# SORTING ALGORITHMS

Roger L. Wainwright

Computer Science Department
The University of Tulsa
600 South College Avenue
Tulsa, Oklahoma 74104

An Article Appearing in the Encyclopedia
of Computer Science and Technology

# ENCYCLOPEDIA OF COMPUTER SCIENCE AND TECHNOLOGY

EXECUTIVE EDITORS

*Allen Kent* *James G. Williams*

UNIVERSITY OF PITTSBURGH
PITTSBURGH, PENNSYLVANIA

ADMINISTRATIVE EDITORS

*Rosalind Kent* *Carolyn M. Hall*

PITTSBURGH, PENNSYLVANIA

## VOLUME 21
## *SUPPLEMENT 6*

MARCEL DEKKER, INC. • NEW YORK and BASEL

# SORTING ALGORITHMS

TABLE OF CONTENTS Page

# SORTING ALGORITHMS

Sorting is the process by which a list of items is arranged in order by some criterion based on the content of each record. A record consists of a collection of one or more fields and a collection of records is called a file. Generally one of the fields in a record is used to distinguish one record from another. This is called the key field. For example, consider a telephone directory as a file. Suppose each record in the file consists of three fields: name, address, and phone number. The key field is usually the person's name, since a phone book is sorted (alphabetically) on the name field. However, it may be desirable to organize (sort) the records by the address field or perhaps by phone number. In many instances, several fields are used to sort the records in a file. Consider a telephone directory for a large corporation where each record consists of five fields: subsidiary company name, city, employee name, office number, and phone extension. Suppose the corporation has several subsidiaries and each subsidiary has offices in several cities worldwide. One possible organization for the records is to sort the employee name field within the city field within subsidiary.

We are an information society, collecting, organizing, analyzing, and distributing more information each year than the year before. There is no better way to do this than with computers. It is estimated from 40 to 90 percent of all computer programs use a sort routine at some time during its operation. Sorting algorithms are obviously very important and have been thoroughly studied. Sorting is an integral part of the study of algorithms and a major research topic in computer science. There is only one reason to sort a list of records: to later be able to search more efficiently for a particular record or range of records. For example, consider that we want to find a particular record out of a list of 1000 records. If the records are in no particular order then on the average one would have to look at 500 records before the desired record is found. Furthermore, if the desired record is not present at all in the file, then all 1000 records will need to be looked at before it can determine the desired record is absent. If this list were ordered, however, several more efficient search techniques could be used. Using a binary search, for example, the presence or absence of a particular record can be determined by looking at ten records at most for a file of size 1000. Consider a phone directory from a large city with 6 million names. To search for a particular name would take on the average 3 million examinations to find, and 6 million examinations if the desired name was not in the directory. For a sorted directory, using a binary search the determination of the presence or absence of a particular record can be verified in at most 23 attempts! If an interpolation search is used then even better results can be achieved. In general, if a file has n records arranged in sorted order then using the binary search technique the presence or absence of a particular record can be determined in x examinations

at most, where $2^{x-1} < n <= 2^x$. In general, the number of examinations required to find a record in a sorted list of n records is $\lceil \log_2 n \rceil$ at most, where $\lceil\ \rceil$ is the ceiling function.

## TIMING ANALYSIS

When the need arises to use a sorting algorithm, we are frequently faced with the problem of which sorting algorithm to use. One way to compare algorithms is to compute some measure of their efficiency. To be useful, this measure needs to be machine and language independent. For example, any algorithm written in a high-level language will run faster if rewritten in assembler language. Furthermore, any program running on a microcomputer will run slower than the same program running on a mainframe. One way to measure the running time of an algorithm is to determine the number of key operations used in the algorithm rather than measuring actual execution time. The key operations in sorting algorithms are (1) comparing two items and (2) swapping two items. This measurement is independent of machine and language since no matter how the algorithm is finally implemented the same number of key operations will still need to be performed. Since the number of comparisons in a sorting algorithm is usually much greater than the number of swaps, counting comparisons is a good measure of running time. Clearly, any measure of the speed of a sorting algorithm must be given in terms of the size of its input. Comparing one sorting algorithm sorting 50 items against a second sorting algorithm sorting 5000 items doesn't make sense.

Suppose we have an algorithm that operates on an input size of n items and takes $f(n) = 100n$ comparisons and a second algorithm that takes $g(n) = 2n^2$ comparisons. How do we compare these algorithms? We are interested in the size of the functions, f and g, as the values of n get large. This is called the asymptotic rate of growth. For small n it doesn't matter what sorting algorithm one uses; the sorting time will be very short. In the above example, when n is larger than 50, function g will be larger than function f. Function g has a greater asymptotic growth rate than function f and there is a significant difference between the two algorithms for large n. Suppose n is 1000 then f(n) = 100,000 and g(n) is two million. On the other hand, two algorithms taking 23n + 13 comparisons and 32n + 3 comparisons, respectively, will not be significantly different for large n. The important observation is that the number of comparisons in this case is proportional to the size of the input, n, and these algorithms have the same asymptotic growth rates.

To illustrate growth rates of functions we use what is known as the "big-oh" notation. For example, we say that the running time of a program is $O(n^2)$, read "big oh of order n squared," or "oh of n squared," or "order n squared." The formal definition is as follows: If F(n) and g(n) are functions defined over positive integers then f(n) is $O(g(n))$, that is "f(n) is big-oh of g(n)" if there exists a constant c such that $f(n) <= c \times g(n)$ for all n > some positive integer m. Generally, f(n) will represent the operation count for some algorithm and g(n) will be a function expressed as simply as possible. There are seven common orders used in analysis of algorithms given below in increasing asymptotic growth rate. We write $O(1)$ (constant time) to indicate the computing time is bounded by a constant and not dependent on n; $O(\log n)$ (logarithmic time) is time proportional to the

logarithm of n; $O(n)$ (linear time) means that the time is directly proportional to n. Other orders include $O(n\log n)$, $O(n^2)$ (quadratic time, $O(n^3)$ (cubic time) and $O(2^n)$ (exponential time).

For example, if f(n) = 23n + 13, then f(n) is $O(n)$ because we can choose c as 30 and m as 1; that is, 23n + 13 <= 30n for all n > 1. Furthermore, f(n) = 32n + 3 is also $O(n)$ since we can choose c as 40 and m as 1; that is, 32n + 3 <= 40n for all n > 1. If $f(n) = 2n^3 + 10n^2 + 21n + 40$, then f(n) is $O(n^3)$ since we can choose c as 3 and m as 12, since f(n) <= $3n^3$ for all n > 12. One advantage of the big-oh notation is that the less dominating terms need not be kept. Only the dominant term of the function ($n^3$ in our previous example) need to be considered since it is the fastest growing term. This makes it easier and simpler to compare and study the running times of algorithms. To further illustrate these concepts consider the following program segments:

```
(a)  for i := 1 to n do
       x[i] := x[i] - 1

(b)  for i := 1 to n do
       for j := 1 to n do
         x[i,j] := 2 × x[i,j]

(c)  for i := 1 to n do
       for j := 1 to n do
         begin
         c[i,j] := 0
         for k := 1 to n do
           c[i,j] := c[i,j] + a[i,k] × b[k,j]
         end
```

Segment (a) has a running time of $O(n)$. In segment (b) the last line is executed $n^2$ times, thus the running time is clearly $O(n^2)$. Segment (c) is a matrix multiplication algorithm involving three nested loops. The number of multiplications and additions in this algorithm in $O(n^3)$. For more details on analysis of algorithms, development of algorithms, classification of algorithms, big-oh and other notations the reader is referred elsewhere [*1–6*].

## INTERNAL SORTING TECHNIQUES

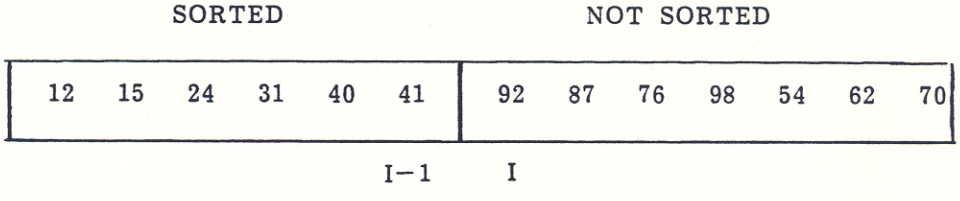
An important criteria to consider when sorting is the distinction between an internal and external environment. When an entire file can be brought into the main memory of a computer and sorted in main memory then this is called "internal sorting." If, however, the file is so large that it cannot fit in main memory then a technique must be used that involves the use of external memory such as tape or disk as well as main memory to sort the file. This is called "external sorting." In the next section we consider internal sorting techniques. External sorting techniques and parallel sorting algorithms will be considered later. To better illustrate the concepts and principles of each of the following sorting algorithms we will consider the problem of sorting an array, A[1..n] of integers in increasing order.

## Straight Selection Sort

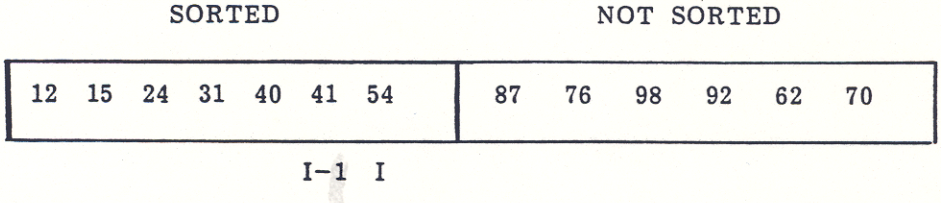
This is one of the easiest and most straightforward algorithms for sorting an array of keys. First the entire array, elements 1..n is searched for the smallest value. When this element is found, it is exchanged with the key in the first position of the array. This process is repeated considering elements 2..n of the array. That is, among elements 2..n of the array the smallest key is located and is exchanged with the element in the second position of the array. At this point the first two elements of the array have been determined. This process of searching for the next smallest key and placing it in its proper position is called a pass. There are n−1 passes required to sort an array of n elements. The algorithm expressed in pseudo code is given below:

```
for I := 1 to N−1 do
     /* invariant: A[1..I−1] is in sorted order */
     /* search elements I..N for the smallest key */
     /* and exchange this element with the key in position I */
```

A loop invariant such as the one used above is a statement that is true at the beginning of every iteration of the loop. To further illustrate the above loop invariant the following diagram indicates what happens during pass i. At the beginning of pass i the loop invariant states that elements A[1..i−1] are in sorted order as shown below for I = 7:




At the end of pass I, elements A[1..I] are in sorted order as shown below:




The program for straight selection sort is given below:

```
/* Sort A[1..N] using the Straight Selection Sort */
for I := 1 to N−1 do
  begin
  /* invariant: A[1..I−1] is in sorted order */
  /* search elements I..N for the smallest key */
  /* and exchange this element with the key in position I */

  MIN := I; /* MIN records the location of the smallest key thus far */
  for J := I+1 to N do
    if A[J] < A[MIN] then /* a new minimum key has been located */
      MIN := J;
```

```
    /* now A[MIN] contains the smallest key among elements A[I..N] */
    /* so exchange A[MIN] with A[I]                                */
    TEMP := A[I]; A[I] := A[MIN]; A[MIN] := TEMP;
    end;
```

*Analysis of Straight Selection Sort*

Straight selection sort makes exactly n−1 exchanges to sort a list of n items. One advantage of this method is that it minimizes data movement. If a key is already located in its final position it will never be moved. Every time an exchange is made at least one of the keys is placed into its final position. Note this is independent of the original ordering of the data. For example, if the data is already in sorted order then selection sort will still perform n−1 exchanges, exchanging each of the first n−1 elements with itself. Therefore one would expect the performance of selection sort to differ very little from its worst-case time to its best-case time. From the program above it is easy to analyze the number of key comparisons and the number of exchanges required. Clearly, n−1 exchanges are required. The number of comparisons is determined as follows: during the first pass when i=1, n−1 comparisons are made. During the second pass when i=2, n−2 comparisons are made, and so forth until pass n−1 when one comparison is made. Hence the total number of comparisons is given by: (n−1) + (n−2) + (n−3) + ... + 1 = 1/2 n(n−1). Therefore straight selection sort is an $O(n^2)$ algorithm.

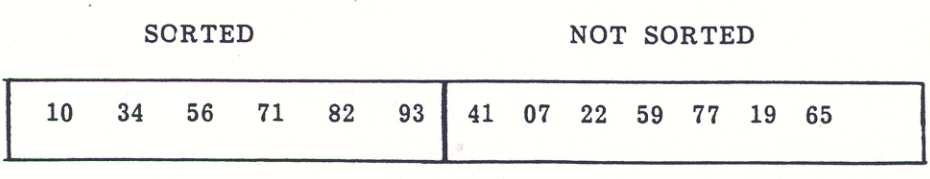
## Straight Insertion Sort

Straight insertion sort assumes that there is a collection of items in an array that are in sorted order. The next entry is then inserted into its proper place in the sorted list. This is exactly the same method most card players use to sort their cards. The straight insertion sort algorithm to sort the array A[1..n] in increasing order is given as follows: By definition subarray A[1..1] is in sorted order. Next element A[2] is inserted into its proper place and now subarray A[1..2] is in sorted order. Next element A[3] is inserted into its proper place and A[1..3] is now in sorted order. This proceeds until element A[n] is inserted into its proper place in subarray A[1..n−1] generating the final results. The algorithm expressed in pseudo code is given below:

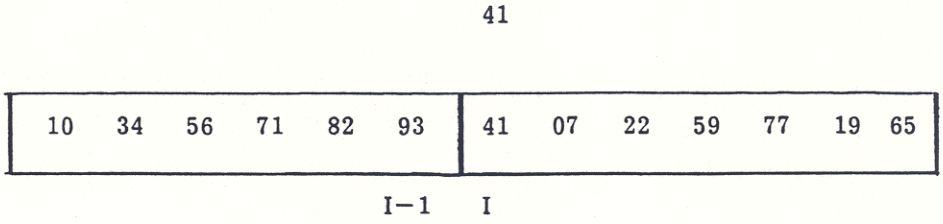
```
for I := 2 to N do
  /* Invariant: A[1..I−1] is in sorted order */
  /* Sift A[I] down to its proper place in A[1..I] */
```

To further illustrate the above pseudo code the following diagrams indicate what happens during a particular step, in this case when I = 7:

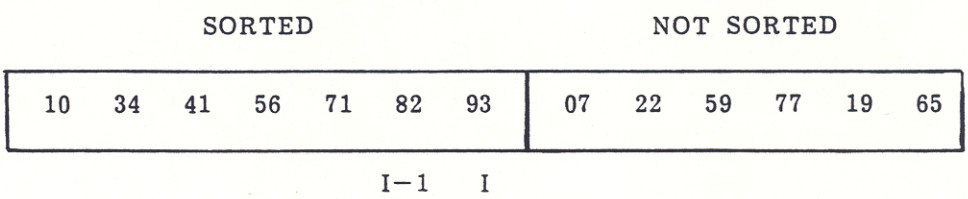
(1) The initial condition (invariant): A[1..I−1] is sorted.



(2) Sift A[I] down to its proper place in the sorted subarray.




(3) At the end of the step: A[1..I] is in sorted order




The program for straight insertion sort is given below:

```
/* Sort A[1..N] using Straight Insertion Sort */

for I := 2 to N do
  begin
  /* Invariant: A[1..I-1] is sorted */
  J := I;
  T := A[J];
    While J > 1 and A[J-1] > T do
    begin
    A[J] := A[J-1];
    J := J - 1;
    end
  A[J] := T;
  end
```

The program first moves A[I] (the next element to be inserted) into T. Then the while loop shifts elements one position to the right into the slot vacated by A[I]. Finally T is placed into its final position.

*Analysis of Straight Insertion Sort*

If the array is initially in sorted order then each element would require one comparison and no insertions (shifting of elements). The total time would be $O(n)$. If the initial array is in reverse sorted order then insertion of the $k$-th item would require k−1 comparisons and one insertion always at the beginning of the array. The total number of comparisons in this case is 1 + 2 + 3 + ... + n−1 plus the same number of shift downs plus n−1 insertions. This represents $n^2$ − 1 total operations and a running time of $C(n^2)$. Insertion sort works best for a sorted array, $O(n)$, and worst for an array in reverse sorted order, $O(n^2)$. What about the "average" case for an array with a random arrangements of values? In this case, we can make a crude assumption that to insert the $k$-th item will require (on the average) (k−1)/2 comparisons and as many shift downs plus one inser-

tion. That is, we assume the $k$-th item will be inserted on the average half way down the sorted subarray. This represents $(n^2 + n)/2 - 1$ operations which is still $O(n^2)$ for the average case. Therefore straight insertion sort is an $O(n^2)$ algorithm.

### Binary Insertion Sort

The straight insertion algorithm is easily improved by noting that at each step A[I] is inserted into a sorted subarray A[1..I−1]. Since the subarray is sorted, using a binary search algorithm will be much faster than using the sift-down approach to locate the insertion point. The binary search algorithm samples the middle of the sorted subarray and continues bisecting until the insertion point is found. This modification to straight insertion sort is called binary insertion sort and the program for this algorithm is given below.

```
/* Sort A[1..N] using Binary Insertion Sort */
for I := 2 to N do
  begin
  /* Invariant: A[1..I-1] is sorted */
  L := 1; R := I-1;
  T := A[I];
    /* binary search for the insert */
    /* position of T within A[1..I-1] */
    While L <= R do
    begin
    M := (L+R) DIV 2;
    if T < A[M] then R := M-1
                else L := M+1;
    end;
  /* sift down to position L */
  for J := I-1 downto L do A[J+1] := A[J];
  /* insert into proper position */
  A[L] := T;
  end;
```

*Analysis of Binary Insertion Sort*

If the insertion position is at the low end of the sorted subarray then the binary search algorithm will locate the insertion position in fewer comparisons than straight insertion sort. If, however, the insertion position is near the high end of the sorted subarray then the binary insertion sort is slower. Therefore, this technique favors those cases where the keys are highly out of order. If the initial array of keys are in sorted order or nearly sorted, straight insertion sort requires only one comparison per step and is much faster than binary insertion sort. Thus far we have compared these algorithms considering only the number of comparisons. It is important to remember that the only difference between these two algorithms is the method for locating the insertion position. Both algorithms require the same number of moves to occur once the insertion position is located. Generally, moving an item is more time consuming than comparing two keys. Thus what looked like an obvious improvement to straight insertion sort turns out to be very little improvement in the average case and in some instances may even be

worse. Binary insertion sort is also an $O(n^2)$ algorithm. More details on this algorithm along with a more rigorous analysis can be found in Ref. 7.

### Bubble Sort (Straight Exchange)

An elementary sorting algorithm usually taught in beginning computer courses is the bubble sort. This method is sometimes called sorting by straight exchange since the principle of this algorithm is to compare and exchange pairs of adjacent items until all items are sorted. This algorithm is one of the easiest to understand and program, however, of all the sorting algorithms we will present it is probably the least efficient, even with all of the suggested improvements to the algorithm. The basic idea of this algorithm is to pass sequentially through the data several times. Each pass consists of comparing and exchanging (if necessary) each pair of adjacent items so that they are in sorted order. Notice after the first pass the largest element will find its way to the last position in the array regardless of its initial location in the array. The largest element is now in its proper position, A[n]. The second pass compares and exchanges elements 1..n−1 of the array since the last element need not be compared again. The largest element in this group will find its way to its final position, A[n−1]. In general, during the $k$-th pass element A[n−k+1] will be in its proper position. As the algorithm progresses the smaller values "bubble" toward the top. The array will be sorted after n−1 passes. Consider the following example for an array of size n = 13 and suppose the fourth pass has just been completed. That is, element A[10] has just been placed into its final position. Thus elements A[10..13] are in sorted order and elements A[1..9] are unordered:

At the beginning of pass i, the loop invariant states that elements A[n-i+2..n] are in sorted order and elements A[1..n−i+1] are unordered as shown below: (i = 5 in this case).

| NOT SORTED | | | | | | | | | SORTED | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | 21 | 41 | 19 | 61 | 55 | 27 | 49 | 12 | 69 | 76 | 91 | 98 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

During this pass, first 32 and 21 are compared and exchanged. Now 32 and 41 are compared and not exchanged since they are in order. Next 41 and 19 are exchanged. Next 41 and 61 are compared and not exchanged. The next step compares and exchanges 61 and 55. Since 61 is the largest value in this subarray we see that the remaining steps in this pass will compare and exchange 61 successively with 27, 49, and finally 12 yielding the following results:

| NOT SORTED | | | | | | | | SORTED | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 32 | 19 | 41 | 55 | 27 | 49 | 12 | 61 | 69 | 76 | 91 | 98 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

At the end of pass i, element A[n−i+1] will be in its proper position and elements A[n−i+1..n] are in sorted order while elements A[1..n−i] are unordered.

The program for bubble sort is given below:

```
/* Sort A[1..N] using Bubble Sort */
for PASS := 1 to N-1 do
  /* this loop controls the number of passes */
  begin
  /* Invariant: elements A[N-PASS+2..N] are sorted */
  /*            elements A[1..N-PASS+1] are unordered */
  for J := 1 to N-PASS do
    /* this loop compares adjacent pairs */
    if A[J] > A[J+1] then /* exchange */
      begin
      TEMP := A[J];
      A[J] := A[J+1];
      A[J+1] := TEMP;
      end
  end;
```

*Analysis of Bubble Sort*

The analysis of this algorithm is fairly simple. During the first pass there are n−1 comparisons. During the second pass there are n−2 comparisons and, in general, during the $k$-th pass there are n−k comparisons. The total number of comparisons is (n−1) + (n−2) + (n−3) + ... +1 = n(n−1)/2. Therefore bubble sort is an O(n**2) algorithm. The total number of exchanges required depends on the original distribution of values. If the array is initially in sorted order then no exchanges are required. Note that the number of exchanges will never exceed the number of comparison.

## Bubble Sort with Flag

This algorithm can be improved in several ways. One improvement is to remember (set a flag) whether any exchanges were made during a pass. If no exchanges occurred during a pass then the array is sorted and further passes are not needed. Bubble sort on a sorted array will go through every pass taking a total of $O(n^2)$ operations. However, with this improvement only one pass with $O(n)$ operations is required.

The program for bubble sort with flag is given below:

```
/* SORT A[1..N] using Bubble Sort with Flag */
PASS := 1;
SORTED := false;
while (PASS <= N-1) and not SORTED do
  /* this loop controls the number of passes */
  begin
  /* Invariant: elements A[N-PASS+2..N] are sorted */
  /*            elements A[1..N-PASS+1] are unordered */
  SORTED := true; /* initially no exchanges */
  for J := 1 to N-PASS do
    /* this loop compares adjacent pairs */
```

```
    if A[J] > A[J+1] then /* exchange */
      begin
      SORTED := false
      TEMP := A[J];
      A[J] := A[J+1];
      A[J+1] := TEMP;
      end;
  PASS := PASS + 1;
  end;
```

*Analysis of Bubble Sort with Flag*

The number of comparisons during pass i is n−i. Suppose k passes are required, then the total number of comparisons is given by (n−1) + (n−2) + (n−3) + ... + (n−k) = $(2kn-k^2-k)/2$. It can be shown that the number of required iterations, k, is a function on n, hence k is $O(n)$. Thus the total number of comparisons is still $O(n^2)$, although the constant factor is smaller than the original bubble sort. There are other improvements that can be made to bubble sort. For example, it is important not only to remember whether an exchange took place during a given pass, but to note the position of the first exchange. Suppose this occurred at index k. Subsequent passes can therefore begin at index k rather than the first position [7]. We observe that the number of passes required to sort the array is the maximum distance any one number needs to move from its initial position toward the beginning of the array to its final position. This is because a value can only move toward the beginning of the array one position per pass [8]. A value can move several positions, however, toward the bottom of the array during one pass. For example, consider the following array of values:

99 23 29 45 50 61 72 10

It will take seven passes to move 10 to the first position where it belongs. Note 99 moves to the bottom of the array to its final position during the first pass. An obvious improvement is to have successive passes go in opposite directions to reduce the number of passes. This method is called the Shakersort. For further analysis of Shakersort including program code see [7].

Thus far we have considered several straight sorting algorithms: selection, insertion, and exchange (bubble). All straight sorting methods move each item one position at each step. Note that the loop invariants for selection and insertion sorts are the same, namely A[1..i−1] is in sorted order at the beginning of step i and A[1..i] is in sorted order at the end of step i. Bubble sort has a similar invariant, only the sorted subarray is growing bottom to top by one item each step while the other two algorithms grow top to bottom by one item each step. All of these algorithms have an $O(n^2)$ running time in both the worst case and average case. Insertion and exchange (bubble with flag) sorts have a best running time of $O(n)$ for sorted files. Selection sort always has an $O(n^2)$ running time even for a sorted file. Thus, for large n (about 100) none of these algorithms compare favorably with the $O(n \log n)$ algorithms that will be discussed next. It can be shown that the average distance that an item will travel during a straight sorting technique is about n/3 [7]. Therefore any significant improvement to straight sorting algorithms will be based on the principle of

moving items greater distances during each step. Next we will discuss three algorithms; each one is an improvement to one of the straight sorting algorithms we have already discussed. Shellsort is an insertion sort, Quicksort is an exchange sort, and Heapsort is a selection sort. Each of these algorithms are capable of moving items great distances during each step.

## Shellsort

Shell sort was developed by Donald L. Shell [*9*] in 1959. This method is sometimes called diminishing increment sort. Shellsort is a simple extension of straight insertion sort which allows exchanging of elements that are far apart. This is done by sorting separate subfiles of the original file. In general, these subfiles contain every *k*-th element of the original file. Suppose the value of k is 5 then this defines five subfiles of the original file as follows: Subarray 1 consists of the elements A[1], A[6], A[11], etc. Subarray 2 consists of elements A[2], A[7], A[12], etc. Subarray 3 consists of elements A[3], A[8], A[13], etc. Subarray 4 contains elements A[4], A[9], A[14], etc. and subarray 5 contains the remaining elements of the original array: A[5], A[10], A[15], etc. In general, the *i*-th element of the *j*-th subarray with increment k is A[(i−1) × k + j]. During the first pass the k subarrays are each sorted (usually by straight insertion sort). Pass two involves choosing a smaller increment size for k, defining k new subfiled in the same manner and sorting these subfiles. Any number of passes may be performed as long as k is diminished at each new pass and the last pass has k = 1.

For example if the original array of values is:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 41 | 76 | 19 | 56 | 91 | 87 | 60 | 12 | 73 | 32 | 39 | 11 |

with diminishing increment k values as (5, 3, 1) then the following array indexes are sorted separately during pass one: (1, 6, 11), (2, 7, 12), (3, 8), (4, 9) and (5, 10). The results after pass one are shown below.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 39 | 11 | 12 | 56 | 32 | 41 | 60 | 19 | 73 | 91 | 87 | 76 |

During pass two with k = 3 the following array indexes are sorted separately: (1, 4, 7, 10), (2, 5, 8, 11), and (3, 6, 9, 12). The results are shown below.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 39 | 11 | 12 | 56 | 19 | 41 | 60 | 32 | 73 | 91 | 87 | 76 |

It turns out that only one subarray changes since the other two happened to be sorted. Finally when k = 1, the entire array is sorted using straight insertion sort.

### *Analysis of Shellsort*

The efficiency of Shellsort is mathematically involved. The running time is difficult to determine because it depends on the file size, distribution of keys, and the sequence of increments used. The running time is approximately $O(n(\log n)^2)$. For some sequences, the running time has been shown to be $O(n^{1.5})$. Empirical studies indicate the running time is of the form $a \times n^b$ where a is between 1.1 and 1.7 and b is approximately 1.25 [5]. It is certainly better than $O(n^2)$ but not as good as $O(n \log n)$.

We have already observed that straight insertion sort is very efficient for sorted or nearly sorted files. Early passes of shellsort through the array tend to move elements close to their final positions, so that the final insertion sort of all of the elements is more efficient.

The choice of diminishing increments to use has been studied extensively. Any sequence that ends with 1 will always work, however, it has been determined that increments that are relatively prime (have no common factor other than 1) work quite well. This guarantees that the successive passes intermingle subarrays so that the entire array is nearly sorted by the time of the last pass. Some suggested sequences include (9, 5, 3, 1), (15, 7, 3, 1), (11, 5, 3, 1), and even longer sequences for larger files. Knuth [5] gives an excellent and exhaustive treatment on this topic. He recommends choosing the increments by defining a function h recursively as follows: h(1) = 1 and h(i+1) = 3×h(i)+1. Let x be the smallest integer such that h(x) > n and set the number of increments to be x−2. For example, if n = 25,000 then x = 10 and the suggested sequence is (3280, 1093, 364, 121, 40, 13, 4, 1). The program for shellsort is fairly simple to implement. The reader is referred to any data structures textbook [*1, 6, 8, 10*] for a shellsort program.

## How Fast Is It Possible to Sort?

What is the best computing time that we could hope for? This can be answered quite easily if we restrict ourselves to sorting methods that are entirely based on comparisons between keys. The sorting algorithms we have discussed thus far fall into this category. Consider the following decision (comparison) tree shown in Figure 1 to sort three items a, b, c. Each node of the tree (circle) represents a comparison between a pair of keys. