2         BLINK(Q) ← P.
        SET    p,q         t2         P ← Q.
        JMP    E6          t2
1H      ANDN   t,t,1       t1         Remove tag bit from T.
        STOU   t,q,BLINK   t1         ATOM(Q) ← 0.
        LDOU   x,q,ALINK   t1         t ← ALINK(Q).
        ANDN   t,x,1       t1         T ← ALINK(Q) without mark bit.
        MUX
```

	MUX	x,x,p	t_1	t ← P retaining MARK(Q).
	STOU	x,q,ALINK	t_1	ALINK(Q) ← P retaining MARK(Q).
	SET	p,q	t_1	P ← Q.
E5	LDOU	r,p,BLINK	n	<u>*E5. Down BLINK.*</u> R ← BLINK(P).
	ANDN	q,r,1	n	Q ← BLINK(P) without atom bit.
	BZ	q,E6	$n[b_2]$	Jump if Q $= \Lambda$.
	LDOU	x,q,ALINK	$n - b_2$	
	BOD	x,E6	$n - b_2[t_1 + 1 - b_2 - a_2]$	Jump if MARK(Q)$= 1$.
	OR	x,x,1	$t_2 + a_2$	Set mark bit.
	STOU	x,q,ALINK	$t_2 + a_2$	MARK(Q) ← 1.
	LDOU	x,q,BLINK	$t_2 + a_2$	
	BOD	x,E6	$t_2 + a_{2[a_2]}$	Jump if ATOM(Q) $= 1$.
	MUX	r,r,t	t_2	R ← T retaining ATOM(P).
	STOU	r,p,BLINK	t_2	BLINK(P) ← T retaining ATOM(P).
E4A	SET	t,p	$n - 1$	T ← P.
	SET	p,q	$n - 1$	P ← Q.
E4	LDOU	r,p,ALINK	n	<u>*E4. Down ALINK.*</u> Q ← ALINK(P).
	ANDN	q,r,1	n	Q ← ALINK(P) without mark bit.
	BZ	q,E5	$n[b_1]$	Jump if Q $= \Lambda$.
	LDOU	x,q,ALINK	$n - b_1$	
	BOD	x,E5	$n - b_{1[t_2 + 1 - b_1 - a_1]}$	Jump if MARK(Q) $=$ 1.
	OR	x,x,1	$t_1 + a_1$	Set mark bit.
	STOU	x,q,ALINK	$t_1 + a_1$	MARK(Q) ← 1.
	LDOU	x,q,BLINK	$t_1 + a_1$	
	BOD	x,E5	$t_1 + a_{1[a_1]}$	Jump if ATOM(Q) $=$ 1.
	LDOU	x,p,BLINK	t_1	
	OR	x,x,1	t_1	Set atom bit.
	STOU	x,p,BLINK	t_1	ATOM(P) ← 1.
	MUX	r,r,t	t_1	R ← T retaining ATOM(P).
	STOU	r,p,ALINK	t_1	ALINK(P) ← T retaining ATOM(P).
	JMP	E4A	t_1	
9H	POP	0,0		▮

By Kirchhoff's law, $t_1 + t_2 + 1 = n$, $a_1 + a_2 = a$, and $b_1 + b_2 = b$. The total time is $(29n + 6t_1 + 4a - 2b - 5)\upsilon + (9n + 4t_1 + 2a - b - 2)\mu$, where n is the number of nonatomic nodes marked, a is the number of atoms marked, b is the number of Λ links encountered in marked nonatomic nodes, and t_1 is the number of times we went down an `ALINK` $(0 \le t_1 < n)$.

2.5. DYNAMIC STORAGE ALLOCATION

[607]

4. The following implementation uses a register `link` to simplify (and speed up) access to the `LINK` field given an address relative to `base`. For the `SIZE` field, no such register is needed since the offset of the `SIZE` field is zero. To improve the readability, however, we define `size` as an alias for `base`.

```
:Allocate  ADDU  link,:base,LINK
size       IS    :base
           LDA   p,:AVAIL         A1. Initialize. P ← LOC(AVAIL).
           SUBU  p,p,link         Convert to relative address.
1H         SET   q,p              Q ← P.
           LDT   p,q,link         A2. End of list? P ← LINK(Q).
           BN    p,9F             If P = Λ, no room.
           LDT   s,p,size         A3. Is SIZE enough?
           SUB   k,s,n            K ← SIZE(P) - N.
           PBN   k,1B             Jump if N > SIZE(P).
           PBNZ  k,1F             A4. Reserve N.
           LDT   t,p,link         If K = 0,
           STT   t,q,link           set LINK(Q) ← LINK(P).
1H         STT   k,p,size         SIZE(P) ← K.
           ADD   p,p,k            P + K.
           ADDU  $0,p,:base       Convert P + K to an absolute address
           POP   1,0                and return it.
9H         POP   0,0              Return Λ. ▮
```

13. The following code uses registers `size`, `rlink`, `llink`, and `psize` to simplify access to the various fields of a node using relative addresses. The notation `PSIZE(P)` is a convenient shorthand for the `SIZE` field that terminates the block *preceding* `NODE(P)` as if it were a field of `NODE(P)`.

```
:Allocate  ADD   n,n,8+7          1          A.1 Initialize.
```

```
        ANDN  n,n,7                 1            Add overhead and round up.
        LDA   size,:AVAIL+SIZE      1            Base address for SIZE field,
        LDA   rlink,:AVAIL+RLINK    1              for RLINK field,
        LDA   llink,:AVAIL+LLINK    1              for LLINK field, and
        SUBU  psize,size,4          1              for preceding SIZE.
        SET   p,:rover              1            P ← ROVER.
        SET   f,0                   1            F ← 0.
        JMP   A2                    1            Start the search.
A3      LDTU  s,size,p              A            A3. Is SIZE enough?
        SUB   k,s,n                 A            K ← SIZE(P) - N.
        BNN   k,A4                  A[1]         Jump if SIZE(P) ≥ N.
1H      LDTU  p,rlink,p             A + B − 1    P ← RLINK(P).
A2      PBNZ  p,A3                  A + B[B]     A2. End of list?
        BNZ   f,9F                  B[0]         Over ow if P = 0 and F ≠ 0.
        SET   f,1                   B            F ← 1.
        JMP   1B                    B
A4      LDTU  :rover,p,rlink        1            A4'. Reserve at least N.
        CMP   t,k,c                 1
        BNN   t,1F                  1[1 − D]     Jump if K ≥ c.
        LDTU  q,llink,p             D            Delete NODE(P) from list.
        STTU  :rover,rlink,q        D
        STTU  q,llink,:rover        D
        SET   l,p                   D            Result is P.
        SET   n,s                   D            Size of result is size of P.
        JMP   2F                    D
1H      ADDU  l,p,k                 1 − D        Split NODE(P) into P and L.
        STTU  k,size,p              1 − D        SIZE(P) ← K.
        STTU  k,psize,l             1 − D        SIZE(P) ← K at block end.
2H      OR    n,n,1                 1
        STTU  n,size,l              1            SIZE(L) ← N, TAG(L) ← 1.
        ADDU  q,l,n                 1            Advance to block after L.
        STTU  n,psize,q             1            SIZE(L) ← N,TAG(L) ← 1.
        ADDU  $0,rlink,l            1            Return absolute address
        POP   1,0                                  of usable memory.
9H      POP   0,0                                Overflow. ▮
```

The running time is $(23 + 5A + 7B + D)\upsilon + (4 + 2A + B + D)\mu$. Here $A \geq 1$ is the number of iterations necessary when searching for an available block that is large enough; $B = 1$, if the iteration wraps around the end of the list; and $D = 1$, if a block is deleted from the list. We can assume that the average value of B is quite small, whereas the average value of D will approach 1 when the system reaches a stable state.

16. This subroutine uses the same conventions as the solution to exercise 13. We use the variables `P1` and `N1`, respectively, for the address and size of the block following `P0`, and `N2` for the size of the block preceding `P0`; `F` is the forward block and `B` the backward block in the linked list.

```
:Free   LDA   size,:AVAIL+SIZE     Base address for SIZE eld,
        LDA   rlink,:AVAIL+RLINK     for RLINK field,
        LDA   llink,:AVAIL+LLINK     for LLINK field, and
        SUBU  psize,size,4           for preceding SIZE.
        SUBU  p0,p0,rlink          Make P0 a relative address.
        LDTU  n,size,p0            D1. Initialize. N ← SIZE(P0).
        ANDN  n,n,1                Remove TAG bit.
        ADDU  p1,p0,n              P1 ← P0 + N.
        LDTU  n1,size,p1           N1 ← SIZE(P1).
        LDTU  n2,psize,p0          N2 ← PSIZE(P0).
        BEV   n1,D4                To D4 if NODE(P1) is free.
        BEV   n2,D7                To D7 if NODE(P2) is free.
D3      LDTU  f,llink,0            D3. Insert P0. F ← LLINK(AVAIL).
        SET   b,0                  B ← AVAIL.
        JMP   D5
D4      ADD   n,n,n1               D4. Delete upper area. N ← N + SIZE(P1).
        LDTU  b,llink,p1           B ← LLINK(P1).
        LDTU  f,rlink,p1           F ← RLINK(P1).
        CMP   t,p1,:rover
        CSZ   :rover,t,0           If P1 = ROVER, set ROVER ← AVAIL.
        ADDU  p1,p1,n1             P1 ← P1 + SIZE(P1).
        BEV   n2,D6                To D6 if NODE(P2) is free.
D5      STTU  f,rlink,p0           D5. Insert NODE(P0). RLINK(P0) ← F.
        STTU  b,llink,p0           LLINK(P0) ← B.
        STTU  p0,rlink,b           RLINK(B) ← P0.
        STTU  p0,llink,f           LLINK(F) ← P0.
```

```
         JMP    D8
D6       STTU   f,rlink,b        D6. Delete. RLINK(B) ← F.
         STTU   b,llink,f        LLINK(F) ← B.
D7       ADD    n,n,n2           D7. Enlarge lower area.
         SUBU   p0,p0,n2         Move P0 to NODE(P2).
D8       STTU   n,size,p0        D8. Store SIZE. SIZE(P0) ← N.
         STTU   n,psize,p1       PSIZE(P1) ← N.
         POP    0,0              ▮
```

The possible running times are 18υ (next block occupied, preceding block free), 22υ (next block occupied, preceding block occupied), 27υ (next block free, preceding block occupied), or 28υ (next block free, preceding block free).

27. The node sizes are 2^k bytes with $4 \le k \le m$; the minimum node size is $2^4 =$ 16 bytes because an available node must contain three tetrabytes for `KVAL`, `LINKF`, and `LINKB`. Addresses are stored relative to the value of the global register `base` and are assumed to fit in a tetrabyte. Consequently, m is some constant $m < 32$. The list heads `AVAIL[4]`, `AVAIL[5]`, . . . , `AVAIL[`m`]` are allocated immediately before the `base`-address such that the relative address of `AVAIL[`k`]` is $16(k - m - 1)$; list heads are the only nodes with negative relative addresses. In the `KVAL` field of a node, we do not store k or 2^k (its size), but rather the relative address of `AVAIL[`k`]`; this is more convenient and the value of k can be easily computed from the address if needed.

For the `TAG` bits—anticipating exercise 29—we use a separate memory area, starting at address `TAGS`, containing one bit for each 16-byte block of available memory. For convenience, we keep `LOC(TAGS)` in the global register `tags`. The following auxiliary function `FindTag` will take any nonnegative relative address `P` as a parameter and return *three* return values: the octabyte containing the `TAG` bit, a mask with the respective bit set to 1, and the relative address of the octabyte within the `TAGS`.

```
          PREFIX  :FindTag:
p         IS      $0               Parameter
tag       IS      $2               Primary return value
mask      IS      $0               Second return value
address   IS      $1               Third return value
t         IS      $3               Temporary variable
:FindTag  SR      address,p,7      address ← ⌊(P/16/64) * 8⌋.
          SR      t,p,4
```

```
        AND     t,t,64-1                   t ← ⌊P/16⌋ mod 64.
        SETH    mask,#8000
        SRU     mask,mask,t                mask ← 2^(63−t).
        LDOU    tag,:tags,address
        POP     3,0                        Return tag, mask, and address. ▮
```

The running time of this function is $9\upsilon + 1\mu$ (including the final `POP`); it is used in the following implementation of Algorithm R and again in the solution of exercise 28.

The function `Allocate` expects one parameter k. On success, it will return an absolute address to 2^k bytes; on failure, it will return $\Lambda = 0$.

```
:Allocate  ADDU    linkf,:base,LINKF     1
           ADDU    linkb,:base,LINKB     1
           CMP     t,k,4                 1
           CSN     k,t,4                 1          k ← max{k, 4}.
           NEG     availk,16*(:m+1)      1          availk ← LOC(AVAIL[0]).
           16ADDU  availk,k,availk       1          availk ← LOC(AVAIL[k]).
           SET     availj,availk         1          R1. Find block. j ← k.
1H         LDT     l,availj,linkf        1 + R      L ← availF[j].
           PBNN    l,R2                  1 + R[R]   To R2 if L ≠ AVAIL[j].
           ADD     availj,availj,16      R          j ← j + 1.
           PBN     availj,1B             R[0]       Is j ≤ m?
           POP     0,0                   0          Return Λ.
R2         GET     rJ,:rJ                1          R2. Remove from list.
           LDT     p,l,linkf             1          P ← LINKF(L).
           STT     p,availj,linkf        1          availF[j] ←P.
           STT     availj,p,linkb        1          LINKB(P) ← LOC(AVAIL[j]).
           SET     t+1,l                 1
           PUSHJ   t,:FindTag            1          Find TAG(L).
           ANDN    t,t,t+1               1          Set tag bit to zero.
           STOU    t,:tags,t+2           1          TAG(L) ← 0.
           SUB     jk,availj,availk      1          R3. Split required?
           SR      jk,jk,4               1          jk ← j − k.
           PBZ     jk,9F                 1[R']      Terminate if j = k.
           SET     bitk,1                R'         bitk ← 2^0.
           SL      bitk,bitk,k           R'         bitk ←2^k.
```

```
R4      SUB     jk,jk,1              R          R4. Split. j ← j − 1.
        SL      t,bitk,jk            R          t ← 2^j.
        ADDU    p,l,t                R          P ← L + 2^j.
        SET     t+1,p                R
        PUSHJ   t,:FindTag           R          Find TAG(P).
        OR      t,t,t+1              R          Set tag bit to one.
        STOU    t,:tags,t+2          R          TAG(P) ← 1.
        16ADDU  availj,jk,availk     R          Get LOC(AVAIL[j]).
        STT     availj,p,kval        R          KVAL(P) ← LOC(AVAIL[j]).
        STT     availj,p,linkf       R          LINKF(P) ← LOC(AVAIL[j]).
        STT     availj,p,linkb       R          LINKB(P) ← LOC(AVAIL[j]).
        STT     p,availj,linkf       R          availF[j] ← P.
        STT     p,availj,linkb       R          availB[j] P.
        BP      jk,R4                R[R − R']  Repeat if j > k.
9H      ADDU    $0,:base,l           1          Return L as absolute address.
        PUT     :rJ,rJ               1
        POP     1,0                             ▮
```

The running time is $(22+22R+2R')\upsilon + (5+7R)\mu$ plus $(R+1)(9\upsilon + 1\mu)$ for the `FindTag` subroutine, where R is the number of times a block is split in two, and R' is 1 if $R > 0$, and 0 otherwise. Since R is quite small on the average, we can assume ave $R' \approx$ ave R. For good performance, the `FindTag` subroutine should be inlined, reducing its cost by $(R + 1)(5\upsilon + 1\mu)$.

28. The function `Free` expects two parameters `L` and k, assuming that `L` was obtained through a call to the function `Allocate` (see exercise 27) with the same value k.

