an output area, and space is made for subsequent items by moving to the right or to the left, whichever is most convenient. This saves about half the running time of ordinary binary insertion, at the expense of a somewhat more complicated program. It is possible to use this method without using up more space than required for N records (see exercise 6); but we shall not dwell any longer on this "two-way" method of insertion, since considerably more interesting methods have been developed.

Shell's method. If we have a sorting algorithm which moves items only one position at a time, its average running time will be, at best, proportional to N^2, since each record must travel an average of about $\frac{1}{3}N$ positions during the sorting process (see exercise 7). Therefore, if we want to make substantial improvements over straight insertion, we need some mechanism by which the records can take long leaps instead of short steps.

Such a method was proposed in 1959 by Donald L. Shell [*CACM* **2** (July, 1959), 30–32]; we shall call it the *diminishing increment sort*. Table 3 illustrates the general idea behind his method: First we divide the 16 records into 8 groups of two each, namely (R_1, R_9), (R_2, R_{10}), . . . , (R_8, R_{16}). Sorting each group of records separately takes us to the second line of Table 3; this is called the "first pass." Note that 154 has changed places with 512; 908 and 897 both jump to the right. Now we divide the records into 4 groups of four each, namely (R_1, R_5, R_9, R_{13}), . . . , $(R_4, R_8, R_{12}, R_{16})$, and again each group is sorted separately; this "second pass" takes us to line 3. A third pass sorts two groups of eight records, then a fourth pass completes the job by sorting all 16 records. Each of the intermediate sorting processes involves either a comparatively short file or a file which is comparatively well ordered, so straight insertion can

Table 3

DIMINISHING INCREMENT SORT (8, 4, 2, 1)

	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
8-sort:																
	503	087	154	061	612	170	765	275	653	426	512	509	908	677	897	703
4-sort:																
	503	087	154	061	612	170	512	275	653	426	765	509	908	677	897	703
2-sort:																
	154	061	503	087	512	170	612	275	653	426	765	509	897	677	908	703
1-sort:																
	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

be used for each sorting operation; the records tend to converge quickly to their final destination.

The sequence of increments 8, 4, 2, 1 is not sacred; *any* sequence $h_t, h_{t-1}, \ldots, h_1$ can be used, so long as the last increment h_1 equals 1. For example, Table 4 shows the same data sorted with increments 7, 5, 3, 1. Some sequences are much better than others; we will discuss the choice of increments later.

Algorithm D (*Diminishing increment sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \cdots \le K_N$. An auxiliary sequence of increments $h_t, h_{t-1}, \ldots, h_1$ is used to control the sorting process, where $h_1 = 1$; proper choice of these increments can significantly decrease the sorting time. This algorithm reduces to Algorithm S when $t = 1$.

D1. [Loop on s.] Perform step D2 for $s = t, t-1, \ldots, 1$; then terminate the algorithm.

D2. [Loop on j.] Set $h \leftarrow h_s$, and perform steps D3 through D6 for $h < j \le N$. (We will use a straight insertion method to sort elements that are h positions apart, so that $K_i \le K_{i+h}$ for $1 \le i \le N - h$. Steps D3 through D6 are essentially the same as steps S2 through S5, respectively, in Algorithm S.)

D3. [Set up i, K, R.] Set $i \leftarrow j - h$, $K \leftarrow K_j$, $R \leftarrow R_j$.

D4. [Compare K, K_i.] If $K \ge K_i$, go to step D6.

D5. [Move R_i, decrease i.] Set $R_{i+h} \leftarrow R_i$, then $i \leftarrow i - h$. If $i > 0$, go back to step D4.

D6. [R into R_{i+h}.] Set $R_{i+h} \leftarrow R$. ▮

The corresponding MIX program is not much longer than our program for straight insertion. Lines 08–19 of the following code are a direct translation of Program S into the more general framework of Algorithm D.

Program D (*Diminishing increment sort*). We assume that the increments are stored in an auxiliary table, with h_s in location H + s; all increments are less than N. Register assignments: rI1 $\equiv j - N$; rI2 $\equiv i$; rA $\equiv R \equiv K$; rI3 $\equiv s$; rI4 $\equiv h$. Note that this program modifies itself, in order to obtain efficient execution of the inner loop.

```
01   START  ENT3  T          1     D1. Loop on s.  s ← t.
02   1H     LD4   H,3        T     D2. Loop on j.  h ← h_s.
03          ENT1  INPUT,4    T     Modify the addresses of three instructions
04          ST1   6F(0:2)    T        in the main loop.
05          ST1   7F(0:2)    T
06          ENN1  -N,4       T     rI1 ← N − h.
07          ST1   4F(0:2)    T
08          ENT1  1-N,4      T     j ← h + 1.
```

09	2H	LDA	INPUT+N,1	$NT - S$	*D3. Set up i, K, R.*
10	4H	ENT2	N-H,1	$NT - S$	$i \leftarrow j - h$. (Instruction modified)
11	5H	CMPA	INPUT,2	$B + NT - S - A$	*D4. Compare K, K_i.*
12		JGE	7F	$B + NT - S - A$	To D6 if $K \geq K_i$.
13		LDX	INPUT,2	B	*D5. Move R_i, decrease i.*
14	6H	STX	INPUT+H,2	B	$R_{i+h} \leftarrow R_i$. (Instruction modified)
15		DEC2	0,4	B	$i \leftarrow i - h$.
16		J2P	5B	B	To D4 if $i > 0$.
17	7H	STA	INPUT+H,2	$NT - S$	*D6. R into R_{i+h}.* (Instruction modified)
18	8H	INC1	1	$NT - S$	$j \leftarrow j + 1$.
19		J1NP	2B	$NT - S$	To D3 if $j \leq N$.
20		DEC3	1	T	
21		J3P	1B	T	$t \geq s \geq 1$. ∎

***Analysis of Shell's Method.** In order to choose a good sequence of increments $h_t, \ldots, h_1$ for use in Algorithm D, we need to analyze the running time as a function of those increments. This leads to some fascinating mathematical problems, not yet completely resolved; nobody has been able to determine the best possible sequence of increments for large values of N. Yet a good many interesting facts are known about the behavior of Shell's diminishing increment sort, and we will summarize them here; details appear in the exercises below. [Readers who are not mathematically inclined should skim over the next few pages, continuing at the formulas numbered (8) below.]

The frequency counts shown with Program D indicate that five factors determine the execution time: the size of the file, N; the number of passes (i.e., the number of increments), $T = t$; the sum of the increments,

$$S = h_1 + \cdots + h_t;$$

the number of comparisons, $B + NT - S - A$; and the number of moves, B. As in the analysis of Program S, A is the number of left-to-right minima encountered in the intermediate sorting operations, and B is the number of inversions in the subfiles. The factor which governs the running time is B, so we shall devote most of our attention to it. For purposes of analysis we shall assume that the keys are distinct and initially in random order.

Let us call the operation of step D2 "h-sorting," so that Shell's method consists of h_t-sorting, followed by h_{t-1}-sorting, . . . , followed by h_1-sorting. A file in which $K_i \leq K_{i+h}$ for $1 \leq i \leq N - h$ will be called "h-ordered."

Consider first the simplest generalization of straight insertion, when there are just two increments, $h_2 = 2$ and $h_1 = 1$. During the second pass we have a 2-ordered sequence of keys, $K_1 K_2 \ldots K_N$. It is easy to see that the number of permutations $a_1 a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$ such that $a_i \leq a_{i+2}$ for $1 \leq i \leq n - 2$ is

$$\binom{n}{\lfloor n/2 \rfloor},$$

since we obtain exactly one 2-ordered permutation for each choice of $\lfloor n/2 \rfloor$ elements to put in even-numbered positions $a_2 a_4 \ldots$, with the remaining $\lceil n/2 \rceil$ elements going in odd-numbered positions. Each 2-ordered permutation

is equally likely after a random file has been 2-sorted. What is the average number of inversions among all such permutations?

Let A_n be the total number of inversions among all 2-ordered permutations of $\{1, 2, \ldots, n\}$. Clearly $A_1 = 0$, $A_2 = 1$, $A_3 = 2$; and by considering the six cases

$$1\ 3\ 2\ 4 \qquad 1\ 2\ 3\ 4 \qquad 1\ 2\ 4\ 3 \qquad 2\ 1\ 3\ 4 \qquad 2\ 1\ 4\ 3 \qquad 3\ 1\ 4\ 2$$

we find that $A_4 = 1 + 0 + 1 + 1 + 2 + 3 = 8$. In order to investigate A_n in general, consider the "lattice diagram" illustrated in Fig. 11 for $n = 15$. A 2-ordered permutation of $\{1, 2, \ldots, n\}$ can be represented as a path from the upper left corner point $(0, 0)$ to the lower right corner point $(\lceil n/2 \rceil, \lfloor n/2 \rfloor)$, if we make the kth step of the path go downwards or to the right, respectively, according as k appears in an odd or an even position in the permutation. This rule defines a one-to-one correspondence between 2-ordered permutations and n-step paths from corner to corner of the lattice diagram; for example, the path shown corresponds to the permutation

$$2\ 1\ 3\ 4\ 6\ 5\ 7\ 10\ 8\ 11\ 9\ 12\ 14\ 13\ 15. \tag{1}$$

Furthermore, we can attach "weights" to the vertical lines of the path, as the diagram shows; a line from (i, j) to $(i + 1, j)$ gets weight $|i - j|$. A little study will convince the reader that the sum of these weights along each path is equal to the number of inversions of the corresponding permutation. (See exercise 12.) Thus, for example, (1) has $1 + 0 + 1 + 0 + 1 + 2 + 1 + 0 = 6$ inversions.

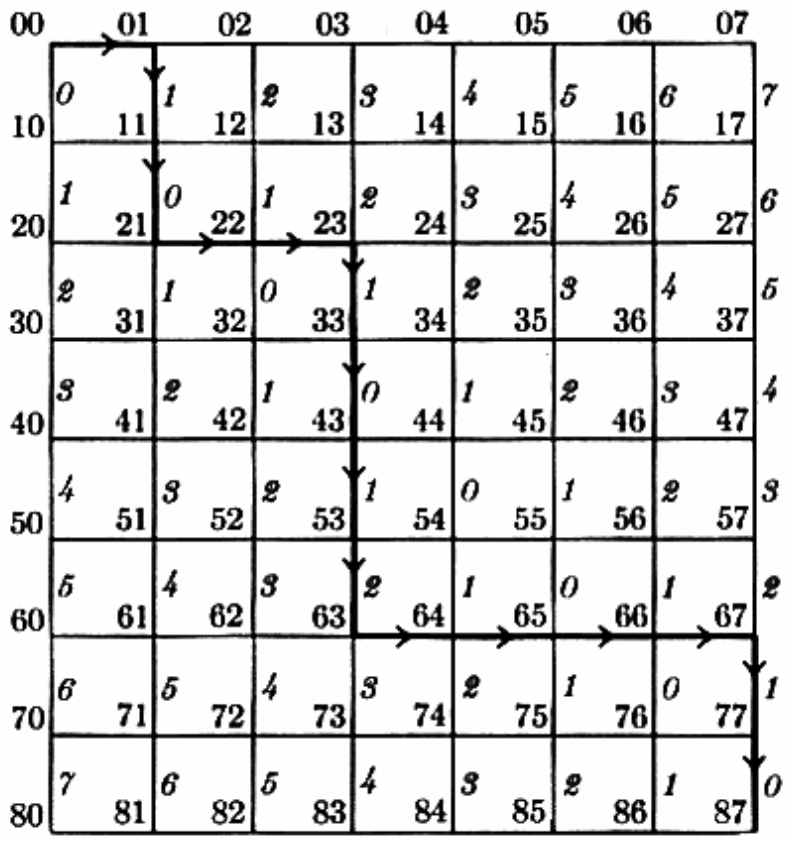


Fig. 11. Correspondence between 2-ordering and paths in a lattice. Italicized numbers are weights which correspond to the number of inversions in the 2-ordered permutation.

When $a \le a'$ and $b \le b'$, the number of relevant paths from (a, b) to (a', b') is the number of ways to mix $a' - a$ vertical lines with $b' - b$ horizontal lines, namely

$$\binom{a' - a + b' - b}{a' - a};$$

hence the number of permutations whose corresponding path traverses the vertical line segment from (i, j) to $(i + 1, j)$ is

$$\binom{i+j}{i}\binom{n - i - j - 1}{\lfloor n/2 \rfloor - j}.$$

Multiplying by the associated weight and summing over all segments gives

$$A_{2n} = \sum_{\substack{0 \le i \le n \\ 0 \le j \le n}} |i - j| \binom{i+j}{i}\binom{2n - i - j - 1}{n - j};$$

$$A_{2n+1} = \sum_{\substack{0 \le i \le n \\ 0 \le j \le n}} |i - j| \binom{i+j}{i}\binom{2n - i - j}{n - j}. \tag{2}$$

The absolute value signs in these sums make the calculations somewhat tricky, but exercise 14 shows that A_n has the surprisingly simple form $\lfloor n/2 \rfloor 2^{n-2}$. Hence the average number of inversions in a random 2-ordered permutation is

$$\lfloor n/2 \rfloor 2^{n-2} \Big/ \binom{n}{\lfloor n/2 \rfloor};$$

by Stirling's approximation this is asymptotically $\sqrt{\pi/128}\, n^{3/2} \approx 0.15 n^{3/2}$. The maximum number of inversions is easily seen to be

$$\binom{\lfloor n/2 \rfloor + 1}{2} \approx \frac{1}{8} n^2.$$

It is instructive to study the distribution of inversions more carefully, by examining the generating functions

$$h_1(z) = 1, \qquad h_2(z) = 1 + z,$$
$$h_3(z) = 1 + 2z, \qquad h_4(z) = 1 + 3z + z^2 + z^3, \qquad \ldots, \tag{3}$$

as in exercise 15. In this way we find that the standard deviation is also proportional to $n^{3/2}$, so the distribution is not extremely stable about the mean.

Now let us consider the general two-pass case of Algorithm D, when the increments are h and 1:

Theorem H. *The average number of inversions in an h-ordered permutation of $\{1, 2, \ldots, n\}$ is*

$$f(n, h) = \frac{2^{2q-1}q!q!}{(2q+1)!}\left(\binom{h}{2} q(q+1) + \binom{r}{2}(q+1) - \frac{1}{2}\binom{h-r}{2} q\right), \quad (4)$$

where $q = \lfloor n/h \rfloor$, $r = n \bmod h$.

This theorem is due to Douglas H. Hunt [Bachelor's thesis, Princeton University (April, 1967)]. Note that when $h \geq n$ the formula correctly gives $f(n, h) = \frac{1}{2}\binom{n}{2}$.

Proof. An h-ordered permutation contains r sorted subsequences of length $q + 1$, and $h - r$ of length q. Each inversion comes from a pair of distinct subsequences, and a given pair of distinct subsequences in a random h-ordered permutation defines a random 2-ordered permutation. The average number of inversions is therefore the sum of the average numbers of inversions between each pair of distinct subsequences, namely

$$\binom{r}{2}\frac{A_{2q+2}}{\binom{2q+2}{q+1}} + r(h-r)\frac{A_{2q+1}}{\binom{2q+1}{q}} + \binom{h-r}{2}\frac{A_{2q}}{\binom{2q}{q}} = f(n, h). \quad \blacksquare$$

Corollary. *If the sequence of increments $h_t, \ldots, h_1$ satisfies the condition*

$$h_{s+1} \bmod h_s = 0, \qquad \text{for} \qquad t > s \geq 1, \quad (5)$$

then the average number of move operations in Algorithm D is

$$\sum_{t \geq s \geq 1} \left(r_s f(q_s + 1, h_{s+1}/h_s) + (h_s - r_s) f(q_s, h_{s+1}/h_s)\right), \quad (6)$$

where $r_s = N \bmod h_s$, $q_s = \lfloor N/h_s \rfloor$, $h_{t+1} = Nh_t$, and f is defined in (4).

Proof. The process of h_s-sorting consists of a straight insertion sort on r_s (h_{s+1}/h_s)-ordered subfiles of length $q_s + 1$, and on $(h_s - r_s)$ such subfiles of length q_s. The divisibility condition implies that each of these subfiles is a "random" (h_{s+1}/h_s)-ordered permutation, in the sense that each (h_{s+1}/h_s)-ordered permutation is equally likely, since we are assuming that the original input was a random permutation of distinct elements. $\blacksquare$

Condition (5) in this corollary is always satisfied for *two-pass* Shell sorts when the increments are h and 1, respectively. If $q = \lfloor N/h \rfloor$ and $r = N \bmod h$, the quantity B in Program D will have an average value of

$$rf(q+1, N) + (h-r)f(q, N) + f(N, h) = \frac{r}{2}\binom{q+1}{2} + \frac{h-r}{2}\binom{q}{2} + f(N, h).$$

To a first approximation, the function $f(n, h)$ equals $(\sqrt{\pi}/8)n^{3/2}h^{1/2}$; compare it to the smooth curve in Fig. 12. Hence the running time for a two-pass Pro-

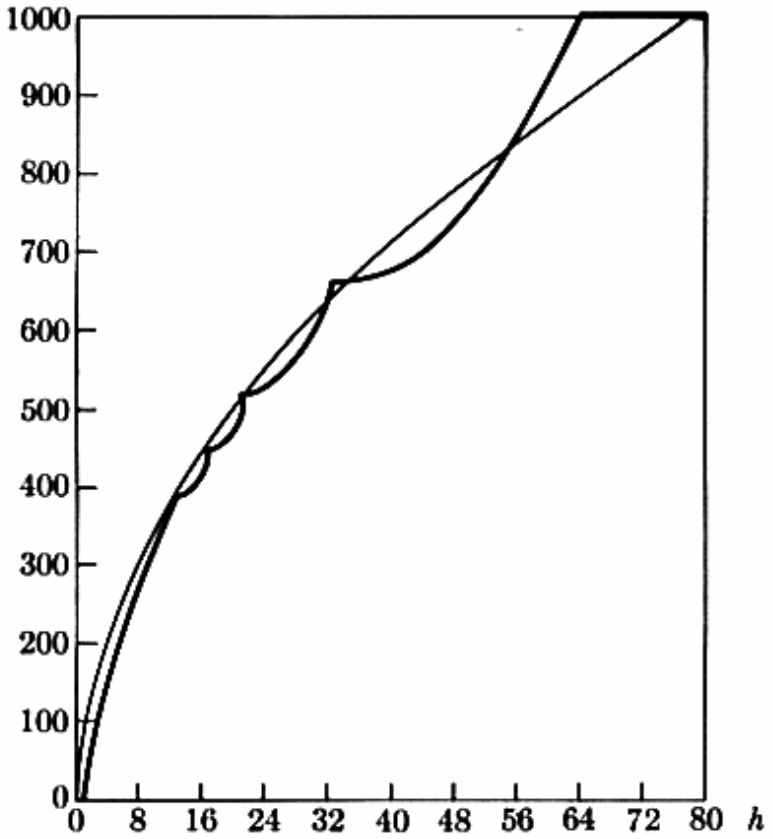


Fig. 12 The average number, $f(n, h)$, of inversions in an h-ordered file of n elements, shown for $n = 64$.

gram D is approximately proportional to $2N^2/h + \sqrt{\pi N^3 h}$. The best choice of h is therefore approximately $\sqrt[3]{16N/\pi} \approx 1.72\sqrt[3]{N}$; and with this choice of h we get an average running time proportional to $N^{5/3}$.

Thus we can make a substantial improvement over straight insertion, from $O(N^2)$ to $O(N^{1.667})$, just by using Shell's method with two increments. Clearly we can do even better when more increments are used. Exercise 18 discusses the optimum choice of $h_t, \ldots, h_1$ when t is fixed and when the h's are constrained by the divisibility condition; the running time decreases to $O(N^{1.5+\epsilon/2})$, where $\epsilon = 1/(2^t - 1)$, for large N. We cannot break the $N^{1.5}$ barrier by using the formulas above, since the last pass always contributes

$$f(N, h_2) \approx (\sqrt{\pi}/8)N^{3/2}h_2^{1/2}$$

inversions to the sum.

But our intuition tells us that we can do even better when the increments $h_t, \ldots, h_1$ do *not* satisfy the divisibility condition (5). For example, 8-sorting followed by 4-sorting followed by 2-sorting does not allow any interaction between keys in even and odd positions; therefore the final 1-sorting pass is faced with $O(N^{3/2})$ inversions. By contrast, 7-sorting followed by 5-sorting followed by 3-sorting jumbles things up in such a way that the final 1-sorting pass cannot encounter more than $2N$ inversions! (See exercise 26.) Indeed, an astonishing phenomenon occurs:

Theorem K. *If a k-ordered file is h-sorted, it remains k-ordered.*

Thus a file which is first 7-sorted, then 5-sorted, becomes both 7-ordered and 5-ordered. And if we 3-sort it, the result is ordered by 7's, 5's, and 3's.

Examples of this remarkable property appear in Table 4.

Table 4

DIMINISHING INCREMENT SORT (7, 5, 3, 1)

	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
7-sort:																
	275	087	426	061	509	170	677	503	653	512	154	908	612	897	765	703
5-sort:																
	154	087	426	061	509	170	677	503	653	512	275	908	612	897	765	703
3-sort:																
	061	087	170	154	275	426	512	503	653	612	509	765	677	897	908	703
1-sort:																
	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Proof. Exercise 20 shows that this theorem comes from the following fact:

Lemma L. *Let m, n, r be nonnegative integers, and let $x_1, \ldots, x_{m+r}$ and $y_1, \ldots, y_{n+r}$ be any sequences of numbers such that*

$$y_1 \le x_{m+1}, \qquad y_2 \le x_{m+2}, \qquad \ldots, \qquad y_r \le x_{m+r}. \tag{7}$$

If the x's and y's are sorted independently, so that $x_1 \le \cdots \le x_{m+r}$ and $y_1 \le \cdots \le y_{n+r}$, the relations (7) will still be valid.

Proof. All but m of the x's are known to dominate (i.e., to be greater than or equal to) some y, where distinct x's dominate distinct y's. Let $1 \le j \le r$. Since x_{m+j} after sorting dominates $m + j$ of the x's, it dominates at least j of the y's; so it dominates the *smallest* j of the y's; hence $x_{m+j} \ge y_j$ after sorting. ▮ ▮

Theorem K shows us that it is desirable to sort with relatively prime increments, but it does not lead directly to exact estimates of the number of moves made in Algorithm D. Since the number of permutations of $\{1, 2, \ldots, n\}$ which are both h-ordered and k-ordered is not always a divisor of $n!$, we can see that Theorem K does not tell the whole story; some k- and h-ordered files are obtained more often than others after k- and h-sorting. Moreover, there is no obvious way to find the "worst case" of Algorithm D for general increments $h_t, \ldots, h_1$. Therefore the analysis of this algorithm in the general case has baffled everyone so far; essentially all that is known is the approximate asymptotic form of the maximum running time in certain cases:

Theorem P. *The running time for Algorithm D is $O(N^{3/2})$ when $h_s = 2^s - 1$, $1 \le s \le t = \lfloor \log_2 N \rfloor$.*

Proof. It suffices to bound B_s, the number of moves in pass s, in such a way that $B_t + \cdots + B_1 = O(N^{3/2})$. During the first $t/2$ passes, for $t \geq s \geq t/2$, we may use the obvious bound $B_s = O(h_s(N/h_s)^2)$, and for subsequent passes we may use the result of exercise 23, $B_s = O(Nh_{s+2}h_{s+1}/h_s)$; hence $B_t + \cdots + B_1 = O(N(2 + 2^2 + \cdots + 2^{t/2} + 2^{t/2} + \cdots + 2)) = O(N^{3/2})$. ∎

This theorem is due to A. A. Papernov and G. V. Stasevich, *Problemy Peredachi Informatsii* **1**, 3 (1965), 81–98. It gives an upper bound on the *maximum* running time of the algorithm, not merely a bound on the *average* running time. The result is not trivial since the maximum running time when the h's satisfy the divisibility constraint (5) is of order N^2; and exercise 24 shows that the exponent 3/2 cannot be lowered.

An interesting improvement of Theorem P was discovered in 1969 by Vaughan Pratt: *If the increments are chosen to be the set of all numbers of the form 2^p3^q which are less than N, the running time of Algorithm D is of order $N(\log N)^2$.* In this case we can also make several important simplifications to the algorithm. Unfortunately, Pratt's method requires a comparatively large number of passes, so it is not the best way to choose the increments unless N is quite large; see exercises 30 and 31.

Let us consider the *total* running time of Program D, namely $(9B + 10NT + 13T - 10S - 3A + 1)u$. Table 5 shows the average running time for various

Table 5

ANALYSIS OF ALGORITHM D WHEN $N = 8$

Increments	A_{ave}	B_{ave}	S	T	MIX time
1	1.718	14.000	1	1	$204.85u$
2 1	2.667	9.657	3	2	$235.91u$
3 1	2.917	9.100	4	2	$220.16u$
4 1	3.083	10.000	5	2	$217.75u$
5 1	2.601	10.000	6	2	$210.00u$
6 1	2.135	10.667	7	2	$206.60u$
7 1	1.718	12.000	8	2	$208.85u$
4 2 1	3.500	8.324	7	3	$272.32u$
5 3 1	3.301	8.167	9	3	$251.60u$
3 2 1	3.320	7.829	6	3	$278.50u$

sequences of increments when $N = 8$; each of the entries in this table can be calculated from the formulas above or in exercise 19, except when the increments are 5 3 1 or 3 2 1; in the latter cases an exhaustive study of all 8! permutations was made. Note that for this small value of N the bookkeeping operations dominate the running time, so that 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 $191.85u$.) Curiously, the best two-pass algorithm occurs when $h_2 = 6$, since a large value of S is more important here than a small value of B. Similarly, the three increments 3 2 1 minimize the average number of moves, but they do not lead to the best three-pass sequence. It may be of interest to record here some "worst case" permutations which maximize the number of moves, since the general construction of such permutations is still unknown:

$$h_3 = 5, \quad h_2 = 3, \quad h_1 = 1: \qquad 8\ 5\ 2\ 6\ 3\ 7\ 4\ 1 \qquad \text{(19 moves)}$$

$$h_3 = 3, \quad h_2 = 2, \quad h_1 = 1: \qquad 8\ 3\ 5\ 7\ 2\ 4\ 6\ 1 \qquad \text{(17 moves)}$$

As N grows larger we have a slightly different picture. Table 6 shows the approximate number of moves for various sequences of increments when $N = 1000$. The first few entries satisfy the divisibility constraints (5), so that formula (6) can be used; empirical tests were used to get approximate average values for other increment sequences. Five random files of 1000 elements were generated, and they were each sorted with each of the sequences of increments.

Some patterns are evident in this data, but the behavior of Algorithm D still remains very obscure. Shell originally suggested using the increments $\lfloor N/2 \rfloor, \lfloor N/4 \rfloor, \lfloor N/8 \rfloor, \ldots$, but this is undesirable when the binary representation of N contains a long string of zeros. Lazarus and Frank [*CACM* **3** (1960), 20–22] suggested using essentially the same sequence, but adding 1 when necessary, to make all increments odd. Hibbard [*CACM* **6** (1963), 206–213] suggested using increments of the form $2^k - 1$; Papernov and Stasevich suggested the form $2^k + 1$. Other natural sequences investigated in Table 6 involve the numbers $(2^k - (-1)^k)/3$, $(3^k - 1)/2$, and the Fibonacci numbers.

The minimum number of moves, about 6600, was observed for increments of the form $2^k + 1$; but it is important to realize that the number of moves is not the only consideration, even though it asymptotically dominates the running time. Since Program D takes $9B + 10NT + \cdots$ units of time, we see that saving one pass is about as desirable as saving $\frac{10}{9}N$ moves; we are willing to add 1100 moves if we can save one pass. Therefore it seems unwise to start with h_t greater than, say, $\frac{1}{3}N$, since a large increment will not decrease the subsequent number of moves enough to justify the first pass.

A large number of experiments on Algorithm D were conducted by James Peterson and David L. Russell at Stanford University in 1971; they found that the average number of moves, B, can be reasonably well approximated by the following formulas for $100 \le N \le 60000$:

$1.09N^{1.27}$ or $.30N(\ln N)^2 - 1.35N \ln N$ for increments $2^k + 1, \ldots, 9, 5, 3, 1$;

$1.22N^{1.26}$ or $.29N(\ln N)^2 - 1.26N \ln N$ for increments $2^k - 1, \ldots, 15, 7, 3, 1$;

$1.12N^{1.28}$ or $.36N(\ln N)^2 - 1.73N \ln N$ for increments $(2^k \pm 1)/3, \ldots, 11, 5, 3, 1$;

$1.66N^{1.25}$ or $.33N(\ln N)^2 - 1.26N \ln N$ for increments $(3^k - 1)/2, \ldots, 40, 13, 4, 1$.

Table 6

ANALYSIS OF ALGORITHM D WHEN $N = 1000$

Increments	A_{ave}	B_{ave}	T
1	6.485	249750	1
17 1	64.66	41666.5	2
60 6 1	158.1	26360.6	3
140 20 4 1	262.1	21912.5	4
256 64 16 4 1	361.9	20458.9	5
576 192 48 16 4 1	419.2	20087.6	6
729 243 81 27 9 3 1	378.3	18533.2	7
512 256 128 64 32 16 8 4 2 1	492.5	16434.7	10
500 250 125 62 31 15 7 3 1	525	7600	9
501 251 125 63 31 15 7 3 1	575	7200	9
511 255 127 63 31 15 7 3 1	550	7100	9
255 127 63 31 15 7 3 1	450	7300	8
127 63 31 15 7 3 1	300	8150	7
63 31 15 7 3 1	200	9700	6
31 15 7 3 1	125	13600	5
513 257 129 65 33 17 9 5 3 1	575	6600	10
257 129 65 33 17 9 5 3 1	450	6800	9
129 65 33 17 9 5 3 1	300	7600	8
65 33 17 9 5 3 1	200	9400	7
33 17 9 5 3 1	125	13000	6
683 341 171 85 43 21 11 5 3 1	500	7050	10
341 171 85 43 21 11 5 3 1	500	7300	9
255 63 15 7 3 1	400	8600	6
257 65 17 5 3 1	400	8700	6
341 85 21 5 3 1	425	9300	6
610 377 233 144 89 55 34 21 13 8 5 3 2 1	550	7400	14
377 144 55 21 8 3 1	475	8800	7
365 122 41 14 5 2 1	450	8300	7
364 121 40 13 4 1	450	9200	6
121 40 13 4 1	275	9900	5

For example when $N = 20000$ we get $B \approx 31000, 33000, 35000, 39000$, with these respective types of increments. Table 7 shows typical breakdowns of

Table 7

MOVES PER PASS; EXAMPLES WITH $N = 20000$

h_s	B_s	h_s	B_s	h_s	B_s
4095	19460	4097	19550	3280	25210
2047	15115	2049	14944	1093	28324
1023	15869	1025	15731	364	35477
511	18891	513	18548	121	47158
255	22306	257	21827	40	62110
127	27400	129	27814	13	88524
63	35053	65	33751	4	74599
31	34677	33	34303	1	34666
15	51054	17	46044		
7	40382	9	35817		
3	24044	5	19961		
1	16789	3	9628		
		1	13277		

moves per pass obtained in three of these experiments. Curiously, *both* of the functional forms $\alpha N(\ln N)^2 + \beta N \ln N$ and βN^α seem to give a pretty good fit to the observed data, although the exponential form was noticeably better than the logarithmic for smaller values of N. Further experiments were made for the increments $2^k - 1$, letting N range as high as 250,000; forty-five trials for $N = 250{,}000$ gave $B \approx 7{,}900{,}000$ with an observed standard deviation of about 50,000. The "best fit" formulas for the range $100 \le N \le 250{,}000$ turned out to be $1.21N^{1.26}$ and $.39N(\ln N)^2 - 2.33\ N \ln N$, respectively. Since the coefficients of the exponential form remained about the same, while the coefficients of the logarithmic form changed rather drastically, it is reasonable to conjecture that the exponential form gives the true asymptotic behavior of Shell's method.

This empirical data by no means exhausts the possibilities, and we have found no grounds for making strong assertions about what sequences of increments are best in Algorithm D. Since the increments $(3^k - 1)/2$ do not take substantially more moves, and since they require only about five-eighths as many passes as the increments of other forms, the evidence indicates that *it is reasonable to choose the increments in the following way:*

$$\textit{Let } h_1 = 1,\ h_{s+1} = 3h_s + 1, \textit{ and stop with } h_t \textit{ when } h_{t+2} \ge N. \tag{8}$$

List insertion. Let us now leave Shell's method and consider other types of improvements over straight insertion. One of the most important general ways

to improve on a given algorithm is to examine its data structures carefully, since a reorganization of data structures to avoid unnecessary operations often leads to substantial improvements. Further discussion of this general idea appears in Section 2.4, where a rather complex algorithm is studied; let us consider how it applies to a very simple algorithm like straight insertion. What is the most appropriate data structure for Algorithm S?

Straight insertion involves two basic operations:

i) scanning an ordered file to find the largest key less than or equal to a given key; and

ii) inserting a new record into a specified part of the ordered file.

The file is obviously a linear list, and Algorithm S handles this list by using sequential allocation (Section 2.2.2); therefore it is necessary to move roughly half of the records in order to accomplish each insertion operation. On the other hand, we know that linked allocation (Section 2.2.3) is ideally suited to insertion, since only a few links need to be changed; and the other operation, sequential scanning, is about as easy with linked allocation as with sequential allocation. Only one-way linkage is needed, since we always scan the list in the same direction. Therefore we conclude that the "right" data structure for straight insertion is a one-way, linked linear list. It also becomes convenient to revise Algorithm S so that the list is scanned in increasing order:

Algorithm L (*List insertion*). Records $R_1, \ldots, R_N$ are assumed to contain keys $K_1, \ldots, K_N$, and "link fields" $L_1, \ldots, L_N$ capable of holding the numbers 0 through N; and there is another link field L_0 in an artificial record R_0 at the beginning of the file. This algorithm sets the link fields so that the records are linked together in ascending order. Thus, if $p(1) \ldots p(N)$ is the "stable" permutation which makes $K_{p(1)} \leq \cdots \leq K_{p(N)}$, this algorithm will yield

$$L_0 = p(1); \qquad L_{p(i)} = p(i+1), \quad \text{for} \quad 1 \leq i \leq N; \qquad L_{p(N)} = 0. \tag{9}$$

L1. [Loop on j.] Set $L_0 \leftarrow N$, $L_N \leftarrow 0$. (L_0 acts as the "head" of the list, and 0 acts as a null link; hence the list is essentially circular.) Perform steps L2 through L5 for $j = N-1, N-2, \ldots, 1$; then terminate the algorithm.

L2. [Set up p, q, K.] Set $p \leftarrow L_0$, $q \leftarrow 0$, $K \leftarrow K_j$. (In the following steps we will insert R_j into its proper place in the linked list, by comparing K with the previous keys in ascending order. The variables p and q act as pointers to the current place in the list, with $p = L_q$ so that q is one step behind p.)

L3. [Compare K, K_p.] If $K \leq K_p$, go to step L5. (We have found the desired position for record R, between R_q and R_p in the list.)

L4. [Bump p, q.] Set $q \leftarrow p$, $p \leftarrow L_q$. If $p > 0$, go back to step L3. (If $p = 0$, K is the largest key found so far; hence record R belongs at the end of the list, between R_q and R_0.)

L5. [Insert into list.] Set $L_q \leftarrow j$, $L_j \leftarrow p$. ∎

This algorithm is important not only because it is a simple sorting method, but also because it occurs frequently as part of other list-processing algorithms. Table 8 shows the first few steps which occur when our sixteen example numbers are sorted.

Table 8

EXAMPLE OF LIST INSERTION

j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K_j:	—	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
L_j:	16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0
L_j:	16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0	15
L_j:	14	—	—	—	—	—	—	—	—	—	—	—	—	—	16	0	15

Program L (*List insertion*). We assume that K_j is stored in INPUT $+ j$(0:3), and L_j is stored in INPUT+j(4:5). rI1 $\equiv j$; rI2 $\equiv p$; rI3 $\equiv q$; rA(0:3) $\equiv K$.

```
01   KEY    EQU   0:3
02   LINK   EQU   4:5
03   START  ENT1  N                    1            L1. Loop on j. j ← N.
04          ST1   INPUT(LINK)          1            L0 ← N.
05          STZ   INPUT+N(LINK)        1            LN ← 0.
06          JMP   6F                   1            Go to decrease j.
07   2H     LD2   INPUT(LINK)        N − 1          L2. Set up p, q, K. p ← L0.
08          ENT3  0                  N − 1          q ← 0.
09          LDA   INPUT,1            N − 1          K ← Kj.
10   3H     CMPA  INPUT,2(KEY)   B + N − 1 − A      L3. Compare K, Kp.
11          JLE   5F             B + N − 1 − A      To L5 if K ≤ Kp.
12   4H     ENT3  0,2                  B            L4. Bump p, q. q ← p.
13          LD2   INPUT,3(LINK)        B            p ← Lq.
14          J2P   3B                   B            To L3 if p > 0.
15   5H     ST1   INPUT,3(LINK)      N − 1          L5. Insert into list. Lq ← j.
16          ST2   INPUT,1(LINK)      N − 1          Lj ← p.
17   6H     DEC1  1                    N
18          J1P   2B                   N            N > j ≥ 1. ∎
```

The running time of this program is $7B + 14N - 3A - 6$ units, where N is the length of the file, A is the number of right-to-left maxima, and B is the number of inversions in the original permutation. (Cf. 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 $20N$ additional units of time.) Program S requires $(9B + 10N - 3A - 9)u$, and since B is about $\frac{1}{4}N^2$, we can see that the extra memory space used for the link fields has saved about 22 percent of the execution time. Another 22 percent can be saved by careful programming (see exercise 33), but the running time remains proportional to N^2.

To summarize what we have done so far: We started with Algorithm S, a simple and natural sorting algorithm which does about $\frac{1}{4}N^2$ comparisons and $\frac{1}{4}N^2$ moves. We improved it in one direction by considering binary insertion, which does about $N \log_2 N$ comparisons and $\frac{1}{4}N^2$ moves. Changing the data structure slightly with "two-way insertion" cuts the number of moves down to about $\frac{1}{8}N^2$. Shell's "diminishing increment" sorting technique cuts the number of comparisons and moves to about $N^{1.3}$, for N in a practical range; as $N \to \infty$ this number can be lowered to order $N(\log N)^2$. Another way to improve on Algorithm S, using a linked data structure, gave us the list insertion method, which does about $\frac{1}{4}N^2$ comparisons, 0 moves, and $2N$ changes of links.

Is it possible to marry the best features of these methods, reducing the number of comparisons to order $N \log N$ as in binary insertion, yet reducing the number of moves as in list insertion? The answer is yes, by going to a tree-structured arrangement. This possibility was first explored about 1957 by D. J. Wheeler, who suggested using two-way insertion until it becomes necessary to move some data; then instead of moving the data, a pointer to another area of memory is inserted, and the same technique is applied recursively to all items which are to be inserted into this new area of memory. Wheeler's original method [see A. S. Douglas, *Comp. J.* **2** (1959), 5] is a complicated combination of sequential and linked memory, with nodes of varying size; for our 16 example numbers the tree of Fig. 13 would be formed. A similar but simpler tree-insertion

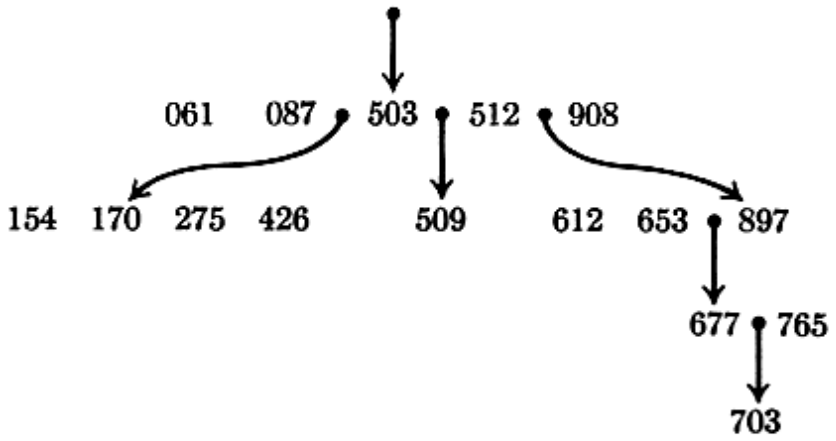


Fig. 13. Example of Wheeler's tree insertion scheme.

scheme, using binary trees, was independently devised by C. M. Berners-Lee about 1958 [see *Comp. J.* **3** (1960), 174, 184]. Since this method and its refinements are quite important for searching as well as sorting, they are discussed at length in Chapter 6.

Still another way to improve on straight insertion is to consider inserting several things at a time. For example, if we have a file of 1000 items, and if 998 of them have already been sorted, Algorithm S makes two more passes through the file (first inserting R_{999}, then R_{1000}). We can obviously save time if we first compare K_{999} with K_{1000}, to see which is larger, then insert them *both* with one look at the file. A combined operation of this kind involves about

$\frac{2}{3}N$ comparisons and moves (cf. exercise 3.4.2–5), instead of two passes each with about $\frac{1}{2}N$ comparisons and moves.

In other words, it is generally a good idea to "batch" operations that require long searches, so that multiple operations can be done together. If we carry this idea to its natural conclusion, we rediscover the method of sorting by merging, which is so important it is discussed below in a separate section.

Address calculation sorting. Surely by now we have exhausted all possible ways to improve on the simple method of straight insertion; but let's look again! Suppose someone gives you a blank sheet of paper and says that he is going to read you some words. Your job is to alphabetize the words, and to give him the sorted list on the paper. Then if he tells you a word starting with A, you will tend to write it near the top of the page, while a word starting with Z will probably be placed near the bottom, and so on. A similar technique is used when putting books into a bookshelf in order by the authors' names, when the books are given to you in random order on the floor: you estimate the final position of each book as you put it in place, thereby reducing the number of comparisons and moves you have to make. (The whole process is somewhat more efficient if you start with a little more shelf space than is absolutely necessary.) This method was first suggested for computer sorting by Isaac and Singleton, *JACM* **3** (1956), 169–174, and it has been further developed by Kronmal and Tartar, *Proc. ACM Nat'l Conf.* **21** (1966), 331–337.

Address calculation sorting usually requires additional storage space proportional to N, either to leave enough room so that excessive moving is not required, or to maintain auxiliary tables which help account for irregularities in the distribution of keys. (See the "distribution counting" sort, Algorithm 5.2D, which is a form of address calculation.) We can probably make best use of this additional memory space if we devote it to link fields, as in the list insertion method. In this way we can also avoid having separate areas for input and output; everything can be done in the same area of memory.

The basic idea is to generalize list insertion so that *several* lists are kept, not just one. Each list is used for certain ranges of keys. We make the important assumption that the keys are pretty evenly distributed, not "bunched up" irregularly: the set of all possible values of the keys is partitioned into M parts, and we assume a probability of $1/M$ that a given key falls into a given part. Then we provide additional storage for M list heads, and each list is maintained as in simple list insertion.

It is not necessary to give the algorithm in great detail here; the method is simply to begin with all list heads set to Λ. As each new item enters, we first decide which of the M parts its key falls into, then we insert it into the corresponding list as in Algorithm L.

To illustrate the method, suppose that the 16 keys used in our examples are divided into the $M = 4$ ranges 0–249, 250–499, 500–749, 750–999. We obtain the following configurations as sorting proceeds:

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

Because linked memory is used, there is no storage allocation problem caused by varying-length lists.

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

$$0 \le \texttt{KEY} < (\texttt{BYTESIZE})^3.$$

The program divides this range into M equal parts by multiplying each key by a suitable constant. The list heads are in locations HEAD+1 through HEAD+M.

```
01   KEY    EQU   1:3
02   LINK   EQU   4:5
03   START  ENT2  M                    1
04          STZ   HEAD,2              M      HEAD[p] ← Λ.
05          DEC2  1                   M
06          J2P   *-2                 M      M ≥ p ≥ 1.
07          ENT1  N                   1      j ← N.
08   2H     LDA   INPUT,1(KEY)        N
09          MUL   =M(1:3)=            N      rA ← ⌊M · K_j/BYTESIZE^3⌋
10          STA   *+1(1:2)            N
11          ENT3  0                   N      q ← rA.
12          INC3  HEAD+1-INPUT        N      q ← LOC(HEAD[q]).
13          LDA   INPUT,1             N      K ← K_j.
14          JMP   4F                  N      To set p.
15   3H     CMPA  INPUT,2(KEY)    B+N-A
16          JLE   5F              B+N-A      To insert, if K ≤ K_p.
17          ENT3  0,2                 B      q ← p.
18   4H     LD2   INPUT,3(LINK)     B+N      p ← LINK(q).
19          J2P   3B                B+N      Jump if not end of list.
20   5H     ST1   INPUT,3(LINK)       N      LINK(q) ← LOC(R_j).
21          ST2   INPUT,1(LINK)       N      LINK(LOC(R_j)) ← p.
22   6H     DEC1  1                   N
23          J1P   2B                  N      N ≥ j ≥ 1. ∎
```

This program is written for general M, but it would be better to fix M at some convenient value; for example, we might choose $M =$ BYTESIZE, so that the list heads could be cleared with a single MOVE instruction and the multiplication sequence of lines 08–11 could be replaced by the single instruction

LD3 INPUT,1 (1:1). 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 Program M is $7B + 31N - 3A + 4M + 2$ units, where M is the number of lists, N is the number of records sorted, and A, B respectively count the right-to-left maxima and the inversions present among the keys belonging to each list. (In contrast to other time analyses of this section, we are here treating the rightmost element of a nonempty permutation as a "right-to-left maximum," instead of ignoring it.) We have already studied A and B for $M = 1$, when their average values are respectively H_N and $\frac{1}{2}\binom{N}{2}$. By our assumption about the distribution of keys, the probability that a given list contains precisely n items at the conclusion of sorting is the "binomial" probability

$$\binom{N}{n}\left(\frac{1}{M}\right)^n\left(1-\frac{1}{M}\right)^{N-n} \tag{10}$$

Therefore the average values of A and B in the general case are

$$A_{\text{ave}} = M\sum_n \binom{N}{n}\left(\frac{1}{M}\right)^n\left(1-\frac{1}{M}\right)^{N-n} H_n; \tag{11}$$

$$B_{\text{ave}} = M\sum_n \binom{N}{n}\left(\frac{1}{M}\right)^n\left(1-\frac{1}{M}\right)^{N-n}\binom{n}{2}. \tag{12}$$

By Theorem 1.2.7A,

$$\sum_n \binom{N}{n}(M-1)^{-n}H_n = \left(1-\frac{1}{M}\right)^{-N}(H_N - \ln M) + \epsilon,$$

$$0 < \epsilon = \sum_{n>N}\frac{1}{n}\left(1-\frac{1}{M}\right)^{n-N} < \frac{M-1}{N+1};$$

hence

$$A_{\text{ave}} = M(H_N - \ln M) + \delta, \qquad 0 < \delta < \frac{M^2}{N+1}\left(1-\frac{1}{M}\right)^{N+1}. \tag{13}$$

(This formula is practically useless when $M \approx N$. A more detailed discussion of the asymptotic behavior of A_{ave}, when $M = N/\alpha$, appears in exercise 5.2.2–57.) Using the identity

$$\binom{N}{n}\binom{n}{2} = \binom{N}{2}\binom{N-2}{n-2},$$

which is a special case of Eq. 1.2.6–20, the sum in (11) is easily evaluated, and we obtain

$$B_{\text{ave}} = \frac{1}{2M}\binom{N}{2}. \tag{14}$$

(For the standard deviation of B, see exercise 37.) Hence the total running time of Program M, for fixed M as $N \to \infty$, is

$$\begin{aligned}
&\min && 31N + M + 2,\\
&\text{ave} && 1.75N^2/M + 31N - 3MH_N + 3M \ln M + 4M - 3 - 1.75N/M + 2,\\
&\max && 3.50N^2 + 24.5N + 4M + 2. \qquad (15)
\end{aligned}$$

Note that when M is not too large *we are speeding up the average time by a factor of M*; $M = 10$ will sort about ten times as fast as $M = 1$! However, the maximum time is much larger than the average time; this reiterates the assumption we have made about a fairly equal distribution of keys, since the worst case occurs when all records pile onto the same list.

If we set $M = N$, the average running time is approximately $34.36N$; when $M = \frac{1}{2}N$ it is slightly more, approximately $34.52N$; and when $M = \frac{1}{10}N$ it is approximately $48.04N$. (Note that $10N$ of these MIX time units are spent in the multiplication instruction alone!) *We have achieved a sorting method of order N, provided only that the keys are reasonably well spread out over their range.*

EXERCISES

1. [*10*] Is Algorithm S a "stable" sorting algorithm?

2. [*11*] Would Algorithm S still sort numbers correctly if the relation "$K \geq K_i$" in step S3 were replaced by "$K > K_i$"?

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

▶ **4.** [*M20*] Find the minimum and maximum running times for Program S, as a function of N.

▶ **5.** [*M27*] Find the generating function $g_N(z) = \sum_{k \geq 0} p_{Nk} z^k$ for the total running time of Program S, where p_{Nk} is the probability that Program S takes exactly k units of time, given a random permutation of $\{1, 2, \ldots, N\}$ as input. Also calculate the standard deviation of the running time, given N.

6. [*23*] The two-way insertion method illustrated in Table 2 seems to imply that there is an output area capable of holding up to $2N + 1$ records, in addition to the input area containing N records. Show that two-way insertion can be done using only enough space for $N + 1$ records, including both input and output.

7. [*M20*] If $a_1 a_2 \ldots a_n$ is a random permutation of $\{1, 2, \ldots, n\}$, what is the average value of $|a_1 - 1| + |a_2 - 2| + \cdots + |a_n - n|$? (This is n times the average net distance traveled by a record during a sorting process.)

8. [*10*] Is Algorithm D a "stable" sorting algorithm?

9. [*20*] What are the quantities A and B, and the total running time of Program D, corresponding to Tables 3 and 4? Discuss the relative merits of Shell's method versus straight insertion in this case.

▶ **10.** [*22*] If $K_j \geq K_{j-h}$ in 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.

11. [*M10*] What path in a lattice like Fig. 11 corresponds to the permutation 1 2 5 3 7 4 8 6 9 11 10 12?

12. [*M20*] Prove that the sum of the vertical weights on a lattice path equals the number of inversions in the corresponding 2-ordered permutation.

▶ **13.** [*M22*] Explain how to put weights on the *horizontal* line segments of a lattice, instead of the vertical segments, so that the sum of the horizontal weights on a lattice path is the number of inversions in the corresponding 2-ordered permutation.

14. [*M24*] (a) Show that, in the sums defined by Eq. (2), $A_{2n+1} = 2A_{2n}$. (b) The general identity of exercise 1.2.6–26 simplifies to

$$\sum_k \binom{2k+s}{k} z^k = \frac{1}{\sqrt{1-4z}}\left(\frac{1-\sqrt{1-4z}}{2z}\right)^s$$

if we set $r = -s-1$, $t = 1$. By considering the sum $\sum_n A_{2n}z^n$, show that

$$A_{2n} = n \cdot 4^{n-1}.$$

▶ **15.** [*HM33*] Let $g_n(z)$, $\bar{g}_n(z)$, $h_n(z)$, $\bar{h}_n(z)$ be $\sum z^{\text{total weight of path}}$ summed over all lattice paths of length $2n$ from $(0, 0)$ to (n, n), subject to certain restrictions on the vertices on the paths: For $h_n(z)$, there is no restriction, but for $g_n(z)$ the path must avoid all vertices (i, j) with $i > j$; $\bar{h}_n(z)$ and $\bar{g}_n(z)$ are defined similarly, except that all vertices (i, i) are also excluded, for $0 < i < n$. Thus

$$g_0(z) = 1, \quad g_1(z) = z, \quad g_2(z) = z^3 + z^2; \quad \bar{g}_1(z) = z, \quad \bar{g}_2(z) = z^3;$$
$$h_0(z) = 1, \quad h_1(z) = z+1, \quad h_2(z) = z^3 + z^2 + 3z + 1;$$
$$\bar{h}_1(z) = z+1, \quad \bar{h}_2(z) = z^3 + z.$$

Find recurrence relations defining these functions, and use the recurrence relations to prove that

$$h_n''(1) + h_n'(1) = \frac{7n^3 + 4n^2 + 4n}{30}\binom{2n}{n}.$$

(The exact formula for the variance of the number of inversions in a random 2-ordered permutation of $\{1, 2, \ldots, 2n\}$ is therefore easily found; it is asymptotically $(\frac{7}{30} - \frac{\pi}{16})n^3$.)

16. [*M24*] Find a formula for the maximum number of inversions in an h-ordered permutation of $\{1, 2, \ldots, n\}$. What is the maximum possible number of moves in Algorithm D when the increments satisfy the divisibility condition (5)?

17. [*M21*] Show that, when $N = 2^t$ and $h_s = 2^{s-1}$ for $t \geq s \geq 1$, there is a unique permutation of $\{1, 2, \ldots, N\}$ which maximizes the number of move operations performed by Algorithm D. Find a simple way to describe this permutation.

18. [*HM24*] For large N the sum (6) can be estimated as

$$\frac{1}{4}\frac{N^2}{h_t} + \frac{\sqrt{\pi}}{8}\left(\frac{N^{3/2}h_t^{1/2}}{h_{t-1}} + \cdots + \frac{N^{3/2}h_2^{1/2}}{h_1}\right).$$

What real values of $h_t, \ldots, h_1$ minimize this expression when N and t are fixed and $h_1 = 1$?

▶ **19.** [*M25*] What is the average value of the quantity A in the timing analysis of Program D, when the increments satisfy the divisibility condition (5)?

20. [*M20*] Show that Theorem K follows from Lemma L.

21. [*M25*] Let h and k be relatively prime positive integers. Show that the largest integer which cannot be represented in the form $ah + bk$, when a and b are *nonnegative* integers, is $hk - h - k$. Hence if a file is both h-ordered and k-ordered, we have $K_i \le K_j$ whenever $j - i \ge (h-1)(k-1)$.

22. [*M30*] Prove that all integers $\ge 2^h(2^h - 1)$ can be represented in the form $a_0(2^h - 1) + a_1(2^{h+1} - 1) + a_2(2^{h+2} - 1) + \cdots$, where the a_j's are nonnegative integers; but $2^h(2^h - 1) - 1$ cannot be so represented. Furthermore, exactly $2^{h-1}(2^h + h - 3)$ positive integers are unrepresentable in this form.

Find analogous formulas when the quantities $2^k - 1$ are replaced by $2^k + 1$ in the representations.

▶ **23.** [*M22*] Prove that if h_{s+2} and h_{s+1} are relatively prime, the number of moves which occur while Algorithm D is using the increment h_s is $O(Nh_{s+2}h_{s+1}/h_s)$. *Hint:* See exercise 21.

24. [*M42*] Prove that Theorem P is best possible, in the sense that the exponent 3/2 cannot be lowered.

▶ **25.** [*M22*] How many permutations of $\{1, 2, \ldots, n\}$ are both 3-ordered and 2-ordered? What is the maximum number of inversions in such a permutation? What is the total number of inversions among all such permutations?

26. [*M35*] Can a file of N elements have more than N inversions if it is 3-, 5-, and 7-ordered?

27. [*M50*] Find an efficient algorithm to construct permutations which maximize the number of moves in Algorithm D, given $N, h_t, h_{t-1}, \ldots, h_1$.

28. [*15*] Which sequence of increments shown in Table 6 is best from the standpoint of Program D, considering the total running time?

29. [*M42*] For $N = 1000$ and various values of t, find empirical values of $h_t, \ldots, h_1$ which seem to minimize the average number of moves made by Algorithm D, in the sense that, if any one of the h's is varied while the others are fixed, the average number of moves increases.

30. [*M23*] (V. Pratt.) If the set of increments is $\{2^p3^q \mid 2^p3^q < N\}$, show that the number of passes is approximately $\frac{1}{2}(\log_2 N)(\log_3 N)$, and the number of moves per pass is at most $N/2$. In fact, if $K_{j-h} > K_j$ on any pass, we will always have $K_{j-3h}, K_{j-2h} \le K_j < K_{j-h} \le K_{j+h}, K_{j+2h}$; so we may simply interchange K_{j-h} and K_j and increase j by $2h$, saving two of the comparisons of Algorithm D. *Hint:* See exercise 25.

▶ **31.** [*25*] Write a MIX 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.

32. [*10*] What would be the final contents of $L_0\, L_1 \ldots L_{16}$ if the list insertion sort in Table 8 were carried through to completion?

▶ **33.** [*25*] Find a way to improve on Program L so that its running time is dominated by $5B$ instead of $7B$, where B is the number of inversions.

34. [*M10*] Verify formula (10).

35. [*18*] Assume that the byte size of MIX is 100, and that the sixteen example keys in Table 8 are actually 503000, 087000, 512000, . . . , 703000. Determine the running time of Programs L and M on this data, when $M = 4$.

36. [*HM16*] If a program runs in approximately $A/M + B$ units of time and uses $C + M$ locations in memory, what choice of M gives the optimum product of space times time?

37. [*M25*] Let $g_n(z)$ be the generating function for inversions in a random permutation of n objects (cf. Eq. 5.1.1–11). Let $g_{NM}(z)$ be the corresponding generating function for the quantity B in Program M. Show that

$$\sum_{N\geq 0} g_{NM}(z)M^N w^N/N! = \left(\sum_{n\geq 0} g_n(z)w^n/n!\right)^M,$$

and use this formula to derive the variance of B.

38. [*HM23*] (R. M. Karp.) Let $F(x)$ be a distribution function for a probability distribution, with $F(0) = 0$ and $F(1) = 1$. Given that the keys $K_1, K_2, \ldots, K_N$ are independently chosen at random from this distribution, and that $M = cN$, where c is constant and $N \to \infty$, prove that the running time of Program M is $O(N)$ when F is sufficiently smooth. (A key K is inserted into list j when $\lfloor MK \rfloor = j - 1$; this occurs with probability $F(j/M) - F((j-1)/M)$. Only the case $F(x) = x, 0 \leq x \leq 1$, is treated in the text.)

39. [*M25*] (*Disorder in a library.*) Casual users of a library often put books back on the shelves in the wrong place. One way to measure the amount of disorder present in a library is to consider the minimum number of times we would have to take a book out of one place and insert it in another, before all books are restored to the correct order.

Thus let $\pi = a_1 a_2 \ldots a_n$ be a permutation of $\{1, 2, \ldots, n\}$. A "deletion-insertion operation" changes π to

$$a_1 \ldots a_{i-1} a_{i+1} \ldots a_j\, a_i\, a_{j+1} \ldots a_n \qquad \text{or to} \qquad a_1 \ldots a_j\, a_i\, a_{j+1} \ldots a_{i-1}\, a_{i+1} \ldots a_n,$$

for some i and j. Let dis (π) be the minimum number of deletion-insertion operations which will sort π into order. Can dis (π) be expressed in terms of simpler characteristics of π?

40. [*40*] Experiment with the following variation of the diminishing increment sort, intended to speed up the "inner loop": For $s = t, t-1, \ldots, 2$, and for $j = h_s + 1, h_s + 2, \ldots, N$, interchange $R_{j-h_s} \leftrightarrow R_j$ if $K_{j-h_s} > K_j$. (Thus, the result of interchanges is not propagated; no comparison of K_{j-h_s}: K_{j-2h_s} is made. The records will not always be h_s-sorted, but these preliminary passes will help to decrease the number of inversions.) Finally apply straight insertion to complete the sort.

5.2.2. Sorting by Exchanging

We come now to the second family of sorting algorithms mentioned near the beginning of Section 5.2: "exchange" or "transposition" methods which systematically interchange pairs of elements that are out of order until no more such pairs exist.

The process of straight insertion, Algorithm 5.2.1S, can be viewed as an exchange method: We take each new record R_j and essentially exchange it with its neighbors to the left until it has been inserted into the proper place. So the classification of sorting methods into various families such as "insertion," "exchange," "selection," etc., is not always clear-cut. In this section, we shall discuss four types of sorting methods for which exchanging is a dominant characteristic: *exchange selection* (the "bubble sort"); *merge exchange* (Batcher's parallel sort); *partition exchange* (Hoare's "quicksort"); and *radix exchange*.

The bubble sort. Perhaps the most obvious way to sort by exchanges is to compare K_1 with K_2, interchanging R_1 and R_2 if the keys are out of order; then do the same to records R_2 and R_3, R_3 and R_4, etc. During this sequence of operations, records with large keys tend to move to the right, and in fact the record with the largest key will move up to become R_N. Repetitions of the process will get the appropriate records into positions R_{N-1}, R_{N-2}, etc., so that all records will ultimately be sorted.

Figure 14 shows this sorting method in action on the sixteen keys 503 087 512 . . . 703; it is convenient to represent the file of numbers vertically instead of horizontally, with R_N at the top and R_1 at the bottom. The method is called

	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7	Pass 8	Pass 9	
703	908	908	908	908	908	908	908	908	908	908
765	703	897	897	897	897	897	897	897	897	897
677	765	703	765	765	765	765	765	765	765	765
612	677	765	703	703	703	703	703	703	703	703
509	612	677	677	677	677	677	677	677	677	677
154	509	612	653	653	653	653	653	653	653	653
426	154	509	612	612	612	612	612	612	612	612
653	426	154	509	512	512	512	512	512	512	512
275	653	426	154	509	509	509	509	509	509	509
897	275	653	426	154	503	503	503	503	503	503
170	897	275	512	426	154	426	426	426	426	426
908	170	512	275	503	426	154	275	275	275	275
061	512	170	503	275	275	275	154	170	170	170
512	061	503	170	170	170	170	170	154	154	154
087	503	061	087	087	087	087	087	087	087	087
503	087	087	061	061	061	061	061	061	061	061

Fig. 14. The bubble sort in action.

"bubble sorting" because large elements "bubble up" to their proper position, by contrast with the "sinking sort" (i.e., straight insertion) in which elements sink down to an appropriate level. The bubble sort is also known by more prosaic names such as "exchange selection" or "propagation."

After each pass through the file, it is not hard to see that all records above and including the last one to be exchanged must be in their final position, so they need not be examined on subsequent passes. Horizontal lines in Fig. 14 show the progress of the sorting from this standpoint; note, for example, that five more elements are known to be in final position as a result of Pass 4. On the final pass, no exchanges are performed at all. With these observations we are ready to formulate the algorithm.

Algorithm B (*Bubble sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$.

B1. [Initialize BOUND.] Set BOUND $\leftarrow N$. (BOUND is the highest index for which the record is not known to be in its final position; thus we are indicating that nothing is known at this point.)

B2. [Loop on j.] Set $t \leftarrow 0$. Perform step B3 for $j = 1, 2, \ldots,$ BOUND-1, and then go to step B4.

B3. [Compare/exchange R_j: R_{j+1}.] If $K_j > K_{j+1}$, interchange $R_j \leftrightarrow R_{j+1}$ and set $t \leftarrow j$.

B4. [Any exchanges?] If $t = 0$, the algorithm terminates. Otherwise set BOUND $\leftarrow t$ and return to step B2. ▮

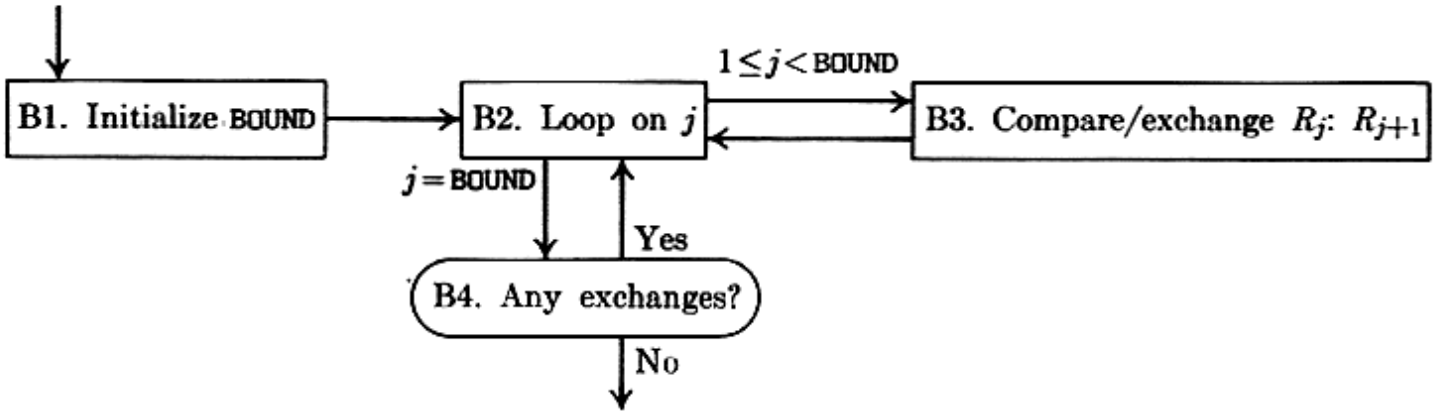


Fig. 15. Flowchart for bubble sorting.

Program B (*Bubble sort*). As in previous MIX programs of this chapter, we assume that the items to be sorted are in locations INPUT+1 through INPUT+N. rI1 $\equiv t$; rI2 $\equiv j$.

```
01  START  ENT1  N            1   B1. Initialize BOUND. t ← N.
02  1H     ST1   BOUND(1:2)   A   BOUND ← t.
03         ENT2  1            A   B2. Loop on j.
04         ENT1  0            A   t ← 0.
05         JMP   BOUND        A   Exit if j ≥ BOUND.
```

```
06  3H      LDA   INPUT,2        C     B3. Compare/exchange R_j:R_{j+1}.
07          CMPA  INPUT+1,2      C
08          JLE   2F             C     No exchange if K_j ≤ K_{j+1}.
09          LDX   INPUT+1,2      B     R_{j+1}
10          STX   INPUT,2        B     → R_j.
11          STA   INPUT+1,2      B     (old R_j) → R_{j+1}.
12          ENT1  0,2            B     t ← j.
13  2H      INC2  1              C     j ← j + 1.
14  BOUND   ENTX  -*,2           A+C   rX ← j − BOUND. (Instruction modified)
15          JXN   3B             A+C   1 ≤ j < BOUND.
16  4H      J1P   1B             A     B4. Any exchanges? To B2 if t > 0. ▮
```

Analysis of the bubble sort. It is quite instructive to analyze the running time of Algorithm B. Three quantities are involved in the timing: the number of passes, A; the number of exchanges, B; and the number of comparisons, C. If the input keys are distinct and in random order, we may assume that they form a random permutation of $1, 2, \ldots, n$. The idea of *inversion tables* (Section 5.1.1) leads to an easy way to describe the effect of each pass in a bubble sort.

Theorem I. *Let $a_1\, a_2 \ldots a_n$ be a permutation of $\{1, 2, \ldots, n\}$, and let $b_1\, b_2 \ldots b_n$ be the corresponding inversion table. If one pass of the bubble sort, Algorithm B, changes $a_1\, a_2 \ldots a_n$ to the permutation $a'_1\, a'_2 \ldots a'_n$, the corresponding inversion table $b'_1\, b'_2 \ldots b'_n$ is obtained from $b_1\, b_2 \ldots b_n$ by decreasing each nonzero entry by* 1.

Proof. If a_i is preceded by a larger element, the largest preceding element is exchanged with it, so b_{a_i} decreases by 1. On the other hand if a_i is not preceded by a larger element, it is never exchanged with a larger element, so b_{a_i} remains 0. ▮

Thus we can see what happens during a bubble sort by studying the sequence of inversion tables betweeen passes. For example, the successive inversion tables corresponding to Fig. 14 are

	3	1	8	3	4	5	0	4	0	3	2	2	3	2	1	0	
Pass 1																	
	2	0	7	2	3	4	0	3	0	2	1	1	2	1	0	0	
Pass 2																	
	1	0	6	1	2	3	0	2	0	1	0	0	1	0	0	0	
Pass 3																	
	0	0	5	0	1	2	0	1	0	0	0	0	0	0	0	0	(1)

and so on. If $b_1\, b_2 \ldots b_n$ is the inversion table of the input permutation, we must therefore have

$$A = 1 + \max(b_1, b_2, \ldots, b_n); \tag{2}$$

$$B = b_1 + b_2 + \cdots + b_n; \tag{3}$$

$$C = c_1 + c_2 + \cdots + c_A, \tag{4}$$

where c_j is the value of BOUND–1 at the beginning of pass j. In terms of the inversion table,

$$c_j = \max \{b_i + i \mid b_i \geq j - 1\} - j \tag{5}$$

(see exercise 5). In example (1) we therefore have $A = 9$, $B = 41$, $C = 15 + 14 + 13 + 12 + 7 + 5 + 4 + 3 + 2 = 75$. The total MIX sorting time for Fig. 14 is $1030u$.

The distribution of B (the total number of inversions in a random permutation) is very well-known to us by now; so we are left with A and C to be analyzed.

The probability that $A \leq k$ is $1/n!$ times the number of inversion tables having no components $\geq k$, namely $k^{n-k}k!$, when $1 \leq k \leq n$. Hence the probability that exactly k passes are required is

$$A_k = \frac{1}{n!}\left(k^{n-k}k! - (k-1)^{n-k+1}(k-1)!\right). \tag{6}$$

The mean value $\sum kA_k$ can therefore be calculated; summing by parts, it is

$$A_{\text{ave}} = n + 1 - \sum_{0 \leq k \leq n} \frac{k^{n-k}k!}{n!} = n + 1 - P(n), \tag{7}$$

where $P(n)$ is the function whose asymptotic value was found to be $\sqrt{\pi n/2} - \frac{2}{3} + O(1/\sqrt{n})$ in Section 1.2.11, Eq. (24). Formula (7) was stated without proof by E. H. Friend in *JACM* **3** (1956), 150; a proof was given by Howard B. Demuth [Ph. D. thesis (Stanford University: October, 1956), 64–68]. For the standard deviation of A, see exercise 7.

The total number of comparisons, C, is somewhat harder to handle, and we will consider only C_{ave}. For fixed n, let $f_j(k)$ be the number of inversion tables $b_1 \ldots b_n$ such that for $1 \leq i \leq n$ we have either $b_i < j - 1$ or $b_i + i - j \leq k$; then

$$f_j(k) = (j + k)!(j - 1)^{n-j-k}, \qquad \text{for} \qquad 0 \leq k \leq n - j. \tag{8}$$

(See exercise 8.) The average value of c_j in (5) is $\left(\sum k(f_j(k) - f_j(k-1))\right)/n!$; summing by parts and then summing on j leads to the formula

$$C_{\text{ave}} = \binom{n+1}{2} - \frac{1}{n!} \sum_{\substack{1 \leq j \leq n \\ 0 \leq k \leq n-j}} f_j(k) = \binom{n+1}{2} - \frac{1}{n!} \sum_{0 \leq r < s \leq n} s!r^{n-s}. \tag{9}$$

Here the asymptotic value is not easy to determine, and we shall return to it at the end of this section.

To summarize our analysis of the bubble sort, the formulas derived above and below may be written as follows:

$$A = (\min 1, \quad \text{ave } N - \sqrt{\pi N/2} + O(1), \quad \max N); \tag{10}$$

$$B = (\min 0, \quad \text{ave } \tfrac{1}{4}(N^2 - N), \quad \max \tfrac{1}{2}(N^2 - N)); \tag{11}$$

$$C = (\min N - 1, \quad \text{ave } \tfrac{1}{2}(N^2 - N \ln N - (\gamma + \ln 2 - 1)N) + O(\sqrt{N}), \quad \max \tfrac{1}{2}(N^2 - N)). \tag{12}$$

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 MIX running time is $8A + 7B + 8C + 1 = (\min 8N + 1, \text{ave } 5.75N^2 + O(N \ln N), \max 7.5N^2 + 0.5N + 1)$.

Refinements of the bubble sort. It took a good deal of work to analyze the bubble sort; and although the techniques used in the calculations are instructive, the results are disappointing since they tell us that the bubble sort isn't really very good at all. Compared to straight insertion (Algorithm 5.2.1S), bubble sorting requires a more complicated program and takes about twice as long!

Some suggestions can be given for improving the bubble sort. For example, in Fig. 14, the first comparison in Pass 4 is redundant, as are the first two in Pass 5 and the first three in Passes 6 and 7. Note also that elements can never move to the left more than one step per pass; so if the smallest item happens to be initially at the far right we are forced to make the maximum number of comparisons. This suggests the "cocktail shaker sort," in which alternate passes go in opposite directions (see Fig. 16). The average number of comparisons is slightly reduced by this approach. K. E. Iverson [*A Programming Language* (Wiley, 1962), 218–219] has made an interesting observation in this regard: If j is an index such that R_j and R_{j+1} are not exchanged with each

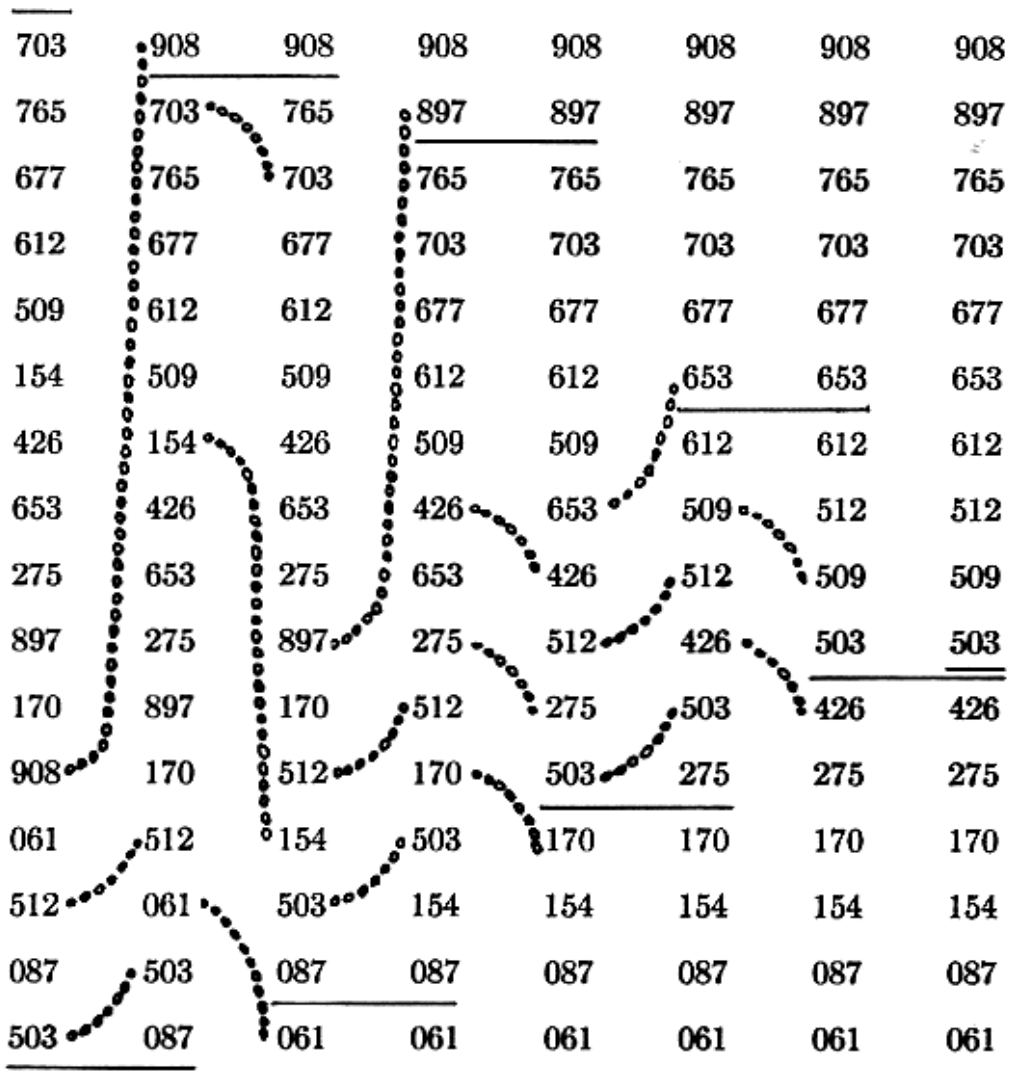


Fig. 16. The "cocktail-shaker sort."

other on two consecutive passes in opposite directions, then R_j and R_{j+1} must be in their final position, and they need not enter into any subsequent comparisons. For example, traversing 4 3 2 1 8 6 9 7 5 from left to right yields 3 2 1 4 6 8 7 5 9; no interchange occurred between R_4 and R_5. Traversing the latter permutation from right to left, R_4 is still less than (the new) R_5, so we may immediately conclude that R_4 and R_5 need not participate in any further comparisons.

But none of these refinements lead to an algorithm better than straight insertion (and we already know that straight insertion isn't suitable for large N). Another idea is to eliminate most of the exchanges; since most elements simply shift left one step during an exchange, we could achieve the same effect by viewing the array differently, shifting the origin of indexing! But the resulting algorithm is no better than straight *selection*, Algorithm 5.2.3S, which we shall study later.

In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

Batcher's parallel method. If we are going to have an exchange algorithm whose running time is faster than order N^2, we need to select some *nonadjacent* pairs of keys (K_i, K_j) for comparisons; otherwise we will need as many exchanges as the original permutation has inversions, and the average number of inversions is $\frac{1}{4}(N^2 - N)$. An ingenious way to program a sequence of comparisons, looking for potential exchanges, was discovered in 1964 by K. E. Batcher [*Proc. AFIPS Spring Joint Computer Conference* **32** (1968), 307–314]. His method is not at all obvious; in fact, it requires a fairly intricate proof just to show that it is valid, since comparatively few comparisons are made. We shall discuss two proofs, one in this section and another in Section 5.3.4.

Batcher's sorting scheme is somewhat like Shell's sort, but the comparisons are done in a novel way so that no propagation of exchanges is necessary. Thus we can compare Table 1 to Table 5.2.1–3; Batcher's method achieves the

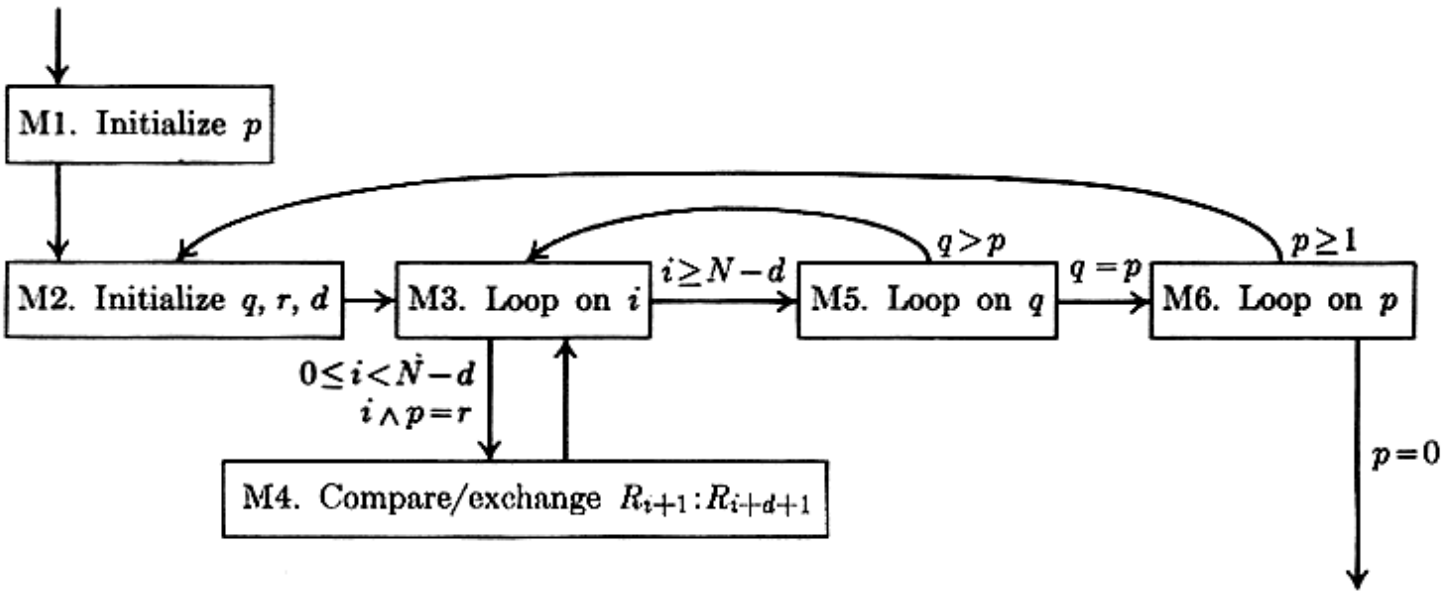


Fig 17. Algorithm M.

effect of 8-sorting, 4-sorting, 2-sorting, and 1-sorting, but the comparisons do not overlap. Since Batcher's algorithm essentially merges pairs of sorted subsequences, it may be called the "merge exchange sort."

Algorithm M (*Merge exchange*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$. We assume that $N \ge 2$.

M1. [Initialize p.] Set $p \leftarrow 2^{t-1}$, where $t = \lceil \log_2 N \rceil$ is the least integer such that $2^t \ge N$. (Steps M2 through M5 will be performed for $p = 2^{t-1}, 2^{t-2}, \ldots, 1$.)

M2. [Initialize q, r, d.] Set $q \leftarrow 2^{t-1}$, $r \leftarrow 0$, $d \leftarrow p$.

M3. [Loop on i.] For all i such that $0 \le i < N - d$ and $i \wedge p = r$, do step M4. Then go to step M5. (Here $i \wedge p$ means the "logical and" of the binary representations of i and p; each bit of the result is zero except where both i and p have 1 bits in corresponding positions. Thus $13 \wedge 21 = (1101)_2 \wedge (10101)_2 = (00101)_2 = 5$. At this point, d is an odd multiple of p, and p is a power of 2, so that $i \wedge p \ne (i + d) \wedge p$; it follows that the actions of step M4 can be done for all relevant i in any order, even simultaneously.)

M4. [Compare/exchange $R_{i+1}: R_{i+d+1}$.] If $K_{i+1} > K_{i+d+1}$, interchange $R_{i+1} \leftrightarrow R_{i+d+1}$.

M5. [Loop on q.] If $q \ne p$, set $d \leftarrow q - p$, $q \leftarrow q/2$, $r \leftarrow p$, and return to M3.

M6. [Loop on p.] (At this point the permutation $K_1\, K_2 \ldots K_N$ is p-ordered.) Set $p \leftarrow \lfloor p/2 \rfloor$. If $p > 0$, go back to M2. ∎

Table 1 illustrates the method for $N = 16$. Note that the algorithm sorts N elements essentially by sorting $R_1, R_3, R_5, \ldots$ and $R_2, R_4, R_6, \ldots$ independently; then we perform steps M2 through M5 for $p = 1$, in order to merge the two sorted sequences together.

In order to prove that the magic sequence of comparison/exchanges specified in Algorithm M actually will sort all possible input files $R_1\, R_2 \ldots R_N$, we must show only that steps M2 through M5 will merge all 2-ordered files $R_1\, R_2 \ldots R_N$ when $p = 1$. For this purpose we can use the lattice-path method of Section 5.2.1 (cf. Fig. 11); each 2-ordered permutation of $\{1, 2, \ldots, N\}$ corresponds uniquely to a path from $(0, 0)$ to $(\lceil N/2 \rceil, \lfloor N/2 \rfloor)$ in a lattice diagram. Figure 18(a) shows an example for $N = 16$, corresponding to the permutation 1 3 2 4 10 5 11 6 13 7 14 8 15 9 16 12. When we perform step M3 with $p = 1$, $q = 2^{t-1}$, $r = 0$, $d = 1$, the effect is to compare (and possibly exchange) $R_1 : R_2$, $R_3 : R_4$, etc. This operation corresponds to a simple transformation of the lattice path, "folding" it about the diagonal if necessary so that it never goes above the diagonal. (See Fig. 18(b) and the proof in exercise 10.) The next iterations of step M3 have $p = r = 1$, and $d = 2^{t-1} - 1, 2^{t-2} - 1, \ldots, 1$; their effect is to compare/exchange $R_2 : R_{2+d}$, $R_4 : R_{4+d}$, etc., and again there is a simple lattice interpretation: the path is "folded" about a line $\frac{1}{2}(d + 1)$ units below the diagonal. See Fig. 18(c) and (d); eventually we get to the path

in Fig. 18(e), which corresponds to a completely sorted permutation. This completes a "geometric proof" that Batcher's algorithm is valid; we might call it sorting by folding!

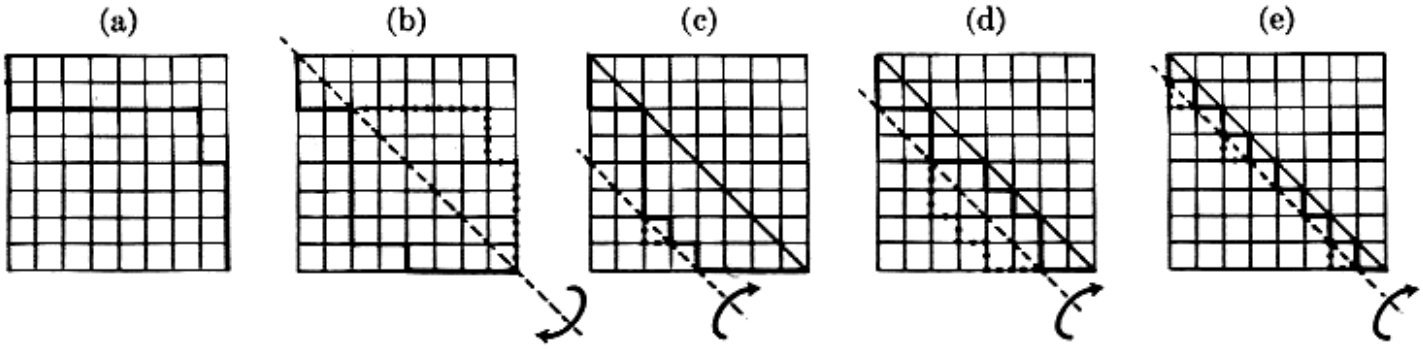


Fig. 18. A geometric interpretation of Batcher's method, $N = 16$.

Table 1

MERGE-EXCHANGE SORTING (BATCHER'S METHOD)

p	q	r	d	
				503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703
8	8	0	8	
				503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703
4	8	0	4	
				503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703
4	4	4	4	
				503 087 154 061 612 170 512 275 653 426 765 509 908 677 897 703
2	8	0	2	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
2	4	2	6	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
2	2	2	2	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
1	8	0	1	
				061 154 087 503 170 512 275 612 426 653 509 765 677 897 703 908
1	4	1	7	
				061 154 087 503 170 512 275 612 426 653 509 765 677 897 703 908
1	2	1	3	
				061 154 087 275 170 426 503 509 512 653 612 703 677 897 765 908
1	1	1	1	
				061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

A MIX program for Algorithm M appears in exercise 12. Unfortunately the amount of bookkeeping needed to control the sequence of comparisons is rather large, so the program is less efficient than other methods we have seen. But it has one important redeeming feature: All comparison/exchanges specified by a given iteration of step M3 can be done *simultaneously*, on computers or logical networks which allow parallel computations. With such parallel operations, sorting is completed in $\frac{1}{2}\lceil \log_2 N \rceil(\lceil \log_2 N \rceil + 1)$ steps, and this is about as fast as any general method known. For example, *1024 elements can be sorted in only 55 parallel steps by Batcher's method.* The nearest competitor is Pratt's method (see exercise 5.2.1–30), which uses either 40 or 73 steps, depending on how we count; if we are willing to allow overlapping comparisons as long as no overlapping exchanges are necessary, Pratt's method requires only 40 comparison/exchange cycles to sort 1024 elements. For further comments, see Section 5.3.4.

"Quicksort." The sequence of comparisons in Batcher's method is predetermined; we compare the same pairs of keys each time, regardless of what we may have learned about the file from previous comparisons. The same is largely true of the bubble sort, although Algorithm B does make limited use of previous knowledge in order to reduce its work at the right end of the file. Let us now turn to a quite different strategy, which uses the result of each comparison to determine what keys are to be compared next. Such a strategy is inappropriate for parallel computations, but on computers which work serially it can be quite fruitful.

Consider therefore the following comparison/exchange scheme: Keep two pointers, i and j, with $i = 1$ and $j = N$ initially. Compare $K_i{:}K_j$, and if no exchange is necessary decrease j by 1 and repeat the process. After an exchange first occurs, increase i by 1, and continue comparing and increasing i until another exchange occurs. Then decrease j again, and so on, "burning the candle at both ends," until $i = j$. For example, consider what happens to our file of sixteen numbers.

Given:	**503**	087	512	061	908	170	897	275	653	426	154	509	612	677	765	**703**	
																	Decrease j
1st exchange:	**154**	087	512	061	908	170	897	275	653	426	**503**	509	612	677	765	703	
																	Increase i
2nd exchange:	154	087	**503**	061	908	170	897	275	653	426	**512**	509	612	677	765	703	
																	Decrease j
3rd exchange:	154	087	**426**	061	908	170	897	275	653	**503**	512	509	612	677	765	703	
																	Increase i
4th exchange:	154	087	426	061	**503**	170	897	275	653	**908**	512	509	612	677	765	703	
																	Decrease j
5th exchange:	154	087	426	061	**275**	170	897	**503**	653	908	512	509	612	677	765	703	
																	Increase i
6th exchange:	154	087	426	061	275	170	**503**	**897**	653	908	512	509	612	677	765	703	
																	Decrease j

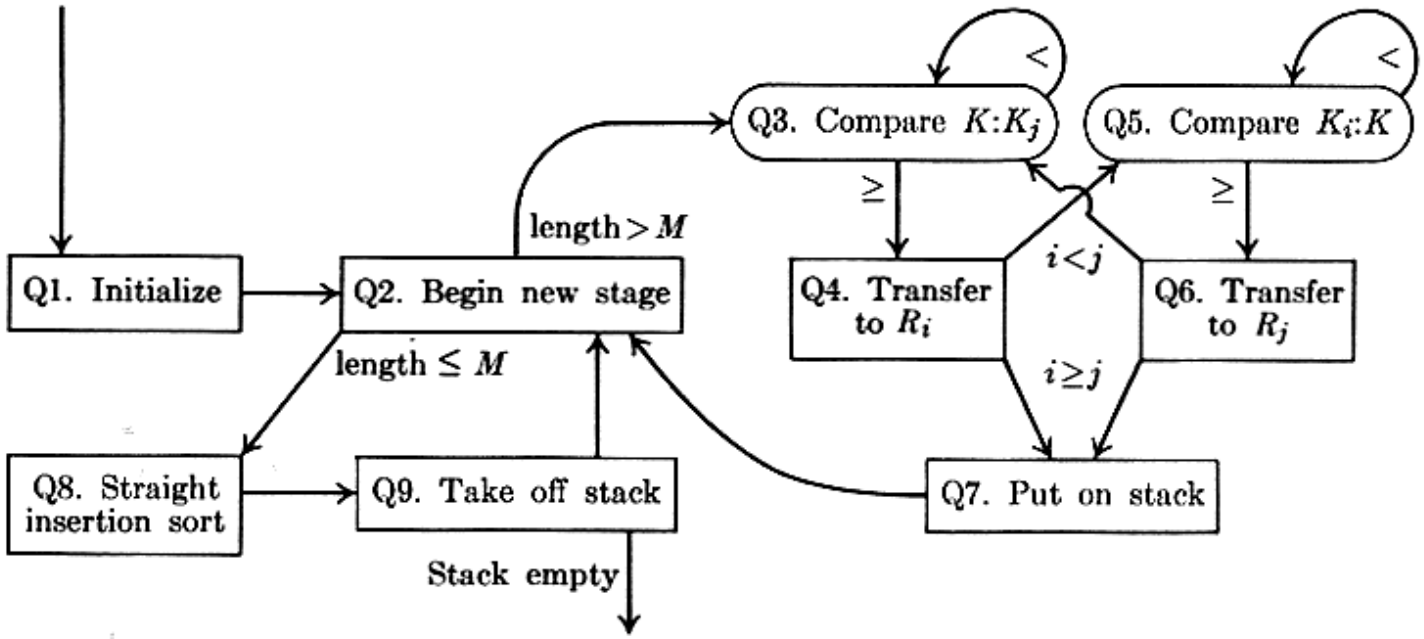


Fig. 19. Partition-exchange sorting ("quicksort").

(In order to indicate the position of i and j, K_i and K_j have been shown in boldface type.) Note that each of the comparisons in this example involves the key 503; in general, each comparison will involve the original value of K_1, because it keeps getting exchanged every time we switch directions. By the time that $i = j$, the original record R_1 will have moved into its final position, since it is easy to see that there will be no greater keys to its left and no smaller keys to its right. The original file will have been partitioned in such a way that the original problem is reduced to two simpler problems, sorting $R_1 \ldots R_{i-1}$ and (independently) sorting $R_{i+1} \ldots R_N$. We can apply the same technique to each of these subfiles.

Table 2 shows how our example file gets completely sorted by this approach, in 11 stages. Brackets indicate subfiles which still need to be sorted; inside a computer, these subfiles can be represented by two variables l and r (the

Table 2

"QUICKSORTING"

																(l, r)	Stack
[503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703]	(1, 16)	—
[154	087	426	061	275	170]	503	[897	653	908	512	509	612	677	765	703]	(1, 6)	(8, 16)
[061	087]	154	[426	275	170]	503	[897	653	908	512	509	612	677	765	703]	(1, 2)	(4, 6)(8, 16)
061	087	154	[426	275	170]	503	[897	653	908	512	509	612	677	765	703]	(4, 6)	(8, 16)
061	087	154	[170	275]	426	503	[897	653	908	512	509	612	677	765	703]	(4, 5)	(8, 16)
061	087	154	170	275	426	503	[897	653	908	512	509	612	677	765	703]	(8, 16)	—
061	087	154	170	275	426	503	[703	653	765	512	509	612	677]	897	908	(8, 14)	—
061	087	154	170	275	426	503	[677	653	612	512	509]	703	765	897	908	(8, 12)	—
061	087	154	170	275	426	503	[509	653	612	512]	677	703	765	897	908	(8, 11)	—
061	087	154	170	275	426	503	509	[653	612	512]	677	703	765	897	908	(9, 11)	—
061	087	154	170	275	426	503	509	[512	612]	653	677	703	765	897	908	(9, 10)	—
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908	—	—

boundaries of the subfile currently under examination) and a stack of additional pairs (l_k, r_k). Each time the file is subdivided, we put the *largest* subfile on the stack and commence work on the other one, until we reach trivially short files; this procedure assures that the stack will never contain more than about $\log_2 N$ entries, as shown in exercise 20.

The sorting procedure just described may be called *partition-exchange sorting;* it is due to C. A. R. Hoare, whose interesting paper [*Comp. J.* **5** (1962), 10–15] contains one of the most comprehensive accounts of a sorting method that has ever been published. Hoare dubbed his method "quicksort"; such a name is not inappropriate, since, for example, the entire sorting process in Table 2 requires only 48 comparisons (the least we have seen so far, except for binary insertion which takes 47). All comparisons in a given stage are made against the same key, so this key may be kept in a register. Furthermore, the amount of data movement is quite reasonable; the computation in Table 2 makes only 17 exchanges, most of which are simply "half-exchanges" (simple transfers) because one key stays in the register and it need not be stored until the end of a stage.

The extra bookkeeping (required to control i, j, and the stack) is not difficult, but it makes the quicksort partitioning procedure most suitable for large N; therefore it is desirable to sort short subfiles in a special manner as in the following algorithm.

Algorithm Q (*Partition-exchange sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$. An auxiliary stack with at most $\log_2 N$ entries is needed for temporary storage. This algorithm follows the "quicksort" partitioning procedure described in the text above, with slight modifications for extra efficiency:

a) We assume the presence of artificial keys $K_0 = -\infty$ and $K_{N+1} = +\infty$ such that

$$K_0 \le K_i \le K_{N+1} \qquad \text{for} \qquad 1 \le i \le N. \tag{13}$$

(Equality is allowed.)

b) Subfiles of M or fewer elements are sorted by straight insertion, where $M \ge 1$ is a parameter which should be chosen as described in the text below.

c) One or two extra comparisons are made during particular stages (allowing the pointers i, j to cross), so that the main comparison loops can be as fast as possible.

d) Records with equal keys are exchanged, although it is not strictly necessary to do so. (This idea, due to R. C. Singleton, helps to split subfiles nearly in half when equal elements are present; see exercise 18.)

Q1. [Initialize.] Set the stack empty, and set $l \leftarrow 1$, $r \leftarrow N$.

Q2. [Begin new stage.] (We now wish to sort the subfile $R_l \ldots R_r$; from the nature of the algorithm, we have $r \geq l - 1$, and $K_{l-1} \leq K_i \leq K_{r+1}$ for $l \leq i \leq r$.) If $r - l < M$, go to step Q8. Otherwise set $i \leftarrow l$, $j \leftarrow r$, $K \leftarrow K_l$, $R \leftarrow R_l$. (For a discussion of better choices for R and K, see the text below.)

Q3. [Compare $K{:}K_j$.] If $K < K_j$, decrease j by 1 and repeat this step.

Q4. [Transfer to R_i.] (At this point K_i is an irrelevant key $\geq K$, and $K \geq K_j$.) If $j \leq i$, set $R_i \leftarrow R$ and go to Q7. Otherwise set $R_i \leftarrow R_j$ and increase i by 1.

Q5. [Compare $K_i{:}K$.] If $K_i < K$, increase i by 1 and repeat this step.

Q6. [Transfer to R_j.] (At this point K_j is an irrelevant key $\leq K$, and $K \leq K_i$.] If $j \leq i$, set $R_j \leftarrow R$ and $i \leftarrow j$. Otherwise set $R_j \leftarrow R_i$, decrease j by 1, and go to Q3.

Q7. [Put on stack.] (Now the subfile $R_l \ldots R_i \ldots R_r$ has been partitioned so that $K_k \leq K_i$ for $l \leq k \leq i$ and $K_i \leq K_k$ for $i \leq k \leq r$.) If $r - i \geq i - l$, insert $(i + 1, r)$ on top of the stack and set $r \leftarrow i - 1$. Otherwise insert $(l, i - 1)$ on top of the stack and set $l \leftarrow i + 1$. (Each entry (a, b) on the stack is a request to sort the subfile $R_a \ldots R_b$ at some future time.) Now go back to step Q2.

Q8. [Straight insertion sort.] For $j = l + 1, l + 2, \ldots$ until $j > r$ do the following operations: Set $K \leftarrow K_j$, $R \leftarrow R_j$, $i \leftarrow j - 1$; then set $R_{i+1} \leftarrow R_i$, $i \leftarrow i - 1$ zero or more times until $K_i \leq K$; then set $R_{i+1} \leftarrow R$. (This is essentially Algorithm 5.2.1S, applied to a subfile of M or fewer elements.)

Q9. [Take off stack.] If the stack is empty, we are done sorting; otherwise remove its top entry (l', r'), set $l \leftarrow l'$, $r \leftarrow r'$, and return to step Q2. ▮

The corresponding MIX program is rather long, but not complicated; in fact, a large part of the coding is devoted to step Q7, which just fools around with the variables in a very straightforward way.

Program Q (*Partition-exchange sort*). Records to be sorted appear in locations INPUT+1 through INPUT+N; assume that locations INPUT and INPUT+N+1 contain, respectively, the smallest and largest values possible in MIX. The stack is kept in locations STACK+1, STACK+2, . . . ; see exercise 20 for the exact number of locations to set aside for the stack. rI1 $\equiv l$, rI2 $\equiv r$, rI3 $\equiv i$, rI4 $\equiv j$, rI6 $\equiv$ size of stack, rA $\equiv K \equiv R$.

	A	EQU	2:3		First component of stack entry
	B	EQU	4:5		Second component of stack entry
01	START	ENT1	1	1	*Q1. Initialize.* $l \leftarrow 1$.
02		ENT2	N	1	$r \leftarrow N$.
03		ENT6	0	1	Set stack empty.
04	2H	ENTX	0,2	$2A+1$	*Q2. Begin new stage.*
05		DECX	M,1	$2A+1$	$rX \leftarrow r - l - M$.
06		JXN	8F	$2A+1$	To Q8 if subfile size $\leq M$.
07		ENT3	0,1	A	$i \leftarrow l$.
08		ENT4	0,2	A	$j \leftarrow r$.
09		LDA	INPUT,3	A	$K \leftarrow K_i$.
10		JMP	3F	A	To Q3.
11	0H	LDX	INPUT,3	B	
12		STX	INPUT,4	B	$R_j \leftarrow R_i$.
13		DEC4	1	$C' - A$	$j \leftarrow j - 1$.
14	3H	CMPA	INPUT,4	C'	*Q3. Compare* $K:K_j$.
15		JL	*-2	C'	If $<$, decrease j and repeat.
16	4H	ENTX	0,3	$B+A$	*Q4. Transfer to* R_i.
17		DECX	0,4	$B+A$	
18		JXNN	7F	$B+A$	To Q7 if $i \geq j$.
19		LDX	INPUT,4	$B+X$	
20		STX	INPUT,3	$B+X$	$R_i \leftarrow R_j$.
21		INC3	1	C''	$i \leftarrow i+1$.
22	5H	CMPA	INPUT,3	C''	*Q5. Compare* $K_i:K$.
23		JG	*-2	C''	If $<$, increase i and repeat.
24	6H	ENTX	0,3	$B+X$	*Q6. Transfer to* R_j.
25		DECX	0,4	$B+X$	
26		JXN	0B	$B+X$	To Q3 if $i < j$.
27		ENT3	0,4	X	$i \leftarrow j$.
28	7H	STA	INPUT,3	A	*Q7. Put on stack.* $R_i \leftarrow R$.
29		ENTA	0,2	A	
30		DECA	0,3	A	
31		DECA	0,3	A	
32		INCA	0,1	A	
33		INC6	1	A	rI6 $\leftarrow$ rI6 $+ 1$.
34		JANN	1F	A	Jump if $r - i \geq i - l$.
35		ST1	STACK,6(A)	A'	
36		DEC3	1	A'	
37		ST3	STACK,6(B)	A'	$(l, i-1) \Rightarrow$ stack.
38		ENT1	2,3	A'	$l \leftarrow i+1$.
39		JMP	2B	A'	To Q2.
40	1H	INC3	1	$A - A'$	
41		ST3	STACK,6(A)	$A - A'$	
42		ST2	STACK,6(B)	$A - A'$	$(i+1, r) \Rightarrow$ stack.
43		ENT2	-2,3	$A - A'$	$r \leftarrow i - 1$.
44		JMP	2B	$A - A'$	To Q2.
45	8H	DEC2	0,1	$A+1$	*Q8. Straight insertion sort.*
46		J2NP	9F	$A+1$	To Q9 if $r \leq l$.
47		ENT4	1,1	$A+1-L$	$j \leftarrow l+1$.

48	1H	LDA	INPUT,4	D	$R \leftarrow R_j$.
49		ENT3	-1,4	D	$i \leftarrow j - 1$.
50		JMP	*+4	D	Jump into the loop.
51		LDX	INPUT,3	E	
52		STX	INPUT+1,3	E	$R_{i+1} \leftarrow R_i$.
53		DEC3	1	E	$i \leftarrow i - 1$.
54		CMPA	INPUT,3	$D + E$	
55		JL	*-4	$D + E$	Jump if $K < K_i$.
56		STA	INPUT+1,3	D	$R_{i+1} \leftarrow R$.
57		INC4	1	D	$j \leftarrow j + 1$.
58		DEC2	1	D	
59		J2P	1B	D	Repeat $r - l$ times.
60	9H	LD1	STACK,6(A)	$A + 1$	*Q9. Take off stack.*
61		LD2	STACK,6(B)	$A + 1$	$(l, r) \Leftarrow$ stack.
62		DEC6	1	$A + 1$	rI6 $\leftarrow$ rI6 $- 1$.
63		J6NN	2B	$A + 1$	To Q2 if stack wasn't empty. ▮

Analysis of "quicksort." The timing information shown with Program Q is not hard to derive using Kirchhoff's conservation law (cf. Section 1.3.3) and the fact that everything put on the stack is eventually removed again. The total running time depends on the following quantities:

$$\begin{aligned}
A &= \text{number of stages in which the given subfile has more than } M \text{ elements;}\\
B &= \text{number of times } R_j \leftarrow R_i \text{ in step Q6;}\\
C' &= \text{number of times step Q3 is performed;}\\
C'' &= \text{number of times step Q5 is performed;}\\
D &= \text{number of times } R \leftarrow R_j \text{ in step Q8;}\\
E &= \text{number of times } R_{i+1} \leftarrow R_i \text{ in step Q8;}\\
L &= \text{number of times } r \le l \text{ in step Q8;}\\
X &= \text{number of times } i \leftarrow j \text{ in step Q6.}
\end{aligned} \tag{14}$$

Instead of analyzing C' and C'' independently, we shall consider only

$$C = C' + C'' = \text{total number of comparisons,} \tag{15}$$

since the running time for C' and C'' is identical. By analyzing these seven quantities A, B, C, D, E, L, and X, we will be able to make an intelligent choice of the parameter M which specifies the "threshold" between straight insertion sorting and partitioning. (Nonmathematical readers, please skip to Eq. (25).)

As in most other analyses of this chapter, we shall assume that the keys to be sorted are distinct; exercise 18 indicates that equalities between keys do not seriously harm the efficiency of Algorithm Q, and in fact they seem to help it. Since the method depends only on the relative order of the keys, we may as well assume that they are simply $\{1, 2, \ldots, N\}$ in some order.

When we first get to step Q2, the sorting reduces to straight insertion if $N \le M$, and we have already analyzed straight insertion; so let us assume that $N > M$. We need to study only the computations which take us to step Q7

for the first time; once the partitioning has been achieved, it is easy to see that both subfiles $R_1 \ldots R_{i-1}$ and $R_{i+1} \ldots R_N$ are in random order if the original file was in random order, so the contribution of subsequent computations can be determined by induction on N.

Let s be the value of the first key, K_1, and assume that exactly t of the keys $K_1, \ldots, K_s$ are greater than s. Let

$$h = \begin{cases} 1, & \text{if} \quad K_s < s; \\ 0, & \text{if} \quad K_s \geq s. \end{cases} \tag{16}$$

In other words, h is 1 if the key which initially occupies position s (where K_1 will be placed) will be moved to the left. Remember that the keys being sorted are the integers $\{1, 2, \ldots, N\}$.

If $s = 1$, it is easy to see what happens during the first stage of partitioning: Step Q3 is performed N times, and then step Q4 takes us to Q7. So the contributions of the first stage in this case are $A = 1$, $B = 0$, $C = N$, $X = 0$. A similar but slightly more complicated argument when $s > 1$ (see exercise 21) shows that the contributions of the first stage to the total running time are, in general,

$$A = 1, B = t, C = N + 1 - \delta_{s1}, X = h, \qquad \text{for} \qquad 1 \leq s \leq N. \tag{17}$$

To this we must add the contributions of the later stages, which sort subfiles of $s - 1$ and $N - s$ elements, respectively.

If we assume that the original file is in random order, it is now possible to write down formulas which define the generating functions for the probability distributions of $A, B, \ldots, X$ (see exercise 22). But for simplicity we shall consider here only the *average* values of these quantities, $A_N, B_N, \ldots, X_N$, as functions of N. Consider, for example, the average number of comparisons, C_N, which occur during the partitioning process. When $N \leq M$, $C_N = 0$. Since any given value of s occurs with probability $1/N$, we have

$$\begin{aligned} C_N &= \frac{1}{N} \sum_{1 \leq s \leq N} (N + 1 - \delta_{s1} + C_{s-1} + C_{N-s}) \\ &= N + 1 - \frac{1}{N} + \frac{2}{N} \sum_{0 \leq k < N} C_k, \qquad \text{for} \qquad N > M. \end{aligned} \tag{18}$$

Similar formulas hold for the other quantities $A_N, B_N, \ldots, X_N$ (see exercise 23).

There is a simple way to solve recurrence relations of the form

$$x_n = f_n + \frac{2}{n} \sum_{0 \leq k < n} x_k, \qquad \text{for} \qquad n \geq m. \tag{19}$$

The first step is to get rid of the summation sign: Since

$$(n+1)x_{n+1} = (n+1)f_{n+1} + 2 \sum_{0 \le k \le n} x_k,$$

$$nx_n = nf_n + 2 \sum_{0 \le k < n} x_k,$$

we may subtract, obtaining

$$(n+1)x_{n+1} - nx_n = g_n + 2x_n, \qquad \text{where} \qquad g_n = (n+1)f_{n+1} - nf_n.$$

Now the recurrence takes the much simpler form

$$(n+1)x_{n+1} = (n+2)x_n + g_n, \qquad \text{for} \qquad n \ge m. \tag{20}$$

Any recurrence relation which has the general form

$$a_n x_{n+1} = b_n x_n + g_n \tag{21}$$

can be reduced to a summation if we multiply both sides by the "summing factor" $a_0\, a_1 \ldots a_{n-1}/b_0\, b_1 \ldots b_n$; we obtain

$$y_{n+1} = y_n + c_n, \quad \text{where} \quad y_n = \frac{a_0 \ldots a_{n-1}}{b_0 \ldots b_{n-1}} x_n, \quad c_n = \frac{a_0 \ldots a_{n-1}}{b_0\, b_1 \ldots b_n} g_n. \tag{22}$$

In our case (20), the summing factor is simply $n!/(n+2)! = 1/(n+1)(n+2)$, so we find that the simple relation

$$\frac{x_{n+1}}{n+2} = \frac{x_n}{n+1} + \frac{(n+1)f_{n+1} - nf_n}{(n+1)(n+2)}, \qquad n \ge m, \tag{23}$$

is a consequence of (19).

For example, if we set $f_n = 1/n$, we get the unexpected result $x_n/(n+1) = x_m/(m+1)$ for all $n \ge m$. If we set $f_n = n+1$, we get

$$\begin{aligned} x_n/(n+1) &= 2/(n+1) + 2/n + \cdots + 2/(m+2) + x_m/(m+1) \\ &= 2(H_{n+1} - H_{m+1}) + x_m/(m+1), \end{aligned}$$

for all $n \ge m$. Combining these two solutions, and setting $m = M+1$ and $x_n = 0$ for $n \le M$, gives the formula

$$\begin{aligned} C_N &= (N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{1}{(M+1)(M+2)}\right) \\ &\approx 2(N+1)\ln\left(\frac{N+1}{M+2}\right), \qquad \text{for} \qquad N > M. \end{aligned} \tag{24}$$

In Section 6.2.2, we shall prove that the standard deviation of C_N is asymptotically $\sqrt{(21 - 2\pi^2)/3}\, N$; this is reasonably small compared to (24).

The other quantities can be found in a similar way (see exercise 23); we have

$$\begin{aligned}
A_N &= 2(N+1)/(M+2) - 1,\\
B_N &= \frac{1}{6}(N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2}\right) + \frac{1}{2},\\
D_N &= (N+1)M(M-1)/(M+2)(M+1),\\
E_N &= \tfrac{1}{6}(N+1)M(M-1)/(M+2),\\
L_N &= 4(N+1)/(M+2)(M+1),\\
X_N &= (N+1)/(M+2) - \tfrac{1}{2}, \qquad \text{for} \qquad N > M.
\end{aligned} \tag{25}$$

The above discussion shows that it is possible to carry out an exact analysis of the average running time of a fairly complex program, by using techniques that we have previously applied only to simpler cases.

Formulas (24) and (25) can be used to determine the "best" value of M on a particular computer. In MIX's case, Program Q requires $37A + 14B + 4C + 12D + 8E - L + 8X + 15$ units of time; this comes to $\frac{1}{3}\left(38(N+1)H_N + (N+1)f(M)\right) - 19$ units on the average, for $N > M$, where

$$f(M) = 4M - 38H_{M+2} + 43 + \frac{84}{M+2} + \frac{48}{(M+2)(M+1)}. \tag{26}$$

We want to choose M so that $f(M)$ is a minimum. In this case

$$f(M) - f(M-1) = 4 - \frac{38}{M+2} - \frac{84}{(M+2)(M+1)} - \frac{96}{(M+2)(M+1)M},$$

and we want to find M such that $f(M) - f(M-1) \le 0$, $f(M+1) - f(M) \ge 0$; the solution, $M = 9$, is easily found. When $M = 9$ the average running time of Program Q is approximately $12.67(N+1)\ln N - 1.92N - 14.59$ for large N.

So Program Q is quite fast, on the average, considering that it requires very little memory space. But what is the *worst* case of Algorithm Q? Are there some inputs which it does not handle efficiently? The answer to this question is quite embarrassing: If the original file is already in order, with $K_1 < K_2 < \cdots < K_N$, each "partitioning" operation is almost useless, since it reduces the size of the subfile by only one element! So this situation (which ought to be easiest of all to sort) makes quicksort anything but quick; its running time becomes proportional to N^2 instead of $N \log N$. (See exercise 25.) Unlike the other sorting methods we have seen, Algorithm Q likes a disordered file!

Hoare suggested two ways to remedy the situation, in his original paper, by choosing a better value of the test key K which governs the partitioning. One of his recommendations is to choose a *random* integer q between l and r in the last part of step Q2; we can change the instructions "$K \leftarrow K_l$, $R \leftarrow R_l$" to

$$K \leftarrow K_q, \qquad R \leftarrow R_q, \qquad R_q \leftarrow R_l \tag{27}$$

in that step. According to Eqs. (25), such random integers need to be calculated only $2(N+1)/(M+2) - 1$ times on the average, so the additional running

time is not substantial; and the random choice gives good protection against the occurrence of the worst case.

Hoare's second suggestion is to look at a small sample of the file and to choose a median value of the sample. This approach has been followed by R. C. Singleton [*CACM* **12** (1969), 185–187], who suggests letting K_q be the median of the three values

$$K_l, \qquad K_{\lfloor (l+r)/2 \rfloor}, \qquad K_r. \tag{28}$$

Singleton's procedure cuts the number of comparisons down from $2N \ln N$ to about $\frac{12}{7} N \ln N$ (see exercise 29). It can be shown that B_N is asymptotically $C_N/5$ instead of $C_N/6$ in this case, so the median method slightly increases the amount of time spent in transferring the data; the total running time therefore decreases by roughly 8 percent. (See exercise 56 for a detailed analysis.) The worst case is still of order N^2, but such slow behavior will hardly ever occur.

W. D. Frazer and A. C. McKellar [*JACM* **17** (1970), 496–507] have suggested taking a much larger sample consisting of $2^k - 1$ records, where k is chosen so that $2^k \approx N/\ln N$. The sample can be sorted by the usual quicksort method, then inserted among the remaining records by taking k passes over the file (partitioning it into 2^k subfiles, bounded by the elements of the sample). Finally the subfiles are sorted. The average number of comparisons required by such a "samplesort" procedure is about the same as in Singleton's median method, when N is in a practical range, but it decreases to the asymptotic value $N \log_2 N$ as $N \to \infty$.

Radix exchange. We come now to a method which is quite different from any of the sorting schemes we have seen before; it makes use of the *binary representation* of the keys, so it is intended only for binary computers. Instead of comparing two keys with each other, this method inspects individual bits of the keys, to see if they are 0 or 1. In other respects it has the characteristics of exchange sorting, and, in fact, it is rather similar to quicksort. Since it depends on radix 2 representations, we call it "radix exchange sorting." The algorithm can be described roughly as follows:

1) Sort the sequence on its *most significant binary bit*, so that all keys which have a leading 0 come before all keys which have a leading 1. This sorting is done by finding the leftmost key K_i which has a leading 1, and the rightmost key K_j with a leading 0. Then R_i and R_j are exchanged and the process is repeated until $i > j$.

2) Let F_0 be the elements with leading bit 0, and let F_1 be the others. Apply the radix exchange sorting method to F_0 (starting now at the *second* bit from the left instead of the most significant bit), until F_0 is completely sorted; then do the same for F_1.

For example, Table 3 shows how the radix exchange sort acts on our 16 random numbers, which have been converted to octal notation. Stage 1 in the table shows the initial input, and after exchanging on the first bit we get to

Table 3

RADIX-EXCHANGE SORTING

Stage																	l	r	b	Stack
1	^{1}[0767	0127	1000	0075	1614	0252	1601	0423	1215	0652	0232	0775	1144	1245	1375	1277]	1	16	1	—
2	^{2}[0767	0127	0775	0075	0232	0252	0652	0423]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	1	8	2	(16, 2)
3	^{3}[0252	0127	0232	0075]	^{3}[0775	0767	0652	0423]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	1	4	3	(8, 3)(16, 2)
4	^{4}[0075	0127]	^{4}[0232	0252]	^{3}[0775	0767	0652	0423]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	1	2	4	(4, 4)(8, 3)(16, 2)
5	0075	0127	^{4}[0232	0252]	^{3}[0775	0767	0652	0423]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	3	4	4	(8, 3)(16, 2)
6	0075	0127	^{5}[0232	0252]	^{3}[0775	0767	0652	0423]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	3	4	5	(8, 3)(16, 2)
7	0075	0127	0232	0252	^{3}[0775	0767	0652	0423]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	5	8	3	(16, 2)
8	0075	0127	0232	0252	0423	^{4}[0767	0652	0775]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	6	8	4	(16, 2)
9	0075	0127	0232	0252	0423	0652	^{5}[0767	0775]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	7	8	5	(16, 2)
10	0075	0127	0232	0252	0423	0652	^{6}[0767	0775]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	7	8	6	(16, 2)
11	0075	0127	0232	0252	0423	0652	^{7}[0767	0775]	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	7	8	7	(16, 2)
12	0075	0127	0232	0252	0423	0652	0767	0775	^{2}[1215	1601	1614	1000	1144	1245	1375	1277]	9	16	2	—
13	0075	0127	0232	0252	0423	0652	0767	0775	^{3}[1215	1277	1375	1000	1144	1245]	^{3}[1614	1601]	9	14	3	(16, 3)
14	0075	0127	0232	0252	0423	0652	0767	0775	^{4}[1144	1000]	^{4}[1375	1277	1215	1245]	^{3}[1614	1601]	9	10	4	(14, 4)(16, 3)
15	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	^{4}[1375	1277	1215	1245]	^{3}[1614	1601]	11	14	4	(16, 3)
16	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	^{5}[1245	1277	1215]	^{5}[1375]	^{3}[1614	1601]	11	13	5	(14, 5)(16, 3)
17	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	1215	^{6}[1277	1245]	^{5}[1375]	^{3}[1614	1601]	12	13	6	(14, 5)(16, 3)
18	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	1215	1245	1277	1375	^{3}[1614	1601]	15	16	3	—
19	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	1215	1245	1277	1375	^{4}[1614	1601]	15	16	4	—
20	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	1215	1245	1277	1375	^{5}[1614	1601]	15	16	5	—
21	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	1215	1245	1277	1375	^{6}[1614	1601]	15	16	6	—
22	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	1215	1245	1277	1375	^{7}[1614	1601]	15	16	7	—
23	0075	0127	0232	0252	0423	0652	0767	0775	1000	1144	1215	1245	1277	1375	1614	1601	17	—	—	—

stage 2. Stage 2 sorts the first group on bit 2, and stage 3 works on bit 3. (The reader should mentally convert the octal notation to 10-bit binary numbers.) When we reach stage 5, after sorting on bit 4, we find that each group remaining has but a single element, so this part of the file need not be further examined. The notation "^{4}[*0232 0252*]" means that the subfile *0232 0252* is waiting to be sorted on bit 4 from the left. In this particular case, no progress occurs when sorting on bit 4; we need to go to bit 5 before the items are separated.

The complete sorting process shown in Table 3 takes 22 stages, somewhat more than the comparable number for quicksort (Table 2). Similarly, the number of bit inspections, 82, is rather high; but we shall see that the number of bit inspections for large N is actually less than the number of comparisons made by quicksort, assuming a uniform distribution of keys. The total number of exchanges in Table 3 is 17, which is quite reasonable. Note that bit inspections never have to go past bit 7 here, although 10-bit numbers are being sorted.

As in quicksort, we can use a stack to keep track of the "boundary line information" for waiting subfiles. Instead of sorting the smallest subfile first, it is convenient simply to go from left to right, since the stack size in this case can never exceed the number of bits in the keys being sorted. In the following algorithm the stack entry (r, b) is used to indicate the right boundary r of a subfile waiting to be sorted on bit b; the left boundary need not be recorded in the stack, it is implicit because of the left-to-right nature of the procedure.

Algorithm R (*Radix-exchange sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \cdots \le K_N$. Each key is assumed to be an m-bit binary number, e.g. $(a_1\, a_2 \ldots a_m)_2$; the ith most significant bit, a_i, is called "bit i" of the key. An auxiliary stack with room for at most $m - 1$ entries is needed for temporary storage. This algorithm essentially follows the radix-exchange partitioning procedure described in the text above; certain improvements in its efficiency are possible, as described in the text and exercises below.

R1. [Initialize.] Set the stack empty, and set $l \leftarrow 1$, $r \leftarrow N$, $b \leftarrow 1$.

R2. [Begin new stage.] (We now wish to sort the subfile $R_l \le \cdots \le R_r$ on bit b; from the nature of the algorithm, we have $l \le r$.) If $l = r$, go to step R10 (since a one-word file is already sorted). Otherwise set $i \leftarrow l$, $j \leftarrow r$.

R3. [Inspect K_i for 1.] Examine bit b of K_i. If it is a 1, go to step R6.

R4. [Increase i.] Increase i by 1. If $i \le j$, return to step R3; otherwise go to step R8.

R5. [Inspect K_{j+1} for 0.] Examine bit b of K_{j+1}. If it is a 0, go to step R7.

R6. [Decrease j.] Decrease j by 1. If $i \le j$, go to step R5; otherwise go to step R8.

R7. [Exchange R_i, R_{j+1}.] Interchange records $R_i \leftrightarrow R_{j+1}$; then go to step R4.

R8. [Test special cases.] (At this point a partitioning stage has been completed; $i = j + 1$, bit b of keys $K_l, \ldots, K_j$ is 0, and bit b of keys $K_i, \ldots, K_r$ is 1.) Increase b by 1. If $b > m$, where m is the total number of bits in the keys, go to step R10. (In such a case, the subfile $R_l \ldots R_r$ has been sorted. This test need not be made if there is no chance of having equal keys present in the file.) Otherwise if $j < l$ or $j = r$, go back to step R2 (all bits examined were 1 or 0, respectively). Otherwise if $j = l$, increase l by 1 and go to step R2 (there was only one 0 bit).

R9. [Put on stack.] Insert the entry (r, b) on top of the stack; then set $r \leftarrow j$ and go to step R2.

R10. [Take off stack.] If the stack is empty, we are done sorting; otherwise set $l \leftarrow r + 1$, remove the top entry (r', b') of the stack, set $r \leftarrow r'$, $b \leftarrow b'$, and return to step R2. ▮

Program R (*Radix-exchange sort*). The following MIX code uses essentially the same conventions as Program Q. We have rI1 $\equiv l - r$, rI2 $\equiv r$, rI3 $\equiv i$, rI4 $\equiv j$, rI5 $\equiv m - b$, rI6 $\equiv$ size of stack, except that it proves convenient for certain instructions (designated below) to leave rI3 $= i - j$ or rI4 $= j - i$. Because of the binary nature of radix-exchange, this program uses the operations SRB (shift right AX binary), JAE (jump A even), and JAO (jump A odd) which are defined in Section 4.5.2. We assume that $N \geq 2$.

```
01  START  ENT6  0              1     R1. Initialize. Set stack empty.
02         ENT1  1-N            1     l ← 1.
03         ENT2  N              1     r ← N.
04         ENT5  M-1            1     b ← 1.
05         JMP   1F             1     To R2 (omit testing l = r).
06  9H     INC6  1              S     R9. Put on stack.          [rI4 = l − j]
07         ST2   STACK,6(A)     S
08         ST5   STACK,6(B)     S     (r, b) ⇒ stack.
09         ENN1  0,4            S     rI1 ← l − j.
10         ENT2  -1,3           S     r ← j.
11  1H     ENT3  0,1            A     R2. Begin new stage.       [rI3 = i − j]
12         ENT4  0,2            A     i ← l, j ← r.              [rI3 = i − j]
13  3H     INC3  0,4            C'    R3. Inspect K_i for 1.
14         LDA   INPUT,3        C'
15         SRB   0,5            C'    units bit of rA ← bit b of K_i.
16         JAE   4F             C'    To R4 if it is 0.
17  6H     DEC4  1,3            C''+X R6. Decrease j. j ← j − 1.  [rI4 = j − i]
18         J4N   8F             C''+X To R8 if j < i.            [rI4 = j − i]
19  5H     INC4  0,3            C''   R5. Inspect K_{j+1} for 0.
20         LDA   INPUT+1,4      C''
21         SRB   0,5            C''   units bit of rA ← bit b of K_{j+1}.
22         JAO   6B             C''   To R6 if it is 1.
23  7H     LDA   INPUT+1,4      B     R7. Exchange R_i, R_{j+1}.
24         LDX   INPUT,3        B
25         STX   INPUT+1,4      B
26         STA   INPUT,3        B
```

27	4H	DEC3	-1,4	$C' - X$	*R4. Increase i.* $i \leftarrow i + 1$. [rI3 $= i - j$]
28		J3NP	3B	$C' - X$	To R3 if $i \leq j$. [rI3 $= i - j$]
29		INC3	0,4	$A - X$	rI3 $\leftarrow i$.
30	8H	J5Z	0F	A	*R8. Test special cases.* [rI4 = unknown]
31		DEC5	1	$A - G$	To R10 if $b = m$, else $b \leftarrow b - 1$.
32		ENT4	-1,3	$A - G$	rI4 $\leftarrow j$.
33		DEC4	0,2	$A - G$	rI4 $\leftarrow j - r$.
34		J4Z	1B	$A - G$	To R2 if $j = r$.
35		DEC4	0,1	$A - G - R$	rI4 $\leftarrow j - l$.
36		J4N	1B	$A - G - R$	To R2 if $j < l$.
37		J4NZ	9B	$A - G - L - R$	To R9 if $j \neq l$.
38		INC1	1	K	$l \rightarrow l + 1$.
39	2H	J1NZ	1B	$K + S$	Jump if $l \neq r$.
40	0H	ENT1	1,2	$S + 1$	*R10. Take off stack.*
41		LD2	STACK,6(A)	$S + 1$	
42		DEC1	0,2	$S + 1$	
43		LD5	STACK,6(B)	$S + 1$	stack $\Rightarrow (r, b)$.
44		DEC6	1	$S + 1$	
45		J6NN	2B	$S + 1$	To R2 if stack was nonempty. ▮

The running time of this radix exchange program depends on

$$\begin{aligned}
A &= \text{number of stages encountered with } l < r;\\
B &= \text{number of exchanges;}\\
C &= C' + C'' = \text{number of bit inspections;}\\
G &= \text{number of times } b > m \text{ in step R8;}\\
K &= \text{number of times } b \leq m,\ j = l \text{ in step R8;}\\
L &= \text{number of times } b \leq m,\ j < l \text{ in step R8;}\\
R &= \text{number of times } b \leq m,\ j = r \text{ in step R8;}\\
S &= \text{number of times things are entered onto the stack;}\\
X &= \text{number of times } j < i \text{ in step R6.}
\end{aligned} \tag{29}$$

By Kirchhoff's law, $S = A - G - K - L - R$; so the total running time comes to $27A + 8B + 8C - 23G - 14K - 17L - 19R - X + 13$ units. The bit-inspection loops can be made somewhat faster, as shown in exercise 34, at the expense of a more complicated program. It is also possible to increase the speed of radix exchange by using straight insertion whenever $r - l$ is sufficiently small, as we did in Algorithm Q; but we shall not dwell on these refinements.

In order to analyze the running time of radix exchange, two kinds of input data suggest themselves:

i) Assume that $N = 2^m$ and that the keys to be sorted are simply the integers $0, 1, 2, \ldots, 2^m - 1$ in random order; or

ii) Assume that $m = \infty$ (unlimited precision) and that the keys to be sorted are independent uniformly distributed real numbers in $[0, 1)$.

The analysis of case (i) is relatively easy, so it has been left as an exercise for the reader (see exercise 35). Case (ii) is comparatively difficult, so it has *also* been left as an exercise. The following table shows crude approximations to the results of these analyses:

Quantity	Case (i)	Case (ii)
A	N	αN
B	$\frac{1}{4}N \log_2 N$	$\frac{1}{4}N \log_2 N$
C	$N \log_2 N$	$N \log_2 N$
G	$\frac{1}{2}N$	0
K	0	$\frac{1}{2}N$
L	0	$\frac{1}{2}(\alpha - 1)N$
R	0	$\frac{1}{2}(\alpha - 1)N$
S	$\frac{1}{2}N$	$\frac{1}{2}N$
X	$\frac{1}{2}N$	$\frac{1}{4}(\alpha + 1)N$

(30)

Here $\alpha = 1/(\ln 2) = 1.4427$. Note that the average number of exchanges, bit inspections, and stack accesses is essentially the same for both cases, even though case (ii) takes about 44 percent more stages. Our MIX program takes approximately $14.4\, N \ln N$ units of time, on the average, to sort N items in case (ii), and this could be cut to about $11.5\, N \ln N$ using the suggestion of exercise 34; the corresponding figure for Program Q is $12.7\, N \ln N$, which can be decreased to about $11.7\, N \ln N$ 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 MIX it is actually a little quicker than quicksort. Exercise 53 indicates to what extent the process slows down for a nonuniform distribution. It is important to note that our entire analysis is predicated on the assumption that keys are distinct; *radix exchange as defined above is not especially efficient when equal keys are present*, since it goes through several time-consuming stages trying to separate sets of identical keys before b becomes $> m$. One plausible way to remedy this defect is suggested in the answer to exercise 40.

Both radix exchange and quicksort are essentially based on the idea of partitioning. Records are exchanged until the file is split into two parts: a left-hand subfile, in which all keys are $\leq K$, for some K, and a right-hand subfile in which all keys are $\geq K$. Quicksort chooses K to be an actual key in the file, while radix exchange essentially chooses an artificial key K based on binary representations. From a historical standpoint, radix exchange was discovered by P. Hildebrandt, H. Isbitz, H. Rising, and J. Schwartz [*JACM* **6** (1959), 156–163], about a year earlier than quicksort. Other partitioning schemes are also possible; for example, John McCarthy has suggested setting $K \approx \frac{1}{2}(u + v)$, if all keys are known to lie between u and v.

Still another partitioning strategy has been proposed by M. H. van Emden [*CACM* **13** (1970), 563–567]: Instead of choosing K in advance, we "learn"

what a good K might be, by keeping track of $K' = \max(K_l, \ldots, K_i)$ and $K'' = \min(K_j, \ldots, K_r)$ as partitioning proceeds. We may increase i until encountering a key greater than K', then decrease j until encountering a key less than K'', then exchange and/or adjust K' and K''. Empirical tests on this "interval-exchange sort" method indicate that it requires about $1.64\, N \ln N = 1.14\, N \log_2 N$ comparisons; it is the only sorting method discussed in this book for which no adequate theoretical explanation of the behavior has yet been found.

A generalization of radix exchange to radices higher than 2 is discussed in Section 5.2.5.

***Asymptotic methods.** The analysis of exchange sorting algorithms leads to some particularly instructive mathematical problems which enable us to learn more about how to find the asymptotic behavior of functions. For example, we came across the function

$$W_n = \frac{1}{n!} \sum_{0 \le r < s \le n} s! r^{n-s} \tag{31}$$

in our analysis of the bubble sort (Eq. 9); what is its asymptotic value?

We can proceed as in our study of the number of involutions, Eq. 5.1.4–41; the reader will find it helpful to review the discussion at the end of Section 5.1.4 before reading further.

Inspection of (31) shows that the contribution for $s = n$ is larger than that for $s = n - 1$, etc; this suggests replacing s by $n - s$. In fact, we soon discover that it is most convenient to use the substitutions $t = n - s + 1$, $m = n + 1$, so that (31) becomes

$$\frac{1}{m} W_{m-1} = \frac{1}{m!} \sum_{1 \le t < m} (m - t)! \sum_{0 \le r < m-t} r^{t-1}. \tag{32}$$

The inner sum has a well-known asymptotic series obtained from Euler's summation formula,

$$\begin{aligned} \sum_{0 \le r < N} r^{t-1} &= \frac{N^t}{t} - \frac{1}{2}(N^{t-1} - \delta_{t1}) + \frac{B_2}{2!}(t-1)(N^{t-2} - \delta_{t2}) + \cdots \\ &= \frac{1}{t} \sum_{0 \le j \le k} \binom{t}{j} B_j (N^{t-j} - \delta_{tj}) + O(N^{t-k}) \end{aligned} \tag{33}$$

(see exercise 1.2.11.2–4), hence our problem reduces to studying sums of the form

$$\frac{1}{m!} \sum_{1 \le t < m} (m - t)!(m - t)^t t^k, \qquad k \ge -1. \tag{34}$$

As in Section 5.1.4 we can show that the value of this summand is negligible, $O(\exp(-n^\delta))$, whenever t is greater than $m^{1/2+\epsilon}$; hence we may put $t =$

$O(m^{1/2+\epsilon})$ and replace the factorials by Stirling's approximation:

$$\frac{(m-t)!(m-t)^t}{m!}$$
$$= \sqrt{1-\frac{t}{m}}\exp\left(\frac{t}{12m^2} - \left(\frac{t^2}{2m} + \frac{t^3}{3m^2} + \frac{t^4}{4m^3} + \frac{t^5}{5m^4}\right) + O(m^{-2+6\epsilon})\right).$$

We are therefore interested in the asymptotic value of

$$r_k(m) = \sum_{1\le t<m} e^{-t^2/2m}t^k, \qquad k \ge -1. \tag{35}$$

(The sum could also be extended to the full range $1 \le t < \infty$ without changing its asymptotic value, since the values for $t > m^{1/2+\epsilon}$ are negligible as stated above.)

Let $g_k(x) = x^k e^{-x^2}$ and $f_k(x) = g_k(x/\sqrt{2m})$. When $k \ge 0$, Euler's summation formula tells us that

$$\sum_{0\le t<m} f_k(t) = \int_0^m f_k(x)\,dx + \sum_{1\le j\le p} \frac{B_j}{j!}\left(f_k^{(j-1)}(m) - f_k^{(j-1)}(0)\right) + R_p,$$

$$R_p = \frac{(-1)^{p+1}}{p!}\int_0^m B_p(\{x\})f_k^{(p)}(x)\,dx$$
$$= \left(\frac{1}{\sqrt{2m}}\right)^p O\left(\int_0^\infty |g_k^{(p)}(y)|\,dy\right) = O(m^{-p/2}); \tag{36}$$

hence we can get an asymptotic series for $r_k(m)$ whenever $k \ge 0$ by using essentially the same ideas we have used before. But when $k = -1$ the method breaks down, since $f_{-1}(0)$ is undefined; we can't merely sum from 1 to m either, because the remainders don't give smaller and smaller powers of m when the lower limit is 1. (This is the crux of the matter, and the reader should make sure he appreciates the problem before reading further.)

To resolve the dilemma we can define $g_{-1}(x) = (e^{-x^2} - 1)/x$, $f_{-1}(x) = g_{-1}(x/\sqrt{2m})$; then $f_{-1}(0) = 0$ and $r_{-1}(m)$ can be obtained from $\sum_{0\le t<m} f_{-1}(t)$ in a simple way. Equation (36) is now valid for $k = -1$, and the remaining integral is "well known,"

$$\frac{2}{\sqrt{2m}}\int_0^m f_{-1}(x)\,dx = 2\int_0^m \frac{e^{-x^2/2m}-1}{x}\,dx = \int_0^{m/2}\frac{e^{-y}-1}{y}\,dy$$
$$= \int_0^1 \frac{e^{-y}-1}{y}\,dy + \int_1^{m/2}\frac{e^{-y}}{y}\,dy - \ln(m/2)$$
$$= -\gamma - \ln(m/2) + O(e^{-m/2})$$

by exercise 43.

Now we have enough facts and formulas to grind out the answer,

$$W_n = \tfrac{1}{2}m \ln m + \tfrac{1}{2}(\gamma + \ln 2)m - \tfrac{2}{3}\sqrt{2\pi m} + \tfrac{31}{36} + O(n^{-1/2}), \qquad m = n + 1, \tag{37}$$

as shown in exercise 44. This completes our analysis of the bubble sort.

For the analysis of radix exchange sorting, we need to know the asymptotic value of the finite sum

$$U_n = \sum_{k \geq 2} \binom{n}{k} (-1)^k \frac{1}{2^{k-1} - 1} \tag{38}$$

as $n \to \infty$. This question turns out to be harder than any of the other asymptotic problems we have met so far; the elementary methods of power series expansions, Euler's summation formula, etc., turn out to be inadequate. The following derivation has been suggested by N. G. de Bruijn.

To get rid of the cancellation effects of the large factors $\binom{n}{k}(-1)^k$ in (38), we start by rewriting the sum as an infinite series

$$U_n = \sum_{k \geq 2} \binom{n}{k} (-1)^k \sum_{j \geq 1} \left(\frac{1}{2^{k-1}}\right)^j = \sum_{j \geq 1} (2^j(1 - 2^{-j})^n - 2^j + n). \tag{39}$$

If we set $x = n/2^j$, the summand is

$$2^j(1 - 2^{-j})^n - 2^j + n = \frac{n}{x}\left(\left(1 - \frac{x}{n}\right)^n - 1 + x\right).$$

When $x \leq n^\epsilon$, we have

$$\left(1 - \frac{x}{n}\right)^n = \exp\left(n \ln\left(1 - \frac{x}{n}\right)\right) = \exp\left(-x + x^2 O(n^{-1})\right), \tag{40}$$

and this suggests approximating (39) by

$$T_n = \sum_{j \geq 1} (2^j e^{-n/2^j} - 2^j + n). \tag{41}$$

To justify this approximation, we have $U_n - T_n = X_n + Y_n$, where

$$\begin{aligned} X_n &= \sum_{\substack{j \geq 1 \\ 2^j < n^{1-\epsilon}}} (2^j(1 - 2^{-j})^n - 2^j e^{-n/2^j}) && \text{[i.e., the terms for } x > n^\epsilon\text{]} \\ &= \sum_{\substack{j \geq 1 \\ 2^j < n^{1-\epsilon}}} O(ne^{-n/2^j}) && \text{[since } 0 < 1 - 2^{-j} < e^{-2^{-j}}\text{]} \\ &= O(n \log n\, e^{-n^\epsilon}) && \text{[since there are } O(\log n) \text{ terms];} \end{aligned}$$

and

$$Y_n = \sum_{\substack{j\ge 1\\ 2^j \ge n^{1-\epsilon}}} \left(2^j(1-2^{-j})^n - 2^j e^{-n/2^j}\right) \qquad \text{[the terms for } x \le n^{-\epsilon}\text{]}$$

$$= \sum_{\substack{j\ge 1\\ 2^j \ge n^{1-\epsilon}}} \left(e^{-n/2^j}\left(\frac{n}{2^j}\right) O(1)\right) \qquad \text{[by (40)].}$$

Our discussion below will demonstrate that the latter sum is $O(1)$; hence $U_n - T_n = O(1)$. (See exercise 47.)

So far we haven't applied any techniques that are really different from those we have used before; but the study of T_n requires a new idea, based on simple principles of complex variable theory. If x is any positive number, we have

$$e^{-x} = \frac{1}{2\pi i}\int_{1/2-i\infty}^{1/2+i\infty} \Gamma(z)x^{-z}\,dz = \frac{1}{2\pi}\int_{-\infty}^{\infty} \Gamma(\tfrac{1}{2}+it)x^{-(1/2+it)}\,dt. \qquad (42)$$

To prove this identity, consider the path of integration shown in Fig 20(a), where N, N', and M are large. The value of the integral along this contour is the sum of the residues inside, namely,

$$\sum_{0\le k<M} x^{-k} \lim_{z\to -k} (z+k)\Gamma(z) = \sum_{0\le k<M} x^{-k}\frac{(-1)^k}{k!}.$$

The integral on the top line is $O(\int_{-\infty}^{1/2} |\Gamma(t+iN)|x^{-t}\,dt)$, and we have the well-known bound

$$\Gamma(t+iN) = O(N^{t-1/2}e^{-\pi N/2}) \qquad \text{as} \qquad N\to\infty.$$

[For properties of the gamma function see, for example, A. Erdélyi et al., *Higher Transcendental Functions* 1 (New York: McGraw-Hill, 1953), Chapter 1.] Therefore the top line integral is quite negligible, $O(e^{-\pi N/2}\int_{-\infty}^{1/2} (N/x)^t\,dt)$. The bottom line integral has a similar innocuous behavior. For the integral along the left line we use the fact that

$$\begin{aligned}\Gamma(\tfrac{1}{2}+it-M) &= \Gamma(\tfrac{1}{2}+it)/(-M+\tfrac{1}{2}+it)\ldots(-1+\tfrac{1}{2}+it)\\ &= \Gamma(\tfrac{1}{2}+it)O(1/(M-1)!);\end{aligned}$$

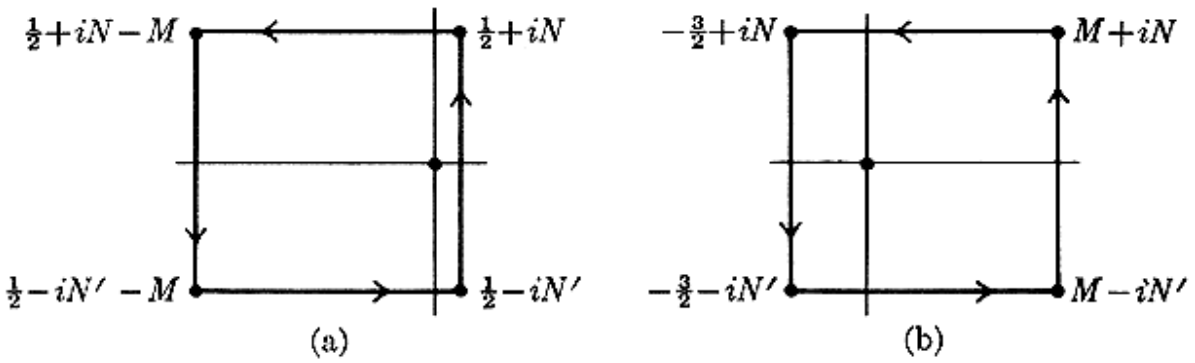


Fig. 20. Contours of integration for gamma-function identities.

hence the left-hand integral is $O(x^{M-1/2}/(M-1)!)\int_{-\infty}^{\infty}|\Gamma(\frac{1}{2}+it)|\,dt$. Therefore as M, N, $N' \to \infty$, only the right-hand integral survives, and this proves (42). In fact, (42) remains valid if we replace "$\frac{1}{2}$" by any positive number.

The same argument can be used to derive many other useful relations involving the gamma function. We can replace x^{-z} by other functions of z; or we can replace the constant $\frac{1}{2}$ by other quantities. For example,

$$\frac{1}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\Gamma(z)x^{-z}\,dz = e^{-x} - 1 + x, \tag{43}$$

and this is the critical quantity in our formula (41) for T_n:

$$T_n = n\sum_{j\geq 1}\frac{1}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\Gamma(z)(n/2^j)^{-1-z}\,dz. \tag{44}$$

The sum may be placed inside the integrals, since convergence is absolutely well-behaved; we have

$$\sum_{j\leq 1}(n/2^j)^w = n^w\sum_{j\leq 1}(1/2^w)^j = n^w/(2^w-1), \qquad \text{when} \qquad \Re(w) > 0,$$

since $|2^w| = 2^{\Re(w)} > 1$. Therefore

$$T_n = \frac{n}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\frac{\Gamma(z)n^{-1-z}}{2^{-1-z}-1}\,dz, \tag{45}$$

and it remains to evaluate the latter integral.

This time we integrate along a path which extends far to the *right*, as in Fig. 20(b). The top line integral is $O(n^{1/2}e^{-\pi M/2}\int_{-3/2}^{M}N^t\,dt)$, if $2^{iN}\neq 1$, and the bottom line integral is equally negligible. The right-hand line integral is $O(n^{-1-M}\int_{-\infty}^{\infty}|\Gamma(M+it)|\,dt)$. Fixing M and letting N, $N' \to \infty$ shows that $-T_n/n$ is $O(n^{-1-M})$ plus the sum of the residues in the region $-3/2 < \operatorname{Re}(z) < M$. Let $M \to \infty$; $\Gamma(z)$ has simple poles at $z = -1$ and 0, n^{-1-z} has no poles, and $1/(2^{-1-z}-1)$ has simple poles when $z = -1 + 2\pi ik/\ln 2$.

The double pole at $z = -1$ is the hardest to handle. We can use the well-known relation

$$\Gamma(z+1) = \exp\left(-\gamma z + \zeta(2)z^2/2 - \zeta(3)z^3/3 + \zeta(4)z^4/4 - \cdots\right),$$

where $\zeta(s) = 1^{-s} + 2^{-s} + 3^{-s} + \cdots = H_\infty^{(s)}$,

to deduce the following expansions when $w = z + 1$ is small:

$$\Gamma(z) = \frac{\Gamma(w+1)}{w(w-1)} = -w^{-1} + (\gamma - 1) + O(w),$$

$$n^{-1-z} = 1 - (\ln n)w + O(w^2),$$

$$1/(2^{-1-z}-1) = -w^{-1}/\ln 2 - \tfrac{1}{2} + O(w).$$

The residue at $z = -1$ is the coefficient of w^{-1} in the product of these three formulas, namely $\frac{1}{2} - (\ln n + \gamma - 1)/\ln 2$. Adding the other residues gives the formula

$$T_n/n = \frac{\ln n + \gamma - 1}{\ln 2} - \frac{1}{2} + f(n) + \frac{2}{n}, \tag{46}$$

where $f(n)$ is a rather strange function,

$$f(n) = \frac{2}{\ln 2} \sum_{k \geq 1} \Re\left(\Gamma(-1 - 2\pi ik/\ln 2) \exp\left(2\pi ik \log_2 n\right)\right). \tag{47}$$

Note that $f(n) = f(2n)$. The average value of $f(n)$ is zero, since the average value of each term is zero. (We may assume that $(\log_2 n) \bmod 1$ is uniformly distributed, in view of the results about floating-point numbers in Section 4.2.4.) Furthermore since $|\Gamma(-1 + it)| = |\pi/(t(1 + t^2) \sinh \pi t)|^{1/2}$, it is not difficult to show that

$$f(n) < 0.0000001725;$$

thus we may safely ignore $f(n)$ for practical purposes. For theoretical purposes, however, we can't obtain an asymptotic expansion of U_n without it; that is why U_n is a comparatively difficult function to analyze.

In summary, we have proved that

$$U_n = n \log_2 n + n\left(\frac{\gamma - 1}{\ln 2} - \frac{1}{2} + f(n)\right) + O(1). \tag{48}$$

Other examples of this gamma-function method appear in exercises 51–53 and in Section 6.3.

EXERCISES

1. [*M20*] Let $a_1 \ldots a_n$ be a permutation of $\{1, \ldots, n\}$, and let i and j be indices such that $i < j$ and $a_i > a_j$. Let $a'_1 \ldots a'_n$ be the permutation obtained from $a_1 \ldots a_n$ by interchanging a_i and a_j. Can $a'_1 \ldots a'_n$ have more inversions than $a_1 \ldots a_n$?

▶ 2. [*M25*] (a) What is the minimum number of exchanges which will sort the permutation 3 7 6 9 8 1 4 5 2? (b) In general, given any permutation $\pi = a_1 \ldots a_n$ of $\{1, \ldots, n\}$, let xch (π) be the minimum number of exchanges which will sort π into increasing order. Express xch (π) in terms of "simpler" characteristics of π. (Cf. exercise 5.2.1–39.)

3. [*10*] Is the bubble sort Algorithm B a "stable" sorting algorithm?

4. [*M23*] If $t = 1$ in step B4, we could actually terminate Algorithm B immediately, because the subsequent step B2 will do nothing useful. What is the probability that $t = 1$ will occur in step B4 when sorting a random permutation?

5. [*M25*] Let $b_1\, b_2 \ldots b_n$ be the inversion table for the permutation $a_1\, a_2 \ldots a_n$. Show that the value of BOUND after r passes of the bubble sort is $\max \{b_i + i \mid b_i \geq r\} - r$, for $0 \leq r \leq \max (b_1, \ldots, b_n)$.

6. [*M22*] Let $a_1 \ldots a_n$ be a permutation of $\{1, \ldots, n\}$ and let $a'_1 \ldots a'_n$ be its inverse. Show that the number of passes to bubble-sort $a_1 \ldots a_n$ is $1 + \max(a'_1 - 1, a'_2 - 2, \ldots, a'_n - n)$.

7. [*M28*] Calculate the standard deviation of the number of passes for the bubble sort, and express it in terms of n and the function $P(n)$. [Cf. Eqs. (6) and (7).]

8. [*M24*] Derive Eq. (8).

9. [*M48*] Analyze the number of passes and the number of comparisons in the cocktail-shaker shorting [shic] algorithm. *Note:* See exercise 5.4.8–9 for partial information.

10. [*M26*] Let $a_1 a_2 \ldots a_n$ be a 2-ordered permutation of $\{1, 2, \ldots, n\}$. (a) What are the coordinates of the endpoints of the a_i-th step of the corresponding lattice path? (Cf. Fig. 11.) (b) Prove that comparison/exchange of $a_1{:}a_2$, $a_3{:}a_4$, ..., corresponds to folding the path about the diagonal, as in Fig. 18(b). (c) Prove that comparison/exchange of $a_2{:}a_{2+d}$, $a_4{:}a_{4+d}$, ..., corresponds to folding the path about a line m units below the diagonal, as in Figs. 18(c), (d), and (e), when $d = 2m - 1$.

▶ **11.** [*M25*] What permutation of $\{1, 2, \ldots, 16\}$ maximizes the number of exchanges done by Batcher's algorithm?

12. [*24*] Write a MIX program for Algorithm M, assuming that MIX is a binary computer with the operations AND, SRB. How much time does your program take to sort the sixteen records in Table 1?

13. [*10*] Is Batcher's method a "stable" sorting algorithm?

14. [*M21*] Let $c(N)$ be the number of key comparisons used to sort N elements by Batcher's method; this is the number of times step M4 is performed. (a) Show that $c(2^t) = 2c(2^{t-1}) + (t-1)2^{t-1} + 1$, for $t \geq 1$. (b) Find a simple expression for $c(2^t)$ as a function of t. *Hint:* Consider the sequence $x_t = c(2^t)/2^t$.

15. [*M38*] The object of this exercise is to analyze the function $c(N)$ of exercise 13, and to find a formula for $c(N)$ when $N = 2^{e_1} + 2^{e_2} + \cdots + 2^{e_r}$, $e_1 > e_2 > \cdots > e_r \geq 0$. (a) Let $a(N) = c(N+1) - c(N)$. Prove that $a(2n) = a(n) + \lfloor \log_2 (2n) \rfloor$, and $a(2n+1) = a(n) + 1$; hence

$$a(N) = \binom{e_1 + 1}{2} - r(e_1 - 1) + (e_1 + e_2 + \cdots + e_r).$$

(b) Let $x(n) = a(n) - a(\lfloor n/2 \rfloor)$, so that $a(n) = x(n) + x(\lfloor n/2 \rfloor) + x(\lfloor n/4 \rfloor) + \cdots$. Let $y(n) = x(1) + x(2) + \cdots + x(n)$; and let $z(2n) = y(2n) - a(n)$, $z(2n+1) = y(2n+1)$. Prove that $c(N+1) = z(N) + 2z(\lfloor N/2 \rfloor) + 4z(\lfloor N/4 \rfloor) + \cdots$. (c) Prove that $y(N) = N + (\lfloor N/2 \rfloor + 1)(e_1 - 1) - 2^{e_1} + 2$. (d) Now put everything together and find a formula for $c(N)$ in terms of the exponents e_j, holding r fixed.

16. [*HM46*] Find the asymptotic value of the *average* number of exchanges occurring when Batcher's method is applied to $N = 2^t$ distinct elements, randomly ordered.

▶ **17.** [*20*] Where in Algorithm Q do we use the fact that K_0 and K_{N+1} have the values postulated in (13)?

▶ **18.** [*20*] Explain how the computation proceeds in Algorithm Q when all of the input keys are equal. What would happen if the "$<$" signs in steps Q3 and Q5 were changed to "$\leq$" instead?

19. [*15*] Would Algorithm Q still work properly if a queue (first-in-first-out) were used instead of a stack (last-in-first-out)?

20. [*M20*] What is the largest possible number of elements that will ever be on the stack at once in Algorithm Q, as a function of M and N?

21. [*20*] Explain why the partitioning phase of Algorithm Q takes the number of comparisons, transfers, etc., specified in (17).

22. [*M25*] Let p_{kN} be the probability that the quantity A in (14) will equal k, when Algorithm Q is applied to a random permutation of $\{1, 2, \ldots, N\}$, and let $A_N(z) = \sum_k p_{kN} z^k$ be the corresponding generating function. Prove that $A_N(z) = 1$ for $N \le M$, and $A_N(z) = z(\sum_{1 \le s \le N} A_{s-1}(z) A_{N-s}(z))/N$ for $N > M$. Find similar recurrence relations defining the other probability distributions $B_N(z)$, $C_N(z)$, $D_N(z)$, $E_N(z)$, $L_N(z)$, $X_N(z)$.

23. [*M24*] Let A_N, B_N, D_N, E_N, L_N, X_N be the average values of the corresponding quantities in (14), when sorting a random permutation of $\{1, 2, \ldots, N\}$. Find recurrence relations for these quantities, analogous to (18); and solve these recurrences to obtain (25).

24. [*M21*] Algorithm Q obviously does a few more comparisons than it needs to, since we can have $i = j$ and even $i > j$ in steps Q3 and Q5. How many comparisons C_N would be done on the average if we avoided all comparisons when $i \ge j$?

25. [*M20*] When the input keys are the numbers $1\ 2 \ldots N$ in order, what are the exact values of the quantities A, B, C, D, E, L, and X in the timing of Program Q? (Assume that $N > M$.)

▶ **26.** [*M21*] Construct an input file which makes Program Q go even more slowly than it does in exercise 25. (Try to find a really bad case.)

27. [*M23*] What input leads to the *best* case of Algorithm Q? Give approximate values of A, B, C, D, E, X in this case.

28. [*M26*] Find the recurrence relation analogous to (20) which is satisfied by the average number of comparisons in Singleton's modification of Algorithm Q (choosing s as the median of $\{K_1, K_{\lfloor (N+1)/2 \rfloor}, K_N\}$ instead of $s = K_1$).

29. [*HM40*] Find the asymptotic value of the number of comparisons in Singleton's "median of three" method (cf. exercise 28).

30. [*25*] (P. Shackleton.) When *multiword keys* are being sorted, many sorting methods become progressively slower as the file gets closer to its final order, since equal and nearly-equal keys require an inspection of several words to determine the proper lexicographic order. (See exercise 5–5.) Files which arise in practice often involve nearly-equal keys, so this phenomenon can have a significant impact on the sorting time. Algorithm Q has this difficulty; Algorithm R does not, since it inspects only one bit each time (although the presence of equal and nearly-equal keys can greatly increase the running time of Algorithm R for other reasons).

Explain how Algorithm Q can be extended to avoid this difficulty; within a subfile in which the leading k words are known to have constant values for all keys, only the $(k + 1)$st words of the keys should be inspected.

▶ **31.** [*20*] (C. A. R. Hoare.) Suppose that, instead of sorting an entire file, we only want to determine the mth smallest of a given set of n elements. Show that "quicksort" can be adapted to this purpose, avoiding many of the computations required to do a complete sort.

32. [*M40*] Find a simple "closed form" expression for C_{nm}, the average number of key comparisons required to find the mth smallest of n elements by the method of

exercise 31. (For simplicity, avoid all comparisons with $i \geq j$ as in exercise 24.) What is the asymptotic behavior of $C_{(2m-1)m}$, the average number of comparisons needed to find the median of $2m - 1$ elements by Hoare's method?

▶ **33.** [*20*] Design an algorithm which rearranges all the numbers in a given table so that all *negative* values precede all positive ones. (The items need not be sorted completely, just separated between negative and nonnegative.) Your algorithm should use the minimum possible number of exchanges.

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

35. [*M23*] Analyze the values of the frequencies A, B, C, G, K, L, R, S, and X which arise in radix exchange sorting using "case (i) input."

36. [*M27*] Given a sequence of numbers $\langle a_n \rangle = a_0, a_1, a_2, \ldots$, define its *binomial transform* $\langle \hat{a}_n \rangle = \hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots$ by the rule

$$\hat{a}_n = \sum_k \binom{n}{k} (-1)^k a_k.$$

(a) Prove that $\langle \hat{\hat{a}}_n \rangle = \langle a_n \rangle$. (b) Find the binomial transform of the sequences $\langle 1 \rangle$; $\langle n \rangle$; $\langle \binom{n}{m} \rangle$, for fixed m; $\langle a^n \rangle$, for fixed a; $\langle \binom{n}{m} a^n \rangle$, for fixed a and m. (c) Given that a sequence $\langle x_n \rangle$ satisfies the relation

$$x_n = a_n + 2^{1-n} \sum_{k \geq 2} \binom{n}{k} x_k, \qquad \text{for} \qquad n \geq 2; \qquad x_0 = x_1 = a_0 = a_1 = 0;$$

prove that

$$x_n = \sum_{k \geq 2} \binom{n}{k} (-1)^k \frac{2^{k-1} \hat{a}_k}{2^{k-1} - 1} = a_n + \sum_{k \geq 2} \binom{n}{k} (-1)^k \frac{\hat{a}_k}{2^{k-1} - 1}.$$

37. [*M28*] Determine all sequences $\langle a_n \rangle$ such that $\langle \hat{a}_n \rangle = \langle a_n \rangle$, in the sense of exercise 36.

▶ **38.** [*M30*] Find A_N, B_N, C_N, G_N, K_N, L_N, R_N, and X_N, the average values of the quantities in (29) when radix exchange is applied to "case (ii) input." Express your answers in terms of N and the functions

$$U_n = \sum_{k \geq 2} \binom{n}{k} \frac{(-1)^k}{2^{k-1} - 1} \qquad V_n = \sum_{k \geq 2} \binom{n}{k} \frac{(-1)^k k}{2^{k-1} - 1} = n(U_n - U_{n-1}).$$

[*Hint:* See exercise 36.]

39. [*20*] The results shown in (30) indicate that radix exchange takes about $1.44N$ stages when it is applied to random input. Prove that quicksort will never require more than N stages; and explain why radix exchange often does.

40. [*21*] Explain how to modify Algorithm R so that it works with reasonable efficiency when sorting files containing numerous equal keys.

41. [*23*] Formulate van Emden's "interval exchange sort," obtaining a precise description of an algorithm.

42. [*M43*] Analyze the interval exchange algorithm of exercise 41.

43. [*HM21*] Prove that $\int_0^1 y^{-1}(e^{-y}-1)\,dy + \int_1^\infty y^{-1}e^{-y}\,dy = -\gamma$. [*Hint:* Consider $\lim_{a\to 0+} y^{a-1}$.]

44. [*HM24*] Derive (37) as suggested in the text.

45. [*HM20*] Explain why (43) is true, when $x > 0$.

46. [*HM20*] What is the value of $(1/2\pi i)\int_{a+i\infty}^{a+i\infty} \Gamma(z)n^{s-z}\,dz/(2^{s-z}-1)$, given that s is a positive integer and $0 < a < s$?

47. [*HM21*] Prove that $\sum_{j\geq 1}(n/2^j)e^{-n/2^j}$ is a bounded function of n.

48. [*HM24*] Find the asymptotic value of the quantity V_n defined in exercise 38, using a method analogous to the text's study of U_n, obtaining terms up to $O(1)$.

49. [*HM24*] Extend the asymptotic formula (47) for U_n to $O(n^{-1})$.

50. [*HM24*] Find the asymptotic value of the function

$$U_{mn} = \sum_{k\geq 2}\binom{n}{k}(-1)^k\frac{1}{m^{k-1}-1},$$

when m is any fixed number greater than 1. (When m is an integer greater than 2, this quantity arises in the study of generalizations of radix exchange, as well as the "trie memory" search algorithms of Section 6.3.)

▶ **51.** [*HM28*] Show that the gamma-function approach to asymptotic problems can be used instead of Euler's summation formula to derive the asymptotic expansion of $r_k(m)$. (Cf. Eq. 35). This gives us a uniform method for studying $r_k(m)$ for all k, without relying on "tricks" such as the text's introduction of $g_{-1}(x) = (e^{-x^2}-1)/x$.)

52. [*HM35*] (N. G. de Bruijn.) What is the asymptotic behavior of the sum

$$S_n = \sum_{t\geq 1}\binom{2n}{n+t}d(t),$$

where $d(t)$ is the number of divisors of t? (Thus, $d(1) = 1$, $d(2) = d(3) = 2$, $d(4) = 3$, $d(5) = 2$, etc. This question arises in connection with the analysis of a tree traversal algorithm, exercise 2.3.1–11.) Find the value of $S_n/\binom{2n}{n}$ to terms of $O(n^{-1})$.

53. [*HM42*] Analyze the average number of bit inspections and exchanges done by radix exchange when the input data consists of infinite-precision binary numbers in $[0, 1)$, each of whose bits is independently equal to 1 with probability p. (Only the case $p = \frac{1}{2}$ is discussed in the text; the methods we have used can be generalized to arbitrary p.) Consider in particular the case $p = 1/\phi = .61803\ldots$.

54. [*HM24*] (S. O. Rice.) Show that U_n can be written

$$U_n = (-1)^n\frac{n!}{2\pi i}\oint_C \frac{dz}{z(z-1)\ldots(z-n)}\,\frac{1}{2^{z-1}-1},$$

where C is a skinny closed curve encircling the points $2, 3, \ldots, n$. Changing C to an arbitrarily large circle centered at the origin, derive the convergent series

$$U_n = n(H_{n-1}-1)/(\ln 2) - \tfrac{1}{2}n + 2 + \frac{2}{\ln 2}\sum_{m\geq 1}\Re\,(B(n+1, -1+ibm)),$$

where $b = 2\pi/(\ln 2)$, and $B(n+1, -1+ibm) = \Gamma(n+1)\Gamma(-1+ibm)/\Gamma(n+ibm) = n!/\Pi_{0\leq k\leq n}(k-1+ibm)$.

▶ **55.** [*22*] Show how Program Q should be modified so that the partitioning element is the median of the three keys (28).

56. [*M44*] Analyze the average behavior of the quantities which 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. [Cf. exercise 29.]

▶ **57.** [*HM24*] Find the asymptotic value of the number of right-to-left maxima occurring in multiple list insertion, Eq. 5.2.1–11, when $M = N/\alpha$, for fixed α as $N \to \infty$. Carry out the expansion to terms of $O(N^{-1})$, expressing your answer in terms of the "exponential integral" function $E_1(z) = \int_z^\infty e^{-t}\, dt/t$.

5.2.3. Sorting by Selection

Another important family of sorting techniques is based on the idea of repeated selection. The simplest selection method is perhaps the following:

i) Find the smallest key; transfer the corresponding record to the output area; then replace the key by the value "∞" (which is assumed to be higher than any actual key).
ii) Repeat step (i). This time the second smallest key will be selected, since the smallest key has been replaced by ∞.
iii) Continue repeating step (i) until N records have been selected.

Note that such a selection method requires all the input items to be present before sorting may proceed, and it generates the final outputs one by one in sequence. This is essentially the opposite of insertion, where the inputs are received sequentially but we do not know any of the final outputs until sorting is completed.

Some computers (e.g., those with a cyclic drum memory) have a built-in "find the smallest" instruction which operates at high speed. This makes selection sorting by the above method especially attractive when N is not too large.

The above method involves $N - 1$ comparisons each time a new record is selected, and it also requires a separate output area in memory. There is an obvious way to improve upon the situation, avoiding the use of ∞: We can take the selected value and move it into its proper position by exchanging it with the record currently occupying that position. Then we need not consider that position again in future selections. This idea yields our first selection sorting algorithm.

Algorithm S (*Straight selection sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \leq \cdots \leq K_N$. Sorting is based on the method indicated above, except that it proves to be more convenient to select the *largest* element first, then the second largest, etc.

S1. [Loop on j.] Perform steps S2 and S3 for $j = N, N - 1, \ldots, 2$.

S2. [Find max $(K_1, \ldots, K_j)$.] Search through keys $K_j, K_{j-1}, \ldots, K_1$ to find a maximal one; let it be K_i.

S3. [Exchange with R_j.] Interchange records $R_i \leftrightarrow R_j$. (Now records $R_j, \ldots, R_N$ are in their final position.) ∎

Table 1 shows this algorithm in action on our sixteen example keys; elements which are candidates for the maximum during the search in step S2 are shown in boldface type.

Table 1

STRAIGHT SELECTION SORTING

503	087	512	061	**908**	170	**897**	275	653	426	154	509	612	677	**765**	**703**
503	087	512	061	703	170	**897**	275	653	426	154	509	612	677	**765**	908
503	087	512	061	703	170	**765**	275	653	426	154	509	612	**677**	897	908
503	087	512	061	**703**	170	**677**	275	**653**	426	154	509	**612**	765	897	908
503	087	512	061	612	170	**677**	275	**653**	426	154	**509**	703	765	897	908
503	087	512	061	612	170	509	275	**653**	**426**	**154**	677	703	765	897	908
...															
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

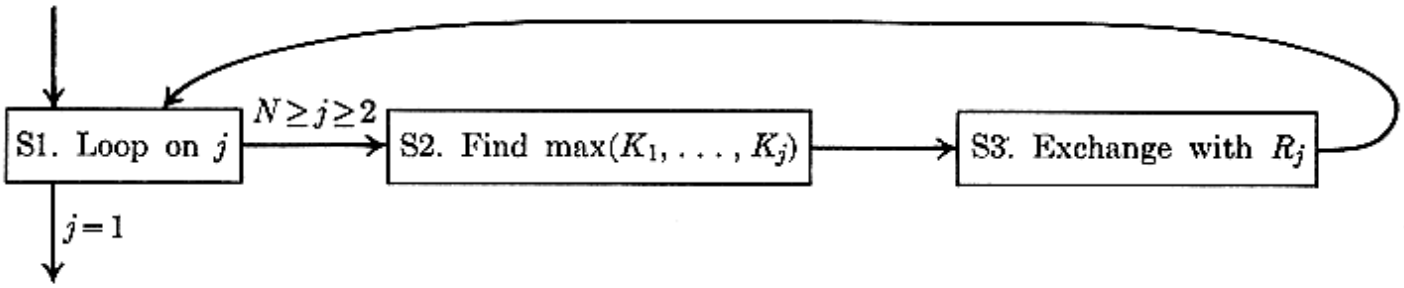


Fig. 21. Straight selection sorting.

The corresponding MIX program is quite simple:

Program 5.2.3S (*Straight selection sort*). As in previous programs of this chapter, the records in locations INPUT+1 through INPUT+N are sorted in place, on a full-word key. rA $\equiv$ current maximum, rI1 $\equiv j - 1$, rI2 $\equiv k$ (index for searching), rI3 $\equiv i$. Assume that $N \geq 2$.

```
01   START  ENT1  N-1          1        S1. Loop on j. j ← N.
02   2H     ENT2  0,1          N − 1    S2. Find max (K1, ..., Kj). k ← j − 1.
03          ENT3  1,1          N − 1    i ← j.
04          LDA   INPUT,3      N − 1    rA ← Ki.
05   8H     CMPA  INPUT,2      A
06          JGE   *+3          A        Jump if Ki ≥ Kk.
07          ENT3  0,2          B        Otherwise set i ← k,
08          LDA   INPUT,3      B        rA ← Ki.
09          DEC2  1            A        k ← k − 1.
10          J2P   8B           A        Repeat if k > 0.
11          LDX   INPUT+1,1    N − 1    S3. Exchange with Rj.
12          STX   INPUT,3      N − 1    Ri ← Rj.
13          STA   INPUT+1,1    N − 1    Rj ← rA.
14          DEC1  1            N − 1
15          J1P   2B           N − 1    N ≥ j ≥ 2. ▮
```

The running time of this program depends on the number of items; N; the number of comparisons, A; and the number of "right-to-left maxima," B. It is easy to see that

$$A = \binom{N}{2} = \frac{1}{2}N(N-1), \tag{1}$$

regardless of the values of the input keys; hence only B is variable. In spite of the simplicity of straight selection, this quantity B is not easy to analyze precisely. Exercises 3 through 6 show that

$$B = (\min 0,\ \text{ave}\ (N+1)H_N - 2N,\ \max \lfloor N^2/4 \rfloor); \tag{2}$$

in this case the maximum value turns out to be particularly interesting. (The standard deviation of B has not yet been determined.)

Thus the average running time of Program S is $2.5N^2 + 3(N+1)H_N + 3.5N - 11$ units, just slightly slower than straight insertion (Program 5.2.1S). It is interesting to compare Algorithm S to the bubble sort (Algorithm 5.2.2B), since bubble sorting may be regarded as a selection algorithm which sometimes selects more than one element at a time. For this reason bubble sorting usually does less comparisons than straight selection and it may seem to be preferable; but in fact Program 5.2.2B is more than twice as slow as Program S! Bubble sorting is handicapped by the fact that it does so many exchanges, while straight selection involves very little data movement.

Refinements of straight selection. Is there any way to improve on the selection method used in Algorithm S? For example, take the search for a maximum in step S2; is there a substantially faster way to find a maximum? The answer to the latter question is *no*!

Lemma M. *Every algorithm for finding the maximum of n elements, based on comparing pairs of elements, must make at least $n - 1$ comparisons.*

Proof. If we have made less than $n - 1$ comparisons, there will be at least two elements which have never been found to be less than any others. Therefore we will not know which of these two elements is larger, and we cannot have determined the maximum. ∎

Thus a selection process which finds the largest element must take at least $n - 1$ steps; perhaps all sorting methods based on n repeated selections are doomed to require order n^2 steps? Fortunately Lemma M applies only to the *first* selection step; subsequent selections can make use of previously-gained information. For example, exercise 8 shows that a comparatively simple change to Algorithm S cuts the average number of comparisons in half.

Consider the 16 numbers in Table 1; one way to save time on repeated selections is to regard them as four groups of four. We can start by determining the largest of each group, namely the respective keys

512, 908, 653, 765;

the largest of these four elements, 908, is then the largest of the entire file. To get the second largest we need only look at the other three elements of the group containing 908; the largest of {170, 897, 275} is 897, and the largest of

$$512,\ 897,\ 653,\ 765$$

is 897. Similarly to get the third largest element we determine the largest of {170, 275} and then the largest of

$$512,\ 275,\ 653,\ 765,$$

and so on. Each selection after the first takes at most 6 additional comparisons. In general, if N is a perfect square, we can divide the file into $\sqrt{N}$ groups of $\sqrt{N}$ elements each; each selection after the first takes at most $\sqrt{N} - 1$ comparisons within the group of the previously selected item, plus $\sqrt{N} - 1$ comparisons among the "group-leaders." This idea is called *quadratic selection*; its total execution time is $O(N\sqrt{N})$, which is substantially better than order N^2.

Quadratic selection was first published by E. H. Friend [*JACM* **3** (1956), 152–154], who pointed out that the same idea can be generalized to cubic, quartic, etc. selection. For example, cubic selection divides the file into $\sqrt[3]{N}$ large groups, each containing $\sqrt[3]{N}$ small groups, each containing $\sqrt[3]{N}$ records; the execution time is proportional to $N\sqrt[3]{N}$. If we carry this idea to its ultimate conclusion we arrive at what Friend called "nth degree selecting," based on a binary tree structure. This method has an execution time proportional to $N \log N$; we shall call it *tree selection*.

Tree selection. The principles of tree selection sorting are easy to understand in terms of matches in a typical "knock-out tournament." Consider, for example, the results of the ping-pong contest shown in Fig. 22; Jim beats Don and Joe beats Jack, then in the next round Joe beats Jim, etc.

Figure 22 shows that Joe is the champion of the eight players, and $8 - 1 = 7$ matches (i.e., comparisons) were required to determine this fact. Dick is not necessarily the second-best player; any of the people defeated by Joe, including the first-round loser Jack, might possibly be second best. We can determine the second-best player by having Jack play Jim, and the winner of that match plays Dick; only two additional matches are required to find the second-best man, because of the structure we have remembered from the earlier games.

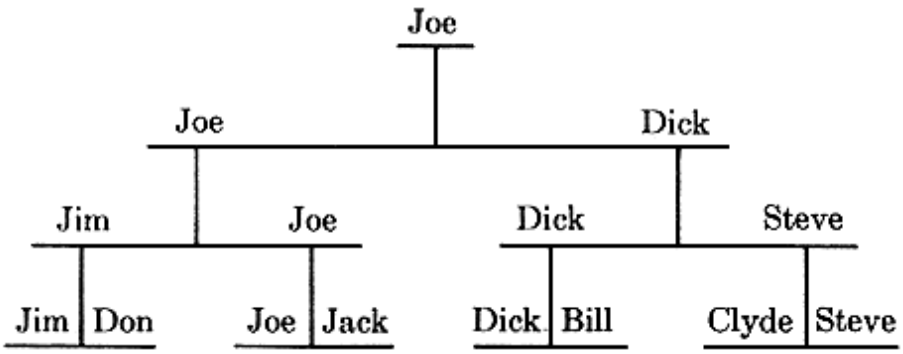
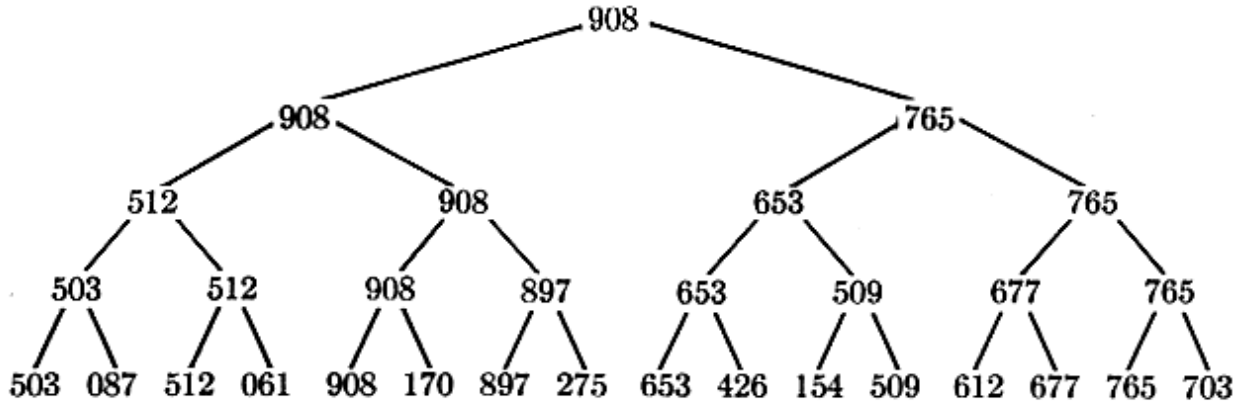


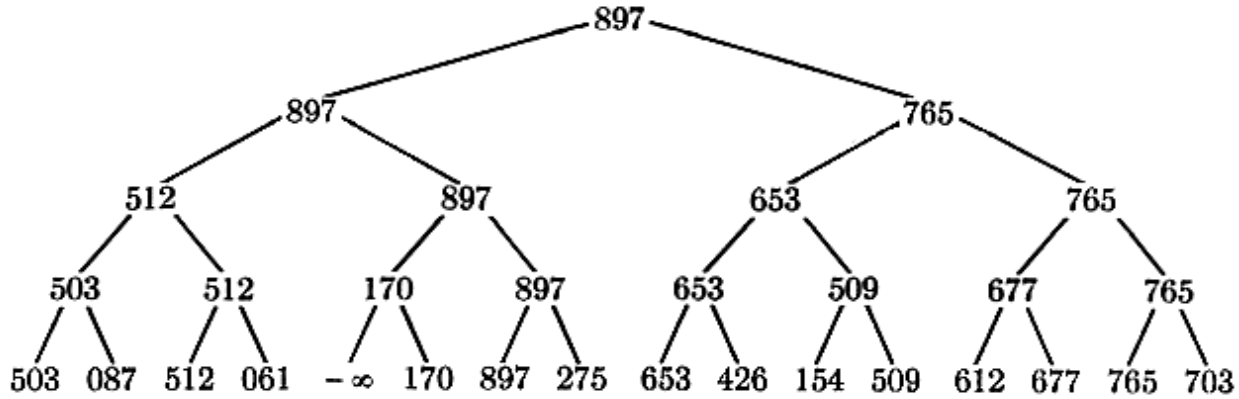
Fig. 22. A ping-pong tournament.

In general, we can "output" the player at the root of the tree, and replace him by an extremely weak player. Substituting this weak player means that the original second-best player will now be best, so he will appear at the root if we recalculate the winners in upper levels of the tree. For this purpose only one path in the tree must be changed, so that less than $\lceil \log_2 N \rceil$ further comparisons are needed to select the next-best player. The total time for such a selection sort is roughly proportional to $N \log N$.

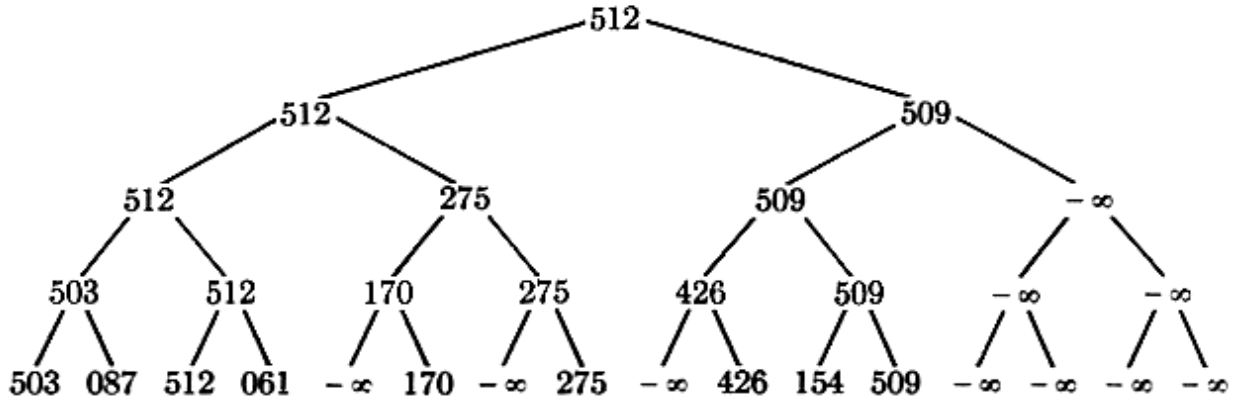
Figure 23 shows tree selection sorting in action, on our 16 example numbers. Note that it is necessary to know where the key at the root came from, in order to know where to insert the next "$-\infty$". Therefore the branch nodes



(a) Initial configuration.



(b) 908 is replaced by $-\infty$, and the second highest element moves up to the root.



(c) Configuration after 908, 897, 765, 703, 677, 653, 612 have been output.

Fig. 23. An example of tree selection sorting.

of the tree actually contain a pointer or index specifying the position of the relevant key, instead of the key itself. It follows that we need memory space for N input records, $N - 1$ pointer words, and N output records. (If the output goes to tape or disk, of course, we don't need to retain the output records in high-speed memory.)

The reader should pause at this point until he understands tree selection, in order to appreciate the remarkable improvements we are about to discuss. For example, it should now be easy to work exercise 10.

One way to modify tree selection, essentially introduced by K. E. Iverson [*A Programming Language*, (Wiley, 1962), 223–227], does away with the need for pointers by "looking ahead" in the following way: When the winner of a match in the bottom level of the tree is moved up, he can be replaced immediately by "$-\infty$" at the bottom level; and when a winner moves up from one branch to another, we can replace him by the one which eventually should move up into his former place (namely the larger of the two keys below). Repeating this operation as often as possible converts Fig. 23(a) into Fig. 24.

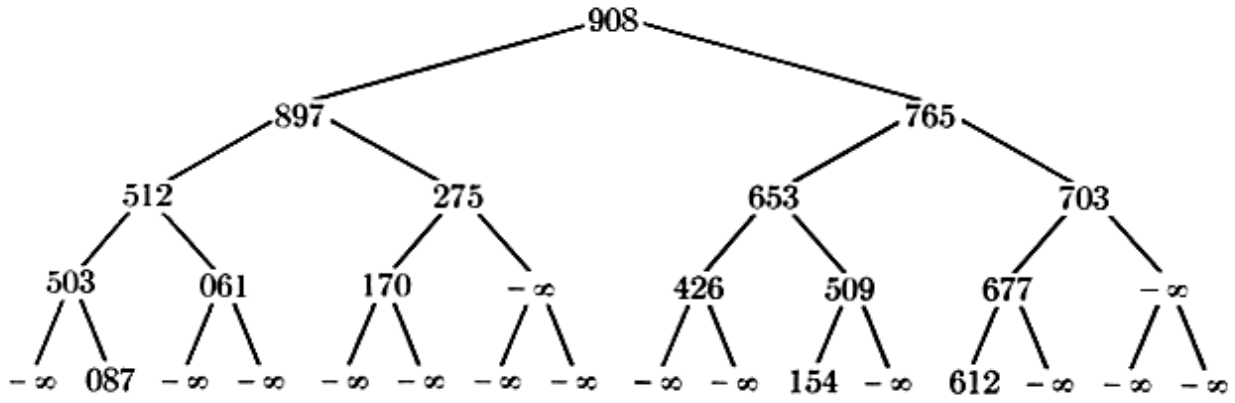


Fig. 24. The Peter Principle applied to sorting. Everyone rises to his level of incompetence in the hierarchy.

Once the tree has been set up in this way we can proceed to sort by a "top-down" method, instead of the "bottom up" method of Fig. 23; we output the root, move up his largest descendant, move up the latter's largest descendant, etc. The process begins to look less like a ping-pong tournament and more like a corporate system of promotions.

The reader should be able to see that this top-down method has the advantage that redundant comparisons of $-\infty$ with $-\infty$ can be avoided. (The bottom-up approach finds $-\infty$ omnipresent in the latter stages of sorting, but the top-down approach can stop modifying the tree during each stage as soon as a $-\infty$ has been stored.)

Figures 23 and 24 are *complete binary trees* with 16 terminal nodes (cf. Section 2.3.4.5), and it is convenient to represent such a tree in consecutive locations as shown in Fig. 25. Note that the father of node number k is node $\lfloor k/2 \rfloor$, and its sons are nodes $2k$ and $2k + 1$. This leads to another advantage of the top-down approach, since it is often considerably simpler to go top-down from node k

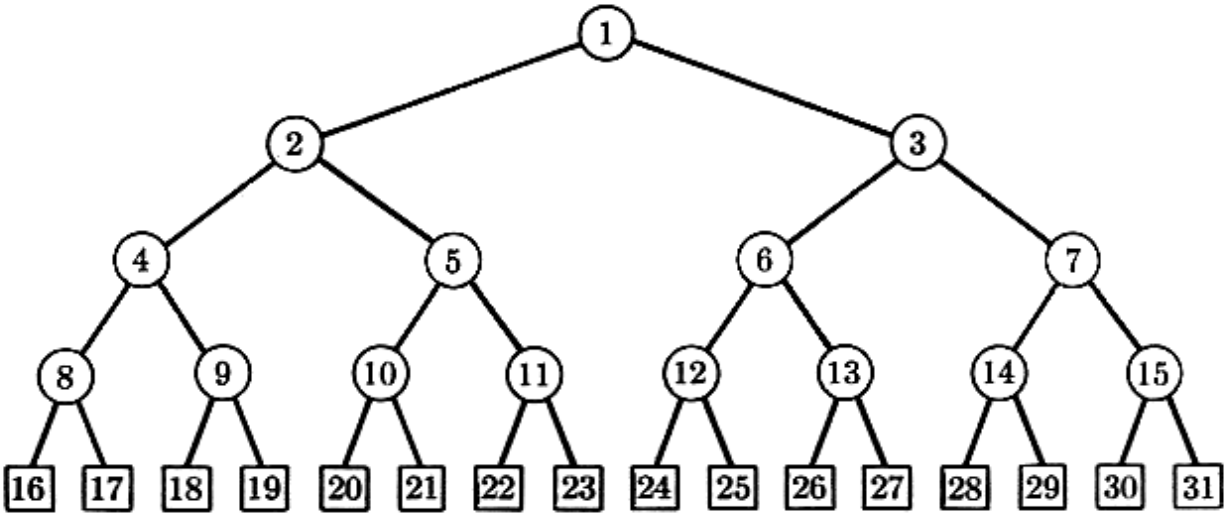


Fig. 25. Sequential storage allocation for a complete binary tree.

to nodes $2k$ and $2k + 1$ than bottom-up from node k to nodes $k \oplus 1$ and $\lfloor k/2 \rfloor$. (Here $k \oplus 1$ stands for $k + 1$ or $k - 1$, according as k is even or odd.)

Our examples of tree selection so far have more or less assumed that N is a power of 2; but actually we can work with arbitrary N, since the complete binary tree with N terminal nodes is readily constructed for any N.

Now we come to the crucial question: Can't we do the top-down method without using "$-\infty$" at all? Wouldn't it be nice if the important information of Fig. 24 were all in locations 1 through 16 of the complete binary tree, without the useless "holes" containing $-\infty$? Some reflection shows that it is indeed possible to achieve this goal, not only eliminating $-\infty$ but also making it possible to sort N records in place without an auxiliary output area. This brings us to an important sorting algorithm which was christened "heapsort" by its discoverer J. W. J. Williams [*CACM* **7** (1964), 347–348].

Heapsort. Let us say that a file of keys $K_1, K_2, \ldots, K_N$ is a "heap" if

$$K_{\lfloor j/2 \rfloor} \geq K_j \qquad \text{for} \qquad 1 \leq \lfloor j/2 \rfloor < j \leq N. \tag{3}$$

Thus, $K_1 \geq K_2$, $K_1 \geq K_3$, $K_2 \geq K_4$, etc.; this is exactly the condition which holds in Fig. 24, and it implies in particular that the largest key appears "on top of the heap,"

$$K_1 = \max (K_1, K_2, \ldots, K_N). \tag{4}$$

If we can somehow transform an arbitrary input file into a heap, we can use a "top-down" selection procedure like that described above to obtain an efficient sorting algorithm.

An efficient approach to heap creation has been suggested by R. W. Floyd [*CACM* **7** (1964), 701]. Let us assume that we have been able to arrange the file so that

$$K_{\lfloor j/2 \rfloor} \geq K_j \qquad \text{for} \qquad l < \lfloor j/2 \rfloor < j \leq N, \tag{5}$$

where l is some number ≥ 1. (In the original file this condition holds "vacu-

ously" for $l = \lfloor N/2 \rfloor$, since no subscript j satisfies the condition $\lfloor N/2 \rfloor < \lfloor j/2 \rfloor < j \le N$.) It is not difficult to see how to transform the file so that the inequalities in (5) are extended to the case $\lfloor j/2 \rfloor = l$, working entirely in the subtree whose root is node l. Therefore we can decrease l by 1, until condition (3) is finally achieved. These ideas of Williams and Floyd lead to the following elegant algorithm which merits careful study:

Algorithm H (*Heapsort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \cdots \le K_N$. First we rearrange the file so that it forms a heap, then we repeatedly remove the top of the heap and transfer it to its proper final position. Assume that $N \ge 2$.

H1. [Initialize.] Set $l \leftarrow \lfloor N/2 \rfloor + 1$, $r \leftarrow N$.

H2. [Decrease l or r.] If $l > 1$, set $l \leftarrow l - 1$, $R \leftarrow R_l$, $K \leftarrow K_l$. (If $l > 1$, we are in the process of transforming the input file into a heap; on the other hand if $l = 1$, the keys $K_1 K_2 \ldots K_r$ presently constitute a heap.) Otherwise set $R \leftarrow R_r$, $K \leftarrow K_r$, $R_r \leftarrow R_1$, and $r \leftarrow r - 1$; if this makes $r = 1$, set $R_1 \leftarrow R$ and terminate the algorithm.

H3. [Prepare for "sift-up."] Set $j \leftarrow l$. (At this point we have

$$K_{\lfloor j/2 \rfloor} \ge K_j \qquad \text{for} \qquad l < \lfloor j/2 \rfloor < j \le r; \tag{6}$$

and record R_k is in its final position for $r < k \le N$. Steps H3–H8 are called the "sift-up" algorithm; their effect is equivalent to setting $R_l \leftarrow R$ and then rearranging $R_l, \ldots, R_r$ so that condition (6) holds also for $\lfloor j/2 \rfloor = l$.)

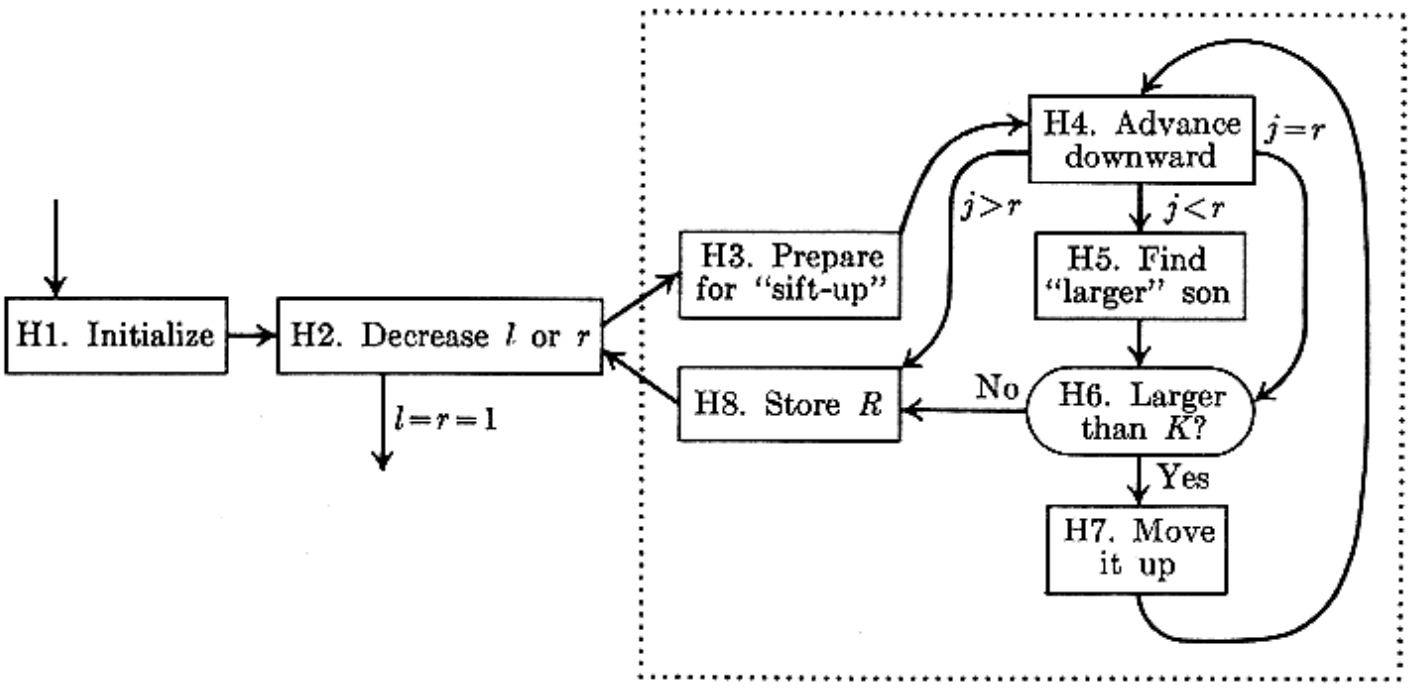


Fig. 26. Heapsort; dotted lines enclose the "sift-up" algorithm.

H4. [Advance downward.] Set $i \leftarrow j$ and $j \leftarrow 2j$. (In the following steps we have $i = \lfloor j/2 \rfloor$.) If $j < r$, go right on to step H5; if $j = r$, go to step H6; and if $j > r$, go to H8.

H5. [Find "larger" son.] If $K_j < K_{j+1}$, then set $j \leftarrow j + 1$.

H6. [Larger than K?] If $K \geq K_j$, then go to step H8.

H7. [Move it up.] Set $R_i \leftarrow R_j$, and go back to step H4.

H8. [Store R.] Set $R_i \leftarrow R$. (This terminates the "sift-up" algorithm initiated in step H3.) Return to step H2. ▮

Heapsort has sometimes been described as the "⋀" algorithm, because of the motion of l and r. The upper triangle represents the heap creation phase, when $r = N$ and l decreases to 1; and the lower triangle represents the selection phase, when $l = 1$ and r decreases to 1. Table 2 shows the process of heapsorting our sixteen example numbers. (Each line in that table shows the state of affairs after step H2, and brackets indicate the position l and r.)

Table 2

EXAMPLE OF HEAPSORT

K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}	K_{16}	l	r	K
503	087	512	061	908	170	897	[275	653	426	154	509	612	677	765	703]	8	16	275
503	087	512	061	908	170	[897	703	653	426	154	509	612	677	765	275]	7	16	897
503	087	512	061	908	[170	897	703	653	426	154	509	612	677	765	275]	6	16	170
503	087	512	061	[908	612	897	703	653	426	154	509	170	677	765	275]	5	16	908
503	087	512	[061	908	612	897	703	653	426	154	509	170	677	765	275]	4	16	061
503	087	[512	703	908	612	897	275	653	426	154	509	170	677	765	061]	3	16	512
503	[087	897	703	908	612	765	275	653	426	154	509	170	677	512	061]	2	16	087
[503	908	897	703	426	612	765	275	653	087	154	509	170	677	512	061]	1	16	503
[908	703	897	653	426	612	765	275	503	087	154	509	170	677	512]	908	1	15	061
[897	703	765	653	426	612	677	275	503	087	154	509	170	061]	897	908	1	14	512
[765	703	677	653	426	612	512	275	503	087	154	509	170]	765	897	908	1	13	061
[703	653	677	503	426	612	512	275	061	087	154	509]	703	765	897	908	1	12	170
[677	653	612	503	426	509	512	275	061	087	154]	677	703	765	897	908	1	11	170
[653	503	612	275	426	509	512	170	061	087]	653	677	703	765	897	908	1	10	154
[612	503	512	275	426	509	154	170	061]	612	653	677	703	765	897	908	1	9	087
[512	503	509	275	426	087	154	170]	512	612	653	677	703	765	897	908	1	8	061
[509	503	154	275	426	087	061]	509	512	612	653	677	703	765	897	908	1	7	170
[503	426	154	275	170	087]	503	509	512	612	653	677	703	765	897	908	1	6	061
[426	275	154	061	170]	426	503	509	512	612	653	677	703	765	897	908	1	5	087
[275	170	154	061]	275	426	503	509	512	612	653	677	703	765	897	908	1	4	087
[170	087	154]	170	275	426	503	509	512	612	653	677	703	765	897	908	1	3	061
[154	087]	154	170	275	426	503	509	512	612	653	677	703	765	897	908	1	2	061
[061]	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908	1	1	061

Program H (*Heapsort*). The records in locations INPUT+1 through INPUT+N are sorted by Algorithm H, with the following register assignments: rI1 $\equiv l - 1$, rI2 $\equiv r - 1$, rI3 $\equiv i$, rI4 $\equiv j$, rI5 $\equiv r - l$, rA $\equiv K \equiv R$, rX $\equiv R_j$.

01	START	ENT1	N/2	1	*H1. Initialize.* $l \leftarrow \lfloor N/2 \rfloor + 1$.
02		ENT2	N-1	1	$r \leftarrow N$.
03	1H	DEC1	1	$\lfloor N/2 \rfloor$	$l \leftarrow l - 1$.
04		LDA	INPUT+1,1	$\lfloor N/2 \rfloor$	$R \leftarrow R_l$, $K \leftarrow K_l$.
05	3H	ENT4	1,1	P	*H3. Prepare for "sift-up."* $j \leftarrow l$.
06		ENT5	0,2	P	
07		DEC5	0,1	P	rI5 $\leftarrow r - j$.
08		JMP	4F	P	To H4.
09	5H	LDX	INPUT,4	$B + A - D$	*H5. Find "larger" son.*
10		CMPX	INPUT+1,4	$B + A - D$	
11		JGE	6F	$B + A - D$	Jump if $K_j \geq K_{j+1}$.
12		INC4	1	C	Otherwise set $j \leftarrow j + 1$.
13		DEC5	1	C	
14	9H	LDX	INPUT,4	$C + D$	rX $\leftarrow R_j$.
15	6H	CMPA	INPUT,4	$B + A$	*H6. Larger than K?*
16		JGE	8F	$B + A$	To H8 if $K \geq K_j$.
17	7H	STX	INPUT,3	B	*H7. Move it up.* $R_i \leftarrow R_j$.
18	4H	ENT3	0,4	$B + P$	*H4. Advance downward.* $i \leftarrow j$.
19		DEC5	0,4	$B + P$	rI5 $\leftarrow$ rI5 $- j$.
20		INC4	0,4	$B + P$	$j \leftarrow j + j$.
21		J5P	5B	$B + P$	To H5 if $j < r$.
22		J5Z	9B	$P - A + D$	To H6 if $j = r$.
23	8H	STA	INPUT,3	P	*H8. Store R.* $R_i \leftarrow R$.
24	2H	J1P	1B	P	*H2. Decrease l or r.*
25		LDA	INPUT+1,2	$N - 1$	If $l = 1$, set $R \leftarrow R_r$, $K \leftarrow K_r$.
26		LDX	INPUT+1	$N - 1$	
27		STX	INPUT+1,2	$N - 1$	$R_r \leftarrow R_1$.
28		DEC2	1	$N - 1$	$r \leftarrow r - 1$.
29		J2P	3B	$N - 1$	To H3 if $r > 1$.
30		STA	INPUT+1	1	$R_1 \leftarrow R$. ▮

This program is only about twice as long as Program S, but it is much more efficient when N is large. Its running time depends on

$P = N + \lfloor N/2 \rfloor - 2$, the number of sift-up passes;

A, the number of sift-up passes in which the key K finally lands in an interior node of the heap;

B, the total number of keys scanned during sift-ups;

C, the number of times $j \leftarrow j + 1$ in step H5; and

D, the number of times $j = r$ in step H4.

These quantities are analyzed below; in practice they show comparatively little fluctuation about their average values,

$$A \approx 0.349N, \qquad B \approx N \log_2 N - 1.87N,$$
$$C \approx \tfrac{1}{2}N \log_2 N - 0.9N, \qquad D \approx \ln N. \tag{7}$$

For example, when $N = 1000$, four experiments on random input gave $(A, B, C, D) = (371, 8055, 4056, 12)$, $(351, 8072, 4087, 14)$, $(341, 8094, 4017, 8)$, $(340, 8108, 4083, 13)$, respectively. The total running time, $7A + 14B + 4C + 20N - 2D + 15\lfloor N/2 \rfloor - 28$, is therefore approximately $16N \log_2 N + 0.2N$ units on the average.

A glance at Table 2 makes it had to believe that heapsort is very efficient; large keys shift 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 $1068u$, while the simple method of straight insertion (Program 5.2.1S) takes only $514u$. Straight selection (Program S) takes $853u$.

For larger N, Program H is more efficient. It invites comparison with Shell's diminishing increment sort (Program 5.2.1D) and Hoare's 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 are

$160000u$ for heapsort,

$130000u$ for Shell's sort,

$80000u$ for quicksort.

(MIX is typical of most present-day computers, but particular machines will of course yield somewhat different relative values.) As N gets larger, heapsort will be superior to Shell's sort, but the asymptotic formula $16N \log_2 N \approx 23.08N \ln N$ will never beat quicksort's $12.67N \ln N$. A modification of heapsort discussed in exercise 18 will speed up the process on many computers, but even this improvement falls short of quicksort.

On the other hand, quicksort is efficient only on the average, and its worst case is of order N^2. Heapsort has the interesting property that its worst case isn't much worse than the average: We always have

$$A \le 1.5N, \qquad B \le N\lfloor \log_2 N \rfloor, \qquad C \le N\lfloor \log_2 N \rfloor, \tag{8}$$

so Program H will take no more than $18N\lfloor \log_2 N \rfloor + 38N$ units of time, regardless of the distribution of the input data. Heapsort is the first sorting method we have seen which is *guaranteed* to be of order $N \log N$. Merge sorting, discussed in Section 5.2.4 below, also has this property, but it requires more memory space.

Largest in, first out. We have seen in Chapter 2 that linear lists can often be classified in a meaningful way by the nature of the insertion and deletion operations performed on them. A *stack* has "last in, first out" behavior, in the sense that every deletion removes the youngest item in the list (the item which was inserted most recently of all items currently present). A simple *queue* has

"first in, first out" behavior, in the sense that every deletion removes the oldest remaining item. In more complex situations, such as the elevator simulation of Section 2.2.5, we want a "smallest in, first out" list, where every deletion removes the item having the smallest key. Such a list may be called a *priority queue*, since the key of each item reflects its relative ability to get out of the list quickly. Selection sorting is a special case of a priority queue in which we do N insertions followed by N deletions.

Priority queues arise in a wide variety of applications. For example, some numerical iterative schemes are based on repeated selection of an item having the largest (or smallest) value of some test criterion; parameters of the selected item are changed, and it is reinserted into the list with a new test value, based on the new values of its parameters. Operating systems often make use of priority queues for the scheduling of jobs. Exercises 15, 29, and 36 mention other typical applications of priority queues, and many other examples will appear in later chapters.

How shall we implement priority queues? One of the obvious methods is to maintain a sorted list, containing the items in order of their keys. Inserting a new item is then essentially the same problem we have treated in our study of insertion sorting, Section 5.2.1. Another even more obvious way to deal with priority queues is to keep the list of elements in arbitrary order, selecting the appropriate element each time a deletion is required by finding the largest (or smallest) key. The trouble with both of these obvious approaches is that they require on the order of N steps either for insertion or deletion, when there are N entries in the list, so they are very time-consuming when N is large.

In his original paper on heapsorting, Williams pointed out that heaps are ideally suited to large priority queue applications, since we can insert or delete elements from a heap in $O(\log N)$ steps; furthermore, all elements of the heap are compactly located in consecutive memory locations. The selection phase of Algorithm H is a sequence of deletion steps of a *largest-in-first-out* process: to delete the largest element K_1 we remove it and "sift-up" K_N into a new heap of $N - 1$ elements. (If we want a smallest-in-first-out algorithm, as in the elevator simulation, we can obviously change the definition of heap so that "$\geq$" becomes "$\leq$" in (3); for convenience, we shall consider only the largest-in-first-out case here.) In general, if we want to delete the largest item and then insert a new element x, we can do the sift-up procedure with $l = 1$, $r = N$, and $K = x$. If we wish to insert an element x without a prior deletion, we can use the "bottom-up" procedure of exercise 16.

A linked representation for priority queues. An efficient way to represent priority queues as linked binary trees was discovered in 1971 by Clark A. Crane. His method requires two link fields and a small count field in every record, but it has the following advantages over a heap:

1) When the priority queue is being treated as a stack, the insertion and deletion operations are more efficient (they take a fixed time independent of the queue size).

2) The records never move, only the pointers change.

3) It is possible to merge two disjoint priority queues, having a total of N elements, into a single priority queue, in only $O(\log N)$ steps.

Crane's original method, slightly modified, is illustrated in Fig. 27, which shows a special kind of binary tree structure. Each node contains a KEY field, a DIST field, and two link fields LEFT and RIGHT. The DIST field is always set equal to the length of the shortest path from that node to a leaf (i.e., an empty Λ node) of the tree. If we regard DIST(Λ) = 0 and KEY(Λ) = $-\infty$, the KEY and DIST fields in the tree satisfy the following properties:

$$\mathtt{KEY(P)} \geq \mathtt{KEY(LEFT(P))}, \qquad \mathtt{KEY(P)} \geq \mathtt{KEY(RIGHT(P))}; \tag{9}$$

$$\mathtt{DIST(P)} = 1 + \min\,(\mathtt{DIST(LEFT(P))}, \mathtt{DIST(RIGHT(P))}); \tag{10}$$

$$\mathtt{DIST(LEFT(P))} \geq \mathtt{DIST(RIGHT(P))}. \tag{11}$$

Relation (9) is analogous to the heap condition (3), it guarantees that the root of the tree has the largest key; and relation (10) is just the definition of the DIST fields as stated above. Relation (11) is the interesting innovation: It implies that a shortest path to a leaf may always be obtained by moving to the right. We shall say that a binary tree with this property is a *leftist tree*, because it tends to "lean" so heavily to the left.

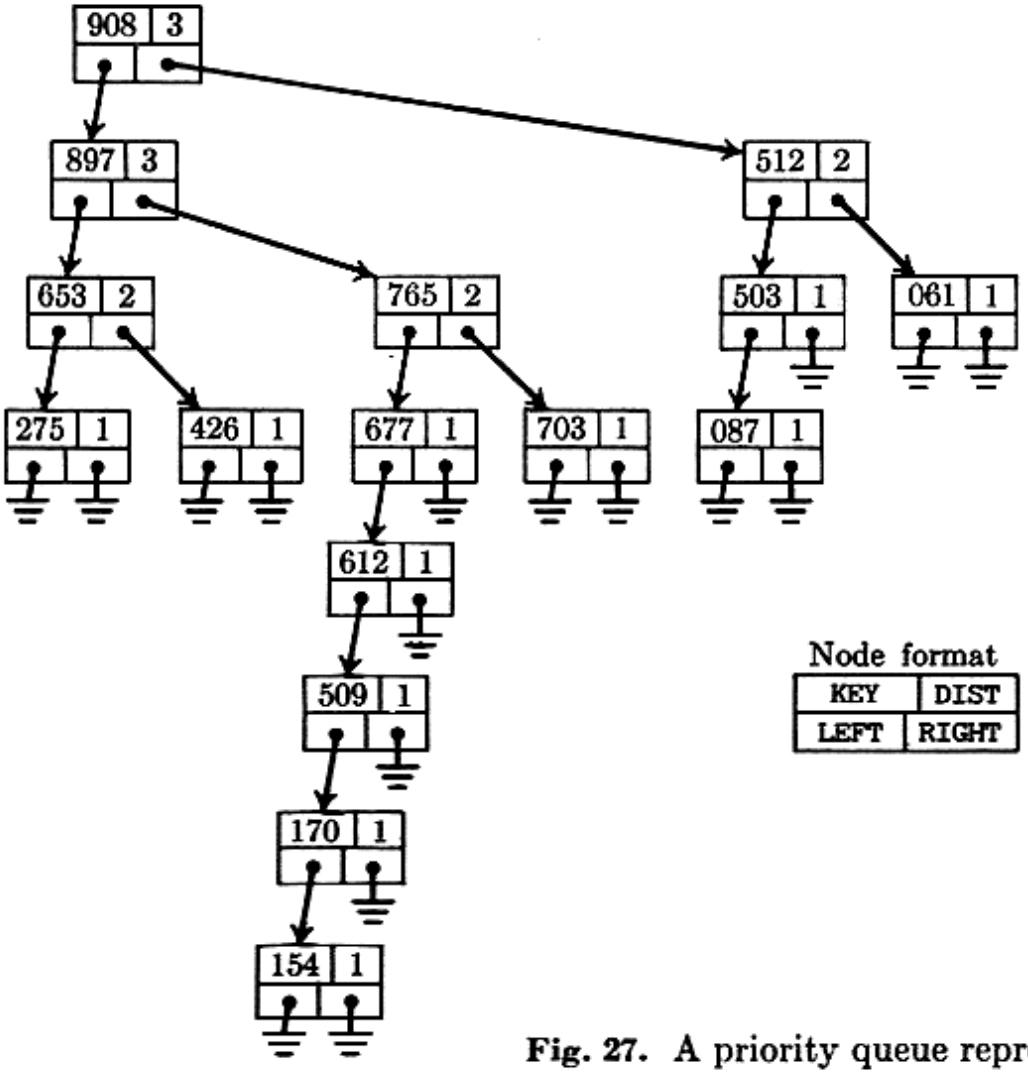


Fig. 27. A priority queue represented as a leftist tree.

It is clear from these definitions that DIST(P) $= n$ implies the existence of at least 2^n leaf nodes below P; otherwise there would be a shorter path from P to a leaf. Thus, if there are N nodes in a leftist tree, the path leading downward from the root towards the right contains at most $\lfloor \log_2 (N + 1) \rfloor$ nodes. It is possible to insert a new node into the priority queue by traversing this path (see exercise 32); hence only $O(\log N)$ steps are needed in the worst case. The best case occurs when the tree is linear (all RIGHT links are Λ), and the worst case occurs when the tree is perfectly balanced.

To remove the node at the root, we simply need to merge its two subtrees. The operation of merging two disjoint leftist trees, pointed to respectively by P and Q, is conceptually simple: If KEY(P) $\geq$ KEY(Q) we take P as the root and merge Q with P's right subtree; then DIST(P) is updated, and LEFT(P) is interchanged with RIGHT(P) if necessary. A detailed description of this process is not difficult to devise (see exercise 32).

Comparison of priority queue techniques. When the number of nodes, N, is small, it is best to use one of the straightforward linear list methods to maintain a priority queue; but when N is large, a $\log N$ method is obviously much faster. Therefore large priority queues are generally represented as heaps or as leftist trees. We shall discuss another way to represent linear lists in Section 6.2.3, via *balanced trees*, and this leads to a third $\log N$ method suitable for representing priority queues. It is therefore appropriate to compare these three techniques.

We have seen that leftist tree operations tend to be slightly faster than heap operations, although heaps take less memory space. Balanced trees take about the same space as leftist trees, perhaps slightly less; the operations are slower than heaps, and the programming is more complicated, but the balanced tree structure is considerably more flexible in several ways. When using a heap or a leftist tree we cannot predict very easily what will happen to two items with equal keys; it is impossible to guarantee that items with equal keys will be treated in a last-in-first-out or first-in-first-out manner, unless the key is extended to include an additional "serial number of insertion" field so that no equal keys are really present. With balanced trees, on the other hand, we can easily stipulate consistent conventions about equal keys, and we can also do things such as "insert x immediately before (or after) y." Balanced trees are symmetrical so that we can delete either the largest or the smallest element at any time, while heaps and leftist trees must be oriented one way or the other. (See exercise 31, however, which shows how to construct *symmetrical* heaps.) Balanced trees can be used for searching as well as for sorting; and we can rather quickly remove consecutive blocks of elements from a balanced tree. But two balanced trees cannot be merged in less than $O(N)$ steps, while leftist trees can be merged in only $O(\log N)$ steps.

In summary, heaps use minimum memory; leftist trees are great for merging disjoint priority queues; and the flexibility of balanced trees is available, if necessary, at reasonable cost.

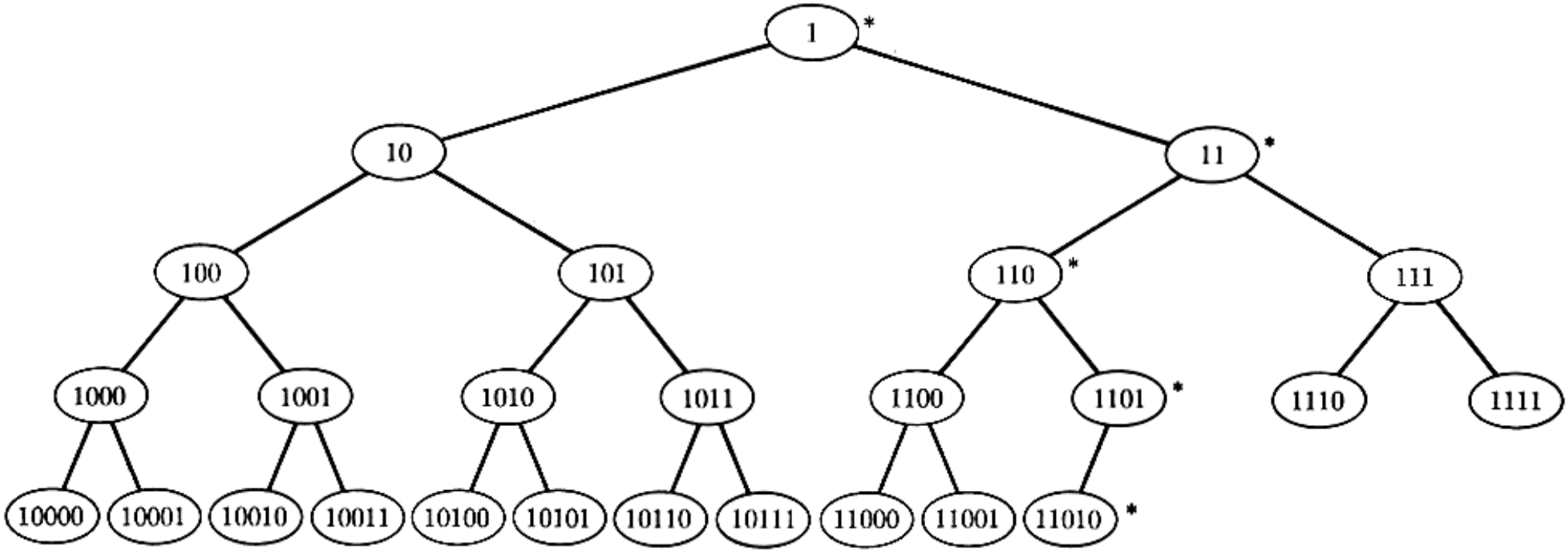


Fig 28. A heap of 26 = $(11010)_2$ elements looks like this.

***Analysis of heapsort.** Algorithm H has not yet been analyzed completely, but several of its properties can be deduced without great difficulty. Therefore we shall conclude this section by studying the anatomy of a heap in some detail.

Figure 28 shows the shape of a heap with 26 elements; each node has been labeled in binary notation corresponding to its subscript in the heap. Asterisks in this diagram denote the so-called *special nodes*, which lie on the path from 1 to N.

One of the most important attributes of a heap is the collection of its subtree sizes. For example, in Fig. 28 the sizes of the subtrees rooted at 1, 2, . . . , 26 are, respectively,

$$26^*, 15, 10^*, 7, 7, 6^*, 3, 3, 3, 3, 3, 3, 2^*, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1^*. \quad (12)$$

Asterisks denote *special subtrees*, rooted at the special nodes; exercise 20 shows that if the binary representation of N is

$$N = (b_n b_{n-1} \ldots b_1 b_0)_2, \qquad n = \lfloor \log_2 N \rfloor, \quad (13)$$

the special subtree sizes are always

$$(1b_{n-1} \ldots b_1 b_0)_2,\ (1b_{n-2} \ldots b_1 b_0)_2, \ldots,\ (1b_1 b_0)_2,\ (1b_0)_2,\ (1)_2. \quad (14)$$

Nonspecial subtrees are always perfectly balanced, so their size is always of the form $2^k - 1$. Exercise 21 shows that the nonspecial sizes consist of exactly

$$\begin{array}{lll} \lfloor (N-1)/2 \rfloor \text{ 1's}, & \lfloor (N-2)/4 \rfloor \text{ 3's}, & \\ \lfloor (N-4)/8 \rfloor \text{ 7's}, & \ldots, & \lfloor (N - 2^{n-1})/2^n \rfloor\ (2^n - 1)\text{'s}. \end{array} \quad (15)$$

For example, Fig. 28 contains twelve nonspecial subtrees of size 1, six of size 3, two of size 7, and one of size 15.

Let s_l be the size of the subtree whose root is l, and let M_N be the multiset $\{s_1, s_2, \ldots, s_N\}$ of all these sizes. We can calculate M_N easily for any given N by using (14) and (15). Exercise 5.1.4–20 tells us that the total number of ways to arrange the integers $\{1, 2, \ldots, N\}$ into a heap is

$$N! \,/\, s_1 s_2 \ldots s_N = N! \,/ \prod_{s \in M_N} s. \quad (16)$$

For example, the number of ways to place the 26 letters $\{A, B, C, \ldots, Z\}$ into Fig. 28 so that vertical lines preserve alphabetic order is

$$26!/(26 \cdot 10 \cdot 6 \cdot 2 \cdot 1 \cdot 1^{12} \cdot 3^6 \cdot 7^2 \cdot 15^1).$$

We are now in a position to analyze the heap-creation phase of Algorithm H, namely, the computations which take place before the condition $l = 1$

occurs for the first time in step H2. Fortunately we can reduce the study of heap creation to the study of independent sift-up operations, because of the following theorem.

Theorem H. *If the input to Algorithm H is a random permutation of* $\{1, 2, \ldots, N\}$, *then each of the* $N!/(\Pi_{s \in M_N} s)$ *possible heaps is an equally likely outcome of the heap creation phase. Moreover, each of the* $\lfloor N/2 \rfloor$ *sift-up operations performed during this phase is "uniform," in the sense that each of the* s_l *possible values of* i *is equally likely when step H8 is reached.*

Proof. We can apply what numerical analysts might call a "backwards analysis"; given a possible result $K_1 \ldots K_N$ of the sift-up operation rooted at l, we see that there are exactly s_l prior configurations $K'_1 \ldots K'_N$ of the file which will siftup to that result. Each of these prior configurations has a different value of K'_l; hence, working backwards, there are exactly $s_l s_{l+1} \ldots s_N$ input permutations of $\{1, 2, \ldots, N\}$ which yield the configuration $K_1 \ldots K_N$ after the siftup at position l has been completed.

The case $l = 1$ is typical: Let $K_1 \ldots K_N$ be a heap, and let $K'_1 \ldots K'_N$ be a file which is transformed by siftup into $K_1 \ldots K_N$ when $l = 1$, $K = K'_1$. If $K = K_i$, we must have $K'_i = K_{\lfloor i/2 \rfloor}$, $K'_{\lfloor i/2 \rfloor} = K_{\lfloor i/4 \rfloor}$, etc., while $K'_j = K_j$ for all j not on the path from 1 to i. Conversely, for each i this construction yields a file $K'_1 \ldots K'_N$ such that (a) sift-up transforms $K'_1 \ldots K'_N$ into $K_1 \ldots K_N$, and (b) $K_{\lfloor j/2 \rfloor} \geq K_j$ for $2 \leq \lfloor j/2 \rfloor < j \leq N$. Therefore exactly N such files $K'_1 \ldots K'_N$ are possible, and the sift-up operation is uniform. (An example of the proof of this theorem appears in exercise 22.) ∎

Referring to the quantities A, B, C, D in the analysis of Program H, we can see that a uniform sift-up operation on a subtree of size s contributes $\lfloor s/2 \rfloor / s$ to the average value of A; it contributes

$$\frac{1}{s}(0 + 1 + 1 + 2 + \cdots + \lfloor \log_2 s \rfloor) = \frac{1}{s}\Bigl(\sum_{1 \leq k \leq s} \lfloor \log_2 k \rfloor\Bigr)$$
$$= \frac{1}{s}\bigl((s+1)\lfloor \log_2 s \rfloor - 2^{\lfloor \log_2 s \rfloor + 1} + 2\bigr)$$

to the average value of B (see exercise 1.2.4–42); and it contributes either $2/s$ or 0 to the average value of D, according as s is even or odd. The corresponding contribution to C is somewhat more difficult to determine, so it has been left to the reader (see exercise 26). Summing over all sift-ups, we find that the average value of A during heap creation is

$$A'_N = \sum_{s \in M_N} \lfloor s/2 \rfloor / s, \tag{17}$$

and similar formulas hold for B, C, and D. It is therefore possible to compute these average values exactly without great difficulty, and the following table shows typical results:

N	A'_N	B'_N	C'_N	D'_N
99	19.18	68.35	42.95	0.00
100	19.93	69.39	42.71	1.84
999	196.16	734.66	464.53	0.00
1000	196.94	735.80	464.16	1.92
9999	1966.02	7428.18	4695.54	0.00
10000	1966.82	7429.39	4695.06	1.97
10001	1966.45	7430.07	4695.84	0.00
10002	1967.15	7430.97	4695.95	1.73

Asymptotically speaking, we may ignore the special subtree sizes in M_N, and we find for example that

$$A'_N = \frac{N}{2}\cdot\frac{0}{1} + \frac{N}{4}\cdot\frac{1}{3} + \frac{N}{8}\cdot\frac{3}{7} + \cdots + O(\log N) = (1 - \tfrac{1}{2}\alpha)N + O(\log N), \tag{18}$$

where

$$\alpha = \sum_{k\geq 1} \frac{1}{2^k - 1} = 1.60669\ 51524\ 15291\ 76378\ 33015\ 23190\ 92458\ 04805-. \tag{19}$$

(This value has been computed by J. W. Wrench, Jr., using the series transformation of exercise 27.) For large N, we may use the approximate formulas

$$\begin{aligned} A'_N &\approx 0.1967N + 0.3\ (N \text{ even}), \qquad 0.1967N - 0.3\ (N \text{ odd});\\ B'_N &\approx 0.74403N - 1.3 \ln N;\\ C'_N &\approx 0.47034N - 0.8 \ln N;\\ D'_N &\approx 1.8 \pm 0.2\ (N \text{ even}), \qquad 0\ (N \text{ odd}). \end{aligned} \tag{20}$$

The minimum and maximum values are also readily determined. Only $O(N)$ steps are needed to create the heap (cf. exercise 23).

This essentially disposes of the heap creation phase of Algorithm H. But the selection phase is another story, which remains to be written! Let A''_N, B''_N, C''_N, and D''_N denote the average values of A, B, C, and D during the selection phase when N elements are being heapsorted. The behavior of Algorithm H on random input is subject to comparatively little fluctuation about the empirically-determined average values

$$\begin{aligned} A''_N &\approx 0.152N;\\ B''_N &\approx N \log_2 N - 2.61N;\\ C''_N &\approx \tfrac{1}{2}N \log_2 N - 1.4N;\\ D''_N &\approx \ln N \pm 2; \end{aligned} \tag{21}$$

but no adequate theoretical explanation for these constants has yet been found. It is not hard to deduce the upper bounds stated in (8), by considering individual sift-up operations, but the upper bound for C should probably be lowered to about $\frac{1}{2}N \log_2 N$ when the algorithm as a whole is considered.

EXERCISES

1. [*10*] Is straight selection (Algorithm S) a "stable" sorting method?

2. [*15*] Why does it prove to be more convenient to select the largest key, then the second-largest, etc., in Algorithm S, instead of first finding the smallest, then the second-smallest, etc.?

3. [*M21*] (a) Prove that if Algorithm S starts with a random permutation of $\{1, 2, \ldots, N\}$, the first iteration of steps S2 and S3 yields a random permutation of $\{1, 2, \ldots, N-1\}$ followed by N. (In other words, the presence of each permutation of $\{1, 2, \ldots, N-1\}$ in $K_1 \ldots K_{N-1}$ is equally likely.) (b) Therefore if B_N denotes the average value of the quantity B in Program S, given randomly ordered input, we have $B_N = H_N - 1 + B_{N-1}$. [*Hint:* Cf. Eq. 1.2.10–16.]

▸ **4.** [*M25*] Step S3 of Algorithm S accomplishes nothing when $i = j$; is it a good idea to test whether or not $i = j$ before doing step S3? What is the average number of times the condition $i = j$ will occur in step S3 for random input?

5. [*20*] What is the value of the quantity B in the analysis of Program S, when the input is $N \ldots 3\ 2\ 1$?

6. [*M29*] (a) Let $a_1\, a_2 \ldots a_N$ be a permutation of $\{1, 2, \ldots, N\}$ having C cycles, I inversions, and B changes to the right-to-left maxima when sorted by Program S. Prove that $2B \leq I + N - C$. [*Hint:* See exercise 5.2.2–1.] (b) Show that $I + N - C \leq \lfloor n^2/2 \rfloor$; hence B can never exceed $\lfloor n^2/4 \rfloor$.

7. [*M46*] Find the variance of the quantity B in Program S, as a function of N, assuming random input.

▸ **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 $K_j, \ldots, K_2, K_1$ as in Program S, it is often possible to reduce the number of comparisons needed on the next iteration of step S2. Write a MIX program based on this observation.

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

10. [*12*] What will be the configuration of the tree in Fig. 23 after 14 of the original 16 items have been output?

11. [*10*] What will be the configuration of the tree in Fig. 24 after the element 908 has been output?

12. [*M20*] How many times will $-\infty$ be compared with $-\infty$ when the "bottom up" method of Fig. 23 is used to sort a file of 2^n elements into order?

13. [*20*] (J. W. J. Williams.) Step H4 of Algorithm H distinguishes between the three cases $j < r$, $j = r$, and $j > r$. Show that if $K \geq K_{r+1}$ it would be possible to simplify step H4 so that only a two-way branch is made. How could the condition $K \geq K_{r+1}$ be ensured throughout the heapsort process, by modfying step H2?

14. [*10*] Show that simple queues are special cases of priority queues. (Explain how keys can be assigned to the elements so that a largest-in-first-out procedure is equivalent to first-in-first-out.) Is a stack also a special case of a priority queue?

▶**15.** [*M22*] (B. A. Chartres.) Design a high-speed algorithm which builds a table of the prime numbers $\leq N$, making use of a *priority queue* to avoid division operations. [*Hint:* Let the smallest key in the priority queue be the least odd nonprime number greater than the last odd number considered as a prime candidate. Try to minimize the number of elements in the queue.]

16. [*20*] Design an efficient algorithm which inserts a new key into a given heap of n elements, producing a heap of $n + 1$ elements.

17. [*20*] The algorithm of exercise 16 can be used for heap creation, instead of the "decrease l to 1" method used in Algorithm H. Do both methods create the same heap when they begin with the same input file?

▶**18.** [*21*] (R. W. Floyd.) During the selection phase of heapsort, the key K tends to be quite small, so that nearly all of the comparisons in step H6 find $K < K_j$. Show how to modify the algorithm so that K is not compared with K_j in the main loop of the computation.

19. [*21*] Design an algorithm which deletes a given element of a heap of length N, producing a heap of length $N - 1$.

20. [*M20*] Prove that (14) gives the special subtree sizes in a heap.

21. [*M24*] Prove that (15) gives the nonspecial subtree sizes in a heap.

▶**22.** [*20*] What permutations of $\{1, 2, 3, 4, 5\}$ are transformed into 5 3 4 1 2 by the heap creation phase of Algorithm H?

23. [*M28*] (a) Prove that the length of scan, B, in a sift-up algorithm never exceeds $\lfloor \log_2 (r/l) \rfloor$. (b) According to (8), B can never exceed $N\lfloor \log_2 N \rfloor$ in any particular application of Algorithm H. Find the maximum value of B as a function of N, taken over all possible input files. (You must prove that an input file exists such that B takes on this maximum value.)

24. [*M24*] Derive an exact formula for the standard deviation of B'_N (the total length of scan during the heap-creation phase of Algorithm H).

25. [*M20*] What is the average value of the contribution to C made during the sift-up pass when $l = 1$ and $r = N$, if $N = 2^{n+1} - 1$?

26. [*M30*] Solve exercise 25, (a) for $N = 26$, (b) for general N.

27. [*M25*] (J. W. Wrench, Jr.) Prove that $\sum_{n\geq 1} x^n/(1 - x^n) = \sum_{n\geq 1} x^{n^2}(1 + x^n)/(1 - x^n)$. (Setting $x = \frac{1}{2}$ gives a very rapidly converging series for evaluating (19).)

28. [*35*] Explore the idea of *ternary heaps*, based on complete ternary trees instead of binary trees. Do ternary heaps sort faster than binary heaps?

29. [*26*] (W. S. Brown.) Design an algorithm for multiplication of polynomials or power series $(a_i x^{i_1} + a_2 x^{i_2} + \cdots)(b_1 x^{j_1} + b_2 x^{j_2} + \cdots)$, in which the coefficients of the answer $c_1 x^{i_1+j_1} + \cdots$ are generated in order as the input coefficients are being multiplied. [*Hint:* Use an appropriate priority queue.]

30. [*M48*] Can C ever exceed $\frac{1}{2}N \log_2 N$ when a file is heapsorted? What is the maximum value of C? What is the minimum value?

31. [*37*] (J. W. J. Williams.) Show that if two heaps are placed "back to back" in a suitable way, it is possible to maintain a structure in which either the smallest or the largest element can be deleted at any time in $O(\log n)$ steps. (Such a structure may be called a *priority deque*.)

32. [*21*] Design an algorithm which merges two disjoint priority queues, represented as leftist trees, into one. (In particular, if one of the given queues contains a single element, your algorithm will insert it into the other queue.)

▶**33.** [*15*] In a priority queue represented as a leftist tree, why is the operation of removing the root done by merging the two subtrees, instead of by "promoting" nodes toward the top as in a heap?

34. [*M47*] How many leftist trees with N nodes are possible, ignoring the KEY values? [The sequence begins 1, 1, 2, 4, 8, 17, 38, . . . ; is there a simple asymptotic formula?]

35. [*26*] If UP links are added to a leftist tree (cf. the discussion of triply-linked trees in Section 6.2.3), it is possible to delete an arbitrary node P from within the priority queue as follows: Replace P by the merger of LEFT(P) and RIGHT(P); then adjust the DIST fields of P's ancestors until either reaching the root or reaching a node whose DIST is unchanged.

Prove that this process never requires changing more than $O(\log N)$ of the DIST fields, if there are N nodes in the tree, even though the tree may contain very long upward paths.

36. [*18*] (*Least-recently-used page replacement.*) Many operating systems make use of the following type of algorithm: A collection of nodes is subjected to two operations, "using" a node, and replacing the least-recently-"used" node by a new node. What data structure makes it easy to ascertain the least-recently-"used" node?

5.2.4. Sorting by Merging

Merging (or *collating*) means the combination of two or more ordered files into a single ordered file. For example, we can merge the two files 503 703 765 and 087 512 677 to obtain 087 503 512 677 703 765. A simple way to accomplish this is to compare the two smallest items, output the smallest, and then repeat the same process. Starting with

$$\begin{cases} 503 & 703 & 765 \\ 087 & 512 & 677 \end{cases}$$

we obtain

$$087 \begin{cases} 503 & 703 & 765 \\ 512 & 677 & \end{cases},$$

then

$$087 \quad 503 \begin{cases} 703 & 765 \\ 512 & 677 \end{cases},$$

and so on. Some care is necessary when one of the two files becomes exhausted; a detailed description of the process appears in the following algorithm:

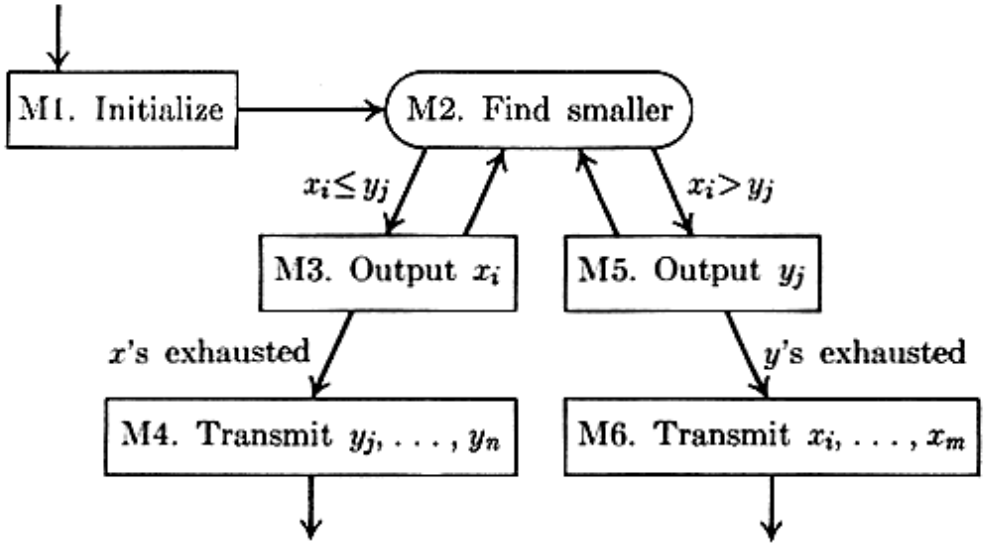


Fig. 29. Merging $x_1 \le \cdots \le x_m$ with $y_1 \le \cdots \le y_n$.

Algorithm M (*Two-way merge*). This algorithm merges the ordered files $x_1 \le x_2 \le \cdots \le x_m$ and $y_1 \le y_2 \le \cdots \le y_n$ into a single file $z_1 \le z_2 \le \cdots \le z_{m+n}$.

M1. [Initialize.] Set $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$.

M2. [Find smaller.] If $x_i \le y_j$, go to step M3, otherwise go to M5.

M3. [Output x_i.] Set $z_k \leftarrow x_i$, $k \leftarrow k + 1$, $i \leftarrow i + 1$. If $i \le m$, return to M2.

M4. [Transmit $y_j, \ldots, y_n$.] Set $(z_k, \ldots, z_{m+n}) \leftarrow (y_j, \ldots, y_n)$ and terminate the algorithm.

M5. [Output y_j.] Set $z_k \leftarrow y_j$, $k \leftarrow k + 1$, $j \leftarrow j + 1$. If $j \le n$, return to M2.

M6. [Transmit $x_i, \ldots, x_m$.] Set $(z_k, \ldots, z_{m+n}) \leftarrow (x_i, \ldots, x_m)$ and terminate the algorithm. ▮

We shall see in Section 5.3.2 that this straightforward procedure is essentially the "best possible" way to merge on a conventional computer, when $m \approx n$. (On the other hand, when m is much smaller than n, it is possible to devise more efficient merging algorithms, although they are rather complicated in general.) Algorithm M could be made slightly simpler without much loss of efficiency by placing artificial "sentinel" elements $x_{m+1} = y_{n+1} = \infty$ at the end of the input files, stopping just before ∞ is output. For an analysis of Algorithm M, see exercise 2.

The total amount of work involved in Algorithm M is essentially proportional to $m + n$, so it is clear that merging is a simpler problem than sorting. Furthermore, we can reduce the problem of sorting to merging, for it is possible to continue merging longer and longer subfiles until everything is in sort. We may consider this to be an extension of the idea of insertion sorting: inserting a new element into a sorted file is the special case $n = 1$ of merging! If we want to speed up the insertion process we can consider inserting several elements at a time, "batching" them, and this leads naturally to the general idea of merge sorting. From a historical point of view, merge sorting is one of the very first

methods proposed for computer sorting; it was suggested by John von Neumann as early as 1945 (see Section 5.5).

We shall study merging in considerable detail in Section 5.4, with regard to external sorting algorithms; in the present section, we shall focus only on the question of merge sorting within a high-speed random-access memory.

Table 1 shows a merge sort which "burns the candle at both ends" in a manner similar to the scanning procedure we have used in quicksort, radix exchange, etc.: we examine the input from the left and from the right, working towards the middle. Ignoring the top line of the table for a moment, let us consider the transformation from line 2 to line 3. At the left we have the ascending run 503 703 765; at the right, reading leftwards, we have the run 087 512 677. Merging these two sequences leads to 087 503 512 677 703 765, which is placed at the left of line 3. Then the keys 061 612 908 in line 2 are merged with 170 509 897, and the result (061 170 509 612 897 908) is recorded at the *right* end of line 3. Finally, 154 275 426 653 is merged with 653 (the overlap is discovered before it causes any harm), and the result is placed at the left. Line 2 of the table was formed in the same way from the original input in line 1.

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Vertical lines in Table 1 represent the boundaries between runs. They are the so-called "step-downs," where a smaller element follows a larger one. In the middle of the file we generally have an ambiguous situation where we read the same key from both directions; this causes no problem if we are a little bit careful as in the following algorithm. The method is traditionally called a "natural" merge because it makes use of the runs which occur "naturally" in its input.

Algorithm N (*Natural two-way merge sort*). Records $R_1, \ldots, R_N$ are sorted using two areas of memory, each of which is capable of holding N records. For convenience, we shall say that the records of the second area are $R_{N+1}, \ldots, R_{2N}$, although it is not really necessary that R_{N+1} be adjacent to R_N. The initial contents of $R_{N+1}, \ldots, R_{2N}$ is immaterial. After sorting is complete, the keys will be in order, $K_1 \leq \cdots \leq K_N$.

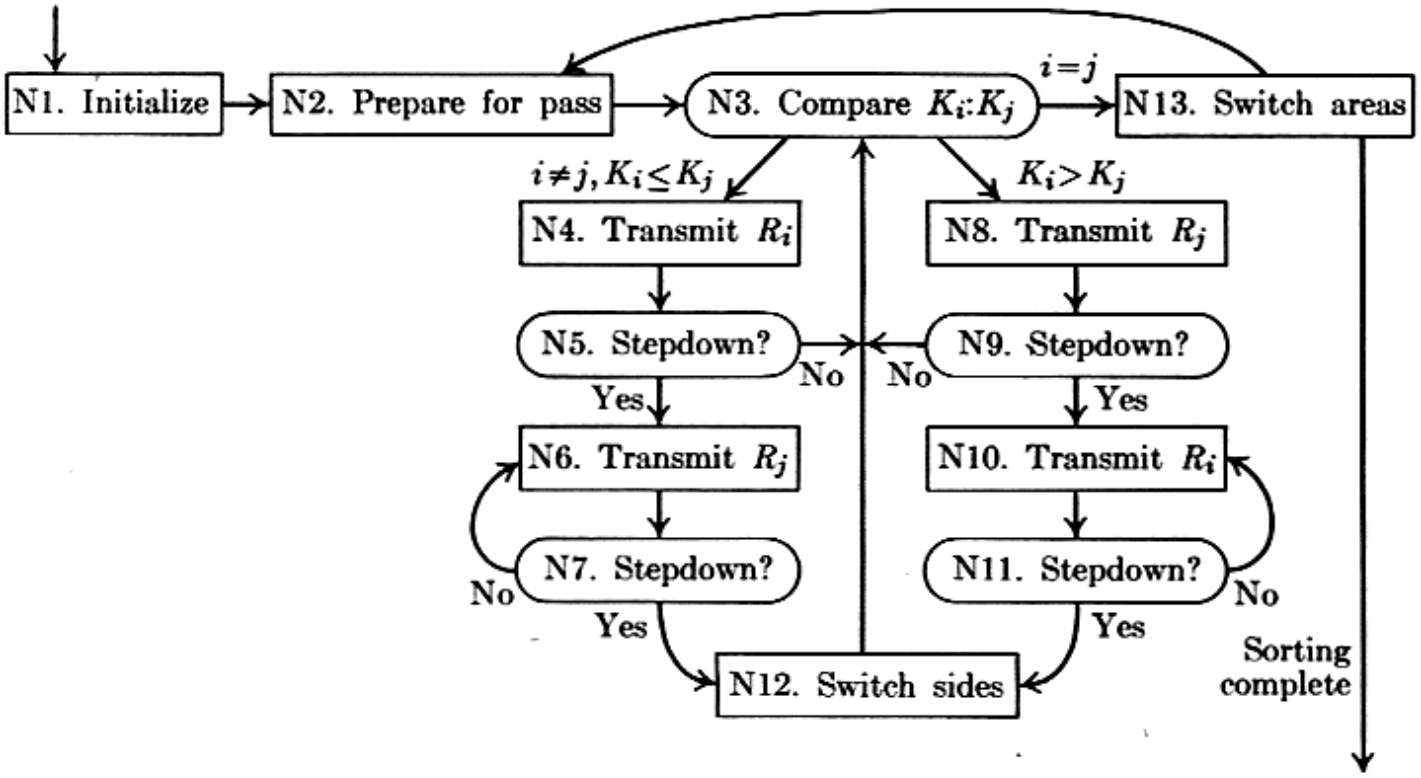


Fig. 30. Merge sorting.

N1. [Initialize.] Set $s \leftarrow 0$. (When $s = 0$, we will be transferring records from the $(R_1, \ldots, R_N)$ area to the $(R_{N+1}, \ldots, R_{2N})$ area; when $s = 1$, we will be going the other way.)

N2. [Prepare for pass.] If $s = 0$, set $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N + 1$, $l \leftarrow 2N$; if $s = 1$, set $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 1$, $l \leftarrow N$. (Variables i, j, k, l point to the current positions in the "source files" being read and the "destination files" being written.) Set $d \leftarrow 1$, $f \leftarrow 1$. (Variable d gives the current direction of output; f is set to zero if future passes are necessary.)

N3. [Compare K_i:K_j.] If $K_i > K_j$, go to step N8. If $i = j$, set $R_k \leftarrow R_i$ and go to N13.

N4. [Transmit R_i.] (Steps N4–N7 are analogous to steps M3–M4 of Algorithm M.) Set $R_k \leftarrow R_i$, $k \leftarrow k + d$.

N5. [Stepdown?] Increase i by 1. Then if $K_{i-1} \leq K_i$, go back to step N3.

N6. [Transmit R_j.] Set $R_k \leftarrow R_j$, $k \leftarrow k + d$.

N7. [Stepdown?] Decrease j by 1. Then if $K_{j+1} \leq K_j$, go back to step N6; otherwise go to step N12.

N8. [Transmit R_j.] (Steps N8–N11 are dual to steps N4–N7.) Set $R_k \leftarrow R_j$, $k \leftarrow k + d$.

N9. [Stepdown?] Decrease j by 1. Then if $K_{j+1} \leq K_j$, go back to step N3.

N10. [Transmit R_i.] Set $R_k \leftarrow R_i$, $k \leftarrow k + d$.

N11. [Stepdown?] Increase i by 1. Then if $K_{i-1} \leq K_i$, go back to step N10.

N12. [Switch sides.] Set $f \leftarrow 0$, $d \leftarrow -d$, and interchange $k \leftrightarrow l$. Return to step N3.

N13. [Switch areas.] If $f = 0$, set $s \leftarrow 1 - s$ and return to N2. Otherwise sorting is complete; if $s = 0$, set $(R_1, \ldots, R_N) \leftarrow (R_{N+1}, \ldots, R_{2N})$. (This last copying operation is unnecessary if it is acceptable to have the output in $(R_{N+1}, \ldots, R_{2N})$ at times.) ▮

This algorithm contains one tricky feature which is explained in exercise 5.

It would not be difficult to program Algorithm N for MIX, but we can deduce the essential facts of its behavior without constructing the entire program. The number of ascending runs in the input will be about $\frac{1}{2}N$, under random conditions, since we have $K_i > K_{i+1}$ with probability $\frac{1}{2}$; detailed information about the number of runs, under slightly different hypotheses, has been derived in Section 5.1.3. Each pass cuts the number of runs in half (except in unusual cases such as the situation in exercise 6). So the number of passes will usually be about $\log_2 N$. Each pass requires us to transmit each of the N records, and by exercise 2 most of the time is spent in steps N3, N4, N5, N8, N9. 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	CMPA, JG, JE	$3.5u$
Either	N4	STA, INC	$3u$
	N5	INC, LDA, CMPA, JGE	$6u$
Or	N8	STX, INC	$3u$
	N9	DEC, LDX, CMPX, JGE	$6u$

Thus about $12.5u$ is spent on each record in each pass, and the total running time will be asymptotically $12.5N \log_2 N$, for both the average case and the worst case. This is 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 $16N \log_2 N$.

The boundary lines between runs are determined in Algorithm N entirely by "step-downs." This has the possible advantage that input files with a preponderance of increasing *or decreasing* order can be handled very quickly; but it slows down the main loop of the calculation. Instead of testing step-downs, we can determine the length of runs artifically, by saying that all runs in the input have length 1, all runs after the first pass (except possibly the last run) have length 2, . . . , all runs after k passes (except possibly the last run) have length 2^k. This is called a *straight* two-way merge, as opposed to the "natural" merge in Algorithm N.

Straight two-way merging is very similar to Algorithm N, and it has essentially the same flowchart; but things are just different enough to make it worth while to write down the whole algorithm again:

Algorithm S (*Straight two-way merge sort*). Records $R_1, \ldots, R_N$ are sorted using two memory areas as in Algorithm N.

S1. [Initialize.] Set $s \leftarrow 0$, $p \leftarrow 1$. (For the significance of variables s, i, j, k, l, d, see Algorithm N. Here p represents the size of ascending runs to be merged on the current pass; q and r keep track of the number of unmerged items in a run.)

S2. [Prepare for pass.] If $s = 0$, set $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N$, $l \leftarrow 2N + 1$; if $s = 1$, set $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 0$, $l \leftarrow N + 1$. Then set $d \leftarrow 1$, $q \leftarrow p$, $r \leftarrow p$.

S3. [Compare K_i:K_j.] If $K_i > K_j$, go to step S8.

S4. [Transmit R_i.] Set $k \leftarrow k + d$, $R_k \leftarrow R_i$.

S5. [End of run?] Set $i \leftarrow i + 1$, $q \leftarrow q - 1$. If $q > 0$, go back to step S3.

S6. [Transmit R_j.] Set $k \leftarrow k + d$. Then if $k = l$, go to step S13; otherwise set $R_k \leftarrow R_j$.

S7. [End of run?] Set $j \leftarrow j - 1$, $r \leftarrow r - 1$. If $r > 0$, go back to step S6; otherwise go to S12.

S8. [Transmit R_j.] Set $k \leftarrow k + d$, $R_k \leftarrow R_j$.

S9. [End of run?] Set $j \leftarrow j - 1$, $r \leftarrow r - 1$. If $r > 0$, go back to step S3.

S10. [Transmit R_i.] Set $k \leftarrow k + d$. Then if $k = l$, go to step S13; otherwise set $R_k \leftarrow R_i$.

S11. [End of run?] Set $i \leftarrow i + 1$, $q \leftarrow q - 1$. If $q > 0$, go back to step S10.

S12. [Switch sides.] Set $q \leftarrow p$, $r \leftarrow p$, $d \leftarrow -d$, and interchange $k \leftrightarrow l$. Return to step S3.

S13. [Switch areas.] Set $p \leftarrow p + p$. If $p < N$, set $s \leftarrow 1 - s$ and return to S2. Otherwise sorting is complete; if $s = 0$, set

$$(R_1, \ldots, R_N) \leftarrow (R_{N+1}, \ldots, R_{2N}).$$

(The latter copying operation will be done if and only if $\lceil \log_2 N \rceil$ is odd, regardless of the distribution of the input. Therefore it is possible to predict the location of the sorted output in advance, and copying will usually be unnecessary.) ∎

For an example of this algorithm, see Table 2. It is somewhat amazing that the method works properly when N is not a power of 2; the runs being merged are not all of length 2^k, yet no provision has apparently been made for the exceptions! (See exercise 8.) The former tests for step-downs have been replaced by decrementing q or r and testing the result for zero; this reduces the asymptotic MIX running time to $11N \log_2 N$ units, slightly faster than we were able to achieve with Algorithm N.

Table 2

STRAIGHT TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	512	677	509	908	426	897	653	275	170	154	612	061	765	087
087	503	703	765	154	170	509	908	897	653	426	275	677	612	512	061
061	087	503	512	612	677	703	765	908	897	653	509	426	275	170	154
061	087	154	179	275	426	503	509	512	612	653	677	703	765	897	908

In practice it would be worth while to combine Algorithm S with straight insertion; we can sort groups of, say, 16 items using straight insertion, in place of the first four passes of Algorithm S, thereby avoiding the comparatively wasteful bookkeeping operations involved in short merges. As we saw with quicksort, such a combination of methods does not affect the asymptotic running time, but it gives us a reasonable improvement nevertheless.

Let us now study Algorithms N and S from the standpoint of data structures. Why did we need $2N$ record locations instead of N? The reason is comparatively simple: We are dealing with four lists of varying size (two "source" lists and two "destination" lists on each pass); and we are using the standard "growing together" idea discussed in Section 2.2.2, for each pair of sequentially allocated lists. But half of the memory space is always unused, and a little reflection shows that we should really make use of a *linked* allocation for the four lists. If we add one link field to each of the N records, we can do everything required by the merging algorithms using simple link manipulations without moving the records at all! Adding N link fields is generally better than adding the space needed for N more records, and the reduced record movement may also save us time. Therefore we ought to consider a merging algorithm like the following one:

Algorithm L (*List merge sort*). Records $R_1, \ldots, R_N$ are assumed to contain keys $K_1, \ldots, K_N$, and "link fields" $L_1, \ldots, L_N$ capable of holding the numbers $-(N+1)$ through $(N+1)$. There are two auxiliary link fields L_0 and L_{N+1} in artificial records R_0 and R_{N+1} at the beginning and end of the file. This algorithm is a "list sort" which sets the link fields so that the records are linked together in ascending order. After sorting is complete, L_0 is the index of the record with the smallest key; and for $1 \le k \le N$, L_k is the index of the record which follows R_k, or $L_k = 0$ if R_k is the record with the largest key. (See Eq. 5.2.1–9.)

During the course of this algorithm, R_0 and R_{N+1} serve as "list heads" for two linear lists whose sublists are being merged. A negative link denotes the end of a sublist known to be ordered; a zero link denotes the end of the entire list. We assume that $N \ge 2$.

The notation "$|L_s| \leftarrow p$" means "Set L_s to p or $-p$, retaining the previous sign of L_s." This operation is well-suited to MIX, but unfortunately not to most computers; it is possible to modify the algorithm in straightforward ways to obtain an equally efficient method for most other machines.

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

L2. [Begin new pass.] Set $s \leftarrow 0$, $t \leftarrow N+1$, $p \leftarrow L_s$, $q \leftarrow L_t$. If $q = 0$, the algorithm terminates. (During each pass, p and q traverse the lists being merged; 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 $p > 0$, 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 $q \le 0$. 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 $q > 0$, 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 $p \le 0$.

L8. [End of pass?] (At this point, $p \le 0$ and $q \le 0$, since both pointers have moved to the end of their respective sublists.) Set $p \leftarrow -p$, $q \leftarrow -q$. If $q = 0$, set $|L_s| \leftarrow p$, $|L_t| \leftarrow 0$ and return to L2. Otherwise return to L3. ∎

An example of this algorithm in action appears in Table 3, where we can see the link settings each time step L2 is encountered. It is possible to rearrange the records $R_1, \ldots, R_N$ at the end of this algorithm, so that their keys are in order, using the method of exercise 5.2–12. There is an interesting similarity between list merging and the addition of sparse polynomials (see Algorithm 2.2.4A).

Table 3

LIST MERGE SORTING

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
K_j	—	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703	—
L_j	1	−3	−4	−5	−6	−7	−8	−9	−10	−11	−12	−13	−14	−15	−16	0	0	2
L_j	2	−6	1	−8	3	−10	5	−11	7	−13	9	12	−16	14	0	0	15	4
L_j	4	3	1	−11	2	−13	8	5	7	0	12	10	9	14	16	0	15	6
L_j	4	3	6	7	2	0	8	5	1	14	12	10	13	9	16	0	15	11
L_j	4	12	11	13	2	0	8	5	10	14	1	6	3	9	16	7	15	0

Let us now construct a MIX 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 word long, with L_j in the (0:2) field and K_j in the (3:5) field of location INPUT+j; rI1 $\equiv p$, rI2 $\equiv q$, rI3 $\equiv s$, rI4 $\equiv t$, rA $\equiv R_q$. $N \geq 2$.

01	L	EQU	0:2		Definition of field names
02	ABSL	EQU	1:2		
03	KEY	EQU	3:5		
04	START	ENT1	N-2	1	*L1. Prepare two lists.*
05		ENNA	2,1	$N-2$	
06		STA	INPUT,1(L)	$N-2$	$L_i \leftarrow -(i+2)$.
07		DEC1	1	$N-2$	
08		J1P	*-3	$N-2$	$N-2 \geq i > 0$.
09		ENTA	1	1	
10		STA	INPUT(L)	1	$L_0 \leftarrow 1$.
11		ENTA	2	1	
12		STA	INPUT+N+1(L)	1	$L_{N+1} \leftarrow 2$.
13		STZ	INPUT+N-1(L)	1	$L_{N-1} \leftarrow 0$.
14		STZ	INPUT+N(L)	1	$L_N \leftarrow 0$.
15		JMP	L2	1	To L2. (See next page.)
16	L3Q	LDA	INPUT,2	$C''+B'$	*L3. Compare K_p:K_q.*
17	L3P	CMPA	INPUT,1(KEY)	C	
18		JL	L6	C	To L6 if $K_q < K_p$.
19	L4	ST1	INPUT,3(ABSL)	C'	*L4. Advance p.* $\lvert L_s\rvert \leftarrow p$.
20		ENT3	0,1	C'	$s \leftarrow p$.
21		LD1	INPUT,1(L)	C'	$p \leftarrow L_q$.
22		J1P	L3P	C'	To L3 if $p > 0$.
23	L5	ST2	INPUT,3(L)	B'	*L5. Complete the sublist.* $L_s \leftarrow q$.
24		ENT3	0,4	B'	$s \leftarrow t$.
25		ENT4	0,2	D'	$t \leftarrow q$.
26		LD2	INPUT,2(L)	D'	$q \leftarrow L_q$.
27		J2P	*-2	D'	Repeat if $q > 0$.
28		JMP	L8	B'	To L8.
29	L6	ST2	INPUT,3(ABSL)	C''	*L6. Advance q.* $\lvert L_s\rvert \leftarrow q$.
30		ENT3	0,2	C''	$s \leftarrow q$.
31		LD2	INPUT,2(L)	C''	$q \leftarrow L_q$.
32		J2P	L3Q	C''	To L3 if $q > 0$.
33	L7	ST1	INPUT,3(L)	B''	*L7. Complete the sublist.* $L_s \leftarrow p$.
34		ENT3	0,4	B''	$s \leftarrow t$.
35		ENT4	0,1	D''	$t \leftarrow p$.
36		LD1	INPUT,1(L)	D''	$p \leftarrow L_p$.
37		J1P	*-2	D''	Repeat if $p > 0$.
38	L8	ENN1	0,1	B	*L8. End of pass?* $p \leftarrow -p$.
39		ENN2	0,2	B	$q \leftarrow -q$.
40		J2NZ	L3Q	B	To L3 if $q \neq 0$.
41		ST1	INPUT,3(ABSL)	A	$\lvert L_s\rvert \leftarrow p$.
42		STZ	INPUT,4(ABSL)	A	$\lvert L_t\rvert \leftarrow 0$.

43	L2	ENT3	0	$A+1$	*L2. Begin new pass.* $s \leftarrow 0$.
44		ENT4	N+1	$A+1$	$t \leftarrow N+1$.
45		LD1	INPUT(L)	$A+1$	$p \leftarrow L_s$.
46		LD2	INPUT+N+1(L)	$A+1$	$q \leftarrow L_t$.
47		J2NZ	L3Q	$A+1$	To L3 if $q \neq 0$. ▮

The running time of this program can be deduced using techniques we have seen many times before (see exercises 13, 14); it comes to approximately $(10N \log_2 N + 4.92N)u$ on the average, with a small standard deviation of order $\sqrt{N}$. Exercise 15 shows that the running time can be reduced to about $9N \log_2 N$ at the expense of a somewhat longer program.

Thus we have a clear victory for linked-memory techniques over sequential allocation, when internal merging is being done: Less memory space is required, and the program runs about 10 to 20 percent faster. Similar algorithms have been published by L. J. Woodrum [*IBM Systems J.* **8** (1969), 189–203] and A. D. Woodall [*Comp. J.* **13** (1970), 110–111].

EXERCISES

1. [*20*] Generalize Algorithm M to a *k-way merge* of the input files $x_{i1} \le \cdots \le x_{im_i}$ for $i = 1, 2, \ldots, k$.

2. [*M24*] Assuming that each of the $\binom{m+n}{m}$ possible arrangement of m x's among n y's is equally likely, find the mean and standard deviation of the number of times step M2 is performed during Algorithm M. What are the maximum and minimum values of this quantity?

▶ 3. [*20*] (*Updating.*) Given records $R_1, \ldots, R_M$ and $R'_1, \ldots, R'_N$ whose keys are distinct and in order, so that $K_1 < \cdots < K_M$ and $K'_1 < \cdots < K'_N$, show how to modify Algorithm M to obtain a merged file in which records R_i of the first file have been *discarded* if their key appears also in the second file.

4. [*21*] The text observes that merge sorting may be regarded as a generalization of insertion sorting. Show that merge sorting is also strongly related to tree selection sorting as depicted in Fig. 23.

▶ 5. [*21*] Prove that i can never be equal to j in steps N6 or N10. (Therefore it is unnecessary to test for a possible jump to N13 in those steps.)

6. [*22*] Find a permutation $K_1 K_2 \ldots K_{16}$ of $\{1, 2, \ldots, 16\}$ such that

$$K_2 > K_3, \quad K_4 > K_5, \quad K_6 > K_7, \quad K_8 > K_9,$$
$$K_{10} < K_{11}, \quad K_{12} < K_{13}, \quad K_{14} < K_{15},$$

yet Algorithm N will sort the file in only two passes. (Since there are eight or more runs, we would expect to have at least four runs after the first pass, two runs after the second pass, and sorting would ordinarily not be complete until after at least three passes. How can we get by with only two passes?)

7. [*16*] Give a formula for the exact number of passes required by Algorithm S, as a function of N.

8. [*22*] During Algorithm S, the variables q and r are supposed to represent the lengths of the unmerged elements in the runs currently being processed; q and r both start out equal to p, while the runs are not always this long. How can this possibly work?

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

10. [*25*] (D. A. Bell.) Show that sequentially-allocated straight two-way merging can be done with at most $\frac{3}{2}N$ memory locations, instead of $2N$ as in Algorithm S.

11. [*21*] Is Algorithm L a "stable" sorting method?

▶**12.** [*22*] Revise step L1 of Algorithm L so that the two-way merge is "natural," taking advantage of ascending runs which are initially present. (In particular, if the input is already sorted, step L2 should terminate the algorithm immediately after your step L1 has acted.)

▶**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?

14. [*M24*] Let the binary representation of N be $2^{e_1} + 2^{e_2} + \cdots + 2^{e_t}$, where $e_1 > e_2 > \cdots > e_t \geq 0$, $t \geq 1$. Prove that the maximum number of key comparisons performed by Algorithm L is $1 - 2^{e_t} + \sum_{1 \leq k \leq t} (e_k + k - 1)2^{e_k}$.

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?

16. [*28*] Design a list merging algorithm like Algorithm L but based on three-way merging.

17. [*20*] (J. McCarthy.) Let the binary representation of N be as in exercise 14, and assume that we are given N records arranged in t ordered subfiles of respective sizes $2^{e_1}, 2^{e_2}, \ldots, 2^{e_t}$. Show how to maintain this state of affairs when a new $(N + 1)$st record is added and $N \leftarrow N + 1$. (The resulting algorithm may be called an "on-line" merge sort.)

18. [*40*] (M. A. Kronrod.) Given a file on N records containing only two runs,

$$K_1 \leq \cdots \leq K_M \qquad \text{and} \qquad K_{M+1} \leq \cdots \leq K_N,$$

is it possible to sort the file in $O(N)$ operations in a random-access memory, *using only a small fixed amount of additional memory space* regardless of the size of M and N? (All of the merging algorithms described in this section make use of extra memory space proportional to N.)

19. [*26*] Consider a railway switching network with n "stacks", as shown in Fig. 31 when $n = 5$; such a network has some aspects relevant to n-pass sorting algorithms. We considered one-stack networks in exercises 2.2.1–2 through 2.2.1–5. If N railroad cars enter at the right, we observed that only comparatively few of the $N!$ permutations of those cars could appear at the left, in the one-stack case.

In the n-stack network, assume that 2^n cars enter at the right. Prove that each of the $2^n!$ possible permutations of these cars *is* achievable at the left, by a suitable

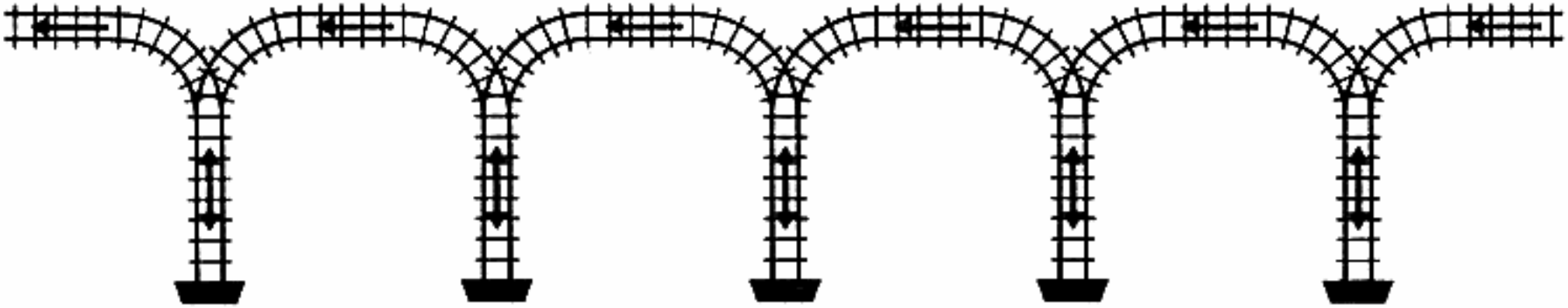


Fig. 31. A railway network with five "stacks."

sequence of operations. (Each stack is big enough to accommodate all the cars, if necessary.)

20. [*47*] In the notation of exercise 2.2.1–4, at most a_N^n permutations of N elements can be produced with an n-stack railway network; hence the number of stacks needed to obtain all $N!$ permutations is at least $\log N!/\log a_N \approx \log_4 N$. Exercise 19 shows that at most $\lceil \log_2 N \rceil$ stacks are needed. What is the true rate of growth of the necessary number of stacks, as $N \to \infty$?

21. [*23*] (A. J. Smith.) Explain how to extend Algorithm L so that, in addition to sorting, it computes the number of *inversions* present in the input permutation.

5.2.5. Sorting by Distribution

We come now to an interesting class of sorting methods, which is shown in Section 5.4.7 to be essentially the exact *opposite* of merging. Readers who are familiar with punched card equipment are well aware of the efficient procedure used on card sorters, based on the digits of the keys; the same idea can be adapted to computer programming, and it is generally known as "radix sorting," "digital sorting," or "pocket sorting."

Suppose we want to sort a 52-card deck of playing cards. We may define

$$\mathtt{A} < \mathtt{2} < \mathtt{3} < \mathtt{4} < \mathtt{5} < \mathtt{6} < \mathtt{7} < \mathtt{8} < \mathtt{9} < \mathtt{10} < \mathtt{J} < \mathtt{Q} < \mathtt{K},$$

as an ordering of the face values, and for the suits we may define

$$\clubsuit < \diamondsuit < \heartsuit < \spadesuit$$

One card is to precede another if either (i) its suit is less than the other suit, or (ii) the suits are equal but its face value is less. (This is a particular case of a *lexicographic ordering* between ordered pairs of objects; cf. exercise 5–2). Thus

$$\mathtt{A}\clubsuit < \mathtt{2}\clubsuit < \cdots < \mathtt{K}\clubsuit < \mathtt{A}\diamondsuit < \cdots < \mathtt{Q}\spadesuit < \mathtt{K}\spadesuit.$$

We could sort the cards by any of the methods already discussed; people generally use a technique somewhat analogous to the idea behind radix exchange. It is natural to sort the cards first by their suits into four piles, then to fiddle with each of the individual piles until they are in order.

But there is a faster way to do the trick! First deal the cards face up into 13 piles, one for each face value. Then collect these piles by putting the aces

on the bottom, the 2's face up on top of them, then the 3's, etc., finally putting the kings (face up) on top. Turn the deck face down and deal again, this time into four piles for the four suits. By putting the resulting piles together, with clubs on the bottom, then diamonds, hearts, and spades, the deck will be in perfect order.

The same idea applies to the sorting of numbers and alphabetic data. Why does it work? Because (in our playing card example) if two cards go into different piles in the final deal, they have different suits, so the one with the lower suit is lowest. But if two cards have the same suit (and consequently go into the same pile), they are already in proper order because of the previous sorting. In other words, the face values will be in increasing order on each of the four piles as we deal the cards on the second pass. The same proof can be abstracted to show that any lexicographic ordering can be sorted in this manner; for details, see the answer to exercise 5–2, at the beginning of this chapter.

The sorting method just described is not immediately obvious, and it isn't clear who first discovered the fact that it works so conveniently. A 19-page pamphlet entitled "The Inventory Simplified," published by the Tabulating Machines Company division of IBM in 1923, presents an interesting Digit Plan method for forming sums of products on their Electric Sorting Machine: Suppose, for example, that we want to multiply the number punched in columns 1–10 by the number punched in columns 23–25, and to sum all of these products for a large number of cards. We can sort first on column 25, then use the Tabulating Machine to find the quantities $a_1, a_2, \ldots, a_9$, where a_k is the total of columns 1–10 summed over all cards having k in column 25. Then we can sort on column 24, finding the analogous totals $b_1, b_2, \ldots, b_9$; also on column 23, obtaining $c_1, c_2, \ldots, c_9$. The desired sum of products is easily seen to be

$$a_1 + 2a_2 + \cdots + 9a_9 + 10b_1 + 20b_2 + \cdots + 90b_9 + 100c_1 + 200c_2 + \cdots + 900c_9.$$

This punched-card tabulating method leads naturally to the discovery of least-significant-digit-first radix sorting, so it probably became known to the operators of the machines. The first published reference to this principle for sorting appears in L. J. Comrie's early discussion of punched-card equipment [*Transactions of the Office Machinery Users' Assoc., Ltd.* (1929), 25–37, esp. p. 28].

In order to handle radix sorting inside a computer, we must decide what to do with the piles. Suppose that there are M piles; we could set aside M areas of memory, moving each record from an input area into its appropriate pile area. But this is unsatisfactory, since each area must be large enough to hold N items, and $(M + 1)N$ record spaces would be required. Such an excessive memory requirement originally caused most people to reject the idea of radix sorting within a computer, until H. H. Seward [Master's thesis, M.I.T. Digital Computer Laboratory Report R-232 (Cambridge, Mass.: 1954), 25–28] pointed out that we can achieve the same effect with only $2N$ record areas and M count

fields. We simply count how many elements will lie in each of the M piles, by making a preliminary pass over the data; this tells us precisely how to allocate memory for the piles. We have already made use of the same idea in the "distribution sort," Algorithm 5.2D.

Thus radix sorting can be carried out as follows: Start with a distribution sort based on the *least significant digit* of the keys (in radix M notation), moving records from the input area to an auxiliary area. Then do another distribution sort, on the next least significant digit, moving the records back into the original input area; and so on, until the final pass (on the most significant digit) puts all records into the desired order.

If we have a decimal computer with 12-digit keys, and if N is rather large, we can choose $M = 1000$ (considering three decimal digits as one radix-1000 digit); sorting will be complete in four passes, regardless of the size of N. Similarly, if we have a binary computer and a 40-bit key, we can set $M = 1024$ and complete the sorting in four passes. Actually each "pass" consists of three parts (counting, allocating, moving); Friend [*JACM* **3** (1956), 151] has suggested combining two of these parts at the expense of M more memory locations, by accumulating the counts for pass $k + 1$ while moving the records in pass k.

Table 1 shows how such a radix sort can be applied to our 16 example numbers, with $M = 10$. Radix sorting is generally not useful for such small N, so a small example like this is intended to illustrate the sufficiency rather than the efficiency of the method.

Table 1

RADIX SORTING

Input area contents:	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
Counts for units digit distribution:					1	1	2	3	1	2	1	3	1	1		
Storage allocations based on these counts:					1	2	4	7	8	10	11	14	15	16		
Auxiliary area contents:	170	061	512	612	503	653	703	154	275	765	426	087	897	677	908	509
Counts for tens digit distribution:					4	2	1	0	0	2	2	3	1	1		
Storage allocations based on these counts:					4	6	7	7	7	9	11	14	15	16		
Input area contents:	503	703	908	509	512	612	426	653	154	061	765	170	275	677	087	897
Counts for hundreds digit distribution:					2	2	1	0	1	3	3	2	1	1		
Storage allocations based on these counts:					2	4	5	5	6	9	12	14	15	16		
Auxiliary area contents:	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

An alert, "modern" reader will note, however, that the whole idea of making digit counts for the storage allocation is tied to "old-fashioned" ideas about sequential data representation; we know that *linked* allocation is specifically

designed to handle multiple tables of variable size, so it is natural to choose a linked data structure for radix sorting. Since we traverse each pile serially, all that is necessary is a single link from each item to its successor. Furthermore, we never need to move the records, we merely adjust the links and proceed merrily down the lists. The amount of memory required is $(1 + \epsilon)N + 2\epsilon M$ records, where ϵ is the amount of space taken up by a link field. Formal details of this procedure are rather interesting since they furnish an excellent example of typical data structure manipulations, combining sequential and linked allocation:

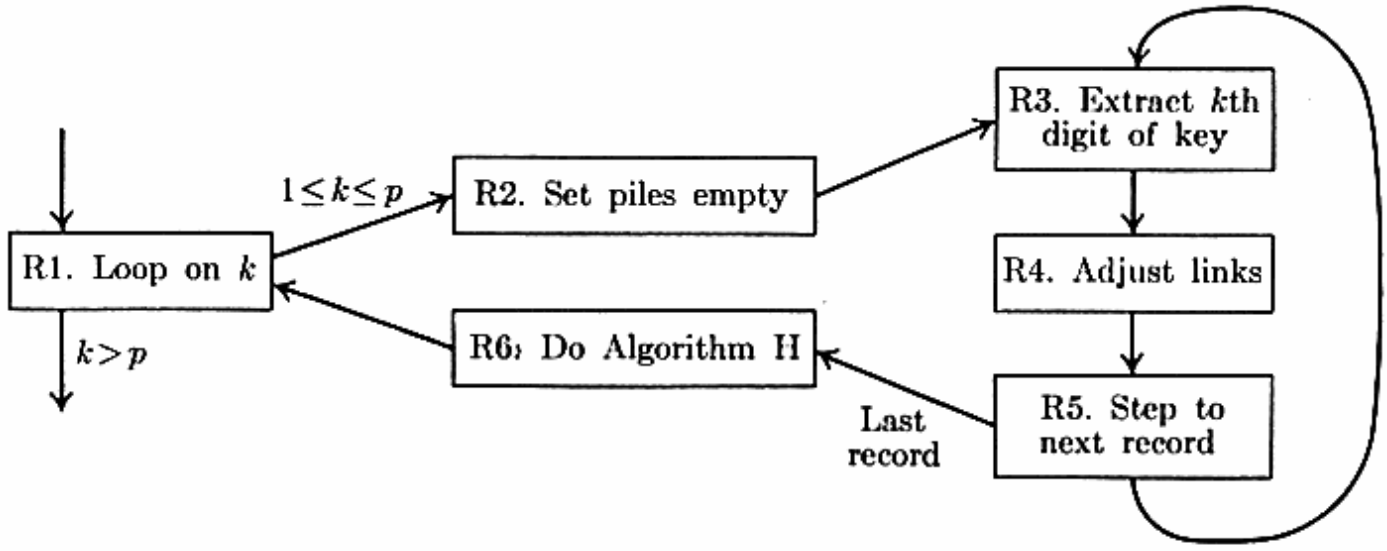


Fig. 32. Radix list sort.

Algorithm R (*Radix list sort*). Records $R_1, \ldots, R_N$ are each assumed to contain a LINK field. Their keys are assumed to be p-tuples

$$(a_p, \ldots, a_2, a_1), \qquad 0 \le a_i < M,$$

where the order is defined lexicographically so that

$$(a_p, \ldots, a_2, a_1) < (b_p, \ldots, b_2, b_1)$$

if and only if for some j, $1 \le j \le p$, we have

$$a_i = b_i \qquad \text{for all} \qquad i > j, \qquad \text{but} \qquad a_j < b_j.$$

The keys may, in particular, be thought of as numbers written in radix M notation,

$$a_p M^{p-1} + \cdots + a_2 M + a_1,$$

and in this case lexicographic order corresponds to the normal ordering of nonnegative numbers. The keys may also be strings of alphabetic letters, etc.

Sorting is done by keeping M "piles" of records, in a manner that exactly parallels the action of a card sorting machine. The piles are really queues in the sense of Chapter 2, since we link them together so that they are traversed

in a first-in-first-out manner. There are two pointer variables TOP[i] and BOTM[i] for each pile, $0 \le i < M$, and we assume as in Chapter 2 that

$$\texttt{LINK(LOC(BOTM([}i\texttt{])))} \equiv \texttt{BOTM[}i\texttt{]}.$$

R1. [Loop on k.] In the beginning, set P $\leftarrow$ LOC(R_N), a pointer to the last record. Then perform steps R2 through R6 for $k = 1, 2, \ldots, p$. (Steps R2 through R6 constitute one "pass.") Then the algorithm terminates, with P pointing to the record with smallest key, LINK(P) to the record with next smallest, then LINK(LINK(P)), etc.; the LINK in the final record will be Λ.

R2. [Set piles empty.] Set TOP[i] $\leftarrow$ LOC(BOTM[i]) and BOTM[i] $\leftarrow \Lambda$, for $0 \le i < M$.

R3. [Extract kth digit of key.] Let KEY(P), the key in the record referenced by P, be $(a_p, \ldots, a_2, a_1)$; set $i \leftarrow a_k$, the kth least significant digit of this key.

R4. [Adjust links.] Set LINK(TOP[i]) $\leftarrow$ P, then set TOP[i] $\leftarrow$ P.

R5. [Step to next record.] If $k = 1$ (the first pass) and if P = LOC(R_j), for some $j \ne 1$, set P $\leftarrow$ LOC(R_{j-1}) and return to R3. If $k > 1$ (subsequent passes), set P $\leftarrow$ LINK(P), and return to R3 if P $\ne \Lambda$.

R6. [Do Algorithm H.] (We are now done distributing all elements onto the piles.) Perform Algorithm H below, which "hooks together" the individual piles into one list, in preparation for the next pass. Then set P $\leftarrow$ BOTM[0], a pointer to the first element of the hooked-up list. (See exercise 3.) ▮

Algorithm H (*Hooking-up of queues*). Given M queues, linked according to the conventions of Algorithm R, this algorithm adjusts at most M links so that a single queue is created, with BOTM[0] pointing to the first element, and with pile 0 preceding pile 1 $\cdots$ preceding pile $(M - 1)$.

H1. [Initialize.] Set $i \leftarrow 0$.

H2. [Point to top of pile.] Set P $\leftarrow$ TOP[i].

H3. [Next pile.] Increase i by 1. If $i = M$, set LINK(P) $\leftarrow \Lambda$ and terminate the algorithm.

H4. [Is pile empty?] If BOTM[i] $= \Lambda$, go back to H3.

H5. [Tie piles together.] Set LINK(P) $\leftarrow$ BOTM[i]. Return to H2. ▮

Figure 33 shows the contents of the piles after each of the three passes, when our 16 example numbers are sorted with $M = 10$. Algorithm R is very easy to program for MIX, once a suitable way to treat the pass-by-pass variation of steps R3 and R5 has been found. The following program does this without sacrificing any speed in the inner loop, by overlaying two of the instructions. Note that TOP[i] and BOTM[i] can be packed into the same word.

Program R (*Radix list sort*). The input in locations INPUT+1 through INPUT+N is assumed to have $p = 3$ components (a_3, a_2, a_1), stored respectively in the (1:1), (2:2), and (3:3) fields. (Thus M is assumed to be less than or equal to

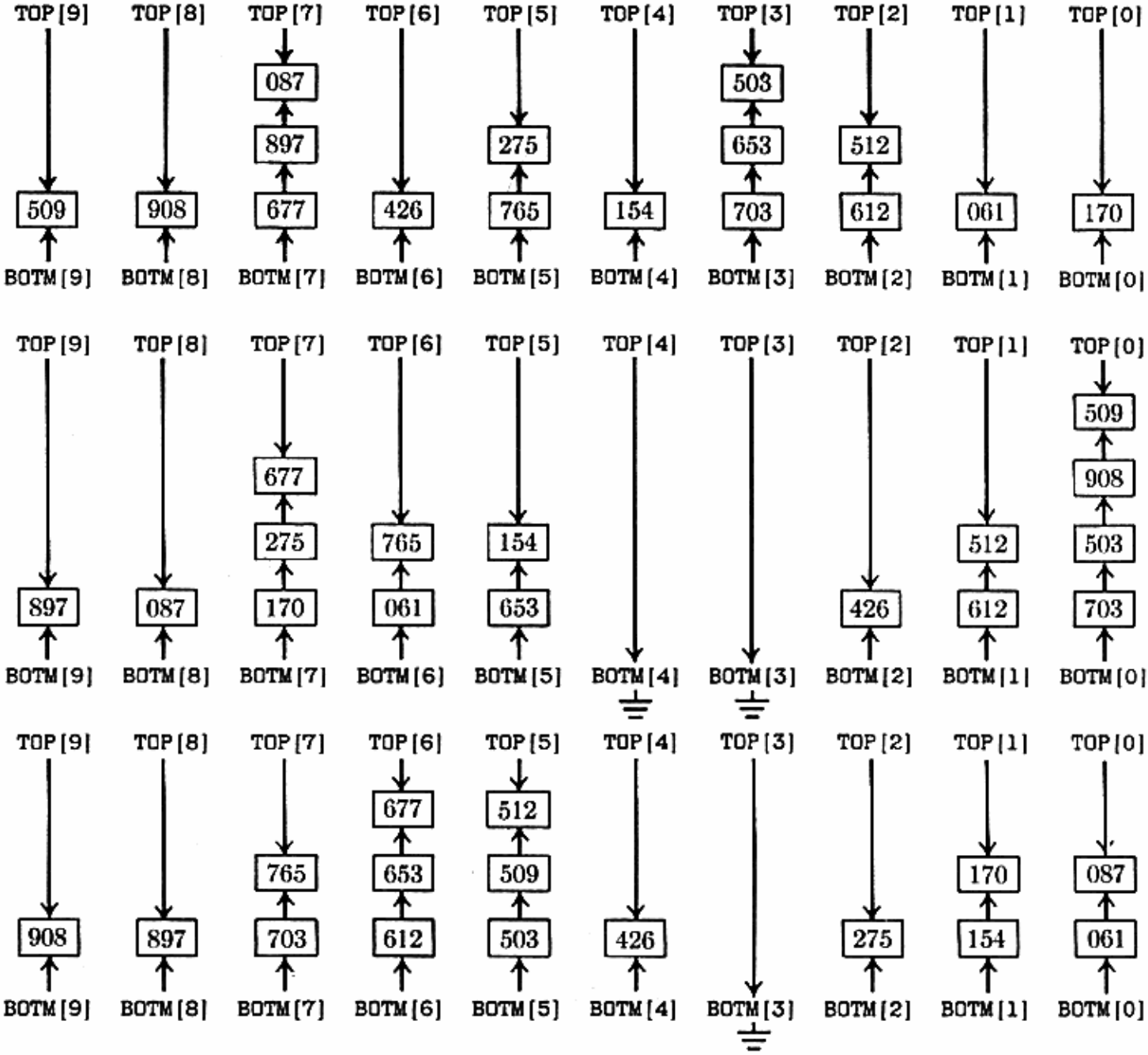


Fig. 33. Radix sort using linked allocation: contents of the ten piles after each pass.

the byte size of MIX.) The (4:5) field of each record is its LINK. We let TOP[i] $\equiv$ PILES+i(1:2) and BOTM[i] $\equiv$ PILES+i(4:5), for $0 \le i < M$. It is convenient to make links relative to location INPUT, so that LOC(BOTM[i]) = PILES + i − INPUT; to avoid negative links we therefore want the PILES table to be in higher locations than the INPUT table. Index registers are assigned as follows: rI1 $\equiv$ P, rI2 $\equiv i$, rI3 $\equiv 3 - k$, rI4 $\equiv$ TOP[i]; during Algorithm H, rI2 $\equiv i - M$.

```
01   LINK   EQU   4:5
02   TOP    EQU   1:2
03   START  ENT1  N                  1     R1. Loop on k.  P ← LOC(R_N).
04          ENT3  2                  1     k ← 1.
05   2H     ENT2  M-1                3     R2. Set piles empty.
06          ENTA  PILES-INPUT,2      3M    LOC(BOTM[i])
07          STA   PILES,2(TOP)       3M        → TOP[i]
08          STZ   PILES,2(LINK)      3M    BOTM[i] ← Λ.
09          DEC2  1                  3M
10          J2NN  *-4                3M    M > i ≥ 0.
```

11		LDA	R3SW,3	3	
12		STA	3F	3	Modify instructions for pass k.
13		LDA	R5SW,3	3	
14		STA	5F	3	
15	3H	[LD2	INPUT,1(3:3)]		*R3. Extract kth digit of key.*
16	4H	LD4	PILES,2(TOP)	$3N$	*R4. Adjust links.*
17		ST1	INPUT,4(LINK)	$3N$	LINK(TOP[i]) ← P.
18		ST1	PILES,2(TOP)	$3N$	TOP[i] ← P.
19	5H	[DEC1	1]		*R5. Step to next record.*
20		J1NZ	3B	$3N$	To R3 if end of pass.
21	6H	ENN2	M	3	*R6. Do Algorithm H.*
22		JMP	7F	3	To H2 with $i \leftarrow 0$.
23	R3SW	LD2	INPUT,1(1:1)	N	Instruction for R3 when $k = 3$.
24		LD2	INPUT,1(2:2)	N	Instruction for R3 when $k = 2$.
25		LD2	INPUT,1(3:3)	N	Instruction for R3 when $k = 1$.
26	R5SW	LD1	INPUT,1(LINK)	N	Instruction for R5 when $k = 3$.
27		LD1	INPUT,1(LINK)	N	Instruction for R5 when $k = 2$.
28		DEC1	1	N	Instruction for R5 when $k = 1$.
29	9H	LDA	PILES+M,2(LINK)	$3M - 3$	*H4. Is pile empty?*
30		JAZ	8F	$3M - 3$	To H3 if BOTM[i] = Λ.
31		STA	INPUT,1(LINK)	$3M - 3 - E$	*H5. Tie piles together.* LINK(P) ← BOTM[i].
32	7H	LD1	PILES+M,2(TOP)	$3M - E$	*H2. Point to top of pile.*
33	8H	INC2	1	$3M$	*H3. Next pile.* $i \leftarrow i + 1$.
34		J2NZ	9B	$3M$	To H4 if $i \neq M$.
35		STZ	INPUT,1(LINK)	3	LINK(P) ← Λ.
36		LD1	PILES(LINK)	3	P ← BOTM[0].
37		DEC3	1	3	
38		J3NN	2B	3	$1 \le k \le 3$. ▮

The running time of Program R is $32N + 48M + 38 - 4E$, where N is the number of input records, M is the radix (the number of piles), and E is the number of occurrences of empty piles. This compares very favorably with other programs we have constructed based on similar assumptions (Programs 5.2.1M, 5.2.4L). A p-pass version of the program would take $(11p - 1)N + O(pM)$ units of time; the critical factor in the timing is the inner loop, which involves five references to memory and one branch. On a typical computer we will have $M = b$ and $p = \lceil t/r \rceil$, where t is the number of radix-b digits in the keys; increasing r will decrease p, so the formulas can be used to determine a "best" value of r.

The only variable in the timing is E, the number of empty piles observed in step H4. If we consider each of the M^N sequences of radix-M digits to be equally probable, we know from our study of the "poker test" in Section 3.3.2D that there are $M - r$ empty piles with probability

$$\frac{M(M-1)\cdots(M-r+1)}{M^N}\left\{{N \atop r}\right\}$$

on each pass, where $\left\{{N \atop r}\right\}$ is a Stirling number of the second kind. By exercise 5,

$$E = \left(\min\ \max(M-N,0)p, \qquad \text{ave}\ M\left(1-\frac{1}{M}\right)^N p, \qquad \max\ (M-1)p\right).$$

An ever-increasing number of "pipeline" or "number crunching" computers have appeared in recent years. These machines have multiple arithmetic units

and "look-ahead" circuitry so that memory references and computation can be highly overlapped; but their efficiency deteriorates noticeably in the presence of conditional branch instructions unless the branch almost always goes the same way. The inner loop of a radix sort is well adapted to such machines, because it is a straight iterative calculation of typical "number crunching" form. Therefore *radix sorting is usually more efficient than any other known method for internal sorting on such machines*, provided that N is not too small and the keys are not too long.

Of course, radix sorting is not very efficient when the keys are extremely long. For example, imagine sorting cards on an 80-column key; very few pairs of cards will tend to have identical keys in their first five columns, so the first 75 passes accomplish very little. In our analysis of radix-exchange sorting, we found that it was unnecessary to inspect many bits of the key, when we approach the keys from the left instead of the right. Let us therefore reconsider the idea of a radix sort which starts at the most significant digit (MSD) instead of the least significant digit (LSD).

We have already remarked that an MSD-first radix method suggests itself naturally; in fact, it is not hard to see why the post office uses such a method to sort mail. A large collection of letters can be sorted into separate bags for different geographical areas; each of these bags then contains a smaller number of letters which can be sorted independently of the other bags, into finer and finer geographical divisions. (Indeed, they can be transported nearer to their destinations before they are sorted further.) This principle of "divide and conquer" is quite appealing, and the only reason it doesn't work especially well for sorting punched cards is that the multiplicity of piles tends to become confusing. The same phenomenon accounts for the relative efficiency of Algorithm R, even though it considers LSD first, since we never have more than M piles, and they need to be hooked together only p times. On the other hand, it would not be difficult to design an MSD-first radix method using linked memory, with negative links as in Algorithm 5.2.4L to denote the boundaries between piles. (See exercise 10.)

Perhaps the best compromise has been suggested by M. D. MacLaren [*JACM* **13** (1966), 404–411], who recommends an LSD-first sort as in Algorithm R, but *applied only to the most significant digits*. This does not completely sort the file, but it usually brings the file very nearly into order so that very few inversions remain; therefore straight insertion can be used to finish up. Our analysis of Algorithm 5.2.1M applies also to this situation, so that if the keys are uniformly distributed we will have an average of

$$\tfrac{1}{4}N(N-1)M^{-p}$$

inversions remaining in the file after sorting on the leading p digits. (See Eq. 5.2.1–14 and exercise 5.2.1–38.) MacLaren has computed the average number of memory references per item sorted, and the optimum choice of M and p (assuming that M is a power of 2, that the keys are uniformly distributed,

and that $N/M^p \leq 0.1$ so that deviations from uniformity are tolerable) turns out to be given by the following table:

$N =$	100	1000	5000	10000	50000	100000
best $M =$	32	128	256	512	1024	1024
best $p =$	2	2	2	2	2	2
$\bar{\beta}(N) =$	19.3	18.5	18.2	18.2	18.1	18.0

Here $\bar{\beta}(N)$ denotes the average number of memory references per item sorted; it is bounded as $N \to \infty$, if we take $p = 2$ and $M > \sqrt{N}$, so the average sorting time is actually $O(N)$ instead of order $N \log N$. This method is an improvement over multiple list insertion (Algorithm 5.2.1M) which is essentially the case $p = 1$. Exercise 12 gives MacLaren's interesting procedure for final rearrangement of a partially list-sorted file.

It is also possible to avoid the link fields, using the methods of Algorithm 5.2D and exercise 5.2–13, so that only $O(\sqrt{N})$ memory locations are needed in addition to the space required for the records themselves. The average sorting time is proportional to N if the input records are uniformly distributed.

EXERCISES

▶ **1.** [*20*] The algorithm of exercise 5.2–13 shows how to do a distribution sort with only N record areas (and M count fields), instead of $2N$ record areas. Does this lead to an improvement over the radix sorting algorithm illustrated in Table 1?

2. [*13*] Is Algorithm R a "stable" sorting method?

3. [*15*] Explain why Algorithm H makes BOTM[0] point to the first record in the "hooked-up" queue, *even though pile* 0 *might be empty.*

▶ **4.** [*23*] Algorithm R keeps the M piles linked together as queues (first-in-first-out). Explore the idea of linking the piles as *stacks* instead. (The arrows in Fig. 33 would go downward instead of upward, and the BOTM table would be unnecessary.) Show that if the piles are "hooked together" in an appropriate order, it is possible to achieve a valid sorting method. Does this lead to a simpler or a faster algorithm?

5. [*M24*] Let $g_{MN}(z) = \sum p_{MNk} z^k$, where p_{MNk} is the probability that exactly k empty piles are present after a random radix-sort pass puts N elements into M piles. (a) Show that $g_{M,N+1}(z) = g_{MN}(z) + ((1 - z)/M) g'_{MN}(z)$. (b) Use this relation to find simple expressions for the mean and variance of this probability distribution, as a function of M and N.

6. [*20*] What changes are necessary to Program R so that it sorts eight-byte keys instead of three-byte keys? Assume that the most significant bytes of K_i are stored in location KEY+i(1:5), while the three least significant bytes are in location INPUT+i(1:3) as presently. What is the running time of the program, after these changes have been made?

7. [*20*] Discuss the similarities and differences between Algorithm R and radix-exchange sorting (Algorithm 5.2.2R).

▶ **8.** [*20*] The radix-sorting algorithms discussed in the text assume that all keys being sorted are nonnegative. What changes should be made to the algorithms when the keys are numbers expressed in *two's complement* or *ones' complement* notation?

9. [*20*] Continuing exercise 8, what changes should be made to the algorithms when the keys are numbers expressed in *signed-magnitude* notation?

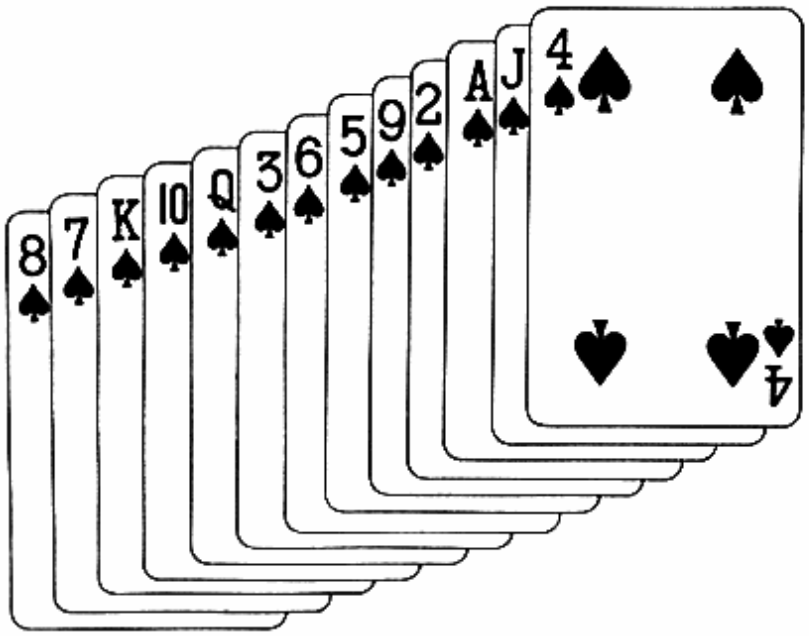
10. [*30*] Design an efficient most-significant-digit-first radix-sorting algorithm which uses linked memory. (As the size of the subfiles decreases, it is wise to decrease M, and to use a nonradix method on the really short subfiles.)

11. [*16*] The sixteen input numbers shown in Table 1 start with 41 inversions; after sorting is complete, of course, there are no inversions remaining. How many inversions would be present in the file if we omitted pass 1, doing a radix sort only on the tens and hundreds digits? How many inversions would be present if we omitted both pass 1 and pass 2?

12. [*24*] (M. D. MacLaren.) Suppose that Algorithm R has been applied only to the p leading digits of the actual keys; thus the file is nearly sorted when we read it in the order of the links, but keys which agree in their first p digits may be out of order. Design an algorithm which rearranges the records in place so that their keys are in order, $K_1 \leq K_2 \leq \ldots \leq K_N$. [*Hint:* The special case that the file is perfectly sorted appears in the answer to exercise 5.2–12; it is possible to combine this with straight insertion without loss of efficiency, since few inversions remain in the file.]

13. [*40*] Implement the internal sorting method suggested in the text at the close of this section, producing a subroutine which sorts random data in $O(N)$ units of time with only $O(\sqrt{N})$ additional memory locations.

14. [*22*] The sequence of playing cards



can be sorted into increasing order A 2 . . . J Q K from top to bottom in two passes, using just two piles for intermediate storage: Deal the cards face down into two piles containing respectively A 2 9 3 10 and 4 J 5 6 Q K 7 8 (from bottom to top); then put the second pile on the first, turn the deck face up, and deal into two piles A 2 3 4 5 6 7 8, 9 10 J Q K. Combine these piles, turn them face up, and you're done.

Prove that the above sequence of cards cannot be sorted into *decreasing* order K Q J . . . 2 A from top to bottom in two passes, even if you are allowed to use up to three piles for intermediate storage. (Dealing must always be from the top of the deck, turning the cards face down as they are dealt. Top to bottom is right to left in the illustration.)

15. [*M25*] Consider the problem of exercise 14 when all cards must be dealt face up instead of face down. Thus, one pass can be used to convert increasing order into decreasing order. How many passes are required?

As soon as an Analytical Engine exists,
it will necessarily guide the future course of the science.
Whenever any result is sought by its aid,
the question will then arise—
By what course of calculation can these
results be arrived at by the machine
in the shortest time?

—CHARLES BABBAGE (1864)

5.3. OPTIMUM SORTING

Now that we have analyzed a great many methods for internal sorting, it is time to turn to a broader question: *What is the best possible way to sort?* Can we place limits on the maximum sorting speeds that will ever be achievable, no matter how clever a programmer might be?

Of course there *is* no best possible way to sort; we must define precisely what is meant by "best," and there is no best possible way to define "best." We have discussed similar questions about the theoretical optimality of algorithms in Sections 4.3.3, 4.6.3, and 4.6.4, where high-precision multiplication and polynomial evaluation were considered. In each case it was necessary to formulate a rather simple definition of a "best possible" algorithm, in order to give sufficient structure to the problem to make it workable. And in each case we ran into interesting problems that are so difficult they still haven't been completely resolved. The same situation holds for sorting; some very interesting discoveries have been made, but there are many fascinating questions still unanswered.

Studies of the inherent complexity of sorting have usually been directed towards minimizing the number of times we make comparisons between keys while sorting n items, or merging m items with n, or selecting the tth largest of an unordered set of n items. Sections 5.3.1, 5.3.2, and 5.3.3 discuss these questions in general, and Section 5.3.4 deals with similar issues under the interesting restriction that the pattern of comparisons must essentially be fixed in advance. Some other types of interesting theoretical questions related to optimum sorting appear in the exercises for Section 5.3.4, and in the discussion of external sorting in Section 5.4.4.

5.3.1. Minimum-Comparison Sorting

The minimum number of key comparisons needed to sort n elements is obviously *zero*, because we have seen radix methods that do no comparisons at all. In fact, it is possible to write MIX programs that are able to sort, although they contain no conditional jump instructions at all! (See exercise 5–6 at the beginning of this chapter.) We have also seen several sorting methods which are based essentially on comparisons of keys, yet their running time in practice is dominated by other considerations such as data movement, housekeeping operations, etc.

Therefore it is clear that comparison counting is not the only way to measure the effectiveness of a sorting method. But it is fun to scrutinize the number of comparisons anyway, since a theoretical study of this subject gives us a good deal of useful insight into the nature of sorting processes, and it also helps us to sharpen our wits for the more mundane problems that may confront us at other times.

In order to rule out radix-sorting methods, which do no comparisons at all, we shall restrict our discussion to sorting techniques which are based solely on

an abstract linear ordering relation "<" between keys, as discussed at the beginning of this chapter. For simplicity, we shall also confine our discussion to the case of *distinct* keys, so that there are only two possible outcomes of any comparison of K_i vs. K_j: either $K_i < K_j$ or $K_i > K_j$. (For an extension of the theory to the general case where equal keys are allowed, see exercises 3 through 12.)

The problem of sorting by comparisons can also be expressed in other equivalent ways. Given a set of n distinct weights and a balance scale, we can ask for the least number of weighings necessary to completely rank the weights in order of magnitude, when the pans of the balance scale can each accommodate only one weight. Alternatively, given a set of n players in a tournament, we can ask for the smallest number of games which suffice to rank all contestants, assuming that the strengths of the players can be linearly ordered (with no ties).

All n-element sorting methods which satisfy the above constraints can be represented in terms of an extended binary tree structure such as that shown in Fig. 34. Each *internal node* (drawn as a circle) contains two indices "i:j" denoting a comparison of K_i versus K_j. The left subtree of this node represents the subsequent comparisons to be made if $K_i < K_j$, and the right subtree represents the actions to be taken when $K_i > K_j$. Each *external node* of the tree (drawn as a box) contains a permutation $a_1\, a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$, denoting the fact that the ordering

$$K_{a_1} < K_{a_2} < \cdots < K_{a_n}$$

has been established. (If we look at the path from the root to this external node, each of the $n - 1$ relationships $K_{a_i} < K_{a_{i+1}}$ for $1 \le i < n$ will be the result of some comparison a_i:a_{i+1} or a_{i+1}:a_i on this path.)

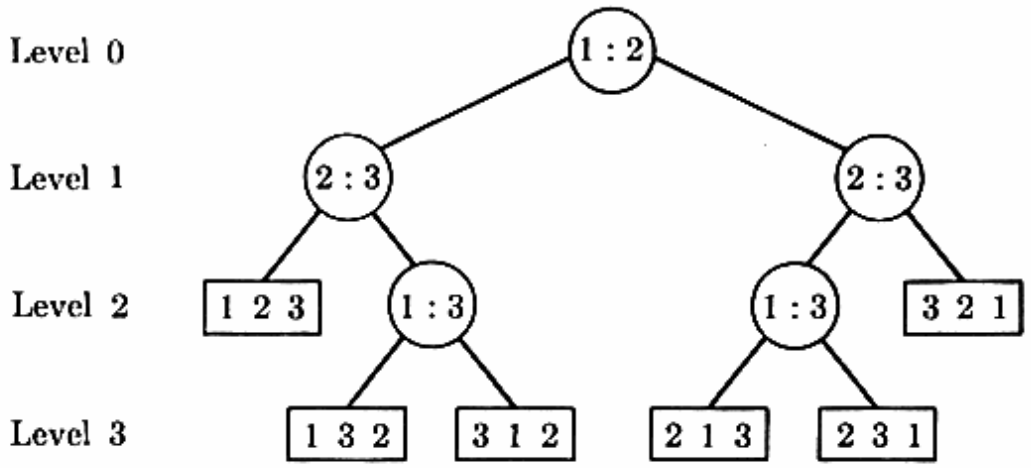


Fig. 34. A comparison tree for sorting three elements.

Thus Fig. 34 represents a sorting method which first compares K_1 with K_2; if $K_1 > K_2$, it goes on (via the right subtree) to compare K_2 with K_3, and then if $K_2 < K_3$ it compares K_1 with K_3; finally if $K_1 > K_3$ it knows that $K_2 < K_3 < K_1$. An actual sorting algorithm will usually also move the keys around in the file, but we are interested here only in the comparisons, so we ignore

all data movement. A comparison of K_i with K_j in this tree always means the *original* keys K_i and K_j, not the keys which might currently occupy the ith and jth positions of the file after the records have been shuffled around.

It is possible to make redundant comparisons; for example, in Fig. 35 there is no reason to compare 3:1, since $K_1 < K_2$ and $K_2 < K_3$ implies that $K_1 < K_3$. No permutation can possibly correspond to the left subtree of node 3:1 in Fig. 35, so that part of the algorithm will never be performed! Since we are interested in minimizing the number of comparisons, we may assume that no redundant comparisons are made; hence we have an extended binary tree structure in which every external node corresponds to a permutation. All permutations of the input keys are possible, and every permutation defines a unique path from the root to an external node; it follows that *there are exactly $n!$ external nodes in a comparison tree which sorts n elements with no redundant comparisons.*

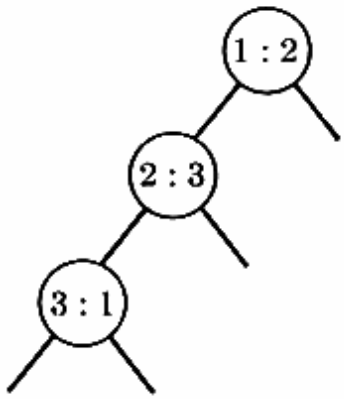


Fig. 35. Example of a redundant comparison.

The best worst case. The first problem that naturally arises is to find comparison trees which minimize the *maximum* number of comparisons made. (Later we shall consider the *average* number of comparisons.)

Let $S(n)$ be the minimum number of comparisons which will suffice to sort n elements. If all the internal nodes of a comparison tree are at levels $< k$, it is obvious that there can be at most 2^k external nodes in the tree. Hence, letting $k = S(n)$, we have

$$n! \leq 2^{S(n)}.$$

Since $S(n)$ is an integer, we can rewrite this formula to obtain the lower bound

$$S(n) \geq \lceil \log_2 n! \rceil. \tag{1}$$

Note that by Stirling's approximation,

$$\lceil \log_2 n! \rceil = n \log_2 n - n/(\ln 2) + \tfrac{1}{2} \log_2 n + O(1), \tag{2}$$

so that about $n \log_2 n$ comparisons are needed.

Relation (1) is often called the "information theoretic lower bound," since cognoscenti of information theory would say that ($\log_2 n!$) "bits of information" are being acquired during a sorting process; each comparison yields at most one "bit of information." Trees such as Fig. 34 have also been called "questionnaires," and their mathematical properties have been explored in Claude Picard's book *Théorie des questionnaires* (Paris: Gauthier-Villars, 1965).

Of all the sorting methods we have seen, the three that require fewest comparisons are binary insertion (cf. Section 5.2.1), tree selection (cf. Section 5.2.3), and straight two-way merging as reflected in Algorithm 5.2.4L. The maximum number of comparisons for binary insertion is readily seen to be

$$B(n) = \sum_{1 \le k \le n} \lceil \log_2 k \rceil = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1 \tag{3}$$

(cf. exercise 1.2.4–42), and the maximum number of comparisons in Algorithm 5.2.4L is given in exercise 5.2.4–14. It turns out (see Section 5.3.3) that tree selection has the same bound on its comparisons as either binary insertion or two-way merging, depending on how the tree is set up. In all three cases we achieve an asymptotic value of $n \log_2 n$; combining these lower and upper bounds for $S(n)$ proves that

$$\lim_{n \to \infty} \frac{S(n)}{n \log_2 n} = 1. \tag{4}$$

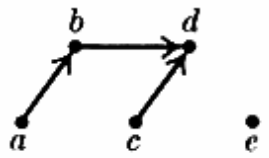
Thus we have an approximate formula for $S(n)$, but it is desirable to obtain more precise information. The following table gives exact values of the above quantities, for small n:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\lceil \log_2 n! \rceil =$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45	49
$B(n) =$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	45	49	54
$L(n) =$	0	1	3	5	9	11	14	17	25	27	30	33	38	41	45	49	65

Here $B(n)$ and $L(n)$ refer respectively to binary insertion and list merging. It can be shown that $B(n) \le L(n)$ for all n (see exercise 2).

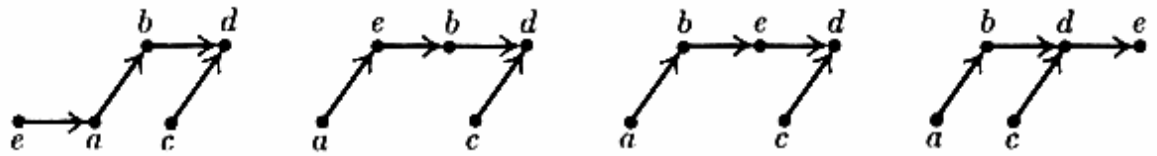
From the above table, we can see that $S(4) = 5$, but $S(5)$ might be either 7 or 8. This brings us back to a problem stated at the beginning of Section 5.2: What is the best way to sort five elements? Can five elements be sorted using only seven comparisons?

The answer is yes, but it is not especially easy to find the method. We begin as if we were sorting four elements by merging, first comparing $K_1 : K_2$, then $K_3 : K_4$, then the larger elements of these pairs. This produces a configuration which may be diagramed as



(5)

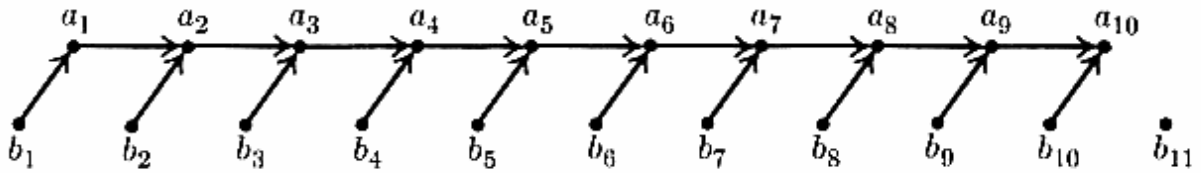
indicating that $a < b < d$ and $c < d$. (It is convenient to represent known ordering relations between elements by drawing directed graphs such as this, where x is known to be less than y if and only if there is a path from x to y in the graph.) At this point we insert the fifth element $K_5 = e$ into its proper place among $\{a, b, d\}$; only two comparisons are needed, since we may compare it first with b and then with a or d. This leaves one of four possibilities,



(6)

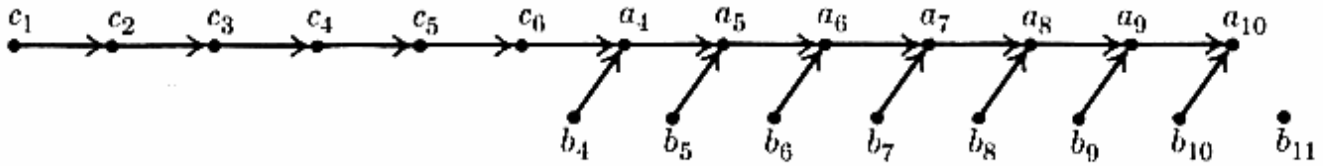
and in each case we can insert c among the remaining elements less than d in two more comparisons. This method for sorting five elements was first discovered by H. B. Demuth [Ph.D. thesis, Stanford University (Oct. 1956), 41–43].

Merge insertion. A pleasant generalization of the above method has been discovered by Lester Ford, Jr. and Selmer Johnson. Since it involves some aspects of merging and some aspects of insertion, we shall call it *merge insertion*. For example, consider the problem of sorting 21 elements. We start by comparing the ten pairs $K_1{:}K_2$, $K_3{:}K_4$, . . . , $K_{19}{:}K_{20}$; then we sort the ten larger elements of the pairs, using merge insertion. As a result we obtain the configuration



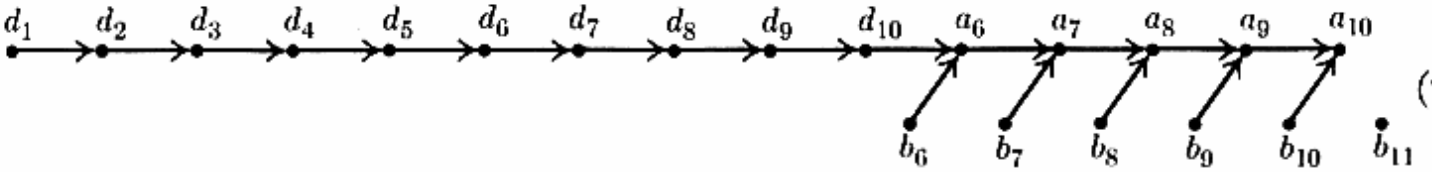
(7)

analogous to (5). The next step is to insert b_3 among $\{b_1, a_1, a_2\}$, then b_2 among the other elements less than a_2; we arrive at the configuration



(8)

Let us call the upper-line-elements the *main chain*. We can insert b_5 into its proper place in the main chain, using three comparisons (first comparing it to c_4, then c_2 or c_6, etc.); then b_4 can be moved into the main chain in three more steps, leading to



(9)

The next step is crucial; is it clear what to do? We insert b_{11} (*not* b_7) into the main chain, using only four comparisons. Then each of b_{10}, b_9, b_8, b_7, b_6 (in this order) can also be inserted into their proper places in the main chain, using at most four comparisons each.

A careful count of the comparisons involved here shows that the 21 elements have been sorted in at most $10 + 22 + 2 + 2 + 3 + 3 + 4 + 4 + 4 + 4 + 4 + 4 = 66$ steps. Since

$$2^{65} < 21! < 2^{66},$$

we also know that no fewer than 66 would be possible in any event; hence

$$S(21) = 66. \tag{10}$$

(Binary insertion would have required 74 comparisons.)

In general, merge insertion proceeds as follows for n elements:

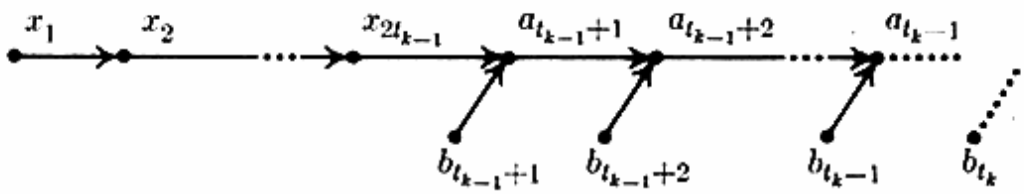
i) Make pairwise comparisons of $\lfloor n/2 \rfloor$ disjoint pairs of elements. (If n is odd, leave one element out.)

ii) Sort the $\lfloor n/2 \rfloor$ larger numbers, found in step (i), by merge insertion.

iii) Name the elements $a_1, a_2, \ldots, a_{\lfloor n/2 \rfloor}, b_1, b_2, \ldots, b_{\lceil n/2 \rceil}$ as in (7), where $a_1 \le a_2 \le \cdots \le a_{\lfloor n/2 \rfloor}$ and $b_i \le a_i$ for $1 \le i \le \lfloor n/2 \rfloor$; call b_1 and the a's the "main chain." Insert the remaining b's into the main chain, using binary insertion, in the following order, leaving out all b_j for $j > \lceil n/2 \rceil$:

$$b_3, b_2; b_5, b_4; b_{11}, b_{10}, \ldots, b_6; \ldots; b_{t_k}, b_{t_k - 1}, \ldots, b_{t_{k-1}+1}; \ldots \tag{11}$$

We wish to define the sequence $(t_1, t_2, t_3, t_4, \ldots) = (1, 3, 5, 11, \ldots)$, which appears in (11), in such a way that each of $b_{t_k}, b_{t_k - 1}, \ldots, b_{t_{k-1}+1}$ can be inserted into the main chain with at most k comparisons. Generalizing (7), (8), and (9), we obtain the diagram



where the main chain up to and including $a_{t_k - 1}$ contains $2t_{k-1} + (t_k - t_{k-1} - 1)$ elements. This number must be less than 2^k; our best bet is to set it equal to $2^k - 1$, so that

$$t_{k-1} + t_k = 2^k. \tag{12}$$

Since $t_1 = 1$, we may set $t_0 = 1$ for convenience, and we find that

$$\begin{aligned} t_k = 2^k - t_{k-1} = 2^k - 2^{k-1} + t_{k-2} = \cdots = 2^k - 2^{k-1} + \cdots + (-1)^k 2^0 \\ = (2^{k+1} + (-1)^k)/3 \end{aligned} \tag{13}$$

by summing a geometric series. (Curiously, this same sequence arose in our study of an algorithm for calculating the greatest common divisor of two integers; cf. exercise 4.5.2–27.)

Let $F(n)$ be the number of comparisons required to sort n elements by merge insertion. Clearly

$$F(n) = \lfloor n/2 \rfloor + F(\lfloor n/2 \rfloor) + G(\lceil n/2 \rceil), \tag{14}$$

where G represents the amount of work involved in step (iii). If $t_{k-1} \le m \le t_k$, we have

$$\begin{aligned} G(m) &= \sum_{1 \le j < k} j(t_j - t_{j-1}) + k(m - t_{k-1}) \\ &= km - (t_0 + t_1 + \cdots + t_{k-1}), \end{aligned} \tag{15}$$

summing by parts. Let us set

$$w_k = t_0 + t_1 + \cdots + t_{k-1} = \lfloor 2^{k+1}/3 \rfloor, \tag{16}$$

so that $(w_0, w_1, w_2, w_3, w_4, \ldots) = (0, 1, 2, 5, 10, 21, \ldots)$. Exercise 13 shows that

$$F(n) - F(n-1) = k \qquad \text{iff} \qquad w_k < n \le w_{k+1}, \tag{17}$$

and the latter condition is equivalent to

$$\frac{2^{k+1}}{3} < n \le \frac{2^{k+2}}{3},$$

$$k + 1 < \log_2 (3n) \le k + 2;$$

hence

$$F(n) - F(n-1) = \lceil \log_2 (\tfrac{3}{4}n) \rceil. \tag{18}$$

(This formula is due to A. Hadian [Ph. D. thesis, Univ. of Minnesota (1969), 38–42].) It follows that $F(n)$ has a remarkably simple expression,

$$F(n) = \sum_{1 \le k \le n} \lceil \log_2 (\tfrac{3}{4}k) \rceil, \tag{19}$$

quite similar to the corresponding formula (3) for binary insertion. A "closed form" for this sum appears in exercise 14.

Equation (19) makes it easy to construct a table of $F(n)$; we have

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\lceil \log_2 n! \rceil =$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45	49
$F(n) =$	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50

$n =$	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
$\lceil \log_2 n! \rceil =$	53	57	62	66	70	75	80	84	89	94	98	103	108	113	118	123
$F(n) =$	54	58	62	66	71	76	81	86	91	96	101	106	111	116	121	126

Note that $F(n) = \lceil \log_2 n! \rceil$ for $1 \le n \le 11$ and for $20 \le n \le 21$, so we know that merge insertion is optimum for these n:

$$S(n) = \lceil \log_2 n! \rceil = F(n) \quad \text{for} \quad 1 \le n \le 11 \quad \text{and} \quad 20 \le n \le 21. \tag{20}$$

Hugo Steinhaus posed the problem of finding $S(n)$ in the second edition of his classic book *Mathematical Snapshots* (Oxford University Press, 1950), pp. 38–39. He described the method of binary insertion, which is the best possible way to sort n objects if we start by sorting $n - 1$ of them first before the nth is considered; and he conjectured that binary insertion is optimum in general. Several years later [*Calcutta Math. Soc. Golden Jubilee Commemoration* **2** (1959), 323–327], he reported that two of his colleagues, S. Trybuła and C. Ping, had "recently" disproved his conjecture, and that they had determined $S(n)$ for $n \le 11$. Trybuła and Ping may have independently discovered the method of merge insertion, which was published soon afterwards by Ford and Johnson [*AMM* **66** (1959), 387–389].

After the discovery of merge insertion, the first unknown value of $S(n)$ was $S(12)$. Table 1 shows that 12! is quite close to 2^{29}, so that the existence of a 29-step sorting procedure for 12 elements is somewhat unlikely. An exhaustive search (about 60 hours on a Maniac II computer) was therefore carried out by Mark Wells, who discovered that $S(12) = 30$ [*Proc. IFIP Congress* 65 **2** (1965), 497–498]. Thus the merge insertion procedure turns out to be optimum for $n = 12$ as well.

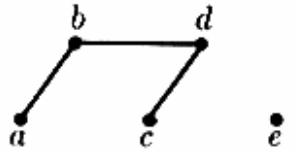
Table 1

VALUES OF FACTORIALS IN BINARY NOTATION

1	= 1!
10	= 2!
110	= 3!
11000	= 4!
1111000	= 5!
1011010000	= 6!
1001110110000	= 7!
1001110110000000	= 8!
1011000100110000000	= 9!
11011101011111100000000	= 10!
100110000100001010100000000	= 11!
1110010001100111111000000000000	= 12!
101110011001010001100110000000000	= 13!
1010001001101111110110010100000000000	= 14!
100110000011101110111011101011000000000000	= 15!
1001100000111011101110111010110000000000000000	= 16!

***A slightly deeper analysis.** In order to study $S(n)$ more carefully, let us look more closely at partial ordering diagrams such as (5). After several comparisons

have been made, we can represent the knowledge we have gained in terms of a directed graph. This directed graph contains no cycles, in view of the transitivity of the $<$ relation, so we can draw it in such a way that all arcs go from left to right; it is therefore convenient to leave arrows off the diagram. In this way (5) becomes



(21)

If G is such a directed graph, let $T(G)$ be the number of permutations consistent with G, i.e., the number of ways to assign the integers $\{1, 2, \ldots, n\}$ to the vertices of G so that the number on vertex x is less than the number on vertex y whenever $x \to y$ in G. For example, one of the permutations consistent with (21) has $a = 1$, $b = 4$, $c = 2$, $d = 5$, $e = 3$. We have studied $T(G)$ for various G in Section 5.1.4, where we observed that $T(G)$ is the number of ways in which G can be "topologically sorted."

If G is a graph on n elements which can be obtained after k comparisons, we define the *efficiency* of G to be

$$E(G) = \frac{n!}{2^k T(G)}. \tag{22}$$

(This idea is due to Frank Hwang and Shen Lin.) Strictly speaking, the efficiency is not a function of the graph G alone, it depends on the way we arrived at G during a sorting process, but it is convenient to be a little careless in our language. After making one more comparison, between elements i and j, we obtain two graphs G_1 and G_2, one for the case $K_i < K_j$ and one for the case $K_i > K_j$. Clearly

$$T(G) = T(G_1) + T(G_2).$$

If $T(G_1) \geq T(G_2)$, we have

$$T(G) \leq 2T(G_1),$$

$$E(G_1) = \frac{n!}{2^{k+1}T(G_1)} = \frac{E(G)T(G)}{2T(G_1)} \leq E(G). \tag{23}$$

Therefore each comparison leads to a graph of less or equal efficiency; we can't improve the efficiency by making further comparisons.

Note that when G has no arcs at all, we have $k = 0$ and $T(G) = n!$, so the initial efficiency is 1. And when G is a graph representing the final result of sorting, G looks like a straight line, and $T(G) = 1$. Thus, for example, if we want to find a sorting procedure that sorts five elements in seven or less steps, we must obtain the linear graph whose efficiency is $5!/(2^7 \times 1) = 120/128 = 15/16$. It follows that all of the graphs arising in the sorting procedure must have efficiency $\geq \frac{15}{16}$; if any less efficient graph were to appear, at least one of its descendants would also be less efficient, and

we would ultimately reach a linear graph whose efficiency is $< \frac{15}{16}$. In general, this argument proves that all graphs corresponding to the tree nodes of a sorting procedure for n elements must have efficiency $\geq n!/2^l$, where $l+1$ is the number of levels of the tree. This is another way to prove that $S(n) \geq \lceil \log_2 n! \rceil$, although the argument is not really much different than what we said before.

The graph (21) has efficiency 1, since $T(G) = 15$ and since G has been obtained in three comparisons. In order to see what vertices should be compared next, we can form the *comparison matrix*

$$C(G) = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 15 & 10 & 15 & 11 \\ 0 & 0 & 5 & 15 & 7 \\ 5 & 10 & 0 & 15 & 9 \\ 0 & 0 & 0 & 0 & 3 \\ 4 & 8 & 6 & 12 & 0 \end{pmatrix} \end{array}, \tag{24}$$

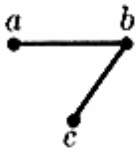
where C_{ij} is $T(G_1)$ for the graph G_1 obtained by adding the arc $i \to j$ to G. For example, if we compare K_c with K_e, the 15 permutations consistent with G split up into $C_{ec} = 6$ having $K_e < K_c$ and $C_{ce} = 9$ having $K_c < K_e$. The latter graph would have efficiency $15/(2 \times 9) = \frac{5}{6} < \frac{15}{16}$, so it could not lead to a seven-step sorting procedure. The next comparison *must* be $K_b : K_e$ in order to keep the efficiency $\geq \frac{15}{16}$.

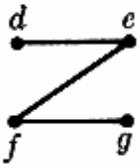
The concept of "efficiency" is especially useful when we consider the "connected components" of graphs. Consider for example the graph

$G =$

;

it has two components,

$G' =$ 

and $G'' =$ 

with no arcs connecting G' to G'', so it has been formed by making some comparisons entirely within G' and others entirely within G''. In general, assume that $G = G' + G''$ has no arcs between G' and G'', where G' and G'' have respectively n' and n'' vertices; it is easy to see that

$$T(G) = \binom{n' + n''}{n'} T(G')T(G''), \tag{25}$$

since each consistent permutation of G is obtained by choosing n' elements to

assign to G' and then making consistent permutations within G' and G'' independently. If k' comparisons have been made within G' and k'' within G'', we have the basic result

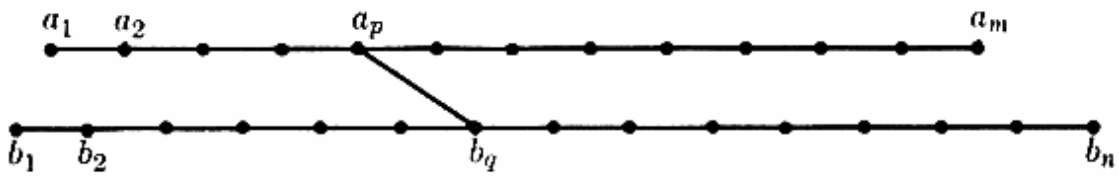
$$E(G) = \frac{(n' + n'')!}{2^{k'+k''}T(G)} = \frac{n'!}{2^{k'}T(G')} \cdot \frac{n''!}{2^{k''}T(G'')} = E(G')E(G''), \tag{26}$$

showing that the efficiency of a graph is related in a simple way to the efficiency of its components. Therefore we may restrict consideration to graphs having only one component.

Now suppose that G' and G'' are one-component graphs, and suppose that we want to hook them together by comparing a vertex x of G' with a vertex y of G''. We want to know how efficient this will be. For this purpose we need a function which can be denoted by

$$\binom{p}{m} < \binom{q}{n}, \tag{27}$$

defined to be the number of permutations consistent with the graph



(28)

Thus $\binom{p}{m}<\binom{q}{n}$ is $\binom{m+n}{m}$ times the probability that the pth smallest of a set of m numbers is less than the qth smallest of an independently chosen set of n numbers. Exercise 17 shows that we can express $\binom{p}{m}<\binom{q}{n}$ in two ways in terms of binomial coefficients,

$$\begin{aligned}\binom{p}{m} > \binom{q}{n} &= \sum_{0\le k<q} \binom{m-p+n-k}{m-p}\binom{p-1+k}{p-1} \\ &= \sum_{p\le j\le m} \binom{n-q+m-j}{n-q}\binom{q-1+j}{q-1}.\end{aligned} \tag{29}$$

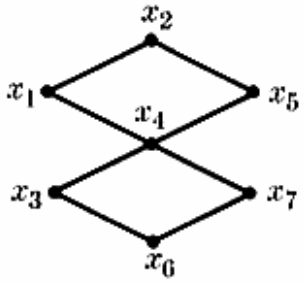
(Incidentally, it is by no means obvious on algebraic grounds that these two sums of products of binomial coefficients should come out to be equal.) We also have the formulas

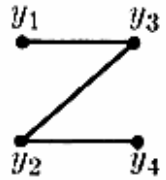
$$\binom{p}{m} < \binom{q}{n} + \binom{q}{n} < \binom{p}{m} = \binom{m+n}{m}, \tag{30}$$

$$\binom{q}{n} < \binom{p}{m} = \binom{m+1-p}{m} < \binom{n+1-q}{n}. \tag{31}$$

For definiteness, let us now consider the two graphs

$$G' = \quad , \qquad G'' = \qquad (32)$$





It is not hard to show by direct enumeration that $T(G') = 42$ and $T(G'') = 5$; so if G is the 11-vertex graph having G' and G'' as components, we have $T(G) = \binom{11}{4} \cdot 42 \cdot 5 = 69300$ by Eq. (25). This is a formidable number of permutations to list, if we want to know how many of them have $x_i < y_j$ for each i and j. But the calculation can be done by hand, in less than an hour, as follows. We form the matrices $A(G')$ and $A(G'')$, where A_{ik} is the number of consistent permutations of G' (or G'') in which x_i (or y_i) is equal to k. Thus the number of permutations of G in which x_i is less than y_j is the (ip) element of $A(G')$ times $\binom{p}{7}<\binom{q}{4}$ times the (jq) element of $A(G'')$, summed over $1 \le p \le 7$ and $1 \le q \le 4$. In other words, we want to form the matrix product $A(G') \cdot L \cdot A(G'')^T$, where $L_{pq} = \binom{p}{7}<\binom{q}{4}$. This comes to

$$\begin{pmatrix} 21 & 16 & 5 & 0 & 0 & 0 & 0 \\ 0 & 5 & 10 & 12 & 10 & 5 & 0 \\ 21 & 16 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 18 & 12 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 16 & 21 \\ 0 & 5 & 10 & 12 & 10 & 5 & 0 \\ 0 & 0 & 0 & 0 & 5 & 16 & 21 \end{pmatrix} \begin{pmatrix} 210 & 294 & 322 & 329 \\ 126 & 238 & 301 & 325 \\ 70 & 175 & 265 & 315 \\ 35 & 115 & 215 & 295 \\ 15 & 65 & 155 & 260 \\ 5 & 29 & 92 & 204 \\ 1 & 8 & 36 & 120 \end{pmatrix} \begin{pmatrix} 2 & 3 & 0 & 0 \\ 2 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 0 & 0 & 3 & 2 \end{pmatrix} = \begin{pmatrix} 48169 & 42042 & 66858 & 64031 \\ 22825 & 16005 & 53295 & 46475 \\ 48169 & 42042 & 66858 & 64031 \\ 22110 & 14850 & 54450 & 47190 \\ 5269 & 2442 & 27258 & 21131 \\ 22825 & 16005 & 53295 & 46475 \\ 5269 & 2442 & 27258 & 21131 \end{pmatrix}.$$

Thus the "best" way to hook up G' and G'' is to compare x_1 with y_2; this gives 42042 cases with $x_1 < y_2$ and $69300 - 42042 = 27258$ cases with $x_1 > y_2$. (By symmetry, we could also compare x_3 with y_2, x_5 with y_3, or x_7 with y_3, leading to essentially the same results.) The efficiency of the resulting graph for $x_1 < y_2$ is

$$\frac{69300}{84084} E(G')E(G''),$$

which is none too good; hence it is probably a bad idea to hook G' up with G'' in any sorting method! The point of this example is that we are able to make such a decision without excessive calculation.

These ideas can be used to provide independent confirmation of Mark Well's proof that $S(12) = 30$. Starting with a graph containing one vertex, we can repeatedly try to add a comparison to one of our graphs G or to a pair of graph components G' and G'' in such a way that the two resulting graphs have 12 or less vertices and efficiency $\ge 12!/2^{29} \approx 0.89221$. Whenever this is possible, we take the resulting graph of least efficiency and add it to our set, unless it is isomorphic to a graph we already have included (or to the dual of such a graph, obtained by reversing the order). If both of the resulting graphs

have the same efficiency, we arbitrarily choose one of them. The first 24 graphs obtained in this way are displayed in Fig. 36 together with their efficiencies.

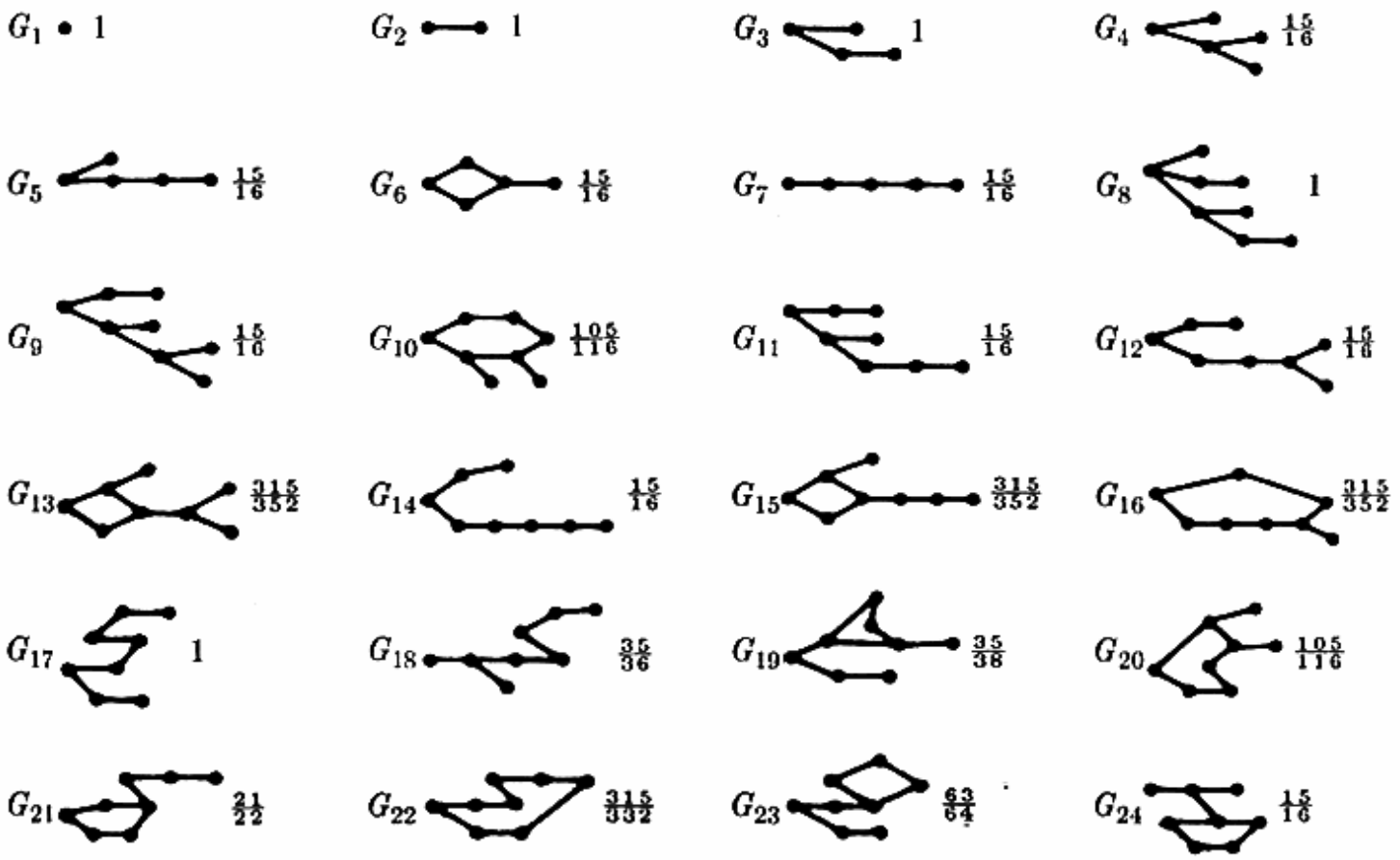
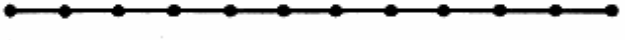


Fig. 36. Some graphs and their efficiencies, obtained at the beginning of a long proof that $S(12) > 29$.

Exactly 1594 graphs were generated, by computer, before this process terminated. Since the graph  was not obtained, we may conclude that $S(12) > 29$. It is plausible that a similar experiment could be performed to deduce that $S(22) > 70$ in a fairly reasonable amount of time, since $22!/2^{70} \approx 0.952$ requires extremely high efficiency to sort in 70 steps. (Only 92 of the 1594 graphs found on 12 or less vertices had such high efficiency.)

The intermediate results suggest strongly that $S(13) = 33$, so that merge insertion would not be optimum when $n = 13$. But as yet nobody has discovered *any* n such that $S(n) < F(n)$.

It should certainly be possible to prove that $S(16) < F(16)$, since $F(16)$ takes no fewer comparisons than sorting ten elements with $S(10)$ comparisons and then inserting six others by binary insertion, one at a time. There must be a way to improve upon such a method!

The average number of comparisons. So far we have been considering procedures which are best possible in the sense that their worst case isn't bad; we have looked for "minimax" procedures which minimize the *maximum* number of comparisons. Now let us look for a "minimean" procedure which minimizes the *average* number of comparisons, assuming that the input is random so that each permutation is equally likely.

Consider once again the tree representation of a sorting procedure, as shown in Fig. 34. The average number of comparisons in that tree is

$$\frac{2+3+3+3+3+2}{6} = 2\tfrac{2}{3},$$

averaging over all permutations. In general, the average number of comparisons in a sorting method is the *external path length of the tree* divided by $n!$. (Recall that the external path length is the sum of the distances from the root to each of the external nodes; see Section 2.3.4.5.) It is easy to see from the considerations in Section 2.3.4.5 that the minimum external path length occurs in a binary tree with N external nodes if there are $2^q - N$ external nodes at level $q - 1$ and $2N - 2^q$ at level q, where $q = \lceil \log_2 N \rceil$. (The root is at level zero.) The minimum external path length is therefore

$$(q-1)(2^q - N) + q(2N - 2^q) = (q+1)N - 2^q. \tag{33}$$

The minimum path length can also be characterized in another interesting way: *An extended binary tree has minimum external path length if and only if there is a number l such that all external nodes appear on levels l and $l+1$.* (See exercise 20.)

If we set $q = \log_2 N + \theta$, where $0 \le \theta < 1$, the formula for minimum external path length becomes

$$N(\log_2 N + 1 + \theta - 2^\theta). \tag{34}$$

The function $1 + \theta - 2^\theta$ is shown in Fig. 37; for $0 < \theta < 1$ it is positive but very small, never exceeding

$$1 - (1 + \ln\ln 2)/(\ln 2) = 0.08607\ 13320\ 55934+. \tag{35}$$

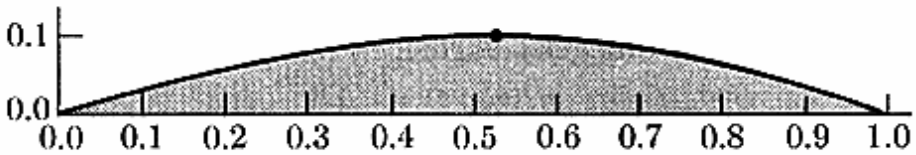


Fig. 37. The function $1 + \theta - 2^\theta$.

Thus the minimum possible average path length, obtained by dividing (34) by N, is never less than $\log_2 N$ and never more than $\log_2 N + 0.0861$. (This result was first obtained by A. Gleason in an unpublished memorandum (1956).)

Now if we set $N = n!$ we get a lower bound for the average number of comparisons in any sorting scheme. Note that this lower bound is $\log_2 n! + O(1) = n \log_2 n - n/(\ln 2) + O(\log n)$.

Let $\overline{F}(n)$ be the average number of comparisons performed by the merge insertion algorithm; we have

$n =$	1	2	3	4	5	6	7	8
lower bound (33) $=$	0	2	16	112	832	6896	62368	619904
$n!\,\overline{F}(n) =$	0	2	16	112	832	6912	62784	623232

Thus merge insertion is optimum in both senses for $n \leq 5$, but for $n = 6$ it averages $6912/720 = 9.6$ comparisons while our lower bound says that an average of $6896/720 = 9.577777\cdots$ comparisons may be possible. A moment's reflection shows why this is true: Some "fortunate" permutations of six elements are sorted after only eight comparisons by merge insertion, so the comparison tree has external nodes appearing on three levels instead of two. This forces the overall path length to be higher. Exercise 24 shows that it is possible to construct a six-element sorting procedure which requires nine or ten comparisons in each case; it follows that this method is superior to merge insertion, on the average, and no worse than merge insertion in its worst case.

When $n = 7$, Y. Césari [thesis (Paris, 1968), p. 37] has shown that no sorting method can attain the lower bound 62368 on external path length. (It is possible to prove this fact by hand, using the results of exercise 22.) On the other hand, he has constructed procedures which do achieve the lower bound (33) when $n = 9$ or 10. In general, the problem of minimizing the average number of comparisons turns out to be substantially more difficult than the problem of determining $S(n)$. It may even be true that, for some n, all methods which minimize the *average* number of comparisons require *more* than $S(n)$ comparisons in their worst case.

EXERCISES

1. [*20*] Draw the comparison trees for sorting four elements by the method of (a) binary insertion; (b) straight two-way merging. What are the external path lengths of these trees?

2. [*M24*] Prove that $B(n) \le L(n)$, and find all n for which equality holds.

3. [*M22*] When equality between keys is allowed, there are 13 possible outcomes when sorting three elements:

$$\begin{array}{ccc} K_1 = K_2 = K_3 & K_1 = K_2 < K_3 & K_1 = K_3 < K_2 \\ K_2 = K_3 < K_1 & K_1 < K_2 = K_3 & K_2 < K_1 = K_3 \\ K_3 < K_1 = K_2 & K_1 < K_2 < K_3 & K_1 < K_3 < K_2 \end{array}$$

$$K_2 < K_1 < K_3 \qquad K_2 < K_3 < K_1 \qquad K_3 < K_1 < K_2 \qquad K_3 < K_2 < K_1.$$

Let P_n denote the number of possible outcomes when n elements are sorted with ties allowed, so that $(P_0, P_1, P_2, P_3, P_4, P_5, \ldots) = (1, 1, 3, 13, 75, 541, \ldots)$. Prove that the generating function $P(z) = \sum_{n\ge 0} P_n z^n/n!$ is equal to $1/(2 - e^z)$. *Hint:* Show that

$$P_n = \sum_{k>0} \binom{n}{k} P_{n-k} \qquad \text{when} \qquad n > 0.$$

4. [*HM27*] (O. A. Gross.) Determine the asymptotic value of the numbers P_n of exercise 3, as $n \to \infty$. [*Possible hint:* Consider the partial fraction expansion of cot z.]

5. [*16*] When keys can be equal, each comparison may have three results instead of two: $K_i < K_j$, $K_i = K_j$, $K_i > K_j$. Sorting algorithms for this general situation can be represented as extended *ternary* trees, in which each internal node $i{:}j$ has three subtrees; the left, middle, and right subtrees correspond respectively to the three possible outcomes of the comparison.

Draw an extended ternary tree which defines a sorting algorithm for $n = 3$, when equal keys are allowed. There should be 13 external nodes, corresponding to the 13 possible outcomes listed in exercise 3.

▶ **6.** [*M22*] Let $S'(n)$ be the minimum number of comparisons necessary to sort n elements and to determine all equalities between keys, when each comparison has three outcomes as in exercise 5. The "information theory" argument of the text can readily be generalized to show that $S'(n) \geq \lceil \log_3 P_n \rceil$, where P_n is the function studied in exercises 3 and 4; but prove that, in fact, $S'(n) = S(n)$.

7. [*20*] Draw an extended ternary tree in the sense of exercise 5 for sorting four elements, when it is known that all keys are either 0 or 1. (Thus if $K_1 < K_2$ and $K_3 < K_4$, we know that $K_1 = K_3$ and $K_2 = K_4$!) Use the minimum average number of comparisons, assuming that the 2^4 possible inputs are equally likely.

8. [*26*] Draw an extended ternary tree as in exercise 7, for sorting four elements, when it is known that all keys are either -1, 0, or $+1$. Use the minimum average number of comparisons, assuming that the 3^4 possible inputs are equally likely.

9. [*M20*] When sorting n elements as in exercise 7, knowing that all keys are 0 or 1, what is the minimum number of comparisons in the worst case?

▶ **10.** [*M25*] When sorting n elements as in exercise 7, knowing that all keys are 0 or 1, what is the minimum *average* number of comparisons as a function of n?

11. [*HM25*] When sorting n elements as in exercise 5, and knowing that all keys are members of the set $\{1, 2, \ldots, m\}$, let $S_m(n)$ be the minimum number of comparisons needed in the worst case. [Thus by exercise 6, $S_n(n) = S(n)$.] Prove that, for fixed m, $S_m(n)$ is asymptotically $n \log_2 m$ as $n \to \infty$.

▶ **12.** [*M25*] (W. G. Bouricius, circa 1954.) Suppose that equal keys may occur, but we merely want to sort the elements $\{K_1, K_2, \ldots, K_n\}$ so that a permutation $a_1 a_2 \ldots a_n$ is determined with $K_{a_1} \leq K_{a_2} \leq \cdots \leq K_{a_n}$; we do not need to know whether or not equality occurs between K_{a_i} and $K_{a_{i+1}}$.

Let us say that a comparison tree sorts a sequence of keys *strongly* if it will sort the sequence in the above sense no matter which branch is taken below the nodes $i{:}j$ for which $K_i = K_j$. (The tree is binary, not ternary.)

a) Prove that a comparison tree with no redundant comparisons sorts every sequence of keys strongly if and only if it sorts every sequence of distinct keys.

b) Prove that a comparison tree sorts every sequence of keys strongly if and only if it sorts every sequence of zeros and ones strongly.

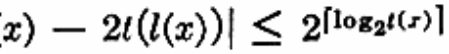
13. [*M28*] Prove (17).

14. [*M24*] Find a "closed form" for the sum (19).

15. [*M21*] Determine the asymptotic behavior of $B(n)$ and $F(n)$ up to $O(\log n)$. [*Hint:* Show that in both cases the coefficient of n involves the function shown in Fig. 37.]

16. [*HM26*] (F. Hwang and S. Lin.) Prove that $F(n) > \lceil \log_2 n! \rceil$ for $n \geq 22$.

17. [*M20*] Prove (29).

18. [*20*] If the procedure which starts in Fig. 36 had produced the graph  with efficiency $12!/2^{29}$, would this have proved that $S(12) = 29$?

19. [*40*] Experiment with the following heuristic rule for deciding which pair of elements to compare next while designing a comparison tree: At each stage of sorting $\{K_1, \ldots, K_n\}$, let u_i be the number of keys known to be $\leq K_i$ as a result of the comparisons made so far, and let v_i be the number of keys known to be $\geq K_i$, for $1 \leq i \leq n$. Renumber the keys in terms of increasing u_i/v_i, so that $u_1/v_1 \leq u_2/v_2 \leq \cdots \leq u_n/v_n$. Now compare $K_i{:}K_{i+1}$ for some i which minimizes $|u_i v_{i+1} - u_{i+1} v_i|$. (Although this method is based on far less information than a full comparison matrix as in (24), it appears to give optimum results in many cases.)

▶**20.** [*M26*] Prove that an extended binary tree has minimum external path length if and only if there is a number l such that all external nodes appear on levels l and $l+1$ (or perhaps all on a single level l).

21. [*M21*] The *height* of an extended binary tree is the maximum level number of its external nodes. If x is an internal node of an extended binary tree, let $t(x)$ be the number of external nodes below x, and let $l(x)$ denote the root of x's left subtree. If x is an external node, let $t(x) = 1$. Prove that an extended binary tree has minimum height among all binary trees with the same number of nodes if and only if

$$|t(x) - 2t(l(x))| \leq 2^{\lceil \log_2 t(x) \rceil} - t(x)$$

for all internal nodes x.

22. [*M24*] Continuing exercise 21, prove that a binary tree has minimum external path length among all binary trees with the same number of nodes if and only if

$$|t(x) - 2t(l(x))| \leq 2^{\lceil \log_2 t(x) \rceil} - t(x) \qquad \text{and} \qquad |t(x) - 2t(l(x))| \leq t(x) - 2^{\lfloor \log_2 t(x) \rfloor}$$

for all internal nodes x. [Thus, for example, if $t(x) = 67$, we must have $t(l(x)) = 32$, 33, 34, or 35. If we merely wanted to minimize the height of the tree we could have $3 \leq t(l(x)) \leq 64$, by the preceding exercise.]

23. [*10*] The text proves [cf. (34)] that the average number of comparisons made by any sorting method for n elements must be at least $\lceil \log_2 n! \rceil \approx n \log_2 n$. But multiple list insertion (Algorithm 5.2.1M) takes only $O(n)$ units of time on the average. How can this be?

24. [*27*] (C. Picard.) Find a sorting tree for six elements such that all external nodes appear on levels 10 and 11.

25. [*11*] If there were a sorting procedure for seven elements which achieves the minimum average number of comparisons predicted by the use of Eq. (34), how many external nodes would there be on level 13?

26. [*M42*] Find a sorting procedure for seven elements which minimizes the average number of comparisons performed.

▶**27.** [*20*] Suppose it is known that the configurations ($K_1 < K_2 < K_3$, $K_1 < K_3 < K_2$, $K_2 < K_1 < K_3$, $K_2 < K_3 < K_1$, $K_3 < K_1 < K_2$, $K_3 < K_2 < K_1$) occur with respective probabilities (.01, .25, .01, .24, .25, .24). Find a comparison tree which sorts these three elements with the smallest average number of comparisons.

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

29. [*M25*] (S. M. Chase.) Let $a_1 a_2 \ldots a_n$ be a permutation of $\{1, 2, \ldots, n\}$. Prove that any algorithm which decides whether this permutation is even or odd (i.e., whether it has an even or odd number of inversions), based solely on comparisons between the a's, must make at least $n \log_2 n$ comparisons, even though the algorithm has only two possible outcomes.

30. [*M23*] (*Optimum exchange sorting.*) Every exchange sorting algorithm as defined in Section 5.2.2 can be represented as a *comparison-exchange tree*, namely a binary tree structure whose internal nodes have the form $(i:j)$ for $i < j$, interpreted as the following operation: "If $K_i \leq K_j$, continue by taking the left branch of the tree; if $K_i > K_j$, continue by interchanging records i and j and then taking the right branch of the tree." When an external node is encountered, it must be true that $K_1 \leq K_2 \leq \cdots \leq K_n$. Thus, a comparison-exchange tree differs from a comparison tree in that it specifies data movement as well as comparison operations.

Let $S_e(n)$ denote the minimum number of comparison-exchanges needed, in the worst case, to sort n elements by means of a comparison-exchange tree. Prove that $S_e(n) \leq S(n) + n - 1$.

31. [*M38*] Continuing exercise 30, prove that $S_e(5) = 8$.

32. [*M42*] Continuing exercise 31, investigate $S_e(n)$ for small values of $n > 5$.

33. [*M30*] (T. N. Hibbard.) A *real-valued search tree* of order x and resolution δ is an extended binary tree in which all nodes contain a nonnegative real value, such that (i) the value in each external node is $\leq \delta$; (ii) the value in each internal node is $\leq$ the sum of the values in its two sons; (iii) the value in the root is x. The *weighted path length* of such a tree is defined to be the sum, over all external nodes, of the level of that node times the value it contains.

Prove that a real-valued search tree of order x and resolution 1 has minimum weighted path length, taken over all such trees of the same order and resolution, if and only if equality holds in (ii) and the following further conditions hold for all pairs of values x_0 and x_1 that are contained in brother nodes: (iv) There is no integer $k \geq 0$ such that $x_0 < 2^k < x_1$ or $x_1 < 2^k < x_0$; (v) $\lceil x_0 \rceil - x_0 + \lceil x_1 \rceil - x_1 < 1$. (In particular if x is an integer, condition (v) implies that all values in the tree are integers, and condition (iv) is equivalent to the result of exercise 22.)

Also prove that the corresponding minimum weighted path length is $x\lceil \log_2 x \rceil + \lceil x \rceil - 2^{\lceil \log_2 x \rceil}$.

34. [*M50*] Determine the exact value of $S(n)$ for infinitely many n.

*5.3.2. Minimum-Comparison Merging

Let us now consider a related question: What is the best way to merge an ordered set of m elements with an ordered set of n? Denoting the elements to be merged by

$$\begin{aligned} A_1 < A_2 < \cdots < A_m, \\ B_1 < B_2 < \cdots < B_n, \end{aligned} \tag{1}$$

we shall assume as in Section 5.3.1 that the $m + n$ elements are distinct. The

A's may appear among the B's in $\binom{m+n}{m}$ ways, so the arguments we have used for the sorting problem tell us immediately that at least

$$\left\lceil \log_2 \binom{m+n}{m} \right\rceil \tag{2}$$

comparisons are required. If we set $m = \alpha n$ and let $n \to \infty$, while α is fixed, Stirling's approximation tells us that

$$\log_2 \binom{\alpha n + n}{\alpha n} = n((1+\alpha)\log_2(1+\alpha) - \alpha \log_2 \alpha) - \tfrac{1}{2}\log_2 n + O(1). \tag{3}$$

The normal merging procedure, Algorithm 5.2.4M, takes $m + n - 1$ comparisons in its worst case.

Let $M(m, n)$ denote the function analogous to $S(n)$, namely the minimum number of comparisons which will always suffice to merge m things with n. By the observations we have just made,

$$\left\lceil \log_2 \binom{m+n}{m} \right\rceil \le M(m, n) \le m + n - 1 \qquad \text{for all} \qquad m, n \ge 1. \tag{4}$$

Formula (3) shows how far apart this lower bound and upper bound can be. When $\alpha = 1$ (i.e., $m = n$), the lower bound is $2n - \frac{1}{2}\log_2 n + O(1)$, so both bounds have the right order of magnitude but the difference between them can be arbitrarily large. When $\alpha = 0.5$ (i.e., $m = \frac{1}{2}n$), the lower bound is

$$\tfrac{3}{2}n(\log_2 3 - \tfrac{2}{3}) + O(\log n),$$

which is about $\log_2 3 - \frac{2}{3} \approx 0.918$ times the upper bound. And as α decreases, the bounds get farther and farther apart, since the standard merging algorithm is primarily designed for files with $m \approx n$.

When $m = n$, the merging problem has a fairly simple solution; it turns out that the *lower* bound of (4), not the upper bound, is at fault. The following theorem was discovered independently by R. L. Graham and R. M. Karp about 1968:

Theorem M. $M(m, m) = 2m - 1$, *for* $m \ge 1$.

Proof. Consider any algorithm which merges $A_1 < \cdots < A_m$ with $B_1 < \cdots < B_m$. When it compares $A_i{:}B_j$, take the branch $A_i < B_j$ if $i < j$, the branch $A_i > B_j$ if $i \ge j$. Merging must eventually terminate with the configuration

$$B_1 < A_1 < B_2 < A_2 < \cdots < B_m < A_m, \tag{5}$$

since this is consistent with all of the branches taken. And each of the $2m - 1$ comparisons $B_1{:}A_1$, $A_1{:}B_2$, $B_2{:}A_2, \ldots, B_m{:}A_m$ must have been made explicitly, or else there would be at least two configurations consistent with

the known facts. For example, if A_1 has not been compared to B_2, the configuration

$$B_1 < B_2 < A_1 < A_2 < \cdots < B_m < A_m$$

is indistinguishable from (5). ▮

A simple modification of this proof yields the companion formula

$$M(m, m+1) = 2m, \qquad \text{for} \qquad m \geq 0. \tag{6}$$

Oracular lower bounds. Theorem M shows that the "information theoretic" lower bound can be considerably lower than the true lower bound; thus the technique used to prove Theorem M gives us another way to discover lower bounds. It is convenient to regard this proof technique as the construction of an "oracle": When merging by some algorithm, we decide to compare $A_i{:}B_j$ and we say, "Pray tell me, O thou wise Oracle, which is the larger, A_i or B_j?" After we have made a suitable offering, the answer comes back:

I am Sir Oracle, and I ope my lips to say that A_i is the greater.

If we can construct a suitable oracle, as in the proof of Theorem M, we can ensure that every valid merging algorithm will have to ask a rather large number of questions.

It turns out to be important to consider the idea of a *constrained oracle*, which has been enjoined to give specific answers to certain questions. (The merging method which consults the oracle does not know, of course, that the oracle is constrained, so it may be necessary to ask the question anyway.) The constraints we speak of apply to the left and right ends of the files. Left constraints are symbolized by

. (meaning no left constraint),
\ (meaning that all pronouncements must be consistent with $A_1 < B_1$),
/ (meaning that all pronouncements must be consistent with $A_1 > B_1$);

right constraints are symbolized by

. (meaning no right constraint),
\ (meaning that all pronouncements must be consistent with $A_m < B_n$),
/ (meaning that all pronouncements must be consistent with $A_m > B_n$).

There are nine kinds of oracles, denoted by $\nabla M \phi$, where ∇ is a left constraint and ϕ is a right constraint. For example, a "$\backslash M \backslash$" oracle must say that $A_1 < B_j$ and $A_i < B_n$; a "$.M.$" oracle is unconstrained. For some small m and n, constrained oracles of certain kinds are impossible; when $m = 1$ we obviously can't have a "$\backslash M /$" oracle.

Let us now construct a rather complicated, but very wise, oracle for merging. It does not always produce optimum results, but it gives lower bounds

which cover a lot of interesting cases. Given m, n, and the left and right constraints ∇ and ϕ, suppose the oracle is asked which is the greater of A_i or B_j. The oracle can use six strategies in general which reduce the problem to cases of smaller $m + n$:

Strategy $A(k, l)$, for $i \leq k \leq m$ and $1 \leq l \leq j$. Say that $A_i < B_j$, and require that subsequent operations merge $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_{l-1}\}$ and $\{A_{k+1}, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$. Thus future questions $A_p{:}B_q$ will be answered $A_p < B_q$ if $p \leq k$ and $q \geq l$; $A_p > B_q$ if $p > k$ and $q < l$; they will be handled by a $(k, l - 1, \nabla, .)$ oracle if $p \leq k$ and $q < l$; by an $(m - k, n + 1 - l, ., \phi)$ oracle if $p > k$ and $q \geq l$.

Strategy $B(k, l)$, for $i \leq k \leq m$ and $1 \leq l < j$. Say that $A_i < B_j$, and require that subsequent operations merge $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_{k+1}, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$, stipulating that $A_k < B_l < A_{k+1}$. (Note that B_l appears in both lists to be merged. The condition $A_k < B_l < A_{k+1}$ ensures that merging one group gives no information that could help to merge the other.) Thus future questions $A_p{:}B_q$ will be answered $A_p < B_q$ if $p \leq k$ and $q \geq l$; $A_p > B_q$ if $p > k$ and $q \leq l$; they will be handled by a $(k, l, \nabla, \backslash)$ oracle if $p \leq k$ and $q \leq l$; by an $(m - k, n + 1 - l, /, \phi)$ oracle if $p > k$ and $q \geq l$.

Strategy $C(k, l)$ for $i < k \leq m$ and $1 \leq l \leq j$. Say that $A_i < B_j$, and require that subsequent operations merge $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_{l-1}\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$, stipulating that $B_{l-1} < A_k < B_l$. (Analogous to Strategy B, interchanging the roles of A, B.)

Strategy $A'(k, l)$ for $1 \leq k \leq i$ and $j \leq l \leq n$. Say that $A_i > B_j$ and require the merging of $\{A_1, \ldots, A_{k-1}\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_{l+1}, \ldots, B_n\}$. (Analogous to Strategy A.)

Strategy $B'(k, l)$ for $1 \leq k \leq i$ and $j < l \leq n$. Say that $A_i > B_j$, and require the merging of $\{A_1, \ldots, A_{k-1}\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$, subject to $A_{k-1} < B_l < A_k$. (Analogous to Strategy B.)

Strategy $C'(k, l)$ for $1 \leq k < i$ and $j \leq l \leq n$. Say that $A_i > B_j$, and require the merging of $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_{l+1}, \ldots, B_n\}$, subject to $B_l < A_k < B_{l+1}$. (Analogous to Strategy C.)

Because of the constraints, the above strategies cannot be used in certain cases summarized here:

Strategy	Must be omitted when
$A(k, 1)$, $B(k, 1)$, $C(k, 1)$	$\nabla = /$
$A'(1, l)$, $B'(1, l)$, $C'(1, l)$	$\nabla = \backslash$
$A(m, l)$, $B(m, l)$, $C(m, l)$	$\phi = /$
$A'(k, n)$, $B'(k, n)$, $C'(k, n)$	$\phi = \backslash$

Let $\nabla M\phi(m, n)$ denote the maximum lower bound for merging which is obtainable by an oracle of the class described above. Each strategy, when applicable, gives us an inequality relating these nine functions, when the first comparison is $A_i{:}B_j$, namely,

$$
\begin{aligned}
A(k, l)&: \quad \nabla M\phi(m, n) \geq 1 + \nabla M.(k, l-1) + .M\phi(m-k, n+1-l);\\
B(k, l)&: \quad \nabla M\phi(m, n) \geq 1 + \nabla M\backslash(k, l) + /M\phi(m-k, n+1-l);\\
C(k, l)&: \quad \nabla M\phi(m, n) \geq 1 + \nabla M/(k, l-1) + \backslash M\phi(m+1-k, n+1-l);\\
A'(k, l)&: \quad \nabla M\phi(m, n) \geq 1 + \nabla M.(k-1, l) + .M\phi(m+1-k, n-l);\\
B'(k, l)&: \quad \nabla M\phi(m, n) \geq 1 + \nabla M\backslash(k-1, l) + /M\phi(m+1-k, n+1-l);\\
C'(k, l)&: \quad \nabla M\phi(m, n) \geq 1 + \nabla M/(k, l) + \backslash M\phi(m+1-k, n-l).
\end{aligned}
$$

For fixed i and j, the oracle will adopt a strategy which maximizes the lower bound given by the righthand side; then we define $\nabla M\phi(m, n)$ to be the minimum of these lower bounds taken over $1 \leq i \leq m$ and $1 \leq j \leq n$. When m or n is zero, $\nabla M\phi(m, n)$ is zero.

For example, suppose that $m = 2$ and $n = 3$, and our oracle is unconstrained. If the first comparison is $A_1{:}B_1$, the oracle may adopt strategy $A'(1, 1)$, requiring $.M.(0, 1) + .M.(2, 2) = 3$ further comparisons. If the first comparison is $A_1{:}B_3$, the oracle may adopt strategy $B(1, 2)$, requiring $.M\backslash(1, 2) + /M.(1, 2) = 4$ further comparisons. No matter what comparison $A_i{:}B_j$ is made first, the oracle can guarantee that at least three further comparisons must be made. Hence $.M.(2, 3) = 4$.

It isn't easy to do these calculations by hand, but a computer can grind out tables of the $\nabla M\phi$ functions rather quickly. There are some obvious symmetries, e.g.,

$$/M.(m, n) = .M\backslash(m, n) = \backslash M.(n, m) = .M/(n, m), \tag{7}$$

by means of which we can reduce the nine functions to just four,

$$.M.(m, n), \qquad /M.(m, n), \qquad /M\backslash(m, n), \qquad \text{and} \qquad /M/(m, n).$$

Table 1 shows the resulting values for all $m, n \leq 10$; our merging oracle has been defined in such a way that

$$.M.(m, n) \leq M(m, n) \qquad \text{for all} \qquad m, n \geq 0. \tag{8}$$

This relation includes Theorem M as a special case, because our oracle will use the simple strategy of that theorem when $|m - n| \leq 1$.

Let us now consider some simple relations satisfied by the M function:

$$M(m, n) = M(n, m); \tag{9}$$

$$M(m, n) \leq M(m, n+1); \tag{10}$$

$$M(k+m, n) \leq M(k, n) + M(m, n); \tag{11}$$

Table 1

LOWER BOUNDS FOR MERGING, FROM THE "ORACLE"

$.M.(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	1	2	2	3	3	3	3	4	4	4
2	2	3	4	5	5	6	6	6	7	7
3	2	4	5	6	7	7	8	8	9	9
4	3	5	6	7	8	9	10	10	11	11
5	3	5	7	8	9	10	11	12	12	13
6	3	6	7	9	10	11	12	13	14	15
7	3	6	8	10	11	12	13	14	15	16
8	4	6	8	10	12	13	14	15	16	17
9	4	7	9	11	12	14	15	16	17	18
10	4	7	9	11	13	15	16	17	18	19

$/M.(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	1	2	2	3	3	3	3	4	4	4
2	1	3	4	4	5	5	6	6	7	7
3	1	3	5	6	7	7	8	8	9	9
4	1	4	5	7	8	9	9	10	10	11
5	1	4	6	8	9	10	11	12	12	13
6	1	4	6	8	10	11	12	13	14	14
7	1	4	7	9	10	12	13	14	15	16
8	1	5	7	9	11	13	14	15	16	17
9	1	5	8	10	11	13	15	16	17	18
10	1	5	8	10	12	14	15	17	18	19

$/M\backslash(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	$-\infty$	2	2	3	3	3	3	4	4	4
2	$-\infty$	2	4	4	5	5	6	6	7	7
3	$-\infty$	2	4	6	6	7	8	8	8	9
4	$-\infty$	2	5	6	8	8	9	10	10	11
5	$-\infty$	2	5	7	8	10	10	11	12	13
6	$-\infty$	2	5	7	9	10	12	13	14	14
7	$-\infty$	2	5	8	10	11	12	14	15	16
8	$-\infty$	2	6	8	10	12	13	15	16	17
9	$-\infty$	2	6	9	10	12	14	16	17	18
10	$-\infty$	2	6	9	11	13	15	16	18	19

$/M/(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	1	3	3	4	4	4	4	5	5	5
3	1	3	5	5	6	6	7	7	8	8
4	1	4	5	7	7	8	9	9	9	10
5	1	4	6	7	9	9	10	11	11	12
6	1	4	6	8	9	11	11	12	13	14
7	1	4	7	9	10	11	13	14	15	15
8	1	5	7	9	11	12	14	15	16	17
9	1	5	8	9	11	13	15	16	17	18
10	1	5	8	10	12	14	15	17	18	19

$$M(m, n) \leq \max\left(M(m, n-1) + 1, M(m-1, n) + 1\right); \tag{12}$$

$$M(m, n) \leq \max\left(M(m, n-2) + 1, M(m-1, n) + 2\right), \text{ for } m \geq 1, n \geq 2. \tag{13}$$

Relation (12) comes from the usual merging procedure, if we first compare $A_1 : B_1$. Relation (13) is derived similarly, by first comparing $A_1 : B_2$; if $A_1 > B_2$, we need $M(m, n-2)$ more comparisons, but if $A_1 < B_2$, we can insert A_1 into its proper place and merge $\{A_2, \ldots, A_m\}$ with $\{B_1, \ldots, B_n\}$. Generalizing, we can see that

$$M(m, n) \leq \max\left(M(m, n-k) + 1, M(m-1, n) + 1 + \lceil \log_2 k \rceil\right),$$
$$\text{for} \qquad m \geq 1, \qquad n \geq k, \tag{14}$$

by first comparing $A_1 : B_k$ and using binary search if $A_1 < B_k$.

It turns out that $M(m, n) = .M.(m, n)$ for all $m, n \leq 10$, so Table 1 actually gives the optimum values for merging. This can be proved by using (9)–(14) together with special constructions for $(m, n) = (2, 8)$, $(3, 6)$, and $(5, 9)$ which are given in exercises 8, 9, and 10.

On the other hand, our oracle doesn't always give the best possible lower bounds; the simplest example is $m = 3$, $n = 11$, when $.M.(3, 11) = 9$ but $M(3, 11) = 10$. To see where the oracle has "failed" in this case, we must study the reasons for its decisions; further scrutiny reveals that if $(i, j) \neq (2, 6)$, the oracle can find a strategy which demands 10 comparisons; but when $(i, j) = (2, 6)$, no strategy beats Strategy $A(2, 4)$, leading to the lower bound $1 + .M.(2, 3) + .M.(1, 8) = 9$. It is necessary but not sufficient to finish by merging $\{A_1, A_2\}$ with $\{B_1, B_2, B_3\}$ and $\{A_3\}$ with $\{B_4, \ldots, B_{11}\}$, so the lower bound fails to be sharp in this case.

Similarly it can be shown that $.M.(2, 38) = 10$ while $M(2, 38) = 11$, so our oracle isn't good enough to solve the case $m = 2$. But there is an infinite class of values for which it excels:

Theorem K.

$$M(m, m+2) = 2m+1 \quad \textit{for} \quad m \geq 2;$$
$$M(m, m+3) = 2m+2 \quad \textit{for} \quad m \geq 4;$$
$$M(m, m+4) = 2m+3 \quad \textit{for} \quad m \geq 6.$$

Proof. We can in fact prove the result with M replaced by $.M.$; for small m the results have been obtained by computer, so we may assume that m is sufficiently large. We may also assume that the first comparison is $A_i{:}B_j$ where $i \leq \lceil m/2 \rceil$. If $j \leq i$ we use strategy $A'(i, i)$, obtaining

$$.M.(m, m+d) \geq 1 + .M.(i-1, i) + .M.(m+1-i, m+d-i) = 2m+d-1$$

by induction on d, $d \leq 4$. If $j > i$ we use strategy $A(i, i+1)$, obtaining

$$.M.(m, m+d) \geq 1 + .M.(i, i) + .M.(m-i, m+d-i) = 2m+d-1$$

by induction on m. ∎

The first two parts of Theorem K were obtained by F. Hwang and S. Lin in 1969. This proof makes it reasonable to conjecture that $M(m, m+d) = 2m+d-1$ for all sufficiently large m, when d is fixed. (Cf. exercise 6.)

Upper bounds. Now let us consider *upper* bounds for $M(m, n)$; good upper bounds correspond to efficient merging algorithms.

When $m = 1$ the merging problem is equivalent to an insertion problem, and there are $n+1$ places in which A_1 might fall among $B_1, \ldots, B_n$. For this case it is easy to see that *any* extended binary tree with $n+1$ external nodes is the tree for some merging method! (See exercise 2.) Hence we may choose an optimum binary tree, realizing the information-theoretic lower bound

$$1 + \lfloor \log_2 n \rfloor = M(1, n) = \lceil \log_2 (n+1) \rceil. \tag{15}$$

Binary search (Section 6.2.1) is, of course, a simple way to attain this value.

The case $m = 2$ is extremely interesting, but considerably harder. It has

been solved completely by R. L. Graham, F. K. Hwang, and S. Lin (see exercises 11, 12, 13); we have

$$M(2, n) = \lceil \log_2 \tfrac{7}{12}(n+1) \rceil + \lceil \log_2 \tfrac{14}{17}(n+1) \rceil. \tag{16}$$

We have seen that the usual merging procedure is optimum when $m = n$, and the rather different binary search procedure is optimum when $m = 1$. What we need is an in-between method which combines the normal merging algorithm with binary search in such a way that the best features of both are retained. Formula (14) suggests the following merging algorithm, due to F. K. Hwang and S. Lin [*SIAM J. Computing* 1 (1972), 31–39]:

Algorithm H (*Binary merging*).

H1. If m or n is zero, stop. If $m \le n$, set $t \leftarrow \lfloor \log_2 (n/m) \rfloor$. If $m > n$, set $t \leftarrow \lfloor \log_2 (m/n) \rfloor$ and go to H4.

H2. Compare $A_m : B_{n+1-2^t}$. If A_m is smaller, set $n \leftarrow n - 2^t$ and return to step H1.

H3. Using binary search (which requires exactly t more comparisons), insert A_m into its proper place among $\{B_{n+1-2^t}, \ldots, B_n\}$. If k is maximal such that $B_k < A_m$, set $m \leftarrow m - 1$ and $n \leftarrow k$. Return to H1.

H4. (Steps H4 and H5 are like H2 and H3, interchanging the roles of m, n, A, B.) If $B_n < A_{m+1-2^t}$, set $m \leftarrow m - 2^t$ and return to step H1.

H5. Insert B_n into its proper place among the A's. If k is maximal such that $A_k < B_n$, set $m \leftarrow k$ and $n \leftarrow n - 1$. Return to H1. ▮

As an example of this algorithm, Table 2 shows the process of merging the three keys {087, 503, 512} with thirteen keys {061, 154, . . . , 908}; eight comparisons are required in this example.

Table 2

EXAMPLE OF BINARY MERGING

A (boldface elements are compared)			*B*															Output	
{087	503	512}	{051	154	170	275	426	509	612	653	677	703	765	897	908}				
{087	503	512}	{061	154	170	275	426	509	612	653	677}					{703	765	897	908}
{087	503	512}	{061	154	170	275	426	509	612}					{653	677	703	765	897	908}
{087	503	512}	{061	154	170	275	426	509	612}					{653	677	703	765	897	908}
{087	503}		{061	154	170	275	426	509}				{512	612	653	677	703	765	897	908}
{087	503}		{061	154	170	275	426	509}				{512	612	653	677	793	765	897	908}
{087}			{061	154	170	275	426}			{503	509	512	612	653	677	703	765	897	908}
{087}			{061}		{154	170	275	426	503	509	512	612	653	677	703	765	897	908}	
			{061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908}	

Let $H(m, n)$ be the maximum number of comparisons required by Hwang and Lin's algorithm. To calculate $H(m, n)$, we may assume that $k = n$ in step H3 and $k = m$ in step H5, since it is not difficult to prove that $H(m, n) \le H(m, n+1)$ for all $n \ge m$ by induction on n. Thus when $m \le n$ we have

$$H(m, n) = \max\left(H(m, n - 2^t) + 1, \quad H(m - 1, n) + t + 1\right), \quad \text{for} \quad 2^t m \le n < 2^{t+1} m. \tag{17}$$

Replace n by $2n + \epsilon$, $\epsilon = 0$ or 1, to get

$$H(m, 2n + \epsilon) = \max\left(H(m, 2n + \epsilon - 2^{t+1}) + 1, \quad H(m - 1, 2n + \epsilon) + t + 2\right), \quad \text{for} \quad 2^t m \le n < 2^{t+1} m;$$

whence it follows by induction on n that

$$H(m, 2n + \epsilon) = H(m, n) + m, \quad \text{for} \quad m \le n, \quad \epsilon = 0 \text{ or } 1. \tag{18}$$

It is also easy to see that $H(m, n) = m + n - 1$ when $m \le n < 2m$; hence a repeated application of (18) yields the general formula

$$H(m, n) = m + \lfloor n/2^t \rfloor - 1 + tm, \quad \text{for} \quad m \le n, \quad t = \lfloor \log_2 (n/m) \rfloor. \tag{19}$$

Setting $m = \alpha n$ and $\theta = \log_2 (n/m) - t$ gives

$$H(\alpha n, n) = \alpha n(1 + 2^\theta - \theta - \log_2 \alpha) + O(1), \tag{20}$$

as $n \to \infty$. We know by Eq. 5.3.1–35 that $1.9139 < 1 + 2^\theta - \theta \le 2$; hence (20) may be compared with the information-theoretic lower bound (3). Hwang and Lin have proved (see exercise 17) that

$$H(m, n) < \left\lceil \log_2 \binom{m+n}{m} \right\rceil + \min(m, n). \tag{21}$$

The Hwang-Lin merging algorithm, which might be called "binary merging," does not always give optimum results, but it has the great virtue that it can be programmed rather easily. It reduces to "uncentered binary search" when $m = 1$, and it reduces to the usual merging procedure when $m \approx n$, so it represents an excellent compromise between those two methods. Furthermore, it *is* optimum in many cases (see exercise 16).

Formula (18) suggests that the M function itself might satisfy

$$M(m, n) \le M(m, \lfloor n/2 \rfloor) + m. \tag{22}$$

This is actually true (see exercise 19). Tables of $M(m, n)$ suggest several other plausible relations, such as

$$M(m + 1, n) \ge 1 + M(m, n) \ge M(m, n + 1), \quad \text{for} \quad m \le n; \tag{23}$$

$$M(m + 1, n + 1) \ge 2 + M(m, n); \tag{24}$$

but no proof of these inequalities is known.

EXERCISES

1. [*15*] Find an interesting relation between $M(m, n)$ and the function S defined in Section 5.3.1. [*Hint:* Consider $S(m+n)$.]

▸ **2.** [*22*] When $m = 1$, every merging algorithm without redundant comparisons defines an extended binary tree with $\binom{m+n}{m} = n+1$ external nodes. Prove that, conversely, every extended binary tree with $n+1$ external nodes corresponds to some merging algorithm with $m = 1$.

3. [*M24*] Prove that $.M.(1, n) = M(1, n)$ for all n.

4. [*M44*] Is $.M.(m, n) \geq \lceil \log_2 \binom{m+n}{m} \rceil$ for all m and n?

5. [*M30*] Prove that $.M.(m, n) \leq .M\backslash(m, n+1)$.

6. [*M26*] The stated proof of Theorem K requires that a lot of cases be verified by computer. How can the number of such cases be drastically reduced?

7. [*21*] Prove (11).

▸ **8.** [*24*] Prove that $M(2, 8) \leq 6$, by finding an algorithm which merges two elements with eight others using at most six comparisons.

9. [*27*] Prove that three elements can be merged with six in at most seven steps.

10. [*33*] Prove that five elements can be merged with nine in at most twelve steps. [*Hint:* Experience with the oracle suggests starting by comparing $A_1{:}B_2$, then if $A_1 < B_2$ trying $A_5{:}B_8$.]

11. [*M40*] (F. Hwang, S. Lin.) Let $g_{2k} = \lfloor 2^k \cdot \frac{17}{14} \rfloor$, $g_{2k+1} = \lfloor 2^k \cdot \frac{12}{7} \rfloor$, for $k \geq 0$, so that we have $(g_0, g_1, g_2, \ldots) = (1, 1, 2, 3, 4, 6, 9, 13, 19, 27, 38, 54, 77, \ldots)$. Prove that it takes more than t comparisons to merge two elements with g_t elements, in the worst case; but two elements can be merged with $g_t - 1$ in at most t steps. [*Hint:* Show that if $n \geq g_{t-1}$ and if we want to merge $\{A_1, A_2\}$ with $\{B_1, B_2, \ldots, B_n\}$ in t comparisons, we can't do better than to compare $A_2{:}B_{g_{t-1}}$ on the first step.]

12. [*M21*] Let $R_n(i, j)$ be the least number of comparisons required to sort the distinct objects $\{\alpha, \beta, X_1, X_2, \ldots, X_n\}$, given the relations

$$\alpha < \beta, \qquad X_1 < X_2 < \cdots < X_n, \qquad \alpha < X_{i+1}, \qquad \beta > X_{n-j}.$$

(The condition $\alpha < X_{i+1}$ or $\beta > X_{n-j}$ becomes vacuous when $i \geq n$ or $j \geq n$. Therefore $R_n(n, n) = M(2, n)$.)

Clearly, $R_n(0, 0) = 0$. Prove that

$$R_n(i, j) = 1 + \min\Big(\min_{1 \leq k \leq i} \max\big(R_n(k-1, j), R_{n-k}(i-k, j)\big),$$
$$\min_{1 \leq k \leq j} \max\big(R_n(i, k-1), R_{n-k}(i, j-k)\big)\Big)$$

for $0 \leq i \leq n$, $0 \leq j \leq n$, $i + j > 0$.

13. [*M42*] (R. L. Graham.) Show that the solution to the recurrence in exercise 12 may be expressed as follows. Define the function $G(x)$, for $0 < x < \infty$, by the rules

$$G(x) = \begin{cases} 1, & 0 < x \leq \frac{5}{7}; \\ \frac{1}{2} + \frac{1}{8}G(8x-5), & \frac{5}{7} < x \leq \frac{3}{4}; \\ \frac{1}{2}G(2x-1), & \frac{3}{4} < x \leq 1; \\ 0, & 1 < x < \infty. \end{cases}$$

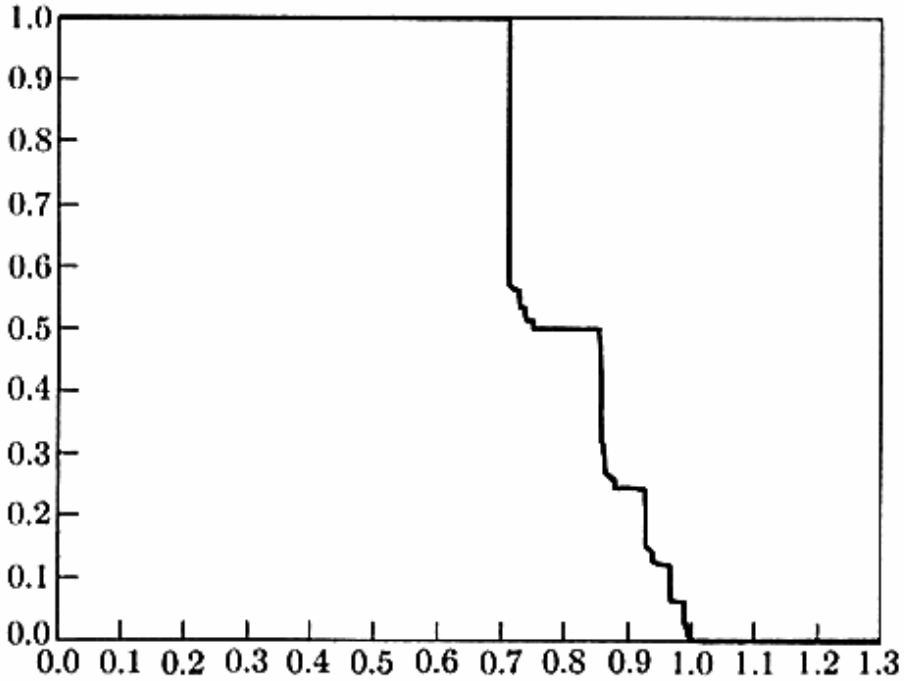


Fig. 38. Graham's function (see exercise 13).

(See Fig. 38.) Since $R_n(i, j) = R_n(j, i)$ and since $R_n(0, j) = M(1, j)$, we may assume that $1 \le i \le j \le n$. Let $p = \lfloor \log_2 i \rfloor$, $q = \lfloor \log_2 j \rfloor$, $r = \lfloor \log_2 n \rfloor$, and let $t = n - 2^r + 1$. Then

$$R_n(i, j) = p + q + S_n(i, j) + T_n(i, j),$$

where S_n and T_n are functions which are either 0 or 1:

$$\begin{aligned} S_n(i, j) &= 1 \quad \text{if and only if } q < r \text{ or } (i - 2^p \ge u \text{ and } j - 2^r \ge u), \\ T_n(i, j) &= 1 \quad \text{if and only if } p < r \text{ or } (t > \tfrac{6}{7}2^{r-2} \text{ and } i - 2^r \ge v), \end{aligned}$$

where $u = 2^p G(t/2^p)$ and $v = 2^{r-2} G(t/2^{r-2})$.

(This may be the most formidable recurrence relation that will ever be solved!)

14. [*46*] (Hwang and Lin.) Let $h_{3k} = 2^k + 2^{k-1} - 1$, $h_{3k+1} = g_{2k} + g_{2k-3} + 2^{k-2}$, $h_{3k+2} = 2g_{2k}$, for $k \ge 2$, except that $h_8 = 9$, with initial values adjusted so that we have $(h_0, h_1, h_2, \ldots) = (1, 1, 2, 2, 3, 4, 5, 7, 9, 11, 14, 18, 23, 29, 38, 47, 59, 76, \ldots)$. Here g_k is the function defined in exercise 11. Prove (or disprove) that $M(3, h_t) > t$, $M(3, h_t - 1) \le t$, for all t.

15. [*12*] Step H1 of the binary merge algorithm may require the calculation of $\lfloor \log_2 (n/m) \rfloor$. Explain how to compute this easily without division or calculation of a logarithm.

16. [*18*] For which m and n is Hwang and Lin's binary merging algorithm optimum, for $1 \le m \le n \le 10$?

17. [*M25*] Prove (21). [*Hint:* The inequality isn't very tight.]

18. [*M40*] Study the *average* number of comparisons used by binary merge.

▶**19.** [*23*] Prove that the M function satisfies (22).

20. [*20*] Show that if $M(m, n+1) \le M(m+1, n)$ for all $m \le n$, then $M(m, n+1) \le 1 + M(m, n)$ for all $m \le n$.

21. [*M47*] Prove or disprove (23), (24).

22. [*M50*] Study the minimum *average* number of comparisons needed to merge m things with n.

23. [*HM30*] (E. Reingold.) Let $\{A_1, \ldots, A_n\}$ and $\{B_1, \ldots, B_n\}$ be sets containing n elements each. Consider an algorithm which attempts to test *equality* of these two sets solely by making comparisons for equality between elements. Thus, the algorithm asks questions of the form "Is $A_i = B_j$?" for certain i and j, and it branches depending on whether the answer is yes or no.

By defining a suitable oracle, prove that any such algorithm must make at least $\frac{1}{2}n(n+1)$ comparisons in its worst case.

*5.3.3. Minimum-Comparison Selection

A similar class of interesting problems arises when we look for best possible procedures to select the tth largest of n elements.

The history of this question goes back to Rev. C. L. Dodgson's amusing (though serious) essay on lawn tennis tournaments which appeared in *St. James's Gazette*, August 1, 1883, pp. 5–6. Dodgson, who is of course better known as Lewis Carroll, was concerned about the unjust manner in which prizes were (and still are) awarded in tennis tournaments. Consider, for example, Fig. 39 which shows a typical "knockout tournament" between 32 players labeled *01*, *02*, . . . , *32*. In the "finals," player *01* defeats player *05*, so it is clear that player *01* is the champion and he deserves first prize. The inequity arises in connection with the fact that player *05* usually gets second prize, although he may not be the second-best player. It is possible to win second prize even if you are worse than half of the players in the tournament. In fact, as Dodgson observed, the second-best player wins second prize if and only if he and the champion are originally in opposite halves of the tournament; for 2^n players this occurs with probability $2^{n-1}/(2^n - 1)$, so the wrong player receives second prize almost half of the time! If the losers of the semifinal round (players *25* and *17* in Fig. 39) compete for third prize, it is highly unlikely that the third-best player receives third prize.

Dodgson therefore set out to design a tournament that determines the true second- and third-best players, assuming a transitive ranking. (In other words if player A beats player B, and B beats C, then we may assume that A would beat C.) He devised a procedure in which losers are allowed to play further games until they are known to be definitely inferior to three other players. An example of Dodgson's scheme appears in Fig. 40, which is a supplementary tournament to be run in conjunction with Fig. 39. An attempt is made to pair off players with equal records so far, and to avoid matches in which both players have been defeated by the same man. For example, *16* loses to *11* and *13* loses to *12* in Round 1; after *16* loses to *13* in the second round, *16* is eliminated since he is now known to be inferior to *11*, *12*, and *13*. In Round 3 we do not allow *19* to play with *21*, since they have both been defeated by *18* and we could not automatically eliminate the loser of *19* vs. *21*.

It would be nice to report that Lewis Carroll's tournament turns out to be

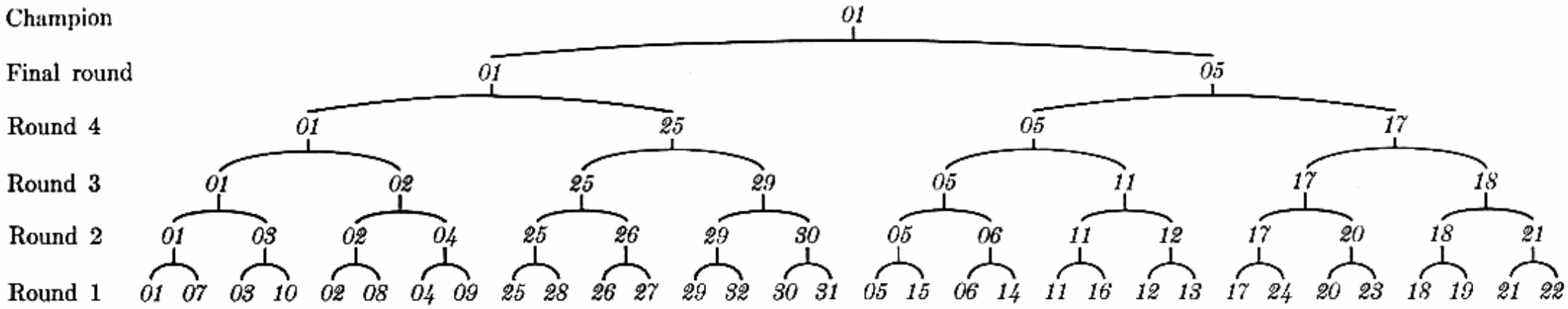


Fig. 39. A knockout tournament with 32 players.

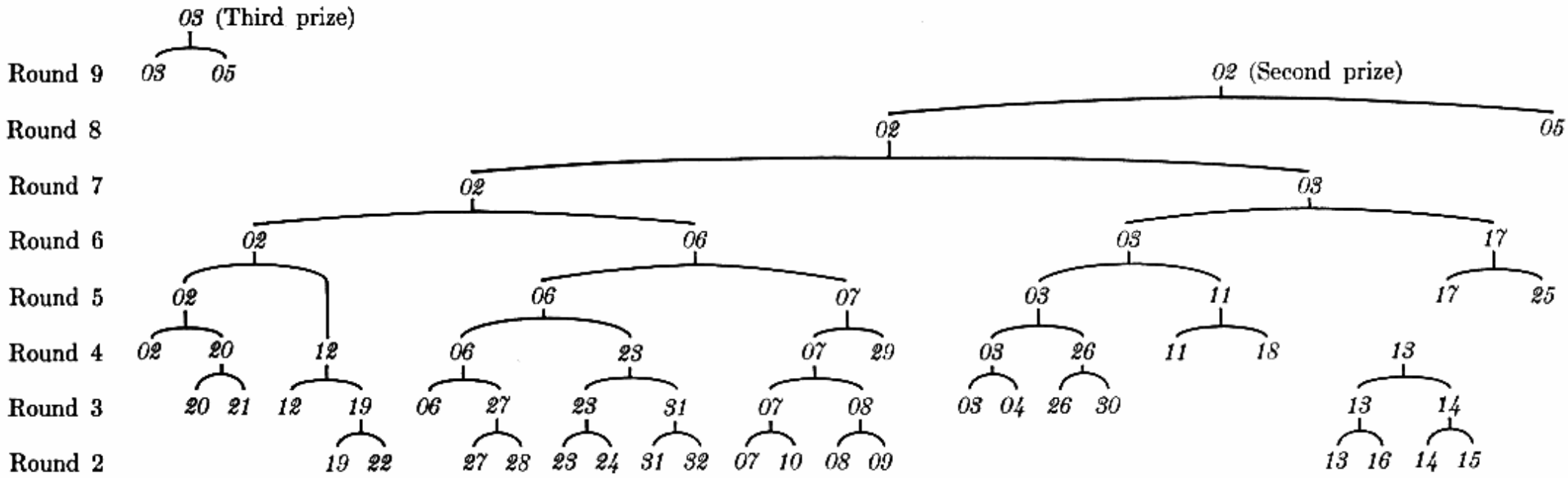


Fig. 40. Lewis Carroll's lawn tennis tournament (played in conjunction with Fig. 39).

optimal, but unfortunately that is not the case. His diary entry for July 23, 1883, says that he composed the essay in about six hours, and he felt "we are now so late in the (tennis) season that it is better it should appear soon than be written well." His procedure makes more comparisons than necessary, and it is not formulated precisely enough to qualify as an algorithm. On the other hand, it has some rather interesting aspects from the standpoint of parallel computation. And it appears to be an excellent plan for a tennis tournament, because he built in some dramatic effects; for example, he specified that the two finalists should sit out round 5, playing an extended match during rounds 6 and 7. But tournament directors presumably thought the proposal was too logical, and so Carroll's system has apparently never been tried. Instead a method of "seeding" is used to keep the best players in different parts of the tree.

In a mathematical seminar during 1929–1930, Hugo Steinhaus posed the problem of finding the *minimum* number of tennis matches required to determine the first and second best players in a tournament, when there are $n \geq 2$ players in all. J. Schreier [*Mathesis Polska* 7 (1932), 154–160] gave a procedure which requires at most $n - 2 + \lceil \log_2 n \rceil$ matches, using essentially the same method as the first two stages in what we have called tree selection sorting (cf. Section 5.2.3, Fig. 23), avoiding redundant comparisons that involve $-\infty$. Schreier also claimed that $n - 2 + \lceil \log_2 n \rceil$ is best possible, but his proof was incorrect, as was another attempted proof by J. Słupecki [*Colloquium Mathematicum* 2 (1951), 286–290]. Thirty-two years went by before a correct, although rather complicated, proof was finally published by S. S. Kislitsyn [*Sibirskiĭ Mat. Zhurnal* 5 (1964), 557–564].

Let $V_t(n)$ denote the minimum number of comparisons needed to determine the tth largest of n elements, for $1 \leq t \leq n$, and let $W_t(n)$ be the minimum number required to determine the largest, second largest, . . . , and tth largest, collectively. By symmetry, we have

$$V_t(n) = V_{n+1-t}(n), \tag{1}$$

and it is obvious that

$$V_1(n) = W_1(n), \tag{2}$$

$$V_t(n) \leq W_t(n), \tag{3}$$

$$W_n(n) = W_{n-1}(n) = S(n). \tag{4}$$

We have observed in Section 5.2.3 that

$$V_1(n) = n - 1. \tag{5}$$

In fact, there is an astonishingly simple proof of this fact, since everyone in a tournament except the champion must lose at least one game! By extending this idea and using an "oracle," we can prove the Schreier-Kislitsyn theorem without much difficulty:

Theorem S. $V_2(n) = W_2(n) = n - 2 + \lceil \log_2 n \rceil$, for $n \geq 2$.

Proof. Assume that n players have participated in a tournament which has determined the second-best player by some given procedure, and let a_j be the number of players who have lost j or more matches. The total number of matches played is then $a_1 + a_2 + a_3 + \cdots$. We cannot determine the second-best player without also determining the champion (cf. exercise 2), so our previous argument shows that $a_1 = n - 1$. To complete the proof, we will show that there is always some sequence of outcomes of the matches which makes $a_2 \geq \lceil \log_2 n \rceil - 1$.

Suppose that at the end of the tournament the champion has played (and beaten) p players; one of these is the second best, and the others must have lost at least one other time, so $a_2 \geq p - 1$. Therefore we can complete the proof by finding an oracle who decides the results of the games in such a way that the champion must play at least $\lceil \log_2 n \rceil$ other people.

Let the oracle declare A to be better than B if A is previously undefeated and B has lost at least once, or if both are undefeated and B has won fewer matches than A at that time. In other circumstances the oracle may make an arbitrary decision consistent with some partial ordering.

Consider the outcome of a complete tournament whose matches have been decided by such an oracle. Let us say that "A supersedes B" if and only if $A = B$ or A supersedes the player who first defeated B. (Only a player's first defeat is relevant in this relation, his subsequent games are ignored. According to the mechanism of the oracle, any player who *first* defeats another must be previously unbeaten.) It follows that a player who won his first p matches supersedes at most 2^p players on the basis of those p contests. (The latter is clear for $p = 0$, and for $p > 0$ the pth match was against someone who was either previously beaten or who supersedes at most 2^{p-1} players.) The champion supersedes everyone, so he must have played at least $\lceil \log_2 n \rceil$ matches. ∎

Thus the problem of finding the second largest is completely resolved in the minimax sense. Exercise 6 shows, in fact, that it is possible to give a simple formula for the minimum number of comparisons needed to find the second largest element of a set when an arbitrary partial ordering of the elements is known.

What if t > 2? In the paper cited above, Kislitsyn went on to consider larger values of t, proving that

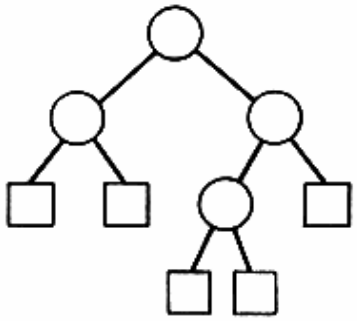
$$W_t(n) \leq n - t + \sum_{n+1-t<j\leq n} \lceil \log_2 j \rceil, \qquad \text{for} \qquad n \geq t. \tag{6}$$

For $t = 1$ and $t = 2$ we have seen that equality actually holds in this formula; for $t = 3$ it can be slightly improved (see exercise 21).

We shall prove Kislitsyn's theorem by showing that the first t stages of *tree selection* require at most $n - t + \sum_{n+1-t<j\leq n} \lceil \log_2 j \rceil$ comparisons (ignoring all of the comparisons which involve $-\infty$). It is interesting to note that the

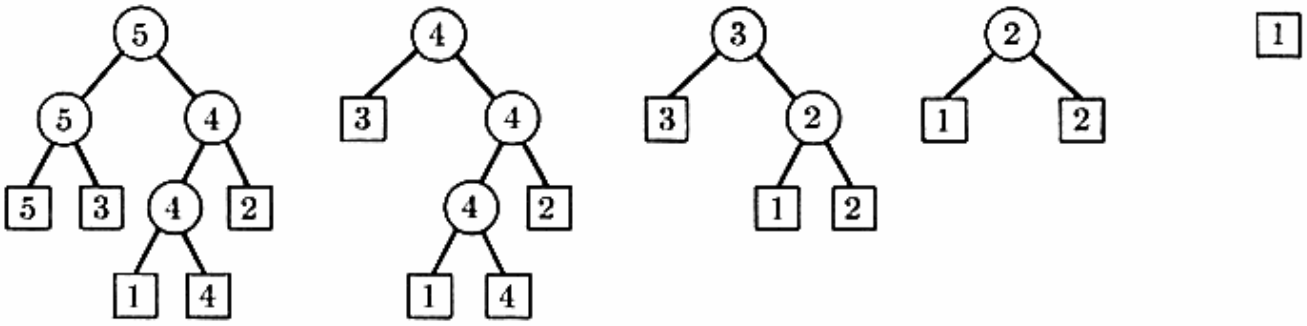
right-hand side of (6) equals $B(n)$ when $t = n - 1$ and $t = n$ (see Eq. 5.3.1–3); hence tree selection and binary insertion yield the same upper bound for the sorting problem, although they are quite different methods.

Let α be an extended binary tree with n external nodes, and let π be a permutation of $\{1, 2, \ldots, n\}$. Place the elements of π into the external nodes, from left to right in symmetric order, and fill in the internal nodes according to the rules of a knockout tournament as in tree selection. When the resulting tree is subjected to repeated selection operations, it defines a sequence $c_{n-1}\, c_{n-2} \ldots c_1$, where c_j is the number of comparisons required to bring element j to the root of the tree when element $j + 1$ has been replaced by $-\infty$. For example, if α is the tree



(7)

and if $\pi = 5\ 3\ 1\ 4\ 2$, we obtain the successive trees



$c_4 = 1$ $\quad c_3 = 2$ $\quad c_2 = 0$ $\quad c_1 = 0$

If π had been $3\ 1\ 5\ 4\ 2$, the sequence $c_4\, c_3\, c_2\, c_1$ would have been $2\ 1\ 1\ 0$ instead. It is easy to see that c_1 is always zero.

Let $\mu(\alpha, \pi)$ be the multiset $\{c_{n-1}, c_{n-2}, \ldots, c_1\}$ determined by α and π. If

$\alpha =$

and if elements 1 and 2 do not both appear in α' or both in α'', it is easy to see that

$$\mu(\alpha, \pi) = (\mu(\alpha', \pi') + 1) \uplus (\mu(\alpha'', \pi'') + 1) \uplus \{0\} \tag{8}$$

for appropriate permutations π' and π'', where $(\mu + 1)$ denotes the multiset obtained by adding 1 to each element of μ. On the other hand, if elements 1 and 2 both appear in α', we have

$$\mu(\alpha, \pi) = (\mu(\alpha', \pi') + \epsilon) \uplus (\mu(\alpha'', \pi'') + 1) \uplus \{0\},$$

where $(\mu + \epsilon)$ denotes some multiset obtained by adding 1 to some elements of μ and 0 to the others. A similar formula holds when 1 and 2 both appear in α''. Let us say that multiset μ_1 *dominates* μ_2 if both μ_1 and μ_2 contain the same number of elements, and the kth largest element of μ_1 is greater than or equal to the kth largest element of μ_2 for all k; and let us define $\mu(\alpha)$ to be the "dominant" $\mu(\alpha, \pi)$, taken over all permutations π, in the sense that $\mu(\alpha)$ dominates $\mu(\alpha, \pi)$ for all π and $\mu(\alpha) = \mu(\alpha, \pi)$ for some π. The above formulas show that

$$\mu(\square) = \emptyset, \qquad \mu(\underset{\alpha' \quad \alpha''}{\bigcirc}) = (\mu(\alpha') + 1) \uplus (\mu(\alpha'') + 1) \uplus \{0\}; \tag{9}$$

hence $\mu(\alpha)$ *is the multiset of all distances from the root to the internal nodes of* α.

The reader who has followed the above train of thought will now see that we are ready to prove Kislitsyn's theorem (6); for $W_t(n)$ is less than or equal to $n - 1$ plus the $t - 1$ largest elements of $\mu(\alpha)$, where α is any tree being used in tree selection sorting. We may take α to be the complete binary tree with n external nodes (cf. Section 2.3.4.5), when

$$\begin{aligned} \mu(\alpha) &= \{\lfloor \log_2 1 \rfloor, \lfloor \log_2 2 \rfloor, \ldots, \lfloor \log_2 (n-1) \rfloor\} \\ &= \{\lceil \log_2 2 \rceil - 1, \lceil \log_2 3 \rceil - 1, \ldots, \lceil \log_2 n \rceil - 1\}. \end{aligned} \tag{10}$$

We obtain formula (6) when we consider the $t - 1$ largest elements of this multiset.

Kislitsyn's theorem gives a good upper bound for $W_t(n)$; he remarked that $V_3(5) = 6 < W_3(5) = 7$, but he was unable to find a better bound for $V_t(n)$, in general. A. Hadian and M. Sobel discovered a way to do this using *replacement selection* instead of tree selection (cf. Section 5.4.1). Their formula [Univ. of Minnesota, Dept. of Statistics, report 121 (May, 1969)],

$$V_t(n) \le n - t + (t-1)\lceil \log_2 (n + 2 - t) \rceil, \qquad n \ge t, \tag{11}$$

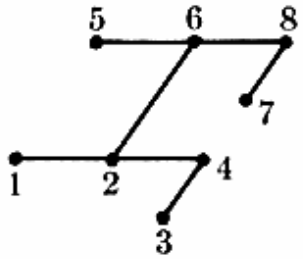
differs from (6) in that each element in the sum of (6) is replaced by the smallest element.

Hadian and Sobel's theorem (11) can be proved by using the following construction: First set up a binary tree for a knockout tournament on $n - t + 2$ items. (This takes $n - t + 1$ comparisons.) The largest item is greater than $n - t + 1$ others, so it can't be tth largest. Replace it, where it appears at an external node of the tree, by one of the $t - 2$ elements held in reserve, and find the largest element of the resulting $n - t + 2$; this requires at most $\lceil \log_2 (n + 2 - t) \rceil$ comparisons, because we need to recompute only one path in the tree. Repeat this operation $t - 2$ times in all, for each element held in reserve. Finally, replace the currently largest element by $-\infty$, and determine the largest of the remaining $n + 1 - t$; this requires at most $\lceil \log_2 (n + 2 - t) \rceil - 1$ comparisons, and it brings the tth largest element of the original set to the root of the tree. Summing the comparisons yields (11).

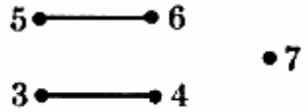
In relation (11) we should of course replace t by $n + 1 - t$ on the right-hand side whenever $n + 1 - t$ gives a better value (as when $n = 6$, $t = 3$).

Curiously, the formula gives a smaller bound for $V_7(13)$ than it does for $V_6(13)$. The upper bound in (11) is exact for $n \leq 6$, but as n and t get larger it is possible to obtain much better estimates of $V_t(n)$.

For example, the following elegant method (due to David G. Doren) can be used to show that $V_4(8) \leq 12$. Let the elements be $X_1, \ldots, X_8$; first compare $X_1:X_2$ and $X_3:X_4$ and the two winners, and do the same to $X_5:X_6$ and $X_7:X_8$ and their winners. Relabel elements so that $X_1 < X_2 < X_4 > X_3$, $X_5 < X_6 < X_8 > X_7$, then compare $X_2:X_6$; by symmetry assume that $X_2 < X_6$, so that we have the configuration



(Now X_1 and X_8 are out of contention and we must find the third largest of $\{X_2, \ldots, X_7\}$.) Compare $X_2:X_7$, and discard the smaller; in the worst case we have $X_2 < X_7$ and we must find the third largest of



This can be done in $V_3(5) - 2 = 4$ more steps, since the $V_3(5) = 6$ procedure of (11) begins by comparing two disjoint pairs of elements.

Other tricks of this kind can be used to produce the results shown in Table 1; no general method is evident as yet, and the values listed in the table for $n \geq 8$ may be subject to further improvement. Note that $V_3(10) \leq 14$, so (11) does not always give equality when $t = 3$. The fact that $V_4(7) = 10$ shows that (11) is already off by 2 when $n = 7$.

Table 1

BEST UPPER BOUNDS KNOWN FOR $V_t(n)$

n	$V_1(n)$	$V_2(n)$	$V_3(n)$	$V_4(n)$	$V_5(n)$	$V_6(n)$	$V_7(n)$	$V_8(n)$	$V_9(n)$	$V_{10}(n)$
1	0									
2	1	1								
3	2	3	2							
4	3	4	4	3						
5	4	6	6	6	4					
6	5	7	8	8	7	5				
7	6	8	10	10*	10	8	6			
8	7	9	11	12	12	11	9	7		
9	8	11	12	14	15*	14	12	11	8	
10	9	12	14*	15	17	17	15	14*	12	9

*Exercises 10–12 give constructions which improve on Eq. (11) in these cases.

A linear method. When n is odd and $t = \lceil n/2 \rceil$, the tth largest (and tth smallest) element is called the median. According to (11), we can find the median of n elements in $\approx \frac{1}{2} n \log_2 n$ comparisons; but this is only about twice as fast as sorting, even though we are asking for much less information. For several years, concerted efforts were made by a number of people to find an improvement over (11) when t and n are large; finally in 1971, Manuel Blum discovered a method which needed only $O(n \log \log n)$ steps. Blum's approach to the problem suggested a new class of techniques which led to the following construction due to R. Rivest and R. Tarjan:

Theorem L. $V_t(n) \leq 15n - 163$ *for* $1 \leq t \leq n$, *when* $n > 32$.

Proof. The theorem is trivial when n is small, since $V_t(n) \leq S(n) \leq 10n \leq 15n - 163$ for $32 < n \leq 2^{10}$. By adding at most 13 dummy "$-\infty$" elements, we may assume that $n = 7(2q + 1)$ for some integer $q \geq 73$. The following method may now be used to select the tth largest:

Step 1. Divide the elements into $2q + 1$ groups of seven elements each, and sort each of the groups. This takes at most $13(2q + 1)$ comparisons.

Step 2. Find the median of the $2q + 1$ median elements obtained in Step 1, and call it x. By induction on q, this takes at most $V_{q+1}(2q + 1) \leq 30q - 148$ comparisons.

Step 3. The $n - 1$ elements other than x have now been partitioned into three sets (see Fig. 41):

$4q + 3$ elements known to be greater than x (Region B);

$4q + 3$ elements known to be less than x (Region C);

$6q$ elements whose relation to x is unknown (Regions A, D).

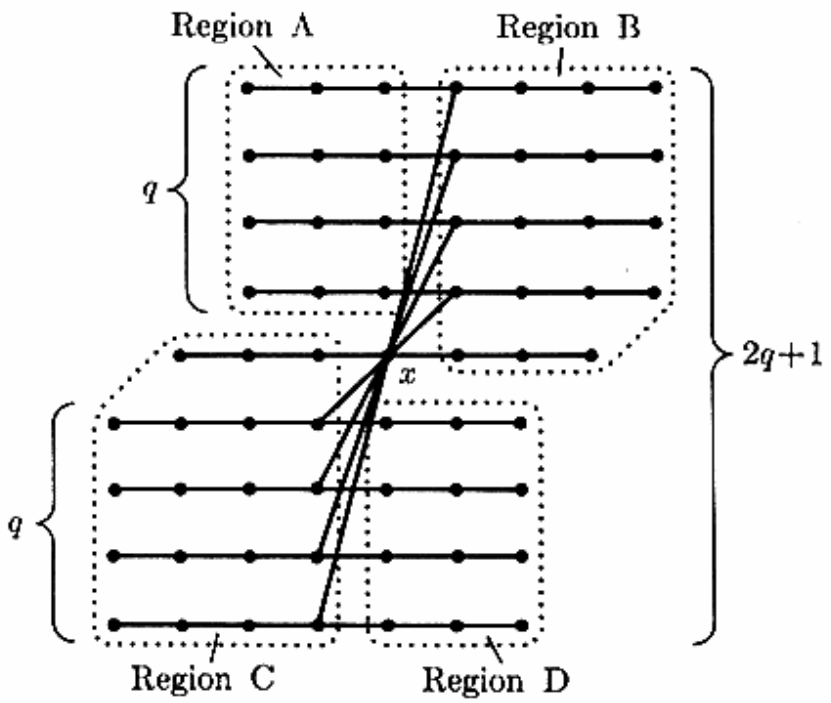


Fig. 41. The selection algorithm of Rivest and Tarjan ($q = 4$).

By making $4q$ additional comparisons, we can tell exactly which of the elements in regions A and D are less than x. (We first test x against the middle element of each triple.)

Step 4. We have now found r elements greater than x and $n - 1 - r$ elements less than x, for some r. If $t = r + 1$, x is the answer; if $t < r + 1$, we need to find the tth largest of the r large elements; and if $t > r + 1$, we need to find the $(t - 1 - r)$th largest of the $n - 1 - r$ small elements. The point is that r and $n - 1 - r$ are both less than or equal to $10q + 3$ (the size of regions A and D, plus either B or C). By induction on q this step therefore requires at most $15(10q + 3) - 163$ comparisons.

The total number of comparisons comes to at most

$$13(2q + 1) + 30q - 148 + 4q + 15(10q + 3) - 163 = 15(14q - 6) - 163.$$

Since we started with at least $14q - 6$ elements, the proof is complete. ∎

The method used in this proof is a little crude because it throws away good information in Step 4. Careful refinements by V. Pratt, R. Rivest, and R. Tarjan have shown that the constant 15 can be reduced to 5.43.

The average number. Instead of minimizing the *maximum* number of comparisons, we can ask instead for an algorithm which minimizes the *average* number of comparisons, assuming random order. As usual, this problem is considerably harder, and it is still unsolved even in the case $t = 2$. Claude Picard mentioned the problem in his book *Théorie des Questionnaires* (1965), and an extensive exploration was undertaken by Milton Sobel [*Univ. of Minnesota, Dept. of Statistics*, report 113 (November, 1968)].

Sobel constructed the procedure of Fig. 42, which finds the second largest of six elements using only $6\frac{1}{2}$ comparisons on the average. In the worst case, 8 comparisons are required, and this is worse than $V_2(6) = 7$; but all known procedures for this problem which require at most 7 comparisons use at least $6\frac{2}{3}$ comparisons on the average. Thus it is likely that no procedure which finds the second largest of six elements will be optimal in both the minimax and the minimean senses simultaneously.

Let $\overline{V}_t(n)$ denote the minimum average number of comparisons needed to find the tth largest of n elements. The following table shows the best upper bounds known for $\overline{V}_2(n)$, as computed by Sobel:

$$\begin{array}{rcccccccccc} n = & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \overline{V}_2(n) \le & 1 & 2\frac{2}{3} & 4 & 5\frac{4}{15} & 6\frac{1}{2} & 7\frac{17}{21} & 9 & 10\frac{1}{15} & 11\frac{4}{135} \end{array} \tag{12}$$

Sobel has conjectured that

$$\overline{V}_2(n) \ge n - 2 + \lfloor 2 \log_2 n \rfloor / 2. \tag{13}$$

R. W. Floyd discovered in 1970 that the median of n elements can be found

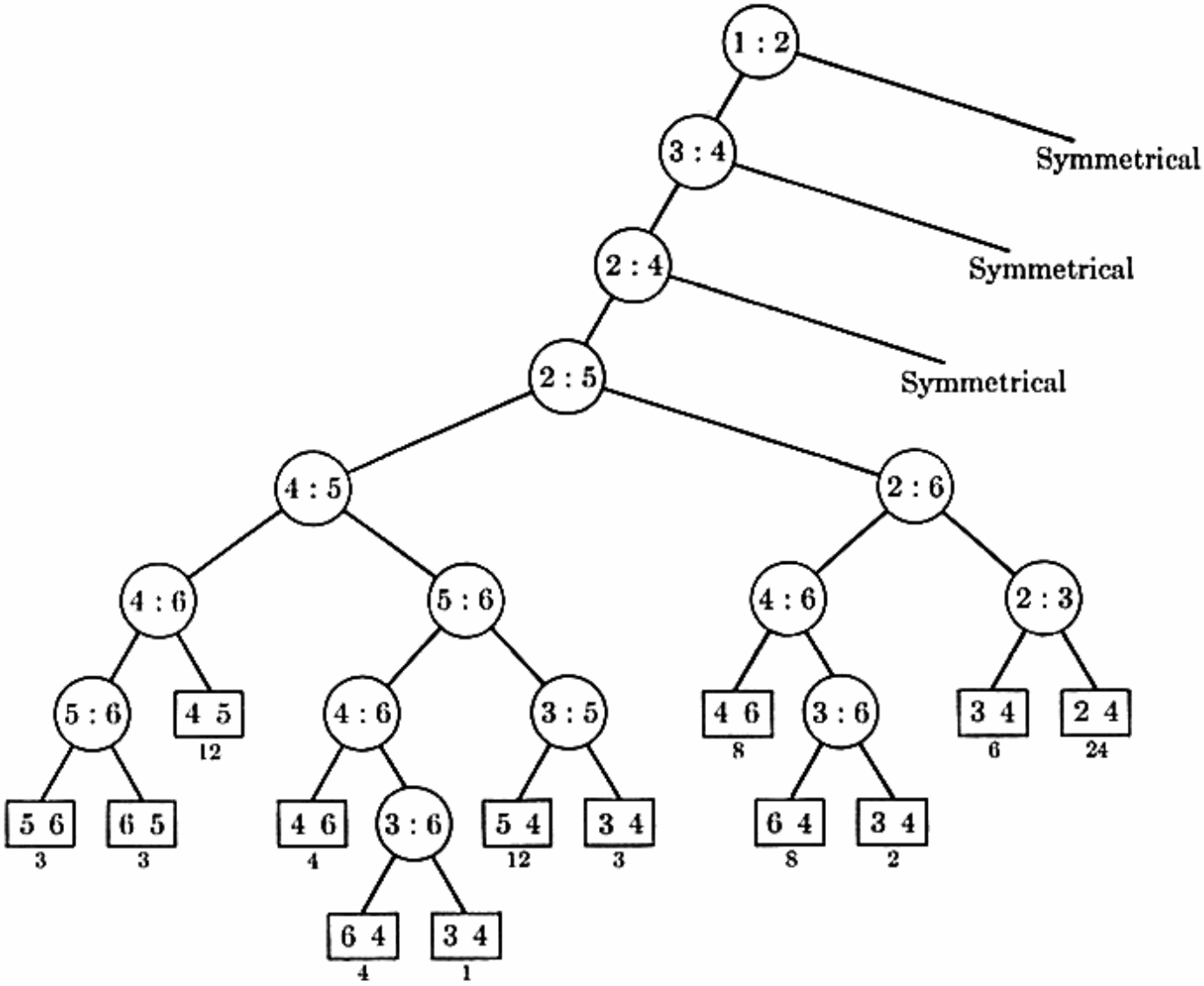


Fig. 42. A procedure which selects the second largest of $\{X_1, X_2, X_3, X_4, X_5, X_6\}$, using $6\frac{1}{2}$ comparisons on the average. Each "symmetrical" branch is identical to its brother, with names permuted in some appropriate manner. External nodes contain "j k" when X_j is known to be the second largest and X_k the largest; the number of permutations leading to this node appears immediately below it.

with only $\frac{3}{2}n + O(n^{2/3} \log n)$ comparisons, on the average. (See exercise 13.) He proved in fact that

$$\overline{V}_t(n) \leq n + t + f(n), \qquad \text{where} \qquad \lim_{n\to\infty} f(n)/n = 0. \tag{14}$$

This result is conjectured to be the best possible asymptotic formula, but no satisfactory lower bound has yet been obtained.

EXERCISES

1. [*15*] In Lewis Carroll's tournament (Figs. 39 and 40), why does player *13* get eliminated even though he has won his match in Round 3?

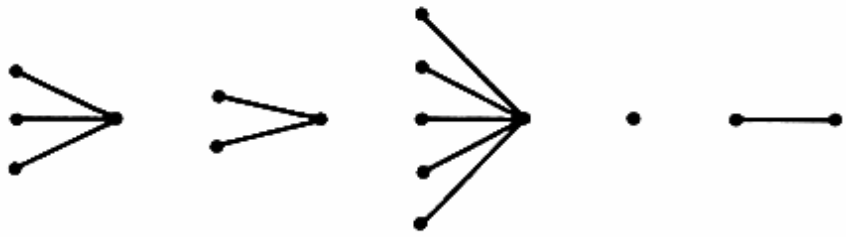
▶ **2.** [*M25*] Prove that after we have found the tth largest of n elements by a sequence of comparisons, we also know which $t - 1$ elements are greater than it, and which $n - t$ elements are less than it.

3. [*M21*] Prove that $V_t(n) \geq V_t(n-1) + 1$, for $1 \leq t \leq n$.

4. [*M20*] Prove that $W_t(n) \geq \lceil \log_2 n^{\underline{t}} \rceil$, where $n^{\underline{t}} = n(n-1)\ldots(n+1-t)$.

5. [*10*] Prove that $W_3(n) \leq V_3(n) + 1$.

▸ **6.** [*M26*] (R. W. Floyd.) Given n distinct elements $\{X_1, \ldots, X_n\}$ and relations $X_i < X_j$ for certain pairs (i, j), we wish to find the second largest element. If we know that $X_i < X_j$ and $X_i < X_k$ for $j \neq k$, X_i cannot possibly be the second largest, so it can be eliminated. The resulting relations now have a form such as



namely, m groups of elements which can be represented by a vector $(l_1, l_2, \ldots, l_m)$; the jth group contains $l_j + 1$ elements, one of which is known to be greater than the others. For example, the above configuration can be described by the vector $(3, 2, 4, 0, 1)$; when no relations are known we have a vector of n zeros.

Let $f(l_1, l_2, \ldots, l_m)$ be the minimum number of comparisons needed to find the second largest element of such a partially ordered set. Prove that

$$f(l_1, l_2, \ldots, l_m) = m - 2 + \lceil \log_2 (2^{l_1} + 2^{l_2} + \cdots + 2^{l_m}) \rceil.$$

[*Hint:* Show that the best strategy is always to compare the largest elements of the two smallest groups, until reducing m to unity; use induction on $l_1 + l_2 + \cdots + l_m + 2m$.]

7. [*M20*] Prove (8).

8. [*M21*] Kislitsyn's formula (6) is based on tree selection sorting using the complete binary tree with n external nodes. Would a tree selection method based on some *other* tree give a better bound, for any t and n?

▸ **9.** [*20*] Draw a comparison tree for finding the median of five elements in at most six steps, using the replacement-selection method of Hadian and Sobel [cf. (11)].

10. [*35*] Show that the median of seven elements can be found in at most 10 steps.

11. [*28*] (Hadian and Sobel.) Show that the median of nine elements can be found in at most 15 steps, by extending the method of Doren stated in the text.

12. [*21*] (Hadian and Sobel.) Prove that $V_3(n) \leq V_3(n-1) + 2$. [*Hint:* Start by discarding the smallest of $\{X_1, X_2, X_3, X_4\}$.]

▸ **13.** [*HM28*] (R. W. Floyd.) Show that if we start by finding the median element of $\{X_1, \ldots, X_{n^{2/3}}\}$, using a recursively defined method, we can go on to find the median of $\{X_1, \ldots, X_n\}$ with an average of $\frac{3}{2}n + O(n^{2/3} \log n)$ comparisons.

▸ **14.** [*20*] (M. Sobel.) Show that it is possible to find the largest two elements of $\{X_1, X_2, X_3, X_4, X_5\}$, without necessarily knowing their relative order, using at most five comparisons.

15. [*22*] (I. Pohl.) Suppose that we are interested in minimizing space instead of time. What is the minimum number of data words needed in memory in order to compute the tth largest of n elements, if each element fills one word and if the elements are input one at a time into a single register?

▶**16.** [*25*] (I. Pohl.) Show that we can find both the maximum and the minimum of a set of n elements, using at most $\lceil \frac{3}{2}n \rceil - 2$ comparisons; and the latter number cannot be lowered. [*Hint:* Any stage in such an algorithm can be represented as a quadruple (a, b, c, d), where a elements have never been compared, b have won but never lost, c have lost but never won, d have both won and lost. Construct a suitable oracle.]

17. [*20*] (R. W. Floyd.) Show that it is possible to select, in order, both the k largest and the l smallest elements of a set of n elements, using at most $\lceil \frac{3}{2}n \rceil - k - l + \sum_{n+1-k<j\le n} \lceil \log_2 j \rceil + \sum_{n+1-l<j\le n} \lceil \log_2 j \rceil$ comparisons.

18. [*M20*] If groups of size 5, not 7, had been used in the proof of Theorem L, what theorem would have been obtained?

19. [*M44*] Find the exact value of $\overline{V}_2(6)$. Can this be achieved by a procedure which never makes more than seven comparisons?

20. [*M47*] Prove (or disprove) Sobel's conjecture (13).

21. [*25*] (S. Lin.) Prove that $W_3(2^k + 2) \le 2^k + 2k$, when $k \ge 3$.

22. [*M47*] Determine the values of $V_3(n)$ and $W_3(n)$ for all n.

23. [*M49*] What is the asymptotic value of $V_{\lceil n/2 \rceil}(n)$, as $n \to \infty$?

24. [*M48*] What is the asymptotic value of $\overline{V}_{\lceil n/2 \rceil}(n)$, as $n \to \infty$?

5.3.4. Networks for Sorting

In this section we shall study a "constrained" type of sorting which is especially interesting because of its applications and its rich underlying theory. The new constraint is to insist on a *homogeneous* sequence of comparisons, in the sense that whenever we compare K_i versus K_j the subsequent comparisons for the case $K_i < K_j$ are exactly the same as for the case $K_i > K_j$, but with i and j interchanged. Figure 43(a) shows a comparison tree in which this homogeneity condition is satisfied. (Note that the same number of comparisons is made on each level, so there are 2^m outcomes after m comparisons have been made; since $n!$ is not a power of 2, it follows that some of the comparisons will be redundant, in the sense that one of their subtrees can never arise in practice. In other words, some branches of the tree must make more comparisons than necessary in order to ensure that all of the corresponding branches of the tree will sort properly.)

Since each path from top to bottom of such a tree determines the entire tree, such a sorting scheme is most easily represented as a *network* as in Fig. 42(b). The boxes in such a network represent "comparator modules" which have two inputs (represented as lines coming into the module from above) and two outputs (represented as lines leading downward); the left-hand output is the smaller of the two inputs, and the right-hand output is the larger. At the bottom of the network, K_1' is the smallest of $\{K_1, K_2, K_3, K_4\}$, K_2' the second smallest, etc. It is not difficult to prove that any sorting network corresponds to a homogeneous comparison tree in the above sense, and any homogeneous tree corresponds to a network of comparator modules.

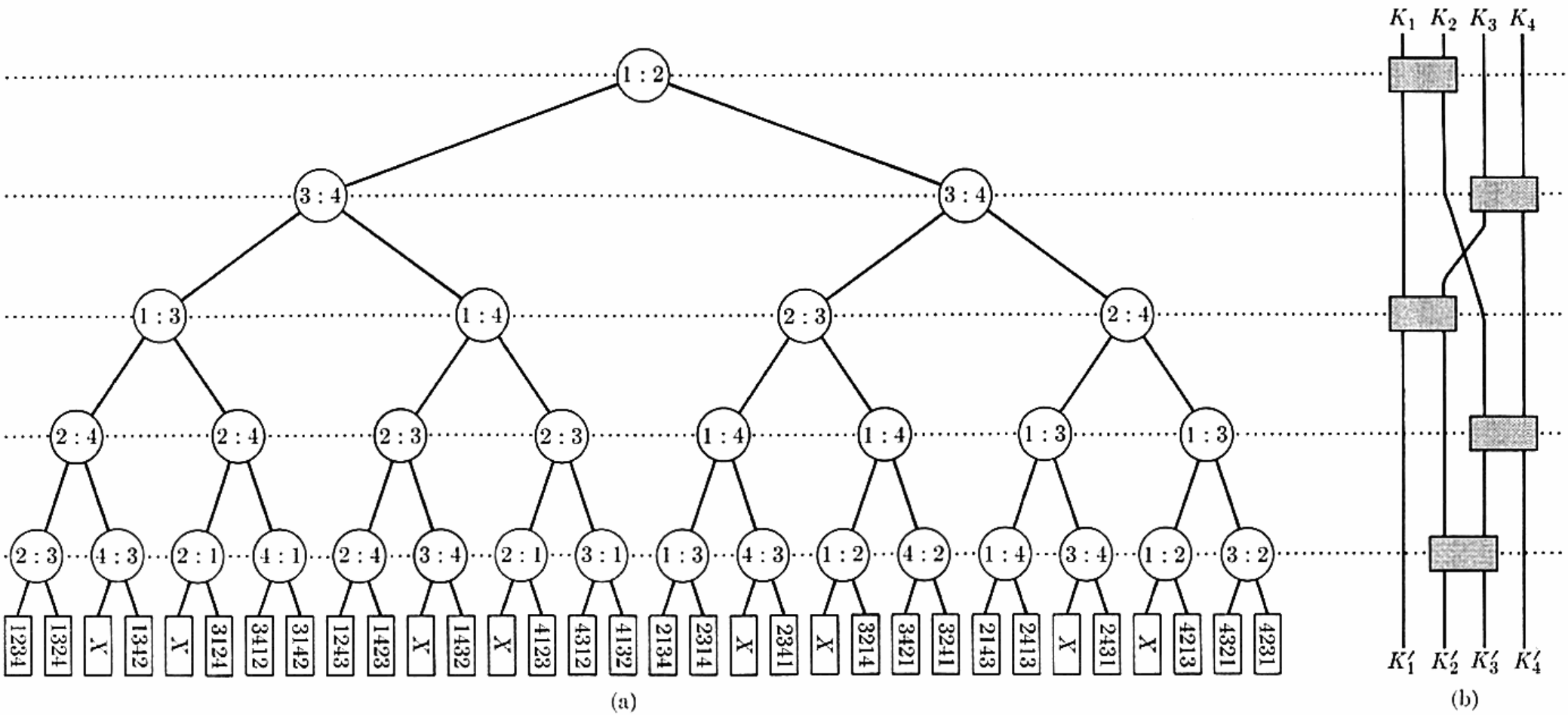


Fig. 43. (a) A homogeneous comparison tree. (b) The corresponding network.

Incidentally, we may note that comparator modules are fairly easy to manufacture, from an engineering point of view. For example, assume that the lines contain binary numbers, where one bit enters each module per unit time, most significant bit first. Each comparator module has three states, and behaves as follows:

Time t		Time $(t+1)$	
State	Inputs	State	Outputs
0	0 0	0	0 0
0	0 1	1	0 1
0	1 0	2	0 1
0	1 1	0	1 1
1	x y	1	x y
2	x y	2	y x

Initially all modules are in state 0 and are outputting 0 0. A module enters either state 1 or state 2 as soon as its inputs differ. Numbers which begin to be transmitted at the top of Fig. 43(b) at time t will begin to be output at the bottom, in sorted order, at time $t+3$, if a suitable delay element is attached to the K_1' and K_4' lines.

In order to develop the theory of sorting networks it is convenient to represent them in a slightly different way, illustrated in Fig. 44. Here numbers enter at the left, and comparator modules are represented by vertical connections between two lines; each comparator causes an interchange of its inputs, if necessary, so that the larger number appears on the lower line after passing the comparator. At the right of the diagram all the numbers are in order from top to bottom.

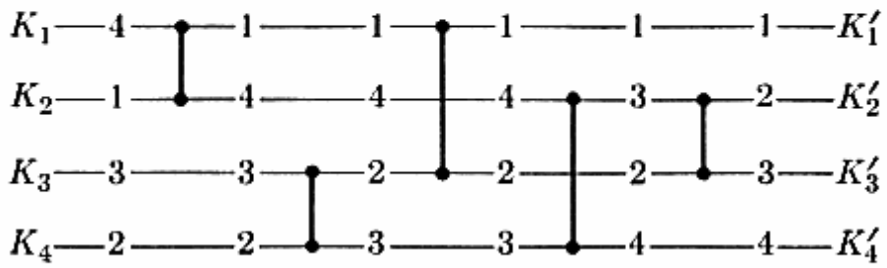


Fig. 44. Another way to represent the network of Fig. 43, as it sorts the sequence $\langle 4, 1, 3, 2 \rangle$.

Our previous studies of optimal sorting have concentrated on minimizing the number of comparisons with little or no regard for any underlying data movement or for the complexity of the decision structure that may be necessary. In this respect sorting networks have some advantages, since the data can be maintained in n locations and the decision structure is "straight line"; there is no need to remember the results of previous comparisons—the plan is immutably fixed in advance. Another important advantage of sorting networks is that it may be possible to overlap several of the operations, performing them

simultaneously (on a suitable machine). For example, the five steps in Figs. 43 and 44 can be collapsed into three when simultaneous nonoverlapping comparisons are allowed, since the first two and the second two can be combined; we shall exploit this property of sorting networks later in this section. Thus sorting networks can be very useful, although it is not at all obvious that efficient n-element sorting networks can be constructed for large n; we may find that many additional comparisons are needed in order to keep the decision structure homogeneous.

There are two simple ways to construct a sorting network for $n + 1$ elements when an n-element network is given, using either the principle of *insertion* or the principle of *selection*. Figure 45(a) shows how the $(n + 1)$st element can be inserted into its proper place after the first n elements have been sorted; and part (b) of the figure shows how the largest element can be selected before we proceed to sort the remaining ones. Repeated application of Fig. 45(a) gives the network analog of straight insertion sorting (Algorithm 5.2.1S), and repeated application of Fig. 45(b) yields the network analog of the bubble sort (Algorithm 5.2.2B). Figure 46 shows the corresponding six-element networks. It is interesting to note that when we collapse either network together to allow simultaneous operations, both methods reduce to the same "triangular" $(2n - 3)$-stage procedure (Fig. 47).

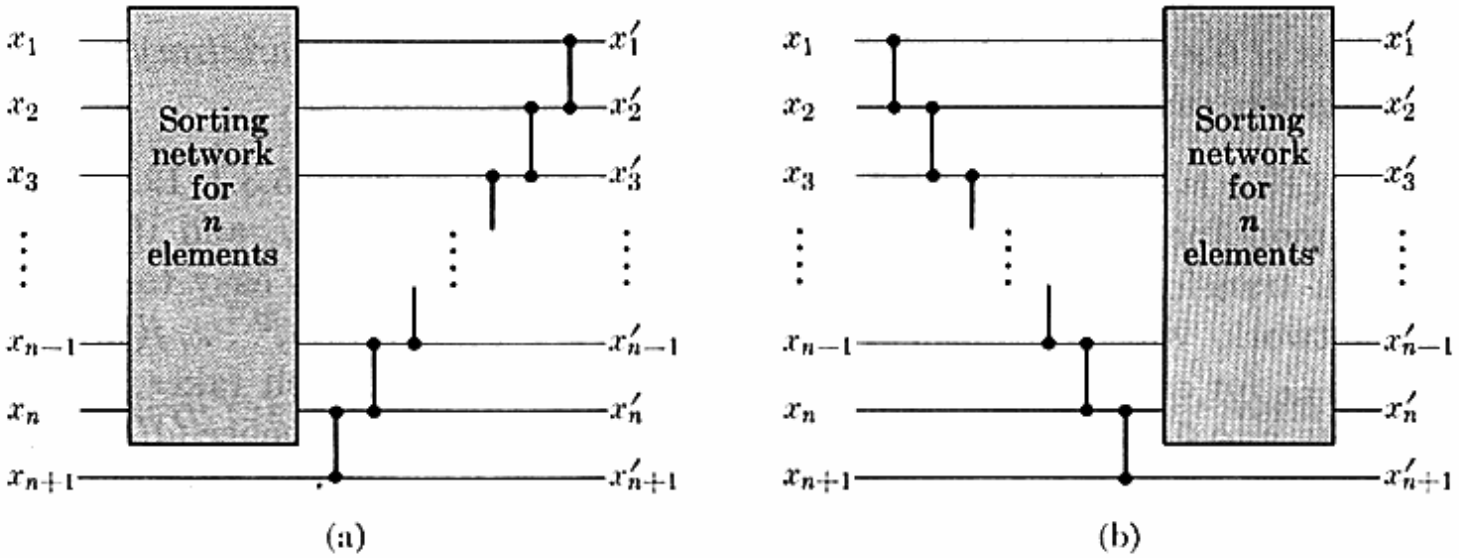


Fig. 45. Making $(n + 1)$-sorters from n-sorters: (a) insertion, (b) selection.

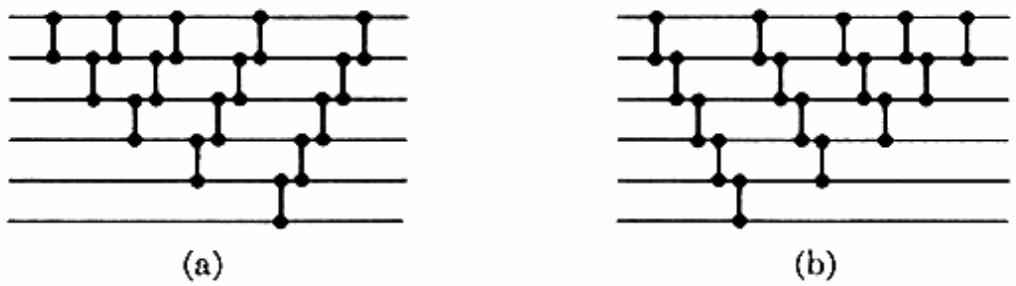


Fig. 46. Network analogs of elementary internal sorting schemes, obtained by applying Fig. 45 repeatedly: (a) straight insertion, (b) bubble sort.

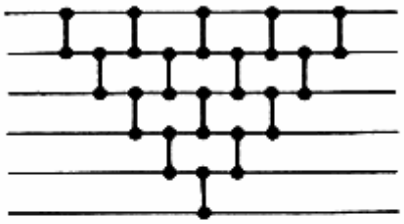
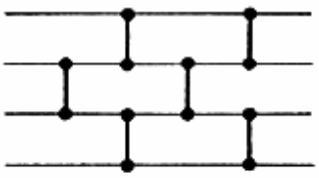
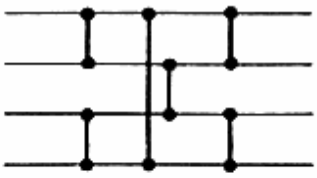


Fig. 47. With parallelism, straight insertion = bubble sort!

It is easy to prove that the network of Figs. 43 and 44 will sort any set of four numbers into order, since the first four comparators route the smallest and the largest elements to the correct places, and the last comparator puts the remaining two elements in order. But it is not always so easy to tell whether or not a given network will sort all possible input sequences; for example, both



and



are valid 4-element sorting networks, but the proofs of their validity are not trivial. It would be sufficient to test each n-element network on all $n!$ permutations of n distinct numbers, but in fact we can get by with far fewer tests:

Theorem Z (*Zero-one principle*). *If a network with n input lines sorts all 2^n sequences of 0's and 1's into nondecreasing order, it will sort any arbitrary sequence of n numbers into nondecreasing order.*

Proof. (This is a special case of Bouricius's theorem, exercise 5.3.1–12.) If $f(x)$ is any monotonic function, with $f(x) \le f(y)$ whenever $x \le y$, and if a given network transforms $\langle x_1, \ldots, x_n\rangle$ into $\langle y_1, \ldots, y_n\rangle$, then it is easy to see that the network will transform $\langle f(x_1), \ldots, f(x_n)\rangle$ into $\langle f(y_1), \ldots, f(y_n)\rangle$. If $y_i > y_{i+1}$ for some i, consider the monotonic function f which takes all numbers $< y_i$ into 0 and all numbers $\ge y_i$ into 1; this defines a sequence $\langle f(x_1), \ldots, f(x_n)\rangle$ of 0's and 1's which is not sorted by the network. Hence if all $0 - 1$ sequences are sorted, we have $y_i \le y_{i+1}$ for $1 \le i < n$. ∎

The zero-one principle is quite helpful in the construction of sorting networks. As a nontrivial example, we can derive a generalized version of Batcher's "merge exchange" sort (Algorithm 5.2.2M). The idea is to sort $m + n$ elements by sorting the first m and the last n independently, then applying an (m, n)-*merging network* to the result. An (m, n)-merging network can be constructed inductively as follows:

a) If $m = 0$ or $n = 0$, the network is empty. If $m = n = 1$, the network is a single comparator module.
b) If $mn > 1$, let the sequences to be merged be $\langle x_1, \ldots, x_m\rangle$ and $\langle y_1, \ldots, y_n\rangle$. Merge the "odd sequences" $\langle x_1, x_3, \ldots, x_{2\lceil m/2\rceil-1}\rangle$ and $\langle y_1, y_3, \ldots, y_{2\lceil n/2\rceil-1}\rangle$, obtaining the sorted result $\langle v_1, v_2, \ldots, v_{\lceil m/2\rceil+\lceil n/2\rceil}\rangle$; and merge

the "even sequences" $\langle x_2, x_4, \ldots, x_{\lfloor m/2 \rfloor}\rangle$ and $\langle y_2, y_4, \ldots, y_{\lfloor n/2 \rfloor}\rangle$, obtaining the sorted result $\langle w_1, w_2, \ldots, w_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor}\rangle$. Finally, apply the comparison-interchange operations

$$w_1 : v_2, \quad w_2 : v_3, \quad w_3 : v_4, \quad \ldots, \quad w_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor} : v^* \tag{1}$$

to the sequence

$$\langle v_1, w_1, v_2, w_2, v_3, w_3, \ldots, v_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor}, w_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor}, v^*, v^{**}\rangle; \tag{2}$$

the result will be sorted. (!) (Here $v^* = v_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor + 1}$ does not exist if both m and n are even, and $v^{**} = v_{\lfloor m/2 \rfloor + \lfloor n/2 \rfloor + 2}$ does not exist unless both m and n are odd; the total number of comparator modules indicated in (1) is $\lfloor (m + n - 1)/2. \rfloor$)

Batcher's (m, n)-merging network is called the *odd-even merge*. A (4, 7)-merge constructed according to these principles is illustrated in Fig. 48.

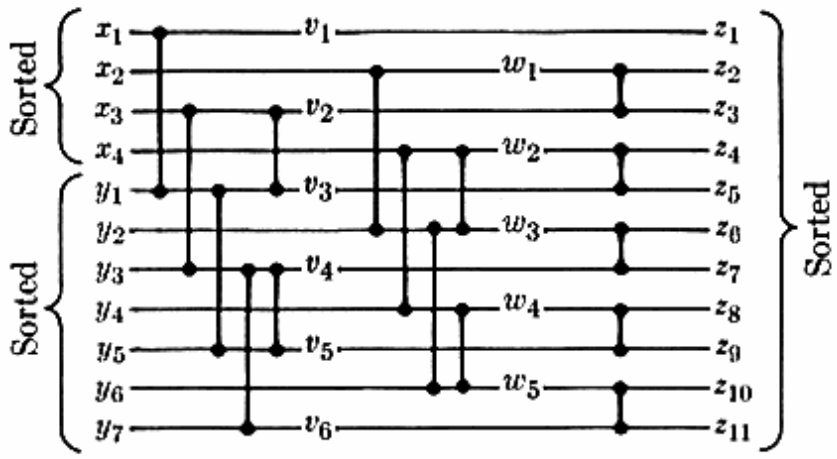


Fig. 48. The odd-even merge, when $m = 4$ and $n = 7$.

To prove that this rather strange merging procedure actually works, when $mn > 1$, we use the zero-one principle, testing it on all sequences of 0's and 1's. After the initial m-sort and n-sort, the sequence $\langle x_1, \ldots, x_m\rangle$ will consist of k 0's followed by $m - k$ 1's, and the sequence $\langle y_1, \ldots, y_n\rangle$ will be l 0's followed by $n - l$ 1's, for some k and l. Hence the sequence $\langle v_1, v_2, \ldots\rangle$ will consist of exactly $\lceil k/2 \rceil + \lceil l/2 \rceil$ 0's, followed by 1's; and $\langle w_1, w_2, \ldots\rangle$ will consist of $\lfloor k/2 \rfloor + \lfloor l/2 \rfloor$ 0's, followed by 1's. Now here's the point:

$$(\lceil k/2 \rceil + \lceil l/2 \rceil) - (\lfloor k/2 \rfloor + \lfloor l/2 \rfloor) = 0, \quad 1, \quad \text{or} \quad 2. \tag{3}$$

If this difference is 0 or 1, the sequence (2) is already in order, and if the difference is 2 one of the comparison-interchanges in (1) will fix everything up. This completes the proof. (Note that the zero-one principle reduces the merging problem from a consideration of $\binom{m+n}{m}$ cases to only $(m + 1)(n + 1)$, represented by the two parameters k and l.)

Let $C(m, n)$ be the number of comparator modules used in the odd-even merge for m and n, not counting the initial m-sort and n-sort; we have

$$C(m, n) = \begin{cases} mn, & \text{if} \quad mn \le 1; \\ C(\lceil m/2 \rceil, \lceil n/2 \rceil) + C(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor) & \\ \quad + \lfloor (m + n - 1)/2 \rfloor, & \text{if} \quad mn > 1. \end{cases} \tag{4}$$

This is not an especially simple function of m and n, in general, but by noting that $C(1, n) = n$ and that

$$\begin{aligned} &C(m + 1, n + 1) - C(m, n) \\ &\quad = 1 + C(\lfloor m/2 \rfloor + 1, \lfloor n/2 \rfloor + 1) - C(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor), \qquad \text{if} \qquad mn \ge 1, \end{aligned}$$

we can derive the relation

$$C(m + 1, n + 1) - C(m, n) = t + 2 + \lfloor n/2^{t+1} \rfloor, \qquad \text{if} \quad n \ge m \ge 1 \quad \text{and} \quad t = \lfloor \log_2 m \rfloor. \tag{5}$$

Consequently,

$$C(m, m + r) = B(m) + m + R_m(r), \qquad \text{for} \qquad m \ge 0, r \ge 0, \tag{6}$$

where $B(m)$ is the "binary insertion" function $\sum_{1 \le k \le m} \lceil \log_2 k \rceil$ of Eq. 5.3.1–3, and where $R_m(r)$ denotes the sum of the first m terms of the series

$$\left\lfloor \frac{r+0}{1} \right\rfloor + \left\lfloor \frac{r+1}{2} \right\rfloor + \left\lfloor \frac{r+2}{4} \right\rfloor + \left\lfloor \frac{r+3}{4} \right\rfloor + \left\lfloor \frac{r+4}{8} \right\rfloor + \cdots + \left\lfloor \frac{r+j}{2^{\lfloor \log_2 j \rfloor + 1}} \right\rfloor + \cdots. \tag{7}$$

In particular, when $r = 0$, we have the important special case

$$C(m, m) = B(m) + m. \tag{8}$$

Furthermore if $t = \lceil \log_2 m \rceil$,

$$\begin{aligned} R_m(r + 2^t) &= R_m(r) + 1 \cdot 2^{t-1} + 2 \cdot 2^{t-2} + \cdots + 2^{t-1} \cdot 2^0 + m \\ &= R_m(r) + m + t \cdot 2^{t-1}. \end{aligned}$$

Hence $C(m, n + 2^t) - C(m, n)$ has a simple form, and

$$C(m, n) = \left(\frac{t}{2} + \frac{m}{2^t}\right) n + O(1), \qquad \text{for } m \text{ fixed}, \quad n \to \infty, \quad t = \lceil \log_2 m \rceil; \tag{9}$$

the $O(1)$ term is an eventually periodic function of n, with period length 2^t. The asymptotic value of $C(n, n)$ as $n \to \infty$ is $n \log_2 n$, by Eq. (8) and exercise 5.3.1–15.

Minimum-comparison networks. Let $\hat{S}(n)$ be the minimum number of comparators needed in a sorting network for n elements; clearly $\hat{S}(n) \ge S(n)$,

where $S(n)$ is the minimum number of comparisons needed in an unrestricted sorting procedure (cf. Section 5.3.1). We have $\hat{S}(4) = 5 = S(4)$, so the new constraint causes no loss of efficiency when $n = 4$; but already when $n = 5$ it turns out that $\hat{S}(5) = 9$ while $S(5) = 7$. The problem of determining $\hat{S}(n)$ seems to be even harder than the problem of determining $S(n)$; even the asymptotic behavior of $\hat{S}(n)$ is still unknown.

It is interesting to trace the history of this problem, since each step was forged with some difficulty. Sorting networks were first explored by P. N. Armstrong, R. J. Nelson, and D. J. O'Connor, about 1954 [see *U.S. Patent 3029413*]; they showed that $\hat{S}(n+1) \le \hat{S}(n) + n$. In the words of their patent attorney, "By the use of skill, it is possible to design economical n-line sorting switches using a reduced number of two-line sorting switches"; they gave special constructions for $4 \le n \le 8$, using 5, 9, 12, 18, and 19 comparators, respectively. In Nelson's subsequent work on the problem together with R. C. Bose, a general procedure for constructing sorting networks was devised prior to 1960, showing that $\hat{S}(2^n) \le 3^n - 2^n$ for all n; hence $\hat{S}(n) = O(n^{\log_2 3}) = O(n^{1.585})$. Bose and Nelson published their interesting method in *JACM* **9** (1962), 282–296, where they conjectured that it was best possible; T. N. Hibbard [*JACM* **10** (1963), 142–150] found a similar but slightly simpler method which used the same number of comparisons, thereby reinforcing the conjecture.

In 1964, R. W. Floyd and D. E. Knuth found a new way to approach the problem, leading to an asymptotic bound of the form $\hat{S}(n) = O(n^{1+c/\sqrt{\log n}})$. Working independently, K. E. Batcher discovered the general merging strategy outlined above; using a number of comparators defined by

$$c(1) = 0, \quad c(n) = c(\lceil n/2 \rceil) + c(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil, \lfloor n/2 \rfloor) \quad \text{for} \quad n \ge 2, \tag{10}$$

he proved that (see exercise 5.2.2–14)

$$c(2^t) = (t^2 - t + 4)2^{t-2} - 1;$$

and it follows that $\hat{S}(n) = O(n(\log n)^2)$. Neither Batcher nor Floyd and Knuth published their constructions until some time later [*Notices of the Amer. Math. Soc.* **14** (1967), 283; *Proc. AFIPS Spring Joint Computer Conference* **32** (1968), 307–314].

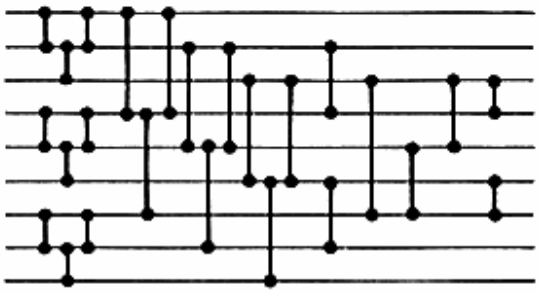
Several people have found ways to improve on the number of comparators used by Batcher's merge-exchange construction; the following table shows the best upper bounds currently known for $\hat{S}(n)$:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
$c(n) =$	0	1	3	5	9	12	16	19	26	31	37	41	48	53	59	63	(11)
$\hat{S}(n) \le$	0	1	3	5	9	12	16	19	25	29	35	39	46	51	56	60	

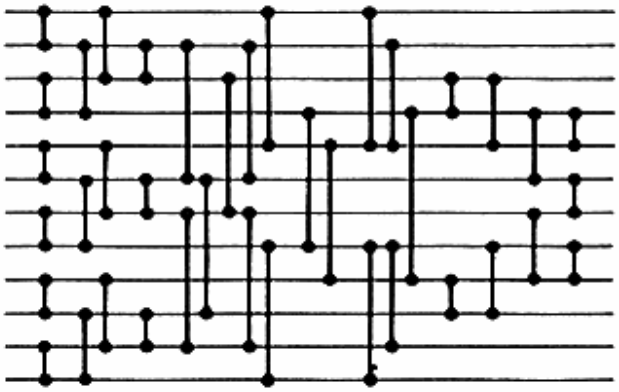
Since $\hat{S}(n) < c(n)$ for $8 < n \le 16$, merge exchange is nonoptimal for all $n > 8$. When $n \le 8$, merge exchange is equivalent in the number of comparators to

the construction of Bose and Nelson. Floyd and Knuth proved in 1964–1966 that the above values of $\hat{S}(n)$ are *exact* for $n \leq 8$ [see *A Survey of Combinatorial Theory* (North-Holland, 1972), Chapter 15]; the values of $\hat{S}(n)$ for $n > 8$ are still not known.

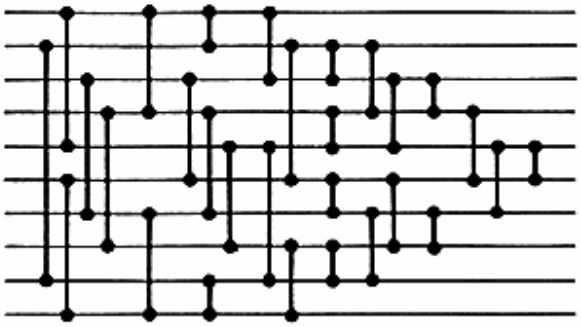
Constructions which lead to the above values for $\hat{S}(n)$ are shown in Fig. 49. The network for $n = 9$, based on an interesting three-way merge, was found by R. W. Floyd in 1964; its validity can be established by using the general principle described in exercise 27. The network for $n = 10$ was discovered by A. Waksman in 1969, by regarding the inputs as permutations of $\{1, 2, \ldots, 10\}$ and trying to reduce as much as possible the number of possible values that can appear on each line at a given stage, while maintaining some symmetry.



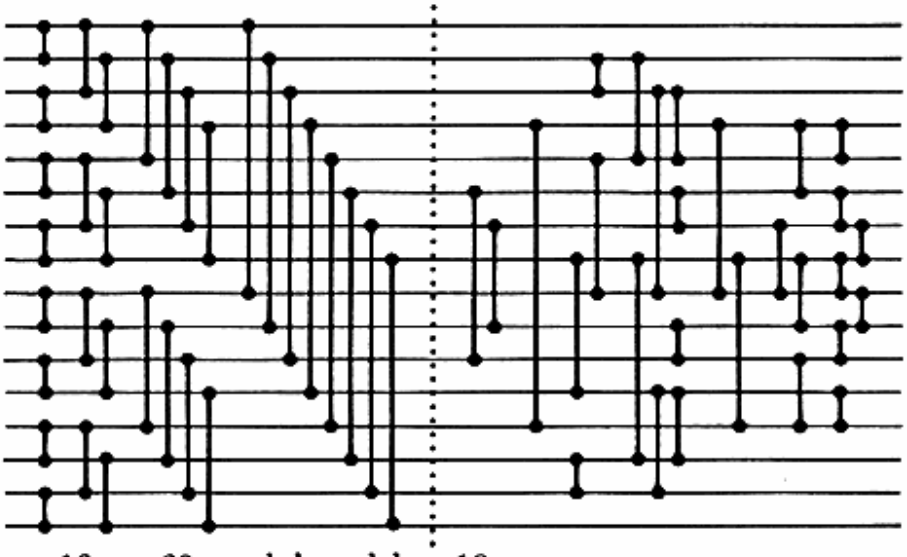
$n = 9$ 25 modules, delay 10



$n = 12$ 39 modules, delay 9



$n = 10$ 29 modules, delay 9



$n = 16$ 60 modules, delay 10

Fig. 49. Efficient sorting networks.

A 62-comparator sorting network for 16 elements was found by G. Shapiro in 1969, and this was rather surprising since Batcher's method (63 comparisons) would appear to be at its best when n is a power of 2. Soon after hearing of Shapiro's construction, M. W. Green tripled the amount of surprise by finding the 60-comparison sorter in Fig. 49. The first portion of Green's construction is fairly easy to understand; after the 32 comparison/interchanges to the left of the dotted line have been made, the lines can be labeled with the 16 subsets of $\{a, b, c, d\}$, in such a way that the line labeled s is known to contain a number less than or equal to the contents of the line labeled t whenever s is a subset of t. The state of the sort at this point is discussed further in exercise 32. Comparisons made on subsequent levels of Green's network become increasingly mysterious, however, and as yet nobody has seen how to generalize the construction in order to obtain correspondingly efficient networks for higher values of n.

Shapiro and Green also discovered the network shown for $n = 12$. When $n = 11$, 13, 14, or 15, good networks can be found by removing the bottom line of the network for $n + 1$, together with all comparators touching that line.

For the best networks currently known as $n \to \infty$, see the Ph.D. thesis of D. Van Voorhis (Stanford University, 1971); his networks require asymptotically $\frac{1}{4}n(\log_2 n)^2 - \alpha n \log_2 n$ comparators, where $\alpha = \frac{1}{4} + \frac{1}{6}\sum_{k\geq 0} 2^{-2k-k} \approx 0.356852$.

Minimum-time networks. In physical realizations of sorting networks, and on parallel computers, it is possible to do nonoverlapping comparison-exchanges at the same time; therefore it is natural to try to minimize the delay time. A moment's reflection shows that the delay time of a sorting network is equal to the maximum number of comparators in contact with any "path" through the network, if we define a path to consist of any left-to-right route which possibly switches lines at the comparators. We can put a sequence number on each comparator indicating the earliest time it can be executed; this is one higher than the maximum of the sequence numbers of the comparators which precede it. (See Fig. 50(a); part (b) of the figure shows the same network redrawn so that each comparison is done at the earliest possible moment.)

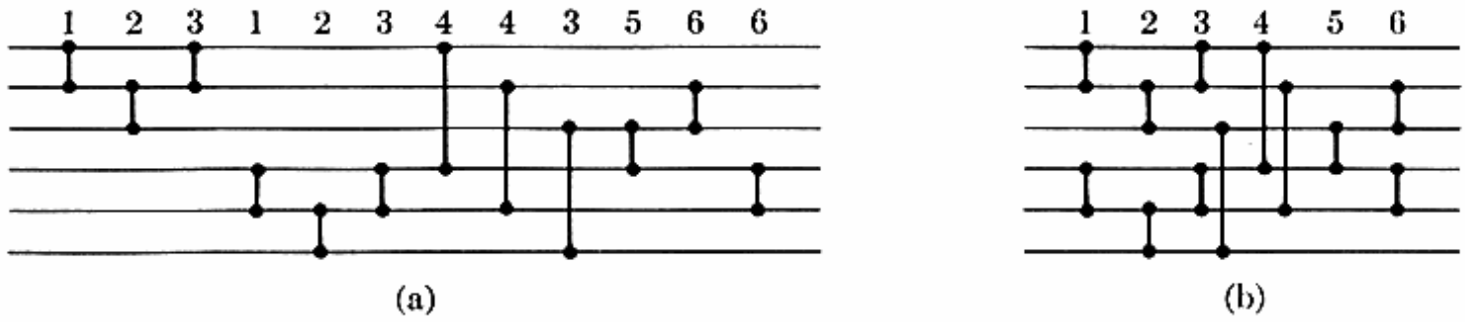


Fig. 50. Doing each comparison at the earliest possible time.

Batcher's odd-even merging network described above takes $T_B(m, n)$ units of time, where $T_B(m, 0) = T_B(0, n) = 0$, $T_B(1, 1) = 1$, and

$$T_B(m, n) = 1 + \max\left(T_B(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor), \quad T_B(\lceil m/2 \rceil, \lceil n/2 \rceil)\right) \quad \text{for} \quad mn \geq 2.$$

We can use these relations to prove that $T_B(m, n+1) \geq T_B(m, n)$, by induction; hence $T_B(m, n) = 1 + T_B(\lceil m/2 \rceil, \lceil n/2 \rceil)$ for $mn \geq 2$, and it follows that

$$T_B(m, n) = 1 + \lceil \log_2 \max(m, n) \rceil, \qquad \text{for} \qquad mn \geq 1. \tag{12}$$

Exercise 5 shows that Batcher's sorting method therefore has a delay time of

$$\binom{1 + \lceil \log_2 n \rceil}{2}. \tag{13}$$

Let $\hat{T}(n)$ be the minimum achievable delay time in any sorting network for n elements. It is possible to improve some of the networks described above so that they have less delay time but use no more comparators, as shown for $n = 6$ and $n = 9$ in Fig. 51, and for $n = 10$ in exercise 7. Still smaller delay time can be achieved if we add one or two extra modules, as shown in the remarkable networks for $n = 10$, 12, and 16 in Fig. 51. These constructions yield the following upper bounds on $\hat{T}(n)$ for small n:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\hat{T}(n) \leq$	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9

(14)

For $n \leq 8$ the values given here are known to be exact (see exercise 4). The networks in Fig. 51 merit careful study, because it is by no means obvious that they sort; they were discovered in 1969–1971 by G. Shapiro ($n = 6$, 9, 12) and D. Van Voorhis ($n = 10$, 16).

Merging networks. Let $\hat{M}(m, n)$ denote the minimum number of comparator modules needed in a network that merges m elements $x_1 \leq \cdots \leq x_m$ with n elements $y_1 \leq \cdots \leq y_n$ to form the sorted sequence $z_1 \leq \cdots \leq z_{m+n}$. At present no merging networks have been discovered which are superior to the odd-even merge described above; hence the function $C(m, n)$ in (6) represents the best upper bound known for $\hat{M}(m, n)$.

R. W. Floyd has discovered an interesting way to find *lower* bounds for this merging problem.

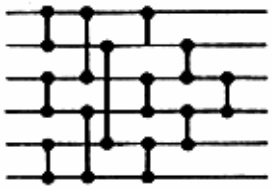
Theorem F. $\hat{M}(2n, 2n) \geq 2\hat{M}(n, n) + n$, *for all* $n \geq 1$.

Proof. Consider a network with $\hat{M}(2n, 2n)$ comparator modules, capable of sorting all input sequences $\langle z_1, \ldots, z_{4n} \rangle$ such that $z_1 \leq z_3 \leq \cdots \leq z_{4n-1}$ and $z_2 \leq z_4 \leq \cdots \leq z_{4n}$. We may assume that each module replaces (z_i, z_j) by $(\min(z_i, z_j), \max(z_i, z_j))$, for some $i < j$ (see exercise 16). The comparators can therefore be divided into three classes:

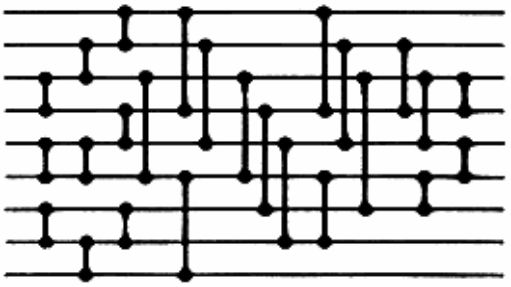
a) $i \leq 2n$ and $j \leq 2n$.

b) $i > 2n$ and $j > 2n$.

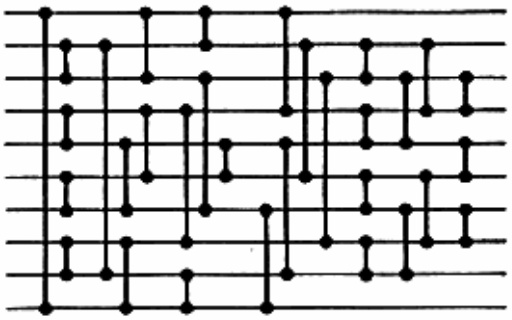
c) $i \leq 2n$ and $j > 2n$.



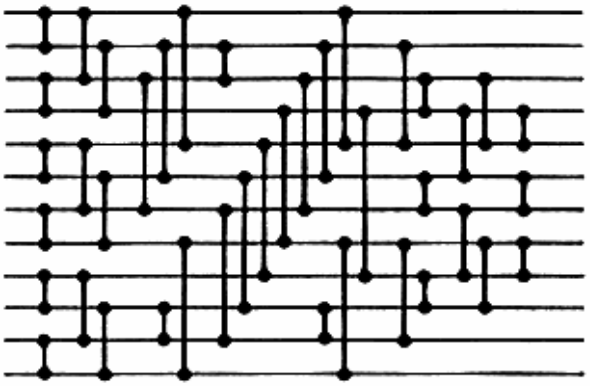
$n=6$ 12 modules, delay 5



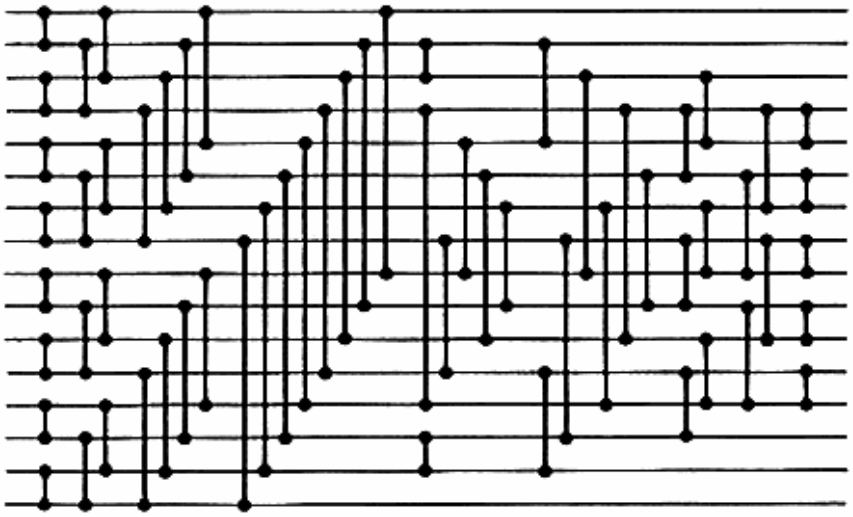
$n=9$ 25 modules, delay 8



$n=10$ 31 modules, delay 7



$n=12$ 40 modules, delay 8



$n=16$ 61 modules, delay 9

Fig. 51. Sorting networks which are unusually fast, when comparisons are performed in parallel.

Class (a) must contain at least $\hat{M}(n, n)$ comparators, since $z_{2n+1}, z_{2n+2}, \ldots, z_{4n}$ may be already in their final position when the merge starts; similarly, there are at least $\hat{M}(n, n)$ comparators in class (b). Furthermore the input sequence $\langle 0, 1, 0, 1, \ldots, 0, 1\rangle$ shows that class (c) contains at least n comparators, since n zeros must move from $\{z_{2n+1}, \ldots, z_{4n}\}$ to $\{z_1, \ldots, z_{2n}\}$. ∎

Repeated use of Theorem F proves that $\hat{M}(2^m, 2^m) \geq \frac{1}{2}(m+2)2^m$; hence $\hat{M}(n, n) \geq \frac{1}{2}n \log_2 n + O(n)$. Merging *without* the network restriction requires only $M(n, n) = 2n - 1$ comparisons; hence we have proved that merging with networks is intrinsically harder than merging in general. The odd-even merge

shows that $\hat{M}(n, n) \le C(n, n) = n \log_2 n + O(n)$, so the asymptotic behavior of $\hat{M}(n, n)$ is known within a factor of 2. (The exact value of $\hat{M}(n, n)$ is known for $n \le 5$, see exercise 9.)

Bitonic sorting. When simultaneous comparisons are allowed, we have seen in Eq. (12) that the odd-even merge uses $\lceil \log_2 (2n) \rceil$ units of delay time, when $1 \le m \le n$. Batcher has devised another type of network for merging, called a *bitonic sorter*, which lowers the delay time to $\lceil \log_2 (m + n) \rceil$ although it requires more comparator modules.

Let us say that a sequence $\langle z_1, \ldots, z_p \rangle$ of p numbers is *bitonic* if $z_1 \ge \cdots \ge z_k \le \cdots \le z_p$ for some k, $1 \le k \le p$. (Compare this with the ordinary definition of "monotonic" sequences.) A bitonic sorter of order p is a comparator network capable of sorting any bitonic sequence of length p into nondecreasing order. The problem of merging $x_1 \le \cdots \le x_m$ with $y_1 \le \cdots \le y_n$ is a special case of the bitonic sorting problem, since merging can be done by applying a bitonic sorter of order $m + n$ to the sequence $\langle x_m, \ldots, x_1, y_1, \ldots, y_n \rangle$.

Note that when the sequence $\langle z_1, \ldots, z_p \rangle$ is bitonic, so are all of its subsequences. Shortly after Batcher discovered the odd-even merging networks, he observed that we can construct a bitonic sorter of order p in an analogous way, by first sorting the bitonic subsequences $\langle z_1, z_3, z_5, \ldots \rangle$ and $\langle z_2, z_4, z_6, \ldots \rangle$ independently, then comparing and interchanging $z_1 : z_2$, $z_3 : z_4$, ... (See exercise 10 for a proof.) If $C'(p)$ is the corresponding number of comparator modules, we have

$$C'(p) = C'(\lceil p/2 \rceil) + C'(\lfloor p/2 \rfloor) + \lfloor p/2 \rfloor, \qquad \text{for} \qquad p \ge 2; \tag{15}$$

and the delay time is clearly $\lceil \log_2 p \rceil$. Figure 52 shows the bitonic sorter of order 7 constructed in this way: it can be used as a (3, 4)- as well as a (2, 5)-merging network, with three units of delay; the odd-even merge for $m = 2$ and $n = 5$ has one less comparator but one more level of delay.

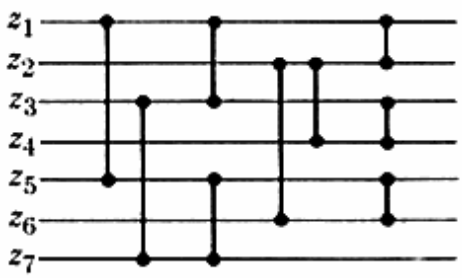


Fig. 52. Batcher's bitonic sorter of order 7.

Batcher's bitonic sorter of order 2^k is particularly interesting; it consists of k levels of 2^{k-1} comparators each. If we number the input lines $z_0, z_1, \ldots, z_{2^k-1}$, element z_i is compared to z_j on level l if and only if i and j differ only in the lth most significant bit of their binary representations. This simple structure leads to parallel sorting networks which are as fast as merge exchange (Algorithm 5.2.2M) but considerably easier to implement. (See exercises 11 and 13.)

When $m = n$, it is not difficult to show that both the odd-even merge and Batcher's bitonic sorter provide the absolute minimum delay time achievable in any merging network. The nth smallest output of an (n, n) merging network must depend upon all $2n$ of the inputs, and if it can be computed in l steps, it involves at most 2^l of the inputs; hence $2^l \geq 2n$, $l \geq \lceil \log_2 (2n) \rceil$.

When $m < n$, the nth output of an (m, n) merging network depends on $2m + 1$ of the inputs (cf. exercise 29), so the same argument gives a minimum delay time of $\lceil \log_2 (2m + 1) \rceil$ for merging in this case. Batcher has shown [Report GER-14122 (Akron, Ohio: Goodyear Aerospace Corporation, 1968]) that this minimum delay time is achievable if we allow "multiple fanout" in the network, i.e., the splitting of lines so that the same number is fed to many modules at the same time. For example, his network which is capable of merging one item with n others in only two levels of delay is illustrated for $n = 6$ in Fig. 53. Of course, networks with multiple fanout do not conform to our conventions, and it is fairly easy to see that any $(1, n)$ merging network without multiple fanout must have a delay time of $\log_2 (n + 1)$ or more. (See exercise 14.)

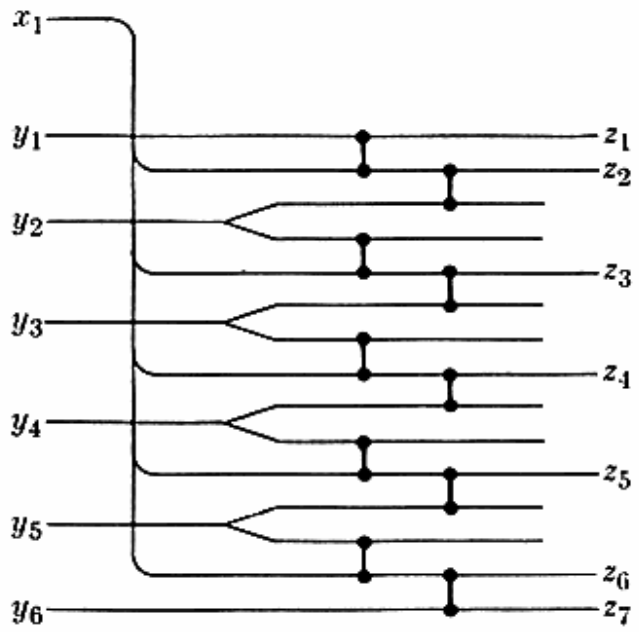


Fig. 53. Merging one item with six others, with multiple fanout, in order to achieve the minimum possible delay time.

Selection networks. We can also use networks to approach the problem of Section 5.3.3. Let $\hat{U}_t(n)$ denote the minimum number of comparators required in a network which moves the t largest of n distinct inputs into t specified output lines; they are allowed to appear in any order on these output lines. Let $\hat{V}_t(n)$ denote the minimum number of comparators required to move the tth largest of n distinct inputs into a specified output line; and let $\hat{W}_t(n)$ denote the minimum number of comparators required to move the t largest of n distinct inputs into t specified output lines in nondecreasing order. It is not difficult to see (cf. exercise 17) that

$$\hat{U}_t(n) \leq \hat{V}_t(n) \leq \hat{W}_t(n). \tag{16}$$

Suppose first that we have $2t$ elements $\langle x_1, \ldots, x_{2t}\rangle$ and we wish to select the largest t; V. E. Alekseyev [*Kibernetika* **5**, 5 (1969), 99–103] has observed that we can do the job by first sorting $\langle x_1, \ldots, x_t\rangle$ and $\langle x_{t+1}, \ldots, x_{2t}\rangle$, then comparing and interchanging

$$x_1 : x_{2t}, \qquad x_2 : x_{2t-1}, \qquad \ldots, \qquad x_t : x_{t+1}. \tag{17}$$

Since none of these pairs can contain more than one of the largest t elements (why?), Alekseyev's procedure must select the largest t elements.

If we want to select the t largest of nt elements, we can apply the procedure $n - 1$ times (eliminating t elements each time); hence

$$\hat{U}_t(nt) \le (n-1)(2\hat{S}(t) + t). \tag{18}$$

Alekseyev has also derived an interesting *lower* bound for the selection problem.

Theorem A. $\hat{U}_t(n) \ge (n-t)\lceil \log_2(t+1)\rceil$.

Proof. It is most convenient to consider the equivalent problem of selecting the *smallest* t elements. We can attach numbers (l, u) to each line of a comparator network, as shown in Fig. 54, where l and u denote respectively the minimum and maximum values that can appear at that position when the input is a permutation of $\{1, 2, \ldots, n\}$. Let l_i and l_j be the lower bounds on lines i and j before a comparison of $x_i : x_j$, and let l_i' and l_j' be the corresponding lower bounds after the comparison. It is obvious that $l_i' = \min(l_i, l_j)$, and exercise 24 proves the (nonobvious) relation

$$l_j' \le l_i + l_j. \tag{19}$$

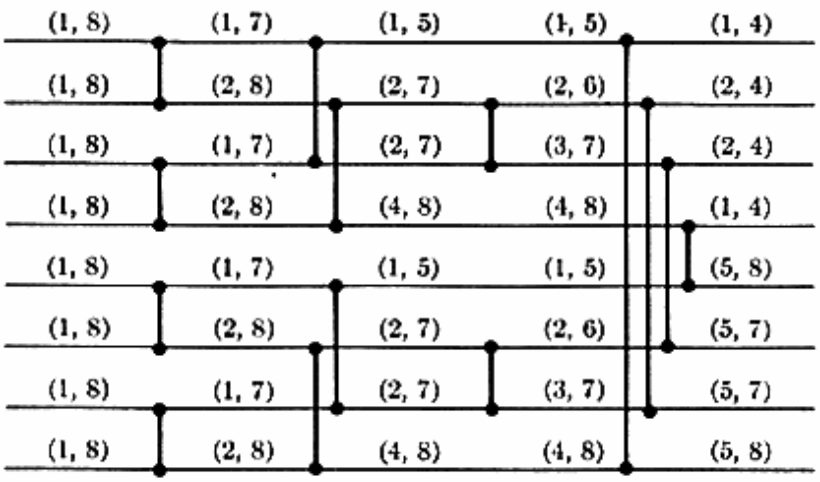


Fig. 54. Separating the largest four from the smallest four. (Numbers on these lines are used in the proof of Theorem A.)

Now let us reinterpret the network operations in another way (see Fig. 55): All input lines are assumed to contain zero, and each "comparator" now places the smaller of its inputs on the upper line and the larger *plus one* on the lower line. The resulting numbers $\langle m_1, m_2, \ldots, m_n \rangle$ have the property that

$$2^{m_i} \geq l_i \tag{20}$$

throughout the network, since this holds initially and it is preserved by each comparator because of (19). Furthermore, the final value of

$$m_1 + m_2 + \cdots + m_n$$

is the total number of comparators in the network, since each comparator adds unity to this sum.

If the network selects the smallest t numbers, $n - t$ of the l_i are $\geq t + 1$; hence $n - t$ of the m_i must be $\geq \lceil \log_2 (t + 1) \rceil$. ∎

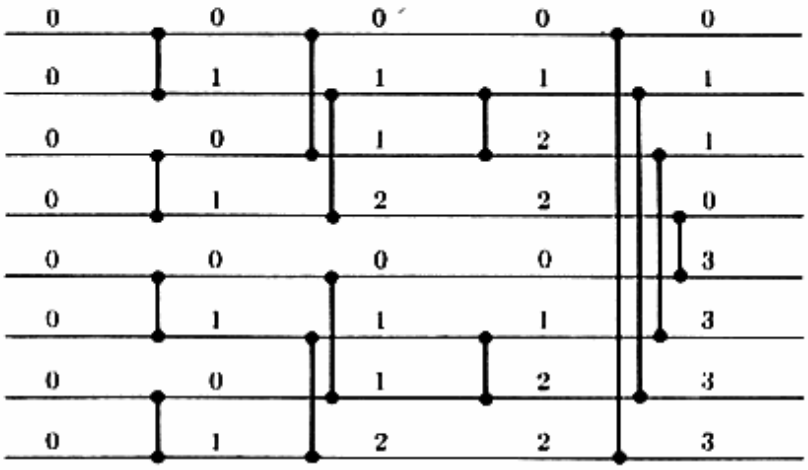


Fig. 55. Another interpretation for the network of Fig. 54.

The lower bound in Theorem A turns out to be exact when $t = 1$ and when $t = 2$ (see exercise 19). Table 1 gives some values of $\hat{U}_t(n)$, $\hat{V}_t(n)$, and $\hat{W}_t(n)$ for small t and n.

Table 1

COMPARISONS NEEDED IN SELECTION NETWORKS $(\hat{U}_t(n), \hat{V}_t(n), \hat{W}_t(n))$

	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
$n = 1$	(0, 0, 0)					
$n = 2$	(1, 1, 1)	(0, 1, 1)				
$n = 3$	(2, 2, 2)	(2, 3, 3)	(0, 2, 3)			
$n = 4$	(3, 3, 3)	(4, 5, 5)	(3, 5, 5)	(0, 3, 5)		
$n = 5$	(4, 4, 4)	(6, 7, 7)	(6, 7, 8)	(4, 7, 9)	(0, 4, 9)	
$n = 6$	(5, 5, 5)	(8, 9, 9)	(8, 10, 10)	(8, 10, 12)	(5, 9, 12)	(0, 5, 12)

EXERCISES—First Set

Several of the following exercises develop the theory of sorting networks in detail, and it is convenient to introduce some notation. We let $[i:j]$ stand for a comparison/interchange module. A network with n inputs and r comparator modules is written $[i_1:j_1][i_2:j_2]\ldots[i_r:j_r]$, where each of the i's and j's is $\leq n$; we shall call it an *n-network* for short. A network is called *standard* if $i_q < j_q$ for $1 \leq q \leq r$. Thus, for example, Fig. 44 on page 222 depicts a standard 4-network which is denoted by the comparator sequence $[1:2][3:4][1:3][2:4][2:3]$.

The text's convention for drawing network diagrams represents only standard networks; all comparators $[i:j]$ are represented by a line from i to j, where $i < j$. When nonstandard networks must be drawn, we can use an *arrow* from i to j, indicating that the larger number goes to the point of the arrow. For example, Fig. 56 illustrates a nonstandard network for 16 elements, whose comparators are $[1:2][4:3][5:6][8:7]$ etc. Exercise 11 proves that Fig. 56 is a sorting network.

If $x = \langle x_1, \ldots, x_n\rangle$ is an n-vector and α is an n-network, we write $x\alpha$ for the vector of numbers $\langle (x\alpha)_1, \ldots, (x\alpha)_n\rangle$ produced by the network. For brevity, we also let $a \vee b = \max(a, b)$, $a \wedge b = \min(a, b)$, $\bar{a} = 1 - a$. Thus $(x[i:j])_i = x_i \wedge x_j$, $(x[i:j])_j = x_i \vee x_j$, and $(x[i:j])_k = x_k$ when $i \neq k \neq j$. We say α is a *sorting network* iff $(x\alpha)_i \leq (x\alpha)_{i+1}$ for all x and for $1 \leq i < n$.

The symbol $e^{(i)}$ stands for a vector which is 1 in position i, 0 elsewhere; thus $(e^{(i)})_j = \delta_{ij}$. The symbol D_n stands for the set of all 2^n n-place vectors of 0's and 1's, and P_n stands for the set of all $n!$ vectors which are permutations of $\{1, 2, \ldots, n\}$. We write $x \wedge y$ and $x \vee y$ for the vectors $\langle x_1 \wedge y_1, \ldots, x_n \wedge y_n\rangle$ and $\langle x_1 \vee y_1, \ldots, x_n \vee y_n\rangle$, and we write $x \leq y$ if $x_i \leq y_i$ for all i. Thus $x \leq y$ iff $x \vee y = y$ iff $x \wedge y = x$. If x and y are in D_n, we say that x *covers* y if $x = y \vee e^{(i)} \neq y$ for some i. Finally for all x in D_n we let $\nu(x)$ be the number of 1's in x, and $\zeta(x)$ the number of 0's; thus $\nu(x) + \zeta(x) = n$.

1. [*20*] Draw a network diagram for the odd-even merge when $m = 3$ and $n = 5$.

2. [*22*] Show that V. Pratt's sorting algorithm (exercise 5.2.1–30) leads to a sorting network for n elements which has approximately $(\log_2 n)(\log_3 n)$ levels of delay. Draw the corresponding network for $n = 12$.

3. [*M20*] (K. E. Batcher.) Find a simple relation between $C(m, m-1)$ and $C(m, m)$.

▶ **4.** [*M23*] Prove that $\hat{T}(6) = 5$.

5. [*M21*] Prove that (13) is the delay time associated with the sorting network outlined in (10).

6. [*28*] Let $T(n)$ be the minimum number of stages needed to sort by making *simultaneous disjoint comparisons* (without the network constraint); each such set of comparisons can be represented as a node containing a set of pairs $i_1:j_1, i_2:j_2, \ldots, i_r:j_r$ where $i_1, j_1, i_2, j_2, \ldots, i_r, j_r$ are distinct, with 2^r branches below this node for the respective cases

$$\langle K_{i_1} < K_{j_1}, K_{i_2} < K_{j_2}, \ldots, K_{i_r} < K_{j_r}\rangle,$$
$$\langle K_{i_1} > K_{j_1}, K_{i_2} < K_{j_2}, \ldots, K_{i_r} < K_{j_r}\rangle, \qquad \text{etc.}$$

Prove that $T(5) = T(6) = 5$.

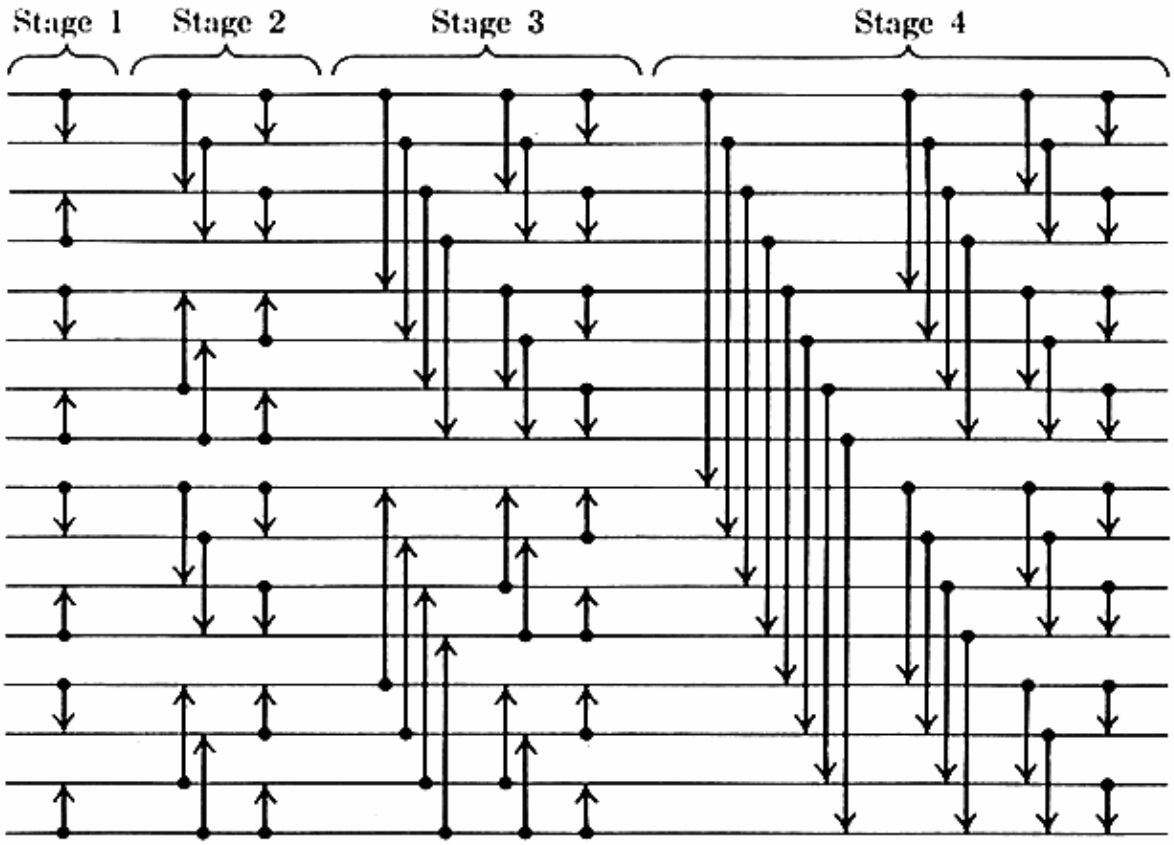


Fig. 56. A nonstandard sorting network based on bitonic sorting.

7. [*25*] Show that if the final comparator of the network for $n = 10$ in Fig. 49 is moved just ahead of the second- and third-last comparators, the network will still sort.

8. [*M20*] Prove that $\hat{M}(m_1 + m_2, n_1 + n_2) \geq \hat{M}(m_1, n_1) + \hat{M}(m_2, n_2) + \min(m_1, n_2)$, for $m_1, m_2, n_1, n_2 \geq 0$.

9. [*M25*] (R. W. Floyd.) Prove that $\hat{M}(3, 3) = 6$, $\hat{M}(4, 4) = 9$, $\hat{M}(5, 5) = 13$.

10. [*M22*] Prove that Batcher's bitonic sorter, as defined in the remarks preceding (15), is valid. [*Hint:* It is only necessary to prove that all sequences consisting of k 1's followed by l 0's followed by $n - k - l$ 1's will be sorted.]

11. [*M23*] Prove that Batcher's bitonic sorter of order 2^p will not only sort sequences $\langle z_0, z_1, \ldots, z_{2^p-1}\rangle$ for which $z_0 \geq \cdots \geq z_k \leq \cdots \leq z_{2^p-1}$, it also will sort any sequence for which $z_0 \leq \cdots \leq z_k \geq \cdots \geq z_{2^p-1}$. [As a consequence, the network in Fig. 56 will sort 16 elements, since each stage separated by dotted lines consists of bitonic sorters or reverse-order bitonic sorters, applied to sequences that have been sorted in opposite directions.]

12. [*M20*] Prove or disprove: If x and y are bitonic sequences of the same length, so are $x \vee y$ and $x \wedge y$.

▶ **13.** [*24*] (H. S. Stone.) Show that a sorting network for 2^t elements can be constructed by following the pattern illustrated for $t = 4$ in Fig. 57. Each of the t^2 steps in this scheme consists of a "perfect shuffle" of the first 2^{t-1} elements with the last 2^{t-1}, followed by simultaneous operations performed on 2^{t-1} pairs of adjacent elements. Each of the latter operations is either "0" (no operation), "+" (a standard comparator module), or "−" (a reverse comparator module). The sorting proceeds in t stages of t steps each; during the last stage all operations are "+". During stage s, for $s < t$, we do $t - s$ steps in which all operations are "0", followed by s steps in which the operations within step 8 consist alternately of 2^q "+" followed by 2^q "−", for $q = 1, 2, \ldots, s$.

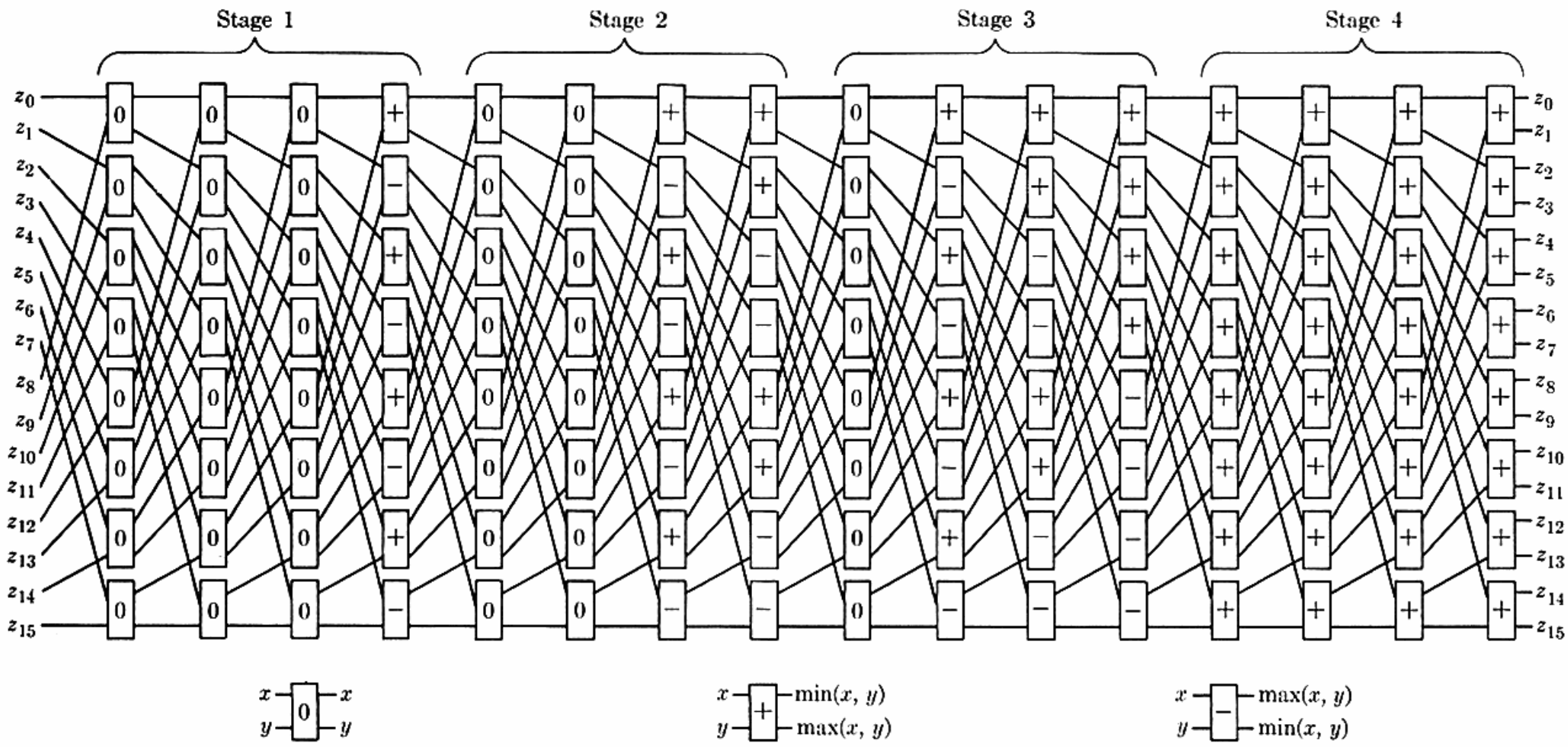


Fig. 57. Sorting 16 elements with "perfect shuffles."

[Note that this sorting scheme could be performed by a fairly simple device whose circuitry performs one "shuffle-and-operate" step and feeds the output lines back into the input. The first three steps in Fig. 57 could of course be eliminated; they have been retained only to make the pattern clear. Stone notes that the same pattern "shuffle/operate" occurs in several other algorithms, such as the fast Fourier transform (cf. Section 4.6.4).]

14. [*M20*] Prove that any $(1, n)$-merging network without multiple fanout must have at least $\lceil \log_2 (n+1) \rceil$ levels of delay.

15. [*20*] Find a nonstandard sorting network for four elements which has only five comparator modules.

16. [*M22*] Prove that the following algorithm transforms any sorting network $[i_1:j_1] \ldots [i_r:j_r]$ into a standard sorting network:

T1. Let q be the smallest index such that $i_q > j_q$. If no such index exists, stop.

T2. Change all occurrences of i_q to j_q, and all occurrences of j_q to i_q, in all comparators $[i_s:j_s]$ for $q \le s \le r$. Return to T1. ▮

For example, the network [4:1][3:2][1:3][2:4][1:2][3:4] is first transformed into [1:4][3:2][4:3][2:1][4:2][3:1], then [1:4][2:3][4:2][3:1][4:3][2:1], then [1:4][2:3][2:4][3:1][2:3][4:1], etc., until the standard network [1:4][2:3][2:4][1:3][1:2][3:4] is obtained.

17. [*M25*] Let D_{tn} be the set of all $\binom{n}{t}$ sequences $\langle x_1, \ldots, x_n \rangle$ of 0's and 1's having exactly t 1's. Show that $\hat{U}_t(n)$ is the minimum number of comparators needed in a network which sorts all the elements of D_{tn}; $\hat{V}_t(n)$ is the minimum number needed to sort $D_{tn} \cup D_{(t-1)n}$; and $\hat{W}_t(n)$ is the minimum number needed to sort $\bigcup_{0 \le k \le t} D_{kn}$.

▸ **18.** [*M20*] Prove that a network which finds the median of $2t - 1$ elements requires at least $(t-1)\lceil \log_2 (t+1) \rceil + \lceil \log_2 t \rceil$ comparator modules. [*Hint:* See the proof of Theorem A.]

19. [*M22*] Prove that $\hat{U}_2(n) = 2n - 4$ and $\hat{V}_2(n) = 2n - 3$, for all $n \ge 2$.

20. [*24*] Prove that $\hat{V}_3(5) = 7$.

21. [*M15*] Let α be any n-network, and let x and y be n-vectors. Prove that $x \le y$ implies that $x\alpha \le y\alpha$.

22. [*M15*] Prove that if x and y are n-vectors of real numbers, then $x \cdot y \le (x\alpha) \cdot (y\alpha)$. (Here $x \cdot y$ is the dot product, $x_1y_1 + \cdots + x_ny_n$.)

23. [*M17*] Let α be an n-network. Prove that there is a permutation $p \in P_n$ such that $(p\alpha)_i = j$ if and only if there are vectors x, y in D_n such that x covers y, $(x\alpha)_i = 1$, $(y\alpha)_i = 0$, and $\zeta(y) = j$.

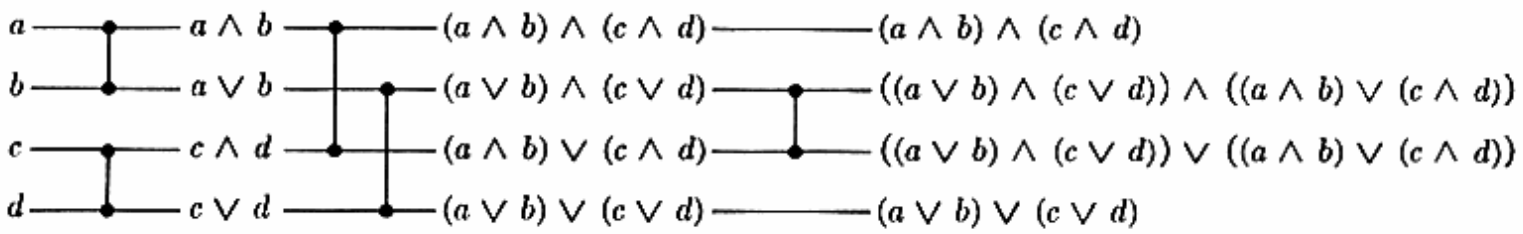
▸ **24.** [*M21*] (V. E. Alekseyev.) Let α be an n-network, and for $1 \le k \le n$ let $l_k = \min \{(p\alpha)_k \mid p \text{ in } P_n\}$, $u_k = \max \{(p\alpha)_k \mid p \text{ in } P_n\}$ denote the lower and upper bounds on the range of values which may appear in line k of the output. Let l'_k and u'_k be defined similarly for the network $\alpha' = \alpha[i:j]$. Prove that $l'_i = l_i \wedge l_j$, $l'_j \le l_i + l_j$, $u'_i \ge u_i + u_j - (n+1)$, $u'_j = u_i \vee u_j$. [*Hint:* Given vectors x and y in D_n with $(x\alpha)_i = (y\alpha)_j = 0$, $\zeta(x) = l_i$, $\zeta(y) = l_j$, find a vector z in D_n with $(z\alpha')_j = 0$, $\zeta(z) \le l_i + l_j$.]

25. [*M30*] Let l_k and u_k be as defined in exercise 24. Prove that the set $\{(p\alpha)_k \mid p \text{ in } P_n\}$ includes all integers between l_k and u_k inclusive.

26. [*M24*] (R. W. Floyd.) Let α be an n-network. Prove that the set $D_n\alpha = \{x\alpha \mid x \text{ in } D_n\}$ can be determined from the set $P_n\alpha = \{p\alpha \mid p \text{ in } P_n\}$; and conversely, $P_n\alpha$ can be determined from $D_n\alpha$.

▶**27.** [*M20*] Let x and y be vectors, and let $x\alpha$ and $y\alpha$ be sorted. Prove that $(x\alpha)_i \le (y\alpha)_j$ if and only if, for every choice of j elements from y we can choose i elements from x, such that every chosen x element is $\le$ some chosen y element. Use this principle to prove that *if we sort the rows of any matrix, then sort the columns, the rows will remain in order.*

▶**28.** [*M20*] The following diagram shows that it is possible to write down formulas for the contents of all lines in a sorting network in terms of the inputs:



Using the commutative laws $x \wedge y = y \wedge x$, $x \vee y = y \vee x$, the associative laws $x \wedge (y \wedge z) = (x \wedge y) \wedge z$, $x \vee (y \vee z) = (x \vee y) \vee z$, the distributive laws $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$, the absorption laws $x \wedge (x \vee y) = x \vee (x \wedge y) = x$, and the idempotent laws $x \wedge x = x \vee x = x$, we can reduce the formulas at the right of this network to $(a \wedge b \wedge c \wedge d)$, $(a \wedge b \wedge c) \vee (a \wedge b \wedge d) \vee (a \wedge c \wedge d) \vee (b \wedge c \wedge d)$, $(a \wedge b) \vee (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$, $a \vee b \vee c \vee d$, respectively.

Prove that, in general, the tth largest element of $\{x_1, \ldots, x_n\}$ is given by the "elementary symmetric function"

$$\sigma_t(x_1, \ldots, x_n) = \bigvee \{x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_t} \mid 1 \le i_1 < i_2 < \cdots < i_t \le n\}.$$

[There are $\binom{n}{t}$ terms being $\vee$'d together. Thus the problem of finding minimum-cost sorting networks is equivalent to the problem of computing the elementary symmetric functions with a minimum number of "and/or" circuits, where at every stage we replace two quantities ϕ and ψ by $\phi \wedge \psi$ and $\phi \vee \psi$.]

29. [*M20*] Given that $x_1 \le x_2 \le x_3$ and $y_1 \le y_2 \le y_3 \le y_4 \le y_5$, and that $z_1 \le z_2 \le \cdots \le z_8$ is the result of merging the x's with the y's, find formulas for each of the z's in terms of the x's and the y's, using the operators $\wedge$ and $\vee$.

30. [*HM24*] Prove that any formula involving $\wedge$ and $\vee$ and the independent variables $\{x_1, \ldots, x_n\}$ can be reduced using the identities in exercise 28 to a "canonical" form $\tau_1 \vee \tau_2 \vee \cdots \vee \tau_k$, where $k \ge 1$, each τ_i has the form $\bigwedge\{x_j \mid j \text{ in } S_i\}$ where S_i is a subset of $\{1, 2, \ldots, n\}$, and no set S_i is included in S_j for $i \ne j$. Prove also that two such canonical forms are equal for all $x_1, \ldots, x_n$ if and only if they are identical (up to order).

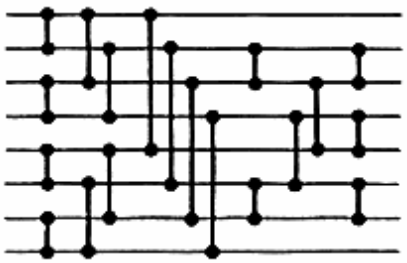
31. [*M24*] (R. Dedekind, 1897.) Let δ_n be the number of distinct canonical forms on $x_1, \ldots, x_n$ in the sense of exercise 30. Thus $\delta_1 = 1$, $\delta_2 = 4$, and $\delta_3 = 18$. What is δ_4?

32. [*M28*] (M. W. Green.) Let $G_1 = \{00, 01, 11\}$, and let G_{n+1} be the set of all strings $\theta\phi\psi\omega$ such that θ, ϕ, ψ, ω have length 2^{n-1} and $\theta\phi$, $\psi\omega$, $\theta\psi$, and $\phi\omega$ are in G_n.

Let α be the network consisting of the first four levels of the 16-sorter shown in Fig. 48. Show that $D_{16}\alpha = G_4$, and prove that it has exactly $\delta_4 + 2$ elements. (Cf. exercise 31.)

▶ **33.** [*M22*] Not all δ_n of the functions of $\langle x_1, \ldots, x_n \rangle$ in exercise 31 can appear in comparator networks. In fact, prove that the function $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4)$ cannot appear as an output of any comparator network on $\langle x_1, \ldots, x_n \rangle$.

34. [*23*] Is the following a sorting network?



35. [*20*] Prove that any *standard* sorting network must contain each of the "adjacent" comparators $[i{:}i+1]$, for $1 \le i < n$, at least once.

▶ **36.** [*22*] The network of Fig. 47 involves only *adjacent* comparisons $[i{:}i+1]$; let us call such a network *primitive*. (a) Prove that a primitive sorting network for n elements must have at least $\binom{n}{2}$ comparators. [*Hint:* Consider the inversions of a permutation.] (b) (R. W. Floyd, 1964.) Let α be a primitive network for n elements, and let x be a vector such that $(x\alpha)_i > (x\alpha)_j$ for some $i < j$. Prove that $(y\alpha)_i > (y\alpha)_j$, where y is the vector $\langle n, n-1, \ldots, 1 \rangle$. (c) As a consequence of (b), a primitive network is a sorting network if and only if it sorts the single vector $\langle n, n-1, \ldots, 1 \rangle$!

37. [*M22*] The *odd-even transposition sort* for n numbers, $n \ge 3$, is a network n levels deep with $\frac{1}{2}n(n-1)$ comparators, arranged in a brick-like pattern as shown in Fig. 58. (When n is even, there are two possibilities.) Such a sort is especially easy to implement in hardware, since only two kinds of actions are performed alternately. Prove that such a network is, in fact, a valid sorting network. [*Hint:* See exercise 36.]

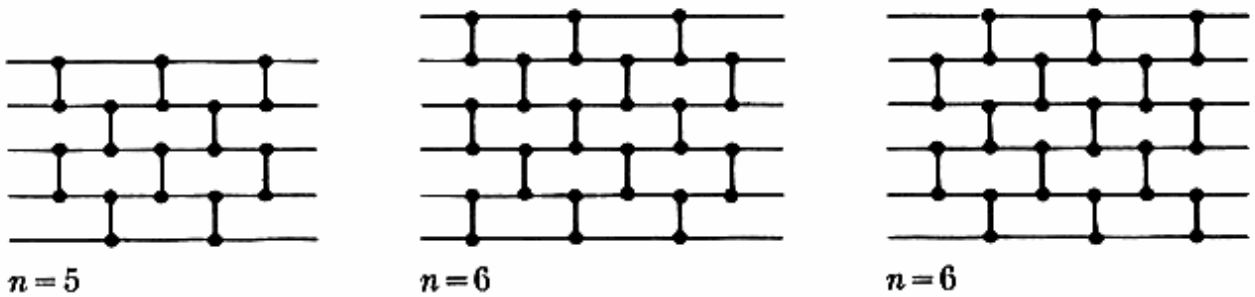


Fig. 58. The odd-even transposition sort.

38. [*29*] We may interpret sorting networks in another way, letting each line carry a multiset of m numbers, instead of a single number; under this interpretation, the operation $[i{:}j]$ replaces x_i and x_j, respectively, by $x_i \mathbin{\hat{\wedge}} x_j$ and $x_i \mathbin{\hat{\vee}} x_j$, the least m and the greatest m of the $2m$ numbers $x_i \uplus x_j$. (See Fig. 59 for an illustration of this interpretation when $m = 2$.) If a and b are multisets of m numbers each, we say that $a \ll b$ iff $a \mathbin{\hat{\wedge}} b = a$ (equivalently, $a \mathbin{\hat{\vee}} b = b$; the largest element of a is less than or equal to the smallest of b). Thus $a \mathbin{\hat{\wedge}} b \ll a \mathbin{\hat{\vee}} b$.

Let α be an n-network, and let $x = \langle x_1, \ldots, x_n \rangle$ be a vector in which each x_i is a multiset of m elements. Prove that if $(x\alpha)_i$ is not $\ll (x\alpha)_j$ in the above interpretation, there is a vector y in D_n such that $(y\alpha)_i = 1$ and $(y\alpha)_j = 0$. [Consequently, a sorting

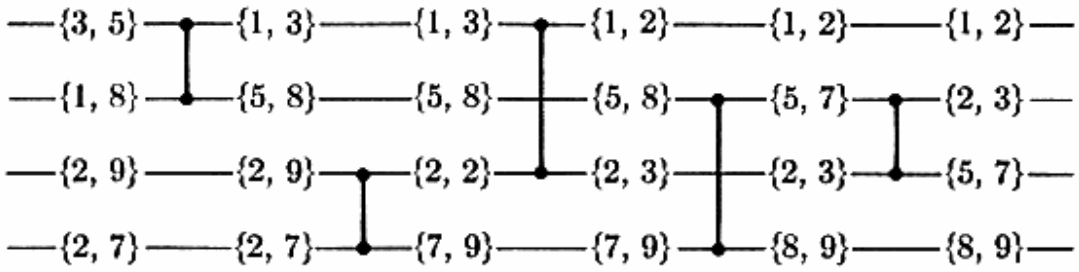


Fig. 59. Another interpretation of the sorting network in Fig. 44: each comparator module performs a merge operation.

network for n elements becomes a sorting network for mn elements if we replace comparisons by m-way merges. Figure 60 shows an 8-element sorter constructed from a 4-element sorter by using this observation.]

▶**39.** [*M23*] Show that, in the notation of exercise 38, $(x \wedge\!\!\!\wedge y) \wedge\!\!\!\wedge z = x \wedge\!\!\!\wedge (y \wedge\!\!\!\wedge z)$ and $(x \vee\!\!\!\vee y) \vee\!\!\!\vee z = x \vee\!\!\!\vee (y \vee\!\!\!\vee z)$; however $(x \vee\!\!\!\vee y) \wedge\!\!\!\wedge z$ is *not* always equal to $(x \wedge\!\!\!\wedge z) \vee\!\!\!\vee (y \wedge\!\!\!\wedge z)$, and $(x \wedge\!\!\!\wedge y) \vee\!\!\!\vee (x \wedge\!\!\!\wedge z) \vee\!\!\!\vee (y \wedge\!\!\!\wedge z)$ does *not* always equal the middle m elements of $x \uplus y \uplus z$. Find a correct formula, in terms of x, y, z and the $\wedge\!\!\!\wedge$ and $\vee\!\!\!\vee$ operations, for those middle elements.

▶**40.** [*M25*] (R. L. Graham.) The comparator $[i:j]$ is called *redundant* in the network $\alpha_1[i:j]\alpha_2$ if either $(x\alpha_1)_i \le (x\alpha_1)_j$ for all vectors x, or $(x\alpha_1)_i \ge (x\alpha_1)_j$ for all vectors x. Prove that if α is a network with r irredundant comparators, there are at least r distinct ordered pairs (i, j) of distinct indices such that $(x\alpha)_i \le (x\alpha)_j$ for all vectors x. (Consequently, a network with no redundant comparators contains at most $\binom{n}{2}$ modules.)

▶**41.** [*M27*] (V. E. Alekseyev.) Let $\alpha = [i_1:j_1] \ldots [i_r:j_r]$ be an n-network; for $1 \le s \le r$ we define $\alpha^s = [i'_1:j'_1] \ldots [i'_{s-1}:j'_{s-1}][i_s:j_s] \ldots [i_r:j_r]$, where the i'_k and j'_k are obtained from i_k and j_k by changing i_s to j_s and changing j_s to i_s wherever they appear. For example, if $\alpha = [1:2][3:4][1:3][2:4][2:3]$, then $\alpha^4 = [1:4][3:2][1:3][2:4][2:3]$.

a) Prove that $D_n\alpha = D_n(\alpha^s)$.

b) Prove that $(\alpha^s)^t = (\alpha^t)^s$.

c) A *conjugate* of α is any network of the form $(\ldots((\alpha^{s_1})^{s_2}) \ldots)^{s_k}$. Prove that α has at most 2^{r-1} conjugates.

d) Let $g_\alpha(x) = 1$ if $x \in D_n\alpha$, 0 if $x \notin D_n\alpha$; and let

$$f_\alpha(x) = (\bar{x}_{i_1} \vee x_{j_1}) \wedge \cdots \wedge (\bar{x}_{i_r} \vee x_{j_r}).$$

Prove that $g_\alpha(x) = \bigvee \{f_{\alpha'}(x) \mid \alpha' \text{ is a conjugate of } \alpha\}$.

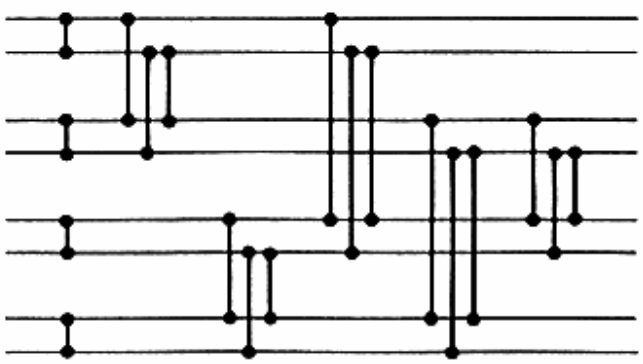


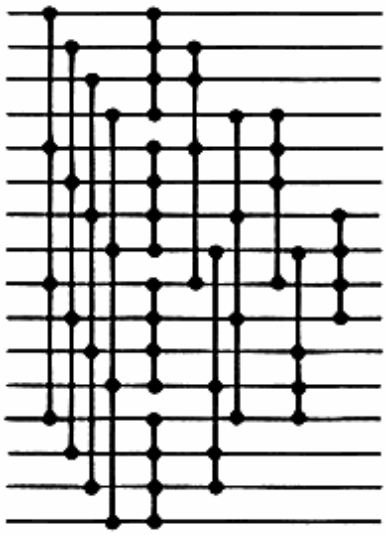
Fig. 60. An 8-sorter constructed from a 4-sorter, by using the merging interpretation.

e) Let G_α be the directed graph with vertices $\{1, \ldots, n\}$, and arcs $i_s \to j_s$ for $1 \le s \le r$. Prove that α is a sorting network if and only if $G_{\alpha'}$ has an oriented path from i to $i+1$ for $1 \le i < n$ and for all α' conjugate to α. [This condition is somewhat remarkable, since G_α does not depend on the order of the comparators in α.]

▶ **42.** [*25*] (D. Van Voorhis.) Prove that $\hat{S}(n) \ge \hat{S}(n-1) + \lceil \log_2 n \rceil$.

43. [*23*] A *permutation network* is a sequence of modules $[i_1:j_1] \ldots [i_r:j_r]$ where each module $[i:j]$ can be set by external controls to pass its inputs unchanged or to switch x_i and x_j (irrespective of the values of x_i and x_j), and such that each permutation of the inputs is achievable on the output lines by some setting of the modules. Every sorting network is clearly a permutation network, but the converse is not true: Find a permutation network for five elements which has only eight modules.

44. [*46*] Study the properties of sorting networks which are made from m-sorter modules instead of 2-sorters. (For example, G. Shapiro has constructed the network



which sorts 16 elements using fourteen 4-sorters. Is this best possible? Are there efficient ways to sort m^2 elements with m-sorter modules, for all m?)

45. [*48*] Find an (m, n) merging network with less than $C(m, n)$ comparators, or prove that none exists.

46. [*48*] Find an (m, n) merging network with less than $\lceil \log_2 (m+n) \rceil$ levels of delay, or prove that none exists.

47. [*48*] Study the class of sorting schemes that can be implemented with perfect shuffle networks as in Fig. 57, with other patterns of "0", "+", and "−" operations.

48. [*HM49*] Explore the properties of the $\wedge\!\!\!\wedge$ and $\vee\!\!\!\vee$ operations defined in exercise 38. Is it possible to characterize all of the identities in this algebra in some nice way, or to derive them all from a finite set of identities? In this regard, identities such as $x \wedge\!\!\!\wedge x \wedge\!\!\!\wedge x = x \wedge\!\!\!\wedge x$, or $x \wedge\!\!\!\wedge (x \vee\!\!\!\vee (x \wedge\!\!\!\wedge (x \vee\!\!\!\vee y))) = x \wedge\!\!\!\wedge (x \vee\!\!\!\vee y)$, which hold only for $m \le 2$, are of comparatively little interest; consider only the identities which are true for *all* m.

49. [*M49*] What is the asymptotic behavior of the function $T(n)$ defined in exercise 6? Is $T(n) < \hat{T}(n)$ for some n?

50. [*50*] Find the exact value of $\hat{S}(n)$ for some $n > 8$.

51. [*M50*] Prove that the asymptotic value of $\hat{S}(n)$ is not $O(n \log n)$.

EXERCISES—Second Set

The following exercises deal with several different types of optimality questions related to sorting. The first few problems are based on an interesting "multi-head" generalization of the bubble sort, investigated by P. N. Armstrong and R. J. Nelson as early as 1954. [See *U.S. Patents 3029413, 3034102.*] Let $1 = h_1 < h_2 < \cdots < h_m = n$ be an increasing sequence of integers; we shall call it a "head sequence" of length m and span n, and we shall use it to define a special kind of sorting method. The sorting of records $R_1 \ldots R_N$ proceeds in several passes, and each pass consists of $N + n - 1$ steps. On step j, for $j = 1 - n, 2 - n, \ldots, N - 1$, the records $R_{j+h[1]}, R_{j+h[2]}, \ldots, R_{j+h[m]}$ are examined and rearranged if necessary so that their keys are in order. (We say that $R_{j+h[1]}, \ldots, R_{j+h[m]}$ are "under the read-write heads." When $j + h[k]$ is < 1 or $> N$, record $R_{j+h[k]}$ is left out of consideration; in effect, the keys $K_0, K_{-1}, K_{-2}, \ldots$ are treated as $-\infty$ and $K_{N+1}, K_{N+2}, \ldots$ are treated as $+\infty$. Therefore step j is actually trivial when $j \le -h[m-1]$ or $j > N - h[2]$.)

For example, the following table shows one pass of a sort when $m = 3$, $N = 9$, and $h_1 = 1$, $h_2 = 2$, $h_3 = 4$:

	K_{-2}	K_{-1}	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_0	K_{10}	K_{11}	K_{12}
$j = -3$	—	—		3	1	4	5	9	2	6	8	7			
$j = -2$		—	—	3	1	4	5	9	2	6	8	7			
$j = -1$			—	3	1	4	5	9	2	6	8	7			
$j = 0$				1	3	4	5	9	2	6	8	7			
$j = 1$				1	3	4	5	9	2	6	8	7			
$j = 2$				1	3	2	4	9	5	6	8	7			
$j = 3$				1	3	2	4	6	5	9	8	7			
$j = 4$				1	3	2	4	5	6	9	8	7			
$j = 5$				1	3	2	4	5	6	7	8	9			
$j = 6$				1	3	2	4	5	6	7	8	9	—		
$j = 7$				1	3	2	4	5	6	7	8	9		—	
$j = 8$				1	3	2	4	5	6	7	8	9	—		—

Note that when $m = 2$, $h_1 = 1$, and $h_2 = 2$, this "multi-head" method reduces to the bubble sort (Algorithm 5.2.2B).

52. [*21*] (James Dugundji.) Prove that if $h[k+1] = h[k] + 1$ for some k, $1 \le k < m$, the multihead sorter defined above will eventually sort any input file in a finite number of passes. But if $h[k+1] \ge h[k] + 2$ for $1 \le k < m$, the input might *never* become sorted.

▶ **53.** [*30*] (Armstrong and Nelson.) Given that $h[k+1] \le h[k] + k$ for $1 \le k < m$, and $N \ge n - 1$, prove that the largest $n - 1$ elements always move to their final destination on the first pass. [*Hint:* Use the zero-one principle; when sorting 0's and 1's, with less than n 1's, prove that it is impossible to have all heads sensing a 1 unless all 0's lie to the left of the heads.]

Prove that sorting will be complete in at most $\lceil (N-1)/(n-1) \rceil$ passes when the heads satisfy the given conditions. Is there an input file which requires this many passes?

54. [*26*] If $n = N$, prove that the first pass can be guaranteed to place the smallest key into position R_1 if and only if $h[k+1] \le 2h[k]$ for $1 \le k < m$.

55. [*34*] (J. Hopcroft.) A "perfect sorter" for N elements is a multi-head sorter with $N = n$ which always finishes in one pass. Exercise 53 proves that the sequence $\langle h_1, h_2, h_3, h_4, \ldots, h_m\rangle = \langle 1, 2, 4, 7, \ldots, 1 + \binom{m}{2}\rangle$ gives a perfect sorter for $N = \binom{m}{2} + 1$ elements, using $m = (\sqrt{8N-7}+1)/2$ heads. For example, the head sequence $\langle 1, 2, 4, 7, 11, 16, 22\rangle$ is a perfect sorter for 22 elements.

Prove that, in fact, the head sequence $\langle 1, 2, 4, 7, 11, 16, 23\rangle$ is a perfect sorter for 23 elements.

56. [*49*] Study the largest N for which m-head perfect sorters exist, given m. Is $N = O(m^2)$?

57. [*23*] (V. Pratt.) When each head h_k is in position 2^{k-1} for $1 \le k \le m$, how many passes are necessary to sort the sequence $z_1 z_2 \ldots z_{2^m-1}$ of 0's and 1's where $z_j = 0$ iff j is a power of 2?

58. [*24*] (*Uniform sorting.*) The tree of Fig. 34 in Section 5.3.1 makes the comparison 2:3 in both branches on level 1, and on level 2 it compares 1:3 in each branch unless that comparison would be redundant. In general, we can consider the class of all sorting algorithms which are uniform in that way; assuming that the $M = \binom{N}{2}$ pairs $\{(a, b) \mid 1 \le a < b \le N\}$ have been arranged into a sequence

$$(a_1, b_1)\ (a_2, b_2), \ldots, (a_M, b_M),$$

we can successively make each of the comparisons $K_{a_1}{:}K_{b_1}$, $K_{a_2}{:}K_{b_2}, \ldots$ whose outcome is not already known. Each of the $M!$ arrangements of the (a, b) pairs defines a uniform sorting algorithm. The concept of uniform sorting is due to H. L. Beus [*JACM* **17** (1970), 482–495], whose work has suggested the next few exercises.

It is convenient to define uniform sorting formally by means of graph theory. Let G be the directed graph on the vertices $\{1, 2, \ldots, N\}$, having no arcs. For $i = 1, 2, \ldots, M$ we add arcs to G as follows:

Case 1. G contains a path from a_i to b_i. Add the arc $a_i \to b_i$ to G.

Case 2. G contains a path from b_i to a_i. Add the arc $b_i \to a_i$ to G.

Case 3. G contains no path from a_i to b_i or b_i to a_i. Compare $K_{a_i}{:}K_{b_i}$; then add the arc $a_i \to b_i$ to G if $K_{a_i} \le K_{b_i}$, the arc $b_i \to a_i$ if $K_{a_i} > K_{b_i}$.

We are concerned primarily with the number of key comparisons made by a uniform sorting algorithm, not with the mechanism by which redundant comparisons are actually avoided; the graph G need not be constructed explicitly, it is used here merely to help define the concept of uniform sorting.

We shall also consider *restricted uniform sorting*, in which only paths of length 2 are counted in cases 1, 2, and 3 above. (A restricted uniform sorting algorithm may make some redundant comparisons, but exercise 59 shows that the analysis is somewhat simpler in the restricted case.)

Prove that the restricted uniform algorithm is the same as the uniform algorithm when the sequence of pairs is taken in lexicographic order

$$(1, 2)(1, 3)(1, 4) \ldots (1, N)(2, 3)(2, 4) \ldots (2, N) \ldots (N-1, N).$$

Show in fact that both algorithms are equivalent to "quicksort" (Algorithm 5.2.2Q) when the keys are distinct and when quicksort's redundant comparisons are removed as in exercise 5.2.2–24. (Disregard the order in which the comparisons are actually made in quicksort, consider only which pairs of keys are compared.)

59. [*M38*] Given a pair sequence $(a_1, b_1) \ldots (a_M, b_M)$ as in exercise 58, let c_i be the number of pairs (j, k) such that $j < k < i$ and (a_i, b_i), (a_j, b_j), (a_k, b_k) forms a triangle. (a) Prove that the average number of comparisons made by the restricted uniform sorting algorithm is $\sum_{1 \le i \le M} 2/(c_i + 2)$. (b) Use the results of (a) and exercise 58 to determine the average number of irredundant comparisons performed by quicksort. (c) The following pair sequence is inspired by (but not equivalent to) merge sorting:

$$(1, 2)(3, 4)(5, 6) \ldots (1, 3)(1, 4)(2, 3)(2, 4)(5, 7) \ldots (1, 5)(1, 6)(1, 7)(1, 8)(2, 5) \ldots$$

Does the uniform method based on this sequence do more or less comparisons than quicksort, on the average?

60. [*M29*] In the worst case, quicksort does $\binom{N}{2}$ comparisons. Do all restricted uniform sorting algorithms (in the sense of exercise 57) perform $\binom{N}{2}$ comparisons in their worst case?

61. [*M48*] (H. L. Beus.) Does quicksort have the minimum average number of comparisons, over all (restricted) uniform sorting algorithms?

62. [*25*] The Ph.D. thesis "Electronic Data Sorting" by Howard B. Demuth (Stanford University: October, 1956) was perhaps the first paper to deal in any detail with questions of computational complexity. Demuth considered several abstract models for sorting devices, and established lower and upper bounds on the mean and maximum execution times achievable with each model. His simplest model, the "circular non-reversible memory" (Fig. 61), is the subject of this exercise.

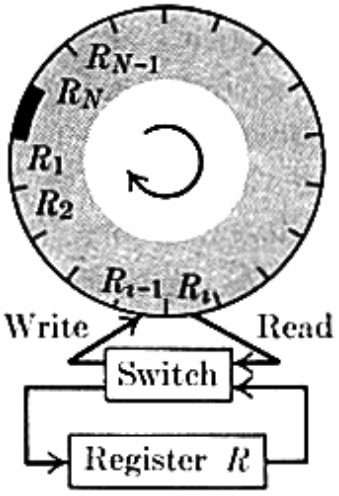


Fig. 61. A device for which the bubble-sort strategy is optimum.

Consider a machine which sorts $R_1 R_2 \ldots R_N$ in a number of passes, where each pass contains the following $N + 1$ steps:

Step 1. Set $R \leftarrow R_1$. (R is an internal machine register.)

Step i, for $1 < i \le N$.

Either (a) Set $R_{i-1} \leftarrow R$, $R \leftarrow R_i$;

or (b) Set $R_{i-1} \leftarrow R_i$, leaving R unchanged.

Step N + 1. Set $R_N \leftarrow R$.

The problem is to find a way to choose between alternatives (a) and (b) each time, in order to minimize the number of passes required to sort.

Prove that the "bubble sort" technique is optimum for this model. In other words, show that the strategy which selects alternative (a) whenever $R \le R_i$ and alternative (b) whenever $R > R_i$ will achieve the minimum number of passes.

5.4. EXTERNAL SORTING

Now it is time for us to study the interesting problems which arise when the number of records to be sorted is larger than our computer can hold in its high-speed internal memory. External sorting is quite different from internal sorting, even though the problem in both cases is to sort a given file into nondecreasing order, since efficient storage accessing on external files is rather severely limited. The data structures must be arranged so that comparatively slow peripheral memory devices (tapes, disks, drums, etc.) can quickly cope with the requirements of the sorting algorithm. Consequently most of the internal sorting techniques we have studied (insertion, exchange, selection) are virtually useless for external sorting, and it is necessary to reconsider the whole question.

Suppose, for example, that we have a file of 5000 records $R_1R_2 \ldots R_{5000}$ to sort, and that each record R_i is 20 words long (although the keys K_i are not necessarily this long). If only 1000 of these records will fit in the internal memory of a particular computer at one time, what shall we do?

One fairly obvious solution is to start by sorting each of the five subfiles $R_1 \ldots R_{1000}$, $R_{1001} \ldots R_{2000}$, $\ldots$, $R_{4001} \ldots R_{5000}$, independently, then to merge the resulting subfiles together. Fortunately the process of merging uses only very simple data structures, namely linear lists which are traversed in a sequential manner as stacks or as queues; hence merging can be done without difficulty on the least expensive external memory devices.

The process just described—internal sorting followed by "external merging"—is very commonly used, and we shall devote most of our study of external sorting to variations on this theme.

The ascending sequences of records, which are produced by the initial internal sorting phase, are often called *strings* in the published literature about sorting; this terminology is fairly widespread, but it unfortunately conflicts with even more widespread usage in other branches of computer science, where "strings" are *arbitrary* sequences of symbols. Our study of permutations has already given us a perfectly good name for the sorted segments of a file, which are conventionally called ascending runs or simply *runs*. Therefore we shall consistently use the word "runs" to describe sorted portions of a file. In this way it is possible to distinguish between "strings of runs" and "runs of strings" without ambiguity. (Of course, "runs of a program" means something else again; we can't have everything.)

Let us first consider the process of external sorting when *magnetic tapes* are used for auxiliary storage. Perhaps the simplest and most appealing way to merge with tapes is the *balanced 2-way merge*, following the central idea which was used in Algorithms 5.2.4N, S, and L. We use four "working tapes" in this process. During the first phase, ascending runs produced by internal sorting are placed alternately on Tapes 1 and 2, until the input is exhausted. Then Tapes 1 and 2 are rewound to their beginnings, and we merge the runs from these tapes, obtaining new runs which are twice as long as the original ones; the new runs are written alternately on Tapes 3 and 4 as they are being formed.

(If Tape 1 contains one more run than Tape 2, an extra "dummy" run of length 0 is assumed to be present on Tape 2.) Then all tapes are rewound, and the contents of Tapes 3 and 4 are merged into quadruple-length runs recorded alternately on Tapes 1 and 2. The process continues, doubling the length of runs each time, until only one run is left (namely the entire sorted file). If S runs were produced during the internal sorting phase, and if $2^{k-1} < S \leq 2^k$, this balanced 2-way merge procedure makes exactly $k = \lceil \log_2 S \rceil$ merging passes over all the data.

For example, in the above situation where 5000 records are to be sorted with an internal memory capacity of 1000, we have $S = 5$. The initial distribution phase of the sorting process places five runs on tape as follows:

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{1000};\ R_{2001} \ldots R_{3000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 2} & R_{1001} \ldots R_{2000};\ R_{3001} \ldots R_{4000}. \\ \text{Tape 3} & \text{(empty)} \\ \text{Tape 4} & \text{(empty)} \end{array} \tag{1}$$

The first pass of merging then produces longer runs on Tapes 3 and 4, as it reads Tapes 1 and 2, as follows:

$$\begin{array}{ll} \text{Tape 3} & R_1 \ldots R_{2000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 4} & R_{2001} \ldots R_{4000}. \end{array} \tag{2}$$

(A dummy run has implicitly been added at the end of Tape 2, so that the last run $R_{4001} \ldots R_{5000}$ on Tape 1 is merely copied onto Tape 3.) After all tapes are rewound, the next pass over the data produces

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{4000}. \\ \text{Tape 2} & R_{4001} \ldots R_{5000}. \end{array} \tag{3}$$

(Again that run $R_{4001} \ldots R_{5000}$ was simply copied; but if we had started with 8000 records, Tape 2 would have contained $R_{4001} \ldots R_{8000}$ at this point.) Finally, after another spell of rewinding, $R_1 \ldots R_{5000}$ is produced on Tape 3, and the sorting is complete.

Balanced merging can easily be generalized to the case of T tapes, for any $T \geq 3$. Choose any number P with $1 \leq P < T$, and divide the T tapes into two "banks," with P tapes on the left bank and $T - P$ on the right. Distribute the initial runs as evenly as possible onto the P tapes in the left bank; then do a P-way merge from the left to the right, followed by a $(T - P)$-way merge from the right to the left, etc., until sorting is complete. The best choice of P usually turns out to be $\lceil T/2 \rceil$ (see exercises 3, 4).

Balanced two-way merging is the special case $T = 4$, $P = 2$. Let us reconsider the example above using more tapes, taking $T = 6$ and $P = 3$.

The initial distribution now gives us

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{1000};\ R_{3001} \ldots R_{4000}. \\ \text{Tape 2} & R_{1001} \ldots R_{2000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 3} & R_{2001} \ldots R_{3000}. \end{array} \tag{4}$$

And the first merging pass produces

$$\begin{array}{ll} \text{Tape 4} & R_1 \ldots R_{3000}. \\ \text{Tape 5} & R_{3001} \ldots R_{5000}. \\ \text{Tape 6} & \text{(Empty)} \end{array} \tag{5}$$

(A dummy run has been assumed on Tape 3.) The second merging pass completes the job, placing $R_1 \ldots R_{5000}$ on Tape 1. In this special case $T = 6$ is essentially the same as $T = 5$, since the sixth tape is used only when $S \geq 7$.

Three-way merging actually requires somewhat more computer processing time than two-way merging, but this is generally negligible compared to the time needed to read, write, and rewind the tapes; we can get a fairly good estimate of the running time by considering only the amount of tape motion. The example in (4) and (5) required only two passes over the data, compared to three passes when $T = 4$, so the merging takes only about two-thirds as long when $T = 6$.

Balanced merging is quite simple, but if we look more closely, we find immediately that it isn't the *best* way to handle the particular cases treated above! Instead of going from (1) to (2) and rewinding all of the tapes, we should have stopped the first merging pass after Tapes 3 and 4 contained $R_1 \ldots R_{2000}$ and $R_{2001} \ldots R_{4000}$, respectively, with Tape 1 poised ready to read $R_{4001} \ldots R_{5000}$. Then Tapes 2, 3, 4 could be rewound and we could complete the sort by doing a three-way merge onto Tape 2. The total number of records read from tape during this procedure would be only $4000 + 5000 = 9000$, compared to $5000 + 5000 + 5000 = 15{,}000$ in the balanced scheme. A smart computer would be able to figure this out!

Indeed, when we have five runs and four tapes we can do even better by distributing them as follows:

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{1000};\ R_{3001} \ldots R_{4000}. \\ \text{Tape 2} & R_{1001} \ldots R_{2000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 3} & R_{2001} \ldots R_{3000}. \\ \text{Tape 4} & \text{(Empty)} \end{array}$$

Then a three-way merge to Tape 4, followed by a rewind of Tapes 3 and 4, followed by a three-way merge to Tape 3, would complete the sort with only $3000 + 5000 = 8000$ records read.

And, of course, if we had six tapes we could put the initial runs on Tapes 1 through 5 and complete the sort in one pass by doing a five-way merge to

Tape 6. These considerations indicate that simple balanced merging isn't the best, and it is interesting to look for improved merging patterns.

Subsequent portions of this chapter investigate external sorting more deeply. In Section 5.4.1, we consider the "internal sorting" phase which produces the initial runs; of particular interest is the technique of "replacement selection," which takes advantage of the order present in most data to produce long initial runs that actually exceed the internal memory capacity by a significant amount. Section 5.4.1 also discusses a suitable data structure for multiway merging.

The most important merging patterns are discussed in Sections 5.4.2 through 5.4.5. It is convenient to have a rather naive conception of tape sorting as we learn the characteristics of these patterns, before we come to grips with the harsh realities of real tape drives and real data that is to be sorted. For example, we may blithely assume (as we did above) that the original input records appear magically during the initial distribution phase; in fact, these input records probably will occupy one of our tapes, and they may even fill several tape reels since tapes aren't of infinite length! It is best to ignore such mundane considerations until after an academic understanding of the classical merging patterns has been gained. Then Section 5.4.6 brings the discussion down to earth by discussing real-life constraints which strongly influence the choice of a merging pattern. Section 5.4.6 compares the basic merging patterns of Sections 5.4.2 through 5.4.5, using a variety of assumptions which arise in practice.

Some other approaches to external sorting, *not* based on merging, are discussed in Sections 5.4.7 and 5.4.8. Finally Section 5.4.9 completes our survey of external sorting by treating the important problem of sorting on bulk memories such as disks and drums.

EXERCISES

1. [*15*] The text suggests internal sorting first, followed by external merging. Why don't we do away with the internal sorting phase, simply merging the records into longer and longer runs right from the start?

2. [*10*] What will the sequence of tape contents be, analogous to (1) through (3), when the example records $R_1\ R_2 \ldots R_{5000}$ are sorted using a 3-tape balanced method with $P = 2$? Compare this to the 4-tape merge; how many passes are made over all the data, after the initial distribution of runs?

3. [*20*] Show that the balanced $(P, T - P)$-way merge applied to S initial runs takes $2k$ passes, when $P^k(T - P)^{k-1} < S \leq P^k(T - P)^k$; and it takes $2k + 1$ passes, when $P^k(T - P)^k < S \leq P^{k+1}(T - P)^k$.

   Give simple formulas for (a) the exact number of passes, as a function of S, when $T = 2P$; and (b) the approximate number of passes, as $S \to \infty$, for general P and T.

4. [*HM15*] What value of P, for $1 \leq P < T$, makes $P(T - P)$ a maximum?

5.4.1. Multiway Merging and Replacement Selection

In Section 5.2.4, we studied internal sorting methods based on two-way merging, the process of combining two ordered sequences into a single ordered sequence. It is not difficult to extend this to the notion of P-way merging, where P input runs are combined into a single output run.

Let's assume that we have been given P ascending runs, i.e., sequences of records whose keys are in nondecreasing order. The obvious way to merge them is to look at the first record of each run and to select the record whose key is smallest; this record is transferred to the output and removed from the input, and the process is repeated. At any given time we need to look at only P keys (one from each input run) and select the smallest. If two or more keys are smallest, an arbitrary one is selected.

When P isn't too large, it is convenient to make this selection by simply doing $P - 1$ comparisons to find the smallest of the current keys. But when P is, say, 8 or more, we can save work by using a *selection tree* as described in Section 5.2.3; then only about $\log_2 P$ comparisons are needed each time, once the tree has been set up.

Consider, for example, the case of four-way merging, with a two-level selection tree:

Step 1.
$$087\begin{cases}087\begin{cases}087\ 503\ \infty\\170\ 908\ \infty\end{cases}\\154\begin{cases}154\ 426\ 653\ \infty\\612\ \infty\end{cases}\end{cases}$$

Step 2.
$$087\ 154\begin{cases}170\begin{cases}503\ \infty\\170\ 908\ \infty\end{cases}\\154\begin{cases}154\ 426\ 653\ \infty\\612\ \infty\end{cases}\end{cases}$$

Step 3.
$$087\ 154\ 170\begin{cases}170\begin{cases}503\ \infty\\170\ 908\ \infty\end{cases}\\426\begin{cases}426\ 653\ \infty\\612\ \infty\end{cases}\end{cases}$$

$\vdots$

Step 9. $087\ 154\ 170\ 426\ 503\ 612\ 653\ 908\ \infty\ \begin{cases}\infty\ \begin{cases}\infty\\\infty\end{cases}\\\infty\ \begin{cases}\infty\\\infty\end{cases}\end{cases}$

An additional key "∞" has been placed at the end of each run in this example, so that the merging terminates gracefully. Since external merging generally deals with very long runs, the addition of records with ∞ keys does not add

substantially to the length of the data or to the amount of work involved in merging, and such "sentinel" records frequently serve as a useful way to delimit the runs on a file.

Each step after the first in the above process consists of replacing the smallest element by the succeeding element in its run, and changing the corresponding path in the selection tree. Thus the three positions of the tree which contain 087 in Step 1 are changed in Step 2; the three positions containing 154 in Step 2 are changed in Step 3; and so on. The process of replacing one key by another in the selection tree is called *replacement selection.*

We can look at this four-way merge in several ways. From one standpoint it is equivalent to three two-way merges performed concurrently as coroutines; each node in the selection tree represents one of the sequences involved in concurrent merging processes. The selection tree is also essentially operating as a priority queue, with a "smallest in, first out" discipline.

As in Section 5.2.3 we could use a heap instead of a selection tree, to implement the priority queue. (The heap would, of course, be arranged so that the *smallest* element appears at the top, instead of the largest, reversing the order of Eq. 5.2.3–3.) Since a heap does not have a fixed size, we could therefore avoid the use of "∞" keys; merging would be complete when the heap becomes empty. On the other hand, external sorting applications usually deal with comparatively long records and keys, so that the heap is filled with pointers to keys instead of the keys themselves; we shall see below that selection trees can be represented by pointers in such a convenient manner that they are probably superior to heaps in this situation.

A tree of "losers." Figure 62 shows the complete binary tree with 12 external (square) nodes and 11 internal (circular) nodes; the external nodes have been

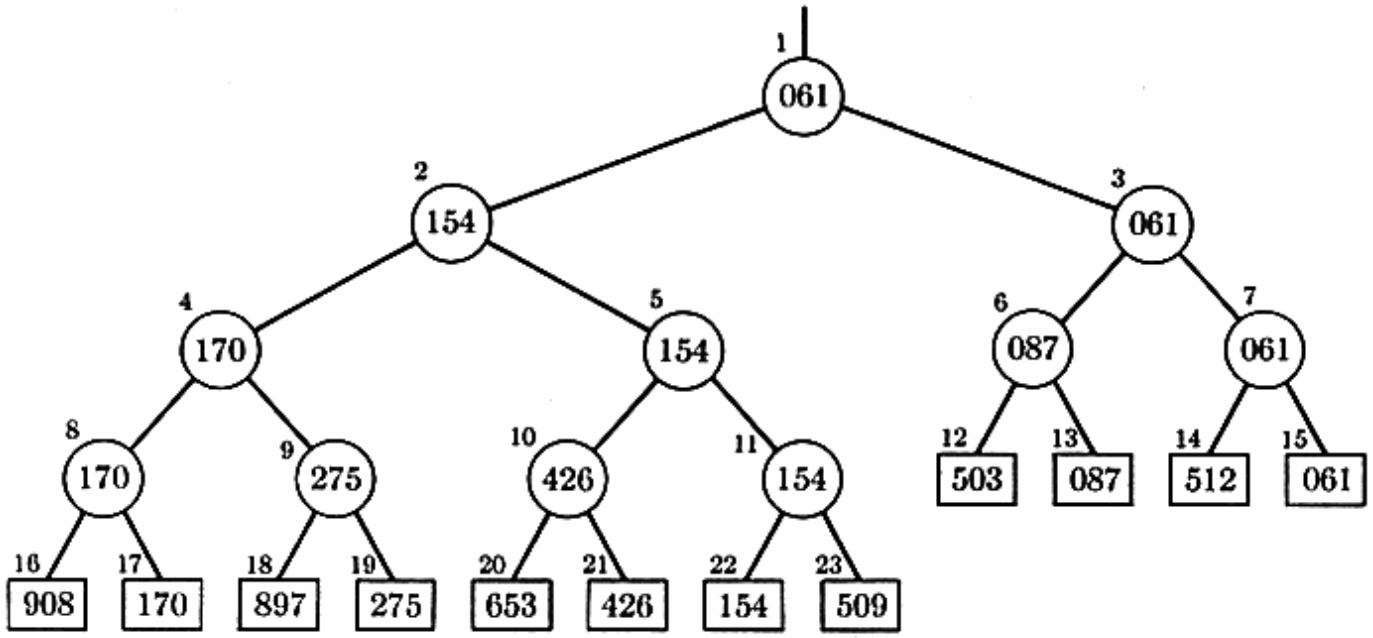


Fig. 62. A tournament to select the smallest key, using a complete binary tree whose nodes are numbered from 1 to 23.

filled with keys, and the internal nodes have been filled with the "winners" if the tree is regarded as a tournament to select the smallest key. The smaller numbers above each node show the traditional way to allocate consecutive storage positions for complete binary trees.

When the smallest key, 061, is to be replaced by another key in the selection tree of Fig. 62, we will have to look at the keys 512, 087, and 154, and no other existing keys, in order to determine the new state of the selection tree. Considering the tree as a tournament, these three keys are the losers in the matches played by 061. This suggests that what we really ought to store in the internal nodes of the tree is the *loser* of each match, instead of the winner; then the information required for updating the tree is readily available.

Figure 63 shows the same tree as Fig. 62, but with the losers represented instead of the winners. An extra node number 0 has been appended at the top of the tree, to indicate the champion of the tournament. Note that each key except the champion is a loser exactly once (cf. Section 5.3.3), so each key appears once in an external node and once in an internal node.

In practice, the external nodes at the bottom of Fig. 63 will represent fairly long records stored in computer memory, and the internal nodes will represent pointers to those records. Note that P-way merging calls for exactly P external nodes and P internal nodes, each in adjacent groups, so that several efficient methods of storage allocation suggest themselves. It is not difficult to see how to use a "loser-oriented" tree for replacement selection; we shall discuss the algorithm in some detail later in this section.

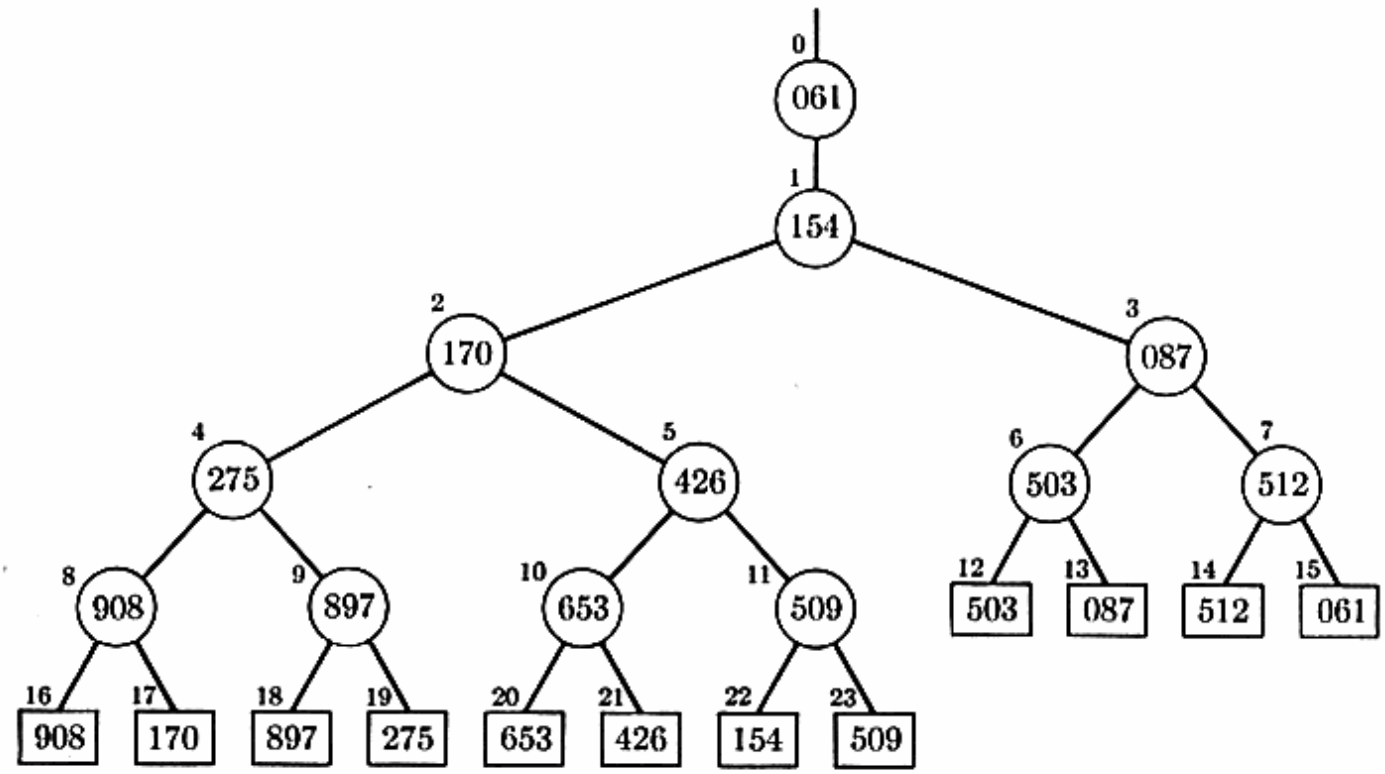


Fig. 63. The same tournament as Fig. 62, but showing the losers instead of the winners; the champion appears at the very top.

Initial runs by replacement selection. The technique of replacement selection can be used also in the first phase of external sorting, if we essentially do a P-way merge of the input data with itself! In this case we take P to be fairly large, so that the internal memory is essentially filled. When a record is output, it is replaced by the next record from the input. If the new record has a smaller key than the one just output, we cannot include it in the current run; but otherwise we can enter it into the selection tree in the usual way and it will form part of the run currently being produced. Thus the runs can contain more than P records each, even though we never have more than P in the selection tree at any time. Table 1 illustrates this process for $P = 4$; parenthesized numbers are waiting for inclusion in the following run.

Table 1

EXAMPLE OF FOUR-WAY REPLACEMENT SELECTION

Memory contents				Output
503	087	512	061	061
503	087	512	908	087
503	170	512	908	170
503	897	512	908	503
(275)	897	512	908	512
(275)	897	653	908	653
(275)	897	(426)	908	897
(275)	(154)	(426)	908	908
(275)	(154)	(426)	(509)	(end run)
275	154	426	509	154
275	612	426	509	275
		etc.		

This important method of forming initial runs was first described by Harold H. Seward [Master's Thesis, Digital Computer Laboratory Report R-232 (Mass. Inst. of Technology, 1954), 29–30], who gave reason to believe that the runs would contain more than $1.5P$ records when applied to random data. A. I. Dumey had also suggested the idea about 1950 in connection with a special sorting device planned by Engineering Research Associates, but he did not publish it. The name "replacement selecting" was coined by E. H. Friend [*JACM* **3** (1956), 154], who remarked that "the expected length of the sequences produced eludes formulation but experiment suggests that $2P$ is a reasonable expectation."

A clever way to show that $2P$ is indeed the expected run length was discovered by E. F. Moore, who compared the situation to a snowplow on a circular track [*U.S. Patent 2983904* (1961), cols. 3–4]. Consider the situation shown in Fig. 64; flakes of snow are falling uniformly on a circular road, and a lone snowplow is continually clearing the snow. Once the snow has been plowed

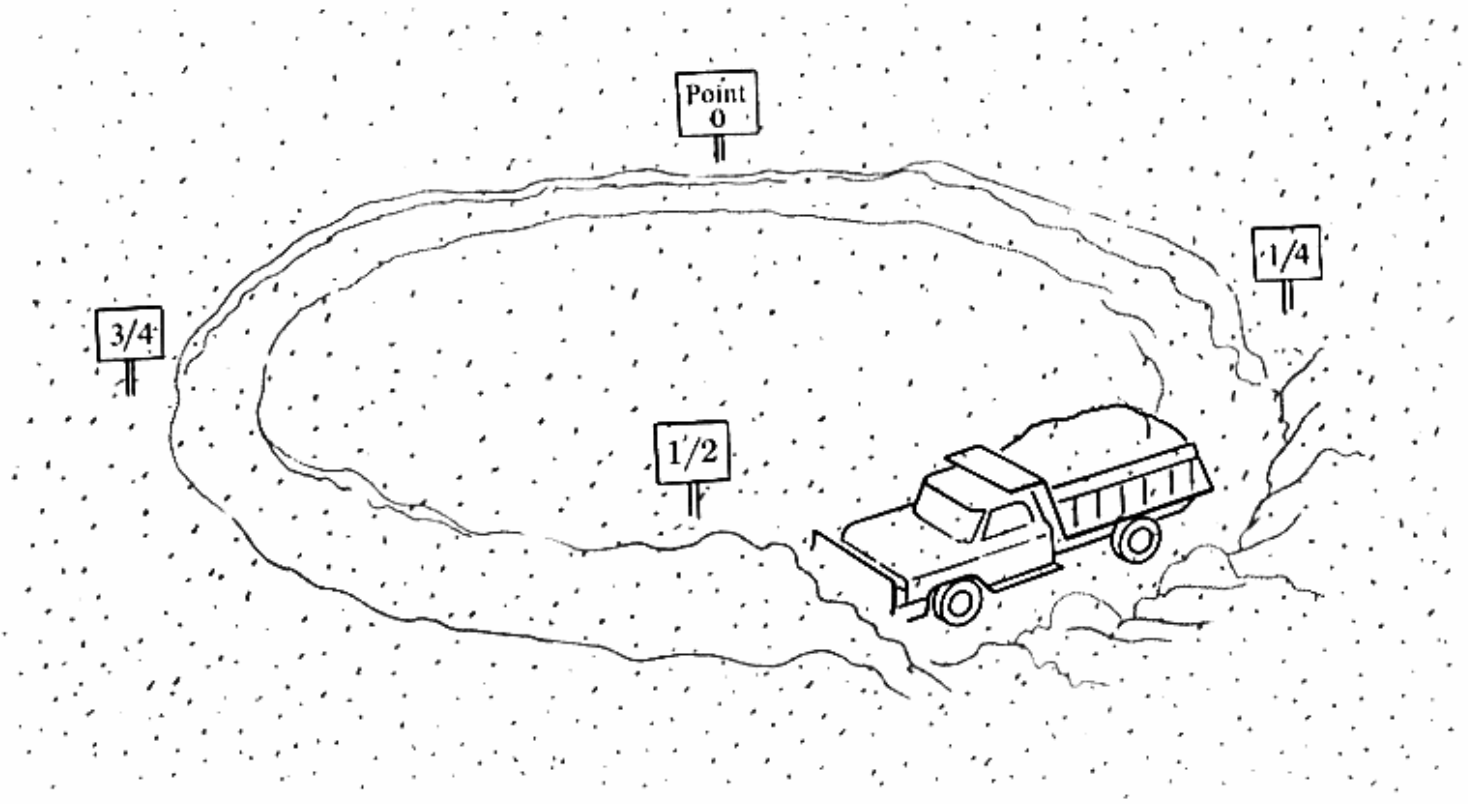


Fig. 64. The perpetual plow on its ceaseless cycle.

off the road, it disappears from the system. Points on the road may be designated by real numbers x, $0 \leq x < 1$; a flake of snow falling at position x represents an input record whose key is x, and the snowplow represents the output of replacement selection. The ground speed of the snowplow is inversely proportional to the height of snow it encounters, and the situation is perfectly balanced so that the total amount of snow on the road at all times is exactly P. A new run is formed in the output whenever the plow passes point 0.

After this system has been in operation for awhile, it is intuitively clear that it will approach a stable situation in which the snowplow runs at constant speed (because of the circular symmetry of the track). This means that the snow is at constant height when it meets the plow, and the height drops off linearly in front of the plow as shown in Fig. 65. It follows that the volume of snow removed in one revolution (namely the run length) is twice the amount present at any one time (namely P).

In many commercial applications the input data is *not* completely random, it already has a certain amount of existing order; therefore the runs produced

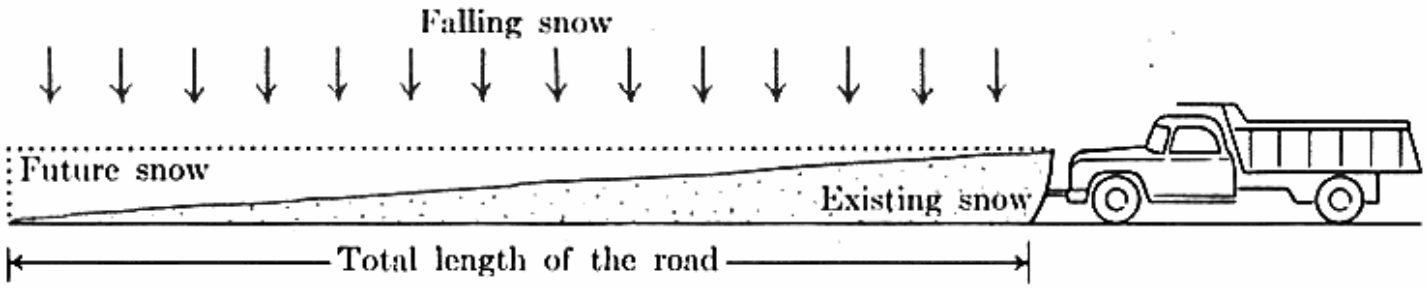


Fig. 65. Cross-section, showing the varying height of snow in front of the plow when the system is in its "steady state."

by replacement selection will tend to contain even more than $2P$ records. We shall see that the time required for external merge sorting is largely governed by the number of runs produced by the initial distribution phase, so that replacement selection becomes especially desirable; other types of internal sorting would produce about twice as many initial runs because of the limitations on memory size.

Let us now consider the process of creating initial runs by replacement selection in detail. The following algorithm is due to John R. Walters, James Painter, and Martin Zalk, who used it in a merge-sort program for the Philco 2000 in 1958. It incorporates a rather nice way to initialize the selection tree and to distinguish records belonging to different runs, as well as to flush out the last run, with comparatively simple and uniform logic. (The proper handling of the last run produced by replacement selection turns out to be a bit tricky, and it has tended to be a stumbling block for programmers.) The principal idea is to consider each key as a pair (S, K), where K is the original key and S is the run number to which this record belongs. When such extended keys are lexicographically ordered, with S as major key and K as minor key, we obtain the output sequence produced by replacement selection.

The algorithm below uses a data structure containing P nodes to represent the selection tree; the jth node $X[j]$ is assumed to contain c words beginning in $\texttt{LOC}(X[j]) = L_0 + cj$, for $0 \le j < P$, and it represents both internal node number j and external node number $P + j$ in Fig. 63. There are several named fields in each node:

KEY = the key stored in this external node;
RECORD = the record stored in this external node (including KEY as a subfield);
LOSER = pointer to the "loser" stored in this internal node;
RN = run number of the record pointed to by LOSER;
FE = pointer to internal node above this external node in the tree;
FI = pointer to internal node above this internal node in the tree.

For example, when $P = 12$, internal node number 5 and external node number 17 of Fig. 63 would both be represented in $X[5]$, by the fields KEY $= 170$, LOSER $= L_0 + 9c$ (the address of external node number 21), FE $= L_0 + 8c$, FI $= L_0 + 2c$.

The FE and FI fields have constant values, so they need not appear explicitly in memory; however, the initial phase of external sorting sometimes has trouble keeping up with the I/O devices, and it is often worth while to store these redundant values with the data instead of recomputing them each time.

Algorithm R (*Replacement selection.*) This algorithm reads records sequentially from an input file and writes them sequentially onto an output file, producing RMAX runs whose length is P or more (except for the final run). There are $P \ge 2$ nodes, $X[0], \ldots, X[P-1]$, having fields as described above.

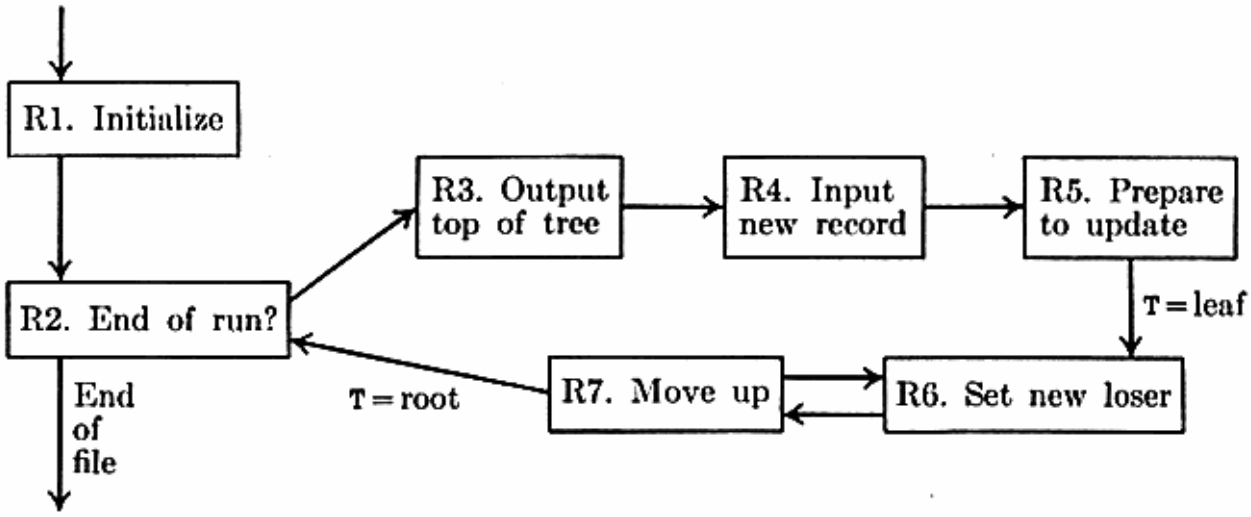


Fig. 66. Making initial runs by replacement selection.

R1. [Initialize.] Set RMAX $\leftarrow 0$, RC $\leftarrow 0$, LASTKEY $\leftarrow \infty$, Q $\leftarrow$ LOC($X[0]$), and RQ $\leftarrow 0$. (RC is the number of the current run and LASTKEY is the key of the last record output. The initial setting of LASTKEY should be larger than any possible key; cf. exercise 8.) For $0 \le j < P$, set the initial contents of $X[j]$ as follows, when J = LOC($X[j]$):

$$\texttt{LOSER(J)} \leftarrow \texttt{J}; \qquad \texttt{RN(J)} \leftarrow 0;$$
$$\texttt{FE(J)} \leftarrow \texttt{LOC}(X[\lfloor (P+j)/2 \rfloor]); \qquad \texttt{FI(J)} \leftarrow \texttt{LOC}(X[\lfloor j/2 \rfloor]).$$

(The settings of LOSER(J) and RN(J) are artificial ways to get the tree initialized by considering a fictitious run number 0 which is never output. This is tricky; see exercise 10.)

R2. [End of run?] If RQ = RC, go on to step R3. (Otherwise RQ = RC + 1 and we have just completed run number RC; any special actions required by a merging pattern for subsequent passes of the sort would be done at this point.) If RQ > RMAX, stop; otherwise set RC $\leftarrow$ RQ.

R3. [Output top of tree.] (Now Q points to the "champion," and RQ is its run number.) If RQ $\neq 0$, output RECORD(Q) and set LASTKEY $\leftarrow$ KEY(Q).

R4. [Input new record.] If the input file is exhausted, set RQ $\leftarrow$ RMAX+1 and go on to step R5. Otherwise set RECORD(Q) to the next record from the input file. If KEY(Q) < LASTKEY (so that this new record does not belong to the current run), set RQ $\leftarrow$ RQ + 1 and then if RQ > RMAX set RMAX $\leftarrow$ RQ.

R5. [Prepare to update.] (Now Q points to a new record, whose run number is RQ.) Set T $\leftarrow$ FE(Q). (T is a pointer variable which will move up the tree.)

R6. [Set new loser.] If RN(T) < RQ or if RN(T) = RQ and KEY(LOSER(T)) < KEY(Q), then interchange LOSER(T) $\leftrightarrow$ Q, RN(T) $\leftrightarrow$ RQ. (Variables Q and RQ keep track of the current winner and its run number.)

R7. [Move up.] If T = LOC($X[1]$) then go back to R2, otherwise set T $\leftarrow$ FI(T) and return to R6. ∎

Algorithm R speaks of input and output of records one at a time, while in practice it is best to read and write relatively large blocks of records. Therefore some input and output buffers are actually present in memory, behind the scenes, effectively lowering the size of P. We shall illustrate this in Section 5.4.6.

E. H. Friend [*JACM* **3** (1956), 154] suggested extending replacement selection so that whenever an input key is less than LASTKEY (so that it doesn't fit in the current run), but greater than or equal to the last key actually written onto tape (so that it actually could fit into the current run after all), it should be inserted into the output buffer. Furthermore, some computers have "scatter read" and "gather write" facilities for doing I/O to and from nonconsecutive locations, and this leads to some techniques which overlap buffers with the selection tree.

***Delayed reconstitution of runs.** A very interesting way to improve on replacement selection has been suggested by R. J. Dinsmore [*CACM* **8** (1965), 48] using a concept which we shall call *degrees of freedom*. As we have seen, each block of records on tape within a run is in nondecreasing order, so that its first element is the lowest and its last element is the highest. In the ordinary process of replacement selection, the lowest element of each block within a run is never less than the highest element of the preceding block in that run; this is "1 degree of freedom." Dinsmore suggests relaxing this condition to "m degrees of freedom," where the lowest element of each block may be less than the highest element of the preceding block so long as it is *not less than the highest elements in m different preceding blocks of the same run*. Records within individual blocks are ordered, as before, but adjacent blocks need not be in order.

For example, suppose that there are just two records per block; the following sequence of blocks is a run with three degrees of freedom:

$$| \ 08 \ 50 \ | \ 06 \ 90 \ | \ 17 \ 27 \ | \ 42 \ 67 \ | \ 51 \ 89 \ | \qquad (1)$$

A subsequent block which is to be part of the same run must begin with an element not less than the third largest element of {50, 90, 27, 67, 89}, namely 67. The sequence (1) would not be a run if there were only two degrees of freedom, since 17 is less than both 50 and 90.

A run with m degrees of freedom can be "reconstituted" while it is being read during the next phase of sorting, so that for all practical purposes it is a run in the ordinary sense. We start by reading the first m blocks into m buffers, and doing an m-way merge on them; when one buffer is exhausted, we replace it with the $(m + 1)$st block, and so on. In this way we can recover the run as a single sequence, for the first word of every newly-read block must be greater than or equal to the last word of the just-exhausted block (lest it be less than the highest elements in m different blocks that precede it). This method of reconstituting the run is essentially like an m-way merge using a single tape unit for all the input blocks! The reconstitution procedure acts as a coroutine which is called upon to deliver one record of the run at a time. We could be reconstituting different runs from different tape units with different degrees of

freedom, and merging the resulting runs, all at the same time, in essentially the same way as the four-way merge illustrated at the beginning of this section may be thought of as several two-way merges going on at once.

This ingenious idea has not yet been thoroughly analyzed. Preliminary indications are that when P is large compared to the block size the run length is about $2.1P$ when $m = 2$, $2.3P$ when $m = 4$, $2.5P$ when $m = 8$. This increase may not be enough to justify the added complication; on the other hand, it may be advantageous when there is room for a rather large number of buffers during the second phase of sorting.

***Natural selection.** Another way to increase the run lengths produced by replacement selection has been explored by W. D. Frazer and C. K. Wong. Their idea is to proceed as in Algorithm R, but when `KEY(Q)` < `LASTKEY` in step R4, the new `RECORD(Q)` is not left in the tree, it is output into an external *reservoir* instead, and another new record is read in. This process continues until the reservoir is filled with a certain number of records, P'; at this time the remainder of the current run is output from the tree, and the reservoir items are used as input for the next run.

This method tends to produce longer runs than replacement selection because it avoids "dead" records from the next string, instead of letting them fill up the tree; but it requires extra time for input and output to and from the reservoir. When $P' > P$ it is possible that some records will be placed into the reservoir twice, but when $P' \leq P$ this will never happen.

Frazer and Wong made extensive empirical tests of their method, noticing that when P is reasonably large (say $P \geq 32$) and $P' = P$ the average run length for random data is given by eP, where $e \approx 2.718$ is the base of natural logarithms. This phenomenon, and the fact that the method is an evolutionary improvement over simple replacement selection, naturally led them to call their method *natural selection*.

The "natural" law for run lengths can be proved by considering the snowplow of Fig. 64 again, and applying elementary calculus. Let L be the length of the track, and let $x(t)$ be the position of the snowplow at time t, for $0 \leq t \leq T$. The reservoir is assumed to be full at time T, when the snow stops temporarily while the plow returns to its starting position (clearing the P units of snow remaining in its path). The situation is the same as before except that the "balance condition" is different; instead of P units of snow on the road at all times, we have P units of snow in front of the plow, and the reservoir (behind the plow) gets up to P units. The snowplow advances by dx during a time interval dt if $h(x, t)\, dx$ records are output, where $h(x, t)$ is the height of the snow at time t and position $x = x(t)$, measured in suitable units; hence $h(x, t) = h(x, 0) + Kt$ for all x. Since the number of records in memory stays constant, $h(x, t)\, dx$ is also the number of records that are input *ahead* of the plow, namely $K\, dt(L - x)$, where K is the rate of snowfall (see Fig. 67). Thus

$$\frac{dx}{dt} = \frac{K(L - x)}{h(x, t)}. \tag{2}$$

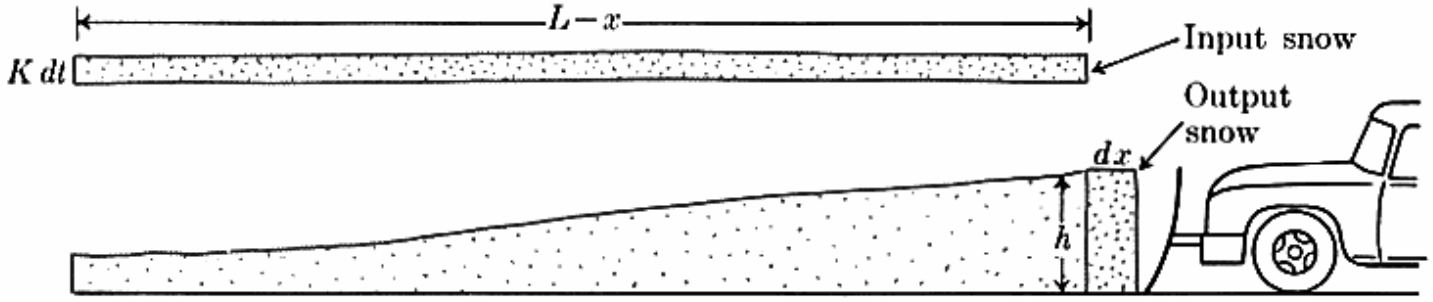


Fig. 67. Equal amounts of snow are input and output; the plow moves dx in time dt.

Fortunately, it turns out that $h(x, t)$ is constant, equal to KT, whenever $x = x(t)$ and $0 \le t \le T$, since the snow falls steadily at position $x(t)$ for $T - t$ units of time after the plow passes that point, plus t units of time before it comes back. In other words, the plow sees all snow at the same height on his journey, assuming that a steady state has been reached where each journey is the same. Hence the total amount of snow cleared (the run length) is KTL; and the amount of snow in memory is the amount cleared after time T, namely $KT(L - x(T))$. The solution to (2) such that $x(0) = 0$ is

$$x(t) = L(1 - e^{-t/T}); \tag{3}$$

hence $P = KTLe^{-1} = (\text{run length})/e$; and this is what we set out to prove.

Exercises 21 through 23 show that this analysis can be extended to the case of general P'; for example, when $P' = 2P$ the average run length turns out to be $e^{\theta}(e - \theta)P$, where $\theta = \frac{1}{2}(e - \sqrt{e^2 - 4})$, a result that probably wouldn't have been guessed offhand! Table 2 shows the dependence of run length on reservoir size; the usefulness of natural selection in a given computer environment can be estimated by referring to this table.

Table 2

RUN LENGTHS BY NATURAL SELECTION

Reservoir size	Run length	Parameter	Reservoir size	Run length	Parameter
1.00000P	2.71828P	1.00000	0.38629P	2.00000P	0.69315
2.00000P	3.53487P	1.43867	1.30432P	3.00000P	1.15881
3.00000P	4.16220P	1.74773	2.72294P	4.00000P	1.66862
4.00000P	4.69445P	2.01212	4.63853P	5.00000P	2.16714
5.00000P	5.16369P	2.24938	21.72222P	10.00000P	4.66667
10.00000P	7.00877P	3.17122	5.29143P	5.29143P	2.31329

The "parameter" $k + \theta$ is defined in exercise 22.

***Analysis of replacement selection.** Let us now return to the case of replacement selection without an auxiliary reservoir. The snowplow analogy gives us a fairly good indication of the average length of runs obtained by replacement

selection "in the limit," but it is possible to get much more precise information about Algorithm R by applying the facts about runs in permutations which we have studied in Section 5.1.3. For this purpose it is convenient to assume that the input file is an arbitrarily long sequence of independent random real numbers between 0 and 1.

Let

$$g_P(z_1, z_2, \ldots, z_k) = \sum_{l_1, l_2, \ldots, l_k \geq 0} a_P(l_1, l_2, \ldots, l_k) z_1^{l_1} z_2^{l_2} \cdots z_k^{l_k}$$

be the generating function for run lengths produced by P-way replacement selection on such a file, where $a_P(l_1, l_2, \ldots, l_k)$ is the probability that the first run has length l_1, the second has length $l_2, \ldots,$ the kth has length l_k. The following "independence theorem" is basic, since it reduces the analysis to the case $P = 1$:

Theorem K. $g_P(z_1, z_2, \ldots, z_k) = g_1(z_1, z_2, \ldots, z_k)^P$.

Proof. Let the input keys be $X_1, X_2, X_3, \ldots$. Algorithm R partitions them into P subsequences, according to which external node position they land in the tree; the subsequence containing X_n is determined by the values of $X_1, \ldots, X_{n-1}$. Each of these subsequences is therefore an independent sequence of independent random numbers between 0 and 1. Furthermore, the output of replacement selection is precisely what would be obtained by doing a P-way merge on these subsequences; an element belongs to the jth run of a subsequence if and only if it belongs to the jth run produced by replacement selection (since in step R4, LASTKEY and KEY(Q) belong to the same subsequence).

In other words, we might just as well assume that Algorithm R is being applied to P independent random input files, and that step R4 reads the next record from the file corresponding to external node Q; in this sense, the algorithm is equivalent to a P-way merge, with "step-downs" marking the ends of the runs.

Thus the output has runs of lengths $(l_1, \ldots, l_k)$ if and only if the subsequences have runs of respective lengths $(l_{11}, \ldots, l_{1k}), \ldots, (l_{P1}, \ldots, l_{Pk})$, where the l_{ij} are some nonnegative integers satisfying $\sum_{1 \leq i \leq P} l_{ij} = l_j$ for $1 \leq j \leq k$. It follows that

$$a_P(l_1, \ldots, l_k) = \sum_{\substack{l_{11} + \cdots + l_{P1} = l_1 \\ \vdots \\ l_{1k} + \cdots + l_{Pk} = l_k}} a_1(l_{11}, \ldots, l_{1k}) \cdots a_1(l_{P1}, \ldots, l_{Pk}),$$

and this is equivalent to the desired result. ∎

We have discussed the average length L_k of the kth run, when $P = 1$, in Section 5.1.3, where the values are tabulated in Table 5.1.3–2. Theorem K implies that the average length of the kth run for general P is P times as long as the average when $P = 1$, namely $L_k P$; and the variance is also P times as

large, so the standard deviation of the run length is proportional to $\sqrt{P}$. These results were first derived by B. J. Gassner about 1958.

Thus the first run produced by Algorithm R will be about $(e - 1)P \approx 1.718P$ records long, for random data; the second run will be about $(e^2 - 2e)P \approx 1.952P$ records long; the third, about $1.996P$; and subsequent runs will be very close to $2P$ records long until we get to the last two runs (see exercise 14). The standard deviation of most of these run lengths is approximately $(4e - 10)P \approx 0.934\sqrt{P}$ [*CACM* **6** (1963), 685–687]. Furthermore, exercise 5.1.3–10 shows that the *total* length of the first k runs will be fairly close to $(2k - \frac{1}{3})P$, with a standard deviation of $((\frac{2}{3}k + \frac{2}{9})P)^{1/2}$. The generating functions $g_1(z, z, \ldots, z)$ and $g_1(1, \ldots, 1, z)$ are derived in exercises 5.1.3–9 and 11.

The above analysis has assumed that the input file is infinitely long, but the proof of Theorem K shows that the same probability $a_p(l_1, \ldots, l_k)$ would be obtained in any random input sequence containing at least $l_1 + \cdots + l_k + P$ elements. So the above results are applicable for, say, files of size $N > (2k + 1)P$, in view of the small standard deviation.

We will be seeing some applications in which the merging pattern wants some of the runs to be ascending and some to be descending. Since the residue accumulated in memory at the end of an ascending run tends to contain numbers somewhat smaller on the average than random data, a change in the direction of ordering decreases the average length of the runs. Consider, for example, a snowplow which must make a U-turn every time it reaches an end of a straight road; it will go very speedily over the area just plowed. The run lengths when directions are reversed vary between $1.5P$ and $2P$ for random data (see exercise 24).

EXERCISES

1. [*10*] What is Step 4, in the example of four-way merging at the beginning of this section?

2. [*12*] What changes would be made to the tree of Fig. 63 if the key 061 were replaced by 612?

3. [*16*] (E. F. Moore.) What output is produced by four-way replacement selection when it is applied to successive words of the following sentence:

```
fourscore and seven years ago our fathers brought forth
on this continent a new nation conceived in liberty and
dedicated to the proposition that all men are created equal.
```

(Use ordinary alphabetic order, treating each word as one key.)

4. [*16*] Apply four-way *natural* selection to the sentence in exercise 3, using a reservoir of capacity 4.

5. [*00*] True or false: Replacement selection using a tree works only when P is a power of 2 or the sum of two powers of 2.

6. [*15*] Algorithm R specifies that P must be ≥ 2; what comparatively small changes to the algorithm would make it valid for all $P \geq 1$?

7. [*17*] What does Algorithm R do when there is no input at all?

8. [*20*] Algorithm R makes use of an artificial key "∞" which must be larger than any possible key. Show that the algorithm might fail if an actual key were equal to ∞, and explain how to modify the algorithm in case the implementation of a true ∞ is inconvenient.

▶ 9. [*23*] How would you modify Algorithm R so that it causes certain specified runs (depending on RC) to be output in ascending order, and others in descending order?

10. [*26*] The initial setting of the LOSER pointers in step R1 usually doesn't correspond to any actual tournament, since external node $P + j$ may not lie in the subtree below internal node j. Explain why Algorithm R works anyway. [*Hint:* Would the algorithm work if {LOSER(LOC(X[0])), . . . , LOSER(LOC(X[P − 1]))} were set to an *arbitrary* permutation of {LOC(X[0]), . . . , LOC(X[P − 1])} in step R1?]

11. [*M25*] True or false: The probability that KEY(Q) < LASTKEY in step R4 is approximately $\frac{1}{2}$, assuming random input.

12. [*M46*] Carry out a detailed analysis of the number of times each portion of Algorithm R is executed; for example, how often are interchanges made in step R6?

13. [*13*] Why is the second run produced by replacement selection usually longer than the first run?

▶ 14. [*HM25*] Use the snowplow analogy to estimate the average length of the *last two runs* produced by replacement selection on a long sequence of input data.

15. [*20*] True or false: The final run produced by replacement selection never contains more than P records. Discuss your answer.

16. [*M26*] Find a "simple" necessary and sufficient condition that a file $R_1\ R_2 \ldots R_N$ will be completely sorted in one pass by P-way replacement selection. What is the probability that this happens, as a function of P and N, when the input is a random permutation of $\{1, 2, \ldots, N\}$?

17. [*20*] What is output by Algorithm R when the input keys are in decreasing order, $K_1 \geq K_2 \geq \cdots \geq K_N$?

▶ 18. [*22*] What happens if Algorithm R is applied *again* to an output file that was produced by Algorithm R?

19. [*HM22*] Use the snowplow analogy to prove that the first run produced by replacement selection is approximately $(e - 1)P$ records long.

20. [*HM24*] Approximately how long is the first run produced by natural selection, when $P = P'$?

▶ 21. [*HM23*] Determine the approximate length of runs produced by natural selection when $P' < P$.

22. [*HM40*] The purpose of this exercise is to determine the average run length obtained in natural selection, when $P' > P$. Let $\kappa = k + \theta$ be a real number ≥ 1,

where $k = \lfloor \kappa \rfloor$ and $\theta = \kappa \bmod 1$, and consider the function $F(\kappa) = F_k(\theta)$, where $F_k(\theta)$ is the polynomial defined by the generating function

$$\sum_{k \geq 0} F_k(\theta) z^k = e^{-\theta z}/(1 - ze^{1-z}).$$

Thus, $F_0(\theta) = 1$, $F_1(\theta) = e - \theta$, $F_2(\theta) = e^2 - e - e\theta + \frac{1}{2}\theta^2$, etc.

Suppose that a snowplow starts out at time 0 to simulate the process of natural selection, and suppose that after T units of time exactly P snowflakes have fallen behind it. At this point a second snowplow begins on the same journey, occupying the same position at time $t + T$ as the first snowplow did at time t. Finally, at time κT, exactly P' snowflakes have fallen behind the first snowplow; he instantaneously plows the rest of the road and disappears.

Using this model to represent the process of natural selection, show that a run length equal to $e^\theta F(\kappa) P$ is obtained when

$$P'/P = k + 1 + e^\theta \left(\kappa F(\kappa) - \sum_{0 \leq j \leq \kappa} F(\kappa - j) \right).$$

23. [*HM35*] The preceding exercise analyzes natural selection when the records from the reservoir are always read in the same order that they were written, first-in-first-out. Find the approximate run length which would be obtained if the reservoir contents from the preceding run were read in completely *random* order, as if the records in the reservoir had been thoroughly shuffled between runs.

24. [*HM39*] The purpose of this exercise is to analyze the effect caused by haphazardly changing the direction of runs in replacement selection.

a) Let $g_P(z_1, z_2, \ldots, z_k)$ be a generating function defined as in Theorem K, but with each of the k runs specified as to whether they are to be ascending or descending. For example, we might say that all odd-numbered runs are ascending, all even-numbered runs are descending. Show that Theorem K is valid for each of the 2^k generating functions of this type.

b) As a consequence of (a), we may assume that $P = 1$. We may also assume that the input is a uniformly distributed sequence of independent random numbers between 0 and 1. Let

$$a(x, y) = \begin{cases} e^{1-x} - e^{y-x}, & \text{if} \quad x \leq y; \\ e^{1-x}, & \text{if} \quad x > y. \end{cases}$$

Given that $f(x)\,dx$ is the probability that a certain ascending run begins with x, prove that $(\int_0^1 a(x, y) f(x)\,dx)\,dy$ is the probability that the following run beings with y. [*Hint:* Consider, for each $n \geq 0$, the probability that $x \leq X_1 \leq \cdots \leq X_n > y$, when x and y are given.]

c) Consider runs that change direction with probability p; in other words, the direction of each run after the first is randomly chosen to be the same as that of the previous run, $q = (1 - p)$ of the time, but it is to be in the opposite direction p of the time. (Thus when $p = 0$, all runs have the same direction; when $p = 1$,

the runs alternate in direction; and when $p = \frac{1}{2}$, the runs are independently random.) Let

$$f_1(x) = 1, \qquad f_{n+1}(y) = p\int_0^1 a(x, y)f_n(1 - x)\,dx + q\int_0^1 a(x, y)f_n(x)\,dx.$$

Show that the probability that the nth run begins with x is $f_n(x)\,dx$ when the $(n - 1)$st run is ascending, $f_n(1 - x)\,dx$ when the $(n - 1)$st run is descending.

d) Find a solution f to the "steady state" equations

$$f(y) = p\int_0^1 a(x, y)f(1 - x)\,dx + q\int_0^1 a(x, y)f(x)\,dx, \qquad \int_0^1 f(x)\,dx = 1.$$

[*Hint:* Show that $f''(x)$ is independent of x.]

e) Show that the sequence $f_n(x)$ in part (c) converges rather rapidly to the function $f(x)$ in part (d). [*Hint:* See, for example, the proof of Theorem 4.5.3L.]

f) Show that the average length of an ascending run starting with x is e^{1-x}.

g) Finally, put the above results together to prove the following theorem: *If the directions of consecutive runs are independently reversed with probability p in replacement selection, the average run length approaches* $(6/(3 + p))P$.

(The case $p = 1$ of this theorem was first derived by Knuth [*CACM* **6** (1963), 685–688]; the case $p = \frac{1}{2}$ was first proved by A. G. Konheim in 1970.)

25. [*HM40*] Consider the following procedure:

N1. Read a record into a one-word "reservoir." Then read another record, R, and let K be its key.

N2. Output the reservoir, set LASTKEY to its key, and set the reservoir empty.

N3. If K < LASTKEY then output R and set LASTKEY ← K and go to N5.

N4. If the reservoir is nonempty, return to N2; otherwise enter R into the reservoir.

N5. Read in a new record, R, and let K be its key. Go to N3. ▮

This is essentially equivalent to natural selection with $P = 1$ and with $P' = 1$ or 2 (depending on whether you choose to empty the reservoir at the moment it fills or at the moment it is about to overfill), except that it produces *descending* runs, and it never stops. The latter anomalies are convenient and harmless assumptions for the purposes of this problem.

Proceeding as in exercise 24, let $f_n(x, y)\,dy\,dx$ be the probability that (x, y) are the respective values of (LASTKEY, K) just after the nth time step N2 is performed. Prove that there is a function $g_n(x)$ of one variable such that $f_n(x, y) = g_n(x)$ when $x < y$, and $f_n(x, y) = g_n(x) - e^{-y}(g_n(x) - g_n(y))$ when $x > y$. This function $g_n(x)$ is defined by the relations $g_1(x) = 1$,

$$g_{n+1}(x) = \int_0^x e^u g_n(u)\,du + \int_0^x dv(v + 1)\int_v^1 du((e^v - 1)g_n(u) + g_n(v))$$
$$+ x\int_x^1 dv\int_v^1 du((e^v - 1)g_n(u) + g_n(v)).$$

Show further that the expected length of the nth run is

$$\int_0^1 dx \int_0^x dy\big(g_n(x)(e^y - 1) + g_n(y)\big)(2 - \tfrac{1}{2}y^2) + \int_0^1 dx(1 - x)g_n(x)e^x.$$

[*Note:* The "steady state" solution to these equations appears to be very complicated; it has been obtained numerically by J. McKenna, showing that the run lengths approach a limiting value of 2.61307209. Theorem K does not apply to natural selection, so the case $P = 1$ does not carry over to other P.

26. [*M33*] Considering the algorithm in exercise 25 as a definition of natural selection when $P' = 1$, find the expected length of the *first* run when $P' = r$, for any $r \geq 0$, as follows.

a) Show that the first run has length n with probability

$$(n + r)\begin{bmatrix} n + r \\ n \end{bmatrix} / (n + r + 1)!.$$

b) Define "second order Stirling numbers" $\left[\left[{n \atop m}\right]\right]$ by the rules

$$\left[\left[{0 \atop m}\right]\right] = \delta_{m0}, \qquad \left[\left[{n \atop m}\right]\right] = (n + m - 1)\left(\left[\left[{n-1 \atop m}\right]\right] + \left[\left[{n-1 \atop m-1}\right]\right]\right) \qquad \text{for} \qquad n > 0.$$

Prove that

$$\begin{bmatrix} n + r \\ n \end{bmatrix} = \sum_{0 \leq k \leq r} \binom{n + r}{k + r}\left[\left[{r \atop k}\right]\right].$$

c) Prove that the average length of the first run is therefore $c_r e - r - 1$, where

$$c_r = \sum_{0 \leq k \leq r} \left[\left[{r \atop k}\right]\right](r + k + 1)/(r + k)!.$$

27. [*25*] The text considers only the case that all records to be sorted have a fixed size. How can replacement selection be done reasonably well on *variable-length* records?

5.4.2. The Polyphase Merge

Now that we have seen how initial runs can be built up, we shall consider various patterns which can be used to distribute them onto tapes and to merge them together until only a single run remains.

Let us begin by assuming that there are three tape units, T1, T2, and T3, available; the technique of "balanced merging," described near the beginning of Section 5.4, can be used with $P = 2$ and $T = 3$, when it takes the following form:

B1. Distribute initial runs alternately on tapes T1 and T2.

B2. Merge runs from T1 and T2 onto T3; then stop if T3 contains only one run.

B3. Copy the runs of T3 alternately onto T1 and T2, then return to B2. ▮

If the initial distribution pass produces S runs, the first merge pass will produce $\lceil S/2 \rceil$ runs on T3, the second will produce $\lceil S/4 \rceil$, etc. Thus if, say, $17 \le S \le 32$, we will have 1 distribution pass, 5 merge passes, and 4 copy passes; in general, if $S > 1$, the number of passes over all the data is $2\lceil \log_2 S \rceil$.

The copying passes in this procedure are undesirable, since they do not reduce the number of runs. Half of the copying can be avoided if we use a *two-phase* procedure:

A1. Distribute initial runs alternately on tapes T1 and T2.

A2. Merge runs from T1 and T2 onto T3; then stop if T3 contains only one run.

A3. Copy *half* of the runs from T3 onto T1.

A4. Merge runs from T1 and T3 onto T2; then stop if T2 contains only one run.

A5. Copy *half* of the runs from T2 onto T1. Return to A2. ▮

The number of passes over the data has been reduced to $\frac{3}{2}\lceil \log_2 S \rceil + \frac{1}{2}$, since steps A3 and A5 do only "half a pass"; about 25 percent of the time has therefore been saved.

The copying can actually be eliminated *entirely*, if we start with F_n runs on T1 and F_{n-1} runs on T2, where F_n and F_{n-1} are consecutive Fibonacci numbers. Consider, for example, the case $n = 7$, $S = F_n + F_{n-1} = 13 + 8 = 21$:

	Contents of T1	Contents of T2	Contents of T3	Remarks
Phase 1.	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	1, 1, 1, 1, 1, 1, 1, 1		Initial distribution
Phase 2.	1, 1, 1, 1, 1	—	2, 2, 2, 2, 2, 2, 2, 2	Merge 8 runs to T3
Phase 3.	—	3, 3, 3, 3, 3	2, 2, 2	Merge 5 runs to T2
Phase 4.	5, 5, 5	3, 3	—	Merge 3 runs to T1
Phase 5.	5	—	8, 8	Merge 2 runs to T3
Phase 6.	—	13	8	Merge 1 run to T2
Phase 7.	21	—	—	Merge 1 run to T1

Here, for example, "2, 2, 2, 2, 2, 2, 2, 2" denotes eight runs of relative length 2, considering each initial run to be of relative length 1. Fibonacci numbers are omnipresent in this chart!

Only phases 1 and 7 are complete passes over the data; phase 2 processes only 16/21 of the initial runs, phase 3 only 15/21, etc., and so the total number of "passes" comes to $(21 + 16 + 15 + 15 + 16 + 13 + 21)/21 = 5\frac{4}{7}$ if we assume that the initial runs have approximately equal length. By comparison, the two-phase procedure above would have required 8 passes to sort these 21 initial runs. We shall see that in general this "Fibonacci" pattern requires approximately $1.04 \log_2 S + 0.99$ passes, making it competitive with a *four*-tape balanced merge although it requires only three tapes.

The same idea can be generalized to T tapes, for any $T \ge 3$, using $(T - 1)$-way merging. We shall see, for example, that the four-tape case requires only about $.703 \log_2 S + 0.96$ passes over the data. The generalized pattern involves generalized Fibonacci numbers. Consider the following six-tape example:

	T1	T2	T3	T4	T5	T6	Initial runs processed
Phase 1.	1^{31}	1^{30}	1^{28}	1^{24}	1^{16}	—	$31 + 30 + 28 + 24 + 16 = 129$
Phase 2.	1^{15}	1^{14}	1^{12}	1^{8}	—	5^{16}	$16 \times 5 = 80$
Phase 3.	1^{7}	1^{6}	1^{4}	—	9^{8}	5^{8}	$8 \times 9 = 72$
Phase 4.	1^{3}	1^{2}	—	17^{4}	9^{4}	5^{4}	$4 \times 17 = 68$
Phase 5.	1^{1}	—	33^{2}	17^{2}	9^{2}	5^{2}	$2 \times 33 = 66$
Phase 6.	—	65^{1}	33^{1}	17^{1}	9^{1}	5^{1}	$1 \times 65 = 65$
Phase 7.	129^{1}	—	—	—	—	—	$1 \times 129 = 129$

Here 1^{31} stands for 31 runs of relative length 1, etc.; five-way merges have been used throughout. This general pattern was developed by R. L. Gilstad [*Proc. AFIPS Eastern Jt. Computer Conf.* **18** (1960), 143–148], who called it the *polyphase merge*. The three-tape case had been discovered earlier by B. K. Betz [unpublished memorandum, Minneapolis-Honeywell Regulator Co. (1956)].

In order to make polyphase merging work as in the above examples, we need to have a "perfect Fibonacci distribution" of runs on the tapes after each phase. By reading the above table from bottom to top, we can see that the first seven perfect Fibonacci distributions when $T = 6$ are $\{1, 0, 0, 0, 0\}$, $\{1, 1, 1, 1, 1\}$, $\{2, 2, 2, 2, 1\}$, $\{4, 4, 4, 3, 2\}$, $\{8, 8, 7, 6, 4\}$, $\{16, 15, 14, 12, 8\}$, and $\{31, 30, 28, 24, 16\}$. The big questions now facing us are

1. What is the rule underlying these perfect Fibonacci distributions?

2. What do we do if S does not correspond to a perfect Fibonacci distribution?

3. How should we design the initial distribution pass so that it produces the desired configuration on the tapes?

4. How many "passes" over the data will a T-tape polyphase merge require, as a function of S (the number of initial runs)?

We shall discuss these four questions in turn, first giving "easy answers" and then making a more intensive analysis.

The perfect Fibonacci distributions can be obtained by running the pattern backwards, cyclically rotating the tape contents. For example, when $T = 6$ we have the following distributions of runs:

Level	T1	T2	T3	T4	T5	Total	Final output will be on
0	1	0	0	0	0	1	T1
1	1	1	1	1	1	5	T6
2	2	2	2	2	1	9	T5
3	4	4	4	3	2	17	T4
4	8	8	7	6	4	33	T3
5	16	15	14	12	8	65	T2
6	31	30	28	24	16	129	T1
7	61	59	55	47	31	253	T6
8	120	116	108	92	61	497	T5

. . . . . .

n	a_n	b_n	c_n	d_n	e_n	t_n	$T(k)$	
$n+1$	a_n+b_n	a_n+c_n	a_n+d_n	a_n+e_n	a_n	t_n+4a_n	$T(k-1)$	(1)

. . . . . .

(Tape T6 will always be empty after the initial distribution.)

The rule for going from level n to level $n+1$ shows that the condition

$$a_n \geq b_n \geq c_n \geq d_n \geq e_n \tag{2}$$

will hold in every level. In fact, it is easy to see from (1) that

$$\begin{aligned} e_n &= a_{n-1}, \\ d_n &= a_{n-1} + e_{n-1} = a_{n-1} + a_{n-2}, \\ c_n &= a_{n-1} + d_{n-1} = a_{n-1} + a_{n-2} + a_{n-3}, \\ b_n &= a_{n-1} + c_{n-1} = a_{n-1} + a_{n-2} + a_{n-3} + a_{n-4}, \\ a_n &= a_{n-1} + b_{n-1} = a_{n-1} + a_{n-2} + a_{n-3} + a_{n-4} + a_{n-5}, \end{aligned} \tag{3}$$

where $a_0 = 1$ and where we let $a_n = 0$ for $n = -1, -2, -3, -4$.

The *pth order Fibonacci numbers* $F_n^{(p)}$ are defined by the rules

$$\begin{aligned} F_n^{(p)} &= F_{n-1}^{(p)} + F_{n-2}^{(p)} + \cdots + F_{n-p}^{(p)}, \qquad \text{for} \qquad n \geq p; \\ F_n^{(p)} &= 0 \qquad \text{for} \qquad 0 \leq n \leq p-2; \qquad F_{p-1}^{(p)} = 1. \end{aligned} \tag{4}$$

In other words, we start with $p-1$ 0's, then 1, and then each number is the sum of the preceding p values. When $p = 2$, this is the usual Fibonacci sequence; for larger values of p the sequence was apparently first studied by V. Schlegel in *El Progreso Matemático* 4 (1894), 173–174. Schlegel derived the generating function

$$\sum_{n\geq 0} F_n^{(p)} z^n = \frac{z^{p-1}}{1 - z - z^2 - \cdots - z^p} = \frac{z^{p-1} - z^p}{1 - 2z + z^{p+1}}. \tag{5}$$

Equation (3) shows that the number of runs on T1 during a six-tape polyphase merge is a fifth-order Fibonacci number: $a_n = F_{n+4}^{(5)}$.

In general, if we set $P = T - 1$, the polyphase merge distributions for T tapes will correspond to Pth order Fibonacci numbers in the same way. The kth tape gets

$$F_{n+P-2}^{(P)} + F_{n+P-3}^{(P)} + \cdots + F_{n+k-2}^{(P)}$$

initial runs in the perfect nth level distribution, for $1 \leq k \leq P$, and the total number of initial runs on all tapes is therefore

$$t_n = PF_{n+P-2}^{(P)} + (P-1)F_{n+P-3}^{(P)} + \cdots + F_{n-1}^{(P)}. \tag{6}$$

This settles the issue of "perfect Fibonacci distributions." But what should we do if S is not exactly equal to t_n, for any n? And how do we get the runs onto the tapes in the first place?

When S isn't perfect (and so few values are), we can do just as we did in balanced P-way merging, adding artificial "dummy runs" so that we can pretend S is perfect after all. There are several ways to add the dummy runs, and we aren't ready yet to analyze the "best" way of doing this. We shall discuss first a method of distribution and dummy-run assignment which isn't strictly optimal, although it has the virtue of simplicity and appears to be better than all other equally simple methods.

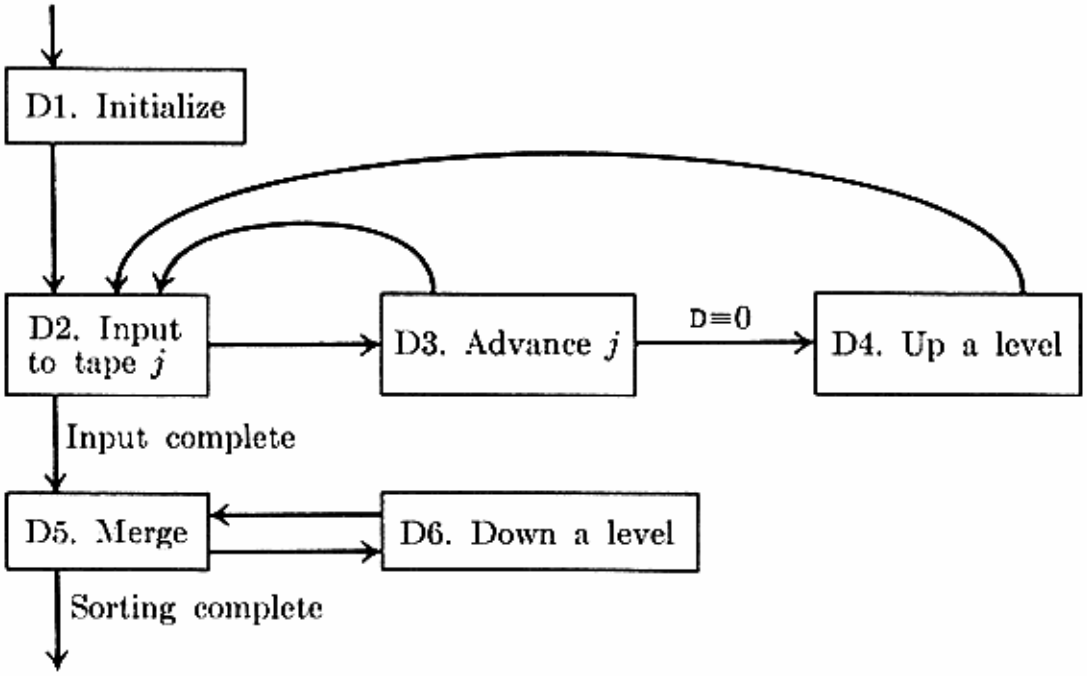


Fig. 68. Polyphase merge sorting.

Algorithm D (*Polyphase merge sorting with "horizontal" distribution*). This algorithm takes initial runs and disperses them to tapes, one run at a time, until the supply of initial runs is exhausted. Then it specifies how the tapes are to be merged, assuming that there are $T = P + 1 \geq 3$ available tape units, using P-way merging. Tape T may be used to hold the input, since it does not receive any initial runs. The following tables are maintained:

A[j], $1 \leq j \leq T$: The perfect Fibonacci distribution we are striving for.

D[j], $1 \leq j \leq T$: Number of dummy runs assumed to be present at the beginning of logical tape unit number j.

TAPE[j], $1 \leq j \leq T$: Number of the physical tape unit corresponding to logical tape unit number j.

(It is convenient to deal with "logical tape unit numbers" whose assignment to physical tape units varies as the algorithm proceeds.)

D1. [Initialize.] Set A[j] $\leftarrow$ D[j] $\leftarrow 1$ and TAPE[j] $\leftarrow j$, for $1 \leq j < T$. Set A[T] $\leftarrow$ D[T] $\leftarrow 0$ and TAPE[T] $\leftarrow T$. Then set $l \leftarrow 1$; $j \leftarrow 1$.

D2. [Input to tape j.] Write one run on tape number j, and decrease $\mathtt{D}[j]$ by 1. Then if the input is exhausted, rewind all the tapes and go to step D5.

D3. [Advance j.] If $\mathtt{D}[j] < \mathtt{D}[j+1]$, increase j by 1 and return to D2. Otherwise if $\mathtt{D}[j] = 0$, go on to D4. Otherwise set $j \leftarrow 1$ and return to D2.

D4. [Up a level.] Set $l \leftarrow l + 1$, $a \leftarrow \mathtt{A}[1]$, and then for $j = 1, 2, \ldots, P$ (in this order) set $\mathtt{D}[j] \leftarrow a + \mathtt{A}[j+1] - \mathtt{A}[j]$ and $\mathtt{A}[j] \leftarrow a + \mathtt{A}[j+1]$. (See (1) and note that $\mathtt{A}[P+1]$ is always zero. At this point we will have $\mathtt{D}[1] \geq \mathtt{D}[2] \geq \cdots \geq \mathtt{D}[T]$.)
Now set $j \leftarrow 1$ and return to D2.

D5. [Merge.] If $l = 0$, sorting is complete and the output is on $\mathtt{TAPE}[1]$. Otherwise, merge runs from $\mathtt{TAPE}[1], \ldots, \mathtt{TAPE}[P]$ onto $\mathtt{TAPE}[T]$ until $\mathtt{TAPE}[P]$ is empty and $\mathtt{D}[P] = 0$. The merging process should operate as follows, for each run merged: If $\mathtt{D}[j] > 0$ for all j, $1 \leq j \leq P$, then increase $\mathtt{D}[T]$ by 1 and decrease each $\mathtt{D}[j]$ by 1 for $1 \leq j \leq P$; otherwise merge one run from each $\mathtt{TAPE}[j]$ such that $\mathtt{D}[j] = 0$, and decrease $\mathtt{D}[j]$ by 1 for each other j. (Thus the dummy runs are imagined to be at the *beginning* of the tape instead of at the ending.)

D6. [Down a level.] Set $l \leftarrow l - 1$. Rewind $\mathtt{TAPE}[P]$ and $\mathtt{TAPE}[T]$. (Actually the rewinding of $\mathtt{TAPE}[P]$ could have been initiated during step D5, just after its last block was input.) Then set $(\mathtt{TAPE}[1], \mathtt{TAPE}[2], \ldots, \mathtt{TAPE}[T]) \leftarrow (\mathtt{TAPE}[T], \mathtt{TAPE}[1], \ldots, \mathtt{TAPE}[T-1])$, $(\mathtt{D}[1], \mathtt{D}[2], \ldots, \mathtt{D}[T]) \leftarrow (\mathtt{D}[T], \mathtt{D}[1], \ldots, \mathtt{D}[T-1])$, and return to step D5. ▮

The distribution rule which is stated so succinctly in step D3 of this algorithm is intended to equalize the number of dummies on each tape as well as possible. Figure 69 illustrates the order of distribution when we go from level 4

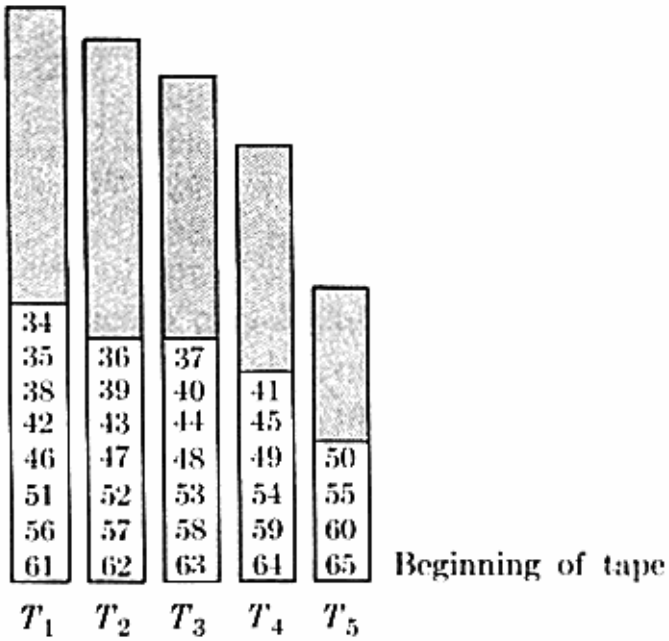


Fig. 69. The order in which runs 34 through 65 are distributed to tapes, when advancing from level 4 to level 5. (See the table of perfect distributions at the bottom of page 268.) Shaded areas represent the first 33 runs which were distributed when level 4 was reached.

(33 runs) to level 5 (65 runs) in a six-tape sort; if there were only, say, 53 initial runs, all runs numbered 54 and higher would be treated as dummies. (The runs are actually being written at the end of the tape, but it is best to imagine them being written at the beginning, since the dummies are assumed to be at the beginning.)

We have now discussed the first three questions listed above, and it remains to consider the number of "passes" over the data. Comparing our six-tape example to the table (1), we see that the total number of initial runs processed when $S = t_6$ was $a_5t_1 + a_4t_2 + a_3t_3 + a_2t_4 + a_1t_5 + a_0t_6$, excluding the initial distribution pass. Exercise 4 derives the generating functions

$$\begin{aligned} a(z) &= \sum_{n\geq 0} a_n z^n = \frac{1}{1 - z - z^2 - z^3 - z^4 - z^5}, \\ t(z) &= \sum_{n\geq 1} t_n z^n = \frac{5z + 4z^2 + 3z^3 + 2z^4 + z^5}{1 - z - z^2 - z^3 - z^4 - z^5}. \end{aligned} \tag{7}$$

It follows that, in general, the number of initial runs processed when $S = t_n$ is exactly the coefficient of z^n in $a(z) \cdot t(z)$, plus t_n (for the initial distribution pass). This makes it possible to calculate the asymptotic behavior of polyphase merging, as shown in exercises 5–7, and we obtain the results displayed in Table 1.

Table 1

APPROXIMATE BEHAVIOR OF POLYPHASE MERGE SORTING

Tapes	Phases	Passes	Pass/phase, percent	Growth ratio
3	$2.078 \ln S + 0.672$	$1.504 \ln S + 0.992$	72	1.6180340
4	$1.641 \ln S + 0.364$	$1.015 \ln S + 0.965$	62	1.8392868
5	$1.524 \ln S + 0.078$	$0.863 \ln S + 0.921$	57	1.9275620
6	$1.479 \ln S - 0.185$	$0.795 \ln S + 0.864$	54	1.9659482
7	$1.460 \ln S - 0.424$	$0.762 \ln S + 0.797$	52	1.9835828
8	$1.451 \ln S - 0.642$	$0.744 \ln S + 0.723$	51	1.9919642
9	$1.447 \ln S - 0.838$	$0.734 \ln S + 0.646$	51	1.9960312
10	$1.445 \ln S - 1.017$	$0.728 \ln S + 0.568$	50	1.9980295
20	$1.443 \ln S - 2.170$	$0.721 \ln S - 0.030$	50	1.9999981

In Table 1, the "growth ratio" is $\lim_{n\to\infty} t_{n+1}/t_n$, the approximate factor by which the number of runs increases at each level. "Passes" denotes the average number of times each record is processed, namely $(1/S)$ times the total number

of initial runs processed during the distribution and merge phases. The stated number of passes and phases is correct in each case up to $O(S^{-\epsilon})$, for some $\epsilon > 0$, for perfect distributions as $S \to \infty$.

Figure 70 shows the average number of times each record is merged, as a function of S, when Algorithm D is used to handle the case of nonperfect numbers. Note that with three tapes there are "peaks" of relative inefficiency occurring just after the perfect distributions, but this phenomenon largely disappears when there are four or more tapes. The use of eight or more tapes gives comparatively little improvement over six or seven tapes.

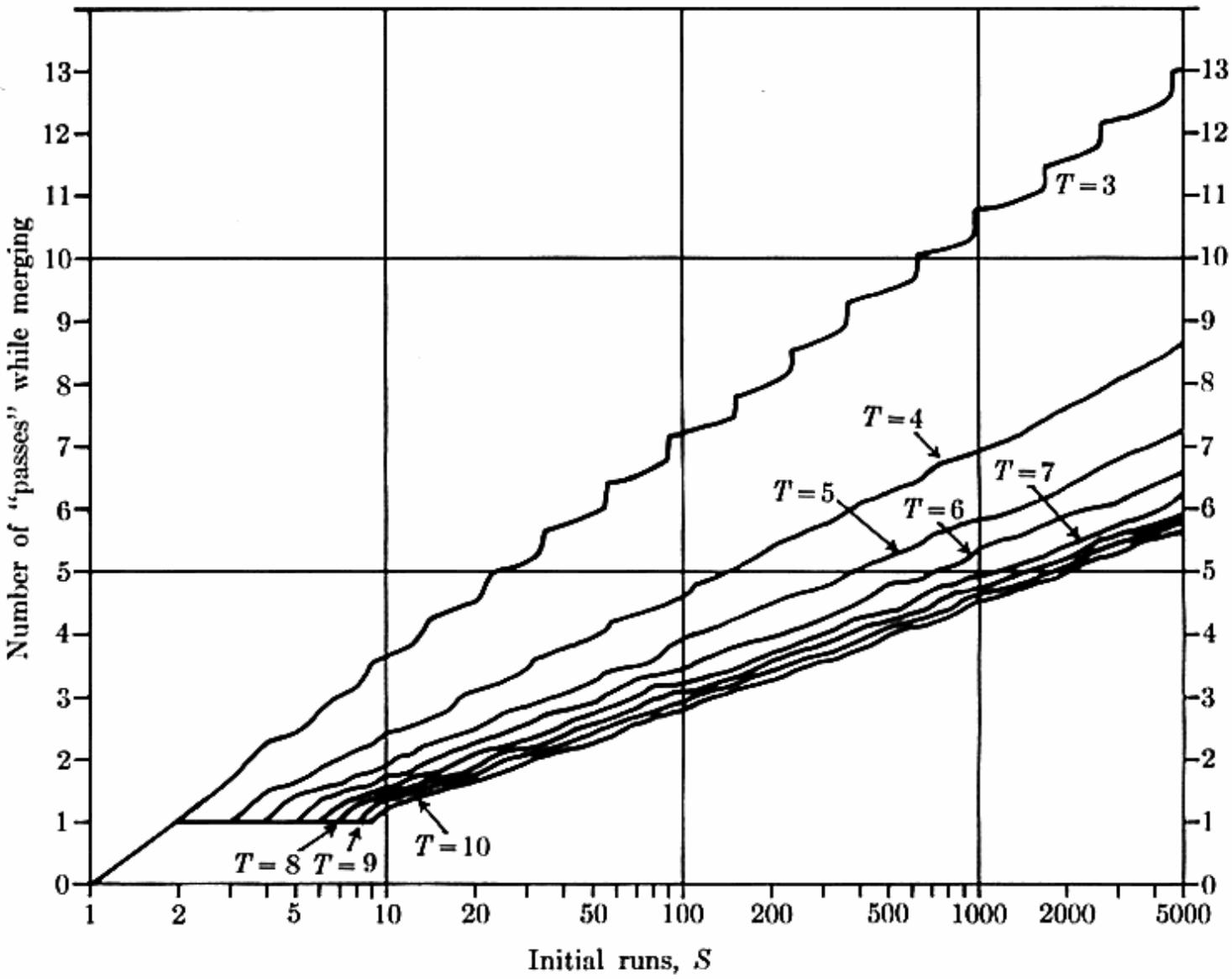


Fig. 70. Efficiency of polyphase merge using Algorithm D.

***A closer look.** In a balanced merge requiring k passes, every record is processed exactly k times during the course of the sort. But the polyphase procedure does not have this lack of bias; some records may get processed many more times than others, and we can gain speed if we arrange to put dummy runs into the oft-processed positions.

Let us therefore study the polyphase distribution more closely; instead of merely looking at the number of runs on each tape, as in (1), let us associate

with each run its *merge number*, the number of times it will be processed during the complete polyphase sort. We get the following table in place of (1):

Level	T1	T2	T3	T4	T5
0	0	—	—	—	—
1	1	1	1	1	1
2	21	21	21	21	2
3	3221	3221	3221	322	32
4	43323221	43323221	4332322	433232	4332
5	5443433243323221	544343324332322	54434332433232	544343324332	54434332
. . .	. . .	. . .	. . .	. . .	. . .
n	A_n	B_n	C_n	D_n	E_n
$n+1$	$(A_n+1)B_n$	$(A_n+1)C_n$	$(A_n+1)D_n$	$(A_n+1)E_n$	A_n+1
. . .	. . .	. . .	. . .	. . .	. . .

(8)

Here A_n is a string of a_n values representing the merge numbers for each run on T1, if we begin with the level n distribution; B_n is the corresponding string for T2; etc. The notation "$(A_n + 1)B_n$" means, "A_n with all values increased by 1, followed by B_n."

Figure 71(a) shows A_5, B_5, C_5, D_5, E_5 tipped on end, showing how the merge numbers for each run appear on tape; note, for example, that the run at the beginning of each tape will be processed five times, while the run at the end of T1 will be processed only once. This discriminatory practice of the polyphase merge makes it much better to put a dummy run at the beginning of the tape than at the end. Figure 71(b) shows an optimum order in which to distribute runs for a five-level polyphase merge, placing each new run into a position with the smallest available merge number. Note that Algorithm D (Fig. 69) is not quite as good, since it fills some "4" positions before all of the "3" positions are used up.

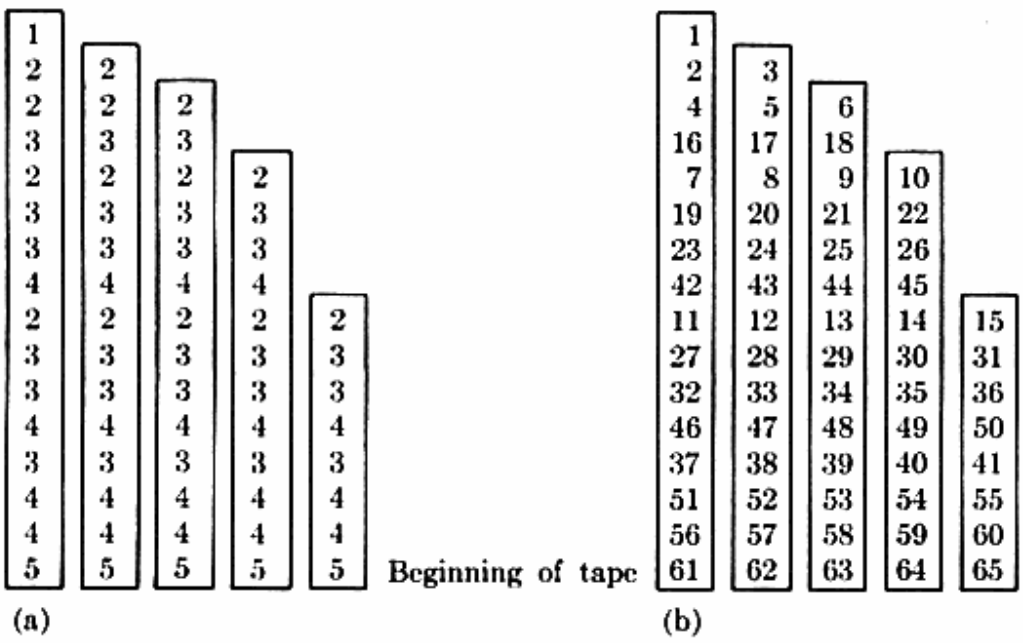


Fig. 71. Analysis of the fifth-level polyphase distribution for six tapes: (a) merge numbers, (b) optimum distribution order.

The recurrence relations (8) show that each of B_n, C_n, D_n, E_n are initial substrings of A_n. In fact, we can use (8) to derive the formulas

$$\begin{aligned} E_n &= (A_{n-1}) + 1, \\ D_n &= (A_{n-1}A_{n-2}) + 1, \\ C_n &= (A_{n-1}A_{n-2}A_{n-3}) + 1, \\ B_n &= (A_{n-1}A_{n-2}A_{n-3}A_{n-4}) + 1, \\ A_n &= (A_{n-1}A_{n-2}A_{n-3}A_{n-4}A_{n-5}) + 1, \end{aligned} \tag{9}$$

generalizing Eqs. (3), which treated only the lengths of these strings. Furthermore, the rule defining the A's implies that essentially the same structure is present at the beginning of every level; we have

$$A_n = n - Q_n, \tag{10}$$

where Q_n is a string of a_n values defined by the law

$$\begin{gathered} Q_n = Q_{n-1}(Q_{n-2} + 1)(Q_{n-3} + 2)(Q_{n-4} + 3)(Q_{n-5} + 4), \quad \text{for} \quad n \geq 1; \\ Q_0 = \text{`0'}; \qquad Q_n = (\text{empty}) \qquad \text{for} \qquad n < 0. \end{gathered} \tag{11}$$

Since Q_n begins with Q_{n-1}, we can consider the *infinite* string Q_∞, whose first a_n elements are equal to Q_n; this string Q_∞ essentially characterizes all the merge numbers in the polyphase distribution. In the six-tape case,

$$Q_\infty = 01121223122323341223233423343441223233423343445233434453445451223 2 \cdots. \tag{12}$$

Exercise 11 contains an interesting interpretation of this string.

Given that A_n is the string $m_1 m_2 \ldots m_{a_n}$, let $A_n(x) = x^{m_1} + x^{m_2} + \cdots + x^{m_{a_n}}$ be the corresponding generating function which counts the number of times each merge number appears; and define $B_n(x)$, $C_n(x)$, $D_n(x)$, $E_n(x)$ similarly. For example, $A_4(x) = x^4 + x^3 + x^3 + x^2 + x^3 + x^2 + x^2 + x = x^4 + 3x^3 + 3x^2 + x$. Relations (9) tell us that

$$\begin{aligned} E_n(x) &= x(A_{n-1}(x)), \\ D_n(x) &= x(A_{n-1}(x) + A_{n-2}(x)), \\ C_n(x) &= x(A_{n-1}(x) + A_{n-2}(x) + A_{n-3}(x)), \\ B_n(x) &= x(A_{n-1}(x) + A_{n-2}(x) + A_{n-3}(x) + A_{n-4}(x)), \\ A_n(x) &= x(A_{n-1}(x) + A_{n-2}(x) + A_{n-3}(x) + A_{n-4}(x) + A_{n-5}(x)), \end{aligned} \tag{13}$$

for $n \geq 1$, where $A_0(x) = 1$ and $A_n(x) = 0$ for $n = -1, -2, -3, -4$. Consequently,

$$\begin{aligned} \sum_{n \geq 0} A_n(x) z^n &= \frac{1}{1 - x(z + z^2 + z^3 + z^4 + z^5)} \\ &= \sum_{k \geq 0} x^k (z + z^2 + z^3 + z^4 + z^5)^k. \end{aligned} \tag{14}$$

Considering the runs on all tapes, we let

$$T_n(x) = A_n(x) + B_n(x) + C_n(x) + D_n(x) + E_n(x), \qquad n \geq 1; \tag{15}$$

from (13) we immediately have

$$T_n(x) = 5A_{n-1}(x) + 4A_{n-2}(x) + 3A_{n-3}(x) + 2A_{n-4}(x) + A_{n-5}(x),$$

hence

$$\sum_{n\geq 1} T_n(x)z^n = \frac{x(5z + 4z^2 + 3z^3 + 2z^4 + z^5)}{1 - x(z + z^2 + z^3 + z^4 + z^5)}, \tag{16}$$

The form of (16) shows that it is easy to compute the coefficients of $T_n(x)$:

	z	z^2	z^3	z^4	z^5	z^6	z^7	z^8	z^9	z^{10}	z^{11}	z^{12}	z^{13}	z^{14}	
x	5	4	3	2	1	0	0	0	0	0	0	0	0	0	
x^2	0	5	9	12	14	15	10	6	3	1	0	0	0	0	
x^3	0	0	5	14	26	40	55	60	57	48	35	20	10	4	(17)
x^4	0	0	0	5	19	45	85	140	195	238	260	255	220	170	
x^5	0	0	0	0	5	24	69	154	294	484	703	918	1088	1168	

The columns of this tableau give $T_n(x)$; for example, $T_4(x) = 2x + 12x^2 + 14x^3 + 5x^4$. After the first row, each entry in the tableau is the sum of the five entries just above and to the left in the previous row.

The number of runs in a "perfect" nth level distribution is $T_n(1)$, and the total amount of processing as these runs are merged is the derivative, $T_n'(1)$. Now

$$\sum_{n\geq 1} T_n'(x)z^n = \frac{5z + 4z^2 + 3z^3 + 2z^4 + z^5}{(1 - x(z + z^2 + z^3 + z^4 + z^5))^2}; \tag{18}$$

setting $x = 1$ in (16) and (18) gives a result in agreement with our earlier demonstration that the merge processing for a perfect nth level distribution is the coefficient of z^n in $a(z)t(z)$ [cf. (7)].

We can use the functions $T_n(x)$ to determine the work involved when dummy runs are added in an optimum way. Let $\Sigma_n(m)$ be the sum of the smallest m merge numbers in an nth level distribution. These values are readily calculated by looking at the columns of (17), and we find that $\Sigma_n(m) =$

$m =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$n = 1$	1	2	3	4	5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$n = 2$	1	2	3	4	6	8	10	12	14	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$n = 3$	1	2	3	5	7	9	11	13	15	17	19	21	24	27	30	33	36	∞	∞	∞	∞
$n = 4$	1	2	4	6	8	10	12	14	16	18	20	22	24	26	29	32	35	38	41	44	47
$n = 5$	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	32	35	38	41	44	47
$n = 6$	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	33	36	39	42	45	48
$n = 7$	2	4	6	8	10	12	14	16	18	20	23	26	29	32	35	38	41	44	47	50	53

(19)

For example, if we wish to sort 17 runs using a level-3 distribution, the total amount of processing is $\Sigma_3(17) = 36$; but if we use a level-4 or level-5 distribution and position the dummy runs optimally, the total amount of processing during the merge phases is only $\Sigma_4(17) = \Sigma_5(17) = 35$. It is better to use level 4, even though 17 corresponds to a "perfect" level-3 distribution! Indeed, as S gets large it turns out that the optimum number of levels is many more than that used in Algorithm D.

Exercise 14 shows that there is a nondecreasing sequence of numbers M_n such that level n is "optimum" for $M_n \le S < M_{n+1}$, but not for $S \ge M_{n+1}$. In the six-tape case the table of $\Sigma_n(m)$ we have just calculated shows that

$$M_0 = 0, \qquad M_1 = 2, \qquad M_2 = 6, \qquad M_3 = 10, \qquad M_4 = 14.$$

The above discussion treats only the case of six tapes, but it is clear that the same ideas apply to polyphase merging with T tapes for any $T \ge 3$; we simply replace 5 by $P = T - 1$ in all appropriate places. Table 2 shows the sequences M_n obtained for various values of T. Table 3 and Fig. 72 indicate the total number of initial runs that are processed after making an optimum

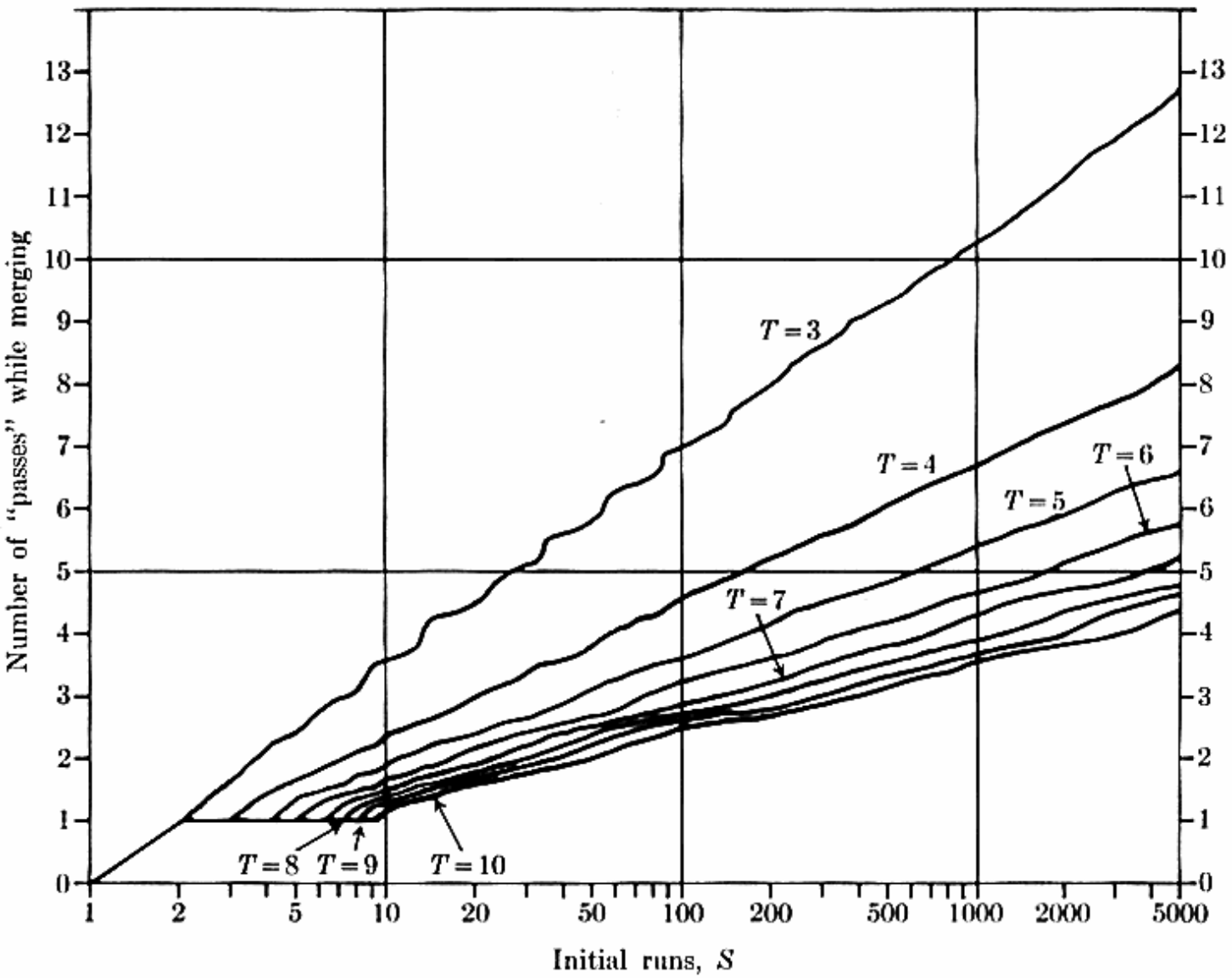


Fig. 72. Efficiency of polyphase merge with optimum initial distribution (cf. Fig. 70).

distribution of dummy runs. (Formulas which appear at the bottom of Table 3 should be taken with a grain of salt, since they are "least-squares fits" over the range $1 \leq S \leq 5000$ ($1 \leq S \leq 10000$ for $T = 3$); this leads to somewhat erratic behavior because the given range of S values is not equally favorable for all T. As $S \rightarrow \infty$, the number of initial runs processed after an optimum polyphase distribution is asymptotically $S \log_P S$, but convergence to this asymptotic limit is extremely slow.)

Table 2

NUMBER OF RUNS FOR WHICH A GIVEN LEVEL IS OPTIMUM

Level	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$	
1	2	2	2	2	2	2	2	2	M_1
2	3	4	5	6	7	8	9	10	M_2
3	4	6	8	10	12	14	16	18	M_3
4	6	10	14	14	17	20	23	26	M_4
5	9	18	23	29	20	24	28	32	M_5
6	14	32	35	43	53	27	32	37	M_6
7	22	55	76	61	73	88	35	41	M_7
8	35	96	109	154	98	115	136	44	M_8
9	56	173	244	216	283	148	171	199	M_9
10	90	280	359	269	386	168	213	243	M_{10}
11	145	535	456	779	481	640	240	295	M_{11}
12	234	820	1197	1034	555	792	1002	330	M_{12}
13	378	1635	1563	1249	1996	922	1228	1499	M_{13}
14	611	2401	4034	3910	2486	1017	1432	1818	M_{14}
15	988	4959	5379	4970	2901	4397	1598	2116	M_{15}
16	1598	7029	6456	5841	10578	5251	1713	2374	M_{16}
17	2574	14953	18561	19409	13097	5979	8683	2576	M_{17}
18	3955	20583	22876	23918	15336	6499	10069	2709	M_{18}
19	6528	44899	64189	27557	17029	30164	11259	15787	M_{19}

Table 3

INITIAL RUNS PROCESSED DURING OPTIMUM POLYPHASE MERGE

S	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$
10	36	24	19	17	15	14	13	12
20	90	60	49	44	38	36	34	33
50	294	194	158	135	128	121	113	104
100	702	454	362	325	285	271	263	254
1000	10371	6680	5430	4672	4347	3872	3739	3632
5000	63578	41286	32905	28620	26426	23880	23114	22073
S	$\{(1.51$	0.951	0.761	0.656	0.589	0.548	0.539	$0.488) \cdot S \ln S$
	$+ (-.11$	$+ .14$	$+ .16$	$+ .19$	$+ .21$	$+ .20$	$+ .02$	$+ .18) \cdot S$

Table 4 shows how the distribution method of Algorithm D compares with the results of optimum distribution in Table 3. It is clear that Algorithm D is not very close to the optimum when S and T become large; but it is not clear how to do much better than Algorithm D without considerable complication in such cases, especially if we do not know S in advance. Fortunately, we rarely have to worry about large S (see Section 5.4.6), so Algorithm D is not too bad in practice; in fact, it's pretty good.

Table 4

INITIAL RUNS PROCESSED DURING STANDARD POLYPHASE MERGE

S	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$
10	36	24	19	17	15	14	13	12
20	90	62	49	44	41	37	34	33
50	294	194	167	143	134	131	120	114
100	714	459	393	339	319	312	292	277
1000	10730	6920	5774	5370	4913	4716	4597	4552
5000	64740	43210	36497	32781	31442	29533	28817	28080

Polyphase sorting was first analyzed mathematically by W. C. Carter [*Proc. IFIP Congress* (1962), 62–66]. Many of the above results about optimal dummy run placement are due originally to B. Sackman and T. Singer ["A vector model for merge sort analysis," unpublished paper presented at the ACM Sort Symposium (November, 1962), 21 pp.]. Sackman later suggested the horizontal method of distribution used in Algorithm D. Donald Shell [*CACM* **14** (1971), 713–719; **15** (1972), 28] developed the theory independently, noted relation (10), and made a detailed study of several different distribution algorithms. Further instructive developments and refinements have been made by Derek A. Zave [*JACM* (to appear)]. The generating function (16) was first investigated by W. Burge [*Proc. IFIP Congress* (1971), to appear].

But what about rewind time? So far we have taken "initial runs processed" as the sole measure of efficiency for comparing tape merge strategies. But after each of phases 2 through 6, in the examples at the beginning of this section, it is necessary for the computer to wait for two tapes to rewind; both the previous output tape and the new current output tape must be repositioned at the beginning, before the next phase can proceed. This can cause a significant delay, since the previous output tape generally contains a significant percentage of the records being sorted (see the "Pass/Phase" column in Table 1). It is a shame to have the computer twiddling its thumbs during all these rewind operations, since useful work could be done with the other tapes if we used a different merging pattern.

A simple modification of the polyphase procedure will overcome this problem, although it requires at least five tapes [see Y. Cesari, thesis, U. of Paris (1968), 25–27, where the idea is credited to J. Caron]. Each phase in Caron's scheme merges runs from $T - 3$ tapes onto another tape, while the remaining two tapes are rewinding.

For example, consider the case of six tapes and 49 initial runs. In the following tableau, R denotes rewinding during the phase, and T5 is assumed to contain the original input:

Phase	T1	T2	T3	T4	T5	T6	Write time	Rewind time
1	1^{11}	1^{17}	1^{13}	1^{8}	—	(R)	49	17
2	(R)	1^{9}	1^{5}	—	R	3^{8}	$8 \times 3 = 24$	$49 - 17 = 32$
3	1^{6}	1^{4}	—	R	3^{5}	R	$5 \times 3 = 15$	max (8, 24)
4	1^{2}	—	R	5^{4}	R	3^{4}	$4 \times 5 = 20$	max (13, 15)
5	—	R	7^{2}	R	3^{3}	3^{2}	$2 \times 7 = 14$	max (17, 20)
6	R	11^{2}	R	5^{2}	3^{1}	—	$2 \times 11 = 22$	max (11, 14)
7	15^{1}	R	7^{1}	5^{1}	—	R	$1 \times 15 = 15$	max (22, 24)
8	R	11^{1}	7^{0}	—	R	23^{1}	$1 \times 23 = 23$	max (15, 15)
9	15^{1}	11^{1}	—	R	33^{0}	R	$0 \times 33 = 0$	max (20, 23)
10	(15^{0})	—	R	49^{1}	(R)	(23^{0})	$1 \times 49 = 49$	14

Here all the rewind time is essentially overlapped, except in phase 9 (a "dummy phase" which prepares for the final merge), and after the initial distribution phase (when all tapes are rewound). If t is the time to merge the number of records in one initial run, and if r is the time to rewind over one initial run, this process takes about $182t + 40r$ plus the time for initial distribution and final rewind. The corresponding figures for standard polyphase using Algorithm D are $140t + 104r$, which is slightly worse when $r = \frac{3}{4}t$, slightly better when $r = \frac{1}{2}t$.

Everything we have said about standard polyphase can be adapted to Caron's polyphase; for example, the sequence a_n now satisfies the recurrence

$$a_n = a_{n-2} + a_{n-3} + a_{n-4} \tag{20}$$

instead of (3). The reader will find it instructive to analyze this method in the same way we analyzed standard polyphase, since it will enhance an understanding of both methods. (See, for example, exercises 19 and 20.)

Table 5 gives facts about Polyphase Caron which are analogous to the facts about Polyphase Ordinaire in Table 1. Note that Caron's method actually becomes *superior* to polyphase on eight or more tapes, in the number of runs processed as well as in the rewind time, even though it does $(T - 3)$-way merging instead of $(T - 1)$-way merging!

Table 5

APPROXIMATE BEHAVIOR OF CARON'S POLYPHASE MERGE SORT

Tapes	Phases	Passes	Pass/phase, percent	Growth ratio
5	$3.556 \ln S + 0.158$	$1.463 \ln S + 1.016$	41	1.3247180
6	$2.616 \ln S - 0.166$	$0.951 \ln S + 1.014$	36	1.4655712
7	$2.337 \ln S - 0.472$	$0.781 \ln S + 1.001$	33	1.5341577
8	$2.216 \ln S - 0.762$	$0.699 \ln S + 0.980$	32	1.5701473
9	$2.156 \ln S - 1.034$	$0.654 \ln S + 0.954$	30	1.5900054
10	$2.124 \ln S - 1.290$	$0.626 \ln S + 0.922$	29	1.6013473
20	$2.078 \ln S - 3.093$	$0.575 \ln S + 0.524$	28	1.6179086

This may seem paradoxical until we realize that *a high order of merge does not necessarily imply an efficient sort.* As an extreme example, consider placing 1 run on T1 and n runs on T2, T3, T4, T5; if we alternately do five-way merging to T6 and T1 until T2, T3, T4, T5 are empty, the processing time is $(2n^2 + 3n)$ initial run lengths, essentially proportional to S^2 instead of $S \log S$, although five-way merging was done throughout.

Tape splitting. Efficient overlapping of rewind time is a problem which arises in many applications, not just sorting, and there is a general approach which can often be used. Consider an iterative process which uses two tapes in the following way:

	T1	T2
Phase 1	Output 1	—
	Rewind	—
Phase 2	Input 1	Output 2
	Rewind	Rewind
Phase 3	Output 3	Input 2
	Rewind	Rewind
Phase 4	Input 3	Output 4
	Rewind	Rewind

and so on, where "Output k" means write the kth output file and "Input k" means read it. The rewind time can be avoided when three tapes are used, as suggested by C. Weisert [*CACM* **5** (1962), 102]:

	T1	T2	T3
Phase 1	Output 1.1	—	—
	Output 1.2	—	—
	Rewind	Output 1.3	—
Phase 2	Input 1.1	Output 2.1	—
	Input 1.2	Rewind	Output 2.2
	Rewind	Input 1.3	Output 2.3
Phase 3	Output 3.1	Input 2.1	Rewind
	Output 3.2	Rewind	Input 2.2
	Rewind	Output 3.3	Input 2.3
Phase 4	Input 3.1	Output 4.1	Rewind
	Input 3.2	Rewind	Output 4.2
	Rewind	Input 3.3	Output 4.3

and so on. Here "Output $k.j$" means, write the jth third of the kth output file, and "Input $k.j$" means, read it. Virtually all of the rewind time will be eliminated if rewinding is at least twice as fast as the read/write speed. Such a procedure, in which the output of each phase is divided between tapes, is called "tape-splitting."

R. L. McAllester [*CACM* 7 (1964), 158–159] has shown that tape-splitting leads to an efficient way of overlapping the rewind time in a polyphase merge. His method can be used with four or more tapes, and it does $(T - 2)$-way merging.

Assuming once again that we have six tapes, let us try to design a merge pattern that operates as follows, splitting the output on each level, where "I", "O", and "R," respectively, denote input, output, and rewinding:

Level	T1	T2	T3	T4	T5	T6	Number of runs output
7	I	I	I	I	R	O	u_7
	I	I	I	I	O	R	v_7
6	I	I	I	R	O	I	u_6
	I	I	I	O	R	I	v_6
5	I	I	R	O	I	I	u_5
	I	I	O	R	I	I	v_5
4	I	R	O	I	I	I	u_4
	I	O	R	I	I	I	v_4
3	R	O	I	I	I	I	u_3
	O	R	I	I	I	I	v_3
2	O	I	I	I	I	R	u_2
	R	I	I	I	I	O	v_2
1	I	I	I	I	R	O	u_1
	I	I	I	I	O	R	v_1
0	I	I	I	R	O	I	u_0
	I	I	I	O	R	I	v_0

(21)

In order to end with one run on T4 and all other tapes empty, we need to have

$$\begin{aligned}
v_0 &= 1,\\
u_0 + v_1 &= 0,\\
u_1 + v_2 &= u_0 + v_0,\\
u_2 + v_3 &= u_1 + v_1 + u_0 + v_0,\\
u_3 + v_4 &= u_2 + v_2 + u_1 + v_1 + u_0 + v_0,\\
u_4 + v_5 &= u_3 + v_3 + u_2 + v_2 + u_1 + v_1 + u_0 + v_0,\\
u_5 + v_6 &= u_4 + v_4 + u_3 + v_3 + u_2 + v_2 + u_1 + v_1,
\end{aligned}$$

etc.; in general, the requirement is that

$$u_n + v_{n+1} = u_{n-1} + v_{n-1} + u_{n-2} + v_{n-2} + u_{n-3} + v_{n-3} + u_{n-4} + v_{n-4} \tag{22}$$

for all $n \geq 0$, if we regard $u_j = v_j = 0$ for all $j < 0$.

There is no unique solution to these equations; indeed, if we let all the u's be zero, we get the usual polyphase merge with one tape wasted! But if we choose $u_n \approx v_{n+1}$, the rewind time will be satisfactorily overlapped.

McAllester suggested taking $u_n = v_{n-1} + v_{n-2} + v_{n-3} + v_{n-4}$, $v_{n+1} = u_{n-1} + u_{n-2} + u_{n-3} + u_{n-4}$, so that the sequence

$$\langle x_0, x_1, x_2, x_3, x_4, x_5, \ldots \rangle = \langle v_0, u_0, v_1, u_1, v_2, u_2, \ldots \rangle$$

satisfies the uniform recurrence $x_n = x_{n-3} + x_{n-5} + x_{n-7} + x_{n-9}$. However, it turns out to be better to let

$$\begin{aligned}
v_{n+1} &= u_{n-1} + v_{n-1} + u_{n-2} + v_{n-2},\\
u_n &= u_{n-3} + v_{n-3} + u_{n-4} + v_{n-4};
\end{aligned} \tag{23}$$

this sequence is not only slightly better in its merging time, it also has the great virtue that the corresponding merging time can be analyzed mathematically. [McAllester's choice is extremely difficult to analyze because runs of different lengths may occur during a single phase; we shall see that this does not happen with (23).]

We can deduce the number of runs on each tape on each level, by working backwards in the pattern (21), and we obtain the following sorting scheme:

Level	T1	T2	T3	T4	T5	T6	Write time	Rewind time
	1^{23}	1^{21}	1^{17}	1^{10}	—	1^{11}	82	23
7	1^{19}	1^{17}	1^{13}	1^{6}	R	$1^{11}4^{4}$	$4 \times 4 = 16$	82 − 23
	1^{13}	1^{11}	1^{7}	—	4^{6}	R	$6 \times 4 = 24$	25
6	1^{10}	1^{8}	1^{4}	R	4^{9}	$1^{8}4^{4}$	$3 \times 4 = 12$	10
	1^{6}	1^{4}	—	4^{4}	R	$1^{4}4^{4}$	$4 \times 4 = 16$	36
5	1^{5}	1^{3}	R	$4^{4}7^{1}$	4^{8}	$1^{3}4^{4}$	$1 \times 7 = 7$	17
	1^{2}	—	7^{3}	R	4^{5}	4^{4}	$3 \times 7 = 21$	23
4	1^{1}	R	$7^{3}13^{1}$	$4^{3}7^{1}$	4^{4}	4^{3}	$1 \times 13 = 13$	21
	—	13^{1}	R	$4^{2}7^{1}$	4^{3}	4^{2}	$1 \times 13 = 13$	34
3	R	$13^{1}19^{1}$	$7^{2}13^{1}$	$4^{1}7^{1}$	4^{2}	4^{1}	$1 \times 19 = 19$	23
	19^{1}	R	$7^{1}13^{1}$	7^{1}	4^{1}	—	$1 \times 19 = 19$	32
2	$19^{1}31^{0}$	$13^{1}19^{1}$	$7^{1}13^{1}$	7^{1}	4^{1}	R	$0 \times 31 = 0$	25
	R	19^{1}	13^{1}	7^{0}	—	31^{1}	$1 \times 31 = 31$	19
1	$19^{1}31^{0}$	19^{1}	13^{1}	7^{0}	R	$31^{1}52^{0}$	$0 \times 52 = 0$	
	$19^{1}31^{0}$	19^{1}	13^{1}	—	52^{0}	R	$0 \times 52 = 0$	max (36, 31, 23)
0	$19^{1}31^{0}$	19^{1}	13^{1}	R	$52^{0}82^{0}$	$31^{1}52^{0}$	$0 \times 82 = 0$	
	(31^{0})	(19^{0})	—	82^{1}	(R)	(52^{0})	$1 \times 82 = 82$	0

Unoverlapped rewinding occurs when the input tape T5 is being rewound (82 units), during the first half of the level 2 phase (25 units), and during the final "dummy merge" phases in levels 1 and 0 (36 units). So we may estimate the time as $273t + 143r$; the corresponding amount for Algorithm D, $268t + 208r$, is almost always inferior.

It is not difficult to see (cf. exercise 23) that the run lengths output during each phase are successively

$$4, 4, 7, 13, 19, 31, 52, 82, 133, \ldots \tag{24}$$

a sequence $\langle t_1, t_2, t_3, \ldots \rangle$ satisfying the law

$$t_n = t_{n-2} + 2t_{n-3} + t_{n-4} \tag{25}$$

if we regard $t_n = 1$ for $n \leq 0$. We can also analyze the optimum placement of dummy runs, by looking at strings of merge numbers as we did for standard polyphase [cf. (8)]:

Level	T1	T2	T3	T4	T6	Final output on
1	1	1	1	1	—	T5
2	1	1	1	—	1	T4
3	21	21	2	2	1	T3
4	2221	222	222	22	2	T2
5	23222	23222	2322	23	222	T1
6	333323222	33332322	333323	3333	2322	T6
. . .	. . .	. . .	. . .	. . .	. . .	. . .
n	A_n	B_n	C_n	D_n	E_n	$T(k)$
$n+1$	$(A''_nE_n + 1)B_n$	$(A''_nE_n + 1)C_n$	$(A''_nE_n + 1)D_n$	$A''_nE_n + 1$	A'_n	$T(k-1)$
. . .	. . .	. . .	. . .	. . .	. . .	. . .

(26)

where $A_n = A_n' A_n''$, and A_n'' consists of the last u_n merge numbers of A_n. The above rule for going from level n to level $n+1$ is valid for *any* scheme satisfying (22). When we define the u's and v's by (23), the strings $A_n, \ldots, E_n$ can be expressed in the following rather simple way [cf. (9)]:

$$\begin{aligned} A_n &= (W_{n-1}W_{n-2}W_{n-3}W_{n-4}) + 1, \\ B_n &= (W_{n-1}W_{n-2}W_{n-3}) + 1, \\ C_n &= (W_{n-1}W_{n-2}) + 1, \\ D_n &= (W_{n-1}) + 1, \\ E_n &= (W_{n-2}W_{n-3}) + 1, \end{aligned} \tag{27}$$

where

$$\begin{aligned} W_n &= (W_{n-3}W_{n-4}W_{n-2}W_{n-3}) + 1 \quad \text{for} \quad n > 0, \\ W_0 &= \text{'0'}, \quad \text{and} \quad W_n = (\text{empty}) \quad \text{for} \quad n < 0. \end{aligned} \tag{28}$$

From these relations it is easy to make a detailed analysis of the six-tape case.

In general, when there are $T \geq 5$ tapes, we let $P = T - 2$, and we define the sequences $\langle u_n \rangle$, $\langle v_n \rangle$ by the rules

$$\begin{aligned} v_{n+1} &= u_{n-1} + v_{n-1} + \cdots + u_{n-r} + v_{n-r}, \\ u_n &= u_{n-r-1} + v_{n-r-1} + \cdots + u_{n-P} + v_{n-P}, \quad \text{for} \quad n \geq 0, \end{aligned} \tag{29}$$

where $r = \lfloor P/2 \rfloor$; $v_0 = 1$, and $u_n = v_n = 0$ for $n < 0$. If $w_n = u_n + v_n$, we therefore have

$$w_n = w_{n-2} + \cdots + w_{n-r} + 2w_{n-r-1} + w_{n-r-2} + \cdots + w_{n-P}, \quad n > 0; \tag{30}$$

$w_0 = 1$; and $w_n = 0$ for $n < 0$. The initial distribution on tapes for level $n+1$ places $w_n + w_{n-1} + \cdots + w_{n-P+k}$ runs on tape k, for $1 \leq k \leq P$, and $w_{n-1} + \cdots + w_{n-r}$ on tape T; tape $T - 1$ is used for input. Then u_n runs are merged to tape T while $T - 1$ is being rewound; v_n are merged to $T - 1$ while T is rewinding; u_{n-1} to $T - 1$ while $T - 2$ is rewinding; etc.

Table 6 shows the approximate behavior of this procedure when S is not too small. The "Pass/Phase" column indicates approximately how much of the entire file is being rewound during each half of a phase, and approximately how much of the file is being written during each full phase. *The tape-splitting method is superior to standard polyphase on six or more tapes*, and probably also on five, at least for large S.

When $T = 4$ the above procedure would become essentially equivalent to balanced two-way merging, *without* overlapping the rewind time, since w_{2n+1} would be 0 for all n. So the entries in Table 6 for $T = 4$ have been obtained by making a slight modification, letting $v_2 = 0$, $u_1 = 1$, $v_1 = 0$, $u_0 = 0$, $v_0 = 1$, and $v_{n+1} = u_{n-1} + v_{n-1}$, $u_n = u_{n-2} + v_{n-2}$ for $n \geq 2$. This leads to a very interesting sorting scheme (see exercises 25 and 26).

Table 6

APPROXIMATE BEHAVIOR OF POLYPHASE MERGE WITH TAPE SPLITTING

Tapes	Phases	Passes	Pass/phase, percent	Growth ratio
4	$2.885 \ln S + 0.000$	$1.443 \ln S + 1.000$	50	1.4142136
5	$2.078 \ln S + 0.232$	$0.929 \ln S + 1.022$	45	1.6180340
6	$2.078 \ln S - 0.170$	$0.752 \ln S + 1.024$	36	1.6180340
7	$1.958 \ln S - 0.408$	$0.670 \ln S + 1.007$	34	1.6663019
8	$2.008 \ln S - 0.762$	$0.624 \ln S + 0.994$	31	1.6454116
9	$1.972 \ln S - 0.987$	$0.595 \ln S + 0.967$	30	1.6604077
10	$2.013 \ln S - 1.300$	$0.580 \ln S + 0.941$	29	1.6433803
20	$2.069 \ln S - 3.164$	$0.566 \ln S + 0.536$	27	1.6214947

EXERCISES

1. [*16*] Figure 69 shows the order in which runs 34 through 65 are distributed to five tapes with Algorithm D; in what order are runs 1 through 33 distributed?

▶ 2. [*21*] True or false: After two merge phases in Algorithm D, i.e., on the second time we reach step D6, all dummy runs have disappeared.

▶ 3. [*22*] Prove that the condition $\mathsf{D}[1] \geq \mathsf{D}[2] \geq \cdots \geq \mathsf{D}[T]$ is always satisfied at the conclusion of step D4. Explain why this condition is important, in the sense that the mechanism of steps D2 and D3 would not work properly otherwise.

4. [*M20*] Derive the generating functions (7).

5. [*HM26*] (E. P. Miles, Jr., 1960.) For all $p \geq 2$, prove that the polynomial $f_p(z) = z^p - z^{p-1} - \cdots - z - 1$ has p distinct roots, of which exactly one has magnitude greater than unity. [*Hint:* Consider the polynomial $z^{p+1} - 2z^p + 1$.]

6. [*HM24*] The purpose of this exercise is to consider how Tables 1, 5, and 6 can be prepared. Assume that we have a merging pattern whose properties are characterized by polynomials $p(z)$ and $q(z)$ in the following way: (1) The number of initial runs present in a "perfect distribution" requiring n merging phases is the coefficient of z^n in $p(z)/q(z)$. (2) The number of initial runs processed during these n merging phases is the coefficient of z^n in $p(z)/q(z)^2$. (3) There is a "dominant root" α of $q(z^{-1})$ such that $q(\alpha^{-1}) = 0$, $q'(\alpha^{-1}) \neq 0$, $p(\alpha^{-1}) \neq 0$, and $q(\beta^{-1}) = 0$ implies that $\beta = \alpha$ or $|\beta| < |\alpha|$.

Prove that there is a number $\epsilon > 0$ such that, if S is the number of runs in a perfect distribution requiring n merging phases, and if ρS initial runs are processed during those phases, we have $n = a \ln S + b + O(S^{-\epsilon})$, $\rho = c \ln S + d + O(S^{-\epsilon})$, where

$$a = (\ln \alpha)^{-1}, \qquad b = -a \ln\left(\frac{p(\alpha^{-1})}{-q'(\alpha^{-1})}\right) - 1, \qquad c = a\frac{\alpha}{-q'(\alpha^{-1})},$$

$$d = \frac{(b+1)\alpha - p'(\alpha^{-1})/p(\alpha^{-1}) + q''(\alpha^{-1})/q'(\alpha^{-1})}{-q'(\alpha^{-1})}.$$

7. [*HM22*] Let α_p be the dominant root of the polynomial $f_p(z)$ in exercise 5. What is the asymptotic behavior of α_p as $p \to \infty$?

8. [*M20*] (E. Netto, 1901.) Let $N_m^{(p)}$ be the number of ways to express m as an ordered sum of the integers $\{1, 2, \ldots, p\}$. For example, when $p = 3$ and $m = 5$, there are 13 ways, namely $1+1+1+1+1 = 1+1+1+2 = 1+1+2+1 = 1+1+3 = 1+2+1+1 = 1+2+2 = 1+3+1 = 2+1+1+1 = 2+1+2 = 2+2+1 = 2+3 = 3+1+1 = 3+2$. Show that $N_m^{(p)}$ is a generalized Fibonacci number.

9. [*M20*] Let $K_m^{(p)}$ be the number of sequences of m 0's and 1's such that there are no p consecutive 1's. For example, when $p = 3$ and $m = 5$ there are 24 ways: 00000, 00001, 00010, 00011, 00100, 00101, 00110, 01000, 01001, . . . , 11011. Show that $K_m^{(p)}$ is a generalized Fibonacci number.

10. [*M27*] (*Generalized Fibonacci number system.*) Prove that every nonnegative integer n has a unique representation as a sum of distinct pth order Fibonacci numbers $F_j^{(p)}$, for $j \geq p$, subject to the condition that no p consecutive Fibonacci numbers are used.

11. [*M24*] Prove that the nth element of the string Q_∞ in (12) is equal to the number of distinct Fibonacci numbers in the fifth-order Fibonacci representation of $n - 1$. (Cf. exercise 10.)

▶ **12.** [*M20*] Find a connection between powers of the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

and the perfect Fibonacci distributions in (1).

▶ **13.** [*22*] Prove the following rather odd property of perfect Fibonacci distributions: When the final output will be on tape number T, the number of runs on each other tape is *odd;* when the final output will be on some tape other than T, the number of runs will be *odd* on that tape, and it will be *even* on the others. [Cf. (1).]

14. [*M35*] Let $T_n(x) = \sum_{k\geq 0} T_{nk}x^k$, where $T_n(x)$ is the polynomial defined in (16). (a) Show that for each k there is a number $n(k)$ such that $T_{1k} \leq T_{2k} \leq \cdots \leq T_{n(k)k} > T_{n(k)+1,k} \geq \cdots$. (b) Given that $T_{n'k'} < T_{nk'}$ and $n' < n$, prove that $T_{n'k} \leq T_{nk}$ for all $k \geq k'$. (c) Prove that there is a nondecreasing sequence $\langle M_n \rangle$ such that $\Sigma_n(S) = \min_{j\geq 1} \Sigma_j(S)$ when $M_n \leq S < M_{n+1}$, but $\Sigma_n(S) > \min_{j\geq 1} \Sigma_j(S)$ when $S \geq M_{n+1}$. [Cf. (19).]

15. [*M43*] Prove or disprove: $\Sigma_{n-1}(m) < \Sigma_n(m)$ implies that $\Sigma_n(m) \leq \Sigma_{n+1}(m) \leq \Sigma_{n+2}(m) \leq \cdots$. [Such a result would greatly simplify the calculation of Table 2.]

16. [*M43*] Determine the asymptotic behavior of the polyphase merge with optimum distribution of dummy runs.

17. [*32*] Prove or disprove: There is a way to disperse runs for an optimum polyphase distribution in such a way that the distribution for $S + 1$ initial runs is formed by adding one run (on an appropriate tape) to the distribution for S initial runs.

18. [*30*] Does the optimum polyphase distribution produce the best possible merging pattern, in the sense that the total number of initial runs processed is minimized, if we insist that the initial runs be placed on at most $T - 1$ of the tapes? (Ignore rewind time.)

19. [*21*] Make a table analogous to (1), for Caron's polyphase sort on six tapes.

20. [*M24*] What generating functions for Caron's polyphase sort on six tapes correspond to (7) and to (16)? What relations, analogous to (9) and (27), define the strings of merge numbers?

21. [*11*] What should appear on level 7 in (26)?

22. [*M21*] Each term of the sequence (24) is approximately equal to the sum of the previous two. Does this phenomenon hold for the remaining numbers of the sequence? Formulate and prove a theorem about $t_n - t_{n-1} - t_{n-2}$.

▶**23.** [*29*] What changes would be made to (25), (27), and (28), if (23) were changed to $v_{n+1} = u_{n-1} + v_{n-1} + u_{n-2}$, $u_n = v_{n-2} + u_{n-3} + v_{n-3} + u_{n-4} + v_{n-4}$?

24. [*HM41*] Compute the asymptotic behavior of the tape-splitting polyphase procedure, when v_{n+1} is defined to be the sum of the first q terms of $u_{n-1} + v_{n-1} + \cdots + u_{n-P} + v_{n-P}$, for various $P = T - 2$ and for $0 \le q \le 2P$. (The text treats only the case $q = 2\lfloor P/2 \rfloor$; cf. exercise 23.)

25. [*19*] Show how the tape-splitting polyphase merge on four tapes, mentioned at the end of this section, would sort 32 initial runs. (Give a phase-by-phase analysis like the 82-run six-tape example in the text.)

26. [*M21*] Analyze the behavior of the tape-splitting polyphase merge on four tapes, when $S = 2^n$ and when $S = 2^n + 2^{n-1}$. (See exercise 25.)

27. [*23*] Once the initial runs have been distributed to tapes in a perfect distribution, the polyphase strategy is simply to "merge until empty." We merge runs from all nonempty input tapes until one of them becomes empty; then we use that tape as the next output tape, and let the previous output tape serve as an input.

Does this merge-until-empty strategy always sort, no matter how the initial runs are distributed, as long as we distribute them onto at least two tapes? (One tape will, of course, be left empty so that it can be the first output tape.)

28. [*M26*] The previous exercise defines a rather large family of merging patterns. Show that polyphase is the *best* of these, in the following sense: If there are six tapes, and if we consider the class of all initial distributions (a, b, c, d, e) such that the merge-until-empty strategy requires n or less phases to sort, then $a + b + c + d + e \le t_n$, where t_n is the corresponding value for polyphase sorting (1).

29. [*M47*] Exercise 28 shows that the polyphase distribution is optimal among all "merge until empty" patterns in the minimum-phase sense. But is it optimal also in the minimum-pass sense?

Let a be relatively prime to b, and assume that $a + b$ is the Fibonacci number F_n. Prove or disprove the following conjecture due to R. M. Karp: The number of initial runs processed during the merge-until-empty pattern starting with distribution (a, b)

is greater than or equal to $((n - 5)F_{n+1} + (2n + 2)F_n)/5$. (The latter figure is achieved when $a = F_{n-1}$, $b = F_{n-2}$.)

30. [*42*] Prepare a table analogous to Table 2, for the tape-splitting polyphase merge.

5.4.3. The Cascade Merge

Another basic pattern, called the "cascade merge," was actually discovered before polyphase [B. K. Betz and W. C. Carter, *ACM Nat'l Conference* **14** (1959), Paper 14]. This approach is illustrated for six tapes and 190 initial runs in the following table, using the notation developed in Section 5.4.2:

	T1	T2	T3	T4	T5	T6	Initial runs processed
Pass 1.	1^{55}	1^{50}	1^{41}	1^{29}	1^{15}	—	190
Pass 2.	—	$1^{5}{*}$	2^{9}	3^{12}	4^{14}	5^{15}	190
Pass 3.	15^{5}	14^{4}	12^{3}	9^{2}	$5^{1}{*}$	—	190
Pass 4.	—	$15^{1}{*}$	29^{1}	41^{1}	50^{1}	55^{1}	190
Pass 5.	190^{1}	—	—	—	—	—	190

A cascade merge, like polyphase, starts out with a "perfect distribution" of runs on tapes, although the rule for perfect distributions is somewhat different from those in Section 5.4.2. Each line in the table represents a complete pass over *all* the data. Pass 2, for example, is obtained by doing a five-way merge from T1, T2, T3, T4, T5 to T6, until T5 is empty (this puts 15 runs of relative length 5 on T6), then a four-way merge from T1, T2, T3, T4 to T5, then a three-way merge to T4, a two-way merge to T3, and finally a one-way merge (a copying operation) from T1 to T2. Pass 3 is obtained in the same way, first doing a five-way merge until one tape becomes empty, then a four-way merge, and so on. (Perhaps the present section of this book should be numbered 5.4.3.2.1 instead of 5.4.3!)

It is clear that the copying operations are unnecessary, and they could be omitted. Actually, however, in the six-tape case this copying takes only a small percentage of the total time. The items marked with an asterisk in the above table are those which were simply copied; only 25 of the 950 runs processed are of this type. Most of the time is devoted to five-way and four-way merging.

At first it may seem that the cascade pattern is a rather poor choice, by comparison with polyphase, since standard polyphase uses $(T - 1)$-way merging throughout while the cascade uses $(T - 1)$-way, $(T - 2)$-way, $(T - 3)$-way, etc. But in fact it is asymptotically *better* than polyphase, on six or more tapes! As we have observed in Section 5.4.2, a high order of merge is not a guarantee of efficiency. Table 1 shows the performance characteristics of cascade merge, by analogy with the similar tables in Section 5.4.2.

Table 1

APPROXIMATE BEHAVIOR OF CASCADE MERGE SORTING

Tapes	Passes (with copying)	Passes (without copying)	Growth Ratio
3	$2.078 \ln S + 0.672$	$1.504 \ln S + 0.992$	1.6180340
4	$1.235 \ln S + 0.754$	$1.102 \ln S + 0.820$	2.2469796
5	$0.946 \ln S + 0.796$	$0.897 \ln S + 0.800$	2.8793852
6	$0.796 \ln S + 0.821$	$0.773 \ln S + 0.808$	3.5133371
7	$0.703 \ln S + 0.839$	$0.691 \ln S + 0.822$	4.1481149
8	$0.639 \ln S + 0.852$	$0.632 \ln S + 0.834$	4.7833861
9	$0.592 \ln S + 0.861$	$0.587 \ln S + 0.845$	5.4189757
10	$0.555 \ln S + 0.869$	$0.552 \ln S + 0.854$	6.0547828
20	$0.397 \ln S + 0.905$	$0.397 \ln S + 0.901$	12.4174426

It is not hard to derive the "perfect distributions" for a cascade merge. With six tapes, they are

Level	T1	T2	T3	T4	T5
0	1	0	0	0	0
1	1	1	1	1	1
2	5	4	3	2	1
3	15	14	12	9	5
4	55	50	41	29	15
5	190	175	146	105	55
. . .	. . .	. . .	. . .	. . .	. . .
n	a_n	b_n	c_n	d_n	e_n
$n+1$	$a_n + b_n + c_n + d_n + e_n$	$a_n + b_n + c_n + d_n$	$a_n + b_n + c_n$	$a_n + b_n$	a_n

(1)

It is interesting to note that the relative magnitudes of these numbers appear also in the diagonals of a regular $(2T - 1)$-sided polygon. For example, the five diagonals in the undecagon of Fig. 73 have relative lengths very nearly equal to 190, 175, 146, 105, and 55! We shall prove this remarkable fact later in this section, and we shall also see that the relative amount of time spent in $(T - 1)$-way merging, $(T - 2)$-way merging, . . . , 1-way merging is approximately proportional to the *squares* of the lengths of these diagonals.

***Initial distribution of runs.** When the actual number of initial runs isn't perfect, we can insert dummy runs as usual. A superficial analysis of this situation indicates that the method of dummy run assignment is immaterial, since cascade merging operates by complete passes; if we have 190 initial runs, each record is processed five times as in the above example, but if there are 191 we must apparently go up a level so that every record is processed six times.

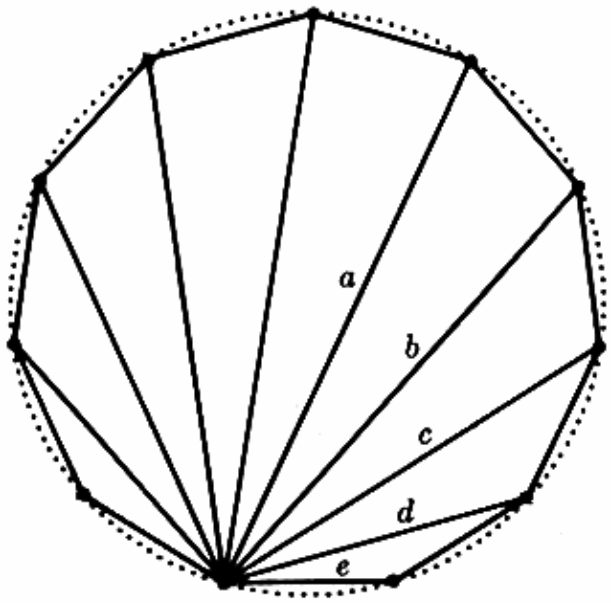


Fig. 73. Geometrical interpretation of cascade numbers.

Fortunately this abrupt change is not actually necessary; David E. Ferguson has found a way to distribute initial runs so that many of the operations during the first merge pass reduce to copying the contents of a tape. When such copying relations are bypassed (by simply changing "logical" tape unit numbers relative to the "physical" numbers as in Algorithm 5.4.2D), we obtain a relatively smooth transition from level to level, as shown in Fig. 74.

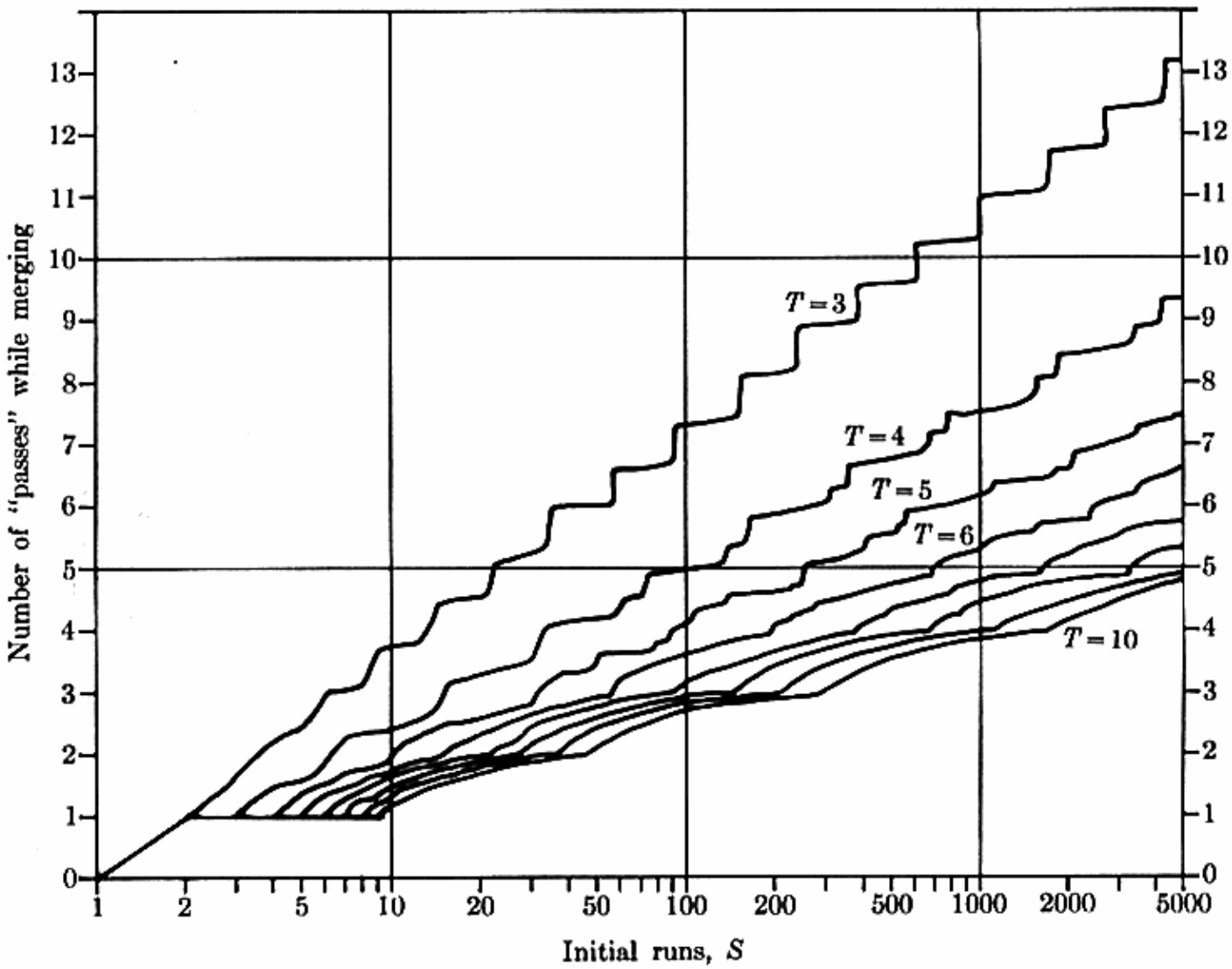


Fig. 74. Efficiency of cascade merge with the distribution of Algorithm D.

Suppose that (a, b, c, d, e) is a perfect distribution, where $a \geq b \geq c \geq d \geq e$. By redefining the correspondence between logical and physical tape units, we can imagine that the distribution is actually (e, d, c, b, a), with a runs on T5, b on T4, etc. The next perfect distribution is $(a + b + c + d + e, a + b + c + d, a + b + c, a + b, a)$; and if the input is exhausted before we reach this next level, let us assume that the tapes contain, respectively, $(D_1, D_2, D_3, D_4, D_5)$ dummy runs, where

$$D_1 \leq a + b + c + d, \qquad D_2 \leq a + b + c, \qquad D_3 \leq a + b,$$
$$D_4 \leq a, \qquad D_5 = 0; \qquad D_1 \geq D_2 \geq D_3 \geq D_4 \geq D_5. \tag{2}$$

We are free to imagine that the dummy runs appear in any convenient place on the tapes. The first merge pass is supposed to produce a runs by five-way merging, then b by four-way merging, etc., and our goal is to arrange the dummies so as to replace merging by copying. It is convenient to do the first merge pass as follows:

1. If $D_4 = a$, subtract a from each of D_1, D_2, D_3, D_4 and pretend that T5 is the result of the merge. If $D_4 < a$, merge a runs from tapes T1 through T5, using the minimum possible number of dummies on tapes T1 through T5 so that the new values of D_1, D_2, D_3, D_4 will satisfy

$$D_1 \leq b + c + d, \qquad D_2 \leq b + c, \qquad D_3 \leq b,$$
$$D_4 = 0; \qquad D_1 \geq D_2 \geq D_3 \geq D_4. \tag{3}$$

Thus, if D_2 was originally $\leq b + c$, we use no dummies from it at this step, while if $b + c < D_2 \leq a + b + c$ we use exactly $D_2 - b - c$ of them.

2. (This step is similar to step 1, but "shifted.") If $D_3 = b$, subtract b from each of D_1, D_2, D_3 and pretend that T4 is the result of the merge. If $D_3 < b$, merge b runs from tapes T1 through T4, reducing the number of dummies if necessary in order to make

$$D_1 \leq b + c, \qquad D_2 \leq b, \qquad D_3 = 0; \qquad D_1 \geq D_2 \geq D_3.$$

3. And so on.

Ferguson's method of distributing runs to tapes can be illustrated by considering the process of going from level 3 to level 4 in (1). Assume that "logical" tapes (T1, . . . , T5) contain respectively (5, 9, 12, 14, 15) runs, and that we want eventually to bring this up to (55, 50, 41, 29, 15). The procedure may be summarized as shown at the top of the facing page.

	Add to T1	Add to T2	Add to T3	Add to T4	Add to T5	"Amount saved"
Step (1, 1)	9	0	0	0	0	15 + 14 + 12 + 5
Step (2, 2)	3	12	0	0	0	15 + 14 + 9 + 5
Step (2, 1)	9	0	0	0	0	15 + 14 + 5
Step (3, 3)	2	2	14	0	0	15 + 12 + 5
Step (3, 2)	3	12	0	0	0	15 + 9 + 5
Step (3, 1)	9	0	0	0	0	15 + 5
Step (4, 4)	1	1	1	15	0	14 + 5
Step (4, 3)	2	2	14	0	0	12 + 5
Step (4, 2)	3	12	0	0	0	9 + 5
Step (4, 1)	9	0	0	0	0	5

We first put nine runs on T1, then (3, 12) on T1 and T2, etc. If the input becomes exhausted during, say, Step (3, 2), then the "amount saved" is $15 + 9 + 5$, meaning that the five-way merge of 15 runs, the two-way merge of 9 runs, and the one-way merge of 5 runs, are avoided by the dummy run assignment. In other words, $15 + 9 + 5$ of the runs present at level 3 are not processed during the first merge phase.

The following algorithm defines the process in detail.

Algorithm C (*Cascade merge sorting with special distribution*). This algorithm takes initial runs and disperses them to tapes, one run at a time, until the supply of initial runs is exhausted. Then it specifies how the tapes are to be merged, assuming that there are $T \geq 3$ available tape units, using at most $(T - 1)$-way merging and avoiding unnecessary one-way merging. Tape T may be used to hold the input, since it does not receive any initial runs. The following tables are maintained:

- A[j], $1 \leq j \leq T$: The perfect cascade distribution we have most recently reached.
- AA[j], $1 \leq j \leq T$: The perfect cascade distribution we are striving for.
- D[j], $1 \leq j \leq T$: Number of dummy runs assumed to be present on logical tape unit number j.
- M[j], $1 \leq j \leq T$: Maximum number of dummy runs desired on logical tape unit number j.
- TAPE[j], $1 \leq j \leq T$: Number of the physical tape unit corresponding to logical tape unit number j.

C1. [Initialize.] Set A[k] ← AA[k] ← D[k] ← 0 for $2 \leq k \leq T$; and set A[1] ← 0, AA[1] ← 1, D[1] ← 1. Set TAPE[k] ← k for $1 \leq k \leq T$. Finally set $i \leftarrow T - 2$, $j \leftarrow 1$, $k \leftarrow 1$, $l \leftarrow 0$, $m \leftarrow 1$, and go to step C5. (This maneuvering is one way to get everything started, by jumping right into the inner loop with appropriate settings of the control variables.)

C2. [Begin new level.] (We have just reached a perfect distribution, and since there is more input we must get ready for the next level.) Increase l by 1.

Set A[k] ← AA[k], for $1 \le k \le T$; then set AA[$T - k$] ← AA[$T - k + 1$] + A[k], for $k = 1, 2, \ldots, T - 1$ in this order. Set (TAPE[1], TAPE[2], . . . , TAPE[$T - 1$]) ← (TAPE[$T - 1$], . . . , TAPE[2], TAPE[1]), and set D[k] ← AA[$k + 1$] for $1 \le k < T$. Finally set $i \leftarrow 1$.

C3. [Begin ith sublevel.] Set $j \leftarrow i$. (The variables i and j represent "Step (i, j)" in the example shown on page 293.)

C4. [Begin Step (i, j).] Set $k \leftarrow j$ and $m \leftarrow$ A[$T - j - 1$]. If $m = 0$ and $i = j$, set $i \leftarrow T - 2$ and return to C3; if $m = 0$ and $i \ne j$, return to C2. (Variable m represents the number of runs to be written onto TAPE[k]; $m = 0$ occurs only when $l = 1$.)

C5. [Input to TAPE[k].] Write one run on tape number TAPE[k], and decrease D[k] by 1. Then if the input is exhausted, rewind all the tapes and go to step C7.

C6. [Advance.] Decrease m by 1. If $m > 0$, return to C5. Otherwise decrease k by 1; if $k > 0$, set $m \leftarrow$ A[$T - j - 1$] − A[$T - j$] and return to C5. Otherwise decrease j by 1; if $j > 0$, go to C4. Otherwise increase i by 1; if $i < T - 1$, return to C3. Otherwise go to C2.

C7. [Prepare to merge.] (At this point the initial distribution is complete, and the A, AA, D, and TAPE tables describe the present states of the tapes.) Set M[k] ← AA[$k + 1$] for $1 \le k < T$, and set FIRST ← 1. (Variable FIRST is nonzero only during the first merge pass.)

C8. [Cascade.] If $l = 0$, stop; sorting is complete and the output is on TAPE[1]. Otherwise, for $p = T - 1, T - 2, \ldots, 1$, in this order, do a p-way merge from TAPE[1], . . . , TAPE[p] to TAPE[$p + 1$] as follows:

If $p = 1$, simulate the one-way merge by simply rewinding TAPE[2], then interchanging TAPE[1] ↔ TAPE[2].
Otherwise if FIRST = 1 and D[$p - 1$] = M[$p - 1$], simulate the p-way merge by simply interchanging TAPE[p] ↔ TAPE[$p + 1$], rewinding TAPE[p], and subtracting M[$p - 1$] from each of D[1], . . . , D[$p - 1$], M[1], . . . , M[$p - 1$].
Otherwise, subtract M[$p - 1$] from each of M[1], . . . , M[$p - 1$]. Then merge one run from each TAPE[j] such that $1 \le j \le p$ and D[j] ≤ M[j]; subtract one from each D[j] such that $1 \le j \le p$ and D[j] > M[j]; and put the output run on TAPE[$p + 1$]. Continue doing this until TAPE[p] is empty. Then rewind TAPE[p] and TAPE[$p + 1$].

C9. [Down a level.] Decrease l by 1, set FIRST ← 0, and set (TAPE[1], . . . , TAPE[T]) ← (TAPE[T], . . . , TAPE[1]). (At this point all D's and M's are zero and will remain so.) Return to C8. ∎

Steps C1–C6 of this algorithm do the distribution, and steps C7–C9 do the merging; the two parts are fairly independent of each other, and it would be possible to store M[k] and AA[$k + 1$] in the same memory locations.

***Analysis of cascade merging.** The cascade merge is somewhat harder to analyze than polyphase, but the analysis is especially interesting because so

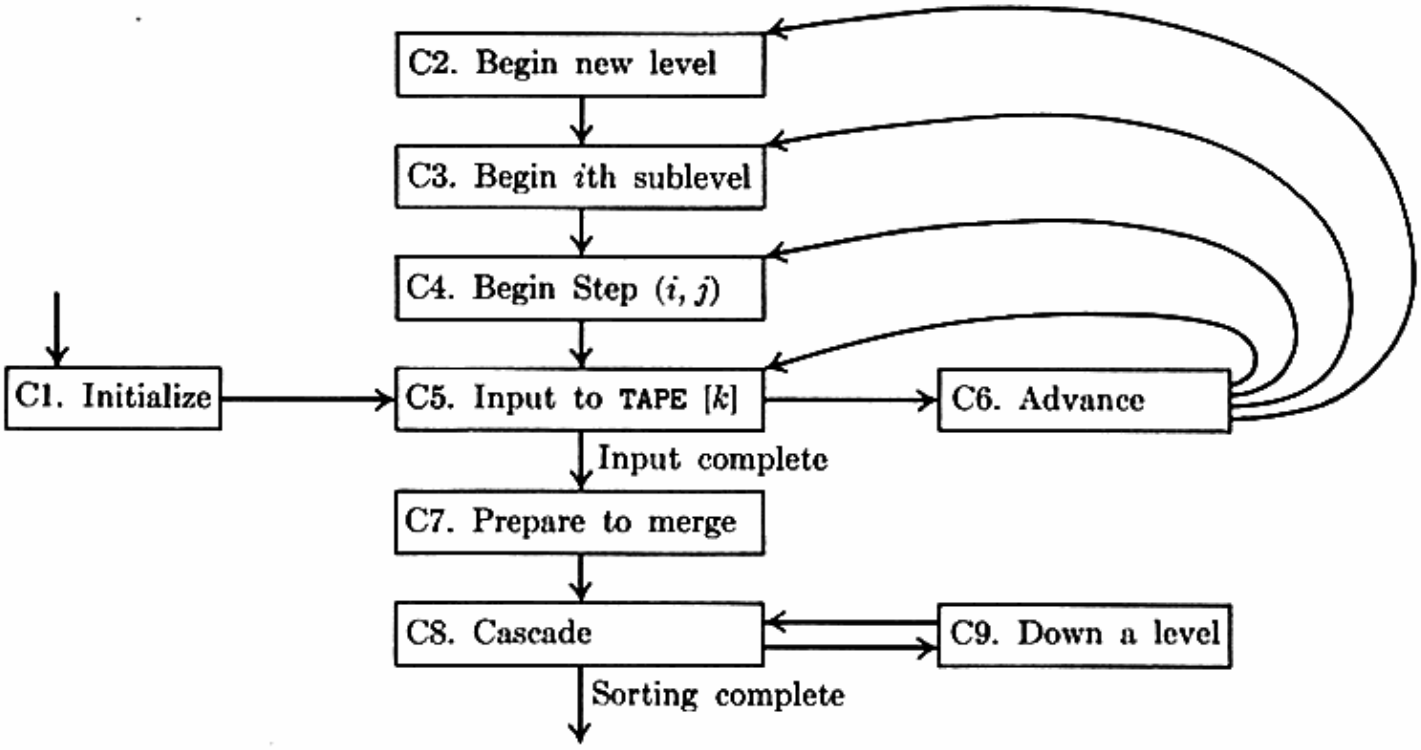


Fig. 75. The cascade merge, with special distribution.

many remarkable formulas are present. Readers who enjoy discrete mathematics are urged to study the cascade distribution for themselves, *before reading further*, since the numbers have so many extraordinary properties which are a pleasure to discover! We shall discuss here one of the many ways to approach the analysis, emphasizing the way in which the results might be discovered.

For convenience, let us consider the six-tape case, looking for formulas which generalize to all T. Relations (1) lead to the first basic pattern:

$$\begin{aligned}
a_n &= a_n &&= \binom{0}{0} a_n,\\
b_n &= a_n - c_{n-1} = a_n - a_{n-2} &&= \binom{1}{0} a_n - \binom{2}{2} a_{n-2},\\
c_n &= b_n - d_{n-1} = b_n - a_{n-2} - b_{n-2} &&= \binom{2}{0} a_n - \binom{3}{2} a_{n-2} + \binom{4}{4} a_{n-4},\\
d_n &= c_n - e_{n-1} = c_n - a_{n-2} - b_{n-2} - c_{n-2} &&= \binom{3}{0} a_n - \binom{4}{2} a_{n-2} + \binom{5}{4} a_{n-4} - \binom{6}{6} a_{n-6},\\
e_n &= d_n - b_{n-1} = d_n - a_{n-2} - b_{n-2} - c_{n-2} - d_{n-2} &&= \binom{4}{0} a_n - \binom{5}{2} a_{n-2} + \binom{6}{4} a_{n-4} - \binom{7}{6} a_{n-6} + \binom{8}{8} a_{n-8}.
\end{aligned} \tag{4}$$

Let $A(z) = \sum_{n\geq 0} a_n z^n, \ldots, E(z) = \sum_{n\geq 0} e_n z^n$, and define the polynomials

$$\begin{aligned}
q_m(z) &= \binom{m}{0} - \binom{m+1}{2} z^2 + \binom{m+2}{4} z^4 - \cdots\\
&= \sum_{k\geq 0} \binom{m+k}{2k} (-1)^k z^{2k} = \sum_{0\leq k\leq m} \binom{2m-k}{k} (-1)^{m-k} z^{2m-2k}.
\end{aligned} \tag{5}$$

The result of (4) can be summarized by saying that $B(z) - q_1(z)A(z), \ldots, E(z) - q_4(z)A(z)$ reduce to finite sums corresponding to the boundary conditions, namely the values of $a_{-1}, a_{-2}, a_{-3}, \ldots$ which appear in (4) for small n

but not in $A(z)$. In order to supply appropriate boundary conditions, let us run the recurrence backwards to negative levels, through level -8:

n	a_n	b_n	c_n	d_n	e_n
0	1	0	0	0	0
-1	0	0	0	0	1
-2	1	-1	0	0	0
-3	0	0	0	-1	2
-4	2	-3	1	0	0
-5	0	0	1	-4	5
-6	5	-9	5	-1	0
-7	0	-1	6	-14	14
-8	14	-28	20	-7	1

(On seven tapes the table would be similar, with entries for odd n shifted right one column.) The sequence $a_0, a_{-2}, a_{-4}, \ldots = 1, 1, 2, 5, 14, \ldots$ is a dead giveaway for computer scientists, since it occurs in connection with so many recursive algorithms (e.g. exercise 2.2.1–4 and Eq. 2.3.4.4–13); we therefore conjecture that in the T-tape case

$$\begin{aligned} a_{-2n} &= \binom{2n}{n}\frac{1}{n+1}, \qquad &&\text{for} \quad 0 \le n \le T-2; \\ a_{-2n-1} &= 0, &&\text{for} \quad 0 \le n \le T-3. \end{aligned} \tag{6}$$

To verify that this choice is correct, it suffices to show that (6) and (4) yield the correct results for levels 0 and 1. On level 1 this is obvious, and on level 0 we have to verify that

$$\binom{m}{0}a_0 - \binom{m+1}{2}a_{-2} + \binom{m+2}{4}a_{-4} - \binom{m+3}{6}a_{-6} + \cdots$$
$$= \sum_{k\ge 0}\binom{m+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1} = \delta_{m0} \tag{7}$$

for $0 \le m \le T-2$. Fortunately this sum can be evaluated by standard techniques (it is "Problem 2," one of the basic examples in the text of Section 1.2.6).

Now we can compute the coefficients of $B(z) - q_1(z)A(z)$, etc. For example, consider the coefficient of z^{2m} in $D(z) - q_3(z)A(z)$: it is

$$\begin{aligned} \sum_{k\ge 0}\binom{3+m+k}{2m+2k}(-1)^{m+k}a_{-2k} &= \sum_{k\ge 0}\binom{3+m+k}{2m+2k}\binom{2k}{k}\frac{(-1)^{m+k}}{k+1} \\ &= (-1)^m\left(\binom{2+m}{2m-1} - \binom{3+m}{2m}\right) \\ &= (-1)^{m+1}\binom{2+m}{2m}, \end{aligned}$$

by the result of "Problem 3" in Section 1.2.6. Therefore we have deduced that

$$\begin{aligned} A(z) &= q_0(z)A(z), \\ B(z) &= q_1(z)A(z) - q_0(z), \\ C(z) &= q_2(z)A(z) - q_1(z), \\ D(z) &= q_3(z)A(z) - q_2(z), \\ E(z) &= q_4(z)A(z) - q_3(z). \end{aligned} \tag{8}$$

Furthermore we have $e_{n+1} = a_n$; hence $zA(z) = E(z)$, and

$$A(z) = q_3(z)/(q_4(z) - z). \tag{9}$$

The generating functions have now been derived in terms of the q polynomials, and so we want to understand the q's better. Exercise 1.2.9–15 is useful in this regard, since it gives us a "closed form" which may be written

$$q_m(z) = \frac{((\sqrt{4 - z^2} + iz)/2)^{2m+1} + ((\sqrt{4 - z^2} - iz)/2)^{2m+1}}{\sqrt{4 - z^2}}. \tag{10}$$

Everything simplifies if we now set $z = 2 \sin \theta$:

$$\begin{aligned} q_m(2 \sin \theta) &= \frac{(\cos \theta + i \sin \theta)^{2m+1} + (\cos \theta - i \sin \theta)^{2m+1}}{2 \cos \theta} \\ &= \frac{\cos (2m + 1)\theta}{\cos \theta}. \end{aligned} \tag{11}$$

(This coincidence leads us to suspect that the polynomial $q_m(z)$ is well known in mathematics; and indeed, a glance at appropriate tables will show that $q_m(z)$ is essentially a Chebyshev polynomial of the second kind, namely $(-1)^m U_{2m}(z/2)$ in conventional notation.)

We can now determine the roots of the denominator in (9): $q_4(2 \sin \theta) = 2 \sin \theta$ reduces to

$$\cos 9\theta = 2 \sin \theta \cos \theta = \sin 2\theta.$$

We can obtain solutions to this relation whenever $\pm 9\theta = 2\theta + (2n - \frac{1}{2})\pi$; and all such θ yield roots of the denominator in (9) provided that $\cos \theta \neq 0$. (When $\cos \theta = 0$, $q_m(\pm 2) = \pm(2m + 1)$ is never equal to ± 2.) The following eight distinct roots are therefore obtained:

$$q_4(z) - z = 0 \qquad \text{when} \qquad z = 2 \sin \frac{-5}{14} \pi,\ 2 \sin \frac{-1}{14} \pi,\ 2 \sin \frac{3}{14} \pi;$$

$$2 \sin \frac{-7}{22} \pi,\ 2 \sin \frac{-3}{22} \pi,\ 2 \sin \frac{1}{22} \pi,\ 2 \sin \frac{5}{22} \pi,\ 2 \sin \frac{9}{22} \pi.$$

Since $q_4(z)$ is a polynomial of degree 8, this accounts for all the roots. The first three of these values make $q_3(z) = 0$, so $q_3(z)$ and $q_4(z) - z$ have a polynomial of degree three as a common factor. The other five roots govern the asymptotic behavior of the coefficients of $A(z)$, if we expand (9) in partial fractions.

Considering the general T-tape case, let $\theta_k = (4k + 1)\pi/(4T - 2)$. The generating function $A(z)$ for the T-tape cascade distribution numbers takes the form

$$\frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \frac{\cos^2 \theta_k}{1 - z/(2 \sin \theta_k)} \tag{12}$$

(see exercise 8); hence

$$a_n = \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos^2 \theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n. \tag{13}$$

Eqs. (8) now lead to the similar formulas

$$\begin{aligned} b_n &= \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos \theta_k \cos 3\theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n, \\ c_n &= \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos \theta_k \cos 5\theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n, \\ d_n &= \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos \theta_k \cos 7\theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n, \end{aligned} \tag{14}$$

and so on. Exercise 9 shows that these equations hold for all $n \geq 0$, not only for large n. In each sum the term for $k = 0$ dominates all the others, especially when n is reasonably large; therefore the "growth ratio" is

$$\frac{1}{2 \sin \theta_0} = \frac{2}{\pi} T - \frac{1}{\pi} + \frac{\pi}{48T} + O(T^{-2}). \tag{15}$$

Cascade sorting was first analyzed by W. C. Carter [*Proc. IFIP Congress* (1962), 62–66], who obtained numerical results for small T, and by David E. Ferguson [see *CACM* **7** (1964), 297], who discovered the first two terms in the asymptotic behavior (15) of the growth ratio. During the summer of 1964, R. W. Floyd discovered the explicit form $1/(2 \sin \theta_0)$ of the growth ratio, so that exact formulas could be used for all T. An intensive analysis of the cascade numbers was independently carried out by G. N. Raney [*Canadian J. Math.* **18** (1966), 332–349], who came across them in quite another way having nothing to do with sorting. Raney observed the "ratio of diagonals" principle of Fig. 73, and derived many other interesting properties of the numbers. Floyd and Raney used matrix manipulations in their proofs (cf. exercise 6).

Modifications of cascade sorting. If one more tape is added, it is possible to overlap nearly all of the rewind time during a cascade sort. For example, we can merge T1–T5 to T7, then T1–T4 to T6, then T1–T3 to T5 (which by now is rewound), then T1–T2 to T4, and the next pass can begin when the comparatively short data on T4 has been rewound. The efficiency of this process can be predicted from the above analysis of cascade. (See Section 5.4.6 for further information.)

A "compromise merge" scheme, which includes both polyphase and cascade as special cases, was suggested by D. E. Knuth in *CACM* **6** (1963), 585–587. Each phase consists of $(T - 1)$-way, $(T - 2)$-way, . . . , P-way merges, where P is any fixed number between 1 and $T - 1$. When $P = T - 1$, this is polyphase, and when $P = 1$ it is "pure cascade"; when $P = 2$ it is cascade without copy phases. Analyses of this scheme have been made by C. E. Radke [*IBM Systems J.* **5** (1966), 226–247] and by W. H. Burge [*Proc. IFIP Congress* (1971), to appear]. Burge found the generating function $\sum T_n(x)z^n$ for each (P, T) compromise merge, generalizing Eq. 5.4.2–16; he showed that the best value of P, from the standpoint of least initial runs processed as a function of S as $S \to \infty$ (using a straightforward distribution scheme and ignoring rewind time), is respectively (2, 3, 3, 4, 4, 4, 3, 3, 4) for $T =$ (3, 4, 5, 6, 7, 8, 9, 10, 11). These values of P lean more towards cascade than polyphase as T increases; and it turns out that the compromise merge is never substantially better than cascade itself. On the other hand, with an optimum choice of levels and distribution of dummy runs, as described in Section 5.4.2, pure polyphase seems to be best of all the compromise merges; unfortunately the optimum distribution is comparatively difficult to implement.

Th. L. Johnsen [*BIT* **6** (1966), 129–143] has studied a combination of balanced and polyphase merging; a rewind-overlap variation of balanced merging has been proposed by M. A. Goetz [*Digital Computer User's Handbook*, ed. by M. Klerer and G. A. Korn (New York: McGraw-Hill, 1967), 1.311–1.312]; and many other hybrid schemes can be imagined.

EXERCISES

1. [*10*] Using Table 1, compare cascade merging with the "tape-splitting" version of polyphase described in Section 5.4.2. Which is better? (Ignore rewind time.)

▶ **2.** [*22*] Compare cascade sorting on three tapes, using Algorithm C, to polyphase sorting on three tapes, using Algorithm 5.4.2D. What similarities and differences are there?

3. [*20*] Prepare a table which shows what happens when 100 initial runs are sorted on six tapes using Algorithm C.

4. [*M20*] (G. N. Raney.) An "nth level cascade distribution" is a multiset defined as follows (in the case of six tapes): $\{1, 0, 0, 0, 0\}$ is a 0th level cascade distribution; and if $\{a, b, c, d, e\}$ is an nth level cascade distribution, $\{a + b + c + d + e, a + b +$

$c+d,\ a+b+c,\ a+b,\ a\}$ is an $(n+1)$st level cascade distribution. (Since a multiset is unordered, up to 5! different $(n+1)$st level distributions can be formed from a single nth level distribution.) (a) Prove that *any* multiset $\{a, b, c, d, e\}$ of relatively prime integers is an nth level cascade distribution, for some n. (b) Prove that the distribution defined for cascade sorting is *optimum*, in the sense that, if $\{a, b, c, d, e\}$ is any nth level distribution with $a \geq b \geq c \geq d \geq e$, we have $a \leq a_n$, $b \leq b_n$, $c \leq c_n$, $d \leq d_n$, $e \leq e_n$, where $\{a_n, b_n, c_n, d_n, e_n\}$ is the distribution defined in (1).

▶ **5.** [*20*] Prove that the cascade numbers defined in (1) satisfy the law

$$a_k a_{n-k} + b_k b_{n-k} + c_k c_{n-k} + d_k d_{n-k} + e_k e_{n-k} = a_n, \qquad \text{for} \qquad 0 \leq k \leq n.$$

[*Hint:* Interpret this relation by considering how many runs of various lengths are output during the kth pass of a complete cascade sort.]

6. [*M20*] Find a 5×5 matrix Q such that the first row of Q^n contains the six-tape cascade numbers $a_n\ b_n\ c_n\ d_n\ e_n$, for all $n \geq 0$.

7. [*M20*] Given that cascade merge is being applied to a perfect distribution of a_n initial runs, find a formula for the amount of processing saved when one-way merging is suppressed.

8. [*HM23*] Derive (12).

9. [*HM26*] Derive (14).

▶ **10.** [*M28*] Instead of using the pattern (4) to begin the study of the cascade numbers, start with the identities

$$\begin{aligned}
e_n &= a_{n-1} &&= \binom{1}{1} a_{n-1}, \\
d_n &= 2a_{n-1} - e_{n-2} &&= \binom{2}{1} a_{n-1} - \binom{3}{3} a_{n-3}, \\
c_n &= 3a_{n-1} - d_{n-2} - 2e_{n-2} &&= \binom{3}{1} a_{n-1} - \binom{4}{3} a_{n-3} + \binom{5}{5} a_{n-5},
\end{aligned}$$

etc. Letting

$$r_m(z) = \binom{m}{1} z - \binom{m+1}{3} z^3 + \binom{m+2}{5} z^5 - \cdots,$$

express $A(z)$, $B(z)$, etc. in terms of these r polynomials.

11. [*M38*] Let

$$f_m(z) = \sum_{0 \leq k \leq m} \binom{\lfloor (m+k)/2 \rfloor}{k} (-1)^{\lceil k/2 \rceil} z^k.$$

Prove that the generating function $A(z)$ for the T-tape cascade numbers is equal to $f_{T-3}(z)/f_{T-1}(z)$, where the numerator and denominator in this expression have no common factor.

12. [*M40*] Prove that Ferguson's distribution scheme is optimum, in the sense that no method of placing the dummy runs, satisfying (2), will cause less initial runs to be processed during the first pass, *provided* that the strategy of steps C7–C9 is used during this pass.

13. [*40*] The text suggests overlapping most of the rewind time, by adding an extra tape. Explore this idea. (For example, the text's scheme involves waiting for T4 to rewind; would it be better to omit T4 from the first merge phase of the next pass?)

5.4.4. Reading Tape Backwards

Many magnetic tape units have the ability to read tape in the opposite direction from which it was written. The merging patterns we have encountered so far always write information onto tape in the "forward" direction, then rewind the tape, read it forwards, and rewind again. (The tape files therefore behave as queues, operating in a first-in-first-out manner.) Backwards reading allows us to eliminate both of these rewind operations: We write the tape forwards and read it backwards. (In this case the files behave as stacks, since they are used in a last-in-first-out manner.)

The balanced, polyphase, and cascade merge patterns can all be adapted to backward reading. The main difference is that *merging reverses the order of the runs* when we read backward and write forward. If two runs are in ascending order on tape, we can merge them while reading backwards, but this produces descending order. The descending runs produced in this way will subsequently become ascending on the next pass; so the merging algorithms must be capable of dealing with runs in either order. A programmer confronted with read-backwards for the first time will often find himself standing on his head!

As an example of backwards reading, consider the process of merging 8 initial runs, using a *balanced* merge on four tapes. The operations can be summarized as follows:

	T1	T2	T3	T4	
Pass 1.	$A_1A_1A_1A_1$	$A_1A_1A_1A_1$	—	—	Initial distribution.
Pass 2.	—	—	D_2D_2	D_2D_2	Merge to T3 and T4.
Pass 3.	A_4	A_4	—	—	Merge to T1 and T2.
Pass 4.	—	—	D_8	—	Final merge to T3.

Here A_r stands for a run which appears on tape in ascending order, if the tape is read forwards as in our previous examples, having relative length r; D_r is the corresponding notation for a descending run of length r. During Pass 2 the ascending runs become descending: they appear to be descending in the input since we are reading backwards. They switch orientation again on Pass 3.

Note that the above process finishes with the result on tape T3, in *descending* order. If this is bad (depending on whether the output is to be read backwards, or to be dismounted and put away for future use), we could copy it to another tape, reversing the direction. A faster way would be to rewind T1 and T2 after Pass 3, producing A_8 during Pass 4. Still faster would be to start with eight *descending* runs during Pass 1, since this would interchange all the A's and D's. However, the balanced merge on 16 initial runs would

require the initial runs to be ascending; and since we usually don't know in advance how many initial runs will be formed, it is necessary to choose one consistent direction. Therefore the idea of rewinding after Pass 3 is probably best.

The *cascade* merge carries over in the same way. For example, consider sorting 14 initial runs on four tapes:

	T1	T2	T3	T4
Pass 1.	$A_1A_1A_1A_1A_1A_1$	$A_1A_1A_1A_1A_1$	$A_1A_1A_1$	—
Pass 2.	—	D_1	D_2D_2	$D_3D_3D_3$
Pass 3.	A_6	A_5	A_3	—
Pass 4.	—	—	—	D_{14}

Again, we could produce A_{14} instead of D_{14}, if we rewound T1, T2, T3 just before the final pass. Note that this is a "pure" cascade merge, in the sense that all of the one-way merges have explicitly been performed. If we had suppressed the copying operations, as in Algorithm 5.4.3D, we would have been confronted with the situation

A_1	—	D_2D_2	$D_3D_3D_3$

after Pass 2, and it would have been impossible to continue with a three-way merge since we cannot merge runs that are in opposite directions! The operation of copying T1 to T2 could be avoided if we rewound T1 and proceeded to read it forward during the next merge phase (while reading T3 and T4 backwards). But it would then be necessary to rewind T1 again after merging, so this trick trades one copy for two rewinds.

Thus the distribution method of Algorithm 5.4.3C does not work as efficiently for read-backwards as for read-forwards; the amount of time required jumps rather sharply every time the number of initial runs passes a "perfect" cascade distribution number. Another dispersion technique can be used to give a smoother transition between perfect cascade distributions (see exercise 17).

Read-backward polyphase. At first glance (and even at second and third glance!), the polyphase merge scheme seems to be totally unfit for reading backwards. For example, suppose that we have 13 initial runs and three tapes:

	T1	T2	T3
Phase 1.	$A_1A_1A_1A_1A_1$	$A_1A_1A_1A_1A_1A_1A_1A_1$	—
Phase 2.	—	$A_1A_1A_1$	$D_2D_2D_2D_2D_2$.

Now we're stuck; we could rewind either T2 or T3 and then read it forwards, while reading the other tape backwards, but this would get things rather mixed up and we would have gained comparatively little by reading backwards.

An ingenious idea which saves the situation is to *alternate the direction of runs on each tape.* Then the merging can proceed in perfect synchronization:

Phase 1.	$A_1D_1A_1D_1A_1$	$D_1A_1D_1A_1D_1A_1D_1A_1$	—
Phase 2.	—	$D_1A_1D_1$	$D_2A_2D_2A_2D_2$
Phase 3.	$A_3D_3A_3$	—	D_2A_2
Phase 4.	A_3	D_5A_5	—
Phase 5.	—	D_5	D_8
Phase 6.	A_{13}	—	—

This principle was mentioned briefly by R. L. Gilstad in his original article on polyphase merging, and he described it more fully in *CACM* **6** (1963), 220–223.

The $ADA\cdots$ technique works properly for polyphase merging on *any* number of tapes; for we can show that the A's and D's will be properly synchronized at each phase, provided only that the initial distribution pass produces alternating A's and D's on each tape and that each tape ends with A (or each tape ends with D): Since the last run written on the output file during one phase is in the opposite direction from the last runs used from the input files, the next phase always finds its runs in the proper orientation. Furthermore we have seen in exercise 5.4.2–13 that most of the perfect Fibonacci distributions call for an *odd* number of runs on one tape (the eventual output tape), and an *even* number of runs on each other tape. If T1 is designated as the final output tape, we can therefore guarantee that all tapes end with an A run, if we start T1 with an A and let the remaining tapes start with a D. A distribution method analogous to Algorithm 5.4.2D can be used, modified so that the distributions on each level have T1 as the final output tape. (We skip levels $1, T+1, 2T+1, \ldots,$ since they are the levels in which the initially empty tape is the final output tape.) For example, in the six-tape case, we can use the following distribution numbers in place of 5.4.2–(1):

Level	T1	T2	T3	T4	T5	Total	Final output will be on	
0	1	0	0	0	0	1	T1	
2	1	2	2	2	2	9	T1	
3	3	4	4	4	2	17	T1	
4	7	8	8	6	4	33	T1	(1)
5	15	16	14	12	8	65	T1	
6	31	30	28	24	16	129	T1	
8	61	120	116	108	92	497	T1	

Thus, T1 always get an odd number of runs, while T2 through T5 get the even numbers, in decreasing order for flexibility in dummy run assignment. Such a distribution has the advantage that the final output tape is known in advance, regardless of the number of initial runs that happen to be present. It turns

out (see exercise 3) that the output will always appear in *ascending* order on T1 when this scheme is used.

Another way to handle the distribution for read-backward polyphase has been suggested by D. T. Goodwin and J. L. Venn [*CACM* 7 (1964), 315]. We can distribute runs almost as in Algorithm 5.4.2D, beginning with a D run on each tape. When the input is exhausted, a dummy A run is imagined to be at the beginning of the unique "odd" tape, unless a distribution with all odd numbers has been reached. Other dummies are imagined at the end of the tapes, or grouped into pairs in the middle. The question of optimum placement of dummy runs is analyzed in exercise 5 below.

Optimum merge patterns. So far we have been discussing various patterns for merging on tape, without asking for "best possible" methods. It appears to be quite difficult to determine the optimal patterns, especially in the read-forward case where the interaction of rewind time with merge time is hard to handle. On the other hand, when merging is done by reading backwards and writing forwards, all rewinding is essentially eliminated, and it is possible to get a fairly good characterization of optimal merging patterns. Richard M. Karp has introduced some very interesting ways to attack this problem, and we shall conclude this section by discussing the theory he has developed.

In the first place we need a more satisfactory way to describe merging patterns, instead of the rather mysterious "tape contents" charts which have been used above. Karp has suggested two ways to do this, the *vector representation* and the *tree representation* of a merge pattern. Both forms of representation are useful in practice, so we shall describe them in turn.

The vector representation of a merge pattern consists of a sequence of "merge vectors" $y^{(m)} \ldots y^{(1)}y^{(0)}$, each of which has T components. The ith-last merge step is represented by $y^{(i)}$ in the following way:

$$y_j^{(i)} = \begin{cases} +1, & \text{if tape number } j \text{ is an input to the merge;} \\ 0, & \text{if tape number } j \text{ is not used in the merge;} \\ -1, & \text{if tape number } j \text{ gets the output of the merge.} \end{cases} \tag{2}$$

Thus, exactly one component of $y^{(i)}$ is -1, and the other components are 0's and 1's. The final vector $y^{(0)}$ is special; it is a unit vector, having 1 in position j if the final sorted output appears on unit j, and 0 elsewhere. These definitions imply that the vector sum

$$v^{(i)} = y^{(i)} + y^{(i-1)} + \cdots + y^{(0)} \tag{3}$$

represents the distribution of runs on tape just before the ith-last merge step, with $v_j^{(i)}$ runs on tape j. In particular, $v^{(m)}$ tells how many runs the initial distribution pass places on each tape.

The three merge patterns described in tabular form earlier in this section have the following vector representations:

Balanced ($T = 4$, $S = 8$)	Cascade ($T = 4$, $S = 14$)	Polyphase ($T = 3$, $S = 13$)
$v^{(7)} = (4, 4, 0, 0)$	$v^{(10)} = (6, 5, 3, 0)$	$v^{(12)} = (5, 8, 0)$
$y^{(7)} = (+1, +1, -1, 0)$	$y^{(10)} = (+1, +1, +1, -1)$	$y^{(12)} = (+1, +1, -1)$
$y^{(6)} = (+1, +1, 0, -1)$	$y^{(9)} = (+1, +1, +1, -1)$	$y^{(11)} = (+1, +1, -1)$
$y^{(5)} = (+1, +1, -1, 0)$	$y^{(8)} = (+1, +1, +1, -1)$	$y^{(10)} = (+1, +1, -1)$
$y^{(4)} = (+1, +1, 0, -1)$	$y^{(7)} = (+1, +1, -1, 0)$	$y^{(9)} = (+1, +1, -1)$
$y^{(3)} = (-1, 0, +1, +1)$	$y^{(6)} = (+1, +1, -1, 0)$	$y^{(8)} = (+1, +1, -1)$
$y^{(2)} = (0, -1, +1, +1)$	$y^{(5)} = (+1, -1, 0, 0)$	$y^{(7)} = (-1, +1, +1)$
$y^{(1)} = (+1, +1, -1, 0)$	$y^{(4)} = (-1, +1, +1, +1)$	$y^{(6)} = (-1, +1, +1)$
$y^{(0)} = (0, 0, 1, 0)$	$y^{(3)} = (0, -1, +1, +1)$	$y^{(5)} = (-1, +1, +1)$
	$y^{(2)} = (0, 0, -1, +1)$	$y^{(4)} = (+1, -1, +1)$
	$y^{(1)} = (+1, +1, +1, -1)$	$y^{(3)} = (+1, -1, +1)$
	$y^{(0)} = (0, 0, 0, 1)$	$y^{(2)} = (+1, +1, -1)$
		$y^{(1)} = (-1, +1, +1)$
		$y^{(0)} = (1, 0, 0)$

It may seem awkward to number these vectors backwards, with $y^{(m)}$ coming first and $y^{(0)}$ last, but this peculiar viewpoint turns out to be advantageous for developing the theory. One good way to search for an optimal method is to start with the sorted output and to imagine "unmerging" it to various tapes, then unmerging these, etc., considering the successive distributions $v^{(0)}$, $v^{(1)}$, $v^{(2)}, \ldots$ in the reverse order from which they actually occur during the sorting process. In fact that is essentially the approach we have taken already in our analysis of polyphase and cascade merging.

Every merge pattern obviously has a vector representation. Conversely, it is easy to see that the sequence of vectors $y^{(m)} \ldots y^{(1)}\, y^{(0)}$ corresponds to an actual merge pattern if and only if the following three conditions are satisfied:

i) $y^{(0)}$ is a unit vector.

ii) $y^{(i)}$ has exactly one component equal to -1, all other components equal to 0 or $+1$, for $m \geq i \geq 1$.

iii) All components of $y^{(i)} + \cdots + y^{(1)} + y^{(0)}$ are nonnegative, for $m \geq i \geq 1$.

The tree representation of a merge pattern gives another picture of the same information. We construct a tree with one external "leaf" node for each initial run, and one internal node for each run that is merged, in such a way that the descendants of each internal node are the runs from which it was fabricated. Each internal node is labeled with the step number on which the corresponding run was formed, numbering steps backwards as in the vector representation; furthermore, the line just above each node is labeled with the name of the tape on which that run appears. For example, the three merge patterns above have the tree representations depicted in Fig. 76, if we call the tapes A, B, C, D instead of T1, T2, T3, T4.

This representation displays many of the relevant properties of the merge pattern in convenient form; for example, if the run on level 0 of the tree (the

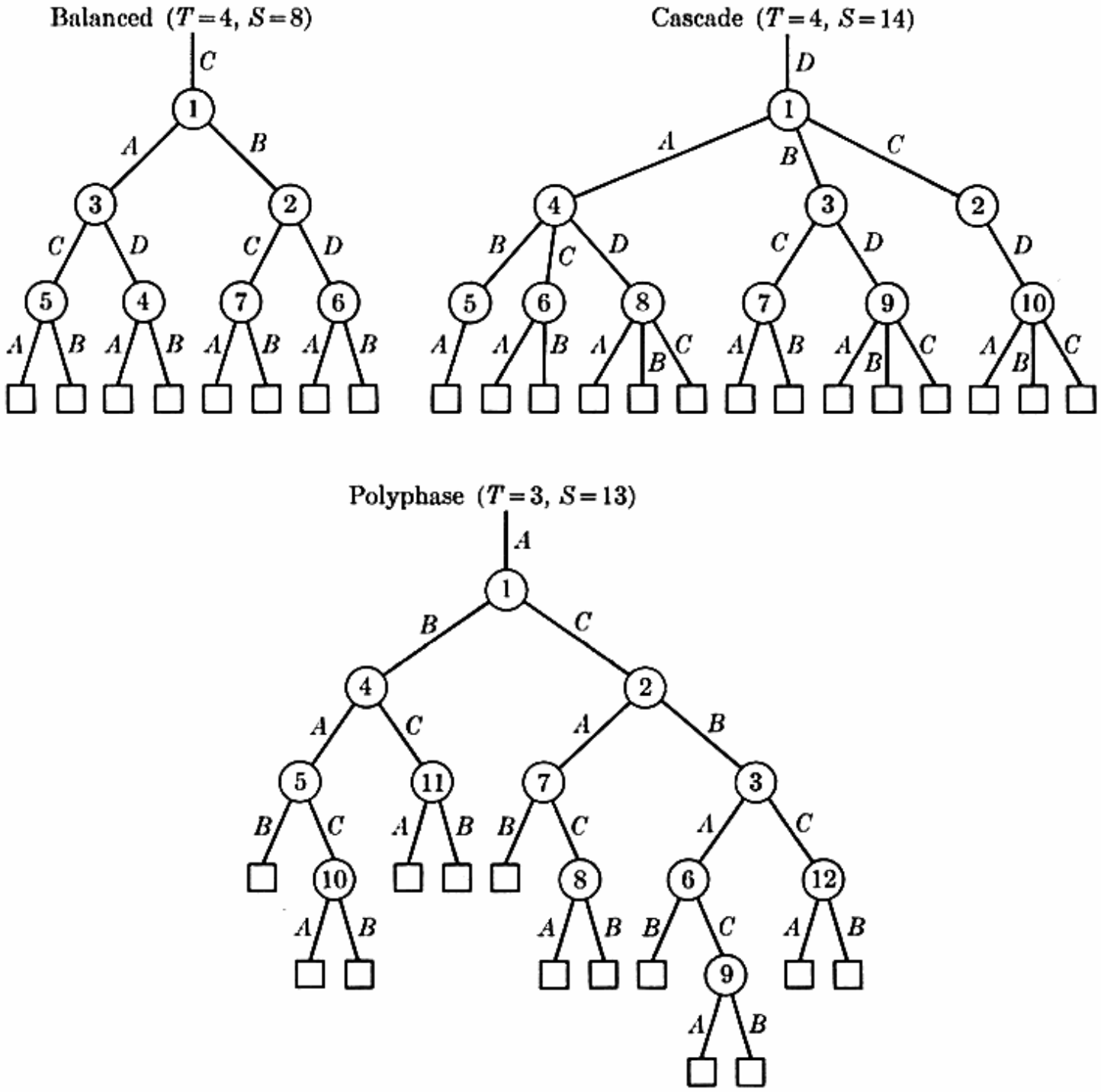


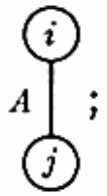
Fig. 76. Tree representations of three merge patterns.

root) is to be ascending, then the runs on level 1 must be descending, those on level 2 must be ascending, etc.; an initial run is ascending if and only if the corresponding external node is on an even-numbered level. Furthermore the total number of initial runs processed during the merging (not including the initial distribution) is exactly equal to the *external path length* of the tree, since each initial run on level k is processed exactly k times.

Every merge pattern has a tree representation, but not every tree defines a merge pattern. A tree whose internal nodes have been labeled with the numbers 1 through m, and whose lines have been labeled with tape names, represents a valid read-backwards merge pattern if and only if

a) No two lines adjacent to the same internal node have the same tape name;

b) If $i > j$, and if A is a tape name, the tree does not contain the configuration



;

c) If $i < j < k < l$, and if A is a tape name, the tree does not contain

both

and

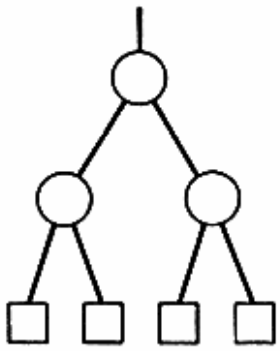
or both

and

. (4)

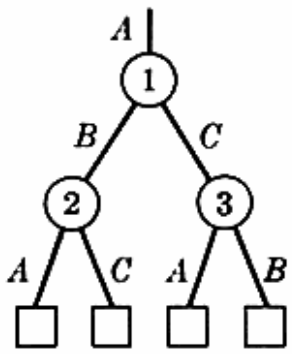
Condition (a) is self-evident, since the input and output tapes in a merge must be distinct; similarly, (b) is obvious. The "no crossover" condition (c) mirrors the last-in-first-out restriction which characterizes read-backwards operations on tape: A run formed at step k must be removed before a run formed previously on that same tape; hence the configurations in (4) are impossible. It is not difficult to verify that any labeled tree satisfying conditions (a), (b), (c) does indeed correspond to a read-backwards merge pattern.

If there are T tape units, condition (a) implies that the degree of each internal node is $T - 1$ or less. It is not always possible to attach suitable labels to all such trees; for example, when $T = 3$ there is no merge pattern whose tree has the shape



(5)

This shape would lead to an optimal merge pattern if we could attach step numbers and tape names in a suitable way, since it is the only way to achieve the minimum external path length in a tree having four external nodes. But there is essentially only one way to do the labeling according to conditions (a) and (b), because of the symmetries of the diagram, namely,



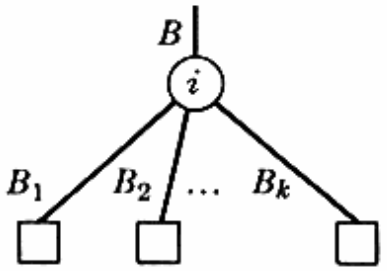
(6)

and this violates condition (c). A shape which *can* be labeled, according to the above conditions, using T or less tape names, is called a *T-lifo* tree.

Another way to characterize all labeled trees that can arise from merge patterns is to consider how all such trees can be "grown." Start with some tape name, say A, and with the seedling tree

Step number i in the tree's growth consists of choosing distinct tape names $B, B_1, B_2, \ldots, B_k$, and changing the *most recently formed* external node corresponding to B

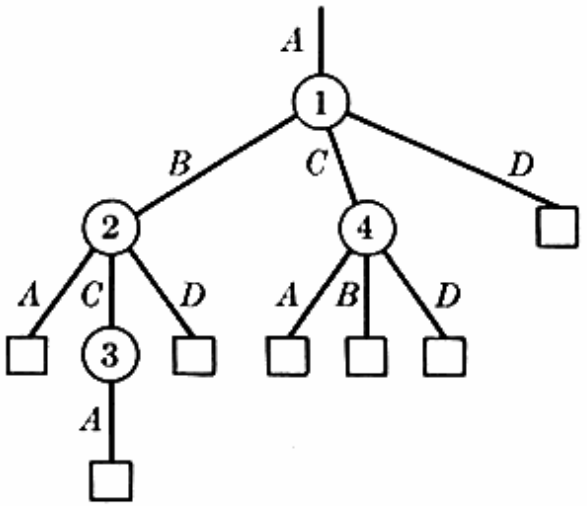
from

to 

(7)

This "last formed, first grown on" rule explains exactly how the tree representation can be constructed directly from the vector representation.

The determination of strictly optimum T-tape merge patterns, i.e. of trees which have the exact minimum path length over all T-lifo trees with a given number of external nodes, seems to be quite difficult. For example, the following nonobvious pattern turns out to be an optimum way to merge seven initial runs on four tapes, reading backwards:



(8)

A one-way merge is actually necessary to achieve the optimum! (See exercise 8.) On the other hand, it is not so difficult to give constructions that are *asymptotically* optimal, for any fixed T.

Let $K_T(n)$ be the minimum external path length achievable in a T-lifo tree with n external nodes. From the theory developed in Section 2.3.4.5, it is not difficult to prove that

$$K_T(n) \geq nq - \lfloor ((T-1)^q - n) / (T-2) \rfloor, \qquad q = \lceil \log_{T-1} n \rceil, \tag{9}$$

since this is the minimum external path length of *any* tree with n external nodes and all nodes of degree $< T$. At the present time comparatively few values of $K_T(n)$ are known exactly. Here are some upper bounds which are probably exact:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$K_3(n) \leq$	0	2	5	9	12	16	21	25	30	34	39	45	50	56	61
$K_4(n) \leq$	0	2	3	6	8	11	14	17	20	24	27	31	33	37	40

(10)

Karp has discovered that *any* tree whose internal nodes have degrees $< T$ is *almost* T-lifo, in the sense that it can be made T-lifo by changing some of the external nodes to one-way merges. In fact, the construction of a suitable labeling is fairly simple. Let A be a particular tape name, and proceed as follows:

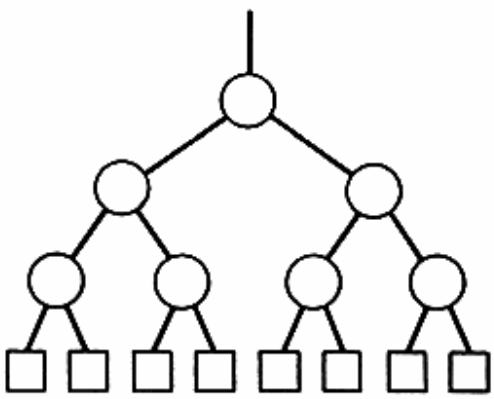
Step 1. Attach tape names to the lines of the tree diagram, in any manner consistent with condition (a) above, provided that the special name A is used only in the leftmost line of a branch.

Step 2. Replace each external node of the form

whenever $B \neq A$.

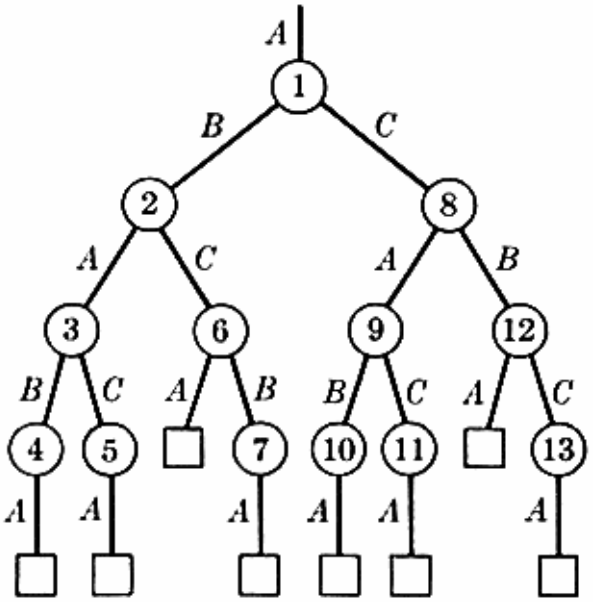
Step 3. Number the internal nodes of the tree in *preorder*. The result will be a labeling satisfying conditions (a), (b), (c).

For example, if we start with the tree



(11)

and three tapes, this procedure might assign labels as follows:



(12)

It is not difficult to verify that Karp's construction satisfies the "last formed, first grown on" discipline, because of the nature of preorder (see exercise 12).

Note that the result of this construction is a merge pattern for which all of the initial runs appear on tape A. This suggests the following distribution and sorting scheme which we may call the *preorder merge:*

P1. Distribute initial runs onto tape A until the input is exhausted. Let S be the total number of initial runs.

P2. Carry out the above construction, using a minimum-path-length $(T - 1)$-ary tree with S external nodes, obtaining a T-lifo tree whose external path length is within S of the lower bound in (9).

P3. Merge the runs according to this pattern. ▮

This scheme will produce its output on any desired tape. But *it has one serious flaw*—does the reader see what will go wrong? The problem is that the merge pattern requires some of the runs initially on tape A to be ascending, and some to be descending, depending on whether the corresponding external node appears on an odd or an even level. This problem can be resolved without knowing S in advance, by copying runs which should be descending onto an auxiliary tape or tapes, just before they are needed. Then the total amount of processing, in terms of initial run lengths, comes to

$$S \log_{T-1} S + O(S). \tag{13}$$

Thus the preorder merge is definitely better than polyphase or cascade, as $S \to \infty$; indeed, it is asymptotically *optimum,* since (9) shows that $S \log_{T-1} S + O(S)$ is the best we could ever hope to achieve on T tapes. On the other hand, for the comparatively small values of S which usually arise in practice, the preorder merge is rather inefficient; polyphase or cascade methods are simpler and faster, when S is reasonably small. Perhaps it will be possible to invent a simple distribution-and-merge scheme that is competitive with polyphase and cascade for small S, and which is asymptotically optimum for large S.

The second set of exercises below shows how Karp has formulated the question of *read-forward* merging in a similar way. The theory turns out to be

rather more complicated in this case, although some very interesting results have been discovered.

EXERCISES—First Set

1. [*17*] It is often convenient, during read-forward merging, to mark the end of each run on tape by including an artificial "sentinel" record whose key is $+\infty$. How should this practice be modified, when reading backwards?

2. [*20*] Will the columns of an array like (1) always be nondecreasing, or is there a chance that we will have to "subtract" runs from some tape as we go from one level to the next?

▶ **3.** [*20*] Prove that the polyphase distribution method described in connection with (1) will always produce an A run on tape T1 when sorting is complete, if T1 originally starts with $ADA\cdots$ and T2 through T5 start with $DAD\cdots$.

4. [*22*] Is it a good idea to do read-backward polyphase merging after distributing all runs in *ascending* order, imagining all the "D" positions to be initially filled with dummies?

▶ **5.** [*23*] What formulas for the strings of merge numbers replace (8), (9), (10), and (11) of Section 5.4.2, when read-backward polyphase merging is used? Show the merge numbers for the fifth level distribution on six tapes, by drawing a diagram like Fig. 71(a).

6. [*07*] What is the vector representation of the merge pattern whose tree representation is (8)?

7. [*16*] Draw the tree representation for the read-backwards merge pattern defined by the following sequence of vectors:

$$
\begin{array}{lll}
v^{(33)} = (\ 20,\ \ 9,\ \ 5) & y^{(22)} = (+1, -1, +1) & y^{(10)} = (+1, +1, -1) \\
y^{(33)} = (+1, -1, +1) & y^{(21)} = (-1, +1, +1) & y^{(9)} = (+1, -1, +1) \\
y^{(32)} = (+1, +1, -1) & y^{(20)} = (+1, +1, -1) & y^{(8)} = (+1, +1, -1) \\
y^{(31)} = (+1, +1, -1) & y^{(19)} = (-1, +1, +1) & y^{(7)} = (+1, +1, -1) \\
y^{(30)} = (+1, +1, -1) & y^{(18)} = (+1, +1, -1) & y^{(6)} = (+1, +1, -1) \\
y^{(29)} = (+1, -1, +1) & y^{(17)} = (+1, +1, -1) & y^{(5)} = (-1, +1, +1) \\
y^{(28)} = (-1, +1, +1) & y^{(16)} = (+1, +1, -1) & y^{(4)} = (+1, -1, +1) \\
y^{(27)} = (+1, -1, +1) & y^{(15)} = (+1, +1, -1) & y^{(3)} = (-1, +1, +1) \\
y^{(26)} = (+1, -1, +1) & y^{(14)} = (+1, -1, +1) & y^{(2)} = (+1, -1, +1) \\
y^{(25)} = (+1, +1, -1) & y^{(13)} = (+1, -1, +1) & y^{(1)} = (-1, +1, +1) \\
y^{(24)} = (+1, -1, +1) & y^{(12)} = (-1, +1, +1) & y^{(0)} = (\ \ 1,\ \ 0,\ \ 0) \\
y^{(23)} = (+1, -1, +1) & y^{(11)} = (+1, +1, -1) &
\end{array}
$$

8. [*23*] Prove that (8) is an optimum way to merge, reading backwards, when $S = 7$ and $T = 4$, and that all methods which avoid one-way merging are inferior.

9. [*M22*] Prove the lower bound (9).

10. [*41*] Prepare a table of the exact values of $K_T(n)$, using a computer.

▶**11.** [*20*] True or false: Any read-backwards merge pattern that uses nothing but $(T-1)$-way merging must always have the runs alternating $ADAD\cdots$ on each tape; it will not work if two adjacent runs appear in the same order.

12. [*22*] Prove that Karp's preorder construction always yields a labeled tree satisfying conditions (a), (b), and (c).

13. [*16*] Make (12) more efficient, by removing as many of the one-way merges as possible so that preorder still gives a valid labeling of the internal nodes.

14. [*40*] Devise an algorithm which carries out the preorder merge without explicitly representing the tree in steps P2 and P3, using only $O(\log S)$ words of memory to control the merging pattern.

15. [*M39*] Karp's preorder construction in the text yields trees with one-way merges at several terminal nodes. Prove that when $T = 3$ it is possible to contruct asymptotically optimal 3-lifo trees in which two-way merging is used throughout.

In other words, let $\hat{K}_T(n)$ be the minimum external path length over all T-lifo trees with n external nodes, such that every internal node has degree $T-1$. Prove that $\hat{K}_3(n) = n\log_2 n + O(n)$.

16. [*M46*] In the notation of exercise 15, is $\hat{K}_T(n) = n\log_{T-1} n + O(n)$ for *all* $T \geq 3$, when $n \equiv 1$ (modulo $T-1$)?

▶**17.** [*28*] (Richard D. Pratt.) To achieve ascending order in a read-backward cascade merge, we could insist on an *even* number of merging passes; this suggests a technique of initial distribution which is somewhat different from Algorithm 5.4.3C. (a) Change 5.4.3–1 so that it shows only the perfect distributions which require an even number of merging passes. (b) Design an initial distribution scheme which interpolates between these perfect distributions. (Thus, if the number of initial runs falls between perfect distributions, it is desirable to merge some, but not all, of the runs twice, in order to reach a perfect distribution.)

18. [*46*] Suppose that T tape units are available, for some $T \geq 3$, and that T1 contains N records while the remaining tapes are empty. Is it possible to reverse the order of the records on T1 in less than $O(N\log N)$ steps, *without* reading backwards? (The operation is, of course, trivial if backwards reading is allowed.) Cf. exercise 5.2.5–14 for a class of such algorithms that *do* require order $N\log N$ steps.

EXERCISES—Second Set

The following exercises develop the theory of tape merging on read-forward tapes; in this case each tape acts as a queue instead of as a stack. A merge pattern can be represented as a sequence of vectors $y^{(m)} \ldots y^{(1)}y^{(0)}$ exactly as in the text, but when we convert the vector representation to a tree representation we change "last formed, first grown on" to "*first* formed, first grown on." Thus the invalid configurations (4) would be changed to

both

and

or both

and

. (4′)

A tree which can be labeled so as to represent a read-forward merge on T tapes is called T-fifo, analogous to the term "T-lifo" in the read-backwards case.

When tapes can be read backwards, they make very good stacks. But unfortunately they don't make very good general purpose queues. If we randomly write and read, in a first-in-first-out manner, we waste a lot of time moving from one part of the tape to another. Even worse, we will soon run off the end of the tape! We run into the same problem as the queue overrunning memory in 2.2.2–(4) and (5), but the solution in 2.2.2–(6) and (7) doesn't apply to tapes since they aren't circular loops. Therefore we shall call a tree *strongly T-fifo* if it can be labeled so that the corresponding merge pattern makes each tape follow the special queue discipline "write, rewind, read all, rewind; write, rewind, read all, rewind; etc."

▶ **19.** [*22*] (R. M. Karp.) Find a binary tree which is not 3-fifo.

▶ **20.** [*22*] Formulate the condition "strongly T-fifo" in terms of a fairly simple rule about invalid configurations of tape labels, analogous to (4′).

21. [*18*] Draw the tree representation for the read-forwards merge pattern defined by the vectors in exercise 7. Is this tree strongly 3-fifo?

22. [*28*] (R. M. Karp.) Show that the tree representations for polyphase and cascade merging, with perfect distributions, are exactly the same for both the read-backwards and the read-forwards case, except for the numbers which label the internal nodes. Find a larger class of vector representations of merging patterns for which this is true.

23. [*24*] (R. M. Karp.) Let us say that a segment $y^{(q)} \ldots y^{(r)}$ of a merge pattern is a *stage* if no output tape is subsequently used as an input tape, i.e. if there do not exist i, j, k with $q \geq i > k \geq r$ and $y_j^{(i)} = -1$ and $y_j^{(k)} = +1$. The purpose of this exercise is to prove that *cascade merge minimizes the number of stages*, over all merge patterns having the same number of tapes and initial runs.

It is convenient to define some notation. Let us write $v \rightarrow w$ if v and w are T-vectors such that there is a merge pattern which reduces w to v in its first stage. (Thus there is a merge pattern $y^{(m)} \ldots y^{(0)}$ such that $y^{(m)} \ldots y^{(l+1)}$ is a stage, $w = y^{(m)} + \cdots + y^{(0)}$, and $v = y^{(l)} + \cdots + y^{(0)}$.) Let us write $v \preceq w$ if v and w are T-vectors such that the sum of the largest k elements of v is $\leq$ the sum of the largest k elements of w, for $1 \leq k \leq T$. Thus, for example, $(2, 1, 2, 2, 2, 1) \preceq (1, 2, 3, 0, 3, 1)$, since $2 \leq 3$, $2 + 2 \leq 3 + 3, \ldots, 2 + 2 + 2 + 2 + 1 + 1 \leq 3 + 3 + 2 + 1 + 1 + 0$. Finally, if $v = (v_1, \ldots, v_T)$, let $C(v) = (s_T, s_{T-2}, s_{T-3}, \ldots, s_1, 0)$, where s_k is the sum of the largest k elements of v.

(a) Prove that $v \rightarrow C(v)$. (b) Prove that $v \preceq w$ implies $C(v) \preceq C(w)$. (c) Assuming the result of exercise 24, prove that cascade merge minimizes the number of stages.

24. [*M35*] In the notation of exercise 23, prove that $v \rightarrow w$ implies $w \preceq C(v)$.

25. [*M36*] (R. M. Karp.) Let us say that a segment $y^{(q)} \ldots y^{(r)}$ of a merge pattern is a *phase* if no tape is used both for input and for output, i.e., if there do not exist i, j, k with $q \geq i, k \geq r$ and $y_j^{(i)} = +1$ and $y_j^{(k)} = -1$. The purpose of this exercise is to investigate merge patterns that minimize the number of phases. We shall write $v \Rightarrow w$ if w can be reduced to v in one phase (cf. the similar notation introduced in exercise 23), and we let $D_k(v) = (s_k + t_{k+1}, s_k + t_{k+2}, \ldots, s_k + t_T, 0, \ldots, 0)$, where t_j denotes the jth largest element of v and $s_k = t_1 + \cdots + t_k$. (a) Prove that $v \Rightarrow D_k(v)$ for $1 \leq k < T$. (b) Prove that $v \preceq w$ implies $D_k(v) \preceq D_k(w)$, for $1 \leq k < T$. (c) Prove that $v \Rightarrow w$ implies $w \preceq D_k(v)$, for some k, $1 \leq k < T$. (d) Consequently,

a merge pattern which sorts the maximum number of initial runs on T tapes in q phases can be represented by a sequence of integers $k_1\, k_2 \ldots k_q$, such that the initial distribution is $D_{k_q}(\ldots(D_{k_2}(D_{k_1}(u)))\ldots)$, where $u = (1, 0, \ldots, 0)$. This minimum-phase strategy has a strongly T-fifo representation, and it also belongs to the class of patterns in exercise 22. When $T = 3$ it is the *polyphase* merge, and for $T = 4, 5, 6, 7$ it is a variation of the *balanced* merge.

26. [*M46*] (R. M. Karp.) Is the optimum sequence $k_1\, k_2 \ldots k_q$ mentioned in exercise 25 always equal to $1\lceil T/2\rceil\lfloor T/2\rfloor\lceil T/2\rceil\lfloor T/2\rfloor\ldots$, for all $T \geq 4$ and all sufficiently large q?

5.4.5. The Oscillating Sort

A somewhat different approach to merge sorting was introduced by Sheldon Sobel in *JACM* **9** (1962), 372–375. Instead of starting with a distribution pass where all the initial runs are dispersed to tapes, he proposed an algorithm which oscillates back and forth between distribution and merging, so that much of the sorting takes place before the input has been completely examined.

Suppose, for example, that there are five tapes available for merging. Sobel's method would sort 16 initial runs as follows:

	Operation	T1	T2	T3	T4	T5	"Cost"
Phase 1.	Distribute	A_1	A_1	A_1	A_1	—	4
Phase 2.	Merge	—	—	—	—	D_4	4
Phase 3.	Distribute	—	A_1	A_1	A_1	D_4A_1	4
Phase 4.	Merge	D_4	—	—	—	D_4	4
Phase 5.	Distribute	D_4A_1	—	A_1	A_1	D_4A_1	4
Phase 6.	Merge	D_4	D_4	—	—	D_4	4
Phase 7.	Distribute	D_4A_1	D_4A_1	—	A_1	D_4A_1	4
Phase 8.	Merge	D_4	D_4	D_4	—	D_4	4
Phase 9.	Merge	—	—	—	A_{16}	—	16

Here as in Section 5.4.4 we use A_r and D_r to stand respectively for ascending and descending runs of relative length r. The method begins by writing an initial run onto each of four tapes, and merges them (reading backwards) onto the fifth tape. Distribution resumes again, this time cyclically shifted one place to the right with respect to the tapes, and a second merge produces another run D_4. When four D_4's have been formed in this way, an additional merge creates A_{16}. We could go on to create three more A_{16}'s, merging them into a D_{64}, and so on until the input is exhausted. It isn't necessary to know the length of the input in advance.

When the number of initial runs, S, is 4^m, it is not difficult to see that this method processes each record exactly $m + 1$ times (once during the distribution and m times during a merge). When S is between 4^{m-1} and 4^m, we could assume that dummy runs are present, bringing S up to 4^m; hence the total sorting time

would essentially amount to $\lceil \log_4 S \rceil + 1$ passes over all the data. This is just what would be achieved by a balanced sort on *eight* tapes; in general, oscillating sort with T work tapes is equivalent to balanced merging with $2(T-1)$ tapes, since it makes $\lceil \log_{T-1} S \rceil + 1$ passes over the data. When S is a power of $T-1$, this is the best *any* T-tape method could possibly do, since it achieves the lower bound in Eq. 5.4.4–9. On the other hand when S is $(T-1)^{m-1}+1$, just one higher than a power of $T-1$, the method wastes nearly a whole pass.

Exercise 2 shows how to eliminate part of this penalty for non-perfect-powers S, by using a special ending routine. A further refinement was discovered in 1966 by Dennis L. Bencher, who called his procedure the "criss-cross merge" [see H. Wedekind, *Datenorganisation* (Berlin: W. de Gruyter, 1970), 164–166; and *U.S. Patent 3540000* (November 10, 1970)]. The main idea is to delay merging until more knowledge of S has been gained. We shall discuss a slightly modified form of Bencher's original scheme.

This improved oscillating sort proceeds as follows:

	Operation	T1	T2	T3	T4	T5	"Cost"
Phase 1.	Distribute	—	A_1	A_1	A_1	A_1	4
Phase 2.	Distribute	—	A_1	A_1A_1	A_1A_1	A_1A_1	3
Phase 3.	Merge	D_4	—	A_1	A_1	A_1	4
Phase 4.	Distribute	D_4A_1	—	A_1	A_1A_1	A_1A_1	3
Phase 5.	Merge	D_4	D_4	—	A_1	A_1	4
Phase 6.	Distribute	D_4A_1	D_4A_1	—	A_1	A_1A_1	3
Phase 7.	Merge	D_4	D_4	D_4	—	A_1	4
Phase 8.	Distribute	D_4A_1	D_4A_1	D_4A_1	—	A_1	3
Phase 9.	Merge	D_4	D_4	D_4	D_4	–	4

We do not merge the D_4's into an A_{16} at this point (unless the input happens to be exhausted); only after building up to

Phase 15.	Merge	D_4D_4	D_4D_4	D_4D_4	D_4	—	4

will we get

Phase 16.	Merge	D_4	D_4	D_4	—	A_{16}	16.

The second A_{16} will occur after three more D_4's have been made,

Phase 22.	Merge	D_4D_4	D_4D_4	D_4	—	$A_{16}D_4$	4
Phase 23.	Merge	D_4	D_4	—	A_{16}	A_{16}	16

and so on (cf. phases 1–5). The advantage of Bencher's scheme can be seen for example if there are only five initial runs: Oscillating sort as modified in exercise 2 would do a four-way merge (in Phase 2) followed by a two-way merge, for a total cost of $4+4+1+5=14$, while Bencher's scheme would do a two-way merge (in Phase 3) followed by a four-way merge, for a total cost of

$4 + 1 + 2 + 5 = 12$. (Both methods also involve a small additional cost, namely one unit of rewind before the final merge.)

A precise description of Bencher's method appears in Algorithm B below. Unfortunately it seems to be a procedure which is harder to understand than to code; it is much easier to explain the technique to a computer than to a computer scientist! This is partly because it is a recursive method that has been expressed in iterative form and then "optimized" somewhat; the reader may find it necessary to trace through the operation of this algorithm several times before he really discovers what is going on.

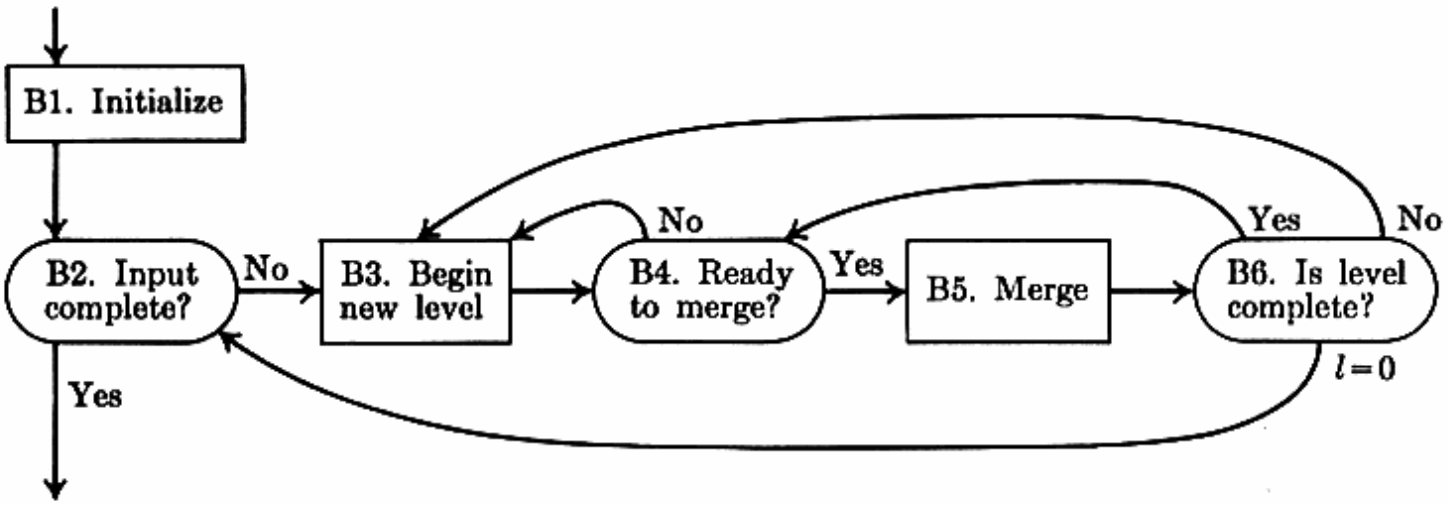


Fig. 77. Oscillating sort, with a "criss-cross" distribution.

Algorithm B (*Oscillating sort with "criss-cross" distribution*). This algorithm takes initial runs and disperses them to tapes, occasionally interrupting the distribution process in order to merge some of the tape contents. The algorithm uses P-way merging, assuming that $T = P + 1 \geq 3$ tape units are available (*not* counting the unit which may be necessary to hold the input data). The tape units must allow reading in both forward and backward directions, and they are designated by the numbers $0, 1, \ldots, P$. The following tables are maintained:

$\mathtt{D}[j]$, $0 \leq j \leq P$	Number of dummy runs assumed to be present at the end of tape j.
$\mathtt{A}[l, j]$, $0 \leq l \leq L$, $0 \leq j \leq P$	Here L is a number which is large enough that at most P^{L+1} initial runs will be input. When $\mathtt{A}[l, j] = k \geq 0$, a run of nominal length P^k is present on tape j, corresponding to "level l" of the algorithm's operation. This run is ascending if k is even, descending if k is odd. When $\mathtt{A}[l, j] = -1$, level l does not use tape j.

The statement "Write an initial run on tape j" is an abbreviation for the following operations:

Set $\mathtt{A}[l, j] \leftarrow 0$. If the input is exhausted, increase $\mathtt{D}[j]$ by 1; otherwise write an initial run (in ascending order) onto tape j.

The statement "Merge to tape j" is an abbreviation for the following operations:

If $D[i] > 0$ for all $i \neq j$, decrease $D[i]$ by 1 for all $i \neq j$ and increase $D[j]$ by 1. Otherwise merge one run to tape j, from all tapes $i \neq j$ such that $D[i] = 0$, and decrease $D[i]$ by 1 for all other $i \neq j$.

B1. [Initialize.] Set $D[j] \leftarrow 0$ for $0 \leq j \leq P$. Then write an initial run on tape j, for $1 \leq j \leq P$. Set $A[0, 0] \leftarrow -1$, $l \leftarrow 0$, $q \leftarrow 0$.

B2. [Input complete?] (At this point tape q is empty and the other tapes contain at most one run each.) If there is more input, go on to step B3. But if the input is exhausted, rewind all tapes $j \neq q$ such that $A[0, j]$ is even; then merge to tape q, reading forward on tapes just rewound, and reading backward on the other tapes. This completes the sort, with the output in ascending order on tape q.

B3. [Begin new level.] Set $l \leftarrow l + 1$, $r \leftarrow q$, $s \leftarrow 0$, and $q \leftarrow (q + 1) \bmod T$. Write an initial run on tape $(q + j) \bmod T$, for $1 \leq j \leq T - 2$. (Thus an initial run is written onto each tape except tapes q and r.) Set $A[l, q] \leftarrow -1$ and $A[l, r] \leftarrow -1$.

B4. [Ready to merge?] If $A[l - 1, q] \neq s$, go back to step B3.

B5. [Merge.] (At this point $A[l - 1, q] = A[l, j] = s$ for all $j \neq q$, $j \neq r$.) Merge to tape r. (See the definition of this operation above.) Then set $s \leftarrow s + 1$, $l \leftarrow l - 1$, $A[l, r] \leftarrow s$, and $A[l, q] \leftarrow -1$. Set $r \leftarrow (2q - r) \bmod T$. (In general, we have $r = (q - 1) \bmod T$ when s is even, $r = (q + 1) \bmod T$ when s is odd.)

B6. [Is level complete?] If $l = 0$, go to B2. Otherwise if $A[l, j] = s$ for all $j \neq q$ and $j \neq r$, go to B4. Otherwise return to B3. ∎

We can use a "recursion induction" style of proof to show that this algorithm is valid, just as we have done for Algorithm 2.3.1T. Suppose that we begin at step B3 with $l = l_0$, $q = q_0$, $s_+ = A[l_0, (q_0 + 1) \bmod T]$, and $s_- = A[l_0, (q_0 - 1) \bmod T]$; and assume furthermore that either $s_+ = 0$ or $s_- = 1$ or $s_+ = 2$ or $s_- = 3$ or $\cdots$. It is possible to verify by induction that the algorithm will eventually get to step B5 without changing rows 0 through l_0 of A, and with $l = l_0 + 1$, $q = q_0 \pm 1$, $r = q_0$, and $s = s_+$ or s_-, where we choose the $+$ sign if $s_+ = 0$ or ($s_+ = 2$ and $s_- \neq 1$) or ($s_+ = 4$ and $s_- \neq 1, 3$) or $\cdots$, and we choose the $-$ sign if ($s_- = 1$ and $s_+ \neq 0$) or ($s_- = 3$ and $s_+ \neq 0, 2$) or $\cdots$. The proof sketched here is not very elegant, but the algorithm has been stated in a form more suited to implementation than to verification.

Figure 78 shows the efficiency of Algorithm B, in terms of the average number of times each record is merged as a function of the number S of initial runs, assuming that the initial runs are approximately equal in length. (Corresponding graphs for polyphase and cascade sort have appeared in Figs. 70 and 74.) A slight improvement, mentioned in exercise 3, has been used in preparing this chart.

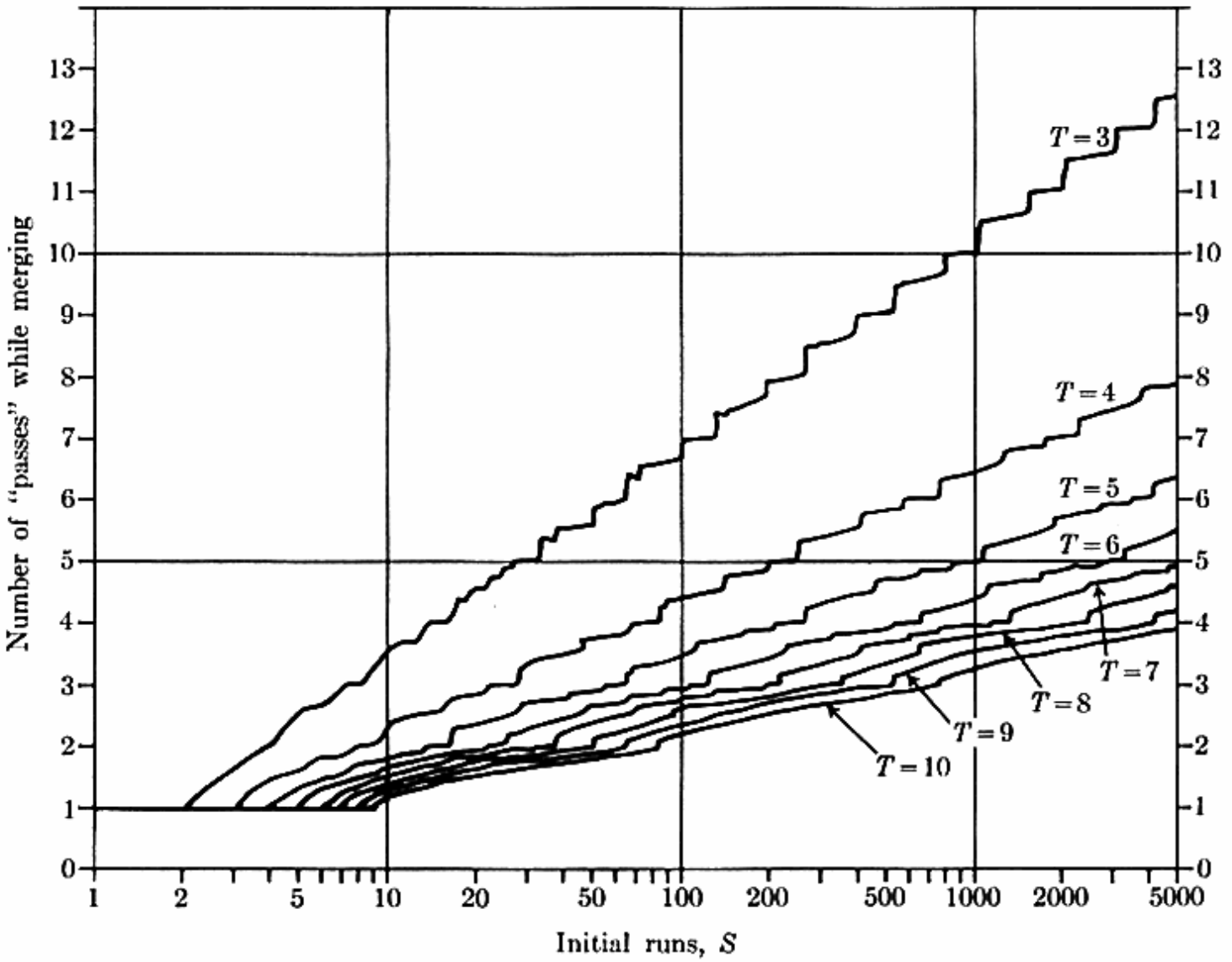


Fig. 78. Efficiency of oscillating sort, using the technique of Algorithm B and exercise 3.

Reading forwards. The oscillating sort pattern appears to require a read-backwards capability, since we need to store long runs somewhere as we merge newly-input short runs. However, M. A. Goetz [*Proc. AFIPS Spring Jt. Comp. Conf.* **25** (1964), 599–607] has discovered a way to perform an oscillating sort using only forward reading and simple rewinding. His method is radically different from the other schemes we have seen in this chapter, because

a) Data is sometimes written at the front of the tape, with the understanding that the existing data in the *middle* of the tape is not destroyed.

b) All initial strings have a fixed maximum length.

Condition (a) violates the "first-in-first-out" property we have assumed to be characteristic of forward reading, but it can be implemented reliably if a sufficient amount of blank tape is left between runs and if "parity errors" are ignored at appropriate times. Condition (b) tends to be somewhat incompatible with an efficient use of replacement selection.

Goetz's read-forward oscillating sort has the somewhat dubious distinction of being one of the first algorithms to be patented as an algorithm instead of as a physical device [*U.S. Patent 3380029* (April 23, 1968)]; unless successfully contested, this means it is illegal to use the algorithm in a program without

permission of the patentee. Bencher's read-backwards oscillating sort technique was patented by IBM several years later. [Thus, we have reached the end of the era when the joy of discovering a new algorithm was satisfaction enough! Since programming is strongly analogous to the fabrication of a machine, and since computer programs are now worth money, patented algorithms are inevitable. Of course the specter of people keeping new techniques completely secret is far worse than the public appearance of algorithms which are proprietary for a limited time.]

The central idea in Goetz's method is to arrange things so that each tape begins with a run of relative length 1, followed by one of relative length P, then P^2, etc. For example, when $T = 5$ the sort begins as follows, using "." to indicate the current position of the read-write head on each tape:

	Operation	T1	T2	T3	T4	T5	"Cost"	Remarks
Phase 1.	Distribute	$.A_1$	$.A_1$	$.A_1$	$.A_1$	$A_1.$	5	[T5 not rewound]
Phase 2.	Merge	$\not{A}_1.$	$\not{A}_1.$	$\not{A}_1.$	$\not{A}_1.$	$A_1\ A_4.$	4	[Now rewind all]
Phase 3.	Distribute	$.A_1$	$.A_1$	$.A_1$	$A_1.$	$.A_1\ A_4$	4	[T4 not rewound]
Phase 4.	Merge	$\not{A}_1.$	$\not{A}_1.$	$\not{A}_1.$	$A_1\ A_4.$	$\not{A}_1.A_4$	4	[Now rewind all]
Phase 5.	Distribute	$.A_1$	$.A_1$	$A_1.$	$.A_1\ A_4$	$.A_1\ A_4$	4	[T3 not rewound]
Phase 6.	Merge	$\not{A}_1.$	$\not{A}_1.$	$A_1\ A_4.$	$\not{A}_1.A_4$	$\not{A}_1.A_4$	4	[Now rewind all]
Phase 7.	Distribute	$.A_1$	$A_1.$	$.A_1\ A_4$	$.A_1\ A_4$	$.A_1\ A_4$	4	[T2 not rewound]
Phase 8.	Merge	$\not{A}_1.$	$A_1\ A_4.$	$\not{A}_1.A_4$	$\not{A}_1.A_4$	$\not{A}_1.A_4$	4	[Now rewind all]
Phase 9.	Distribute	$A_1.$	$.A_1\ A_4$	$.A_1\ A_4$	$.A_1\ A_4$	$.A_1\ A_4$	4	[T1 not rewound]
Phase 10.	Merge	$A_1A_4.$	$\not{A}_1.A_4$	$\not{A}_1.A_4$	$\not{A}_1.A_4$	$\not{A}_1.A_4$	4	[No rewinding]
Phase 11.	Merge	$A_1A_4A_{16}.$	$\not{A}_1\ \not{A}_4.$	$\not{A}_1\ \not{A}_4.$	$\not{A}_1\ \not{A}_4.$	$\not{A}_1\ \not{A}_4.$	16	[Now rewind all]

And so on. During Phase 1, T1 was rewinding while T2 was receiving its input, then T2 was rewinding while T3 was receiving input, etc. Eventually, when the input is exhausted, dummy runs will start to appear, and it sometimes becomes necessary to imagine that they have been written explicitly on the tape at full length. For example, if $S = 18$, the A_1's on T4 and T5 would be dummies during Phase 9; we would have to skip forward on T4 and T5 while merging from T2 and T3 to T1 during Phase 10, because we have to get to the A_4's on T4 and T5 in preparation for Phase 11. On the other hand, the dummy A_1 on T1 need not appear explicitly. Thus the "endgame" is a bit tricky.

A further example of this method appears in the next section.

EXERCISES

1. [*22*] The text illustrates Sobel's original oscillating sort for $T = 5$ and $S = 16$. Give a precise specification of an algorithm which generalizes the procedure, sorting $S = P^L$ initial runs on $T = P + 1 \geq 3$ tapes. Strive for an algorithm which can be described very simply.

2. [*24*] If $S = 6$ in Sobel's original method, we could pretend that $S = 16$ and that 11 dummy runs were present. Then Phase 3 in the text's example would put dummy runs A_0 on T4 and T5; Phase 4 would merge the A_1's on T2 and T3 into a D_2 on T1; Phases 5–8 would do nothing; and Phase 9 would produce A_6 on T4. It

would be better to rewind T2 and T3 just after Phase 3, then to produce A_6 immediately on T4 by three-way merging.

Show how to modify the algorithm of exercise 1, so that an improved ending like this is obtained when S is not a perfect power of P.

▶ **3.** [*24*] Prepare a chart showing the behavior of Algorithm B when $T = 3$, assuming that there are nine initial runs. Show that the procedure is obviously inefficient in one place, and prescribe corrections to Algorithm B which will remedy the situation.

4. [*21*] Step B3 sets both A[l, q] and A[l, r] to -1. Show that one of these two operations is always superfluous, since the corresponding A table entry is never looked at.

5. [*M25*] Let S be the number of initial runs present in the input to Algorithm B. Which values of S require *no rewinding* in step B2?

5.4.6. Practical Considerations for Tape Merging

Here comes the nitty-gritty: Now that we have discussed the various families of merge patterns, it is time to see how they actually apply to real configurations of computers and magnetic tapes, and to compare them in a meaningful way. Our study of internal sorting showed that we can't adequately judge the efficiency of a sorting method merely by counting the number of comparisons it performs; similarly we can't properly evaluate an external sorting method by simply knowing the number of passes it makes over the data.

In this section we shall discuss the characteristics of typical tape units, and the way they affect initial distribution and merging. In particular we shall study some schemes for buffer allocation, and the corresponding effects on running time. We also shall consider briefly the construction of *sort generator* programs.

How tape works. There is considerable variation in the characteristics of tape units provided by different manufacturers. For convenience, we shall define a hypothetical MIXT tape unit, which is reasonably typical of the equipment being manufactured at the time this book was written. By studying how to design a sorting algorithm for MIXT tapes, it will be easier to understand how to deal with other particular tape units.

MIXT reads and writes 800 characters per inch of tape, at a rate of 75 inches per second. This means that one character is read or written every $\frac{1}{60}$ ms, i.e. $16\frac{2}{3}$ microseconds, when the tape is active. [Actual tape units which are currently being sold have densities ranging from 200 to 1600 characters per inch, and tape speeds ranging from $37\frac{1}{2}$ to 150 inches per second, so their effective speed varies from $\frac{1}{8}$ to 4 times as fast as MIXT. In practice, much sorting is done in commercial environments on relatively small and inexpensive equipment which is slower than that considered here. On the other hand, tape units may soon change drastically, making the present assumptions obsolete. Our main concern here is not to obtain particular answers; it is to learn how to combine theory and practice in a reasonable way.]

One of the important considerations to keep in mind is that individual tapes aren't infinitely long. Each reel contains 2400 feet of tape or less, hence

there is room for at most 23,000,000 or so characters per reel of MIXT tape; and it takes about $23000000/3600000 \approx 6.4$ minutes to read them all. If larger files must be sorted, it is generally best to sort one reelfull at a time, and then to merge the individual sorted reels, in order to avoid excess tape handling. This means that the number of initial runs, S, actually present in the merge patterns we have been studying is never extremely large. We will never find $S > 5000$, even with a very small internal memory which produces initial runs only 5000 characters long. Consequently the formulas which give asymptotic efficiency of the algorithms as $S \to \infty$ are primarily of academic interest.

Data appears on tape in *blocks* (Fig. 79), and each read/write instruction transmits a single block. Blocks are often called "records," but we shall avoid this terminology because it conflicts with the fact that we are sorting a file of "records" in another sense. On many of the early sorting programs written during the 1950's, this distinction was unnecessary, since one record was written per block; but we shall see that it is usually advantageous to have quite a few records in every block on the tape.

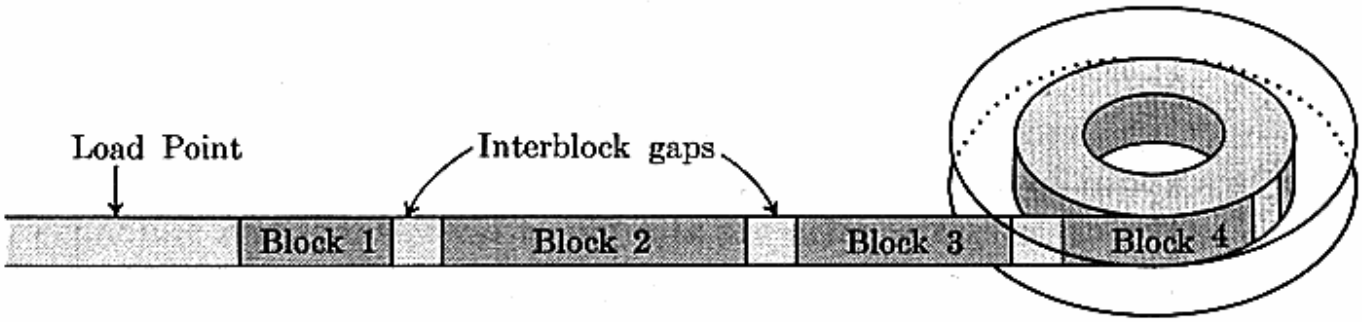


Fig. 79. Magnetic tape with variable-size blocks.

An *interblock gap*, 480 character positions long, appears between adjacent blocks, in order to allow the tape to stop and to start between individual read or write commands. The effect of interblock gaps is to decrease the number of characters per reel of tape, depending on the number of characters per block (see Fig. 80); and the average number of characters transmitted per second decreases in the same way, since tape moves at a fairly constant speed.

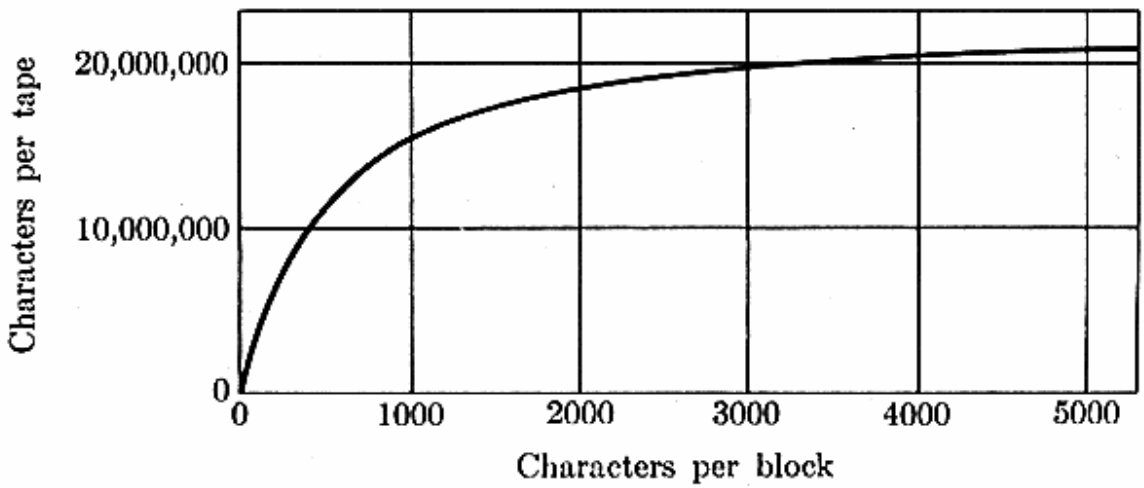


Fig. 80. The number of characters per reel of MIXT tape, as a function of the block size.

Many "old fashioned" computers have fixed block sizes which are rather small. For example, MIX as defined in Chapter 1 always reads and writes 100-word blocks; since this represents about 500 characters per block, and 480 characters per gap, almost half the tape is wasted! Most machines now allow the block size to be variable, and so we shall discuss the choice of appropriate block sizes below.

At the end of a read or write operation, the tape unit "coasts" at full speed over the first 66 characters (or so) of the gap. If the next operation for the same tape is initiated during this time, the tape motion continues without interruption. But if the next operation doesn't come soon enough, the tape will stop and it will also require some time to accelerate to full speed on the next operation. The combined stop-start time delay is 5 ms, 2 for the stop and 3 for the start (see Fig. 81). Thus if we just miss the chance to have continuous full-speed reading, the effect on running time is essentially the same as if there were 780 characters instead of 480 in the interblock gap.

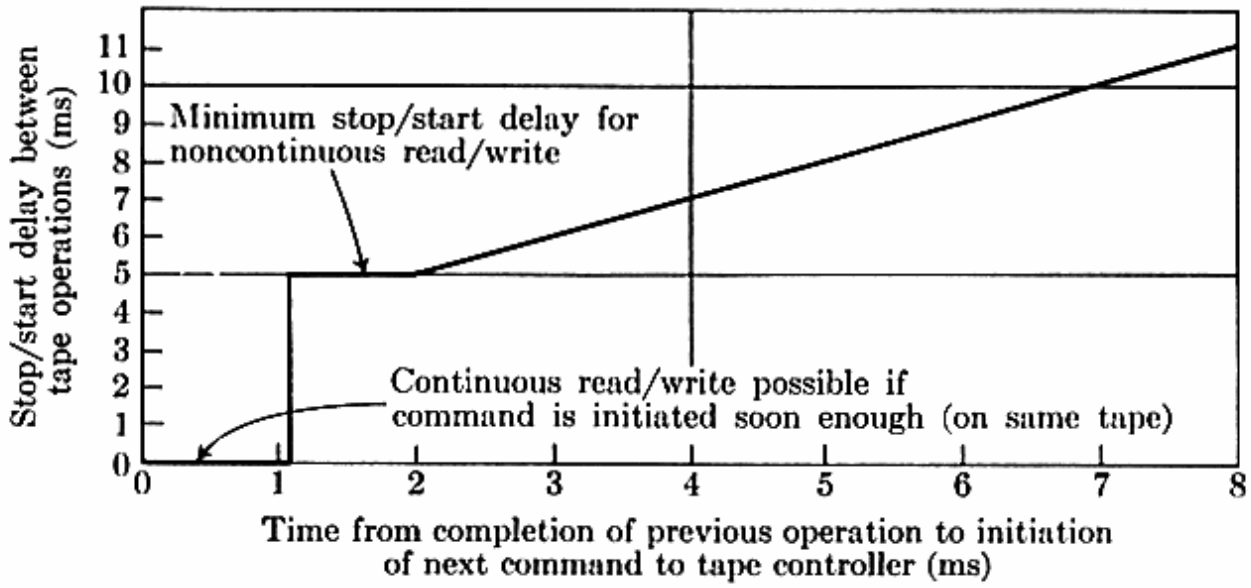


Fig. 81. How to compute the stop/start delay time. (This gets added to the time used for reading or writing the blocks and the gaps.)

Now let us consider the operation of *rewinding*. Unfortunately, it is generally difficult to characterize the exact time needed to rewind over a given number n of characters. On some machines there is a "high-speed rewind" which applies only when n is greater than 5 million or so; for smaller values of n, rewinding goes at normal read/write speed. On other machines there is a special motor which is used on all rewind operations; it gradually accelerates the tape reel to a certain number of revolutions per minute, then puts on the brakes when it is time to stop, and the actual tape speed varies with the fullness of the reel. For simplicity, we shall assume that MIXT requires max (30, $n/150$) ms to rewind over n character positions (including gaps), roughly two-fifths as long as it took to write them. This is a reasonably good approximation to the behavior of many actual tape units, where the ratio of read/write time to rewind time is generally between 2 and 3, but it does not adequately model the

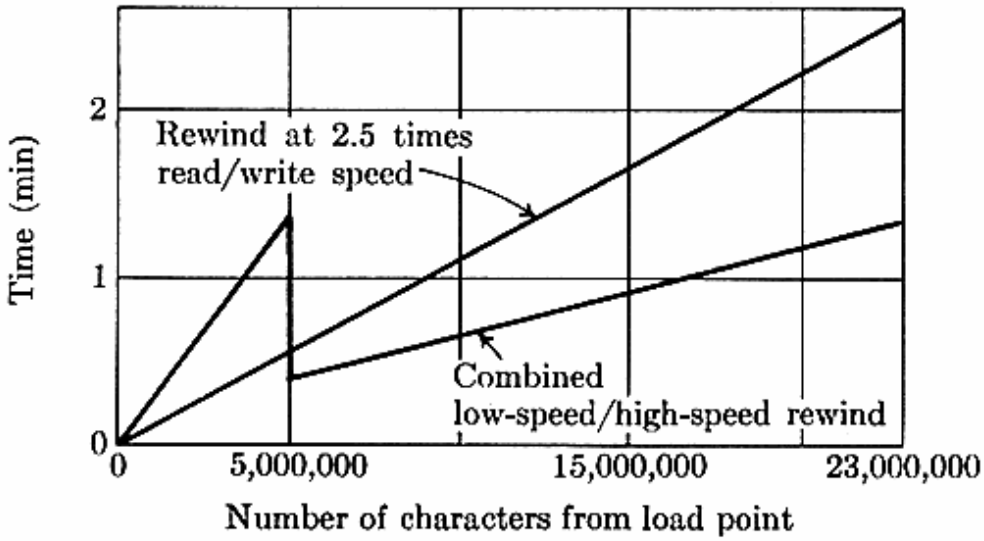


Fig. 82. Approximate running time for two commonly used rewind techniques.

effect of combined low-speed and high-speed rewind which is present on many other machines. (See Fig. 82.)

Initial loading and/or rewinding will position a tape at "load point", and an extra 110 ms are necessary for any read or write operation initiated at load point. When the tape is not at load point, it may be read backwards; an extra 32 ms is added to the time of any backward operation following a forward operation or any forward operation following a backward one.

Finally we must consider the allowable simultaneity of input and output. For economic reasons, several tape units are often attached to a *tape controller*, which is able to deal with only one or two of its units at a time because there are a limited number of data paths between it and the computer. Sometimes controllers are not capable of dealing with more than one unit at a time, but they are often able to read on one unit while writing on another. Somewhat rarer are controllers which can read on two units simultaneously; and the author has never seen a controller that can write on two units at once. Rewinding is a special case: 30 ms after a `MIXT` rewind has been initiated, the tape unit is "detached" from its controller, which becomes free to access other units. In this way a very large number of tape units could be rewinding simultaneously.

Nearly all machines allow input and output to proceed concurrently with computing, although many computers run 20 to 40 percent more slowly when input/output is in progress due to "cycle sharing."

Merging revisited. Let us now look again at the process of P-way merging, with an emphasis on input and output activities, assuming that $P + 1$ tape units are being used for the input files and the output file. Our goal is to overlap the input/output operations as much as possible with each other and with the computations of the program, so that the overall merging time is minimized.

It is instructive to consider the following special case, in which serious restrictions are placed on the amount of simultaneity possible. Suppose that

a) At most one tape may be written on at any one time.
b) At most one tape may be read from at any one time.

c) Reading, writing, and computing may take place simultaneously only when the read and write operations have been initiated simultaneously.

It turns out that a system of $2P$ input buffers and 2 output buffers is sufficient to keep the tape moving at essentially its maximum speed, even though these three restrictions are imposed, unless the computer is unusually slow. Note that (a) is not really a restriction, since there is only one output tape. Furthermore the amount of input is equal to the amount of output, so there is only one tape being read, on the average, at any given time; if condition (b) is not satisfied, there will necessarily be periods when no input at all is occurring. Thus we can minimize the merging time if we keep the output tape busy.

An important technique called prediction or *forecasting* leads to the desired effect. While we are doing a P-way merge, we generally have P "current input buffers," which are being used as the source of data; some of them are more full than others, depending on how much of their data has already been scanned. If all of them become empty at about the same time, we will need to do a lot of reading before we can proceed further, unless we have foreseen this eventuality in advance. Fortunately it is always possible to tell which buffer will empty first, by simply looking at the *last* record in each buffer. The buffer whose last record has the smallest key will always be the first one empty, regardless of the values of any other keys; so we always know which file should be the source of our next input command. The following algorithm spells out this principle in detail.

Algorithm F (*Forecasting with floating buffers*). This algorithm controls the buffering during a P-way merge of long input files, for $P \geq 2$. Assume that the input tapes and files are numbered $1, 2, \ldots, P$. The algorithm uses $2P$ input buffers $\mathtt{I}[1], \ldots, \mathtt{I}[2P]$; two output buffers $\mathtt{O}[0]$ and $\mathtt{O}[1]$; and the following auxiliary tables:

$\mathtt{A}[j]$, $1 \leq j \leq 2P$: 0 if $\mathtt{I}[j]$ is available for input, 1 otherwise;

$\mathtt{B}[i]$, $1 \leq i \leq P$: The buffer containing the last block read so far from file i;

$\mathtt{C}[i]$, $1 \leq i \leq P$: The buffer currently being used for the input from file i;

$\mathtt{L}[i]$, $1 \leq i \leq P$: The last key read so far from file i;

$\mathtt{S}[j]$, $1 \leq j \leq 2P$: The buffer to use when $\mathtt{I}[j]$ becomes empty.

The algorithm described here does not terminate; an appropriate way to "shut it off" is discussed below.

F1. [Initialize.] Read the first block from tape i into buffer $\mathtt{I}[i]$, set $\mathtt{A}[i] \leftarrow 1$, $\mathtt{A}[P+i] \leftarrow 0$, $\mathtt{B}[i] \leftarrow i$, $\mathtt{C}[i] \leftarrow i$, and set $\mathtt{L}[i]$ to the key of the final record in $\mathtt{I}[i]$, for $1 \leq i \leq P$. Then find m such that $\mathtt{L}[m] = \min\{\mathtt{L}[1], \ldots, \mathtt{L}[P]\}$; and set $t \leftarrow 0$, $k \leftarrow P+1$. Begin to read from tape m into buffer $\mathtt{I}[k]$.

F2. [Merge.] Merge records from buffers $\mathtt{I}[\mathtt{C}[1]], \ldots, \mathtt{I}[\mathtt{C}[P]]$ to $\mathtt{O}[t]$, until $\mathtt{O}[t]$ is full. If during this process an input buffer, say $\mathtt{I}[\mathtt{C}[i]]$, becomes empty and $\mathtt{O}[t]$ is not yet full, set $\mathtt{A}[\mathtt{C}[i]] \leftarrow 0$, $\mathtt{C}[i] \leftarrow \mathtt{S}[\mathtt{C}[i]]$, and continue to merge.

F3. [I/O complete.] Wait until the previous read (or read/write) operation is

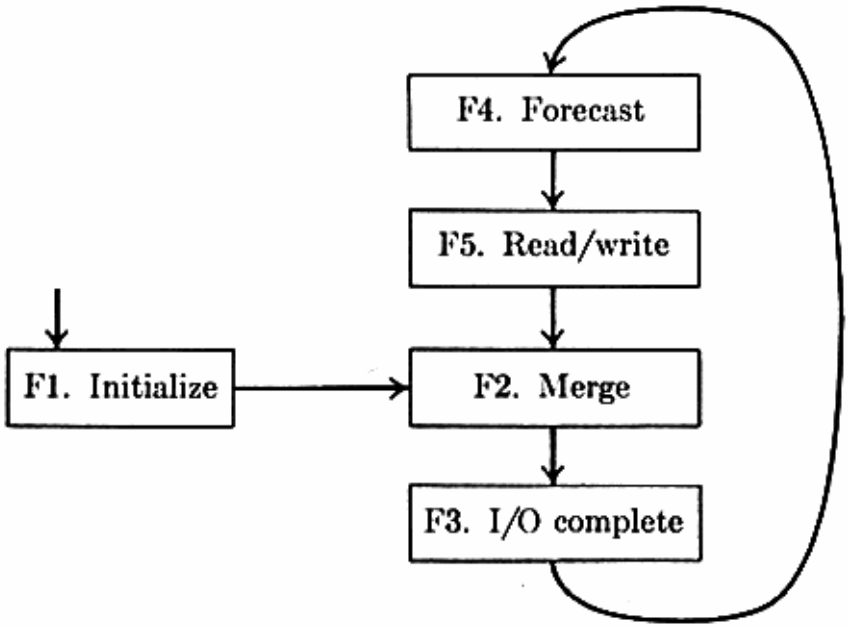


Fig. 83. Forecasting with floating buffers.

complete. Then set $A[k] \leftarrow 1$, $S[B[m]] \leftarrow k$, $B[m] \leftarrow k$, and set $L[m]$ to the key of the final record in $I[k]$.

F4. [Forecast.] Find m such that $L[m] = \min\{L[1], \ldots, L[P]\}$, and find k such that $A[k] = 0$.

F5. [Read/write.] Begin to read from tape m into buffer $I[k]$, and to write from buffer $O[t]$ onto the output tape. Then set $t \leftarrow 1 - t$ and return to F2. ▮

The example in Fig. 84 shows how forecasting works, when $P = 2$, assuming that each block on tape contains only two records. The input buffer contents are illustrated each time we get to the beginning of step F2. Algorithm F essentially forms P *queues of buffers*, with $C[i]$ pointing to the front and $B[i]$

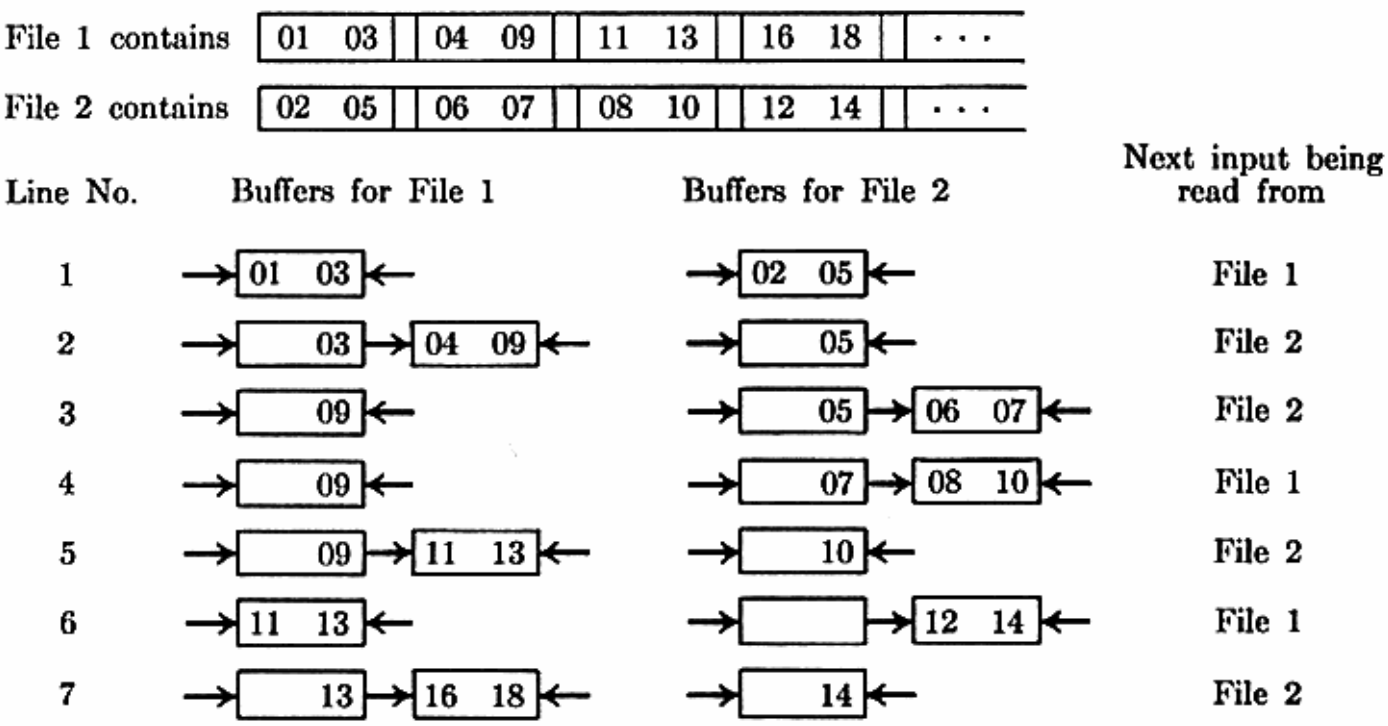


Fig. 84. Buffer queuing, according to Algorithm F.

to the rear of the ith queue, and with $\mathtt{S}[j]$ pointing to the successor of buffer $\mathtt{I}[j]$; these pointers are shown as arrows in Fig. 84. Line 1 illustrates the state of affairs after initialization: There is one buffer for each input file, and another block is being read from File 1 (since $03 < 05$). Line 2 shows the status of things after the first block has been merged: we are outputting a block containing "$\boxed{01\ 02}$", and inputting the next block from File 2 (since $05 < 09$). Note that in line 3, three of the four input buffers are essentially committed to File 2, since we are reading from that file and we already have a full buffer and a partly full buffer in its queue. This "floating buffer" arrangement is an important feature of Algorithm F, since we would be unable to proceed in line 4 if we had chosen File 1 instead of File 2 for the input on line 3.

In order to prove that Algorithm F is valid, we must show two things:

i) There is always an input buffer available (i.e., we can always find a k in step F4).

ii) If an input buffer is exhausted while merging, its successor is already present in memory (i.e., $\mathtt{S}[\mathtt{C}[i]]$ is meaningful in step F2).

Suppose (i) is false, so that all buffers are unavailable at some point when we reach step F4. Each time we get to that step, the total amount of unprocessed data among all the buffers is exactly P bufferloads, i.e. just enough data to fill P buffers if it were relocated, since we are inputting and outputting data at the same rate. Some of the buffers are only partially full; but at most one buffer for each file is partially full, so at most P buffers are partly full. By hypothesis all $2P$ of the buffers are unavailable, so at least P of them must be completely full. This can happen only if P are full and P are empty, otherwise we would have too much data. But at most one buffer can be unavailable and empty at any one time; hence (i) cannot be false.

Suppose (ii) is false, so that we have no unprocessed records in memory, for some file, but the current output buffer is not yet full. By the principle of forecasting, we must have no more than one block of data for all the other files, since we do not read in a block for a file unless that block will be needed before the buffers on any other file are exhausted. Therefore the total number of unprocessed records amounts to at most $P - 1$ blocks; adding the unfilled output buffer leads to less than P bufferloads of data in memory, a contradiction.

This argument establishes the validity of Algorithm F; and it also indicates the possibility of pathological circumstances under which the algorithm just barely avoids disaster. An important subtlety that we have not mentioned, regarding the possibility of equal keys, is discussed in exercise 5.

One way to terminate Algorithm F gracefully is to set $\mathtt{L}[p]$ to ∞ in step F3 if the block just read is the last of a run. (It is customary to indicate the end of a run in some special way.) After all of the data on all of the files has been read, we will eventually find all of the L's equal to ∞ in step F4; then it is usually possible to begin reading the first blocks of the next run on each file, beginning initialization of the next merge phase as the final $P + 1$ blocks are output.

Thus we can keep the output tape going at essentially full speed, without reading more than one tape at a time. An exception to this rule occurs in step F1, where it would be beneficial to read several tapes at once in order to get things going in the beginning; but step F1 can usually be arranged to overlap with the preceding part of the computation.

The idea of looking at the last record in each block, to predict which buffer will empty first, was discovered in 1953 by F. E. Holberton. The technique was first published by E. H. Friend [*JACM* **3** (1956), 144–145, 165]. His rather complicated algorithm used $3P$ input buffers, with three dedicated to each input file; Algorithm F improves the situation by making use of "floating buffers," allowing any single file to claim as many as $P + 1$ input buffers at once, yet never needing more than $2P$ in all.

A discussion of merging with less than $2P$ input buffers appears at the end of this section. Some computers have a "scatter read—gather write" capability, which allows input/output from nonconsecutive cells in memory; the implications of such a feature are beyond the scope of this book.

Comparative behavior of merge patterns. Let us now use what we know about tapes and merging, to compare the effectiveness of the various merge patterns that we have studied in Sections 5.4.2 through 5.4.5. It is very instructive to work out the details of each method as applied to a particular "unbiased" example. Consider therefore the problem of sorting a file whose records each contain 100 characters, when there are 100,000 character positions of memory available for data storage (exclusive of the program and its auxiliary variables, and the comparatively small amount of space needed for links in a selection tree). The input appears in random order on tape, in blocks of 5000 characters each, and the output is to appear in the same format. There are five "scratch tapes" to work with, in addition to the unit containing the input tape.

The total number of records to be sorted is 100,000, but this information is not known in advance to the sorting algorithm.

Chart A (located between pages 340 and 341) summarizes the actions which transpire when ten different merging schemes are applied to this data. The best way to look at this important illustration is to imagine that you are actually watching the sort take place: scan each line slowly from left to right, pretending that you can actually see six tapes reading, writing, rewinding, and/or reading backward, as indicated on the diagram. During a P-way merge the input tapes will be moving only $1/P$ times as often as the output tape. Note that when the original input tape has been completely read (and rewound "with lock"), Chart A assumes that a skilled computer operator dismounts it and replaces it with a scratch tape, in just 30 seconds. In examples 2, 3, and 4 this is "critical path time" when the computer is idly waiting for the operator to finish; but in the remaining examples, the dismount-reload operation is overlapped by other processing.

Example 1. Read-forward balanced merge. Let's review the specifications of the problem: The records are 100 characters long, there is enough internal

memory to hold 1000 records at a time, and each block on the input tape contains 5000 characters (50 records). There are 100,000 records (= 10,000,000 characters = 2000 blocks) in all.

We are free to choose the block size for intermediate files. A six-tape balanced merge uses three-way merging, so the technique of Algorithm F calls for 8 buffers; we may therefore use blocks containing 1000/8 = 125 records (= 12500 characters) each.

The initial distribution pass can make use of replacement selection (Section 5.4.1), and in order to keep the tapes running smoothly we may use two input buffers of 50 records each, plus two output buffers of 125 records each. This leaves room for 650 records in the replacement selection tree. Most of the initial runs will therefore be about 1300 records long (10 or 11 blocks); it turns out that 78 initial runs are produced in Chart A, the last one being rather short.

The first merge pass indicated shows nine runs merged to tape 4, instead of alternating between tapes 4, 5, and 6. This makes it possible to do useful work while the computer operator is loading a scratch tape onto unit 6; since the total number S of runs is known once the initial distribution has been completed, the algorithm knows that $\lceil S/9 \rceil$ runs should be merged to tape 4, then $\lceil (S-3)/9 \rceil$ to tape 5, then $\lceil (S-6)/9 \rceil$ to tape 6.

The entire sorting procedure for this example can be summarized in the following way, using the notation introduced in Section 5.4.2:

1^{26}	1^{26}	1^{26}	—	—	—
—	—	—	3^9	3^9	3^8
9^3	9^3	$9^2 6^1$	—	—	—
—	—	—	27^1	27^1	24^1
78^1	—	—	—	—	—

Example 2. Read-forward polyphase merge. The second example in Chart A carries out the polyphase merge, according to Algorithm 5.4.2D. In this case we do five-way merging, so the memory is split into 12 buffers of 83 records each. During the initial replacement selection we have two 50-record input buffers and two 83-record output buffers, leaving 734 records in the tree; so the initial runs this time are about 1468 records long (17 or 18 blocks). The situation illustrated shows that $S = 70$ initial runs were obtained, the last two actually being only four blocks and one block long, respectively. The merge pattern can be summarized thus:

$0^{13}1^{18}$	$0^{13}1^{17}$	$0^{13}1^{15}$	$0^{12}1^{12}$	$0^8 1^8$	—
1^{15}	1^{14}	1^{12}	1^8	—	$0^8 1^4 2^1 5^3$
1^7	1^6	1^4	—	4^8	$1^4 2^1 5^3$
1^3	1^2	—	8^4	4^4	$2^1 5^3$
1^1	—	$16^1 19^1$	8^2	4^2	5^2
—	34^1	19^1	8^1	4^1	5^1
70^1	—	—	—	—	—

Curiously, polyphase actually took about 25 seconds *longer* than the far less sophisticated balanced merge! There are two main reasons for this:

1) Balanced merge was particularly lucky in this case, since $S = 78$ is so close to a perfect power of 3. If 82 initial runs had been produced, the balanced merge would have taken an extra pass.

2) Polyphase merge wasted 30 seconds while the input tape was being changed, and a total of more than 5 minutes went by while it was waiting for rewind operations to be completed. By contrast the balanced merge needed comparatively little rewind time. In the second phase of the polyphase merge, 13 seconds were saved because the 8 dummy runs on tape 6 could be assumed present even while that tape was rewinding; but no other rewind overlap occurred. Therefore polyphase lost out even though it required significantly less read/write time.

Example 3. Read-forward cascade merge. This case is analogous to the preceding, but using Algorithm 5.4.3C. The merging may be summarized thus:

1^{14}	1^{15}	1^{12}	1^{14}	1^{15}	—
1^{5}	1^{9}	—	1^{14}	1^{15}	$1^{3}2^{3}3^{6}$
$5^{1}6^{3}$	5^{3}	$5^{3}6^{2}$	—	1^{1}	2^{2}
—	12^{1}	6^{1}	18^{1}	18^{1}	16^{1}
70^{1}	—	—	—	—	—

(Remember to watch each of these examples in action, by scanning Chart A.)

Example 4. Tape-splitting polyphase merge. This procedure, described at the end of Section 5.4.2, allows most of the rewind time to be overlapped. It uses four-way merging, so we divide the memory into ten 100-record buffers; there are 700 records in the replacement selection tree, so it turns out that 72 initial runs are formed. The last run, again, is very short. A distribution scheme analogous to Algorithm 5.4.2D has been used, followed by a simple but somewhat ad hoc method of placing dummy runs:

1^{21}	1^{19}	1^{15}	1^{8}	—	$0^{2}1^{9}$
$0^{2}1^{17}$	$0^{2}1^{15}$	$0^{2}1^{11}$	$0^{2}1^{4}$	—	$0^{2}1^{9}4^{4}$
1^{13}	1^{11}	1^{7}	—	$0^{2}4^{4}$	$0^{2}1^{9}4^{4}$
1^{10}	1^{8}	1^{4}	—	$0^{2}4^{4}3^{2}4^{1}$	$1^{8}4^{4}$
1^{6}	1^{4}	—	4^{4}	$0^{2}4^{4}3^{2}4^{1}$	$1^{4}4^{4}$
1^{5}	1^{3}	—	$4^{4}3^{1}$	$0^{1}4^{4}3^{2}4^{1}$	$1^{3}4^{4}$
1^{2}	—	$3^{1}7^{2}$	$4^{4}3^{1}$	$4^{2}3^{2}4^{1}$	4^{4}
1^{1}	—	$3^{1}7^{2}13^{1}$	$4^{3}3^{1}$	$4^{1}3^{2}4^{1}$	4^{3}
—	13^{1}	$3^{1}7^{2}13^{1}$	$4^{2}3^{1}$	$3^{2}4^{1}$	4^{2}
—	$13^{1}14^{1}$	$7^{2}13^{1}$	$4^{1}3^{1}$	$3^{1}4^{1}$	4^{1}
18^{1}	$13^{1}14^{1}$	$7^{1}13^{1}$	3^{1}	4^{1}	—
18^{1}	14^{1}	13^{1}	—	—	27^{1}
—	—	—	72^{1}	—	—

This turns out to give the best running time of all the examples in Chart A that do not read backward. Since S will never get to be very large, it would be possible to develop a more complicated algorithm which places dummy runs in an even better way (cf. Eq. 5.4.2–26).

Example 5. Cascade merge with rewind overlap. This procedure runs almost as fast as the previous example, although the algorithm governing it is much simpler. We simply use the cascade sort method as in Algorithm 5.4.3C for the initial distribution, but with $T = 5$ instead of $T = 6$. Then each phase of each "cascade" staggers the tapes so that we ordinarily don't write on a tape until after it has had a chance to be rewound. The pattern, very briefly, is

1^{21}	1^{22}	1^{19}	1^{10}	—	—
1^{4}	1^{7}	—	—	$1^2 2^2 3^5$	4^{10}
7^{2}	—	8^{3}	$7^2 8^2$	—	4^{1}
—	26^{1}	—	8^{1}	22^{1}	16^{1}
72^{1}	—	—	—	—	—

Example 6. Read-backward balanced merge. This is like example 1 but with all the rewinding eliminated:

A_1^{26}	A_1^{26}	A_1^{26}	—	—	—
—	—	—	D_3^{9}	D_3^{9}	D_3^{8}
A_9^{3}	A_9^{3}	$A_9^{2}A_6^{1}$	—	—	—
—	—	—	D_{24}^{1}	D_{27}^{1}	D_{27}^{1}
A_{78}^{1}	—	—	—	—	—

Since there was comparatively little rewinding in example 1, this scheme is not a great deal better than the read-forward case. In fact, it turns out to be slightly slower than tape-splitting polyphase, in spite of the fortunate value $S = 78$.

Example 7. Read-backward polyphase merge. In this example only five of the six tapes are used, in order to eliminate the time for rewinding and changing the input tape. Thus, only four-way merging is used, and the buffer allocation is like that in examples 4 and 5. A distribution like Algorithm 5.4.2D is used, but with alternating directions of runs, and with tape 1 fixed as the final output tape. First an ascending run is written on tape 1; then descending runs on tapes 2, 3, 4; then ascending runs on 2, 3, 4; then descending on 1, 2, 3; etc. Each time we switch direction, replacement selection usually produces a shorter run, so it turns out that 77 initial runs are formed instead of the 72 in examples 4 and 5.

This procedure results in a distribution of (22, 21, 19, 15) runs, and the next perfect distribution is (29, 56, 52, 44). Exercise 5.4.4–5 shows how to generate strings of merge numbers which can be used to place dummy runs in

an "optimum" way; such a procedure is feasible in practice because the finiteness of a tape reel ensures that S is never too large. Therefore the example in Chart A has been constructed using such a method for dummy run placement (see exercise 7). This turns out to be the fastest of all the examples illustrated.

Example 8. Read-backward cascade merge. As in example 7, only five tapes are used here. This procedure follows Algorithm 5.4.3C, using rewind and forward read to avoid one-way merging (since rewinding is more than twice as fast as reading on MIXT units). Distribution is therefore the same as in example 6. The pattern may be summarized briefly as follows, using $\downarrow$ to denote rewinding:

A_1^{21}	A_1^{22}	A_1^{19}	A_1^{10}	—
$A_1^4\downarrow$	$A_1^7\downarrow$	—	$D_1^2D_2^2D_3^5$	D_4^{10}
$A_8A_7^2$	A_5^2	A_9^4	—	$D_4^1\downarrow$
—	D_{17}	$A_9\downarrow$	D_{25}	D_{21}
A_{72}	—	—	—	—

Example 9. Read-backward oscillating sort. Oscillating sort with $T = 5$ (Algorithm 5.4.5B) can use buffer allocation as in examples 4, 5, 7, and 8, since it does four-way merging. However, replacement selection does not behave in the same way, since a run of length 700 (not 1400 or so) is output just before entering each merge phase, in order to clear the internal memory. Consequently 85 runs are produced in this example, instead of 72. Some of the key steps in the process are

—	A_1	A_1A_1	A_1A_1	A_1A_1
D_4	—	A_1	A_1	A_1
. . .	. . .	. . .	. . .	. . .
D_4D_4	D_4D_4	D_4D_4	D_4	—
D_4	D_4	D_4	—	A_{16}
. . .	. . .	. . .	. . .	. . .
D_4	$A_{16}D_4D_4$	$A_{16}D_4$	$A_{16}D_4A_1$	A_{16}
D_4	$A_{16}D_4D_4$	$A_{16}D_4D_1$	$A_{16}D_4$	A_{16}
—	$A_{16}D_4$	$A_{16}D_4$	A_{16}	$A_{16}A_{13}$
—	$A_{16}D_4$	A_{16}	$A_{16}A_4$	$A_{16}A_{13}$
—	A_{16}	$A_{16}A_4$	$A_{16}A_4$	$A_{16}A_{13}$
D_{37}	—	$A_{16}\downarrow$	$A_{16}\downarrow$	$A_{16}\downarrow$
—	A_{85}	—	—	—

Example 10. Read-forward oscillating sort. In the final example, replacement selection is not used because all initial runs must be the same length. There-

fore full core loads of 1000 records are internally sorted whenever an initial run is required; this makes $S = 100$. Some key steps in the process are

A_1	A_1	A_1	A_1	A_1
—	—	—	—	A_1A_4
. . .	. . .	. . .	. . .	. . .
A_1	A_1	A_1	A_1	A_1A_4
—	—	—	A_1A_4	$\cancel{A_1}A_4$
—	—	—	A_1A_4	A_1A_4
. . .	. . .	. . .	. . .	. . .
A_1	A_1A_4	A_1A_4	A_1A_4	A_1A_4
A_1A_4	$\cancel{A_1}A_4$	$\cancel{A_1}A_4$	$\cancel{A_1}A_4$	$\cancel{A_1}A_4$
$A_1A_4A_{16}$	—	—	—	—
. . .	. . .	. . .	. . .	. . .
—	A_1A_4	A_1A_4	A_1A_4	$A_1A_4A_{16}A_{64}$
A_4	$\cancel{A_1}A_4$	$\cancel{A_1}A_4$	$\cancel{A_1}A_4$	$\cancel{A_1}A_4A_{16}A_{64}$
A_4A_{16}	—	—	—	$\cancel{A_1}\cancel{A_4}A_{16}A_{64}$
$\cancel{A_4}A_{16}$	A_4	—	—	$\cancel{A_1}\cancel{A_4}A_{16}A_{64}$
—	—	—	A_{36}	$\cancel{A_1}\cancel{A_4}\cancel{A_{16}}A_{64}$
A_{100}	—	—	—	—

This routine turned out to be slowest of all, partly because it did not use replacement selection, but mostly because of its rather awkward ending (a two-way merge).

Estimating the running time. Let's see now how to figure out the approximate execution time of a sorting method using MIXT tapes. Could we have predicted the outcomes shown in Chart A without carrying out a detailed simulation?

One way that has traditionally been used to compare different merge patterns is to superimpose graphs such as we have seen in Figs. 70, 74, and 78. These graphs show the effective number of passes over the data, as a function of the number of initial runs, assuming that each initial run has approximately the same length. (See Fig. 85.) But this is *not* a very realistic comparison, because we have seen that different methods lead to different numbers of initial runs; furthermore there is a different "overhead time" caused by the relative frequency of interblock gaps, and the rewind time also has significant effects. All of these machine-dependent features make it impossible to prepare charts which make a valid comparison of the methods on a machine-independent basis. On the other hand, Fig. 85 does show us that, except for balanced merge, the effective number of passes can be reasonably well approximated by smooth curves of the form $\alpha \ln S + \beta$. Therefore we can make a fairly good comparison

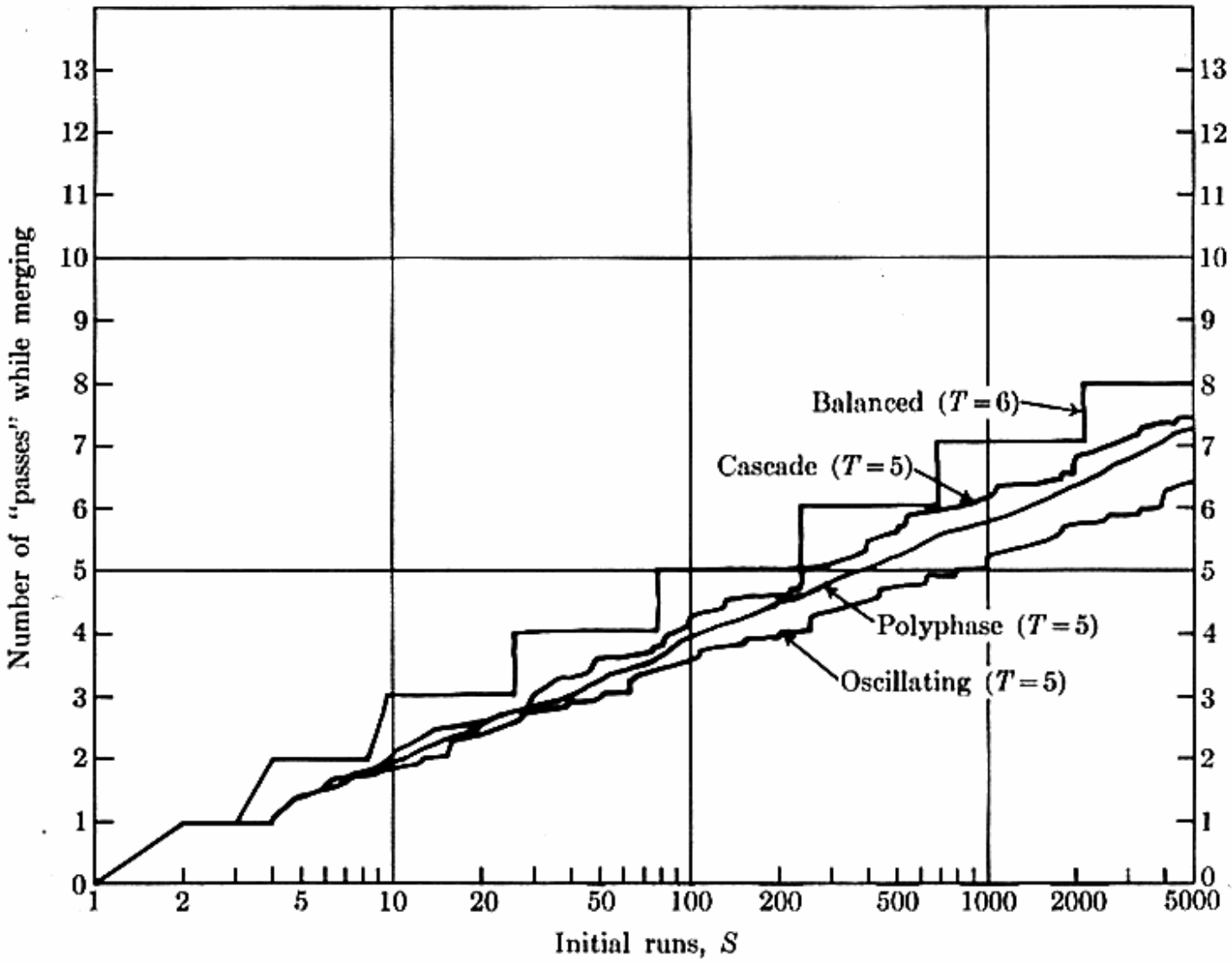


Fig. 85. A somewhat misleading way to compare merge patterns.

of the methods in any particular situation, by studying formulas that approximate the running time. Our goal, of course, is to find formulas that are simple yet sufficiently realistic.

Let us now attempt to develop such formulas, in terms of the following parameters:

N = number of records to be sorted,

C = number of characters per record,

M = number of character positions available in the internal memory (assumed to be a multiple of C),

τ = number of seconds to read or write one character,

$\rho\tau$ = number of seconds to rewind over one character,

$\sigma\tau$ = number of seconds for stop-start time delay,

γ = number of characters per interblock gap,

δ = number of seconds for operator to dismount and replace input tape,

B_i = number of characters per block in the unsorted input,

B_o = number of characters per block in the sorted output.

For MIXT we have $\tau = 1/60000$, $\rho = 2/5$, $\sigma = 300$, $\gamma = 480$. The example application treated above has $N = 100000$, $C = 100$, $M = 100000$, $B_i = B_o = 5000$, $\delta = 30$. These parameters are usually the machine and data characteristics that affect sorting time most critically (although rewind time is often given by a more complicated expression than a simple ratio ρ). Given the above parameters and a merge pattern, we shall compute further quantities such as

P = maximum order of merge in the pattern,

P' = number of records in replacement selection tree,

S = number of initial runs,

$\pi = \alpha \ln S + \beta$ = approximate average number of times each character is read and written, not counting the initial distribution or the final merge,

$\pi' = \alpha' \ln S + \beta'$ = approximate average number of times rewinding over each character during intermediate merge phases,

B = number of characters per block in the intermediate merge phases,

$\omega_i, \omega, \omega_o$ = "overhead ratio," the effective time required to read or write a character (due to gaps and stop/start) divided by the hardware time τ.

The examples of Chart A have chosen block and buffer sizes according to the formula

$$B = \left\lfloor \frac{M}{C(2P+2)} \right\rfloor C, \tag{1}$$

so that the blocks can be as large as possible consistent with the buffering scheme of Algorithm F. (In order to avoid trouble during the final pass, P should be small enough that (1) makes $B \geq B_o$.) The size of the tree during replacement selection is then

$$P' = (M - 2B_i - 2B)/C. \tag{2}$$

For random data the number of initial runs S can be estimated as

$$S \approx \left\lceil \frac{N}{2P'} + \frac{7}{6} \right\rceil, \tag{3}$$

using the results of Section 5.4.1. Assuming that $B_i < B$ and that the input tape can be run at full speed during the distribution (see below), it takes about $NC\omega_i\tau$ seconds to distribute the initial runs, where

$$\omega_i = (B_i + \gamma)/B_i. \tag{4}$$

While merging, the buffering scheme allows simultaneous reading, writing, and computing, but the frequent switching between input tapes means that we must add the stop/start time penalty; therefore we set

$$\omega = (B + \gamma + \sigma)/B, \tag{5}$$

and the merge time is approximately

$$(\pi + \rho\pi')NC\omega\tau. \tag{6}$$

This formula penalizes rewind slightly, since ω includes stop/start time, but other considerations (such as rewind interlock and the penalty for reading from load point) usually compensate for this. The final merge pass, assuming that $B_o \leq B$, is constrained by the overhead ratio

$$\omega_o = (B_o + \gamma)/B_o. \tag{7}$$

We may estimate the running time of the final merge and rewind as

$$NC(1 + \rho)\omega_o\tau;$$

in practice it might take somewhat longer due to the presence of unequal block lengths (input and output are not synchronized as in Algorithm F), but the running time will be pretty much the same for all merge patterns.

Before going into more specific formulas for individual patterns, let us try to justify two of the assumptions made above.

a) *Can replacement selection keep up with the input tape?* In the examples of Chart A it probably can, since it takes about ten iterations of the inner loop of Algorithm 5.4.1R to select the next record, and we have $C\omega_i\tau >$ 1667 microseconds in which to do this. With careful programming of the replacement selection loop, this can be done on many (but not all) machines. Note that the situation is somewhat less critical while merging: the computation time per record is almost always less than the tape time per record during a P-way merge, since P isn't very large.

b) *Should we really choose B to be the maximum possible buffer size, as in* (1)*?* A large buffer size cuts down the overhead ratio ω in (5); but it also increases the number of initial runs S, since P' is decreased. It is not immediately clear which factor is more important. Considering the merging time as a function of $x = CP'$, we can express it in the form

$$\left(\theta_1 \ln\left(\frac{N}{x} + \frac{7}{6}\right) + \theta_2\right)\left(\frac{\theta_3 - x}{\theta_4 - x}\right) \tag{8}$$

for some appropriate constants θ_1, θ_2, θ_3, θ_4, with $\theta_3 > \theta_4$. Differentiating with respect to x shows that there is some N_0 such that for all $N \geq N_0$ it does not pay to increase x at the expense of buffer size. In the sorting

application of Chart A, for example, N_0 turns out to be roughly 10000; when sorting more than 10000 records the large buffer size is superior.

Note however that with balanced merge the number of passes jumps sharply when S passes a power of P. If an approximation to N is known in advance, the buffer size should be chosen so that S will most likely be slightly less than a power of P. For example, the buffer size for the first line of Chart A was 12500; since $S = 78$, this was very satisfactory, but if S had turned out to be 82 it would have been much better to decrease the buffer size a little.

Formulas for the ten examples. Returning to Chart A, let us try to give formulas which approximate the running time in each of the ten methods. In most cases the basic formula

$$NC\omega_i\tau + (\pi + \rho\pi')NC\omega\tau + (1 + \rho)NC\omega_o\tau \tag{9}$$

will be a sufficiently good approximation to the overall sorting time, once we have specified the number of intermediate merge passes $\pi = \alpha \ln S + \beta$ and the number of intermediate rewind passes $\pi' = \alpha' \ln S + \beta'$. Sometimes it is necessary to add a further correction to (9); details for each method can be worked out as follows:

Example 1. Read-forward balanced merge. The formulas $\pi = \lceil \ln S/\ln P \rceil - 1$, $\pi' = \lceil \ln S/\ln P \rceil / P$ may be used for P-way merging on $2P$ tapes.

Example 2. Read-forward polyphase merge. We may take $\pi' \approx \pi$, since every phase is usually followed by a rewind of about the same length as the previous merge. From Table 5.4.2–1 we get the values $\alpha = 0.795$, $\beta = 0.864 - 2$, in the case of six tapes. (This "-2" is due to the fact that the table entry includes the initial and final passes as well as the intermediate ones.) The time for rewinding the input tape after the initial distribution, namely $\rho NC\omega_i\tau + \delta$, should be added to (9).

Example 3. Read-forward cascade merge. Table 5.4.3–1 gives the values $\alpha = 0.773$, $\beta = 0.808 - 2$. Rewind time is comparatively difficult to estimate; perhaps setting $\pi' \approx \pi$ is accurate enough. As in example 2, we need to add the initial rewind time to (9).

Example 4. Tape-splitting polyphase merge. Table 5.4.2–5 gives $\alpha = 0.752$, $\beta = 1.024 - 2$. The rewind time is almost overlapped except after the initialization ($\rho NC\omega_i\tau + \delta$) and two phases near the end ($2\rho NC\omega\tau$ times 36 percent). We may also subtract 0.18 from β since the first half phase is overlapped by the initial rewind.

Example 5. Cascade merge with rewind overlap. Here we use Table 5.4.3–1 for $T = 5$, to get $\alpha = 0.897$, $\beta = 0.800 - 2$. Nearly all of the unoverlapped rewind occurs just after the initial distribution and just after each two-way merge. After a perfect initial distribution, the longest tape contains about $1/g$ of the data, where g is the "growth ratio." After each two-way merge the amount of rewind in the six-tape case is $d_k\, d_{n-k}$ (cf. exercise 5.4.3–5), and it can be shown that in the T-tape case the amount of rewind after two-way merges is approximately

$$(2/(2T - 1))(1 - \cos(4\pi/(2T - 1)))$$

of the file. In our case ($T = 5$) this is $\frac{2}{9}(1 - \cos 80°) \approx 0.183$ of the file, and the number of times it occurs is $0.946 \ln S + 0.796 - 2$.

Example 6. Read-backward balanced merge. This is like example 1, except that

most of the rewinding is eliminated. The change in direction from forward to backward causes some delays, but they are not significant. There is a 50–50 chance that rewinding before the final pass will be necessary, so we may take $\pi' = 1/(2P)$.

Example 7. Read-backward polyphase merge. Since replacement selection in this case produces runs that change direction about every P times, we must replace (3) by another formula for S. A reasonably good approximation (cf. exercise 5.4.1–24) is $S = \lceil N(3 + 1/P)/6P' \rceil + 1$. All rewind time is eliminated, and Table 5.4.2–1 gives $\alpha = 0.863$, $\beta = 0.921 - 2$.

Example 8. Read-backward cascade merge. From Table 5.4.3–1 we have $\alpha = 0.897$, $\beta = 0.800 - 2$. The rewind time can be estimated as twice the difference "passes with copying" minus "passes without copying" in that table, plus $1/(2P)$ in case the final merge must be preceded by rewinding to get ascending order.

Example 9. Read-backward oscillating sort. In this case replacement selection has to be started and stopped many times; bursts of $P - 1$ to $2P - 1$ runs are distributed at a time, averaging P in length; the average length of runs therefore turns out to be approximately $P'(2P - 4/3)/P$, and we may estimate $S = \lceil N/((2 - 4/(3P))P') \rceil + 1$. A little time is used to switch from merging to distribution and vice-versa; this is approximately the time to read in P' records from the input tape, namely $P'C\omega_i\tau$, and it occurs about S/P times. Rewind time and merging time may be estimated as in example 6.

Example 10. Read-forward oscillating sort. This method is not easy to analyze, because the final "cleanup" phases performed after the input is exhausted are not as efficient as the earlier phases. Ignoring this troublesome aspect, and simply calling it one extra pass, we can estimate the merging time by setting $\alpha = 1/\ln P$, $\beta = 0$, and $\pi' = \pi/P$. The distribution of runs is somewhat different in this case, since replacement selection is not used; we set $P' = M/C$ and $S = \lceil N/P' \rceil$. With care it is possible to overlap computing, reading, and writing during the distribution, with an additional factor of about $(M + 2B)/M$ in the overhead. The "mode-switching" time mentioned in example 9 is not needed in the present case because it is overlapped by rewinding. So the estimated sorting time in this case is (9) plus $2BNC\omega_i\tau/M$.

Table 1

SUMMARY OF SORTING TIME ESTIMATES

Example	P	B	P'	S	ω	α	β	α'	β'	(9)	Additions to (9)	Estimated total	Actual total
1	3	12500	650	79	1.062	0.910	−1.000	0.303	0.000	1064		1064	1076
2	5	8300	734	70	1.094	0.795	−1.136	0.795	−1.136	1010	$\rho NC\omega_i\tau + \delta$	1113	1103
3	5	8300	734	70	1.094	0.773	−1.192	0.773	−1.192	972	$\rho NC\omega_i\tau + \delta$	1075	1127
4	4	10000	700	73	1.078	0.752	−0.994	0.000	0.720	844	$\rho NC\omega_i\tau + \delta$	947	966
5	4	10000	700	73	1.078	0.897	−1.200	0.173	0.129	972		972	992
6	3	12500	650	79	1.062	0.910	−1.000	0.000	0.167	981		981	980
7	4	10000	700	79	1.078	0.863	−1.079	0.000	0.000	922		922	907
8	4	10000	700	73	1.078	0.897	−1.200	0.098	0.117	952		952	949
9	4	10000	700	87	1.078	0.721	−1.000	0.000	0.125	846	$P'SC\omega_i\tau/P$	874	928
10	4	10000	—	100	1.078	0.721	0.000	0.180	0.000	1095	$2BNC\omega_i\tau/M$	1131	1158

Table 1 shows that the estimates are not too bad in these examples, although in a few cases there is a discrepancy of 50 seconds or so. The formulas in examples 2 and 3 indicate that cascade merge should be preferable to poly-

phase on six tapes, yet in practice polyphase was better! The reason is that graphs like Fig. 85 (which shows the five-tape case) are more nearly straight lines for the polyphase algorithm; cascade is superior to polyphase on six tapes for $14 \leq S \leq 15$ and $43 \leq S \leq 55$, near the "perfect" cascade numbers 15 and 55, but the polyphase distribution of Algorithm 5.4.2D is equal or better for all other $S \leq 100$. Cascade will win over polyphase as $S \rightarrow \infty$, but S doesn't actually approach ∞! The underestimate in example 9 is due to similar circumstances; polyphase was superior to oscillating even though the asymptotic theory tells us that oscillating will be best for large S.

Some miscellaneous remarks. It is now appropriate to make a few more or less random observations about tape merging:

1) The above formulas show that the sorting is essentially a function of N times C, not of N and C independently. Except for a few relatively minor considerations (such as the fact that B was taken to be a multiple of C), our formulas say that it takes about as long to sort one million records of 10 characters each as to sort 100 thousand records of 100 characters each. Actually there may be a difference, not revealed in our formulas, because of the space used by link fields during replacement selection. In any event the size of the *key* makes hardly any difference, unless keys get so long and complicated that internal computation cannot keep up with the tapes.

With long records and short keys it is tempting to "detach" the keys, sort them first, and then somehow rearrange the records as a whole. But this idea doesn't seem to work out, it merely postpones the agony, because the final rearrangement procedure takes about as long as a conventional merge sort would take.

2) We have seen that it takes from 15 to 19 minutes to sort 100,000 100-character records under the above assumptions. How long would it take to sort them on a card sorter? This question is of practical interest because card sorters are cheaper than computers. Assuming that each record can be squeezed onto an 80-column card, and that the alphabetic key occupies six columns, requiring an average of $1\frac{2}{3}$ passes to sort on each column, we must run each card through the machine about 10 times. At 1000 cards per minute that would take 1000 minutes, which is almost 17 hours. (During this time there is a fairly good chance that some cards will get inadvertently shuffled or jammed in the machine.)

3) When writing a sort routine that is to be used repeatedly, it is wise to estimate its running time very carefully and to compare the theory with actual observed performance. Since the theory of sorting has been fairly well developed, this procedure has been known to turn up bugs in the input/output hardware or software on existing systems; the service was slower than it should have been, yet nobody had noticed it until the sorting routine ran too slowly!

4) Some computer systems have two "banks" of tape units, attached to separate "channels" in such a way that simultaneous reading and writing is

permitted only on tapes from opposite banks. Balanced merging is especially well suited to such configurations. Consider, for example, the case of six tapes with three in each bank, and suppose that we want to do polyphase merging with $T = 6$. During a five-way merge, two of the input tapes will be on the wrong bank, so roughly two-fifths of the input time cannot be overlapped with output. This adds approximately 40 percent to the sorting time, so that balanced merge turns out to be better than polyphase even in the read-backward case.

5) Our analaysis of replacement selection has been carried out for "random" files, but the files which actually arise in practice very often have a good deal of existing order. (In fact, sometimes people will sort a file that is already in order, just to be sure.) Therefore experience has shown that replacement selection is preferable to other kinds of internal sort, even more so than our formulas indicate. This advantage is slightly mitigated in the case of read-backward polyphase sorting, since a number of descending runs must be produced; indeed, R. L. Gilstad (who first published the polyphase merge) originally rejected the read-backward technique for that reason. But he noticed later that alternating directions will still pick up long ascending runs. Furthermore, read-backward polyphase is the only standard technique which likes descending input files as well as ascending ones.

6) Another advantage of replacement selection is that it allows simultaneous reading, writing, and computing. If we merely did the internal sort in an obvious way—filling the memory, sorting it, then writing it out as it becomes filled with the next load—the distribution pass would take about twice as long!

The only other internal sort we have discussed which appears to be amenable to simultaneous reading, writing, and computing is heapsort. (This idea has been used in preparing example 10 of Chart A.) Suppose for convenience that the internal memory holds 1000 records, and that each block on tape holds 100. We may proceed as follows, letting $B_1\, B_2 \ldots B_{10}$ stand for the contents of memory divided into ten 100-record blocks.

Step 0. Fill memory, and make the elements of $B_2 \ldots B_{10}$ satisfy the inequalities for a heap (with smallest element at the root).

Step 1. Make $B_1 \ldots B_{10}$ into a heap, then select out the least 100 records and move them to B_{10}.

Step 2. Write out B_{10}, while selecting the smallest 100 records of $B_1 \ldots B_9$ and moving them to B_9.

Step 3. Read into B_{10}, and write out B_9, while selecting the smallest 100 records of $B_1 \ldots B_8$ and moving them to B_8.

⋮

Step 9. Read into B_4, and write out B_3, while selecting the smallest 100 records of $B_1\, B_2$ and moving them to B_2 and while making the heap inequalities valid in $B_5 \ldots B_{10}$.

Step 10. Read into B_3, and write out B_2, while sorting B_1 and while making the heap inequalities valid in $B_4 \ldots B_{10}$.

Step 11. Read into B_2, and write out B_1, while making the heap inequalities valid in $B_3 \ldots B_{10}$.

Step 12. Read into B_1, while making the heap inequalities valid in $B_2 \ldots B_{10}$. Return to step 1. ▮

7) We have been assuming that the number N of records to be sorted is not known in advance. Actually in most computer applications it would be possible to keep track of the number of records in all files at all times, and we could assume that our computer system is capable of telling us the value of N. How much help would this be? Unfortunately, not very much! We have seen that replacement selection is very advantageous, but it leads to an unpredictable number of initial runs. In a balanced merge we could use information about N to set the buffer size B in such a way that S will probably be just less than a power of P; and in a polyphase distribution with optimum placement of dummy runs we could use information about N to decide what level to shoot for (cf. Table 5.4.2–2).

Alternatively, without using replacement selection we could apply the preorder merge, described at the close of Section 5.4.4, making it into an oscillating sort that distributes initial runs in the appropriate direction at the appropriate moment (instead of taking them from "tape A" as described there). This method is essentially *optimum* among all methods that do internal sorting without replacement selection, since it merges according to the best possible P-ary tree; but it actually runs more slowly than methods based on replacement selection.

8) Tape drives tend to be the least reliable part of a computer. Maintenance men typically have more calls to fix tape units than any other component of a machine, and computer operators must learn how to restart jobs after a tape failure. The author has never yet seen an installation with 10 or more tape units all simultaneously in good working condition! Therefore it is considered axiomatic that *the original input tape should never be destroyed until it is known that the entire sort has been satisfactorily completed.* The "operator dismount time" is annoying in some of the examples of Chart A, but it would be too risky to overwrite the input in view of the probability that something might go wrong during a long sort.

9) When changing from forward write to backward read, we can save some time by never writing the last bufferload onto tape; it will just be read back in again anyway! Chart A shows that this trick actually saves comparatively little time, except in the oscillating sort where directions are reversed frequently.

10) If we have a large number of tape units available, it is not always best to use as many of them as possible in order to get a "high order of merge." For example, a higher order of merge usually implies a smaller block size; and

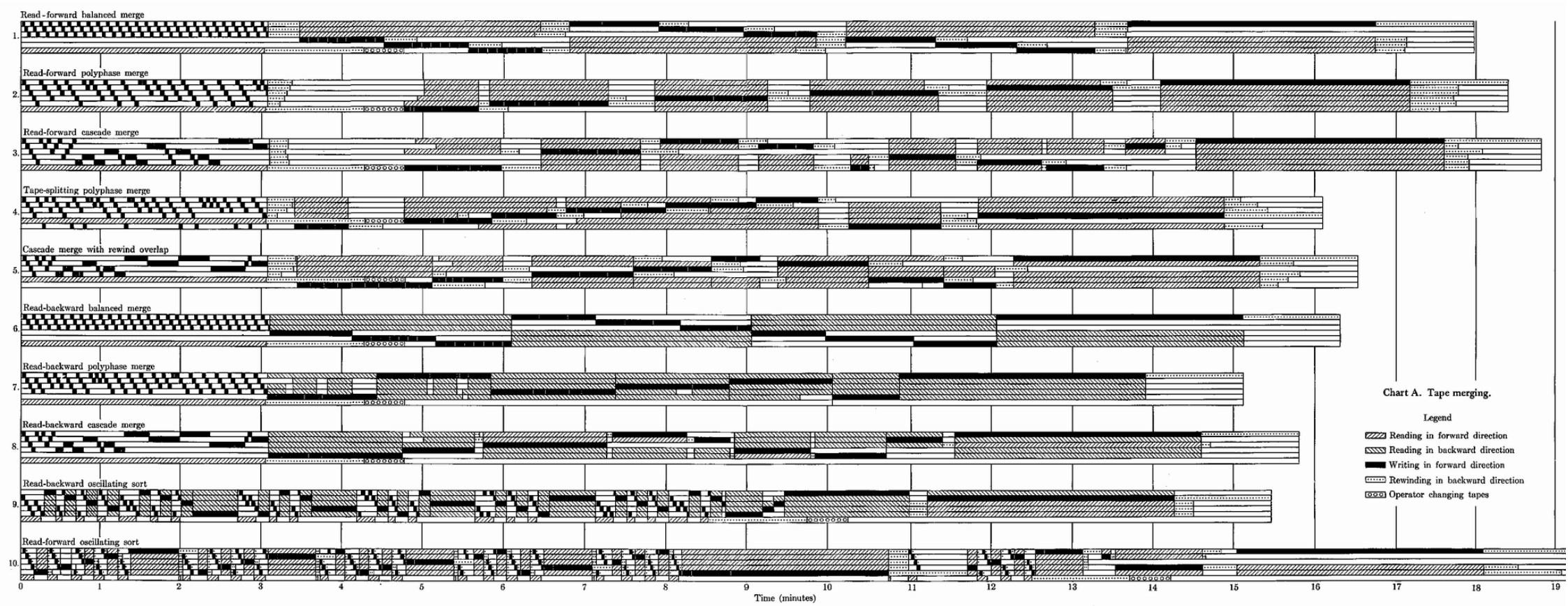


Chart A. Tape merging.

the percentage difference between $\log_P S$ and $\log_{P+1} S$ is not very great when P is large. Consider also the poor computer operator who has to mount all those scratch tapes. On the other hand, exercise 12 describes an interesting way to make use of additional tape units, grouping them so as to overlap input/output time without increasing the order of merge.

11) On machines like MIX, which have fixed rather small block sizes, hardly any internal memory is needed while merging. Oscillating sort then becomes more attractive, because it becomes possible to maintain the replacement selection tree in memory while merging. In fact we can improve on oscillating sort in this case (as suggested by Colin J. Bell in 1962), merging a new initial run into the output every time we merge from the working tapes!

12) We have observed that multireel files should be sorted one reel at a time, in order to avoid excessive tape handling. This is sometimes called a "reel time" application. Actually a balanced merge on six tapes can sort *three* reelfulls, up until the time of the final merge, if it has been programmed carefully.

In order to merge a fairly large number of individually sorted reels, a minimum-path-length merging tree will be fastest (cf. Section 5.4.4). This construction was first made by E. H. Friend [*JACM* **3** (1956), 166–167]; then W. H. Burge [*Information and Control* **1** (1958), 181–197] pointed out that an optimum way to merge runs of given (possibly unequal) lengths is obtained by constructing a tree with minimum *weighted* path length, using the run lengths as weights (cf. Sections 2.3.4.5 and 5.4.9), if we ignore tape handling time. But multireel files should probably be kept on a disk or other large memory instead of tape.

13) Our discussions have blithely assumed that we have direct control over the input/output instructions for tape units, and that no complicated system interface keeps us from using tape as efficiently as the tape designers intended. These idealistic assumptions give us insights into the tape merging problem, and may give some insights into the proper design of operating systems, but we should realize that multiprogramming and multiprocessing can make the situation considerably more complicated.

14) The issues we have studied in this section were first discussed in print by E. H. Friend [*JACM* **3** (1956), 134–168], W. Zoberbier [*Elektron. Datenverarb.* **5** (1960), 28–44], and M. A. Goetz [*Digital Computer User's Handbook* (New York: McGraw-Hill, 1967) 1.292–1.320].

Summary. We can sum up what we have learned about comparing different merge patterns in the following way:

Theorem A. *It is difficult to decide which merge pattern is best in a given situation.* ▮

The examples we have seen in Chart A show how 100,000 randomly ordered 100-character records (or 1 million 10-character records) might be sorted using

six tapes under realistic assumptions. This much data fills about half of a tape, and it can be sorted in about 15 to 19 minutes; however, there is considerable variation in available tape equipment, and running times for such a job may vary between about four minutes and about two hours on different machines. About 3 minutes of the time in our examples were used for initial distribution of runs and internal sorting; about $4\frac{1}{2}$ minutes were used for the final merge and rewinding the output tape; and about $7\frac{1}{2}$ to $11\frac{1}{2}$ minutes were spent in intermediate stages of merging.

Given six tapes that cannot read backward, the best sorting method under our assumptions was the "tape-splitting polyphase merge" (example 4); and for tapes that do allow backward reading, the best method turned out to be read-backward polyphase with a complicated placement of dummy runs (example 7). Oscillating sort (example 9) was a close second. In both cases the cascade merge provided a simpler alternative which was only slightly slower (examples 5 and 8). In the read-forward case, a straightforward balanced merge (example 1) was surprisingly effective, partly by luck in this particular example but partly also because it spends comparatively little time rewinding.

The situation would change somewhat if we had a different number of available tapes.

Sort generators. Given the wide variability of data and equipment characteristics, it is almost impossible to write a single external sorting program that is satisfactory in very many applications. And it is also rather difficult to prepare a program that really handles tapes efficiently. Therefore the preparation of sorting software is a particularly challenging job. A *sort generator* is a program that produces machine code specially tailored to particular sorting applications, based on parameters which describe the data format and the hardware configuration. Such a program is often tied to high-level languages such as COBOL or PL/I, or it might be written as a set of macros to be used with a macroassembler.

One of the features normally provided by a sort generator is the ability to insert "own coding," special instructions to be incorporated into the first and last passes of the sorting routine. First-pass own coding is usually used to edit the input records, often shrinking them or slightly expanding them into a form which is easier to sort. For example, suppose that the input records are to be sorted on a nine-character key which represents a date in month-day-year format:

```
JUL041776  OCT311517  NOV051605  JUL141789  NOV201917
```

On the first pass the three-letter month code can be looked up in a table, and the month codes can be replaced by numbers with the most significant fields at the left:

```
17760704  15171031  16051105  17890714  19171120
```

This decreases the record length and makes subsequent comparisons much

simpler. (An even more compact code could also be substituted.) Last-pass own coding can be used to restore the original format, and/or to make other desired changes to the file, and/or to compute some function of the output records. The merging algorithms we have studied are organized in such a way that it is easy to distinguish the last pass from other merge phases. Note that when own coding is present there must be at least two passes over the file even if it is initially in order. Own coding which changes the record size can make it difficult for the oscillating sort to overlap some of its input/output operations.

Sort generators also take care of system details like tape label conventions, and they often provide for "hash totals" or other checks to make sure that none of the data has been lost or altered. Sometimes there are provisions for stopping the sort at convenient places and resuming later. The fanciest generators allow records to have dynamically varying lengths [cf. D. J. Waks, *CACM* **6** (1963), 267–272].

***Merging with fewer buffers.** We have seen that $2P + 2$ buffers are sufficient to keep tapes moving rapidly during a P-way merge. Let us conclude this section by making a mathematical analysis of the merging time when *less* than $2P + 2$ buffers are present.

Two output buffers are clearly desirable, since we can be writing from one while forming the next block of output in the other. Therefore we may ignore the output question entirely, and concentrate only on the input.

Suppose there are $P + Q$ input buffers, where $1 \le Q \le P$. We shall use the following approximate model of the situation, as suggested by L. J. Woodrum [*IBM Systems J.* **9** (1970), 118–144]: It takes one unit of time to read a block of tape. During this time there is a probability p_0 that no input buffers have been emptied, p_1 that one has been emptied, $p_{\ge 2}$ that two or more have been, etc. When completing a tape read we are in one of $Q + 1$ states:

State 0. Q buffers are empty; we begin to read a block into one of them from the appropriate file, using the forecasting technique explained earlier in this section. After one unit of time we go to state 1 with probability p_0, otherwise we remain in state 0.

State 1. $Q - 1$ buffers are empty; we begin to read into one of them, forecasting the appropriate file. After one unit of time we go to state 2 with probability p_0, to state 1 with probability p_1, and to state 0 with probability $p_{\ge 2}$.

$\vdots$

State $Q - 1$. One buffer is empty; we begin to read into it, forecasting the appropriate file. After one unit of time we go to state Q with probability p_0, to state $Q - 1$ with probability $p_1, \ldots,$ to state 1 with probability p_{Q-1}, and to state 0 with probability $p_{\ge Q}$.

State Q. All buffers are filled. Tape reading stops for an average of μ units of time and then we go to state 1.

We start in state 0. This model of the situation corresponds to a *Markov process* (cf. exercise 2.3.4.2–26), which can be analyzed via generating functions in the following interesting way: Let z be an arbitrary parameter, and assume that each time we have a chance to read from tape we make a decision to do so with probability z, but we decide to terminate the algorithm with probability $1 - z$. Now let $g_Q(z) = \sum_{n\geq 0} a_n^{(Q)} z^n (1 - z)$ be the average number of times that state Q occurs in such a process; it follows that $a_n^{(Q)}$ is the average number of times state Q occurs when exactly n blocks have been read. Then $n + a_n\mu$ is the average total time for input + computation. If we had perfect overlap, as in the $(2P + 2)$-buffer algorithm, the total time would be only n units, so $a_n\mu$ represents the "reading hangup" time.

Let A_{ij} be the probability that we go from state i to state j in this process, for $0 \leq i, j \leq Q + 1$, where $Q + 1$ is a new "stopped" state. For example, the A-matrix takes the following forms for small Q:

$$Q = 1: \quad \begin{pmatrix} p_{\geq 1}z & p_0 z & 1 - z \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$Q = 2: \quad \begin{pmatrix} p_{\geq 1}z & p_0 z & 0 & 1 - z \\ p_{\geq 2}z & p_1 z & p_0 z & 1 - z \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$Q = 3: \quad \begin{pmatrix} p_{\geq 1}z & p_0 z & 0 & 0 & 1 - z \\ p_{\geq 2}z & p_1 z & p_0 z & 0 & 1 - z \\ p_{\geq 3}z & p_2 z & p_1 z & p_0 z & 1 - z \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Exercise 2.3.4.2–26b tells us that $g_Q(z) = \text{cofactor}_{Q0}(I - A)/\det(I - A)$. Thus for example when $Q = 1$ we have

$$g_1(z) = \det\begin{pmatrix} 0 & -p_0 z & z - 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \Big/ \det\begin{pmatrix} 1 - p_{\geq 1}z & -p_0 z & z - 1 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \frac{p_0 z}{1 - p_1 z - p_0 z} = \frac{p_0 z}{1 - z} = \sum_{n\geq 0} n p_0 z^n (1 - z),$$

so $a_n^{(1)} = np_0$. This of course was obvious *a priori*, since the problem is very simple when $Q = 1$. A similar calculation when $Q = 2$ (see exercise 14) gives the less obvious formula

$$a_n^{(2)} = \frac{p_0^2 n}{1 - p_1} - \frac{p_0^2(1 - p_1^n)}{(1 - p_1)^2}. \tag{10}$$

In general we can show that $a_n^{(Q)}$ has the form $\alpha^{(Q)}n + O(1)$ as $n \to \infty$, where the constant $\alpha^{(Q)}$ is not terribly difficult to calculate. (See exercise 15.) It turns out that $\alpha^{(3)} = p_0^3/((1 - p_1)^2 - p_0p_2)$.

The nature of merging makes it fairly reasonable to assume that $\mu = 1/P$ and that we have a binomial distribution

$$p_k = \binom{P}{k}\left(\frac{1}{P}\right)^k\left(\frac{P-1}{P}\right)^{P-k}.$$

For example, when $P = 5$ we have $p_0 = .32768$, $p_1 = .4096$, $p_2 = .2048$, $p_3 = .0512$, $p_4 = .0064$, and $p_5 = .00032$; hence $\alpha^{(1)} = 0.328$, $\alpha^{(2)} = 0.182$, and $\alpha^{(3)} = 0.127$. In other words, if we use $5 + 3$ input buffers instead of $5 + 5$, we can expect an additional "reading hangup" time of about $0.127/5 \approx 2.5$ percent.

Of course this model is only a very rough approximation; we know that when $Q = P$ there is no hangup time at all, but the model says there is. The extra reading hangup time for smaller Q just about counterbalances the savings in overhead gained by having larger blocks, so the simple $Q = P$ scheme seems to be vindicated.

EXERCISES

1. [*13*] Give a formula for the exact number of characters per tape, when every block on the tape contains n characters. Assume that the tape could hold exactly 23000000 characters if there were no interblock gaps.

2. [*15*] Explain why the first buffer for File 2, in line 6 of Fig. 84, is completely blank.

3. [*20*] Would Algorithm F work properly if there were only $2P - 1$ input buffers instead of $2P$? If so, prove it; if not, give an example where it fails.

4. [*20*] How can Algorithm F be changed so that it works also when $P = 1$?

▶ 5. [*21*] When equal keys are present on different files, it is necessary to be very careful in the forecasting process. Explain why, and show how to avoid difficulty by defining the merging and forecasting operations of Algorithm F more precisely.

6. [*22*] What changes should be made to Algorithm 5.4.3C, in order to convert it into an algorithm for cascade merge *with rewind overlap*, on $T + 1$ tapes?

▶ 7. [*26*] The initial distribution in example 7 of Chart A produces

$$(A_1D_1)^{11} \qquad D_1(A_1D_1)^{10} \qquad D_1(A_1D_1)^{9} \qquad D_1(A_1D_1)^{7}$$

on tapes 1–4, where $(A_1D_1)^7$ means $A_1D_1A_1D_1A_1D_1A_1D_1A_1D_1A_1D_1A_1D_1$. Show how to insert additional A_0's and D_0's in a "best possible" way (in the sense that the overall number of initial runs processed while merging is minimized), bringing the distribution up to

$$A(DA)^{14} \qquad (DA)^{28} \qquad (DA)^{26} \qquad (DA)^{22}.$$

Hint: To preserve parity it is necessary to insert many of the A_0's and D_0's as adjacent

pairs. The merge numbers for each initial run may be computed as in exercise 5.4.4–5; some simplification occurs since adjacent runs always have adjacent merge numbers.

8. [*20*] Chart A shows that most of the schemes for initial distribution of runs (with the exception of the initial distribution for the cascade merge) tend to put consecutive runs onto different tapes. If consecutive runs went onto the same tape we could save the stop/start time; would it therefore be a good idea to modify the distribution algorithms so that they switch tapes less often?

▶ **9.** [*22*] Estimate how long the read-backward polyphase algorithm would have taken in Chart A, if we had used all $T = 6$ tapes for sorting, instead of $T = 5$ as in example 7. Was it wise to avoid using the input tape?

10. [*M23*] Use the analyses in Sections 5.4.2 and 5.4.3 to show that the length of each rewind during a standard six-tape polyphase or cascade merge is rarely more than about 54 percent of the file (except for the initial and final rewinds, which cover the entire file).

11. [*23*] By modifying the appropriate entries in Table 1, estimate how long the first nine examples of Chart A would have taken if we had a combined low speed/high speed rewind. Assume that $\rho = 1$ when the tape is less than about one-fourth full, and that the rewind time for fuller tapes is approximately five seconds plus the time which would be obtained for $\rho = \frac{1}{5}$. Change example 8 so that it uses cascade merge *with* copying, since rewinding and reading forward is slower than copying in this case. [*Hint:* Use the result of exercise 10.]

12. [*40*] Consider partitioning six tapes into three pairs of tapes, with each pair playing the role of a single tape in a polyphase merge with $T = 3$. One tape of each pair will contain blocks 1, 3, 5, . . . and the other tape will contain blocks 2, 4, 6, . . . ; in this way we can essentially have two input tapes and two output tapes active at all times while merging, effectively doubling the merging speed. (a) Find an appropriate way to extend Algorithm F to this situation. How many buffers should there be? (b) Estimate the total running time which would be obtained if this method were used to sort 100,000 100-character records, considering both the read-forward and read-backward cases.

13. [*20*] Can a five-tape oscillating sort, as defined in Algorithm 5.4.5B, be used to sort four reelfulls of input data, up until the time of the final merge?

14. [*M19*] Derive (10).

15. [*HM29*] Prove that $g_Q(z) = h_Q(z)/(1 - z)$, where $h_Q(z)$ is a rational function of z having no singularities inside the unit circle; hence $a_n^{(Q)} = h_Q(1)n + O(1)$ as $n \to \infty$. In particular, show that

$$h_3(1) = \det\begin{pmatrix} 0 & -p_0 & 0 & 0 \\ 0 & 1-p_1 & -p_0 & 0 \\ 0 & -p_2 & 1-p_1 & -p_0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \Big/ \det\begin{pmatrix} 1 & -p_0 & 0 & 0 \\ 1 & 1-p_1 & -p_0 & 0 \\ 1 & -p_2 & 1-p_1 & -p_0 \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

16. [*M46*] Analyze merging with $P + Q$ input buffers more carefully than is done in the text, using a more accurate model.

17. [*41*] Carry out detailed studies of the problem of sorting 100,000 100-character records, drawing charts such as those in Chart A, assuming that 3, 4, or 5 tapes are available.

*5.4.7. External Radix Sorting

The previous sections have discussed the process of tape sorting by merging; but there is another way to sort with tapes, based on the radix sorting principle used in mechanical card sorters (cf. Section 5.2.5). This method is sometimes called distribution sorting, column sorting, pocket sorting, digital sorting, separation sorting, etc.; it turns out to be essentially the *opposite* of merging!

Suppose, for example, that we have four tapes and that there are only eight possible keys: 0, 1, 2, 3, 4, 5, 6, 7. If the input data is on tape T1, we can begin by transferring all even keys to T3, all odd keys to T4:

	T1	T2	T3	T4
Given	{0, 1, 2, 3, 4, 5, 6, 7}	—	—	—
Pass 1	—	—	{0, 2, 4, 6}	{1, 3, 5, 7}

Now we rewind, and read T3 and then T4, putting {0, 1, 4, 5} on T1 and {2, 3, 6, 7} on T2:

Pass 2	{0, 4} {1, 5}	{2, 6} {3, 7}	—	—

(The notation "{0, 4} {1, 5}" stands for a file which contains some records whose keys are all 0 or 4, followed by records whose keys are all 1 or 5. Note that T1 now contains those keys whose middle binary digit is 0.) After rewinding again and distributing 0, 1, 2, 3 to T3 and 4, 5, 6, 7 to T4, we have

Pass 3			{0} {1} {2} {3}	{4} {5} {6} {7}

Now we can finish up by copying T4 to the end of T3. In general, if the keys range from 0 to $2^k - 1$, we could sort the file in an analogous way using k passes, followed by a final "collection" phase which copies about half of the data from one tape to another. With six tapes we could use radix 3 representations in a similar way, to sort keys from 0 to $3^k - 1$ in k passes.

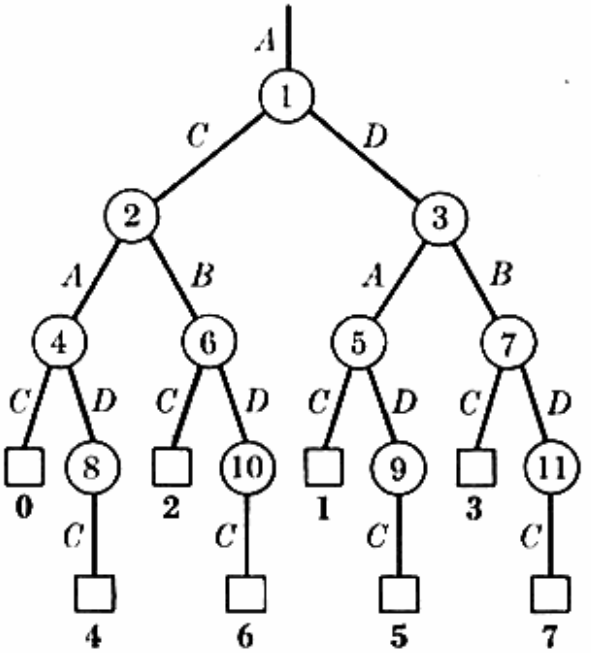
Partial-pass methods can also be used. For example, suppose that there are ten possible keys {0, 1, . . . , 9}, and consider the following procedure due to R. L. Ashenhurst [*Theory of Switching* 7 (Harvard Univ. Comp. Laboratory: May, 1954), I.1–I.76]:

	T1	T2	T3	T4	
Given	{0, 1, . . . , 9}	—	—	—	
Phase 1.	—	{0, 2, 4, 7}	{1, 5, 6}	{3, 8, 9}	1.0 pass
Phase 2.	{0}	—	{1, 5, 6} {2, 7}	{3, 8, 9} {4}	0.4 pass
Phase 3.	{0} {1} {2}	{6} {7}	—	{3, 8, 9} {4} {5}	0.5 pass
Phase 4.	{0} {1} {2} {3}	{6} {7} {8}	{9}	{4} {5}	0.3 pass
Collection	{0} {1} {2} {3} {4} . . . {9}				0.6 pass
					2.8 passes

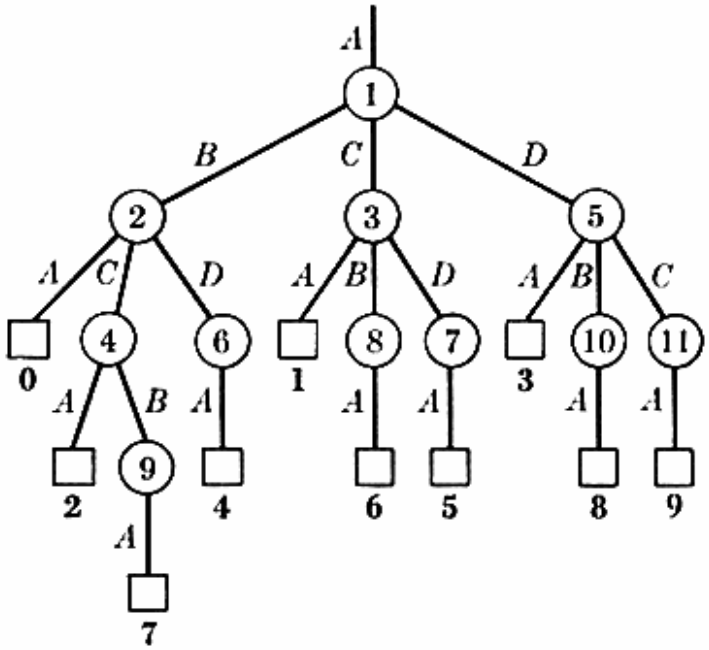
If each key value occurs about one-tenth of the time, the above procedure takes only 2.8 passes to sort ten keys, while the first example required 3.5 passes to sort only eight keys. Therefore we find that a clever distribution pattern can make a significant difference, for radix sorting as well as for merging.

The distribution patterns in the above examples can conveniently be represented as tree structures:

Example 1



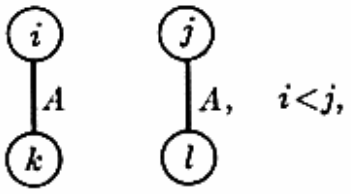
Example 2



The circular internal nodes of these trees are numbered 1, 2, 3, . . . , corresponding to steps 1, 2, 3, . . . of the process. Tape names A, B, C, D (instead of T1, T2, T3, T4) have been placed next to the lines of the trees, in order to show where the records go. Square external nodes represent portions of a file which contain only one key, and that key is shown in boldface type just below the node. The lines just above square nodes all carry the name of the output tape (C in the first example, A in the second).

Thus, step 3 of example 1 consists of reading from tape D and writing 1's and 5's on tape C, 3's and 7's on tape D. It is not difficult to see that the number of passes performed is equal to the *external path length* of the tree divided by the number of external nodes, if we assume that each key occurs equally often.

Because of the sequential nature of tape, and the first-in-first-out discipline of forwards reading, we can't simply use *any* labeled tree as the basis of a distribution pattern. In the tree of example 1, data gets written on tape A during step 2 and step 3; it is necessary to use the data written during step 2 before we use the data written during step 3. In general if we write onto a tape during steps i and j, where $i < j$, we must use the data written during step i first; when the tree contains two branches of the form



we must have $k < l$. Furthermore we cannot write anything onto tape A *between* steps k and l, because we must rewind between reading and writing.

The reader who has worked the exercises of Section 5.4.4 will now immediately perceive that the allowable trees for read-forward radix sorting on T' tapes are precisely the "strongly T-fifo" trees which characterize read-forward *merge* sorting on T tapes! (See exercise 5.4.4–20.) The only difference is that all of the external nodes on the trees we are considering here have the same tape labels. We could remove this restriction by assuming a final "collection" phase which transfers all records to an output tape, or we could add that restriction to the rules for T-fifo trees by requiring that the initial distribution pass of a merge sort be explicitly represented in the corresponding merge tree.

In other words, *every merge pattern corresponds to a distribution pattern, and every distribution pattern corresponds to a merge pattern.* A moment's reflection shows why this is so: Consider running a merge sort backwards, "unmerging" the final output into subfiles which are unmerged into others, etc.; finally we will have unmerged the file into S runs. Such a pattern is possible with tapes if and only if the corresponding radix sort distribution pattern, for S keys, is possible. This duality between merging and distribution is almost perfect; it breaks down only in one respect, namely that the input tape must be saved at different times.

The eight-key example treated at the beginning of this section is clearly dual to a balanced merge on four tapes. The ten-key example with partial passes corresponds to the following ten-run merge pattern (if we suppress the copy phases, steps 6–11 in the tree):

	T1	T2	T3	T4
Initial distribution	1^4	1^3	1^1	1^2
Tree step 5	1^3	1^2	—	$1^2 3^1$
Tree step 4	1^2	1^1	2^1	$1^2 3^1$
Tree step 3	1^1	—	$2^1 3^1$	$1^1 3^1$
Tree step 2	—	4^1	3^1	3^1
Tree step 1	10^1	—	—	—

If we compare this to the radix sort, we see that the methods have essentially the same structure but are reversed in time, with the tape contents also reversed from back to front: $1^2 3^1$ (two runs each of length 1 followed by one of length 3) corresponds to $\{3, 8, 9\}\{4\}\{5\}$ (two subfiles each containing 1 key preceded by one subfile containing 3).

Going the other way, we can in principle construct a radix sort which is dual to polyphase merge, cascade merge, etc. For example, the 21-run polyphase merge on three tapes, illustrated at the beginning of Section 5.4.2, corresponds to the following interesting radix sort:

	T1	T2	T3
Given	{0, 1, . . . , 20}	—	—
Phase 1.	—	{0, 2, 4, 5, 7, 9, 10, 12, 13, 15, 17, 18, 20}	{1, 3, 6, 8, 11, 14, 16, 19}
Phase 2.	{0, 5, 10, 13, 18}	—	{1, 3, 6, 8, 11, 14, 16, 19} {2, 4, 7, 9, 12, 15, 17, 20}
Phase 3.	{0, 5, 10, 13, 18} {1, 6, 11, 14, 19} {2, 7, 12, 15, 20}	{3, 8, 16} {4, 9, 17}	—
Phase 4.	—	{3, 8, 16} {4, 9, 17} {5, 10, 18} {6, 11, 19} {7, 12, 20}	{0, 13} {1, 14} {2, 15}
Phase 5.	{8} {9} {10} {11} {12}	—	{0, 13} {1, 14} {2, 15} {3, 16} . . . {7, 20}
Phase 6.	{8} {9} {10} {11} {12} {13} . . . {20}	{0} {1} . . . {7}	—

The distribution rule used here to decide which keys go on which tapes at each step appears to be magic, but in fact it has a simple connection with the Fibonacci number system! (See exercise 2.)

Reading backward. Duality between radix sorting and merging applies also to algorithms that read tapes backward. We have defined "T-lifo trees" in Section 5.4.4, and it is easy to see that they correspond to radix sorts as well as to merge sorts.

A read backward radix sort was actually considered by John Mauchly already in 1946, in one of the first papers ever to be published about sorting (cf. Section 5.5); Mauchly essentially gave the following construction:

	T1	T2	T3	T4
Given	—	{0, 1, 2, . . . , 9}	—	—
Phase 1.	{4, 5}	—	{2, 3, 6, 7}	{0, 1, 8, 9}
Phase 2.	{4, 5} {2, 7}	{3, 6}	—	{0, 1, 8, 9}
Phase 3.	{4, 5} {2, 7} {0, 9}	{3, 6} {1, 8}	—	—
Phase 4.	{4, 5} {2, 7}	{3, 6} {1, 8}	{9}	{0}
. . . . .				
Phase 8.	—	—	{9} {8} {7} {6} {5}	{0} {1} {2} {3} {4}
Final collection	—	—	—	{0} {1} {2} {3} {4} {5} . . . {9}

This scheme is not the most efficient one possible, but it is interesting because it shows that partial pass methods were considered for radix sorting already in 1946, although they did not appear in the literature for merging until about 1960.

An efficient construction of read-backward distribution patterns has been suggested by A. Bayes [*CACM* **11** (1968), 491–493]: Given $P + 1$ tapes and S keys, divide the keys into P subfiles each containing $\lfloor S/P \rfloor$ or $\lceil S/P \rceil$ keys, and apply this procedure recursively to each subfile. When $S < 2P$, one subfile should consist of the smallest key alone, and it should be written onto the output file. (R. M. Karp's general preorder construction, which appears at the end of Section 5.4.4, includes this method as a special case.)

Backward reading makes merging a little more complicated because it reverses the order of runs. There is a corresponding effect on radix sorting: the outcome is stable or "anti-stable" depending on what level is reached in the tree. After a read-backward radix sort in which some of the external nodes are at odd levels and some are at even levels, the relative order of different

records with equal keys will be the *same* as the original order for some keys, but it will be the *opposite* of the original order for the other keys. (Cf. exercise 6.)

Oscillating merge sorts have their counterparts too, under duality. In an *oscillating radix sort* we continue to separate out the keys until reaching subfiles that have only one key or are small enough to be internally sorted; such subfiles are sorted and written onto the output tape, then the separation process is resumed. For example, if we have three work tapes and one output tape, and if the keys are binary numbers, we may start by putting keys of the form '0x' on tape T1, keys '1x' on T2. If T1 receives more than one memory load, we scan it again and put '00x' on T2 and '01x' on T3. Now if the '00x' subfile is short enough to be internally sorted, we do so and output the result, then continue by processing the '01x' subfile. Such a method was called a "cascading pseudo-radix sort" by E. H. Friend [*JACM* **3** (1956), 157–159]; it has been further developed by H. Nagler [*JACM* **6** (1959), 459–468], who gave it the colorful name "amphisbaenic sort," and by C. H. Gaudette [*IBM Tech. Disclosure Bull.* **12** (April, 1970), 1849–1853].

Does radix sorting beat merging? One important consequence of the duality principle is that *radix sorting is usually inferior to merge sorting.* This happens because the technique of replacement selection gives merge sorting a definite advantage; there is no apparent way to arrange radix sorts so that we can make use of internal sorts encompassing more than one memory load at a time. Indeed, the oscillating radix sort will often produce subfiles which are somewhat smaller than one memory load, so the distribution pattern corresponds to a tree with many more external nodes than would be present if merging and replacement selection were used. Consequently the external path length of the tree (i.e., the sorting time) is increased. (Cf. exercise 5.3.1–33.)

There are, however, some important applications for external radix sorting. Suppose for example that we have a file containing the names of all employees of a large corporation, in alphabetic order; the corporation has 10 divisions, and it is desired to sort the file by division, *retaining* the alphabetic order of the employees in each division. This is perfect situation in which to apply a stable radix sort, if the file is long, since the number of records which belong to each of the 10 divisions is likely to be more than the number of records which would be obtained in initial runs produced by replacement selection. In general, if the range of key values is so small that the collection of records having a given key would fill the internal memory more than twice, it is wise to use a radix sort technique.

We have seen in Section 5.2.5 that *internal* radix sorting is superior to merging, on certain high-speed computers, because the "inner loop" of the radix sort algorithm avoids complicated branching. If the external memory is especially fast, it may be impossible for such machines to merge data rapidly enough to keep up with the input/output equipment. Radix sorting may therefore turn out to be superior to merging in such a situation, especially if the keys are known to be uniformly distributed.

EXERCISES

1. [*20*] The general T-tape balanced merge with parameter P, $1 \leq P < T$, was defined near the beginning of Section 5.4. Show that this corresponds to a radix sort which uses a mixed radix system.

2. [*M28*] The text illustrates the three-tape polyphase radix sort for 21 keys. Generalize to the case of F_n keys; explain what keys appear on what tapes at the end of each phase. [*Hint:* Consider the Fibonacci number system, exercise 1.2.8–34.]

3. [*M40*] Extend the results of exercise 2 to the polyphase merge on four or more tapes. (Cf. exercise 5.4.2–10.)

4. [*M20*] Prove that Ashenhurst's distribution pattern is the best way to sort 10 keys on four tapes without reading backwards, in the sense that the associated tree has minimum external path length over all "strongly 4-fifo trees." (Thus, it is essentially the best method if we ignore rewind time.)

5. [*15*] Draw the 4-lifo tree corresponding to Mauchly's read-backwards radix sort for 10 keys.

▶ 6. [*20*] A certain file contains two-digit keys 00, 01, . . . , 99. After performing Mauchly's radix sort on the unit's digit, we can repeat the same scheme on the ten's digits, interchanging the roles of tapes T2 and T4. In what order will the keys finally appear on T2?

7. [*21*] Does the duality principle apply also to multi-reel files?

*5.4.8. Two-Tape Sorting

Since we need three tapes to carry out a merge process without excessive tape motion, it is interesting to speculate about how we could perform a reasonable external sort using only two tapes.

One approach, suggested by H. B. Demuth in 1956, is sort of a combined replacement-selection and bubble sort. Assume that the input is on tape T1, and begin by reading $P + 1$ records into memory. Now output the record whose key is smallest, to tape T2, and replace it by the next input record. Continue outputting a record whose key is currently the smallest in memory, maintaining a selection tree or a priority queue of $P + 1$ elements. When the input is finally exhausted, the largest P keys of the file will be present in memory; output them is ascending order. Now rewind both tapes and repeat the process by reading from T2 and writing to T1; each such pass puts at least P more records into their proper place. A simple test can be built into the program which determines when the entire file is in sort. At most $\lceil (N - 1)/P \rceil$ passes will be necessary.

A few moment's reflection shows that each pass of this procedure is essentially equivalent to P consecutive passes of the bubble sort (Algorithm 5.2.2B)! If an element has P or more inversions, it will be smaller than everything in the tree when it is input, so it will be output immediately (thereby

losing P inversions). If an element has less than P inversions, it will go into the selection tree and will be output before all greater keys (thereby losing all its inversions). When $P = 1$, this is exactly what happens in the bubble sort, by Theorem 5.2.2I.

The total number of passes will therefore be $\lceil I/P \rceil$, where I is the maximum number of inversions of any element. By the theory developed in Section 5.2.2, the average value of I is $N - \sqrt{\pi N/2} + \frac{2}{3} + O(1/\sqrt{N})$.

If the file is not too much larger than the memory size, or if it is nearly in order to begin with, this order-P bubble sort will be fairly rapid; in fact, it might be advantageous to use it even when extra tape units are available, since it might not take as long as would be required for the computer operator to mount a third tape! On the other hand, it will run quite slowly on fairly long, randomly ordered files, since its running time is approximately proportional to N^2.

Let us consider how this method might be implemented for the 100,000-record example of Section 5.4.6. We need to choose P intelligently, in order to compensate for interblock gaps while doing simultaneous reading, writing, and computing. Since the example assumes that each record is 100 characters long and that 100,000 characters will fit into memory, we can make room for two input buffers and two output buffers of size B by setting

$$100(P + 1) + 4B = 100000. \tag{1}$$

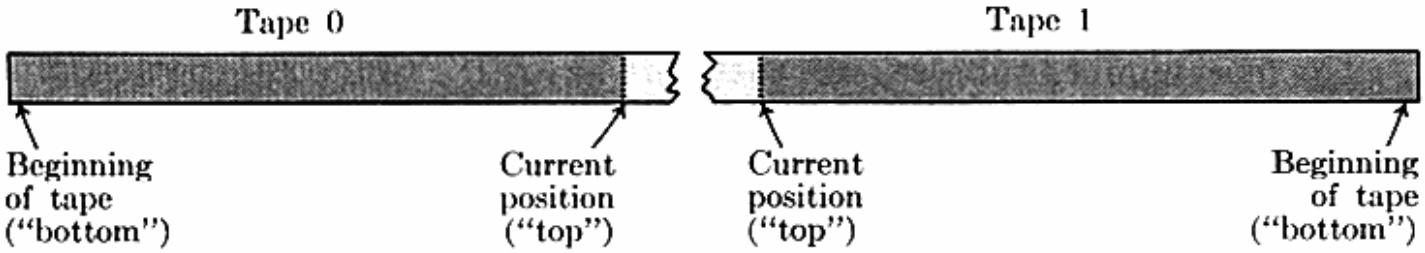
Using the notation of Section 5.4.6, the running time for each pass will be about

$$NC\omega\tau(1 + \rho), \qquad \omega = (B + \gamma)/B. \tag{2}$$

Since the number of passes is inversely proportional to P, we want to choose B to be a multiple of 100 which minimizes the quantity ω/P. Elementary calculus shows that this occurs when B is approximately $\sqrt{24975\gamma} + \gamma^2 - \gamma$, so we take $B = 3000$, $P = 879$. Setting $N = 100000$ in the above formulas shows that the number of passes $\lceil I/P \rceil$ will be about 114, and the total estimated running time will be approximately 8.57 hours (assuming for convenience that the initial input and the final output also have $B = 3000$). This represents approximately 0.44 of a reelfull of data; a full reel would take about five times as long. Some improvements could be made if the algorithm were interrupted periodically, writing the records with largest keys onto an auxiliary tape which is dismounted, since such records are merely copied back and forth once they have been put into order.

Application of quicksort. Another internal sorting method which traverses the data in a nearly sequential manner is the partition exchange or quicksort procedure, Algorithm 5.2.2Q. Can we adapt it to two tapes? [N. B. Yoash, *CACM* **8** (1965), 649.]

It is not difficult to see how this can be done, using backward reading. Assume that the two tapes are numbered 0 and 1, and imagine that the file is laid out as follows:



Each tape serves as a stack; putting them together like this makes it possible to view the file as a linear list in which we can move the current position left or right by copying from one stack to the other. The following recursive subroutines define a suitable sorting procedure:

- `SORT00` [Sort the top subfile on tape 0 and return it to tape 0].

If the subfile fits in the internal memory, sort it internally and return it to tape. Otherwise select one record R from the subfile, and let its key be K. Reading backward on tape 0, copy all records whose key is $> K$, forming a new subfile on the top of tape 1. Now read forward on tape 0, copying all records whose key is $= K$ onto tape 1. Then read backward again, copying all records whose key is $< K$ onto tape 1. Complete the sort by executing `SORT10` on the $< K$ keys, then copying the $= K$ keys to tape 0, and finally executing `SORT10` on the $> K$ keys.

- `SORT01` [Sort the top subfile on tape 0 and write it on tape 1].

Same as `SORT00`, but the final "`SORT10`" is changed to "`SORT11`" followed by copying the $\leq K$ keys to tape 1.

- `SORT10` [Sort the top subfile on tape 1 and write it on tape 0].

Same as `SORT10`, interchanging 0 with 1 and $<$ with $>$.

- `SORT11` [Sort the top subfile on tape 1 and return it to tape 1].

Same as `SORT00`, interchanging 0 with 1 and $<$ with $>$.

The recursive nature of these subroutines can be handled without difficulty by storing appropriate control information on the tapes.

The running time for this algorithm can be estimated as follows, if we assume that the data are in random order, with negligible probability of equal keys. Let M be the number of records that fit into internal memory. Let X_N be the average number of records read while applying `SORT00` or `SORT11` to a subfile of N records, when $N > M$, and let Y_N be the corresponding quantity for `SORT01` or `SORT10`. Then we have

$$X_N = \begin{cases} 0, & \text{if } N \leq M; \\ 3N + 1 + \dfrac{1}{N} \displaystyle\sum_{0 \leq k < N} (Y_k + Y_{N-1-k}), & \text{if } N > M; \end{cases}$$

$$Y_N = \begin{cases} 0, & \text{if } N \leq M; \\ 3N + 2 + \dfrac{1}{N} \displaystyle\sum_{0 \leq k < N} (Y_k + X_{N-1-k} + k), & \text{if } N > M. \end{cases} \qquad (3)$$

The solution to these recurrences (see exercise 2) shows that the total amount of tape reading during the external partitioning phases will be $6\frac{2}{3}N \ln N + O(N)$, on the average, as $N \to \infty$. We also know from Eq. 5.2.2–25 that the average number of internal sort phases will be $2(N + 1)/(M + 2) - 1$.

If we apply this analysis to the 100,000-record example of Section 5.4.6, using 25,000-character buffers and assuming that the sorting time is $2nC\omega\tau$ for a subfile of $n \le M = 1000$ records, we obtain an average sorting time of approximately 103 minutes (including the final rewind as in Chart A). Thus the quicksort method isn't bad, on the average; but of course its *worst* case turns out to be even more awful than the bubble sort discussed above.

Radix sorting. The radix exchange method (Algorithm 5.2.2R) can be adapted to two-tape sorting in a similar way, since it is so much like quicksort. The "trick" which makes both of these methods work is the idea of reading a file more than once, something we never did in our previous tape algorithms.

The same trick can be used to do a conventional "least-significant-digit-first" radix sort on two tapes. Given the input data on T1, we copy all records onto T2 whose key ends with 0 in binary notation; then after rewinding T1 we read it again, copying the records whose key ends with 1. Now both tapes are rewound and a similar pair of passes is made, interchanging the roles of T1 and T2, and using the *second* least significant binary digit. At this point T1 will contain all records whose keys are $(\ldots 00)_2$, followed by those whose keys are $(\ldots 01)_2$, then $(\ldots 10)_2$, then $(\ldots 11)_2$. If the keys are b bits long, we need only $2b$ passes over the file in order to complete the sort.

Such a radix sort could be applied only to the *leading* b bits of the keys, for some judiciously chosen number b; this would reduce the number of inversions by a factor of about 2^b, if the keys were uniformly distributed, so a few passes of the P-way bubble sort could be used to complete the job.

A novel but somewhat more complicated approach to two-tape distribution sorting has been suggested by A. I. Nikitin and L. I. Sholmov [*Kibernetika* **2**, 6 (1966), 79–84]. Counts are made of the number of keys having each possible configuration of leading bits, and artificial keys $\kappa_1, \kappa_2, \ldots, \kappa_M$ based on these counts are constructed so that the number of actual keys lying between κ_i and κ_{i+1} is between predetermined limits P_1 and P_2, for each i. Thus, M lies between $\lceil N/P_2 \rceil$ and $\lceil N/P_1 \rceil$. If the leading bit counts do not give sufficient information to determine such $\kappa_1, \kappa_2, \ldots, \kappa_M$, one or more further passes are made to count the frequency of less significant bit patterns, for certain configurations of most significant bits. After the table of artificial keys $\kappa_1, \kappa_2, \ldots, \kappa_M$ have been constructed, $2\lceil \log_2 M \rceil$ passes will suffice to complete the sort. (This method requires memory space proportional to N, so it can't be used for external sorting as $N \to \infty$. In practice we would not use the technique for multireel files, so M will be comparatively small and the table of artificial keys will fit comfortably in memory.)

Simulation of more tapes. F. C. Hennie and R. E. Stearns have devised a general technique for simulating k tapes on only two tapes, in such a way that

	Zone 0	Zone 1		Zone 2				Zone 3					
Track 1	1	5	9	13	17	21	25	29	33	37	41	45	49
Track 2	2	6	10	14	18	22	26	30	34	38	42	46	
Track 3	3	7	11	15	19	23	27	31	35	39	43	47	
Track 4	4	8	12	16	20	24	28	32	36	40	44	48	

Fig. 86. Layout of tape T1 in the Hennie-Stearns construction; nonblank zones are shaded.

the tape motion required is increased by a factor of only $O(\log L)$, where L is the maximum distance to be traveled on any one tape [*JACM* **13** (1966), 533–546]. Their construction can be simplified slightly in the case of sorting, as in the following method suggested by R. M. Karp.

We shall simulate an ordinary four-tape balanced merge, using two tapes T1 and T2. The first of these, T1, holds the simulated tape contents in a way that may be diagrammed as in Fig. 86; we imagine that the data is written in four "tracks," one for each simulated tape. (In actual fact the tape doesn't have such tracks; blocks 1, 5, 9, 13, . . . are thought of as track 1, and blocks 2, 6, 10, 14, . . . as track 2, etc.) The other tape, T2, is used only for auxiliary storage, to help move things around on T1.

The blocks of each track are divided into *zones*, containing, respectively, $1, 2, 4, 8, \ldots, 2^k, \ldots$ blocks per zone. Zone k on each track is either filled with exactly 2^k blocks of data, or it is completely "blank." In Fig. 86, for example, track 1 has data in zones 1 and 3; track 2 in zones 0, 1, 2; track 3 in zones 0 and 2; track 4 in zone 1; and the other zones are blank.

Suppose that we are merging data from Tracks 1 and 2 to Track 3. The internal computer memory contains two buffers used for input to a two-way merge, plus a third buffer for output. When the input buffer for Track 1 becomes empty, we can refill it as follows: Find the first nonempty zone on Track 1, say zone k, and copy its first block into the input buffer; then copy the other $2^k - 1$ blocks of data onto T2, and move them to zones $0, 1, \ldots, k-1$ of track 1. (Zones $0, 1, \ldots, k-1$ are now full and zone k is blank.) An analogous procedure is used to refill the input buffer for Track 2, whenever it becomes empty. When the output buffer is ready to be written on Track 3, we reverse the process, scanning across T1 to find the first *blank* zone on T3, say zone k, while copying the data from zones $0, 1, \ldots, k-1$ onto T2. The data on T2, augmented by the contents of the output buffer, is now used to fill zone k of Track 3.

This procedure requires the ability to write in the middle of tape T1, without destroying subsequent information on that tape. As in the case of read-forward oscillating sort (Section 5.4.5), it is possible to do this reliably if suitable precautions are taken.

The amount of tape motion required to bring $2^l - 1$ blocks of Track 1 into memory is $\sum_{0 \le k < l} 2^{l-1-k} \cdot c \cdot 2^k = cl2^{l-1}$, for some constant c, since we scan up to zone k only once in every 2^k steps. Thus each merge pass requires

$O(N \log N)$ steps. Since there are $O(\log N)$ passes in a balanced merge, the the constant in $O(N(\log N)^2)$ is much too high to be satisfactory when N is asymptotically much better than the worst case of quicksort.

But this method wouldn't actually work very well if we applied it to the 100,000-record example of Section 5.4.6, since the information specified for tape T1 would overflow the contents of one tape reel. Even if we ignore this fact, and if we use optimistic assumptions about read/write/compute overlap and interblock gap lengths, etc., we find that roughly 37 hours would be required to complete the sort! So this method is purely of academic interest; the constant in $O(N(\log N)^2)$ is much too high to be satisfactory when N is in a practical range.

One-tape sorting. Could we live with only one tape? It is not difficult to see that the order-P bubble sort described above could be converted into a one-tape sort, but the result would be ghastly.

H. B. Demuth [Ph. D. thesis (Stanford University, 1956), 85] observed that a computer with bounded internal memory cannot reduce the number of inversions of a permutation by more than a bounded amount as it moves a bounded distance on tape, hence every one-tape sorting algorithm must take at least $d\,N^2$ units of time on the average (for some positive constant d which depends on the computer configuration).

R. M. Karp has pursued this topic in a very interesting way, discovering what amounts to an *optimum* way to sort with one tape. It is convenient to discuss Karp's algorithm by reformulating the problem as follows: *What is the fastest way to transport people between floors using a single elevator?*

Consider a building with n floors, having room for exactly c people on each floor. The building contains no doors, windows, or stairs, but it does have an elevator that can stop on each floor. There are cn people in the building, and exactly c of them want to go to each particular floor. The elevator holds at most b people, and it takes one unit of time to go from floor i to floor $i+1$. We wish to find the quickest way to get all the people onto the proper floors, if the elevator is required to start and finish on floor 1.

The connection between this elevator problem and one-tape sorting is not hard to see: The people are the records and the building is the tape. The floors are individual blocks on the tape, and the elevator is the internal computer memory. A computer program has more flexibility than an elevator operator (it can, for example, duplicate people, or temporarily chop them into two parts on different floors, etc.); but the solution below solves the problem in the fastest conceivable time without doing such operations.

The following two auxiliary tables are required by Karp's algorithm.

$$u_k,\ 1 \le k \le n\text{: Number of people on floors} \le k \text{ whose destination is} > k;$$
$$d_k,\ 1 \le k \le n\text{: Number of people on floors} \ge k \text{ whose destination is} < k. \quad (4)$$

When the elevator is empty, we always have $u_k = d_{k+1}$ for $1 \le k < n$, since there are c people on every floor; the number of misfits on floors $\{1, \ldots, k\}$

must equal the corresponding number on floors $\{k+1, \ldots, n\}$. By definition, $u_n = d_1 = 0$.

It is clear that the elevator must make at least $\lceil u_k/b \rceil$ trips from floor k to floor $k+1$, for $1 \le k < n$, since only b passengers can ascend on each trip. Similarly it must make at least $\lceil d_k/b \rceil$ trips from floor k to floor $k-1$. Therefore the elevator must necessarily take at least

$$\sum_{1 \le k \le n} (\lceil u_k/b \rceil + \lceil d_k/b \rceil) \tag{5}$$

units of time on any correct schedule. Karp discovered that this lower bound can actually be achieved, when $u_1, \ldots, u_{n-1}$ are nonzero.

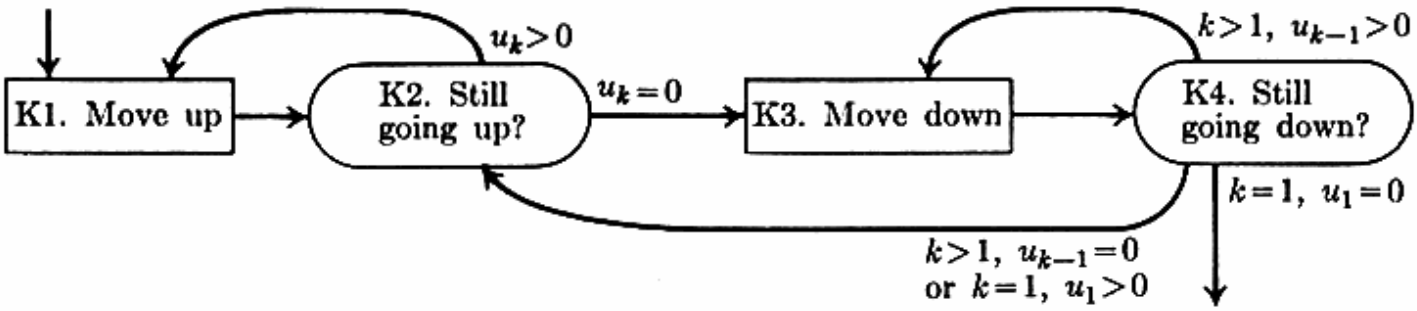


Fig. 87. Karp's elevator algorithm.

Theorem K. *If $u_k > 0$ for $1 \le k < n$, there is an elevator schedule which delivers everyone to his destination in the minimum time* (5).

Proof. Assume that there are b extra people in the building; they start in the elevator and their destination floor is artifically set to 0. The elevator can operate according to the following algorithm, starting with k (the current floor) equal to 1:

K1. [Move up.] From among the $b + c$ people currently in the elevator or on floor k, those b with the highest destinations get into the elevator, and the others remain on floor k.

Let there be u people now in the elevator whose destination is $> k$, and d whose destination is $\le k$. (It will turn out that $u = \min(b, u_k)$; if $u_k < b$ we may therefore be transporting some people away from their destination. This represents their sacrifice to the common good.) Decrease u_k by u, increase d_{k+1} by d, and then increase k by 1.

K2. [Still going up?] If $u_k > 0$, return to step K1.

K3. [Move down.] From among the $b + c$ people currently in the elevator or on floor k, those b with the lowest destinations get into the elevator, and the others remain on floor k.

Let there be u people now in the elevator whose destination is $\ge k$, and d whose destination is $< k$. (It will always turn out that $u = 0$ and $d = b$, but the algorithm is described here in terms of general u and d in order to make the proof a little clearer.) Decrease d_k by d, increase u_{k-1} by u, and then decrease k by 1.

K4. [Still going down?] If $k > 1$ and $u_{k-1} > 0$, return to step K3. If $k = 1$ and $u_1 = 0$, terminate the algorithm (everyone has moved to his destination and the b "extras" are back in the elevator). Otherwise return to step K2.

Figure 88 shows an example of this algorithm, with a nine-floor building and $b = 3$, $c = 2$. Note that one of the 6's is temporarily transported away from his destination in spite of the fact that the elevator travels the minimum possible distance. The idea of testing u_{k-1} in step K4 is the crucial thing which makes the algorithm work, as we shall see.

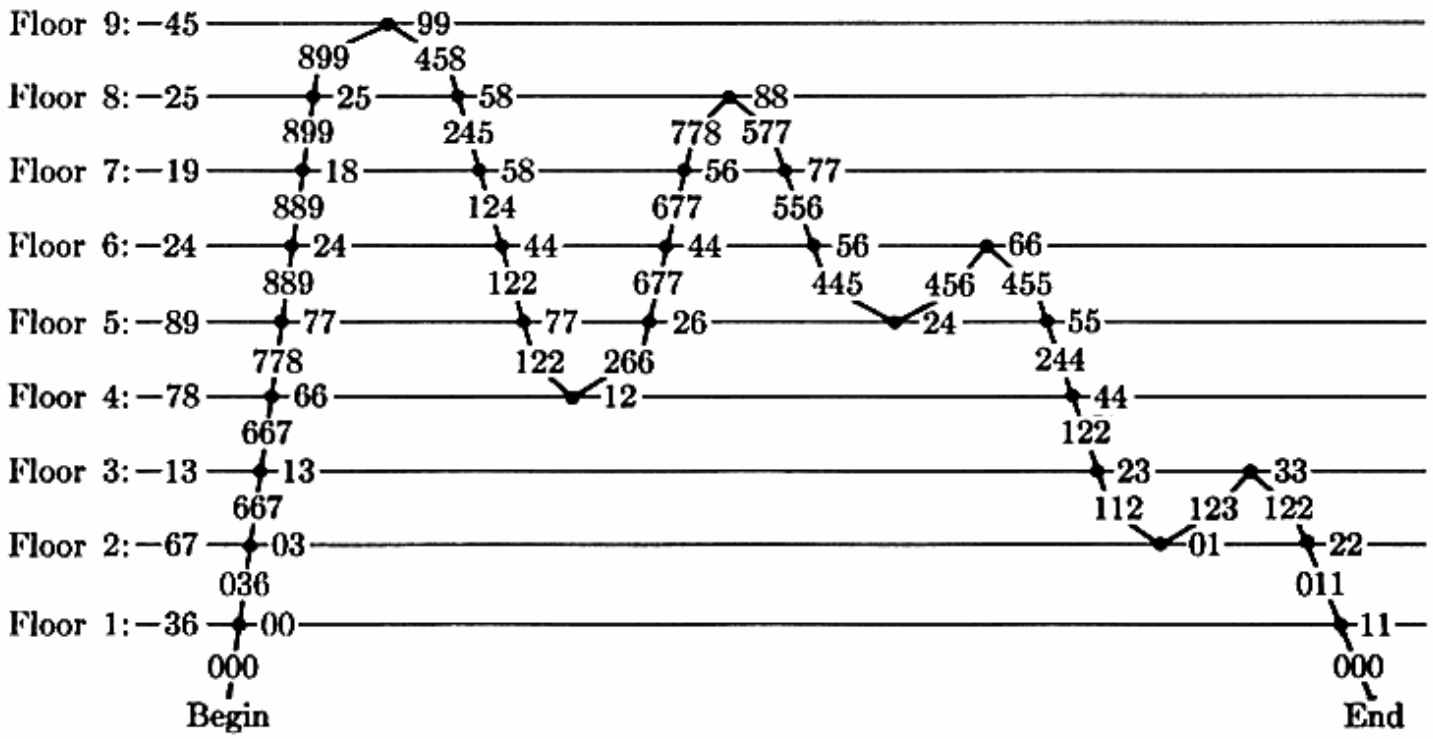


Fig. 88. An optimum way to rearrange people using a small, slow elevator. (Each person is represented by the number of his destination floor.)

To verify the validity of this algorithm, we note that steps K1 and K3 always keep the u and d tables (4) up to date, if we regard the people in the elevator as being on the "current" floor k. It is now possible to prove by induction that the following properties hold at the beginning of each step:

$$u_l = d_{l+1} \qquad \text{for} \qquad k \le l < n; \tag{6}$$

$$u_l = d_{l+1} - b, \qquad \text{for} \qquad 1 \le l < k; \tag{7}$$

$$\text{if} \qquad u_l = 0 \qquad \text{and} \qquad k \le l < n \qquad \text{then} \qquad u_{l+1} = 0. \tag{8}$$

Furthermore, at the beginning of step K1, the min (u_k, b) people with highest destinations, among all people on floors $\le k$ with destination $> k$, are in the elevator or on floor k. At the beginning of step K3, the min (d_k, b) people with lowest destinations, among all people on floors $\ge k$ with destination $< k$, are in the elevator or on floor k. These conditions may also be verified inductively, by considering how we can reach step K1 or K3 (cf. exercise 5).

From these properties it follows that the parenthesized remarks in steps K1 and K3 are valid. Each execution of step K1 therefore decreases $\lceil u_k/b \rceil$ by 1 and leaves $\lceil d_{k+1}/b \rceil$ unchanged; each execution of K3 decreases $\lceil d_k/b \rceil$ by

1 and leaves $\lceil u_{k-1}/b \rceil$ unchanged. The algorithm must therefore terminate in a finite number of steps, and everybody must then be at his destination because of (6) and (8). ∎

When $u_k = 0$ and $u_{k+1} > 0$ we have a "disconnected" situation; the elevator must journey up to floor $k+1$ in order to rearrange the people up there, even though nobody wants to move from floors $\leq k$ to floors $\geq k+1$. Without loss of generality, we may assume that $u_{n-1} > 0$; then every valid elevator schedule must include at least

$$2 \sum_{1 \leq k < n} \max(1, \lceil u_k/b \rceil) \tag{9}$$

moves, since we require the elevator to return to floor 1. A schedule achieving this lower bound is readily constructed (exercise 4).

EXERCISES

1. [*17*] The order-P bubble sort discussed in the text uses only forward reading and rewinding. Can the algorithm be modified to take advantage of *backward* reading?

2. [*M26*] Find explicit "closed form" solutions for the numbers X_N, Y_N defined in (3). [*Hint:* Study the solution to Eq. 5.2.2–19.]

3. [*38*] Is there a two-tape sorting method, based only on comparisons of keys (not digital properties), whose tape motion is $O(N \log N)$ in the worst case, when sorting N records? [Quicksort achieves this on the average, but not in the worst case, and the Hennie-Stearns method (Fig. 86) achieves $O(N(\log(N)^2)$.]

4. [*M23*] In the elevator problem, suppose there are indices p, q with $q \geq p + 2$, $u_p > 0$, $u_q > 0$, and $u_{p+1} = \cdots = u_{q-1} = 0$. Explain how to construct a schedule requiring at most (9) units of time.

▶ 5. [*M23*] True or false: After step K1 of the algorithm in Theorem K, nobody on the elevator has a lower destination than any person on floors $< k$.

6. [*M30*] (R. M. Karp.) Generalize the elevator problem (Fig. 87) to the case that there are c_j passengers initially on floor j, and c'_j passengers whose destination is floor j, for $1 \leq j \leq n$. Show that a schedule exists which takes $2\sum_{1\leq k<n} \max(1, \lceil u_k/b \rceil, \lceil d_{k+1}/b \rceil)$ units of time, never allowing more than $\max(c_j, c'_j)$ passengers to be on floor j at any one time. [*Hint:* Introduce fictitious people, if necessary, to make $c_j = c'_j$ for all j.]

7. [*M40*] (R. M. Karp.) Generalize the problem of exercise 6, replacing the linear path followed by the elevator by a network of roads to be traveled by a bus, given that the network forms any *free tree*. The bus has finite capacity, and it is desired to transport passengers to their destinations in such a way that the bus travels a minimum distance.

8. [*M32*] Let $c = 1$ in the elevator problem treated in the text. How many permutations of the n people on the n floors will make $u_k \leq 1$ for $1 \leq k \leq n$ in (4)? [For example, 3 1 4 5 9 2 6 8 7 is such a permutation.]

▶ 9. [*M25*] Find a significant connection between the "cocktail shaker sort" described

in Section 5.2.2, Fig. 16, and the numbers $u_1, u_2, \ldots, u_n$ of (4) in the case $c = 1$.

10. [*20*] How would you sort a multireel file with only two tapes?

5.4.9. Disks and Drums

So far we have considered tapes as the vehicles for external sorting, but more flexible types of mass storage devices are often available. Although such "bulk memory" or "direct-access storage" units come in many different forms, they may be roughly characterized by the following properties:

i) It doesn't take very long to access any specified part of the stored information;

ii) Blocks of consecutive words can be transmitted rapidly between the internal and external memory.

Magnetic tape satisfies (ii) but not (i), because it takes a long time to get from one end of a tape to the other. Some devices satisfy (i) but not (ii), e.g. bulk core memory units in which the access time to each word is about ten times slower than the internal memory speed.

Every external memory unit has idiosyncrasies which ought to be studied carefully before major programs are written for it; but technology changes so rapidly, it is impossible to give a complete discussion here of all the available varieties of hardware. Therefore we shall consider only some typical memory devices which illustrate useful approaches to the sorting problem.

One of the most common types of external memories satisfying (i) and (ii) is a disk file or disk pack module (see Fig. 89). Data is kept on a number of rapidly-rotating circular disks, covered with magnetic material; a comb-like access arm, containing one or more "read/write heads" for each disk surface, is used to store and retrieve the information. Each individual surface is divided into concentric rings called *tracks*, so that an entire track of data passes a read/write head every time the disk completes one revolution. The access arm can move in and out, shifting the read/write heads from track to track; but this motion takes time. A set of tracks which can be read or written without repositioning the access arm is called a *cylinder* or a *stratum*. For example, Fig. 89

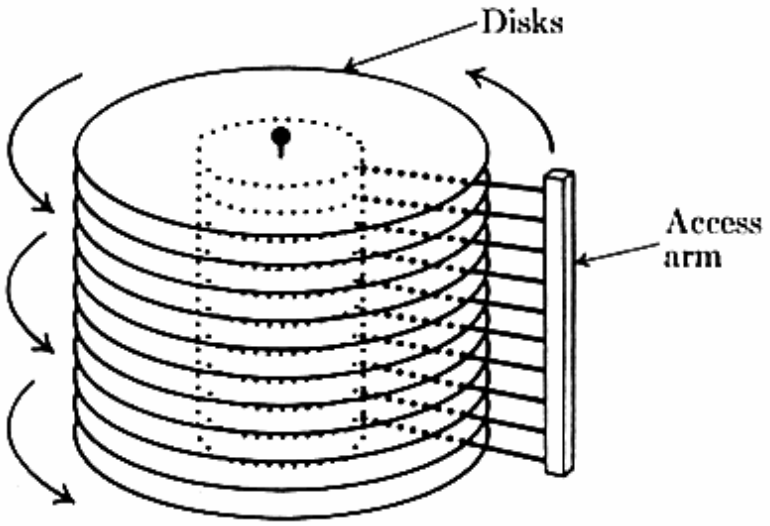


Fig. 89. A disk pack module.

illustrates a disk file which has just one read/write head per surface; the dotted lines show one of the cylinders, consisting of all tracks currently being scanned by the read/write heads.

To fix the ideas, let us consider hypothetical MIXTEC disk pack units, for which

$$\begin{aligned} 1 \text{ track} &= 5000 \text{ characters} \\ 1 \text{ cylinder} &= 20 \text{ tracks} \\ 1 \text{ disk unit} &= 200 \text{ cylinders} \end{aligned}$$

Such a disk unit contains 20 million characters, slightly less than the amount of data that can be stored on a single magnetic tape. On some machines, tracks near the center have fewer characters than tracks near the rim; this tends to make the programming much more complicated, and MIXTEC fortunately avoids such problems.

The amount of time required to read or write on a disk file is essentially the sum of three quantities,

- Seek time (the time to move the access arm to the proper cylinder);
- Latency time (rotational delay until the read/write head reaches the right spot);
- Transmission time (rotational delay while the data passes the read/write head).

On MIXTEC devices the seek time required to go from cylinder i to cylinder j is $25 + \frac{1}{2}|i - j|$ milliseconds. If i and j are randomly-selected integers between 1 and 200, the average value of $|i - j|$ is $2\binom{201}{3}/200^2 \approx 66.7$, so the average seek time is about 60 ms. MIXTEC disks rotate once every 25 ms, so the latency time averages about 12.5 ms. The transmission time for n characters is $(n/5000) \times 25 \text{ ms} = 5n\mu\text{s}$. (This is about $3\frac{1}{3}$ times as fast as the transmission rate of MIXT tapes used in the examples of Section 5.4.6.)

Thus the main differences between MIXTEC disks and MIXT tapes, relative to sorting, are

a) Tapes can only be accessed sequentially;

b) Individual disk operations tend to require significantly more overhead (seek time + latency time compared to stop/start time);

c) The disk transmission rate is faster.

By using clever merge patterns on tape, we were able to compensate somewhat for disadvantage (a). Our goal now is to think of some clever algorithms for disk sorting that will compensate for disadvantage (b).

Overcoming latency time. Let us consider first the problem of minimizing the delays caused by the fact that the disks aren't always positioned properly when we want to start an I/O command. We can't make the disk spin faster, but we can still apply some tricks that reduce or even eliminate all of the latency time.

The addition of more access arms would obviously help, but that would be an expensive hardware modification. Here are some "software" ideas:

1) If we read or write several tracks of a cylinder at a time, we avoid the latency time (*and* the seek time) on all tracks but the first. In general it is often possible to synchronize the computing time with the disk movement in such a way that a sequence of input/output instructions can be carried out without latency delays.

2) Consider the problem of reading half a track of data (Fig. 90): If the read command begins when the heads are at point A, there is no latency delay, and the total time for reading is just the transmission time, $\frac{1}{2} \times 25$ ms. If the command begins with the heads at B, we need $\frac{1}{4}$ of a revolution for latency and $\frac{1}{2}$ for transmission, totalling $\frac{3}{4} \times 25$ ms. The most interesting case occurs when the heads are initially at C: With proper hardware and software we need *not* waste $\frac{3}{4}$ of a revolution for latency delay. Reading can begin immediately, into the second half of the input buffer; then after a $\frac{1}{2} \times 25$ ms pause, reading can resume into the first half of the buffer, so that the instruction is completed when point C is reached again. In a similar manner, we can ensure that the total latency + transmission time will never exceed the time for one revolution, regardless of the initial position of the disk. The average amount of latency delay is reduced by this scheme from half a revolution to $\frac{1}{2}(1 - x^2)$ of a revolution, if we are reading or writing a given fraction x of a track, for $0 < x \leq 1$. When an entire track is being read or written ($x = 1$), this technique eliminates *all* the latency time.

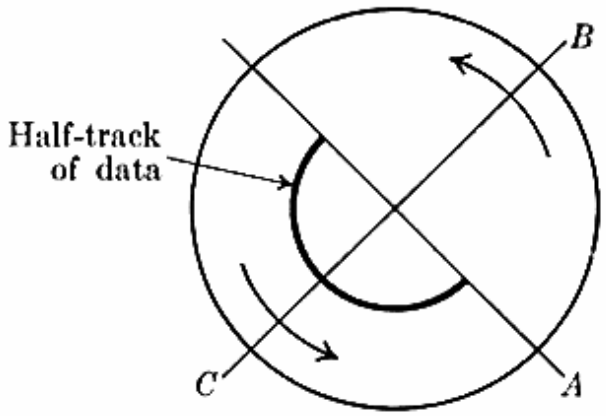


Fig. 90. Analysis of the latency time when reading half of a track.

Drums: The no-seek case. Some external memory units eliminate the seek time by having one read/write head for every track. If the technique of Fig. 90 is employed on such devices, both seek time and latency time reduce to zero, provided that we always read or write a track at a time; this is the ideal situation in which transmission time is the only limiting factor.

Let us consider again the example application of Section 5.4.6, sorting 100,000 records of 100 characters each, with a 100,000-character internal memory. The total amount of data to be sorted fills half of a MIXTEC disk. It is usually impossible to read and write simultaneously on a single disk unit; we shall assume that two disks are available, so that reading and writing can overlap

each other. For the moment we shall assume, in fact, that the "disks" are actually *drums*, containing 4000 tracks of 5000 characters each, with no seek time required.

What sorting algorithm should be used? The method of merging is a fairly natural choice; other methods of internal sorting do not lend themselves so well to a disk implementation, except possibly radix techniques, but the considerations of Section 5.4.7 show that radix sorting is usually inferior to merging for general applications. (It is not hard to see that the duality theorem of that section applies to disks as well as to tapes.)

To begin a merge sort for this problem we can use replacement selection, with two 5000-character input buffers and two 5000-character output buffers. In fact, it is possible to reduce this to *three* 5000-character buffers, if records in the current input buffer are replaced by records that come off the selection tree. That leaves 85,000 characters (850 records) for a selection tree, so one pass over our example data will form about 60 initial runs. (See Eq. 5.4.6–3.) This pass takes only about 50 seconds, if we assume that the internal processing time is fast enough to keep up with the input/output rate (one record moving to the output buffer every 500 microseconds). If the input to be sorted appeared on a MIXT tape, instead of on a drum, this pass would be slower, governed by the tape speed.

With two drums and full-track reading/writing, it is not hard to see that the total transmission time for P-way merging is minimized if we let P be as large as possible. Unfortunately we can't simply do a 60-way merge on all of the initial runs, since there isn't room for 60 buffers in memory. (A buffer of less than 5000 characters would introduce unwanted latency time.) If we do P-way merges, passing all the data from one drum to the other so that reading and writing are overlapped, the number of merge passes is $\lceil \log_P 60 \rceil$, so we may complete the job in two passes if $8 \le P \le 59$. The smallest such P reduces the amount of internal computing, so we choose $P = 8$; if 65 initial runs had been formed we would take $P = 9$. If 82 or more initial runs had been formed, we could take $P = 10$, but since there is room for only 18 input buffers and 2 output buffers there would be a possibility of hangup during the merge (see Algorithm 5.4.6F); it may be better in such a case to do two partial passes over a small portion of the data, reducing the number of initial runs to 81 or less.

Under our assumptions, both of the merging passes will take about 50 seconds, so the entire sort in this ideal situation will be completed in just $2\frac{1}{2}$ minutes (plus a few seconds for bookkeeping, initialization, etc.). This is six times faster than the best six-tape sort considered in Section 5.4.6; the reasons for this speedup are the improved external/internal transmission rate ($3\frac{1}{2}$ times faster), the higher order of merge (we can't do an eight-way tape merge unless we have nine or more tapes), and the fact that the output was left on disk (no final rewind, etc., was necessary). If the initial input and sorted output were required to be on MIXT tapes, with the drums used for merging only, the corresponding sorting time would have been about 8.2 minutes.

If only one drum were available instead of two, the input-output time

would take twice as long, since reading and writing must be done separately. (In fact, the input-output operations would take *three times* as long, since we would be overwriting the initial input data; in such a case it is prudent to follow each write by a "read-back check" operation, lest some of the input data be irretrievably lost, if the hardware does not provide automatic verification of written information.) But some of this excess time can be recovered because we can use partial pass methods which process some data records more often than others. The two-drum case requires all data to be processed an even number or an odd number of times, but the one-drum case can use more general merge patterns.

We observed in Section 5.4.4 that merge patterns can be represented by trees, and that the transmission time corresponding to a merge pattern is proportional to the external path length of its tree. Only certain trees ("T-lifo" or "strongly T-fifo") could be used as efficient tape merging patterns, because some runs get buried in the middle of a tape as the merging proceeds. But *on disks or drums, all trees define usable merge patterns* as long as the degrees of their internal nodes are not too large (i.e., compatible with the available internal memory size).

Therefore we can minimize transmission time by choosing a tree with minimum external path length, such as a complete P-ary tree where P is as large as possible. By Eq. 5.4.4–9, the external path length of such a tree is equal to

$$qS - \lfloor (P^q - S)/(P - 1) \rfloor, \qquad q = \lceil \log_P S \rceil, \tag{1}$$

if there are S external nodes (leaves).

It is particularly easy to design an algorithm which merges according to the complete P-ary tree pattern. See, for example, Fig. 91, which shows the case $P = 3$, $S = 6$. First we add "dummy runs," if necessary, to make $S \equiv 1$ (modulo $P - 1$); then we combine runs according to a "first-in-first-out" discipline, at every stage merging the P "oldest" runs at the front of the queue into a single run which is placed at the rear.

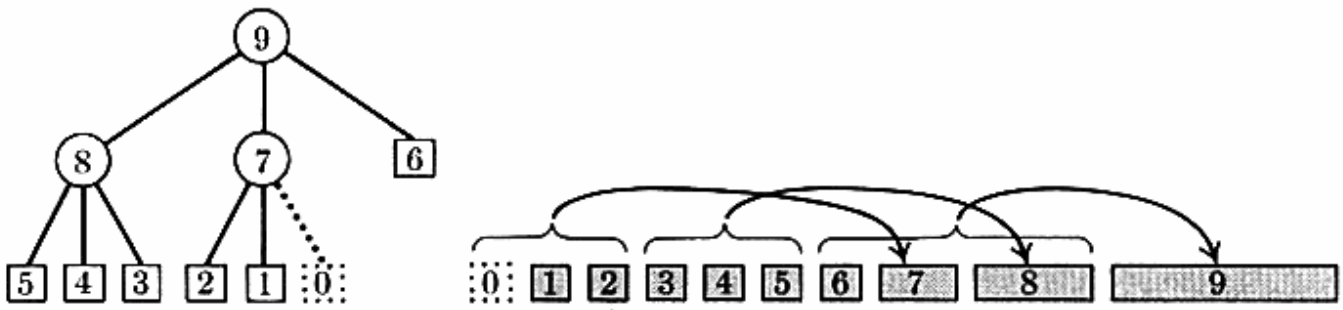


Fig. 91. Complete ternary tree with six leaves, and the corresponding merge pattern.

The complete P-ary tree gives an optimum pattern if all of the initial runs are the same length, but we can often do better if some runs are longer than others. An optimum pattern for this general situation can be constructed without difficulty by using Huffman's method (exercise 2.3.4.5–10), which may be stated in merging language as follows: "First add $(1 - S) \bmod (P - 1)$

dummy runs of length 0. Then repeatedly merge together the P *shortest* existing runs until only one run is left." When all initial runs have the same length this method reduces to the fifo discipline described above.

In our 100,000-record example we can do nine-way merging, since 18 input buffers and two output buffers will fit in memory and Algorithm 5.4.6F will overlap all compute time. The complete 9-ary tree with 60 leaves corresponds to a merging pattern with $1\frac{29}{30}$ passes, if all initial runs have the same length. The total sorting time with one drum, using "read-back check" after every write, therefore comes to about 7.4 minutes. A higher value of P may reduce this running time slightly; but the situation is very unclear because "reading hangup" may occur, since the buffers might become too full or too empty.

The influence of seek time. The above discussion shows that it is relatively easy to construct "optimum" merging patterns for drums, because seek time and latency time can be essentially nonexistent. But when disks are used it often takes longer to seek out information than to read it, so the seek time has a considerable influence on the sorting strategy. Decreasing the order of merge, P, makes it possible to use larger buffers, so fewer seeks are required; this often compensates for the extra transmission time demanded by the smaller value of P.

Seek time depends on the distance traveled by the access arm, and we could try to arrange things so that this distance is minimized. For example, it may be wise to sort the records within cylinders first. However, large-scale merging requires a good deal of jumping around between cylinders (cf. exercise 2). Furthermore, the multiprogramming capability of modern operating systems means that a user rarely has much actual control over the position of disk access arms; fancy seek-minimization schemes usually work only on weekends! We are often justified, therefore, in assuming that each disk command involves a "random" seek.

Our goal is to discover a tree (i.e., a merge pattern) which achieves the best balance between seek time and transmission time; for this purpose we need some way to estimate the goodness of any particular tree with respect to a particular hardware configuration. Consider, for example, the tree in Fig. 92; we want to estimate how long it will take to carry out the corresponding merge, so that we can compare this tree to other trees.

In the following discussion we shall make some simple assumptions about disk merging, in order to illustrate some of the general ideas. Let us suppose that (1) it takes $72.5 + 0.005n$ milliseconds to read or write n characters; (2) 100,000 characters of internal memory are available for working storage; (3) an average of 0.004 milliseconds of computation time are required to transmit each character from input to output; (4) there is to be *no overlap* between reading, writing, or computing; and (5) the buffer size used on output need not be the same as the buffer size used to read the data on the following pass. An analysis of the sorting problem under these simple assumptions will give us some insights when we turn to more complicated situations.

If we do a P-way merge, we can divide the internal working storage into $P+1$ buffer areas, P for input and one for output, with $B = 100000/(P+1)$ characters per buffer. Suppose the files being merged contain a total of L characters; then we will do approximately L/B output operations and about the same number of input operations, so the total merging time under our assumptions will be approximately

$$2\left(72.5\,\frac{L}{B} + 0.005L\right) + 0.004L = (0.00145P + 0.01545)L \qquad (2)$$

milliseconds.

In other words, a P-way merge of L characters takes about $(\alpha P + \beta)L$ units of time, for some constants α and β depending on the seek time, latency time, compute time, and memory size. This formula leads to an interesting way to construct good merge patterns for disks. Consider Fig. 92, for example, and assume that all initial runs (represented by square "leaf" nodes) have length L_0. Then the merges at nodes 9 and 10 each take $(2\alpha + \beta)(2L_0)$ units of time, the merge at node 11 takes $(3\alpha + \beta)(4L_0)$, and the final merge at node 12 takes $(4\alpha + \beta)(8L_0)$. The total merging time therefore comes to $(52\alpha + 16\beta)L_0$ units. The coefficient "16" here is well-known to us, it is simply the external path length of the tree. The coefficient "52" of α is, however, a new concept, which we may call the *degree path length* of the tree; it is the sum, taken over all leaf nodes, of the internal-node degrees on the path from the leaf to the root. For example, in Fig. 92 the degree path length is $(2+4) + (2+4) + (3+4) + (2+3+4) + (2+3+4) + (3+4) + (4) + (4) = 52$.

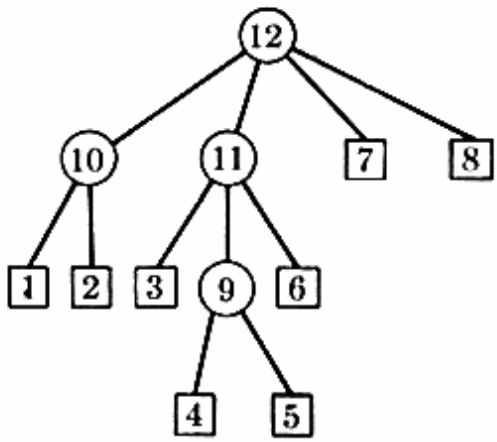


Fig. 92. A tree whose external path length is 16 and whose degree path length is 52.

If $\mathfrak{T}$ is any tree, let $D(\mathfrak{T})$, $E(\mathfrak{T})$ denote its degree path length and its external path length, respectively. Our analysis may be summarized as follows:

Theorem H. *If the time required to do a P-way merge on L characters has the form $(\alpha P + \beta)L$, and if there are S equal-length runs to be merged, the best merge pattern corresponds to a tree $\mathfrak{T}$ for which $\alpha D(\mathfrak{T}) + \beta E(\mathfrak{T})$ is a minimum, over all trees having S leaves.* ∎

(This theorem was implicitly contained in an unpublished paper which George U. Hubbard presented at the ACM National Conference in 1963.)

Let α and β be fixed constants; we shall say a tree is *optimal* if it has the minimum value of $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$ over all trees, $\mathfrak{I}$, with the same number of leaves. It is not difficult to see that *all subtrees of an optimal tree are optimal*, and therefore we can construct optimal trees with n leaves by piecing together optimal trees with $< n$ leaves.

Theorem K. *Let the sequence of numbers $A_m(n)$ be defined for $1 \le m \le n$ by the rules*

$$A_1(1) = 0; \tag{3}$$

$$A_m(n) = \min_{1 \le k \le n/m} \left(A_1(k) + A_{m-1}(n-k)\right), \qquad \text{for} \qquad 2 \le m \le n; \tag{4}$$

$$A_1(n) = \min_{2 \le m \le n} \left(\alpha m n + \beta n + A_m(n)\right), \qquad \text{for} \qquad n \ge 2. \tag{5}$$

Then $A_1(n)$ is the minimum value of $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$, over all trees $\mathfrak{I}$ with n leaves.

Proof. Equation (4) implies that $A_m(n)$ is the minimum value of $A_1(n_1) + \cdots + A_1(n_m)$ taken over all positive integers $n_1, \ldots, n_m$ such that $n_1 + \cdots + n_m = n$. The result now follows by induction on n. ∎

The recurrence relations (3), (4), (5) can also be used to construct the optimal trees themselves: Let $k_m(n)$ be a value for which the minimum occurs in the definition of $A_m(n)$. Then we can construct an optimal tree with n leaves by joining $m = k_1(n)$ subtrees at the root; the subtrees are optimal trees with $k_m(n)$, $k_{m-1}(n - k_m(n))$, $k_{m-2}(n - k_m(n) - k_{m-1}(n - k_m(n)))$, . . . leaves, respectively.

For example, Table 1 illustrates this construction when $\alpha = \beta = 1$. A compact specification of the corresponding optimal trees appears at the right of the table; the entry "4:9:9" when $n = 22$ means, for example, that an optimal tree $\mathfrak{I}_{22}$ with 22 leaves may be obtained by combining $\mathfrak{I}_4$, $\mathfrak{I}_9$, and $\mathfrak{I}_9$ (see Fig. 93). Optimal trees are not unique; for instance, 5:8:9 would be just as good as 4:9:9.

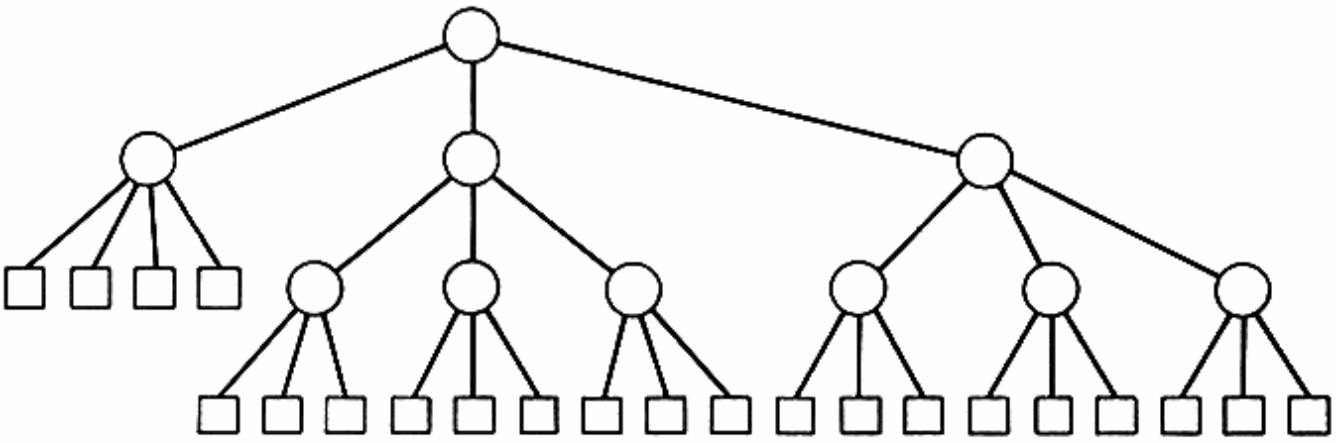


Fig. 93. An optimum way to merge 22 initial runs of equal length, when $\alpha = \beta$ in Theorem H. This pattern minimizes the seek time, under the assumptions leading to Eq. (2) in the text.

Table 1

OPTIMAL TREE CHARACTERISTICS $A_m(n)$, $k_m(n)$ WHEN $\alpha = \beta = 1$

	$m = 1$	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$	$m = 7$	$m = 8$	$m = 9$	$m = 10$	$m = 11$	$m = 12$	Tree specification	
$n = 1$	0, 0												—	$n = 1$
$n = 2$	0, 2	0, 1											1:1	$n = 2$
$n = 3$	12, 3	6, 1	0, 1										1:1:1	$n = 3$
$n = 4$	20, 4	12, 1	6, 1	0, 1									1:1:1:1	$n = 4$
$n = 5$	30, 5	18, 2	12, 1	6, 1	0, 1								1:1:1:1:1	$n = 5$
$n = 6$	42, 2	24, 3	18, 1	12, 1	6, 1	0, 1							3:3	$n = 6$
$n = 7$	52, 3	32, 3	24, 1	18, 1	12, 1	6, 1	0, 1						1:3:3	$n = 7$
$n = 8$	62, 3	40, 4	30, 2	24, 1	18, 1	12, 1	6, 1	0, 1					2:3:3	$n = 8$
$n = 9$	72, 3	50, 4	36, 3	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1				3:3:3	$n = 9$
$n = 10$	84, 3	60, 5	44, 3	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1			3:3:4	$n = 10$
$n = 11$	96, 3	72, 4	52, 3	42, 2	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1		3:4:4	$n = 11$
$n = 12$	108, 3	82, 4	60, 4	48, 3	42, 1	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1	4:4:4	$n = 12$
$n = 13$	121, 4	92, 4	70, 4	56, 3	48, 1	42, 1	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	3:3:3:4	$n = 13$
$n = 14$	134, 4	102, 5	80, 4	64, 3	54, 2	48, 1	42, 1	36, 1	30, 1	24, 1	18, 1	12, 1	3:3:4:4	$n = 14$
$n = 15$	147, 4	114, 5	90, 5	72, 3	60, 3	54, 1	48, 1	42, 1	36, 1	30, 1	24, 1	18, 1	3:4:4:4	$n = 15$
$n = 16$	160, 4	124, 7	102, 4	80, 4	68, 3	60, 1	54, 1	48, 1	42, 1	36, 1	30, 1	24, 1	4:4:4:4	$n = 16$
$n = 17$	175, 4	134, 8	112, 4	90, 4	76, 3	66, 2	60, 1	54, 1	48, 1	42, 1	36, 1	30, 1	4:4:4:5	$n = 17$
$n = 18$	190, 4	144, 9	122, 4	100, 4	84, 3	72, 3	66, 1	60, 1	54, 1	48, 1	42, 1	36, 1	4:4:5:5	$n = 18$
$n = 19$	205, 4	156, 9	132, 5	110, 4	92, 3	80, 3	72, 1	66, 1	60, 1	54, 1	48, 1	42, 1	4:5:5:5	$n = 19$
$n = 20$	220, 4	168, 9	144, 4	120, 5	100, 4	88, 3	78, 2	72, 1	66, 1	60, 1	54, 1	48, 1	5:5:5:5	$n = 20$
$n = 21$	236, 5	180, 9	154, 4	132, 4	110, 4	96, 3	84, 3	78, 1	72, 1	66, 1	60, 1	54, 1	4:4:4:4:5	$n = 21$
$n = 22$	252, 3	192, 10	164, 4	142, 4	120, 4	104, 3	92, 3	84, 1	78, 1	72, 1	66, 1	60, 1	4:9:9	$n = 22$
$n = 23$	266, 3	204, 11	174, 5	152, 4	130, 4	112, 3	100, 3	90, 2	84, 1	78, 1	72, 1	66, 1	5:9:9	$n = 23$
$n = 24$	282, 3	216, 12	186, 5	162, 5	140, 4	120, 4	108, 3	96, 3	90, 1	84, 1	78, 1	72, 1	5:9:10	$n = 24$
$n = 25$	296, 3	229, 12	196, 7	174, 4	150, 5	130, 4	116, 3	104, 3	96, 1	90, 1	84, 1	78, 1	7:9:9	$n = 25$

Our derivation of (2) shows that the relation $\alpha \leq \beta$ will hold whenever $P + 1$ equal buffer areas are used. The limiting case $\alpha = \beta$, shown in Table 1 and Fig. 93, occurs when the seek time itself is to be minimized without regard to transmission time.

Returning to our original application, we still haven't considered how to get the initial runs in the first place; without read/write/compute overlap, replacement selection loses some of its advantages. Perhaps we should fill the entire internal memory, sort it, and output the results; such input and output operations can each be done with one seek. Or perhaps we are better off using, say, 20 percent of the memory as a combination input/output buffer, and doing replacement selection. This requires five times as many seeks (an extra 60 seconds or so!), but it reduces the number of initial runs from 100 to 64; the reduction would be more dramatic if the input file were pretty much in order already.

If we decide not to use replacement selection, the optimum tree for $S = 100$, $\alpha = 0.00145$, $\beta = 0.01545$ [cf. (2)] turns out to be rather prosaic: It is simply a 10-way merge, completed in two passes over the data. Allowing 30 seconds for internal sorting (100 quicksorts, say), the initial distribution pass takes about $2\frac{1}{2}$ minutes, and the merge passes each take almost 5 minutes, for a total of 12.4 minutes. If we decide to use replacement selection, the optimal tree for $S = 64$ turns out to be equally uninteresting (two 8-way merge passes); the initial distribution pass takes about $3\frac{1}{2}$ minutes, the merge passes each take about $4\frac{1}{2}$ minutes, and the estimated total time comes to 12.6 minutes. Remember that both of these methods give up virtually all read/write/compute overlap in order to have larger buffers (reducing seek time). None of these estimated

times includes the time which might be necessary for "read-back check" operations.

In practice the final merge pass tends to be quite different from the others; for example, the output is often expanded and/or written onto tape. In such cases the tree pattern should be chosen using a different optimality criterion at the root.

***Another way to allocate buffers.** David E. Ferguson [*CACM* **14** (1971), 476–478] has pointed out that seek time can be reduced if we don't make all buffers the same size. The same idea occurred at about the same time to several other people [S. J. Waters, *Comp. J.* **14** (1971), 109–112; Ewing S. Walker, Software Age **4** (August-September, 1970), 16–17].

Suppose we are doing a four-way merge on runs of equal length L_0, with M characters of memory. If we divide the memory into equal buffers of size $B = M/5$, we need about L_0/B seeks on each input file and $4L_0/B$ seeks for the output, totalling $8L_0/B = 40L_0/M$ seeks. But if we use four input buffers of size $M/6$ and one output buffer of size $M/3$, we need only about $4 \times (6L_0/M) + 4 \times (3L_0/M) = 36L_0/M$ seeks! The transmission time is the same in both cases, so we haven't lost anything by the change.

In general, suppose that we want to merge sorted files of lengths $L_1, \ldots, L_P$ into a sorted file of length $L_{P+1} = L_1 + \cdots + L_P$, and assume that a buffer of size B_k is being used for the kth file. Thus

$$B_1 + \cdots + B_P + B_{P+1} = M, \tag{6}$$

where M is the total size of available internal memory. The number of seeks will be approximately

$$\frac{L_1}{B_1} + \cdots + \frac{L_P}{B_P} + \frac{L_{P+1}}{B_{P+1}}. \tag{7}$$

Let's try to minimize this quantity, subject to condition (6), assuming for convenience that the B_k's don't have to be integers. If we increase B_j by δ and decrease B_k by the same small amount, the number of seeks changes by

$$\frac{L_j}{B_j + \delta} - \frac{L_j}{B_j} + \frac{L_k}{B_k - \delta} - \frac{L_k}{B_k} = \left(\frac{L_k}{B_k(B_k - \delta)} - \frac{L_j}{B_j(B_j + \delta)}\right)\delta,$$

so the allocation can be improved if $L_j/B_j^2 \neq L_k/B_k^2$. Therefore we get the minimum number of seeks only if

$$\frac{L_1}{B_2^2} = \cdots = \frac{L_P}{B_P^2} = \frac{L_{P+1}}{B_{P+1}^2}. \tag{8}$$

Since a minimum does exist it must occur when

$$B_k = \sqrt{L_k}\, M/(\sqrt{L_1} + \cdots + \sqrt{L_{P+1}}), \qquad 1 \le k \le P + 1, \tag{9}$$

for these are the only values of $B_1, \ldots, B_{P+1}$ which simultaneously satisfy (6) and (8). Putting these values into (7) gives a fairly simple formula for the total number of seeks,

$$(\sqrt{L_1} + \cdots + \sqrt{L_{P+1}})^2/M, \tag{10}$$

which may be compared with the number $(P+1)(L_1 + \cdots + L_{P+1})/M$ obtained if all buffers are equal in length. By exercise 1.2.3–31, the improvement is $\sum_{1 \le j < k \le P+1} (\sqrt{L_j} - \sqrt{L_k})^2/M$. Unfortunately formula (10) does not lend itself to an easy determination of optimum merge patterns as in Theorem K (cf. exercise 14).

Further considerations. If two disk units are being used, one for reading and one for writing, we can overlap some of the write time at the expense of an extra output buffer. If the "forecasting" technique is used (i.e., looking at the last keys in each input buffer in order to determine which buffer will empty first), we can overlap some of the read time. Unfortunately a simple synchronization procedure like that of Algorithm 5.4.6F does not provide continuous input/output, since the variable behavior of seek time means that input and output operations won't finish at the same time. The exact amount of overlap is hard to estimate, and it may be best to simulate the merging process in order to find constants α and β which approximate the running time as in Theorem H.

If we use "floating" input buffers (not dedicated to a particular file), they must all be the same size, so the above buffer allocation technique cannot be used except that the output buffers may be chosen larger than those for input. If $2P$ input buffers are used instead of P, we overlap a good deal of the input time yet double the seek time, so we may be gaining nothing; perhaps $P+1$ or $P+2$ input buffers would be best.

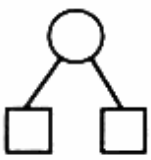
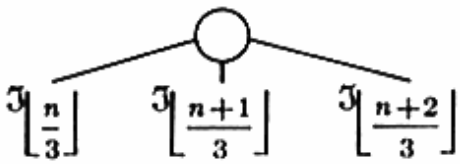
It should be clear to the reader by now that no clear cut strategy for optimum disk sorting has been worked out; the number of available options greatly exceeds the number of strategies that have been theoretically analyzed. Although the theory developed in this section provides some useful insights, a good deal of experimentation still needs to be done.

***A closer look at optimal trees.** It is interesting to examine the extreme case $\beta = 0$ in Theorems H and K, even though practical situations usually lead to parameters with $0 \le \alpha \le \beta$. What tree with n leaves has the smallest possible degree path length? Curiously it turns out that three-way merging is best in this case.

Theorem L. *The degree path length of a tree with n leaves is never less than*

$$f(n) = \begin{cases} 3qn + 2(n - 3^q), & \text{if} \quad 2 \cdot 3^{q-1} \le n \le 3^q; \\ 3qn + 4(n - 3^q), & \text{if} \quad 3^q \le n \le 2 \cdot 3^q. \end{cases} \tag{11}$$

Ternary trees $\mathfrak{I}_n$ defined by the rules

$\mathfrak{I}_1 =$, $\mathfrak{I}_2 =$  , $\mathfrak{I}_n =$ 

(12)

have the minimum degree path length.

Proof. It is important to observe that $f(n)$ is a *convex function*, i.e. that

$$f(n+1) - f(n) \geq f(n) - f(n-1) \qquad \text{for all} \qquad n \geq 2. \tag{13}$$

The relevance of this property is due to the following lemma.

Lemma C. *A function $g(n)$ defined on the positive integers satisfies*

$$\min_{1 \leq k \leq n} (g(k) + g(n-k)) = g(\lfloor n/2 \rfloor) + g(\lceil n/2 \rceil), \qquad n \geq 2, \tag{14}$$

if and only if it is convex.

Proof. If $g(n+1) - g(n) < g(n) - g(n-1)$ for some $n \geq 2$, we have $g(n+1) + g(n-1) < g(n) + g(n)$, contradicting (14). Conversely, if (13) holds for g, and $1 \leq k < n-k$, we have $g(k+1) + g(n-k-1) \leq g(k) + g(n-k)$ by convexity. ∎

The latter part of Lemma C's proof can be extended for any $m \geq 2$ to show that

$$\min_{\substack{n_1+\cdots+n_m=n \\ n_1,\ldots,n_m \geq 1}} (g(n_1) + \cdots + g(n_m))$$
$$= g(\lfloor n/m \rfloor) + g(\lfloor (n+1)/m \rfloor) + \cdots + g(\lfloor (n+m-1)/m \rfloor), \tag{15}$$

whenever g is convex. Let

$$f_m(n) = f(\lfloor n/m \rfloor) + f(\lfloor (n+1)/m \rfloor) + \cdots + f(\lfloor (n+m-1)/m \rfloor); \tag{16}$$

the proof of Theorem L is completed by proving that $f_3(n) + 3n = f(n)$ and $f_m(n) + mn \geq f(n)$ for all $m \geq 2$. (See exercise 11.) ∎

It would be very nice if optimal trees could always be characterized neatly as in Theorem L. But the results we have seen for $\alpha = \beta$ in Table 1 show that the function $A_1(n)$ is not always convex. In fact, Table 1 is sufficient to disprove most simple conjectures about optimal trees! We can, however, salvage part of Theorem L in the general case; M. Schlumberger and J. Vuillemin have shown that *large* orders of merge can always be avoided:

Theorem M. *Given α and β as in Theorem H, there exists an optimal tree in which the degree of every node is at most*

$$d(\alpha, \beta) = \left\lceil \min_{k \geq 1} \left(k + \left(1 + \frac{1}{k}\right)\left(1 + \frac{\beta}{\alpha}\right)\right)\right\rceil. \tag{17}$$

Proof. Let $n_1, \ldots, n_m$ be positive integers such that $n_1 + \cdots + n_m = n$, $A(n_1) + \cdots + A(n_m) = A_m(n)$, and $n_1 \leq \cdots \leq n_m$, and assume that $m \geq d(\alpha, \beta) + 1$. Let k be the value which minimizes (17); we shall show that

$$\alpha n(m-k) + \beta n + A_{m-k}(n) \leq \alpha nm + \beta n + A_m(n), \tag{18}$$

hence the minimum value in (4) is always achieved for some $m \le d(\alpha, \beta)$.

By definition, since $m \ge k + 2$, we must have

$$\begin{aligned} A_{m-k}(n) &\le A_1(n_1 + \cdots + n_{k+1}) + A_1(n_{k+2}) + \cdots + A_1(n_m) \\ &\le \alpha(n_1 + \cdots + n_{k+1})(k+1) + \beta(n_1 + \cdots + n_{k+1}) \\ &\quad + A_1(n_1) + \cdots + A_1(n_m) \\ &= (\alpha(k+1) + \beta)(n_1 + \cdots + n_{k+1}) + A_m(n) \\ &\le (\alpha(k+1) + \beta)(k+1)n/m + A_m(n), \end{aligned}$$

and (18) now follows easily. (Careful inspection of this proof shows that (17) is "best possible," in the sense that some optimal trees must have nodes of degree $d(\alpha, \beta)$; see exercise 13.) ∎

The construction in Theorem K requires $O(N^2)$ memory cells and $O(N^2 \log N)$ steps to evaluate $A_m(n)$ for $1 \le m \le n \le N$; Theorem M shows that only $O(N)$ cells and $O(N^2)$ steps are needed. Schlumberger and Vuillemin have discovered several more very interesting properties of optimal trees ["Optimal disk merge patterns," to appear].

The use of chaining. M. A. Goetz [*CACM* **6** (1963), 245–248] has suggested an interesting way to avoid seek time on output, by linking individual tracks together. His idea requires a fairly fancy set of disk storage management routines, but it applies to many problems besides sorting, and it may therefore be a very worthwhile technique for general-purpose use.

The concept is simple: Instead of allocating tracks sequentially within cylinders of the disk, we link them together and maintain lists of available space, one for each cylinder. When it is time to output a track of information, we write it on the current cylinder (wherever the access arm happens to be), unless that cylinder is full. In this way the seek time usually disappears.

The catch is that we can't store a link-to-next-track within the track itself, since the necessary information isn't known at the right time. (We could store a link-to-previous-track and read the file backward on the next pass, if that were suitable.) A separate table of link addresses for the tracks of each file can be maintained, perhaps partly on disk itself. The available space lists can be compactly represented by using bit tables, with 1000 bits specifying the availability or unavailability of 1000 tracks.

Will keysorting help? When records are long and keys are short, it is very tempting to create a new file consisting simply of the keys together with a serial number specifying their original file location. After sorting this key file, we can replace the keys by the successive numbers 1, 2, . . .; the new file can then be sorted by original file location and we will have a convenient specification of how to unshuffle the records for the final rearrangement. Schematically, the process has the following form:

a)	Original file	$(K_1, I_1)(K_2, I_2) \ldots (K_N, I_N)$	long
b)	Key file	$(K_1, 1)(K_2, 2) \ldots (K_N, N)$	short
c)	Sorted (b)	$(K_{p_1}, p_1)(K_{p_2}, p_2) \ldots (K_{p_N}, p_N)$	short
d)	Edited (c)	$(1, p_1)(2, p_2) \ldots (N, p_N)$	short
e)	Sorted (d)	$(q_1, 1)(q_2, 2) \ldots (q_N, N)$	short
f)	Edited (a)	$(q_1, I_1)(q_2, I_2) \ldots (q_N, I_N)$	long

Here $p_j = k$ if and only if $q_k = j$; the two sorting processes in (c), (e) are comparatively fast (perhaps even internal sorts), since the records aren't very long. In stage (f) we have reduced the problem to sorting a file whose keys are simply the numbers $\{1, 2, \ldots, N\}$; each record now specifies exactly where it is to be moved.

The external rearrangement problem which remains after stage (f) seems trivial, at first glance; but in fact it is rather difficult, and no really good algorithms (significantly better than sorting) have yet been found. We could obviously do the rearrangement in N steps, moving one record at a time; for large enough N this is better than the $N \log N$ of a sorting method. But N is never that large, and N seeks are unthinkable!

A radix sorting method can be used efficiently on the edited records, since their keys have a perfectly uniform distribution. On many high-speed computers with high-speed peripheral devices, the processing time for an eight-way distribution is much faster than the processing time for an eight-way merge; hence a distribution sort may be the best procedure. (See Section 5.4.7, and see also exercise 17.)

But it seems wasteful to do a key sort only to follow it by a full sort! One reason the external rearrangement problem is unexpectedly difficult has been discovered by R. W. Floyd, who found a nontrivial lower bound on the number of seeks required to rearrange records on a disk file.

It is convenient to describe Floyd's result in terms of the elevator problem of Section 5.4.8; this time we want to find an elevator schedule which minimizes the number of *stops*, instead of minimizing the distance traveled. (Minimizing the number of stops is not precisely equivalent to finding the minimum-seek rearrangement algorithm, since a stop combines input to the elevator with output from the elevator; but the stop-minimization criterion is close enough to indicate the basic ideas.)

We shall make use of the "discrete entropy" function

$$F(n) = \sum_{1<k\le n} (\lceil \log_2 k \rceil + 1) = B(n) + n - 1 = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + n, \tag{19}$$

where $B(n)$ is the "binary insertion" function, Eq. 5.3.1–3. By Eq. 5.3.1–33, $F(n)$ is the minimum external path length of a binary tree with n leaves, and

$$n \log_2 n \le F(n) \le n \log_2 n + 0.0861n. \tag{20}$$

Since $F(n)$ is convex and satisfies $F(n) = n + F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$, we know by Lemma C above that

$$F(n) \le F(k) + F(n-k) + n, \qquad \text{for} \qquad 0 \le k \le n. \tag{21}$$

This relation is also evident from the external path length characterization of F; it is the crucial fact used in what follows.

As in Section 5.4.8 we shall assume that the elevator holds b people, each floor holds c people, and there are n floors. Let s_{ij} be the number of people currently on floor i whose destination is floor j. The *togetherness rating* of any configuration of people in the building is defined to be the sum $\sum_{1 \le i,j \le n} F(s_{ij})$.

For example, assume that $b = c = n = 6$ and that the 36 people are initially scattered among the floors as follows:

```
␣␣␣␣␣␣
123456  123456  123456  123456  123456  123456
```
(22)

The elevator is empty, sitting on floor 1; "␣" denotes a vacant position. Each floor contains one person with each possible destination, so all s_{ij} are 1 and the togetherness rating is zero. If the elevator now transports six people to floor 2, we have the configuration

```
        123456
␣␣␣␣␣␣  123456  123456  123456  123456  123456
```
(23)

and the togetherness rating becomes $6F(0) + 24F(1) + 6F(2) = 12$. Suppose the elevator carries 1, 1, 2, 3, 3, 4 to floor 3:

```
                112334
␣␣␣␣␣␣  245566  123456  123456  123456  123456
```
(24)

The togetherness rating has jumped to $4F(2) + 2F(3) = 18$. When everyone has finally been transported to his or her destination, the togetherness rating will be $6F(6) = 96$.

Floyd observed that the togetherness rating can never increase by more than $b + c$ at each stop, since a set of s equal-destination people joining with a similar set of size s' improves the rating by $F(s + s') - F(s) - F(s') \le s + s'$. Therefore we have the following result.

Theorem F. *Let t be the togetherness rating of an initial configuration of people, in terms of the definitions above. The elevator must make at least*

$$\lceil (nF(c) - t)/(b + c) \rceil$$

stops in order to bring them all to their destinations. ∎

Translating this result into disk terminology, let there be N records, with B per block, and suppose the internal memory can hold M records at a time.

Every disk read brings one block into memory, every disk write stores one block, and s_{ij} is the number of records in block i that belong in block j. If $N \geq B^2$, there are initial configurations in which all the s_{ij} are ≤ 1, so at least $(N/B)F(B)/M \approx (N \log_2 B)/M$ block-reading operations are necessary to rearrange the records. (The factor $\log_2 B$ makes this lower bound nontrivial when B is large, but when $B = 1$ the lower bound can obviously be improved.)

EXERCISES

1. [*M22*] The text explains a method by which the average latency time required to read a fraction x of a track is reduced from $\frac{1}{2}$ to $\frac{1}{2}(1 - x^2)$ revolutions. This is the minimum possible value, when there is one access arm. What is the corresponding minimum average latency time if there are *two* access arms, 180° apart, assuming that only one arm can transmit data at any one time?

2. [*M30*] (A. G. Konheim.) The purpose of this problem is to investigate how far the access arm of a disk must move while merging files that are allocated "orthogonally" to the cylinders. Suppose there are P files, each containing L blocks of records, and assume that the first block of each file appears on cylinder 1, the second on cylinder 2, etc. The relative order of the last keys in each block governs the access arm motion during the merge, hence we may represent the situation in the following mathematically tractable way: Consider a set of PL ordered pairs

$$\begin{array}{cccc} (a_{11}, 1) & (a_{21}, 1) & \ldots & (a_{p1}, 1) \\ (a_{12}, 2) & (a_{22}, 2) & \ldots & (a_{p2}, 2) \\ \vdots & \vdots & & \vdots \\ (a_{1L}, L) & (a_{2L}, L) & \ldots & (a_{pL}, L) \end{array}$$

where the set $\{a_{ij} \mid 1 \le i \le P, 1 \le j \le L\}$ consists of the numbers $\{1, 2, \ldots, PL\}$ in some order, and where $a_{ij} < a_{i,j+1}$ for $1 \le j < L$. (Rows represent cylinders, columns represent input files.) Sort the pairs on their first components and let the resulting sequence be $(1, j_1)\ (2, j_2) \ldots (PL, j_{PL})$. Show that, if each of the $(PL)!/L!^P$ choices of the a_{ij} is equally likely, the average value of

$$|j_2 - j_1| + |j_3 - j_2| + \cdots + |j_{PL} - j_{PL-1}|$$

is

$$(L-1)\left(1 + (P-1)2^{2L-2} \Big/ \binom{2L}{L}\right).$$

[*Hint:* See exercise 5.2.1–14.] Note that as $L \to \infty$ this value is asymptotically $\frac{1}{4}(P-1)L\sqrt{\pi L}$.

3. [*M15*] Suppose the internal memory is limited so that 10-way merging is not feasible. How can recurrence relations (3), (4), (5) be modified so that $A_1(n)$ is the minimum value of $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$, over all n-leaved trees $\mathfrak{I}$ having no internal nodes of degree greater than 9?

▸ **4.** [*M21*] (a) Derive a formula corresponding to (2), for the running time of an L-character P-way merge, when a modified form of the square root buffer allocation is used: All P of the input buffers should have equal length, but the output buffer size should be chosen so as to minimize seek time.

(b) Show that the construction in Theorem K can be modified in order to obtain a merge pattern which is optimal according to your formula from part (a).

5. [*M20*] When two disks are being used, so that reading on one is overlapped with writing on the other, we cannot use merge patterns like that of Fig. 93 since some leaves are at even levels and some are at odd levels. Show how to modify the construction of Theorem K in order to produce trees that are optimal subject to the constraint that all leaves appear on even levels or all on odd levels.

▸ **6.** [*22*] Find a tree which is optimum in the sense of exercise 5, when $n = 23$ and $\alpha = \beta = 1$. (You may wish to use a computer.)

▸ **7.** [*M24*] When the initial runs are not all the same length, the best merge pattern (in the sense of Theorem H) minimizes $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$, where $D(\mathfrak{I})$ and $E(\mathfrak{I})$ now represent *weighted* path lengths: Weights $w_1, \ldots, w_n$ (corresponding to the lengths of the initial runs) are attached to each leaf of the tree, and the degree sums and path lengths are multiplied by the appropriate weights. For example, if $\mathfrak{I}$ is the tree of Fig. 92, we would have $D(\mathfrak{I}) = 6w_1 + 6w_2 + 7w_3 + 9w_4 + 9w_5 + 7w_3 + 4w_7 + 4w_8$, $E(\mathfrak{I}) = 2w_1 + 2w_2 + 2w_3 + 3w_4 + 3w_5 + 2w_6 + w_7 + w_8$.

Prove that there is always an optimal pattern in which the *shortest* k runs are merged first, for some k.

8. [*49*] Is there an algorithm which finds optimal trees for given α, β and weights $w_1, \ldots, w_n$, in the sense of exercise 7, taking only $O(n^c)$ steps for some c?

9. [*M49*] (Schlumberger and Vuillemin.) Do there exist n-leaf optimal trees in the sense of Theorem H, such that all leaves appear on at most two adjacent levels, for all n, α, and β? When α and β are fixed, is there a number m such that $A_1(n) = \alpha mn + \beta n + A_m(n)$ for all sufficiently large n? (Both of these conjectures have been verified when $\alpha/\beta \geq 4$.)

10. [*HM49*] Given α and β, what is the asymptotic value of $A_1(n)$ as $n \to \infty$?

11. [*M29*] In the notation of (11) and (16), prove that $f_m(n) + mn \geq f(n)$ for all $m, n \geq 2$, and determine all m and n for which equality holds.

12. [*25*] Prove that, for all $n > 0$, there is a tree with n leaves and minimum degree path length (11), with all leaves at the same level.

13. [*M24*] Show that for $2 \leq n \leq d(\alpha, \beta)$, the unique best merge pattern in the sense of Theorem H is an n-way merge. (Cf. (17).)

14. [*40*] Using the square root method of buffer allocation, the seek time for the merge pattern in Fig. 92 would be proportional to $(\sqrt{2} + \sqrt{4} + \sqrt{1} + \sqrt{1} + \sqrt{8})^2 + (\sqrt{1} + \sqrt{1} + \sqrt{2})^2 + (\sqrt{1} + \sqrt{2} + \sqrt{1} + \sqrt{4})^2 + (\sqrt{1} + \sqrt{1} + \sqrt{2})^2$; this is the sum, over each internal node, of $(\sqrt{n_1} + \cdots + \sqrt{n_m} + \sqrt{n_1 + \cdots + n_m})^2$, where that node's respective subtrees have $(n_1, \ldots, n_m)$ leaves. Write a computer program which generates minimum-seek time trees having 1, 2, 3, . . . leaves, using this formula to estimate the seek time.

15. [*M22*] Show that Theorem F can be improved slightly if the elevator is initially

empty and if $nF(c) \neq t$: At least $\lceil (nF(c) + b - t)/(b + c) \rceil$ stops are necessary in such a case.

16. [*23*] (R. W. Floyd.) Find an elevator schedule which transports all the people of (22) to their destinations in 12 stops or less. (Configuration (23) shows the situation after one, not two, stops.)

17. [*25*] (B. T. Bennett and A. C. McKellar, 1971.) Consider the following approach to keysorting, illustrated on an example file with 10 keys:

a) Original file: $(50, I_0)(08, I_1)(51, I_2)(06, I_3)(90, I_4)(17, I_5)(89, I_6)(27, I_7)(65, I_8)(42, I_9)$
b) Key file: (50, 0)(08, 1)(51, 2)(06, 3)(90, 4)(17, 5)(89, 6)(27, 7)(65, 8)(42, 9)
c) Sorted (b): (06, 3)(08, 1)(17, 5)(27, 7)(42, 9)(50, 0)(51, 2)(65, 8)(89, 6)(90, 4)
d) Bin assignments: (3, 3)(3, 9)(2, 1)(2, 5)(2, 7)(2, 8)(1, 0)(1, 2)(1, 4)(1, 6)
e) Sorted (d): (1, 0)(2, 1)(1, 2)(3, 3)(1, 4)(2, 5)(1, 6)(2, 7)(2, 8)(3, 9)
f) (a) distributed into bins using (e):
   Bin 1: $(50, I_0)(51, I_2)(90, I_4)(89, I_6)$
   Bin 2: $(08, I_1)(17, I_5)(27, I_7)(65, I_8)$
   Bin 3: $(06, I_3)(42, I_9)$
g) The result of replacement selection, reading first bin 3, then bin 2, then bin 1:
   $(06, I_3)(08, I_1)(17, I_5)(27, I_7)(42, I_9)(50, I_0)(51, I_2)(65, I_8)(89, I_6)(90, I_4)$

The assignment of bin numbers in step (d) is made by doing *replacement selection* on (c), *from right to left*, in *decreasing* order of the second component. The bin number is the run number. The above example uses replacement selection with only two elements in the selection tree; the same size tree should be used for replacement selection in both (d) and (g). Note that the bin contents are not necessarily in sorted order!

Prove that this method will sort, i.e., that the replacement selection in (g) will produce only one run. (This technique reduces the number of bins needed in a conventional keysort by distribution, especially if the input is largely in order already.)

▶ **18.** [*25*] Some hardware/software systems provide programmers with a *virtual memory:* Programs can be written as if there were a very large internal memory, able to contain all of the data. This memory is divided into *pages*, only a few of which are in the actual internal memory at any one time; the others are on disks or drums. The programmer need not concern himself with such details, since the system takes care of everything; new pages are automatically brought into memory when needed.

It would seem that the advent of virtual memory technology makes external sorting methods obsolete, since the job can simply be done using the techniques developed for internal sorting. Discuss this situation; in what ways might a hand-tailored external sorting method be better than the application of a general-purpose paging technique to an internal sorting method?

5.5. SUMMARY, HISTORY, AND BIBLIOGRAPHY

Now that we have nearly reached the end of this enormously long chapter, we had better "sort out" the most important facts we have studied.

An algorithm for sorting is a procedure which rearranges a file of records so that the keys are in ascending order. This orderly arrangement is useful because it brings equal-key records together, it allows efficient processing of multiple files sorted on the same key, it leads to efficient retrieval algorithms, and it makes computer output look more authentic.

Internal sorting is used when all of the records fit in the computer's high speed internal memory. We have studied more than two dozen algorithms for internal sorting, in various degrees of detail; and perhaps we would be happier if we didn't know so many different approaches to the problem! It was fun to learn all the techniques, but now we must face the horrible prospect of actually deciding which method ought to be used in a given situation.

It would be nice if only one or two of the sorting methods would dominate all of the others, regardless of the application or the computer being used. But in fact, each method has its own peculiar virtues. For example, the bubble sort (Algorithm 5.2.2B) has no apparent redeeming features, since there is always a better way to do what it does; but even this technique, suitably generalized, turns out to be useful for two-tape sorting (cf. Section 5.4.8). Thus we find that nearly all of the algorithms deserve to be remembered, since there are some applications in which they turn out to be best.

The following brief survey gives the highlights of the most significant algorithms we have encountered for internal sorting. As usual, N stands for the number of records in the given file.

1. *Distribution counting*, Algorithm 5.2D, is very useful when the keys have a small range. It is stable (doesn't affect the order of records with equal keys), but requires memory space for counters and for $2N$ records. A modification which saves N of these record spaces at the cost of stability appears in exercise 5.2–13.

2. *Straight insertion*, Algorithm 5.2.1S, is the simplest method to program, requires no extra space, and is quite efficient for small N (say $N \leq 25$). For large N it is unbearably slow unless the input is nearly in order.

3. *Diminishing increment sort*, Algorithm 5.2.1D (Shell's method), is also quite easy to program, and uses minimum memory space; and it is reasonably efficient for moderately large N (say $N \leq 1000$).

4. *List insertion*, Algorithm 5.2.1L, uses the same basic idea as straight insertion, so it is suitable only for small N. Like the other list sorting methods described below, it saves the cost of moving long records by manipulating links; this is particularly advantageous when the records have variable length or are part of other data structures.

5. *Address calculation* techniques are efficient when the keys have a known

(usually uniform) distribution; the principal variants of this approach are *multiple list insertion* (Algorithm 5.2.1M), and MacLaren's combined radix-insertion method (discussed at the close of Section 5.2.5). The latter can be done with only $O(\sqrt{N})$ cells of additional memory.

6. *Merge exchange,* Algorithm 5.2.2M (Batcher's method) and its cousin the *bitonic sort* (exercise 5.3.4–10) are useful when a large number of comparisons can be made simultaneously.

7. *Partition exchange,* Algorithm 5.2.2Q (Hoare's method, commonly known as *quicksort*) is probably the most useful general-purpose technique for internal sorting, because it requires very little memory space and its average running time on most computers beats that of its competitors. It can run *very* slowly in its worst case, however, so a careful choice of the partitioning elements should be made whenever nonrandom data are likely. Choosing the median of three elements, as suggested in exercise 5.2.2–55, makes the worst-case behavior extremely unlikely and also gives a slight improvement to the average running time.

8. *Straight selection,* Algorithm 5.2.3S, is a simple method especially suitable when special hardware is available to find the smallest element of a list at high speed.

9. *Heapsort,* Algorithm 5.2.3H, requires minimum memory and is guaranteed to run pretty fast; its average time and its maximum time are both roughly twice the average running time of quicksort.

10. *List merging,* Algorithm 5.2.4L, is a list sort which, like heapsort, is guaranteed to be rather fast even in its worst case; moreover, it is stable with respect to equal keys.

11. *Radix sorting,* using Algorithm 5.2.5R, is a list sort especially appropriate for keys which are either rather short or which have an unusual lexicographic collating sequence. Alternatively, the distribution counting method (point 1 above) could be used instead of links; such a procedure requires $2N$ record spaces, plus a table of counters, but the simple form of its inner loop makes it especially good for ultra-fast, "number-crunching" computers which have look-ahead control. *Caution:* Radix sorting should not be used for small N!

12. *Merge insertion,* see Section 5.3.1, is especially suitable for very small N, in a "straight-line-coded" routine; for example, it would be the appropriate method in an application which requires the sorting of numerous five- or six-record groups.

13. Hybrid methods, combining one or more of the above techniques, are also possible. For example, merge insertion could be used for sorting short subfiles that arise in quicksort.

14. Finally, an unnamed method appearing in the answer to exercise 5.2.1–3 seems to require the shortest possible sorting program. But its average running time, proportional to N^3, makes it the slowest sorting routine in this book!

Table 1

A COMPARISON OF INTERNAL SORTING METHODS USING THE MIX COMPUTER

Method	Reference	Stable?	Length of MIX code	Space	Running time: Average	Running time: Maximum	Running time: $N = 16$	Running time: $N = 1000$	Notes
Comparison counting	Ex. 5.2–5	Yes	22	$N(1+\epsilon)$	$4N^2 + 10N$	$5.5N^2$	1065	3992427	c
Distribution counting	Ex. 5.2–9	Yes	26	$2N + 1000\epsilon$	$22N + 10000$	$22N$	10352	32000	a
Straight insertion	Ex. 5.2.1–33	Yes	12	N	$2N^2 + 9N$	$4N^2$	494	1985566	
Diminishing increments	Prog. 5.2.1D	No	21	N	$15N^{1.25}$	$cN^{1.5}$	529	131002	d, h
List insertion	Ex. 5.2.1–33	Yes	19	$N(1+\epsilon)$	$1.25N^2 + 13.25N$	$2.5N^2$	433	1248615	b, c
Multiple list insertion	Prog. 5.2.1M	No	18	$N + \epsilon(N + 100)$	$.0175N^2 + 18N$	$3.5N^2$	645	35246	b, c, f, i
Merge exchange	Ex. 5.2.2–12	No	36	N	$3.7N(\log_2 N)^2$	$cN \ln^2 N$	1284	379104	h
Partition exchange	Prog. 5.2.2Q	No	63	$N + 2\epsilon \log_2 N$	$12.67N \ln N - 1.92N$	$\geq 2N^2$	518	81946	
Median-of-3 quicksort	Ex. 5.2.2–55	No	91	$N + 2\epsilon \log_2 N$	$11.66N \ln N + 1.89N$	$\geq N^2$	567	81485	e
Radix exchange	Prog. 5.2.2R	No	45	$N + 68\epsilon$	$14.43N \ln N + 23.9N$	$272N$	1135	137614	g, i, j
Straight selection	Prog. 5.2.3S	Yes	15	N	$2.5N^2 + 3N \ln N$	$3.25N^2$	853	2525287	j
Heapsort	Prog. 5.2.3H	No	30	N	$23.08N \ln N + 0.2N$	$26N \ln N$	1068	159714	h, j
List merge	Prog. 5.2.4L	Yes	44	$N(1+\epsilon)$	$14.43N \ln N + 4.92N$	$14.4N \ln N$	761	104716	b, c, j
Radix list sort	Prog. 5.2.5R	Yes	36	$N + \epsilon(N + 200)$	$32N + 4838$	$32N$	4250	36838	b, c

a: three-digit keys only.
b: six-digit (i.e., three-byte) keys only.
c: Output not rearranged; final sequence is specified implicitly by links or counters.
d: Increments chosen as in 5.2.1–8; a slightly better sequence appears in exercise 5.2.1–29.
e: $M = 10$, using SRB; for the version with DIV, add $\frac{13}{7}N$ to the average running time.
f: $M = 100$ (the byte size).
g: $M = 34$, since $2^{34} > 10^{10} > 2^{33}$.
h: Average time is based on empirical estimate, since the theory is incomplete.
i: Average time 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 summarizes the speed and space characteristics of many of these methods, when programmed for MIX. It is important to realize that the figures in this table are only rough indications of the relative sorting times; they apply to one computer only, and the assumptions made about input data are not completely consistent for all programs. Comparative tables such as this have been given by many authors, with no two people reaching the same conclusions! On the other hand, the timings do give at least an indication of the kind of speed to be expected from each algorithm, when sorting one-word records, since MIX is a fairly typical computer.

The "space" column in Table 1 gives some information about the amount of auxiliary memory used by each algorithm, in units of record length. Here "ϵ" denotes the fraction of a record needed for one link field; thus, for example, $N(1 + \epsilon)$ means that the method requires space for N records plus N link fields.

The asymptotic average and maximum times appearing in Table 1 give only the leading terms which dominate for large N, assuming random input; c denotes an unspecified constant. These formulas can often be misleading, so actual total running times have also been listed, for sample runs of the program on two particular sequences of input data. The case $N = 16$ refers to the sixteen keys which appear in so many of the examples of Section 5.2; and the case $N = 1000$ refers to the sequence $K_1, K_2, \ldots, K_{1000}$ defined by

$$K_0 = 0; \qquad K_{n+1} = (3141592621K_n + 2113148651) \bmod 10^{10}.$$

A reasonably "high-quality" MIX program has been used to represent each algorithm in the table, often incorporating improvements that have been suggested in the exercises. The byte size for these runs was 100.

External sorting techniques are different from internal sorting, because they must use comparatively primitive data structures, and because there is a great emphasis on minimizing their input/output time. Section 5.4.6 summarizes the interesting methods that have been developed for tape merging, and Section 5.4.9 discusses the use of disks and drums.

Of course, sorting isn't the whole story. While studying all of these sorting techniques, we have learned a good deal about how to handle data structures, how to deal with external memories, and how to analyze algorithms; and perhaps we have even learned a little about how to discover new algorithms.

Early developments. A search for the origin of today's sorting techniques takes us back to the nineteenth century, when the first machines for sorting were invented. The United States conducts a census of all its citizens, every ten years, and by 1880 the problem of processing the voluminous census data was becoming very acute; in fact, the total number of single (as opposed to married) people was never tabulated that year, although the necessary information had been gathered. Herman Hollerith, a 20-year-old employee of the Census Bureau, devised an ingenious electric tabulating machine to meet the need for better statistics-gathering, and about 100 of his machines were successfully used to tabulate the 1890 census rolls.