```
:Free   GET     rJ,:rJ               1
        ADDU    linkf,:base,LINKF    1
        ADDU    linkb,:base,LINKB    1
        CMP     t,k,4                1
        CSN     k,t,4                1          k ← max{k, 4}.
        SUBU    l,l,:base            1          Make L a relative address.
        SUB     availk,k,:m+1        1
        SLU     availk,availk,4      1          availk ← LOC(AVAIL[k]).
S1      SET     t,1                  1 + S      S1. Is buddy available?
        SLU     t,t,k                1 + S      t ← 2^k.
```

	`XOR`	`p,l,t`	$1+S$	$P \leftarrow \text{buddy}_k(\text{L})$.
	`SET`	`t+1,p`	$1+S$	
	`PUSHJ`	`t,:FindTag`	$1+S$	Find `TAG(P)`.
	`AND`	`t,t,t+1`	$1+S$	Extract `TAG(P)`.
	`PBZ`	`t,S3`	$1+S_{[B]}$	To S3 if `TAG(P)` $=0$.
	`LDT`	`t,p,kval`	$B+S$	`t ← KVAL(P)`.
	`CMP`	`t,t,availk`	$B+S$	`KVAL(P)` $= k$?
	`PBNZ`	`t,S3`	$B+S_{[S]}$	To S3 if `KVAL(P)` $\neq k$.
	`LDT`	`r,p,linkf`	S	<u>*S2. Combine with buddy.*</u>
	`LDT`	`q,p,linkb`	S	`R ← LINKF(P); Q ← LINKB(P)`.
	`STT`	`r,q,linkf`	S	`LINKF(LINKB(P)) ← LINKF(P)`.
	`STT`	`q,r,linkb`	S	`LINKB(LINKF(P)) ← LINKB(P)`.
	`ADD`	`k,k,1`	S	Increase k.
	`ADD`	`availk,availk,16`	S	
	`AND`	`l,l,p`	S	If `L` $>$ `P`, set `L` $\leftarrow$ `P`.
	`JMP`	`S1`	S	
`S3`	`SET`	`t+1,l`	1	<u>*S3. Put on list.*</u>
	`PUSHJ`	`t,:FindTag`	1	Find `TAG(L)`.
	`OR`	`t,t,t+1`	1	Set tag bit to one.
	`STOU`	`t,:tags,t+2`	1	`TAG(L)` $\leftarrow 1$.
	`LDT`	`p,availk,linkf`	1	`P` $\leftarrow$ `AVAILF[k]`.
	`STT`	`p,l,linkf`	1	`LINKF(L) ← P`.
	`STT`	`l,p,linkb`	1	`LINKB(P) ← L`.
	`STT`	`availk,l,kval`	1	`KVAL(L)` $\leftarrow k$.
	`STT`	`availk,l,linkb`	1	`LINKB(L)` $\leftarrow$ `LOC(AVAIL[`k`])`.
	`STT`	`l,availk,linkf`	1	`AVAILF[`k`]` $\leftarrow$ `L`.
	`PUT`	`:rJ,rJ`	1	
	`POP`	`0,0`		▮

The running time is $(26 + 20S + 5B)\upsilon + (7 + 5S + B)\mu$ plus $(S + 2)(9\upsilon + 1\mu)$ for the `FindTag` subroutine, where S is the number of times buddy blocks are reunited, and B is the number of times a potential buddy is available but of the wrong size. With $B \approx 0.5$, the running time simplifies to $(46.5 + 29S)\upsilon + (9.5 + 6S)\mu$. Storing the tag bits inside the nodes would improve the performance, but reserving a bit in a node is usually not convenient for a general-purpose memory allocator. Again, inlining the `FindTag` function saves another $(10 + 5S)\upsilon$.

34. The variables `BASE`, `AVAIL`, and `USE` are kept in global registers. These node addresses, as well as `P`, `Q`, and `TOP`, always point *into* the node as described in exercise 33—except during step G9, where `P` and `Q` point to the `LINK` field. The field offsets for `LINK`, `SIZE`, and `T` are negative, and `MMIX` is not specially suited to handle negative constants. Therefore, we use three registers to hold these constants. Step G1 is omitted from the following program.

:GC	NEG	size,16	1	Field offset for `SIZE`
	NEG	t,12	1	Field offset for `T`
	NEG	link,8	1	Field offset for `LINK`
	SET	top,:avail	1	*G2. Initialize marking phase.*
	STCO	0,:avail,link	1	`LINK(AVAIL)` $\leftarrow \Lambda$.
	BZ	:use,G3	$1_{[0]}$	If USE $\neq \Lambda$ push it.
	STOU	top,:use,link	1	`LINK(USE)` $\leftarrow$ `TOP`.
	SET	top,:use	1	`TOP` $\leftarrow$ `USE`.
G3	SET	p,top	$a+1$	*G3. Pop up stack*. `P` ← `TOP`.
	LDOU	top,top,link	$a+1$	`TOP` $\leftarrow$ `LINK(TOP)`.
	BZ	top,G5	$a+1_{[1]}$	If `TOP` $= \Lambda$, go to G5.
	LDTU	k,p,t	a	*G4. Put new links on stack.* k $\leftarrow$ `T(P)`.
1H	BNP	k,G3	$a+b_{[a]}$	While $k > 0$ do:
	SUB	k,k,8	b	decrement k,
	LDOU	q,p,k	b	`Q` ← `LINK(P` $+ k$),
	BZ	q,1B	$b_{[b']}$	continue if `Q = ` Λ,
	LDOU	l,q,link	$b-b'$	`L` $\leftarrow$ `LINK(Q)`,
	BNZ	l,1B	$b-b'_{[a-1]}$	continue if `LINK(Q)` $\neq \Lambda$,
	STOU	top,q,link	$a-1$	`LINK(Q)` $\leftarrow$ `TOP`, and
	SET	top,q	$a-1$	`TOP` $\leftarrow$ `Q`.
	JMP	1B	$a-1$	
G5	SET	q,:base	1	*G5. Initialize next phase.*
	STOU	q,:avail,link	1	`LINK(AVAIL)` $\leftarrow$ `Q`.
	STCO	0,:avail,size	1	`SIZE(AVAIL), T(AVAIL)` $\leftarrow 0$.
	SET	p,:base	1	`P` $\leftarrow$ base.
	JMP	G6	1	
1H	STOU	q,p,link	1	`Q` $\leftarrow$ `LINK(P)`.
	ADDU	q,q,s	1	`Q` $\leftarrow$ `Q + SIZE(P)`.
	ADDU	p,p,s	1	`P` $\leftarrow$ `P + SIZE(P)`.

`G6`	`LDOU`	`l,p,link`	$a+1$	`L` ← `LINK(P)`.
`G6A`	`LDTU`	`s,p,size`	$a+c+1$	`s` ← `SIZE(P)`.
	`BZ`	`l,G7`	$a+c+1_{[c]}$	To G7 if `LINK(P) = `Λ.
	`PBNZ`	`s,1B`	$a+1_{[1]}$	To G8 if `SIZE(P) = 0`.
`G8`	`BZ`	`:use,0F`	1	*G8. Translate all links.*
	`LDOU`	`:use,:use,link`	1	`USE` ← `LINK(USE)`.
`0H`	`SET`	`:avail,q`	1	`AVAIL` ← `Q`.
	`SET`	`p,:base`	1	`P` ← `base`.
	`JMP`	`G8P`	1	
`1H`	`LDTU`	`x,ps,size`	d	`x` ← `SIZE(ps)`.
	`ADDU`	`s,s,x`	d	`s` ← `s + SIZE(ps)`.
`G7`	`ADDU`	`ps,p,s`	$c+d$	*G7. Collapse available area.*
	`LDOU`	`l,ps,link`	$c+d$	`L` ← `LINK(ps)`.
	`BZ`	`l,1B`	$c+d_{[d]}$	Repeat if `LINK(ps)` $=\Lambda$.
	`STTU`	`s,p,size`	c	`SIZE(P)` ← `s`.
	`ADDU`	`p,p,s`	c	`P` ← `P + SIZE(P)`.
	`JMP`	`G6A`	c	
`2H`	`SUB`	`k,k,8`	b	Decrement k.
	`LDOU`	`q,p,k`	b	`Q` ← `LINK(P` $+8+\mathrm{k}$).
	`BZ`	`q,1F`	$b_{[b']}$	Ignore Λ.
	`LDOU`	`l,q,link`	$b-b'$	`L` ← `LINK(Q)`.
	`STOU`	`l,p,k`	$b-b'$	`LINK(P` $+8+k$) ← `L`.
`1H`	`BP`	`k,2B`	$a+b_{[b]}$	Jump if $k>0$.
`3H`	`ADDU`	`p,p,s`	$a+c$	`P` ← `P + SIZE(P)`.
`G8P`	`LDTU`	`s,p,size`	$1+a+c$	`s` ← `SIZE(P)`.
	`LDOU`	`l,p,link`	$1+a+c$	`L` ← `LINK(P)`.
	`BZ`	`l,3B`	$1+a+c_{[c]}$	Is `LINK(P)` $=\Lambda$?
	`LDTU`	`k,p,t`	$1+a$	k ← `T(P)`.
	`PBNZ`	`s,1B`	$1+a_{[1]}$	Jump unless `SIZE(P)` $=0$.
`G9`	`SUBU`	`p,:base,16`	1	*G9. Move.*
	`SET`	`q,p`	1	`Q` and `P` start at `LINK(base)`.
	`JMP`	`G9P`	1	
`1H`	`STCO`	`0,q,8`	a	`LINK(Q)` ← Λ.
	`STOU`	[illegible]		

```
            STOU   st,q,0       a            SIZE(Q), T(Q) ← SIZE(P), T(P).
            ADDU   q,q,s        a            Q ← Q + SIZE(P).
            NEG    s,16,s       a            s ← 16 − s.
     2H     LDOU   x,p,s        w − 2        Copy data from P to Q.
            STOU   x,q,s        w − 2
            ADD    s,s,8        w − 2        s ← s + 8.
     0H     PBN    s,2B         w − 2[a]
     G9P    LDOU   l,p,8        1 + a + c    L ← LINK(P).
            LDOU   st,p,0       1 + a + c    st ← SIZE(P), T(P).
            SRU    s,st,32      1 + a + c    s ← SIZE(P).
            ADDU   p,p,s        1 + a + c    P ← P + SIZE(P).
            BZ     l,G9P        1 + a + c[c] Jump if LINK(P) = Λ.
            PBNZ   s,1B         1 + a[1]     Jump unless SIZE(P) = 0.
            POP    0,0                       ▮
```

The total running time for this program is $(35a + 14b + 4w + 23c + 7d + 37)\upsilon + (12a + 5b - 3b' + 2w + 7c + 2d + 9)\mu$, where a is the number of accessible nodes, b is the number of link fields therein, b' is the number of link fields containing Λ, c is the number of inaccessible nodes that are *not* preceded by an inaccessible node, d is the number of inaccessible nodes that *are* preceded by an inaccessible node, and w is the total number of octabytes in the accessible nodes. If the memory contains n nodes, with ρn of them inaccessible, then we may estimate $a = (1 - \rho)n$, $c = (1 - \rho)\rho n$, $d = \rho^2 n$. Example: five-octabyte nodes (on the average), with two link fields per node (on the average), and a memory of 1000 nodes. Then when $\rho = 0.2$, it takes 352υ per available node recovered; when $\rho = 0.5$, it takes 98υ; and when $\rho = 0.8$, it takes only 31υ.

3.2.1.1. Choice of modulus

[543]

1. Let c' be a solution to the congruence $ac' \equiv c$ (modulo m). (Thus, $c' = a'c \bmod m$, if a' is the number in the answer to exercise 3.2.1–5.) Results derived in Section 3.3.4 imply that $c' = 1$ works about as well as any constant.

2. For a small $c < 2^{16}$, an `INCL` instruction can be used instead of the `ADDU` instruction, which requires c to be in a register.

```
:Random   MULU   x,x,a       X ← aX mod w.
          ADDU   x,x,c       X ← (X + c) mod w.
          SET    $0,x
          POP    1,0         ▮
```

5. A `CMPU` instruction is needed to find out whether $d = x - y$ is negative without overflow.

```
SUBU   d,x,y
CMPU   t,x,y
ZSN    t,t,m
ADDU   d,d,t   ▮
```

The sum $s = x + y \bmod m$ is computed similarly after rewriting it as a difference $x - (m - y) \bmod m$.

```
SUBU   t,m,y
SUBU   s,x,t
CMPU   t,x,t
ZSN    t,t,m
ADDU   s,s,t   ▮
```

But if m is less than 2^{e-1}, the computations can be done directly without `CMPU`, using ordinary two's complement representations.

```
SUB   d,x,y
ZSN   t,d,m
ADD   d,d,t   ▮
```

And for the sum:

```
ADDU   s,x,y
SUBU   t,s,m
CSNN   s,t,t   ▮
```

8.

```
MULU   r,a,x; GET q,rH       Compute q, r with aX = qw + r.
ADDU   x,q,r                 X ← q + r.
CMPU   t,x,q
ZSN    t,t,1                 t ← [q + r ≤ w].
ADDU   x,x,t                 X ← X + t.   ▮
```

3.2.1.3. Potency

1. For `MMIX`, we have $m = 2^{64}$. With $a = 2^k + 1$ and $b = 2^k$, b is a multiple of 2, the only prime dividing m; and b is a multiple of 4, if $k > 1$. So we have a maximum period.

2. If $ks \geq 64$, then $b^s = 2^{ks} \equiv 0$ (modulo m). We conclude: $k \geq 32$ gives potency $s = 2$, $k \geq 22$ gives potency $s = 3$, and $k \geq 16$ gives potency $s = 4$. The only reasonable values for k, considering potency, are less than 16. On the other hand, small values of k yield small multipliers, which should be avoided.

3.2.2. Other Methods

[556]

25. If the subroutine of Program A is invoked as `PUSHJ t,Random`, it puts the next random number in register `t`. The overhead of the subroutine call is 4υ, one for the `PUSHJ` and three for the `POP`. The subroutine itself takes $9\upsilon + 3\mu$ (not counting the `POP`). The total time per random number is $13\upsilon + 3\mu$; the calling overhead is about 30 percent.

We save overhead by using the five instructions

```
         SUB    j,j,8
         PBP    j,1F
         PUSHJ  t,Random55
         SET    j,55*8
1H       LDOU   t,y,j                ▮
```

to put the next random number in register `t`, with the following subroutine:

```
:Random55  SET   j,24*8       j ← 24.
           ADD   ykj,y,31*8   k ← 55, ykj ← Address of Y[k − j].
1H         LDOU  x,y,j        X ← Y[j].
           LDOU  t,ykj,j      t ← Y[k − j + j] = Y[k].
           ADDU  x,x,t        X ← Y[j] + Y[k].
           STOU  x,ykj,j      Y[k] ← Y[k] + Y[j].
           SUB   j,j,8        j ← j − 1.
           PBP   j,1B
k          IS    j            Reuse register j for k.
           SET   k,31*8       k ← 31.
           ADD   ykj,y,24*8   j ← 55, ykj ← Address of Y[j − k].
1H         LDOU  x,ykj,k      X ← Y[j − k + k] = Y[j].
           LDOU  t,y,k        t ← Y[k].
```

```
ADDU   x,x,t       X ← Y[j] + Y[k].
STOU   x,y,k       Y[k] ← Y[k] + Y[j].
SUB    k,k,8       k ← k − 1.
PBP    k,1B
POP    0,0         ▮
```

The cost is now only $11\upsilon + 55(6\upsilon + 3\mu)$ for the subroutine call, and a single random number costs $(9 + 15/55)\upsilon + 4\mu$ on average. [A similar implementation, . . .

3.4.1. Numerical Distributions

[584]

3. If full-word random numbers are given . . .

Unfortunately, however, the "himult" operation in (1) is not supported in many high-level languages; see exercise 3.2.1.1–3. Division by m/k may be best when highmult is unavailable. Indeed, if $k = 2^i$ and $m = 2^{64}$, the division by m/k can be accomplished in a single `MMIX` cycle as well:

`SRU x,u,`$(64 - i)$ $X \leftarrow U/(m/k)$.

In this special but common case, the division by m/k is the same as multiplication with k/m. The remainder method uses the i least significant bits of U, where the multiplication method uses the i most significant bits. The latter is preferable.

3.6. SUMMARY

[599]

1. The following subroutine keeps X in a global register for efficiency; no load or store operations are required. The constant a is loaded in four steps as an immediate value; it could also be in a global register, of course.

```
x          GREG
a          IS     6364136223846793005   See Section 3.3.4, Table 1, line 26.
c          IS     2009                  MMIX
k          IS     $0                    Parameter
t          IS     $1                    Temporary variable
:RandInt   SETH   t,(a>>48)&#FFFF       Load constant a.
           INCMH  t,(a>>32)&#FFFF
```

```
INCML  t,(a>>16)&#FFFF
INCL   t,a&#FFFF
MULU   x,x,t          X ← aX mod m.
INCL   x,c            X ← (aX + c) mod m.
MULU   t,x,k          (rH, t) ← Xk.
GET    t,:rH          t ← ⌊Xk/m⌋.
ADD    $0,t,1         Return ⌊Xk/m⌋ + 1.
POP    1,0            ▮
```

The total running time of the subroutine is 30υ including the final `POP`. Adding the time to pass the parameter k (1υ) and to execute the `PUSHJ` instruction (1υ), a random integer value can be computed in 32υ. Keeping a in another global register will save 4υ.

4.1. POSITIONAL NUMBER SYSTEMS

[605]

4. (a) The product in register `x` has the radix point at the left end. Overflow will occur if the result is greater or equal than $(0.1)_2$. Registers rH and rR are not affected.

(b) The remainder in register rR has the radix point between bytes 3 and 4 (the same as `a`). The quotient in register `x` has the radix point between bytes 6 and 7. Register rH is not affected. The results get a bit confusing if the radix point in the divisor is farther to the left than the radix point in the dividend. Imagine dividing $(00101.000)_{256}$ by $(001.00000)_{256}$. Then after division, register rR will contain a “remainder” of $(00001.000)_{256}$ and the register `x` will be 1, representing a “quotient” of $(100)_{256}$ with the radix point two bytes past the right end of the register.

(c) The product in registers (rH, `x`) has the radix point between rH and `x`. Register rR is not affected.

(d) As long as rD contains zero, the radix points are the same as in (b). The results are also the same, because we assumed `a` and `b` to be nonnegative.

The `DIVU` (divide unsigned) instruction uses the register pair (rD, `a`) to form a 128-bit dividend with the upper 64 bits of the dividend residing in the dividend register rD. As long as the quotient will fit into the single register `x`, the radix points will be as in (b). Otherwise, `MMIX` simply sets $x \leftarrow$ rD and the remainder register rR $\leftarrow b$; register `x` will inherit the radix point from the register pair (rD,

a) and register rR from b.

4.2.1. Single-Precision Calculations

14. The following subroutine has one parameter: u, a normalized floating point number. It returns the nearest signed 64 bit two's complement integer.

:Fix	ZSN	s,u,1	*Unpack.* Record sign.
	ANDNH	u,#8000	Remove sign bit.
	SRU	e,u,52	Get exponent.
	SLU	u,u,11	Get fraction part and add hidden bit.
	ORH	u,#8000	
	SET	t,1023+63; SUB e,e,t	$e \leftarrow e - q - 63$. Now $u =$ u $\times 2^e$.
	BP	e,:Error	Overflow.
	BZ	e,Sign	
	NEG	e,e	*Round.* Set $e \leftarrow -e$.
	NEG	t,64,e	
	SLU	f,u,t	$f \leftarrow$ the fraction part of u $\times 2^e$.
	SRU	u,u,e	u $\leftarrow \lfloor$u $\times 2^e\rfloor$.
	SETH	t,#8000; CMPU t,f,t	Compare f to 0.5.
	CSOD	carry,u,1	u is odd. Round up if $f \geq \frac{1}{2}$.
	CSEV	carry,u,t	u is even. Round up if f $> \frac{1}{2}$.
	ZSNN	carry,t,carry	Round down if $f < \frac{1}{2}$.
	ADDU	u,u,carry	
Sign	BNZ	s,Negative	Attach sign.
	BN	u,:Error	Overflow.
	POP	1,0	Return u.
Negative	NEG	u,u	
	BNN	u,:Error	Overflow.
	POP	1,0	Return u. ▮

15. The following code uses the same register names as Program A; it finally jumps to Program N, except if the return value is zero.

:Fmod	ZSN	s,u,1 1.	*1. Unpack.* Set sign.
	ANDNH	u,#8000	Remove sign bit.
	SRU	e,u,52	Get exponent.
	SETH	t,#FFF0; ANDN f,u,t	Get fraction part and add hidden bit.
	INCH	f,#10	

```
        SET   fl,0                      u = ±(f, fl)2^(e-q)/2^52.
        SET   t,1023; SUB e,e,t         2. Subtract q.
        BN    e,0F                      Branch if u has no integer part.
        ADD   t,e,12; SLU f,f,t         3. Remove integer part.
        SRU   f,f,12
        SET   e,0
0H      BZ    f,6F                      Branch if u has no fraction part.
        BZ    s,5F                      Branch if u is nonnegative.
        ADD   t,e,64; SLU fl,f,t        4. Complement fraction part.
        NEG   t,e; SRU f,f,t            (f, fl) ← (f, 0)/2^e.
        SET   e,0                       e ← 0.
        NEGU  fl,fl
        ZSNZ  carry,fl,1
        ADDU  f,f,carry
        SETH  t,#10; SUBU f,t,f         (f, fl) ← 1 − (f, fl).
        SET   s,0                       (f, fl) > 0.
5H      INCL  e,1023                    5. Add q.
        OR    t,f,fl; BNZ t,:Normalize  6. Normalize if not zero.
6H      POP   0,0                       Else return 0. ▮
```

19. The running time for `Fadd` is $28 - 3[\,|u| < |v|\,] + 4[\,\mathrm{sign}(u) \neq \mathrm{sign}(v)\,]$. The running time for `Normalize` is $4 + [\,u + v \neq 0](22 + 3[\text{fraction overflow}] + 8N + 16[\text{rounding overflow}] - 4[\text{overflow}] - 3[\text{underflow}])$, where N is the number of left shifts during normalization. If there are neither overflows nor underflows and the result is not zero, these formulas simplify to

`Fadd`:	$28 - 3[\|u\| < \|v\|\,] + 4[\,\mathrm{sign}(u) \neq \mathrm{sign}(v)\,]$,
`Normalize`:	$26 + 8N$.

The minimum time for `Fadd` and `Normalize` combined is 51υ. The maximum time is 482υ; it occurs if u and v have opposite signs, $|u| < |v|$, $e_u = e_v$, and $u + v < u/2^{53}$. In this case, the shift-left loop, taking 8υ, runs 53 times. It is tempting to remove the dependency on N by eliminating the loop during normalization (but see exercise 20). [The average time, considering the data in Section 4.2.4, will be about 62.3υ.]

20. Use ‘`MOR t,f,z; MOR t,z,f`’ with `z` ≡ `#0102040810204080` to assign to `t` the bits of `f` in reverse order; then use ‘`SUBU d,t,1; SADD d,d,t`’ to assign to `d` the number of trailing bits of t. This computation will add 4υ to the running time of the normalization routine in place of the loop time. The data in Section 4.2.4 shows, however, that the number of left shifts per normalization is only about 0.9; on average then, adding this computation will make the normalization run slower not faster.

4.2.2. Accuracy of Floating Point Arithmetic

[615]

17. `Fcmpe` is almost like `Fadd` in that it computes $|u - v|$ and compares it to $2^{e-1022}\epsilon$.

```
:Fcmpe  GET    eps,:rE                       Get ϵ.
        SET    su,u                          Sign of u.
        XOR    s,u,v                         Signs different?
        ANDNH  u,#8000; ANDNH v,#8000        Remove sign bits.
        CMPU   x,u,v; BNN x,0F               Compare |u| and |v|.
        SET    t,u; SET u,v; SET v,t         Swap u with v.
0H      CSN    x,s,1                         If signs are different,
        NEG    t,x                             u is larger
        CSN    x,su,t                          unless u < 0.
        SRU    eu,u,52; SRU ev,v,52          Get exponents.
        SETH   t,#FFF0
        ANDN   fu,u,t; ANDN fv,v,t           Get fraction part.
        INCH   fu,#10; INCH fv,#10           Add hidden bit.
        SUBU   d,eu,ev                       Scale right.
        NEG    t,64,d
        CSN    t,t,0                         Keep all low-order bits.
        SLU    f0,fv,t
        SRU    fv,fv,d
        SET    eu,1022                       Divide by 2^(e_u - 1022).
        BN     s,Add                         Add if signs are different;
        NEGU   f0,f0; ZSNZ carry,f0,1          else subtract.
        SUBU   fu,fu,fv; SUBU fu,fu,carry    u ← |u − v| /2^(e_u - 1022).
        OR     t,fu,f0; BZ t,Equal           Jump if |u − v| = 0.
0H      SETH   t,#0010; AND t,fu,t           Normalized?
        BNZ    t,Compare
```

	`SRU`	`carry,f0,63`	
	`SLU`	`fu,fu,1; OR fu,fu,carry`	Adjust left.
	`SLU`	`f0,f0,1`	
	`SUB`	`eu,eu,1`	
	`JMP`	`0B`	
`Add`	`ADDU`	`fu,fu,fv`	$u \leftarrow \lvert u - v \rvert / 2^{e_u - 1022}$.
	`SETH`	`t,#0020; CMP t,fu,t`	Normalized?
	`BN`	`t,Compare`	
	`SLU`	`carry,fu,63`	
	`SRU`	`fu,fu,1; SRU f0,f0,1`	Adjust right.
	`OR`	`f0,f0,carry`	
	`ADD`	`eu,eu,1`	
`Compare`	`ANDNH`	`fu,#FFF0`	Remove hidden bit.
	`SLU`	`eu,eu,52`	
	`OR`	`u,eu,fu`	Combine e_u with f_u and
	`CMPU`	`t,u,eps`	compare to ϵ.
	`CSN`	`x,t,0`	If $u < \epsilon$, then $u \sim v$.
	`CSP`	`f0,t,1`	If $u > \epsilon$, force `f0` $\neq 0$.
`Equal`	`CSZ`	`x,f0,0`	If `f0` $= 0$, then $u \sim v$.
	`SET`	`$0,x`	Return x.
	`POP`	`1,0`	▮

4.2.3. Double-Precision Calculations

[617]

2. Only the two lowest bits in the hi-wyde of `um` are strictly needed during normalization. The hidden bit is tested in step N4, but this bit is set to 1, so there is no need to clear it. The bit left of the hidden bit is tested in step N1, line 37, so it needs to be cleared. Clearing the complete wyde, however, also simplifies the test for zero in line 38.

3. Program M will not cause an overflow exception because it uses "unsigned" instructions; there might be, however, a silent overflow. Working with exponents is safe because exponents are very small; the same holds for the upper 48 bits of the fraction parts. Whenever we work with 64-bit fraction parts, we determine an eventual carry and apply necessary corrections.

In contrast to the implementation of floating point numbers in the `MIX` computer, where both fraction parts are less than 1 and therefore the product is less than 1 as well, `MMIX`'s fraction parts f_u and f_v are in the range $1 \le f_u, f_v < 2$ (due to the hidden bit) and so $1 \le f_u \times f_v < 4$. This might cause an extra increase of the exponent; the normalization routine takes care of this possibility.

4. (a) As can be seen from Fig. 4, using the low 64 bits computed in lines 06 and 08 alone would not improve the precision, because the high 64 bits of $u_l \times v_l$ would still be missing. But the product $u_l \times v_l$ is not computed.

(b) While unpacking, we shift the fraction parts of both operands u and v to the left by 8 bits. The code changes to the following:

```
01  :DFmul   SLU    eu,um,1; SLU ev,vm,1       M1. Unpack.
02           SRU    eu,eu,49; SRU ev,ev,49
03           XOR    s,um,vm; SRU s,s,63        s ← s_u × s_v.
04           ANDNH  um,#FFFF; ORH um,#0001
05           ANDNH  vm,#FFFF; ORH vm,#0001
06           SLU    um,um,8                    Shift (u_m, u_l) left.
07           SRU    carry,ul,64-8
08           ADDU   um,um,carry
09           SLU    ul,ul,8
10           SLU    vm,vm,8                    Shift (v_m, v_l) left.
11           SRU    carry,vl,64-8
```

12	`ADDU`	`vm,vm,carry`	
13	`SLU`	`vl,vl,8`	
14	`MULU`	`t,um,vl`	*M2. Operate.*
15	`GET`	`wl,:rH`	wl $\leftarrow 2^{56}u_m \times 2^{64}\, v_l \times 2^{-64}$.
16	`MULU`	`t,ul,vm`	
17	`GET`	`t,:rH; ADDU wl,wl,t`	wl $\leftarrow$ wl $+\ 2^{56}u_l v_m$.
18	`MULU`	`t,um,vm; GET wm,:rH`	wm $\leftarrow \lfloor 2^{48}u_m \times v_m \rfloor$.
19	`ADDU`	`wl,wl,t`	`wl` $\leftarrow$ `wl + um` $\times$ `vm` $\bmod 2^{64}$.
20	`CMPU`	`t,wl,t; ZSN carry,t,1`	`carry` $\leftarrow 1$ if `wl` $+$ `t` $<$ `t`.
21	`ADDU`	`wm,wm,carry`	
22	`ADD`	`e,eu,ev`	
23	`SET`	`t,#3FFF; SUB e,e,t`	$e \leftarrow e_u + e_v - q$.
24	`JMP`	`:DNormalize`	*M3. Normalize.* ▮

The shifting yields a 50-bit result for `wm` in line 18—just the amount of precision we need. Further, `wm` is still small enough to leave the single shift-right step to the normalization routine if needed. The precision improves by a factor of 2^{16} and the error in the result will be less than $2^{e-q-112}$.

Program M has 28 instructions including 3 multiplications; its running time is 55υ. The new program has 4 additional instructions, each taking 1υ; this increases the running time by about 7 percent to 59υ.

5. We add another register `vll` to keep the lowest bits of v when shifting right. We initialize it to zero after unpacking by adding the following instruction after line 13:

```
        SET     vll,0
```

We replace lines 19–21 by

```
A5   CMP  t,d,64; PBN t,0F                    A5. Scale right.
     SET  vll,vl; SET vl,vm; SET vm,0         Shift right by 64 bits.
     SUB  d,d,64
0H   CMP  t,d,64; PBN t,0F
     SET  vll,vl; SET vl,vm; SET vm,0         Shift right by 64 bits.
     SUB  d,d,64
```

and add after line 22

```
     SRU     vll,vll,d; SLU carry,vl,t; OR vll,vll,carry
```

to accomplish step A5 with three registers.

In case of a subtraction, `vll` must be subtracted from zero and might cause a carry into `wl`. After line 32, we insert the following line:

```
ZSNZ    carry,vll,1; SUBU wl,wl,carry; NEGU vll,vll
```

Next we modify the scale right and scale left steps of the normalization procedure. We replace line 40 with

```
ZSN  t,vll,1; SLU vll,vll,1                                 N3. Scale left.
ZSN  carry,wl,1; SLU wl,wl,1; ADDU wl,wl,t
```

and line 45 with

```
N4    SLU     carry,wl,63; SRU vll,vll,1                N4. Scale right.
      ADDU    vll,vll,carry; SLU carry,wm,63
```

Last but not least, we round the result. The code for step N5 is inserted just before line 50.

```
6H    SETH  t,#8000                 N5. Round.
      CMPU  t,vll,t                 Compare f_l to 1/2.
      CSOD  carry,wl,1              f is odd. Round up if f_l ≥ 1/2.
      CSEV  carry,wl,t              f is even. Round up if f_l > 1/2.
      ZSNN  carry,t,carry           Round down if f_l < 1/2.
      ADDU  wl,wl,carry
      ZSZ   carry,wl,carry
      ADDU  wm,wm,carry
      SET   vll,0
      SRU   t,wm,49; BP t,N4        Rounding overflow.
```

The cost in performance is 6υ for all calls to `DFadd/DFsub` plus 11υ for all calls to `DNormalize`. Further, a scale right step needs an extra 3υ if executed. In case of a subtraction (opposite signs of the operands), the running time increases by $3\upsilon + T3\upsilon$, where T is the number of left shifts executed in step N3. On average, the running time increases by 21υ.

6. The function `ToDouble` expects a single-precision floating point number in register `x` and returns a double-precision floating point number in two registers.

```
:ToDouble   BZ    x,:Zero
            SRU   s,x,63; SLU s,s,63          Extract sign.
            SLU   exm,x,1; SRU exm,exm,5      Position e_x and x_m.
            INCH  exm,#3FFF-#3FF              Adjust exponent.
            SLU   $0,x,64-(52-48)             Extract x_l.
            OR    $1,exm,s                    Add sign bit.
```

	POP	2,0	Return. ▮

The function `ToSingle` expects a double-precision floating point number (f, f_l) as a parameter and returns a single-precision floating point number.

:ToSingle	SRU	s,f,63	Get sign bit.
	SLU	e,f,1; SRU e,e,49	Get exponent.
	SET	t,#3FFF-#3FF-4	
	SUBU	e,e,t	Adjust exponent.
	ANDNH	f,#FFFF	Remove sign and exponent.
	INCH	f,1	Add hidden bit.
	JMP	:Normalize	Normalize, round, and exit. ▮

4.3.1. The Classical Algorithms

[623]

3. We assume that we have four parameters: `u` ≡ `LOC(`u`)`, the address where the first of m numbers each n octabytes wide is stored; then `m` ≡ m; then `w` ≡ `LOC(`w`)`, the address where the result will be stored in $n+1$ octabytes; and finally `n` ≡ n.

:AddC	8ADDU	w,n,w	1	
	SL	j,n,3; NEG j,j	1	$j \leftarrow 0$.
	SET	k,0	1	$k \leftarrow 0$.
	JMP	4F	1	
1H	8ADDU	u,n,u0	N	$i \leftarrow 0$.
	LDOU	t,u,j; ADDU wj,k,t	N	$w_j \leftarrow u_{0j} + k$.
	ZSZ	k,wj,k	N	Carry?
	SET	i,m	N	
	JMP	3F	N	
2H	LDOU	t,u,j; ADDU wj,wj,t	$N(M-1)$	$w_j \leftarrow w_j + u_{ij}$.
	CMPU	t,wj,t; ZSN t,t,1	$N(M-1)$	Carry?
	ADD	k,k,t	$N(M-1)$	
3H	8ADDU	u,n,u	NM	Advance i.
	SUB	i,i,1	NM	
	PBP	i,2B	$NM_{[N]}$	Loop on i.
	STOU	wj,w,j	N	

	ADD	j,j,8	N	$j \leftarrow j + 1$.
4H	PBN	j,1B	$N + 1_{[1]}$	Loop on j.
	STOU	k,w,j	1	$w_n \leftarrow k$.
	POP	0,0		▮

The running time is $(8NM + 6N + 9)\upsilon + (NM + N + 1)\mu$.

8. Given three n-digit numbers u, v, and w, the following subroutine expects four parameters: u ≡ `LOC(`u`)`, v ≡ `LOC(`v`)`, w ≡ `LOC(`w`)`, and n ≡ n. The program will set $w \leftarrow u + v$ using the algorithm of exercise 5.

:Add	SL	j,n,3	1	B1. $j \leftarrow n - 1$.
	STCO	0,w,j	1	$w_n \leftarrow 0$.
	SUB	j,j,8	1	$j \leftarrow n - 1$.
2H	LDOU	wj,u,j	N	B2.
	LDOU	t,v,j; ADDU wj,wj,t	N	$w_j \leftarrow u_j + v_j \bmod b$.
	STOU	wj,w,j	N	
	CMPU	t,wj,t	N	B3.
	PBNN	t,4F	$N_{[L]}$	
	SET	i,j	L	$i \leftarrow j$.
0H	ADD	i,i,8	K	$i \leftarrow i + 1$.
	LDOU	wi,w,i	K	$w_i \leftarrow w_i + 1 \bmod b$.
	ADDU	wi,wi,1	K	
	STOU	wi,w,i	K	
	BZ	wi,0B	$K_{[K-L]}$	Repeat until $w_i + 1 < b$.
4H	SUB	j,j,8	N	B4. $j \leftarrow j - 1$.
	PBNN	j,2B	$N_{[1]}$	If $j \geq 0$, go back to B2.
	POP	0,0		▮

The running time depends on L, the number of positions in which $u_j + v_j \geq b$, and on K, the total number of carries. It is not difficult to see that K is the same quantity that appears in Program A. The analysis in the text shows that L has the average value $N((b-1)/2b)$ and K has the average value $\frac{1}{2}(N - b^{-1} - b^{-2} - \cdots - b^{-n})$. So if we ignore terms of order $1/b$, the running time is $(8N + 7K + L + 5)\upsilon + (3N + 2K + 1)\mu \approx (12N + 5)\upsilon + (4N + 1)\mu$.

10. No. The instruction `CMPU t,wj,vj` compares two unsigned integers w_j and v_j and will set `t` to -1 if $w_j < v_j$; the instruction `SUBU wj,wj,vj` subtracts two unsigned integers w_j and v_j and will set `wj` to $(w_j - v_j) \bmod 2^{64}$. As long as $|w_j - v_j| < 2^{63}$, the difference will be considered negative if $w_j < v_j$; if $|w_j - v_j| \geq 2^{63}$, however, the difference will be considered negative if $w_j > v_j$. The `CMPU` instruction does not suffer from this kind of "overflow."

13. The following subroutine expects four parameters: `u` ≡ `LOC(`u`)`, `v` ≡ v, `w` ≡ `LOC(`w`)`, and `n` ≡ n.

`:MulS`	`4ADDU`	`u,n,u; 4ADDU w,n,w`	1	
	`SL`	`i,n,2; NEG i,i`	1	$i \leftarrow 0$.
	`SET`	`k,0`	1	$k \leftarrow 0$.
`0H`	`LDTU`	`wi,u,i`	N	$w_i \leftarrow u_i$.
	`MUL`	`wi,wi,v`	N	$w_i \leftarrow u_i \times v$.
	`ADD`	`wi,wi,k`	N	$w_i \leftarrow u_i \times v + k$.
	`STTU`	`wi,w,i`	N	$w_i \leftarrow w_i \bmod b$.
	`SRU`	`k,wi,32`	N	$k \leftarrow \lfloor w_i / b \rfloor$.
	`ADD`	`i,i,4`	N	$i \leftarrow i + 1$.
	`PBN`	`i,0B`	$N_{[1]}$	Loop in i.
	`STTU`	`k,w,0`	1	$w_n \leftarrow k$.
	`POP`	`0,0`		▮

The running time is $(16N + 8)\upsilon + (2N + 1)\mu$.

25. As an example, the following subroutine is given with complete details.

	`PREFIX`	`:ShiftLeft:`	
`x`	`IS`	`$0`	`LOC(`x_0`)` } Parameter
`n`	`IS`	`$1`	n } Parameter
`p`	`IS`	`$2`	p } Parameter
`i`	`IS`	`n`	i shares a register with n.

```
q           IS     $3                         64 − p
k           IS     $4                         Carry
xi          IS     $5                         x_i
t           IS     $6                         Temporary variable
:ShiftLeft  NEG    q,64,p                     q ← 64 − p.
            SET    k,0                        k ← 0.
            SLU    i,n,3; ADDU x,x,i; NEG i,i i ← 0.
0H          LDOU   xi,x,i                     Load x_i.
            SLU    t,xi,p; OR t,t,k           Shift and add carry.
            STOU   t,x,i                      Store x_i.
            SRU    k,xi,q                     New carry.
            ADD    i,i,8                      i ← i + 1.
            PBN    i,0B                       Loop on i.
            SET    $0,k                       Return carry.
            POP    1,0                        ▮
```

The running time is $8\upsilon + N(7\upsilon + 2\mu)$.

26. The `ShiftRight` subroutine is very similar to the `ShiftLeft` subroutine.

```
:ShiftRight  NEG   q,64,p              q ← 64 − p.
             SET   k,0 k               k ← 0.
             SLU   i,n,3               i ← n.
             JMP   1F
0H           LDOU  xi,x,i              Load x_i.
             SRU   t,xi,p; OR t,t,k    Shift and add carry.
             STOU  t,x,i               Store x_i.
             SLU   k,xi,q              New carry.
             SUB   i,i,8               i ← i − 1.
1H           PBNN  i,0B                Loop on i.
             SET   $0,k                Return carry.
             POP   1,0                 ▮
```

The running time is $7\upsilon + N(7\upsilon + 2\mu)$.

4.4. RADIX CONVERSION

[636]

8. To replace division by multiplication, we need a value $1/10 < x < 1/10 +$

$1/2^{64}$ in a register. The following code uses a global register x to store $\lceil 2^{64}x \rceil$; it is also possible to load this value into a local register (with an additional 4υ of total running time). As in Program (1), we store the decimal representation of a nonnegative (binary) integer u as an array of BYTE at address U.

```
x      GREG   1+(1<<63)/5            x ← ⌈2^64 × 1/10⌉.
       SET    j,0                    j ← 0.
Loop   MULU   t,u,x; GET ux,rH       ux ← ⌊ux⌋.
       4ADDU  t,ux,ux; SLU t,t,1 t   t ← 10⌊ux⌋.
       SUBU   r,u,t                  r ← u - 10⌊ux⌋.
       PBNN   r,0F
       SUBU   ux,ux,1                (Can occur only on first iteration,
       ADD    r,r,10                    by exercise 7.)
0H     STBU   r,U,j                  U_j ← r = u mod 10.
       SET    u,ux
       ADD    j,j,1                  j ← j + 1.
       PBP    u,Loop                 Repeat until result is zero. ▮
```

The code has a running time of $(19\upsilon + \mu)M + 3\upsilon$. With approximately 19$\upsilon$ per digit, it is about three times faster than Program (1), with 62υ per digit; close to Program (4), with 14υ per digit; and for "small" numbers ($M \le 6$), better than Program (4′), with 128υ for nine digits.

13. We use the multiplication program of exercise 4.3.1–13, with $v = 10^9$ and $w = u$ to get the nine leading decimal digits of u. Then we use 4.4–(4′) to convert these digits to ASCII codes.

```
ToString 4ADDU  u,m,u
         SET    lines,2
         SET    t,'.'                        Start with a decimal point.
1H       STBU   t,buffer; INCL buffer,1
         SET    blocks,7
2H       SL     i,m,3; NEG i,i               i ← 0.
         ⟨See exercise 4.3.1–13, lines 03–10 with w = u.⟩
         SLU    ui,k,32                      ⎫
         ADD    ui,ui,v                      ⎪
         DIV    ui,ui,v                      ⎪
         SET    i,8                          ⎬ See 4.4–(4′).
0H       4ADDU  ui,ui,ui                     ⎪
         SLU    ui,ui,1                      ⎪
         SRU    t,ui,32                      ⎭
         ADD    t,t,'0'                      Convert to ASCII code.
         STB    t,buffer,0; INCL buffer,1
         ANDNMH ui,#FFFF                     ⎫
         SUB    i,i,1                        ⎬ See 4.4–(4′).
         PBNN   i,0B                         ⎭
         SUB    blocks,blocks,1
         SET    t,' '; STBU t,buffer         Insert a space.
         INCL   buffer,1                     Advance to next block.
         BP     blocks,2B
         SET    t,#a; STBU t,buffer          Insert a newline.
         INCL   buffer,1
         SET    t,' '                        Start next line with a space.
         SUB    lines,lines,1                Advance to next line.
         BP     lines,1B
         SET    t,0; STBU t,buffer           Terminate with a zero byte.
         POP    0,0                          ▮
```

19. To convert the ASCII codes to pure numbers, we subtract the ASCII code '`0`' from every byte. Then set m_1 = #`FF00FF00FF00FF00`, m_2 = #`FFFF0000FFFF0000`, m_3 = #`FFFFFFFF00000000`, and $c_i = 1 - (10/256)^{2^{i-1}}$. The division is done by a `SRU` instruction; the multiplication is done by `4ADDU` and `SLU` instructions.

```
ascii   GREG   #3030303030303030          "00000000"
m1      GREG   #FF00FF00FF00FF00
```

```
m2      GREG   #FFFF0000FFFF0000
m3      GREG   #FFFFFFFF00000000
        LDO    u,str
        SUBU   u,u,ascii
        AND    t,u,m1
        SUBU   u,u,t
        4ADDU  t,t,t; SRU t,t,8-1               t ← t × 10/2^8.
        ADD    u,u,t
        AND    t,u,m2
        SUBU   u,u,t
        4ADDU  t,t,t; 4ADDU t,t,t; SRU t,t,16-2  t ← t × 100/2^16.
        ADD    u,u,t
        AND    t,u,m3
        SUBU   u,u,t
        4ADDU  t,t,t; 4ADDU t,t,t; 4ADDU t,t,t
        4ADDU  t,t,t; SRU t,t,32-4              t ← t × 10000/2^32.
        ADD    u,u,t                            ▮
```

The conversion needs 21υ + 1μ, less than half the time needed by (6) for the same eight decimal digits even when (6) is improved to run in 44υ + 8μ.

4.5.2. The Greatest Common Divisor

[647]

43. The replacement has a constant running time of 5υ; step B1 of Program 4.5.2B has a running time of $(8A + 3)$υ. Assuming an average value of $A = \frac{1}{3}$ gives a running time of 5.67υ. In this case, the replacement is only marginally faster, but it can be a good insurance against large values of *k*.

4.5.3. Analysis of Euclid's Algorithm

[647]

1. The running time is about $(44.4T + 3)$υ, which is about 30 percent faster than Program 4.5.2A.

4.6.3. Evaluation of Powers

[691]

2. The following subroutine has two parameters, x and n, and returns x^n mod 2^{64}.

```
A1    SET    y,1        1             A1. Initialize.
      JMP    0F         1
A2    SRU    n,n,1      L+1-K         A2. Halve N. N even.
A5    MULU   z,z,z      L             A5. Square Z.
0H    PBEV   n,A2       L+1[K]        A2. Halve N. N odd.
      SRU    n,n,1      K             N ← ⌊N/2⌋.
      MULU   y,z,y      K             A3. Multiply Y by Z.
      PBNZ   n,A5       K[1]          A4. N = 0?
      SET    $0,y       1             Return Y.
      POP    1,0                      ▮
```

The running time is $(12L + 13K + 7)\upsilon$, where $L = \lambda n = \lfloor \lg n \rfloor$ is one less than the number of bits in the binary representation of n, and $K = \nu n$ is the number of 1 bits in that representation.

The serial program is very simple:

```
A1    SET    y,x        1
      JMP    1F         1
0H    MUL    y,y,x      N-1
1H    SUB    n,n,1      N
      PBP    n,0B       N[1]
      SET    $0,y       1
      POP    1,0              ▮
```

The running time for this program is $(12N - 5)\upsilon$; it is faster than the previous program when $n \le 5$, slower when $n \ge 6$.

4.6.4. Evaluation of Polynomials

[701]

20. Assuming that x and the coefficients α_i are in registers, we can write:

```
FADD  y,x,a0   y ← x + α0.
FMUL  y,y,y    y ← (x + α0)^2.
FADD  u,y,a1   u ← (y + α1).
FMUL  u,u,y    u ← (y + α1)y.
FADD  u,u,a2   u ← (y + α1)y + α2.
FADD  t,x,a3   t ← x + α3.
```

```
FMUL  u,u,t     u ← ((y + α1)y + α2)(x + α3).
FADD  u,u,a4    u ← ((y + α1)y + α2)(x + α3) + α4.
FMUL  u,u,a5    u ← (((y + α1)y + α2)(x + α3) + α4)α5.   ▮
```

5. SORTING

[585]

6. Overflow is possible in the '`SUB $2,$0,$1`' instruction, and it can lead to a false equality indication. He should have written '`CMP $2,$0,$1`'. (The inability to make full-word comparisons by subtraction is a problem on essentially all computers; it is the chief reason for including `CMP`, `CMPU`, and `FCMP` in `MMIX`'s repertoire.)

7. As an example, we show this subroutine in its full length.

```
        PREFIX  :MCmp:      (Begin of local symbols for subroutine MCmp)
n       IS      $0          n > 0      }
a       IS      $1          LOC(a0)    }  Parameters
b       IS      $2          LOC(b0)    }
aj      IS      $3          aj         }
bj      IS      $4          bj         }  Local variables
j       IS      $5          j          }
:MCmp   SUB     j,n,1       :MCmp is a global symbol. j ← n − 1.
0H      LDBU    aj,a,j      Load aj.
        LDBU    bj,b,j      Load bj.
        CMPU    $0,aj,bj    Compare aj and bj.
        BNZ     $0,1F       Jump if $0 is not zero.
        SUB     j,j,1       j ← j − 1.
        PBNN    j,0B        Loop while j ≥ 0.
1H      POP     1,0         Return the value in $0.
        PREFIX  :           (End of local symbols for subroutine MCmp)  ▮
```

8.

```
ODIF  t,a,b; SUB min,a,t; ADD max,b,t             ▮
```

5.2. INTERNAL SORTING

[615]

4. The following code has a running time of $(5N + 6)\upsilon + 3N\mu$.

```
:Finish  SL    i,n,3        1
         JMP   0F           1
1H       LDO   ri,r,i       N
         LDO   ci,count,i   N
         STO   ri,s,ci      N         Counts are already scaled.
0H       SUB   i,i,8        N+1
         PBNN  i,1B         N+1[1]    ■
```

5. The running time is decreased by $(A + 1 - N - B)\upsilon$, and this is almost always an improvement.

9. Let $M = v - u$; assume that a record fits into one octabyte and that the key, in the range from u to v, is stored in the most significant `WYDE` of each record. The following program sorts the records $R_1, \ldots, R_N$ using an auxiliary table `COUNT` of size $M + 1$. The sorted records are written to an output area $S_1, \ldots, S_N$. We maintain two pointers to the array of counters: `count0` points to the fictive counter for the key value zero, and `countv` points to the counter for the key value v. We use the first one as base address with K_j as index, keeping in register `kj` the value of $8K_j$ and we use the second with j and i as index, keeping in registers `i` and `j` the values of $8(v - j)$ and $8(v - i)$, respectively. Further, we assume `key` ≡ `LOC(`K_1`)`, `count` ≡ `LOC(COUNT[1])`, `s` ≡ `LOC(`S_1`)`, `n` ≡ N, `u` ≡ u, and `v` ≡ v.

```
:Sort  NEG    t,u                1
       8ADDU  count0,t,count     1        count0 ← count − 8u.
       8ADDU  countv,v,count0    1        countv ← count0 + 8v.
       SUBU   i,count,countv     1        D1. Clear COUNTs. i ← u.
       JMP    0F                 1
1H     STCO   0,countv,i         M+1      COUNT[j] ← 0.
       ADD    i,i,8              M+1      i ← i + 1.
0H     PBNP   i,1B               M+1[1]   u ≤ i ≤ v.
       SL     j,n,3              1        D2. Loop on j. j ← N + 1.
       JMP    2F                 1
3H     LDWU   kj,key,j           N        D3. Increase COUNT[K_j].
       SL     kj,kj,3            N
       LDO    c,count0,kj        N        COUNT[K_j]
       ADD    c,c,8                         + 1
```

			N	
	STO	c,count0,kj	N	$\rightarrow$ COUNT$[K_j]$.
2H	SUB	j,j,8	$N+1$	$j \leftarrow j-1$.
	PBNN	j,3B	$N+1_{[1]}$	$N > j \geq 0$.
	SUB	i,count,countv	1	D4. Accumulate. $i \leftarrow u$.
	LDO	c,countv,i	1	$c \leftarrow$ COUNT$[i]$.
	JMP	4F	1	
0H	LDO	ci,countv,i	M	COUNT$[i]$
	ADD	c,ci,c	M	$+$ COUNT$[i-1]$
	STO	c,countv,i	M	$\rightarrow$ COUNT$[i]$.
4H	ADD	i,i,8	$M+1$	$i \leftarrow i+1$.
	PBNP	i,0B	$M+1_{[1]}$	$u \leq i \leq v$.
	SL	j,n,3	1	D5. Loop on j. $j \leftarrow N$.
	JMP	5F	1	
6H	LDOU	rj,key,j	N	D6. Output R_j.
	SRU	kj,rj,48-3	N	Extract $8K_j$.
	LDO	i,count0,kj	N	$i \leftarrow$ COUNT$[K_j]$.
	SUB	i,i,8	N	$i \leftarrow i-1$.
	STO	i,count0,kj	N	COUNT$[K_j] \leftarrow i$.
	STOU	rj,s,i	N	$S_i \leftarrow R_j$.
5H	SUB	j,j,8	$N+1$	$j \leftarrow j-1$.
	PBNN	j,6B	$N+1[1]$	■

The running time is $(15N + 8M + 29)\upsilon + (7N + 3M + 2)\mu$.

11. We assume key ≡ LOC(K_1), p ≡ LOC$(p(1))$, and n ≡ N. Further, we use i ≡ i, j ≡ j, k ≡ k, ii ≡ $8i$, jj ≡ $8j$, and kk ≡ $8k$. The program would be simpler if we could assume that the permutation p uses already scaled values.

P1	SET	i,n	1	P1. Loop on i.
	JMP	0F	1	
P2	SL	ii,i,3	N	P2. Is p(i) = i?
	LDO	pi,p,ii	N	
	CMP	eq,pi,i	N	
	BZ	eq,0F	$N_{[N-(A-B)]}$	Jump if $p(i) = i$.

	LDO	t,key,ii	$A-B$	P3. Begin cycle. $t \leftarrow R_i$.
	SET	j,i; SET jj,ii	$A-B$	$j \leftarrow i$.
P4	LDO	k,p,jj	$N-A$	P4. Fix R_j. $k \leftarrow p(j)$.
	SL	kk,k,3	$N-A$	
	LDO	rk,key,kk	$N-A$	
	STO	rk,key,jj	$N-A$	$R_j \leftarrow R_k$.
	STO	j,p,jj	$N-A$	$p(j) \leftarrow j$.
	SET	j,k; SET jj,kk	$N-A$	$j \leftarrow k$.
	LDO	pj,p,jj	$N-A$	
	CMP	eq,pj,i	$N-A$	
	PBNZ	eq,P4	$N-A_{[A-B]}$	Repeat if $p(j) \neq i$.
	STO	t,key,jj	$A-B$	P5. End cycle. $R_j \leftarrow t$.
	STO	j,p,jj	$A-B$	$p(j) \leftarrow j$.
0H	SUB	i,i,1	$N+1$	
	PBNN	i,P2	$N+1_{[1]}$	$N > i \geq 0$. ▮

The running time is $(18N-5A-5B+6)\upsilon + (6N-2A-3B)\mu$, where A is the number of cycles in the permutation $p(1)\ldots p(N)$ and B is the number of fixed points (1-cycles).

We have

$$A = (\min 1, \text{ave } H_N, \max N, \text{dev } \sqrt{H_N - H_N^{(2)}})$$

and

$$B = (\min 0, \text{ave } 1, \max N, \text{dev } 1),$$

for $N \geq 2$, by Eqs. 1.3.3–(21) and 1.3.3–(28).

12. . . .

The following subroutine implements MacLaren's algorithm. It assumes records that consist of two octabytes—first the `LINK` field, then the `KEY` field. It expects the list head in the `LINK` field of an artificial record R_0 preceding record R_1. Further, all `LINK` fields contain relative addresses with `LOC(`R_0`)` as base address. The parameter of the subroutine is `link` ≡ `LOC(LINK(`R_0`))` = `LOC(HEAD)`.

M1	LDOU	p,link,0	1	M1. Initialize. `P` ← `HEAD`.
	SET	k,16	1	$k \leftarrow 1$.

	ADDU	key,link,KEY	1	
	JMP	M2	1	
0H	LDOU	p,link,p	A	P ← LINK(P).
M3	CMPU	t,p,k	$N+A$	*M3. Ensure P is at least k.*
	BN	t,0B	$N_{[A]}$	
	LDOU	t,key,k	N	*M4. Exchange.*
	LDOU	kp,key,p	N	
	STOU	t,key,p	N	
	STOU	kp,key,k	N	
	LDOU	t,link,k	N	
	LDOU	q,link,p	N	Q ← LINK(k).
	STOU	t,link,p	N	
	STOU	p,link,k	N	LINK(k) ← P.
	SET	p,q	N	P ← Q.
	ADDU	k,k,16	N	$k \leftarrow k+1$.
M2	PBNZ	p,M3	$N+1_{[1]}$	*M2. Done?*
	POP	0,0		▮

The total running time is $(13N + 4A + 7)\upsilon + (8N + A + 1)\mu$.

5.2.1. Sorting by Insertion

[618]

3. The following program is conjectured to be the shortest general-purpose MMIX sorting subroutine, although it is not recommended for speed. The routine sorts only BYTE values; otherwise, an additional SL instruction is necessary after line 09 to scale i by the size of the records. A ten-instruction sorting subroutine is possible in the special case where the base address key of the keys is zero. In this case, the ADD in line 07 can be merged with the STB in line 08 to a single STB s,i,1.

01	2H	LDB	r,key,i	B	$r \leftarrow K_i$.
02		SUB	i,i,1	B	Decrement i.
03		LDB	s,key,i	B	$s \leftarrow K_{i-1}$.
04		CMP	t,s,r	B	
05		BNP	t,1F	$B+2A$	Continue if $K_{i-1} \le K_i$;
06		STB	r,key,i	A	else swap K_i

	ADD	i,i,1	A	with K_{i-1}
	STB	s,key,i	A	and start from the beginning.
:Sort	SUB	i,n,1	$A+1$	Initialize $i \leftarrow n-1$.
1H	BNN	i,2B	$B+3$	Loop while $i \geq 0$.
	POP	0,0		▮

Note: The analyses of the MIX and the MMIX programs are the same. The average running time of the MMIX program is roughly $\frac{2}{3}N^3\upsilon + \frac{2}{9}N^3\mu$.

10. Change the loop in lines 12–20 to:

12		LDO	ki,key,i	$NT-S$	D4. Compare $K : K_i$.
13		CMP	c,k,ki	$NT-S$	
14		BNN	c,7F	$NT-S_{[C]}$	If $K_j \geq K_{j-h}$, jump to increment j.
15	D5	STO	ki,keyh,i	B	D5. Move R_i, decrease i.
16		SUB	i,i,h	B	$i \leftarrow i-h$.
17		BN	i,D6	$B_{[A]}$	To D6 if $i < 0$.
18		LDO	ki,key,i	$B-A$	D4. Compare $K : K_i$.
19		CMP	c,k,ki	$B-A$	
20		PBN	c,D5	$B-A_{[NT-S-C-A]}$	To D5 if $K < K_i$.
21	D6	STO	k,keyh,i	$NT-S-C$	D6. R into R_{i+1}.
22	7H	ADD	j,j,8	$NT-S$	$j \leftarrow j+1$.
23	0H	PBN	j,D3	$NT-S+T_{[T]}$	To D3 if $j < N$. ▮

For a net increase of three instructions, this saves $C\upsilon$, where C is the number of times $K_j \geq K_{j-h}$. In Tables 3 and 4 the time saved is 33υ and 29υ, respectively; . . .

[624]

31. The following MMIX program implements Pratt's sorting algorithm.

:Sort	8ADDU	keyn,n,key	1	keyn $\leftarrow$ LOC(K_{N+1}).
	SL	n,n,3	1	Scale N.
	SL	s,t,3	1	$s \leftarrow t-1$.
	JMP	1F	1	
2H	LDO	h,inc,s	T	
	SL	h,h,3	T	Scale h.
	SUB	keyh,keyn,h	T	keyh $\leftarrow$ LOC(K_{h+1}).
	SET	m,h	T	Loop on m.

09		JMP	0F	T	
10	3H	LDO	k,keyn,j	$NT - S - B + A$	Load and compare $K_j : K_{j-h}$.
11		LDO	kh,keyh,j	$NT - S - B + A$	
12		CMP	c,k,kh	$NT - S - B + A$	
13		PBNN	c,7F	$NT - S - B + A_{[B]}$	Jump if $K_j \geq K_{j-h}$.
14		STO	kh,keyn,j	B	Exchange K_j and K_{j-h}.
15		STO	k,keyh,j	B	
16		ADD	j,j,h	B	Increment j.
17	7H	ADD	j,j,h	$NT - B + A$	Increment j.
18		PBN	j,3B	$NT - B + A_{[S]}$	$m < j + N < N$.
19	0H	SUB	m,m,8	$T + S$	Decrement m.
20		SUB	j,m,n	$T + S$	$j \leftarrow n$.
21		PBNN	m,7B	$T + S_{[T]}$	$0 \leq m < h$.
22	1H	SUB	s,s,8	$T + 1$	Loop on s.
23		PBNN	s,2B	$T + 1_{[1]}$	$0 \leq s < t$. ▮

Here A is related to right-to-left maxima in the same way that A in Program D is related to left-to-right minima; both quantities have the same statistical behavior. The simplifications in the inner loop have cut the running time to $(6NT + 6A - B + S + 12T + 8)\upsilon + (2NT + 2A + T - 2S)\mu$. Curiously, the number of load/store operations is independent of B.

When $N = 8$ the increments are 6, 4, 3, 2, 1, and we have $A_{\text{ave}} = 3.892$, $B_{\text{ave}} = 6.762$; the average total running time is $280.59\upsilon + 43.78\mu$. (Compare with Table 5.) Both A and B are maximized in the permutation 7 3 8 4 5 1 6 2. When $N = 1000$ there are 40 increments, 972, 864, 768, 729, . . . , 8, 6, 4, 3, 2, 1; empirical tests like those in Table 6 give $A \approx 875$, $B \approx 4250$, and a total time of about $250533\upsilon + 63700\mu$ (more than twice as long as Program D with the increments of exercise 28). Since many increments are larger than $N/2$, some time is wasted in the loop from line 17 to line 21 until $j = m + h < N$. These iterations can be avoided by inserting the following instructions before line 09:

```
SL    c,m,1; CMP c,c,n; BNP c,0F; SUB m,n,h
```

This will improve the running time by about 8 percent.

33. Two types of improvements can be made. First, by adding the artificial key ∞ at the end of the list, we can omit testing whether or not $p = 0$. (This idea has been used, for example, in Algorithm 2.2.4A.) Secondly, a standard optimization technique: We can make two copies of the inner loop with the register assignments for `p` and `q` interchanged; this avoids the assignment `SET q,p`. (This idea has been used in exercise 1.1–3.)

We put the largest possible value in the key field of R_0, and initialize the link fields of R_0 and R_N to form a circular list (there is no test for the end of the list anyway).

`:Sort`	`ADDU`	`key,link,KEY`	1	*L1. Loop on j.*
	`SL`	`j,n,4`	1	$j \leftarrow N$.
	`NEG`	`t,1; SRU t,t,1`	1	t ← the largest signed 64-bit number.
	`STO`	`t,key,0`	1	$K_0 \leftarrow \infty$. ;-)
	`STOU`	`j,link,0`	1	$L_0 \leftarrow N$.
	`STCO`	`0,link,j`	1	$L_N \leftarrow 0$.
	`JMP`	`0F`	1	Go to decrease j.
`L2`	`SET`	`q,0`	$N-1$	*L2. Set up p, q, K.* $p \leftarrow L_0$.
	`LDO`	`k,key,j`	$N-1$	$K \leftarrow K_j$.
`4H`	`LDOU`	`p,link,q`	B'	*L4. Bump p, q.*
	`LDO`	`kp,key,p`	B'	*L3. Compare* $K : K_p$.
	`CMP`	`t,k,kp`	B'	
	`BNP`	`t,L5`	$B'_{[N']}$	To L5 if $K \le K_p$.
	`LDOU`	`q,link,p`	B''	*L4. Bump q, p.*
	`LDO`	`kp,key,q`	B''	*L3. Compare* $K : K_q$.
	`CMP`	`t,k,kp`	B''	
	`PBP`	`t,4B`	$B''_{[N'']}$	To L5 if $K \le K_q$.
	`STOU`	`j,link,p`	N''	*L5. Insert into list.* $L_p \leftarrow j$.
	`STOU`	`q,link,j`	N''	$L_j \leftarrow q$.
`0H`	`SUB`	`j,j,16`	$N''+1$	$j \leftarrow j-1$.
	`PBP`	`j,L2`	$N''+1_{[A']}$	$N > j \ge 1$.
	`POP`	`1,0`		
`L5`	`STOU`	`j,link,q`	N'	*L5. Insert into list.* $L_q \leftarrow j$.
	`STOU`	`p,link,j`	N'	$L_j \leftarrow p$.
`0H`	`SUB`	`j,j,16`	N'	$j \leftarrow j-1$.
	`PBP`	`j,L2`	$N'_{[A'']}$	$N > j \ge 1$.
	`POP`	`1,0`		▮

Here $B' + B'' = B + N - 1$, $N' + N'' = N - 1$, $A' + A'' = 1$ so the total running time is $(4B + 12N)\upsilon + (2B + 5N - 2)\mu$.

The ∞ trick also speeds up Program S. Unlike `MIX`, however, `MMIX` does not feature a nifty `MOVE` instruction that loads, stores, and increments all in one instruction. The following code simplifies Program S, because j can run down to zero, while i, which is not tested for the end of the array, runs upward, assuming that the last element of the array already contains the largest possible value.

```
:Sort  SUBU  key0,key,8             1             key0 ← LOC(K_0).
       SL    j,n,3; SUB j,j,16      1             j ← N − 1.
       JMP   S1                     1
S2     ADD   i,j,8                  N − 1         S2. Set up j, K, R.
       LDO   k,key,j                N − 1
       JMP   S3                     N − 1
S4     STO   ki,key0,i              B             S4. Move Ri, increase i.
       ADD   i,i,8                  B
S3     LDO   ki,key,i               B + N − 1     S3. Compare K : K_i.
       CMP   t,k,ki                 B + N − 1
       PBP   t,S4                   B + N − 1[N−1]
       STO   k,key0,i               N − 1         S5. R into R_{i−1}.
       SUB   j,j,8                  N − 1
S1     PBNN  j,S2                   N[1]          S1. Loop on j. ▮
```

The running time is reduced to $(5B + 11N - 4)\upsilon + (2B + 3N - 3)\mu$. Doubling the inner loop will not produce any further savings.

35. Passing `head` ≡ `LOC(`H_1`)` and `m` ≡ M as parameters, as in Program M, we have the following subroutine:

```
:ListCat  SL    j,m,3; SUB j,j,8    1              j ← M.
          LDOU  tail,head,j         1              Initialize tail.
          JMP   0F                  1
1H        LDOU  hj,head,j           M − 1          hj ← LOC(H_j).
          BZ    hj,0F               M − 1[E]       Skip empty heads.
          SET   q,hj                M − 1 − E
2H        SET   p,q                 N − L          Bump p and q.
          LDOU  q,link,p            N − L
          PBNZ  q,2B                N − L[M-1-E]
          STOU  tail,link,p         M − 1 − E      Concatenate lists.
```

```
            SET   tail,hj       M − 1 − E   Advance to the next list.
      0H    SUB   j,j,8         M           j ← j − 1.
            PBNN  j,1B          M[1]        Loop on j.
            STOU  hj,head,0     1
            POP   0,0                       ▮
```

The running time depends not only on the number of list heads M and the number of elements N, but also on E, the number of list heads with an empty list, and on L, the length of the list with the biggest elements H_{m-1}. The total running time is $(3N - 3L + 9M - 3E)\upsilon + (N - L + 2M - E)\mu$. For equally distributed keys, we can assume $L = N/M$. There are M^N ways to map N keys to M lists and $(M - 1)^N$ ways to map N keys to M lists while leaving list j empty; therefore the probability of list j being empty is $(M - 1)^N / M^N$ and we should expect ave $E = M(M - 1)^N / M^N$. Using $\lim_{M\to\infty}((M - 1)/M)^M = 1/e$, we conclude that for large N and $M = \alpha N$, ave E approaches $Me^{-1/\alpha}$. In summary, the running time approaches $((3 + 9\alpha - 3\alpha e^{1/\alpha})N - 3/\alpha)\upsilon + ((1 + 2\alpha - \alpha e^{1/\alpha})N - 1/\alpha)\mu$.

Note: If Program M were modified to keep track of the current end of each list in an array at location `tail`, by inserting first ‘`STCO 0,tail,i`’ between lines 03 and 04, and then ‘`STOU j,tail,i`’ between lines 21 and 22, we could save time by hooking the lists together as in Algorithm 5.2.5H.

36. Program L: $A = 3$, $B = 41$, $N = 16$, time $= 426\upsilon + 156\mu$. Program M: $A = 2 + 1 + 2 + 2 = 7$, $B = 2 + 2 + 2 + 1 = 7$, $N = 16$; as given, the running time of Program M is $446\upsilon + 91\mu$. The multiplications are slow! Folding the multiplication by $M = 4$ into the following shift, as suggested in the text, improves the time to $286\upsilon + 91\mu$. (We should also add the time needed by exercise 35, $78\upsilon + 22\mu$, in order to make a strictly fair comparison. Notice also that the improved Program L in exercise 33 takes only $356\upsilon + 160\mu$.)

5.2.2. Sorting by Exchanging

[629]

12. The following program maintains scaled values of j, p, q, d, and r, in order to use them as offsets into an array of octabytes with base address `key` $\equiv$ `LOC(`K_1`)`. Instead of d, keeping the address of K_d in register `d` is more convenient. Aside from moving the test of the loop condition to the bottom of each loop, the following code is a simple translation of Algorithm M.

:Sort	FLOTU	t,ROUND_UP,n	1	M1. Initialize p.
	SETH	c,#FFF0	1	
	NOR	c,c,c	1	
	ADDU	t,t,c	1	Round N up to 2^t.
	SRU	t,t,52	1	Extract t.
	ANDNL	t,#400	1	t ← $\lceil \lg N \rceil - 1$.
	8ADDU	keyn,n,key	1	keyn ← LOC$(K_N{+}1)$.
	SET	p,8	1	$p \leftarrow 1$.
	SLU	p,p,t	1	$p \leftarrow p \cdot 2^t$.
M2	SET	q,8	T	M2. Initialize q, r, d.
	SL	q,q,t	T	$q \leftarrow 2^t$.
	SET	r,0	T	r ← 0.
	ADDU	d,p,key	T	$d \leftarrow p$.
	JMP	M3	T	
M5	ADDU	d,key,d	$A - T$	M5. Loop on q.
	SR	q,q,1	$A - T$	$q \leftarrow q/2$.
	ANDNL	q,7	$A - T$	$q \leftarrow 8 \cdot \lfloor q/8 \rfloor$.
	SET	r,p	$A - T$	$r \leftarrow p$.
M3	SUB	i,keyn,d	A	M3. Loop on i. $i \leftarrow N + 1 - d$.
	JMP	0F	A	
1H	AND	c,i,p	$AN - D$	
	CMP	c,c,r	$AN - D$	If i & $p = r$,
	BNZ	c,0F	$AN - D_{[AN\text{-}D\text{-}C]}$	go to M4.
	LDO	k,key,i	C	M4. Compare/exchange
	LDO	kd,d,i	C	$R_{i+1} : R_{i+d+1}$.
	CMP	c,k,kd	C	
	PBNP	c,0F	$C_{[B]}$	If $K_i + 1 > K_{i+d+1}$,
	STO	k,d,i	B	interchange R_{i+d+1}
	STO	kd,key,i	B	and R_{i+1}.
0H	SUB	i,i,8	$AN + A - D$	$i \leftarrow i - 1$.
	PBNN	i,1B	$AN + A - D_{[A]}$	$0 \le i < N - d$.
	SUB	d,q,p	A	M5. Loop on q. $d \leftarrow q - p$.
	PBNZ	d,M5	$A_{[T]}$	
	SR	p,p,1	T	M6. Loop on p. $p \leftarrow p/2$.

	ANDNL	p,7	T	p $\leftarrow 8 \cdot \lfloor p/8 \rfloor$.
	PBP	p,M2	T[1]	
	POP	0,0		▮

The running time depends on six quantities, only one of which depends on the input data (the remaining five are functions of N alone): $T = t$, the number of "major cycles"; $A = t(t + 1)/2$, the number of passes or "minor cycles"; $B =$ the (variable) number of exchanges; $C =$ the number of comparisons; $D =$ the number of blocks of consecutive comparisons; and $E =$ the number of incomplete blocks. When $N = 2^t$, it is not difficult to prove that $D = (t-2)N + t + 2$ and $E = 0$. For Table 1, we have $T = 4$, $A = 10$, $B = 3+0+1+4+0+0+8+0+4+5 = 25$, $C = 63$, $D = 38$, $E = 0$, so the total running time is $(7NA + 12A + 4B + 2C - 7D + 6T + 14)\upsilon + (2B + 2C)\mu = 1238\upsilon + 176\mu$.

In general when $N = 2^{e_1} + \cdots + 2^{e_r}$, Panny has shown that $D = e_1(N + 1) - 2(2^{e_1} - 1)$, $E = \binom{e_1 - e_r}{2} + (e_1 + e_2 + \cdots + e_{r-1}) - (e_1 - 1)(r - 1)$.

Using the observation by Panny that step M4 is performed for $i = r + 2kp + s$, $k \geq 0$, and $0 \leq s < p$, we have the following program. It maintains $r + d$ in a register instead of r, because the only use of r is in adding it to d when computing the initial value of i.

:Sort	FLOTU	t,ROUND_UP,n	1	M1. Initialize p.
	SETH	c,#FFF0	1	
	NOR	c,c,c	1	
	ADDU	t,t,c	1	Round N up to 2^t.
	SRU	t,t,52	1	Extract t.
	ANDNL	t,#400	1	$t \leftarrow \lceil \lg N \rceil - 1$.
	8ADDU	keyn,n,key	1	keyn $\leftarrow$ LOC(K_{N+1}).
	SET	w,8	1	w $\leftarrow 1$.
	SLU	p,w,t	1	p $\leftarrow 2^t$.
	SL	n,n,3	1	Scale n.
M2	SL	q,w,t	T	M2. Initialize q, r, d. $q \leftarrow 2^t$.
	ADD	r,p,0	T	$r \leftarrow 0$.
	SUBU	d,keyn,p	T	$d \leftarrow p$.
3H	SUB	i,r,n	A	$i \leftarrow r$.
8H	SUB	s,p,w	$D + E$	$s \leftarrow 0$.
M4	LDO	k,d,i	C	M4. Compare/exchange.

	LDO	kd,keyn,i	C	$R_{i+1} : R_{i+d+1}$.
	CMP	c,k,kd	C	
	PBNP	c,0F	$C_{[B]}$	If $K_i + 1 > K_{i+d+1}$,
	STO	kd,d,i	B	interchange R_{i+d+1}
	STO	k,keyn,i	B	and R_{i+1}.
0H	PBNP	s,7F	$C_{[C-D]}$	Jump if $s = p - 1$.
	ADD	i,i,w	$C - D$	$i \leftarrow i + 1$.
	SUB	s,s,w	$C - D$	$s \leftarrow s - 1$.
	PBN	i,M4	$C - D_{[E]}$	Repeat loop if $i + d < N$.
	JMP	5F	E	Otherwise, go to M5.
7H	ADD	i,i,p	D	
	ADD	i,i,w	D	$i \leftarrow i + p + 1$.
	PBN	i,8B	$D_{[A-E]}$	Repeat loop if $i + d < N$.
5H	SUB	d,q,p	A	
	BZ	d,M6	$A_{[T]}$	
	ADD	r,d,p	$A - T$	M5. Loop on q. $r \leftarrow p$.
	SUBU	d,keyn,d	$A - T$	
	SR	q,q,1	$A - T$	$q \leftarrow q/2$.
	ANDNL	q,7	$A - T$	$q \leftarrow 8 \cdot \lfloor q/8 \rfloor$.
	JMP	3B	$A - T$	
M6	SR	p,p,1	T	M6. Loop on p. $p \leftarrow p/2$.
	ANDNL	p,7	T	$p \leftarrow 8 \cdot \lfloor p/8 \rfloor$.
	PBP	p,M2	$T_{[1]}$	▮

The total running time is $(10A + 4B + 10C - D + 2E + 3T + 15)\upsilon + (2B + 2C)\mu$. For Table 1, we have $819\upsilon + 176\mu$.

Using Panny's formula, k and s can be precomputed before entering the loop, thereby reducing the number of tests in each loop to one. The time invested in this optimization is, however, recovered in the loop only for large N.

[634]

34. We can avoid testing whether or not $i \le j$, as soon as we have found at least one 0 bit and at least one 1 bit in each stage—that is, after making the first exchange in each stage. To do so, replace lines 06–19 of Program R by

	JMP	R3B	A	
R5	LDO	kj,j,8	$C'' - D'' - X$	R5. Inspect K_{j+1} for 0.

	`AND`	`t,kj,b`	$C'' - D'' - X$	
	`BNZ`	`t,R6B`	$C'' - D'' - X_{[C''-D''-A]}$	To R6B if it is 1.
	`ADDU`	`i,j,d`	$A - X$	
`R7`	`STO`	`ki,j,8`	B	*R7. Exchange R_i, R_{j+1}.*
	`STO`	`kj,i,0`	B	
`R4A`	`ADD`	`i,i,8`	D'	*R4′. Increase i.* $i \leftarrow i + 1$.
	`LDO`	`ki,i,0`	D'	*R3′. Inspect K_i for 1.*
	`AND`	`t,ki,b`	D'	
	`PBZ`	`t,R4A`	$D'_{[B]}$	To R4A if it is 0.
`R6A`	`SUBU`	`j,j,8`	D''	*R6′. Decrease j.* $j \leftarrow j - 1$.
	`LDO`	`kj,j,8`	D''	*R5′. Inspect K_{j+1} for 0.*
	`AND`	`t,kj,b`	D''	
	`BNZ`	`t,R6A`	$D''_{[D''-B]}$	To R6A if it is 1.
	`SUB`	`d,i,j`	B	
	`PBNP`	`d,R7`	$B_{[A-X]}$	To R7 if $i \le j$;
	`ADDU`	`j,j,8`	$A - X$	else adjust j
	`JMP`	`R8`	$A - X$	and continue with R8.
`R4B`	`ADD`	`d,d,8`	$C' - D' - A$	*R4. Increase i.*
	`BP`	`d,R8`	$C' - D' - A_{[X']}$	To R8 if $i > j$.
`R3B`	`LDO`	`ki,j,d`	$C' - D' - X'$	*R3. Inspect K_i for 1.*
	`AND`	`t,ki,b`	$C' - D' - X'$	
	`PBZ`	`t,R4B`	$C' - D' - X'_{[A-X']}$	To R4B if it is 0.
`R6B`	`SUBU`	`j,j,8`	$C'' - D'' - X'$	*R6. Decrease j.*
	`ADD`	`d,d,8`	$C'' - D'' - X'$	
	`PBNP`	`d,R5`	$C'' - D'' - X'_{[X'']}$	To R8 if $i > j$.

Here $X = X' + X''$ is the number of times $j < i$ before the first exchange, $C' + C''$ is the number of bit inspections before the first exchange, and $D' + D''$ is the number of bit inspections after the first exchange. Assuming $C' \approx C''$, $D' \approx D''$, and $X' \approx X''$, the new program saves $3D' - 2A - 2B + 12X$ compared to Program R. With random bits, the initial loops need an average of 2 bit-inspections each to reach the first exchange. Neglecting the cases where the loops end prematurely because $j < i$, we have $\text{ave}(D' + D'') = C - 4A$. With case (ii) data (see page *127*), the improved program is approximately $(N \ln N - 8N)/\ln 2 + 6N \approx 1.44N \ln N - 5.5N$ cycles faster.

As an alternative, we can apply the optimizations used in Program Q; add a key with all zeros and a key with all ones to the left and right of the array, respectively; and finish with a final run of straight insertion sort (see also exercise 40). This yields the following program:

:Sort	CMP	t,n,M	1	R1. Initialize.
	BNP	t,S1	1	To straight insertion sort, if $N \le M$.
	GET	rJ,:rJ	1	
	SUBU	t+1,key,8	1	$l \leftarrow 0$.
	8ADDU	t+2,n,0	1	$j \leftarrow N + 1$.
	SET	t+3,b	1	$b \leftarrow b$.
	PUSHJ	t,R2	1	To radix exchange sort.
	PUT	:rJ,rJ	1	
	JMP	S1	1	To straight insertion sort.
R2	SET	i,0	A	R2. Begin new stage. $i \leftarrow l$.
	SET	r,j	A	$r \leftarrow j$.
	JMP	0F	A	
R7	STO	ki,l,j	B	R7. Exchange K_i, K_j.
	STO	kj,l,i	B	
R6	SUB	j,j,8	$C'' - A$	R6. Decrease j. $j \leftarrow j - 1$.
0H	LDO	kj,l,j	C''	R5. Inspect K_j for 0.
	AND	t,kj,b	C''	
	PBNZ	t,R6	$C''_{[A+B]}$	To R4 if it is 0.
R4	ADD	i,i,8	C'	R4. Increase i. $i \leftarrow i + 1$.
	LDO	ki,l,i	C'	R3. Inspect K_i for 1.
	AND	t,ki,b	C'	
	PBZ	t,R4	$C'_{[A+B]}$	To R8 if it is 1.
	CMP	t,i,j	$A + B$	R8. Test special cases.
	PBN	t,R7	$A + B_{[A]}$	To R7 if i $< j$.
	BOD	b,R10	$A_{[G]}$	To R10 if $m \le 0$.
	SR	b,b,1	$A - G$	$m \leftarrow m - 1$.
	SUB	d,r,j	$A - G$	$d \leftarrow r - j$.
	CMP	t,j,8*M	$A - G$	
	BNP	t,0F	$A - G_{[R]}$	Jump if left subfile is too small.
	CMP	t,d,8*M	$A - G - R$	
	BNP	t,R2	$A - G - R_{[L]}$	Jump if right subfile is too small.
	GET	rJ,:rJ	S	Now $j > r - j > M + 1$.

	`ADDU`	`t+1,l,j`	S	*R9. Put on stack.* To R2 with
	`SET`	`t+2,d`	S	$l \leftarrow l + j$, $j \leftarrow r - j$,
	`SET`	`t+3,b`	S	2^{b-1},
	`PUSHJ`	`t,R2`	S	and $(l, j,$ `rJ`$) \Rightarrow$ stack.
	`PUT`	`:rJ,rJ`	S	
	`JMP`	`R2`	S	To R2 with l and j.
`0H`	`CMP`	`t,d,8*M+8`	R	
	`PBNP`	`t,R10`	$R_{[R-K]}$	Jump if right subfile is too small.
	`ADD`	`l,l,j`	$R - K$	Now $r - j > M \geq j - 0$.
	`SET`	`j,d`	$R - K$	
	`JMP`	`R2`	$R - K$	To R2 with $l + j$ and $r - j$.
`R10`	`POP`	`0,0`	S	*R10. Take off stack.*
`S1`	`SL`	`j,n,3`	1	*S1. Loop on j.*
	`SUBU`	`key0,key,8`	1	`key0` $\leftarrow$ `LOC`(K_0).
	`SUB`	`j,j,8`	1	$j \leftarrow j - 1$.
	`JMP`	`0F`	1	
`S3`	`LDO`	`ki,key,j`	$N - 1$	*S3.* $k_i \leftarrow K_j$.
	`SUB`	`j,j,8`	$N - 1$	$j \leftarrow j - 1$.
	`LDO`	`kj,key,j`	$N - 1$	$k_j \leftarrow K_j$.
	`CMP`	`t,kj,ki`	$N - 1$	Compare $K_j : K_i$.
	`BNP`	`t,0F`	$N - 1_{[N-1-D]}$	Done if $K_j \leq K_{j+1}$.
	`ADD`	`i,j,8`	D	$i \leftarrow j + 1$.
`S4`	`STO`	`ki,key0,i`	E	*S4.* Move K_i.
	`ADD`	`i,i,8`	E	Increase i.
	`LDO`	`ki,key,i`	E	$k_i \leftarrow K_i$.
	`CMP`	`t,kj,ki`	E	Compare $K_j : K_i$.
	`PBP`	`t,S4`	$E_{[D]}$	Loop while $K_j > K_i$.
	`STO`	`kj,key0,i`	D	*S5.* $K_{i+1} \leftarrow K_j$.
`0H`	`PBP`	`j,S3`	$N_{[1]}$	Continue while $j > 0$. ▮

The program can be analyzed using the quantities A, B, C, G, R, L, K, and N as in Program R, together with the quantities D, E, and M as in Program Q. Looking at the innermost loop, it is clear that the asymptotic running time is the same as in the previous program, but the $O(N)$ part gets smaller; as a consequence, with $m = 32$, $M = 12$, and $N = 10000$, it runs about 33 percent faster.

55. Replace lines 09–10 of Program Q by

```
Q2   LDO   kl,l,8       A         Q2. Begin new stage.
     SUB   r,j,8        A
     LDO   kr,l,r       A
     SR    m,r,1        A
     LDO   k,l,m        A
     CMP   t,kl,k       A
     CSP   kt,t,k       A         Swap K_m and K_l if K_l > K_m.
     CSP   k,t,kl       A
     CSP   kl,t,kt      A
     CMP   t,k,kr       A
     BNP   t,0F         A[A/3]    Done if K ≤ K_r.
     STO   k,l,r        2A/3
     SET   k,kr         2A/3      K ← K_r.
     CMP   t,kl,k       2A/3
     CSP   k,t,kl       2A/3      Swap K_r and K_l if K_l > K_r.
     CSP   kl,t,kr      2A/3
0H   STO   kl,l,8       A
     LDO   kt,l,16      A
     STO   kt,l,m       A
     STO   k,l,16       A
     SET   i,24         A
```

Also, change the instruction in line 25 to `STO kj,l,16` (see the remark after (27)).

On average, this modification adds $A(20\upsilon + 7\frac{2}{3}\mu)$ to the total running time of Program Q.

56.

. .

Similarly $S_N = \frac{3}{7}(N+1)(5M+3)/(2M+3)(2M+1) - 1 + O(N^{-6})$. The total average running time of the program in exercise 55 is $(42.5A_N + 6B_N + 4C_N + 6D_N + 5E_N + 9S_N + 6N + 7.5)\upsilon + (10\frac{2}{3}A_N + 2B_N + C_N + D_N + 2E_N + 2N - 2)\mu$. The choice $M = 11$ is slightly better than $M = 12$, producing an average time of approximately $(8.91(1+N)\ln N - 3.66N - 39.66)\upsilon + (2.4(1+N)\ln N - 0.22N - 10.88)\mu$.

5.2.3. Sorting by Selection

8. We can start the next iteration of step S2 at position i, provided that we have remembered $\max(K_1, \ldots, K_{i-1})$. One way to keep all of this auxiliary information is to use a link table $L_1 \ldots L_N$ such that K_{L_k} is the previous boldface element of Table 1 whenever K_k is boldface; $L_1 = -1$. [We could get by with less auxiliary storage, at the expense of some redundant comparisons.]

The following `MMIX` program has an additional parameter `link` ≡ `LOC(`L_1`)`. The indices i, j, and k are scaled by 8, to be used as offsets. To make the inner loop fast, the offset `k` ≡ $8(k-j)$ is relative to K_j (and L_j), keeping it in the range $-8j$ <= `k` <= 0.

`:Sort`	`SL`	`j,n,3`	1	*S1. Loop on j.* $j \leftarrow N$.
	`SUB`	`j,j,8`	1	$j \leftarrow j-1$.
	`BNP`	`j,9F`	$1_{[0]}$	$j > 0$?
	`NEG`	`t,1`	1	
	`STO`	`t,link,0`	1	$L_1 \leftarrow -1$.
	`JMP`	`1F`	1	
`2H`	`ADDU`	`linkj,link,j`	$N-D$	`linkj` ← `LOC(`L_{j+1}`)`.
	`ADDU`	`keyj,key,j`	$N-D$	`keyj` ← `LOC(`K_{j+1}`)`.
`S2`	`LDO`	`kk,keyj,k`	A	*S2. Find* $\max(K_1, \ldots, K_j)$. `kk` ← K_k.
	`CMP`	`t,max,kk`	A	Compare $K_i : K_k$.
	`PBNN`	`t,0F`	$A_{[N-C]}$	If $K_i < K_k$,
	`STO`	`i,linkj,k`	$N-C$	$L_k \leftarrow i$,
	`ADD`	`i,j,k`	$N-C$	$i \leftarrow k$, and
	`SET`	`max,kk`	$N-C$	`max` ← K_k.
`0H`	`ADD`	`k,k,8`	A	$k \leftarrow k+1$.
	`PBNP`	`k,S2`	$A_{[N-D]}$	Jump if $k \le j$.
`S3`	`LDO`	`t,key,j`	$N-1$	*S3. Exchange with* K_j.
	`STO`	`max,key,j`	$N-1$	
	`STO`	`t,key,i`	$N-1$	
	`SUB`	`j,j,8`	$N-1$	$j \leftarrow j-1$.
	`SUB`	`k,i,j`	$N-1$	$k \leftarrow i$.
	`LDO`	`i,link,i`		

	LDO	i,link,i	$N-1$	$i \leftarrow L_i$.
	PBNN	i,0F	$N-1_{[C-1]}$	If there is no link,
1H	NEG	k,8,j	C	$k \leftarrow 1$ and
	SET	i,0	C	$i \leftarrow 0$.
0H	LDO	max,key,i	N	max $\leftarrow K_i$.
	PBNP	k,2B	$N_{[D]}$	
	PBP	j,S3	$D_{[1]}$	
9H	POP	0,0		▮

9. $N-1+\Sigma_{N\geq k\geq 2}((k-1)/2-1/k)=\frac{1}{2}\binom{N}{2}+N+H_N$. [The average values of C and D are, respectively, H_N+1 and $H_N-\frac{1}{2}$; hence the average running time of the program is $(1.25N^2+21.75N+3H_N-1.5)\upsilon+(0.25N^2+6.75N-4)\mu$.] Program H is much better for large N.

5.2.4. Sorting by Merging

[647]

9. The following subroutine implements Algorithm S. It expects three parameters: `key` ≡ `LOC(`K_1`)` = `LOC(`R_1`)`, the location of the records to be sorted; `key2`, the location of a second area where the records can be stored (which can be `LOC(`R_{N+1}`)`; and `n` ≡ N, the number of records. Switching the output areas is achieved by interchanging `key` and `key2`; a variable s is not needed. The return value is the location of the sorted records, which will be either `key` or `key2`.

The implementation presented here maintains two pointers `q` ≡ `LOC(`K_q`)` and `r` ≡ `LOC(`K_r`)` instead of the counters q and r. The offsets `i` and `j` are relative to `q` and `r`. Hence, we can access the keys K_i and K_j at locations `q+i` and `r+j`, respectively. In the inner loop, decrementing q or p is eliminated and the tests $q > 0$ and $r > 0$ are replaced by `i` < 0 and `j` > 0. This reduces the asymptotic running time to $8N\lg N$ units.

:Sort	SL	n,n,3	1	<u>S1. Initialize.</u>
	SET	p,8	1	$p \leftarrow 1$.
S2	ADDU	q,key,p	A	<u>S2. Prepare for pass.</u> q $\leftarrow$ LOC(R_{1+p}).
	NEG	i,p	A	$i \leftarrow 1$ (i is relative to q).
	LDO	ki,q,i	A	ki $\leftarrow K_i$.

	ADDU	r,key,n	A	r ← LOC(R_{N+1}).
	SUB	r,r,8	A	r ← LOC(R_N).
	SUB	r,r,p	A	r ← LOC(R_{N-p}).
	SET	j,p	A	$j \leftarrow N$ (j is relative to r).
	LDO	kj,r,j	A	kj ← K_j.
	NEG	k,8	A	$k \leftarrow -1$.
	SET	l,n	A	$l \leftarrow N$.
	SET	d,8	A	$d \leftarrow 1$.
S3	CMP	t,ki,kj	C	S3. Compare $K_i : K_j$.
	BP	t,S8	$C_{[C'']}$	If $K_i > K_j$, go to S8.
	ADD	k,k,d	C'	S4. Transmit R_i. $k \leftarrow k + d$.
	STO	ki,key2,k	C'	$R_k \leftarrow R_i$.
	ADD	i,i,8	C'	S5. End of run? $i \leftarrow i + 1$.
	LDO	ki,q,i	C'	ki ← K_i.
	PBN	i,S3	$C'_{[B']}$	If $q > 0$, go to S3.
S6	ADD	k,k,d	D'	S6. Transmit R_j. $k \leftarrow k + d$.
	CMP	t,k,l	D'	
	BZ	t,S13	$D'_{[A']}$	If $k = l$, go to S13.
	STO	kj,key2,k	$D' - A'$	$R_k \leftarrow R_j$.
	SUB	j,j,8	$D' - A'$	S7. End of run? $j \leftarrow j - 1$.
	LDO	kj,r,j	$D' - A'$	kj ← K_j.
	PBNP	j,S12	$D' - A'_{[D'-B']}$	If $r \leq 0$, go to S12;
	JMP	S6	$D' - B'$	otherwise, go to S6.
S8	ADD	k,k,d	C''	S8. Transmit R_j. $k \leftarrow k + d$.
	STO	kj,key2,k	C''	$K_k \leftarrow K_j$.
	SUB	j,j,8	C''	S9. End of run? $j \leftarrow j - 1$.
	LDO	kj,r,j	C''	kj ← K_j.
	PBP	j,S3	$C''_{[B'']}$	If $r > 0$, go to S3.
S10	ADD	k,k,d	D''	S10. Transmit R_i. $k \leftarrow k + d$.
	CMP	t,k,l	D''	
	BZ	t,S13	$D''_{[A'']}$	If $k = l$, go to S13.
	STO	ki,key2,k	$D'' - A''$	$R_k \leftarrow R_i$.
	ADD	i,i,8	$D'' - A''$	S11. End of run? $i \leftarrow i + 1$.

	`LDO`	`ki,q,i`	$D'' - A''$	`ki` $\leftarrow K_i$.
	`BN`	`i,S10`	$D'' - A''_{[D''-B'']}$	If q > 0, go to S10.
`S12`	`SUB`	`ji,r,q`	$B - A$	<u>*S12. Switch sides.*</u> ij $\leftarrow j - i$.
	`ADDU`	`q,q,p`	$B - A$	$q \leftarrow p$.
	`NEG`	`i,p`	$B - A$	`i` is relative to `q`.
	`SUB`	`r,r,p`	$B - A$	$r \leftarrow p$.
	`SET`	`j,p`	$B - A$	`j` is relative to `r`.
	`NEG`	`d,d`	$B - A$	$d \leftarrow -d$.
	`SET`	`t,l`	$B - A$	Interchange $k \leftrightarrow l$.
	`SET`	`l,k`	$B - A$	
	`SET`	`k,t`	$B - A$	
	`CMP`	`t,ji,p`	$B - A$	
	`PBNN`	`t,S3`	$B - A_{[E]}$	If $j - i \geq p$, go to S3;
	`JMP`	`S10`	E	otherwise, go to S10.
`S13`	`ADD`	`p,p,p`	A	<u>*S13. Switch areas.*</u> $p \leftarrow p + p$.
	`CMP`	`t,p,n`	A	
	`BNN`	`t,0F`	$A_{[1]}$	If $p \geq N$, sorting is complete.
	`SET`	`t,key2`	$A - 1$	Interchange `key2` $\leftrightarrow$ `key`.
	`SET`	`key2,key`	$A - 1$	
	`SET`	`key,t`	$A - 1$	
	`JMP`	`S2`	$A - 1$	Go to S2.
`0H`	`SET`	`$0,key2`	1	Return `key2`.
	`POP`	`1,0`		▮

The running time for $N \geq 3$ is $(5A + 11B - B' + 9C - 2C' + 9D + D' + 3E + 1)\upsilon + (2C + 2D)\mu$, where $A = A' + A''$ is the number of passes, where A' is the number of passes that end with step S6; $B = B' + B''$ is the number of subfile-merge operations performed, where B' is the number of such merges in which the q subfile was exhausted first; $C = C' + C''$ is the number of comparisons performed, where C' is the number of such comparisons with $K_i \leq K_j$; $D = D' + D''$ is the number of elements remaining in subfiles when the other subfile has been exhausted, where D' is the number of such elements belonging to the r subfile; and D'' includes E, the number of subfiles that need no merging because the number of subfiles was odd. Using $A \approx \lceil \lg N \rceil$, $A' \approx A/2$, $B = N - 1$, $B' \approx B/2$, $C + D \approx N \lg N$, $C' \approx C/2$, $D \approx 1.26N + O(1)$ (see exercise 13), and $E \approx A/2$, the asymptotic running time is $8N \lg N + 12.4N + 6.5 \lg N + O(1)$.

The innermost loop of the program contains two branch instructions: one in line 15 and the other in line 20 or 33. On a highly pipelined processor, the first of these branches will cause a considerable slowdown, because no branch prediction logic will be able to achieve more than 50 percent of good guesses on average. Using bitwise tricks and techniques, this branch can be eliminated (see Section 7.1.3, page *181*).

13. The running time for $N \geq 3$ is $(16A + 10B + 1B' + 9C - 2C' + 5D + 4N + 21)\upsilon + (6A + 4B + 3C + D + N + 6)\mu$, where A is the number of passes; $B = B' + B''$ is the number of subfile-merge operations performed, where B' is the number of such merges in which the p subfile was exhausted first; $C = C' + C''$ is the number of comparisons performed, where C' is the number of such comparisons with $K_p \leq K_q$; $D = D' + D''$ is the number of elements remaining in subfiles when the other subfile has been exhausted, where D' is the number of such elements belonging to the q subfile. In Table 3 we have $A = 4$, $B' = 6$, $B'' = 9$, $C' = 22$, $C'' = 22$, $D' = 10$, $D'' = 10$, total time $= 757\upsilon + 258\mu$. (The comparable Program 5.2.1L takes only $356\upsilon + 160\mu$, when improved as in exercise 5.2.1–33, so we can see that merging isn't especially efficient when N is small.) . . .

15. Add an extra copy of L3 and L4, replacing line 26 of Program L with

	`BOD`	`p,L5`	$C'_{[B'_1]}$	If $\mathtt{TAG}(p) = 0$, continue with L3A.
`L3A`	`CMP`	`c,kp,kq`	C'_1	<u>L3. Compare $K_p : K_q$.</u>
	`BP`	`c,L6`	$C'_{1[C''_1]}$	If $K_p > K_q$, go to L6.
	`SET`	`s,p`	C'_1	<u>L4. Advance p.</u> $s \leftarrow p$.
	`LDTU`	`p,link,p`	C'_1	$p \leftarrow L_p$.
	`LDT`	`kp,key,p`	C'_1	$\mathtt{kp} \leftarrow K_p$.
	`PBEV`	`p,L3A`	$C'_{1[B'-B'_1]}$	If $\mathtt{TAG}(p) = 0$, return to L3A.

The replacement for line 38 is similar. The elimination of $L_s \leftarrow p$ (and $L_s \leftarrow q$) reduces the asymptotic running time by $0.5C$ to $7.5N \lg N$. A *further* improvement can also be made, removing the assignments $s \leftarrow p$ (and $s \leftarrow q$) from the inner loop by renaming the registers! With twelve copies of the inner loop, corresponding to the different permutations of (p, q, s) and the different knowledge about L_s, we can cut the average running time to $(6.5N \lg N + O(N))\upsilon$.

This is the code for steps L3, L4, and L5 (the code for steps L3, L6, and L7 is similar):

L3pqs	CMP	c,kp,kq	*L3. Compare $K_p : K_q$.*
	BP	c,L6pqs	If $K_p > K_q$, go to L6pqs.
L4pqs	STTU	p,link,s	*L4. Advance p.* $L_s \leftarrow p$.
	LDTU	s,link,p	$p \leftarrow L_p$.
	LDT	kp,key,s	kp $\leftarrow K_p$.
	BOD	s,L5sqp	If TAG$(p) = 1$, continue with L5sqp.
L34sqp	CMP	c,kp,kq	*L3. Compare $K_p : K_q$.*
	BP	c,L6sqp	If $K_p > K_q$, go to L6sqp.
	LDTU	p,link,s	*L4. Advance p.* $p \leftarrow L_s$.
	LDT	kp,key,p	kp $\leftarrow K_p$.
	BOD	p,L5pqs	If TAG$(p) = 1$, continue with L5pqs.
L34pqs	CMP	c,kp,kq	*L3. Compare $K_p : K_q$.*
	BP	c,L6pqs	If $K_p > K_q$, go to L6pqs.
	LDTU	s,link,p	*L4. Advance p.* $s \leftarrow L_p$.
	LDT	kp,key,s	kp $\leftarrow K_s$.
	PBEV	s,L34sqp	If TAG$(p) = 0$, continue with L34sqp.
L5sqp	STTU	q,link,p	*L5. Complete the sublist.* $L_p \leftarrow q$.
	SET	p,s	Undo permutation of (p, q, s).
	JMP	L5A	
L4psq	STTU	p,link,q	*L4. Advance p.* $L_q \leftarrow p$.
	LDTU	q,link,p	$q \leftarrow L_p$.
	LDT	kp,key,q	kp $\leftarrow K_q$.
	BOD	q,L5qsp	If TAG$(q) = 1$, continue with L5qsp.
L34qsp	CMP	c,kp,kq	*L3. Compare $K_p : K_q$.*
	BP	c,L6qsp	If $K_p > K_q$, go to L6qsp.
	LDTU	p,link,q	*L4. Advance p.* $p \leftarrow L_q$.
	LDT	kp,key,p	kp $\leftarrow K_p$.
	BOD	p,L5psq	If TAG$(p) = 1$, continue with L5psq.
L34psq	CMP	c,kp,kq	*L3. Compare $K_p : K_q$.*
	BP	c,L6psq	If $K_p > K_q$, go to L6psq.
	LDTU	q,link,p	*L4. Advance p.* $q \leftarrow L_p$.
	LDT	kp,key,q	kp $\leftarrow K_q$.
	PBEV	q,L34qsp	If TAG$(q) = 0$, continue with L34qsp.

`L5qsp`	`STTU`	`s,link,p`	*L5. Complete the sublist.* $L_p \leftarrow s$.
	`SET`	`p,q`	Undo permutation of (p, q, s).
	`SET`	`q,s`	
	`JMP`	`L5A`	
`L4spq`	`STTU`	`s,link,q`	*L4. Advance p.* $L_s \leftarrow p$.
	`LDTU`	`q,link,s`	$q \leftarrow L_s$.
	`LDT`	`kp,key,q`	$\mathsf{kp} \leftarrow K_q$.
	`BOD`	`q,L5qps`	If $\mathsf{TAG}(q) = 1$, continue with L5qps.
`L34qps`	`CMP`	`c,kp,kq`	*L3. Compare $K_p : K_q$.*
	`BP`	`c,L6qps`	If $K_p > K_q$, go to L6qps.
	`LDTU`	`s,link,q`	*L4. Advance p.* $s \leftarrow L_q$.
	`LDT`	`kp,key,s`	$\mathsf{kp} \leftarrow K_s$.
	`BOD`	`s,L5spq`	If $\mathsf{TAG}(s) = 1$, continue with L5spq.
`L34spq`	`CMP`	`c,kp,kq`	*L3. Compare $K_p : K_q$.*
	`BP`	`c,L6spq`	If $K_p > K_q$, go to L6spq.
	`LDTU`	`q,link,s`	*L4. Advance p.* $q \leftarrow L_s$.
	`LDT`	`kp,key,q`	$\mathsf{kp} \leftarrow K_q$.
	`PBEV`	`q,L34qps`	If $\mathsf{TAG}(q) = 0$, continue with L34qps.
`L5qps`	`STTU`	`p,link,s`	*L5. Complete the sublist.* $L_s \leftarrow p$.
	`SET`	`s,p`	Undo permutation of (p, q, s).
	`SET`	`p,q`	
	`SET`	`q,s`	
	`JMP`	`L5A`	
`L4qps`	`STTU`	`q,link,s`	*L4. Advance p.* $L_s \leftarrow q$.
	`LDTU`	`s,link,q`	$s \leftarrow L_q$.
	`LDT`	`kp,key,s`	$\mathsf{kp} \leftarrow K_s$.
	`PBEV`	`s,L34spq`	If $\mathsf{TAG}(s) = 0$, continue with L34spq.
`L5spq`	`STTU`	`p,link,q`	*L5. Complete the sublist.* $L_q \leftarrow p$.
	`SET`	`q,p`	Undo permutation of (p, q, s).
	`SET`	`p,s`	
	`JMP`	`L5A`	
`L4sqp`	`STTU`	`s,link,p`	*L4. Advance p.* $L_p \leftarrow s$.
	`LDTU`	`p,link,s`	$p \leftarrow L_s$.
	`LDT`	`kp,key,p`	$\mathsf{kp} \leftarrow K_p$.
	`PBEV`	`p,L34pqs`	

```
                         If TAG(s) = 0, continue with L34pqs.
L5pqs   STTU  q,link,s   L5. Complete the sublist. L_s ← q.
L5A     SET   s,t        s ← t.
0H      SET   t,q        t ← q.
        LDTU  q,link,q   q ← L_q.
        BEV   q,0B       Repeat until If TAG(q) = 1.
        LDT   kq,key,q   kq ← K_q.
        JMP   L8
L4qsp   STTU  q,link,p   L4. Advance p. L_p ← q.
        LDTU  p,link,q   p ← L_q.
        LDT   kp,key,p   kp ← K_p.
        PBEV  p,L34psq   If TAG(p) = 0, continue with L34psq.
L5psq   STTU  s,link,q   L5. Complete the sublist. L_q ← s.
        SET   q,s        Undo permutation of (p, q, s).
        JMP   L5A
```

5.2.5. Sorting by Distribution

[650]

5. In Program R, replace lines 07–10 by

```
     NEG   k,3                 1     k ← 1.
     SET   mask,8*((1<<m)-1)   1     mask ← 8(2^m − 1) (the bit mask).
0H   SUBU  P,P,16              N     R5. Step to next record.
     LDOU  i,P,KEY             N     R3. Extract first digit of key.
     SLU   i,i,3               N
     AND   i,i,mask            N     i ← a_1.
```

to initialize the registers `k` (the bitoffset) and `mask` (the bitmask). Here, we assume $m \ge 3$ so that in later passes the bitoffset can be adjusted by adding m. Then replace lines 19 and 21 by

```
     ADD   k,k,m       P − 1        k ← k + 1.
R3   LDOU  i,P,KEY     N(P − 1)     R3. Extract kth digit of key.
     SRU   i,i,k       N(P − 1)
     AND   i,i,mask    N(P − 1)     i ← a_{p+1−k}.
```

The changes to the sort routine add $(NP + 1)\upsilon$ to the running time; it amounts to

$((8P+1)N+11MP+26P+9)\upsilon$. For fixed N and fixed key length Pm, the extra time spent in the sort routine will grow linearly with increasing P and the amount of time spent in the `Hook` and `Empty` subroutines will grow exponentially larger as P gets smaller. So for each N and key length, there will be an optimal number of passes. For $N < 10000$ and keys up to 32 bits long, the changes will always make the program slower. For $N = 100000$ and a full 64-bit key, the improved program with $m = 13$ and $P = 5$ will be about 20 percent faster.

5.3.1. Minimum-Comparison Sorting

28. The simplest and most efficient solution starts by loading all five keys in registers; then implements the decision tree as described in the text, using a `CMP` instruction followed by a `BP` for each node; and finishes off by storing the five keys.

```
:Sort   LDB   a,K,0      1
        LDB   b,K,1      1
        LDB   c,K,2      1
        LDB   d,K,3      1
        LDB   e,K,4      1
        CMP   t,a,b      1
        BP    t,0F       1[0.5]    a < b
        CMP   t,c,d      1
        BP    t,1F       1[0.5]    a < b, c < d
        CMP   t,b,d      1
        BP    t,2F       1[0.5]    a < b < d, c < d
        …
2H      …                          a < b, c < d < b
        …
1H      CMP   t,b,c      1         a < b, d < c
        BP    t,2F       1[0.5]    a < b < c, d < c
        …
2H      …                          a < b, d < c < b
        …
0H      CMP   t,c,d      1         b < a
        BP    t,1F       1[0.5]    b < a, c < d
        CMP   t,a,d      1
        BP    t,2F       1[0.5]    b < a < d, c < d
        …
```

```
2H    …                     b < a, c < d < a
      …
1H    CMP   t,a,c   1       b < a, d < c
      BP    t,2F    1[0.5]  b < a < c, d < c
      …
2H    …                     b < a, d < c < a
      …
```

⟨Using `3H` and `4H` to insert `e`, using `5H` and `6H` to insert the last element `c`, and finishing with 120 variations of `7H`.⟩

```
7H    STB   a,K,0   1
      STB   b,K,1   1
      STB   c,K,2   1
      STB   d,K,3   1
      STB   e,K,4   1
      POP   0,0           ▮
```

The full 1075-line program has an average running time of 30.8υ + 10μ. Its minimum running time is 22υ + 10μ (6 correctly predicted branches); its maximum running time is 38υ + 10μ. The latter appears to be optimal since it is the time for 5 `LDB`, 7 `CMP`, 7 `BP` (all mispredicted), and 5 `STB`. One should not write such a program. If desired, one should implement a generator to produce merge insertion programs for arbitrary (small) N.

Much shorter programs are possible at minimal extra cost. For example, the first test and branch

```
CMP   t,a,b   1
BP    t,0F    1[0.5]
```

can be replaced by a test and a swap of a with b:

```
CMP   t,a,b     1
CSP   x,t,a     1     a ↔ b
CSP   a,t,b     1
CSP   b,t,x     1
```

This cuts the size of the program in half without changing the maximum running time. The average running time will increase by 1 cycle, and the minimum running time by 2 cycles.

A similar replacement can be done for the next test $c < d$. Joining the control flow after the third test $b < d$ requires two swaps: $a \leftrightarrow c$ and $b \leftrightarrow d$. Using

Conditional-Set instructions here is less efficient than a branch. The transformation in lines 14–21 adds 4 cycles to the maximum running time and 2 cycles to the average and minimum running times.

Next, e must be inserted in the sequence $a < b < d$. Swapping values as needed, we can reduce the possibilities to two cases: $a < b < e < d$, $c < d$ and $a < b < d < e$, $c < d$. The endgame inserts c below d. The STB instructions can be issued as soon as the final position is known, further reducing the size of the code without affecting the running time. We obtain:

```
01  :Sort   LDB   a,K,0     1
02          LDB   b,K,1     1
03          LDB   c,K,2     1
04          LDB   d,K,3     1
05          LDB   e,K,4     1
06          CMP   t,a,b     1
07          CSP   x,t,a     1        a ↔ b.
08          CSP   a,t,b     1
09          CSP   b,t,x     1
10          CMP   t,c,d     1        Here a < b.
11          CSP   x,t,c     1        c ↔ d.
12          CSP   c,t,d     1
13          CSP   d,t,x     1
14          CMP   t,b,d     1        Here c < d.
15          BN    t,2F      1[1/2]
16          SET   x,a       1/2      a ↔ c.
17          SET   a,c       1/2
18          SET   c,x       1/2
19          SET   x,b       1/2      b ↔ d.
20          SET   b,d       1/2
21          SET   d,x       1/2
22  2H      CMP   t,e,b     1        Here a < b < d and c < d.
23          BP    t,3F      1[7/15]
24          CMP   t,e,a     8/15     Here a < b < d, e < b, and c < d.
25          SET   x,e       8/15     x ← e.
26          SET   e,b       8/15     e ← b.
27          CSNP  b,t,a     8/15     If e < a, b ← a.
28          CSNP  a,t,x     8/15     If e < a, a ← x.
```

```
         CSP   b,t,x     8/15         If e > a, b ↔ e.
0H       STB   d,K,4     4/5          Here a < b < e < d and c < d.
         CMP   t,c,b     4/5
         BP    t,5F      4/5[2/5]
         STB   e,K,3     2/5          Here a < b < e < d and c < b.
1H       STB   b,K,2     8/15
         CMP   t,c,a     8/15
         BP    t,6F      8/15[4/15]
         STB   c,K,0     4/15         Here c < a < b < e < d.
         STB   a,K,1     4/15
         POP   0,0
6H       STB   a,K,0     4/15         Here a < c < b < e < d.
         STB   c,K,1     4/15
         POP   0,0
5H       STB   a,K,0     2/5          Here a < b < e < d and b < c < d.
         STB   b,K,1     2/5
         CMP   t,c,e     2/5
         BN    t,6F      2/5[1/5]
         STB   e,K,2     1/5          Here a < b < e < c < d.
         STB   c,K,3     1/5
         POP   0,0
6H       STB   e,K,3     1/5          Here a < b < c < e < d.
         STB   c,K,2     1/5
         POP   0,0
3H       CMP   t,e,d     7/15         Here a < b < d, b < e, and c < d.
         PBN   t,0B      7/15[1/5]
         STB   e,K,4     1/5          Here a < b < d < e and c < d.
         STB   d,K,3     1/5
         CMP   t,c,b     1/5
         PBN   t,1B      1/5[1/15]
         STB   a,K,0     1/15         Here a < b < c < d < e.
         STB   b,K,1     1/15
         STB   c,K,2     1/15
         POP   0,0                    ▮
```

The above code has only 62 instructions. Its maximum running time is 42υ +

10μ, the minimum is $32\upsilon + 10\mu$ and the average is $37.2\upsilon + 10\mu$.

On computers, such as MMIX, that have an ODIF instruction for saturating subtraction, implementing a sorting network (see Section 5.3.4) is an attractive alternative. When three instructions (see exercise 5–8) suffice to order two nonnegative numbers

a and b, a sorting network for five numbers that will need nine such comparators (see 5.3.4–(11)) can be implemented with 27 instructions. Add to this five load and store instructions, and you obtain a sorting procedure only 37 instructions long that takes exactly $37\upsilon + 10\mu$ to execute.

This will not beat the $30.8\upsilon + 10\mu$ average running time of the full program with 1075 instructions, but it is much shorter than even the reduced program with 62 instructions, beating its average running time of $37.2\upsilon + 10\mu$ with a constant running time of $37\upsilon + 10\mu$.

For n keys, the minimum possible average number of comparisons is approximately lg n, while the size of the smallest sorting network for n keys is $O(n(\log n)^2)$. Obviously for large n, neither of the two methods can be recommended.

5.5. SUMMARY, HISTORY, AND BIBLIOGRAPHY

[701]

2. For small and medium N, say $N \leq 1000$, multiple list insertion; for large N, radix list sort.

3. The following subroutine expects two parameters: `p`, the location of the first node, and `k` ≡ K, the key. After a successful search, it returns the location of the record found; otherwise, it returns zero.

`S3`	`LDOU`	`p,p,LINK`	$C - S$	S3. Advance. $P \leftarrow$ `LINK`(P).
`:Search`	`BZ`	`p,0F`	$C - S + 1_{[1-S]}$	S4. End of file?
	`LDO`	`kp,p,KEY`	C	`kp` $\leftarrow$ `KEY`(P).
	`CMP`	`t,k,kp`	C	S2. Compare.
	`PBNZ`	`t,S3`	$C_{[S]}$	If $K =$ `KEY`(P), terminate successfully.
`0H`	`POP`	`1,0`		Return `p`. ▮

The running time is $(5C - 2S + 3)\upsilon + (2C - S)\mu$.

5. Program Q′ takes more time than Program Q if $C < S + 2 + (C - S) \bmod 2$. A successful search ($S = 1$) will take more time only for $i \le 2$; an unsuccessful search will take more time only for $N = 1$.

6. We unroll the inner loop three times.

`:Search`	`SL`	`i,n,3`	1	Q1. Initialize.
	`NEG`	`i,i`	1	`i` $\leftarrow -8N$, $i \leftarrow 1$.
	`SUBU`	`key,key,i`	1	`key` $+$ `i` $\leftarrow$ `LOC`(K_{N+1}).
	`ADDU`	`key1,key,8`	1	`key1` $+$ `i` $\leftarrow$ `LOC`(K_{N+2}).
	`ADDU`	`key2,key1,8`	1	`key2` $+$ `i` $\leftarrow$ `LOC`(K_{N+3}).
	`STO`	`k,key,0`	1	$K_N \leftarrow K$.
	`JMP`	`Q2`	1	
`Q3`	`ADD`	`i,i,24`	$\lfloor (C-S)/3 \rfloor$	Q3. Advance. (3 times)
`Q2`	`LDO`	`ki,key,i`	$\lfloor (C-S)/3 \rfloor + 1$	Q2. Compare.
	`CMP`	`t,k,ki`	$\lfloor (C-S)/3 \rfloor + 1$	
	`BZ`	`t,Q4`	$\lfloor (C-S)/3 \rfloor + 1_{[1-F]}$	To Q4 if $K = K_i$.
	`LDO`	`ki,key1,i`	$\lfloor (C-S)/3 \rfloor + F$	Q2. Compare.
	`CMP`	`t,k,ki`	$\lfloor (C-S)/3 \rfloor + F$	
	`BZ`	`t,0F`	$\lfloor (C-S)/3 \rfloor + F_{[F-G]}$	To Q4 if $K = K_{i+1}$.

	LDO	ki,key2,i	$\lfloor (C-S)/3 \rfloor + G$	*Q2. Compare.*
	CMP	t,k,ki	$\lfloor (C-S)/3 \rfloor + G$	
	PBNZ	t,Q3	$\lfloor (C-S)/3 \rfloor + G_{[G]}$	To Q3 if $K \neq K_{i+2}$.
	ADD	i,i,8	G	
0H	ADD	i,i,8	F	
Q4	PBN	i,Success	$1_{[1-S]}$	*Q4. End of file?*
	POP	0,0		Exit if not in table.
Success	ADDU	$0,key,i	S	Return LOC(K_i).
	POP	1,0		▮

The total running time is $(10\lfloor (C-S)/3 \rfloor - S + 4F + 4G + 15)\upsilon + (3\lfloor (C-S)/3 \rfloor + F + G + 2)\mu$. Using $(C-S) \bmod 3 = F + G$, this is about $(3.33C - 4.33S + 0.67((C-S) \bmod 3) + 15)\upsilon + (C - S + 2)\mu$.

6.2.1. Searching an Ordered Table

[705]

4. It must be an unsuccessful search with $N = 127$; hence by Theorem B the answer is 84υ.

5. Program 6.1Q′ has an average running time of $1.75N + 11.5 - (N \bmod 2)/4N$; this beats Program B if and only if $N \leq 17$. [It beats Program C only for $N = 2$, 4, 5, and 6.]

10. Use a "macro-expanded" program with the DELTA's included; thus, for $N = 10$:

:Search	ADDU	i,key,8*5-8	$i \leftarrow$ DELTA[1], DELTA[1] $= 5$.
	LDO	ki,i,0	ki $\leftarrow K_5$.
	CMP	t,k,ki	Compare $K : K_5$.
	BZ	t,Success	
	ADDU	i,i,8*3	$i \leftarrow i +$ DELTA[2], DELTA[2] $= 3$.
	SUBU	l,i,2*8*3	$l \leftarrow i -$ 2DELTA[2].
	CSN	i,t,l	If $K < K_5$, then $i \leftarrow l$.
	LDO	ki,i,0	ki $\leftarrow K_{2,8}$.
	CMP	t,k,ki	Compare $K : K_{2,8}$.
	BZ	t,Success	
	ADDU	i,i,8*1	$i \leftarrow i +$ DELTA[3], DELTA[3] = 1.

	SUBU	l,i,2*8*1	$l \leftarrow i -$ 2DELTA[3].
	CSN	i,t,l	If $K < K_{2,8}$, then $i \leftarrow l$.
	LDO	ki,i,0	ki $\leftarrow K_{1,3,7,9}$.
	CMP	t,k,ki	Compare $K : K_{1,3,7,9}$.
	BZ	t,Success	
	ADDU	i,i,1*8	$i \leftarrow i +$ DELTA[4], DELTA[4] = 1.
	SUBU	l,i,2*8*1	$l \leftarrow i -$ 2DELTA[4].
	CSN	i,t,l	If $K < K_{1,3,7,9}$, then $i \leftarrow l$.
	LDO	ki,i,0	ki $\leftarrow K_{0,2,2,4,6,8,8,10}$.
	CMP	t,k,ki	Compare $K : K_{0,2,2,4,6,8,8,10}$.
	BZ	t,Success	
Failure	POP	0,0	
Success	POP	1,0	▮

[Exercise 23 shows that most of the "BZ t,Success" instructions may be eliminated, yielding a program about 5 lg N lines long that takes only about 5 lg N units of time; but that program will be faster only for $N > 16300$ (approximately).]

6.2.2. Binary Tree Searching

[708]

1. Use an extra octabyte in memory to contain the location of the root node. Call the subroutine with the location of this octabyte in parameter p and replace the first two lines of Program T with the following:

:Search	SET	l,0	1	*T1. Initialize.* l $\leftarrow$ 0.
	JMP	T3	1	
0H	SET	p,q	C	P $\leftarrow$ Q.
	LDO	kp,p,KEY	C	*T2. Compare.* kp $\leftarrow$ KEY(P).

3. We could replace Λ by a valid address, and set KEY(Λ) $\leftarrow K$ at the beginning of the algorithm; then the test for Q $\neq \Lambda$ could be removed from the inner loop. In addition, the instruction SET p,q can be removed by duplicating the code as in Program 6.2.1F. Thus the MMIX time would be reduced to about $5C$ units.

6.2.3. Balanced Trees

[715]

12. The maximum occurs when inserting into the second external node of (12); $C = 4$, $F = H = 1$, $S = G = J = 0$, for a total time of 97υ. The minimum occurs when inserting into the third-last external node of (13); $C = 2$, $S = J = F = G = H = 0$, for a total time of 49υ. [The corresponding figures for Program 6.2.2T are 57υ and 15υ.]

6.3. DIGITAL SEARCHING

[721]

4. Successful searches take place exactly as with the full table, but unsuccessful searches in the compressed table may go through several additional iterations. For example, an input argument such as `ACCD` will make Program T take *six* iterations: The `A` takes the search to node (2), where the `C` is linked again to node (2)! Consequently, any number of `C`'s in the given key will loop here. In our case, the loop is taken just once more before the `D` takes the search to node (3), from where the end of string will take the search one step further to node (12). There, finally, the search ends unsuccessfully with a zero table entry. It is necessary to verify that no infinite looping on zero sequences is possible. . . .

9. This subroutine has two parameters: `p` ≡ `LOC(ROOT)`, a pointer to the root node, and `k` ≡ K, the given key. If the search is successful, it returns the location of the node found; otherwise, it returns zero. We use `s` ≡ K' as a shift register.

```
:Search SET   s,k          1             D1. Initialize. K' ← K.
        JMP   D2           1
0H      SET   p,q          C − 1         P ← Q.
        SLU   s,s,1        C − 1
D2      LDO   kp,p,KEY     C             D2. Compare. kp ← KEY(P).
        CMP   t,k,kp       C
        BZ    t,Success    C[S]          Exit if K = KEY(P).
        ZSNN  l,s,LLINK    C − S         l ← b ? LLINK : RLINK.
        LDOU  q,p,l        C − S         D3/4. Move left/right. Q ← LINK(b,P).
        PBNZ  q,0B         C − S[1 − S]
        ⟨Continue as in Program 6.2.2T.⟩ ▮
```

The running time for the searching phase of this program is $(8C - 3S + 2)\upsilon + (2C - S)\mu$, where $C - S$ is the number of bit inspections. For random data, the approximate average running times are therefore:

Successful Unsuccessful

	Successful	Unsuccessful
Program 6.2.2T	$14 \ln N - 14.92$	$14 \ln N - 4.91$
This program	$11.5 \ln N - 6.73$	$11.5 \ln N - 0.19$

(Consequently, this program is faster on a successful search if $N \geq 28$ and on an unsuccessful search if $N \geq 7$.)

6.4. HASHING

[728]

1. $-4 \leq$ `a` ≤ 58. Therefore the locations preceding and following the table containing the keys must be guaranteed to contain no data that matches any given argument; alternatively, the instructions '`CMP t,a,40; CSNN a,t,4`' inserted before the first `POP` and '`CSN a,a,4; CMP t,a,40; CSNN a,t,4`' inserted before the last `POP` will keep `a` in the range $0 \leq$ `a` ≤ 39. (The middle `POP` will not need such a test.) The extra tests will add 1.4 cycles to the average running time. [Without these precautions, we might say that the method in exercise 6.3–4 uses less space, since the boundaries of that table are never exceeded.]

2. `BLACK` and `DATA` both hash to 4; `FOR` and `SHE` to 6; `DAY` and `NO` to 11; `LOOK` and `STUDENT` (and `PROGRAM`) to 22; `ALL` and `TRY` to 27; `CAN` and `PEOPLE` to 31; `THEM` and `OVER` to 32; `ONE` and `WILL` to 34; `HIM` and `PART` to 35; and `THEY` and `WHAT` to 37.

3. The ASCII codes satisfy `A` + `T` = `O` + `F` and `B` − `E` = `O` − `R`, so we would have either f(`AT`) = f(`OF`) or f(`BE`) = f(`OR`). Notice that the instruction `2ADDU a,a,a` in Table 1 resolves this dilemma rather well.

5. The hash function is bad since it assumes at most 26 different values, and some of them occur much more often than the others. Even with double hashing (letting $h_2(K) = 1$ plus the second byte of K, say, and $M = 257$) the search will be slowed down more than the time saved by faster hashing. Also $M = 256$ is too small, since FORTRAN programs often have more than 256 distinct variables (especially when produced by a program generator).

6. Not on `MMIX`, since $K > M$ will almost always occur. In this case rR will not contain the remainder $(wK) \bmod M$, but rather the value of register `z` = 0. [It would be nice to be able to compute $(wK) \bmod M$, especially if linear probing were being used with $c = 1$, but unfortunately `MMIX`, like most computers, disallows this since the quotient overflows.]

12. We can store K in an extra entry `KEY[`m`]` at the end of the table, and make the odd link that marks the end of the chain point to this entry. So we replace line 23 by

```
C6        8ADDU   t,m,1          1 − S     C6. Insert new key.
```

and replace lines 09–14 by

```
        SL      t,m,3            A
        STT     k,key,t          A         KEY[M] ← K.
        JMP     3F               A
0H      SET     p,i            C − A       Keep previous value of i.
        LDT     i,link,i       C − A       C4. Advance to next.
3H      LDT     t,key,i          C         t ← KEY[i].
        CMP     t,t,k            C         C3. Compare.
        BNZ     t,0B          C[C−A]       Jump if KEY[i] ≠ K.
        PBEV    i,Success     A[A−S]       Exit unless i is odd.
```

The total running time for the searching phase of the "improved" Program is $(7C - S + 69)\upsilon + (2C + 3)\mu$. The time saved is $(C - 5S)\upsilon - S\mu$, which is actually a net *loss* if $S = 1$ and $C < 5$. (An inner loop shouldn't always be optimized!)

72.

(b) . . .

We assume that at location H, a table of 8 × 256 tetrabytes is initialized with random numbers in the range 0 to $M - 1$, and that the address of H is in the global register h ≡ LOC(H). Then we can replace lines 03 and 04 of Program L by the following

```
SRU      j,k,7*8-3; LDTU i,:h,j
SLU      j,k,8; SRU t,j,7*8-3; INCL t,1*4*258; LDTU t,:h,t; XOR i,i,t
SLU      j,j,8; SRU t,j,7*8-3; INCL t,2*4*258; LDTU t,:h,t; XOR i,i,t
SLU      j,j,8; SRU t,j,7*8-3; INCL t,3*4*258; LDTU t,:h,t; XOR i,i,t
SLU      j,j,8; SRU t,j,7*8-3; INCL t,4*4*258; LDTU t,:h,t; XOR i,i,t
SLU      j,j,8; SRU t,j,7*8-3; INCL t,5*4*258; LDTU t,:h,t; XOR i,i,t
SLU      j,j,8; SRU t,j,7*8-3; INCL t,6*4*258; LDTU t,:h,t; XOR i,i,t
SLU      j,j,8; SRU t,j,7*8-3; INCL t,7*4*258; LDTU t,:h,t; XOR i,i,t
```

The above code is lengthy but needs only $37\upsilon + 8\mu$ instead of the 61υ before. Fig. 42 tells us that the running time of Program L is between 70υ and 80υ as long as the load factor is within a reasonable range. In this case, the new code is about one third faster. Under the same conditions, the speedup for Program D will start again at one third for an empty table and will increase to about one half as more second hashes need to be computed. The modified Program D will benefit from a similar speedup as Program L, but over a slightly extended range. It is possible to initialize the table of tetrabytes at `H` with random numbers from the full range 0 to $2^{32} - 1$ and reduce the range to 0 to $M - 1$ by appending a final `AND` instruction to the code. Then the same tables can be used for all $M = 2^m$ with $1 \leq m \leq 32$.

ACKNOWLEDGMENTS

In December 1998, Vladimir Ivanović started a mailing list to coordinate the people who had either responded to the call for volunteers on Donald Knuth's MMIX page or were referred by Donald Knuth. The MMIXmasters project had started. Later he added a web page and a wiki to aid in communication and present the submitted solutions to the public.

In the course of the following years, multiple contributions were received. They aided in completing the collection of programs presented in this book.

Jan-Hendrik Behrmann contributed an implementation for Program 5.2.3H.

Wijtze de Boer and Kenneth Laskoski both contributed an implementation for Program 5.2C.

Andrey Dubinchak contributed implementations for Programs 2.1-(5), 2.2.3–(10), 2.2.3–(11), 2.2.3T, 2.2.4A, 6.1S, 6.1Q, and 6.1Q′ as well as solutions to exercises 2.1–8, 2.1–9, 2.2.3–24, 2.2.4–11, 2.2.4–13, 2.2.4–14, and 2.2.4–15.

Evgeny Eremin contributed an implementation for Program 5.2.2B.

Armin Grodon contributed an implementation for Program 5.2.4L.

Blake Hegerle contributed an implementation for Programs 5.2.1S, 5.2.1L, and 5.2.1D as well as a solution to exercise 5.2.1–3.

Johannes Maier and Georg Schmidl together contributed implementations of Programs 6.2.1B and 6.2.1F.

Ladislav Sladecek contributed solutions to exercises 2.2.6–15 and 2.5–27.

Michael Unverzart contributed an implementation for Program 5.2.3S and a solution to exercise 5.2.3–8.

Chan Vinh Vong contributed implementations for Programs 2.3.2D, 6.4C, 6.4D, and 6.4L as well as solutions to exercises 2.2.3–2, 2.2.3–3, 2.2.3–8, 2.2.3–24, 2.2.3–27, 2.3.5–4, 2.5–4, and 2.5–34.

Yuval Yarom contributed an implementation for Program 2.3.1T.

An unknown contributor submitted Program 2.3.1S.

We want to thank all of them!

INDEX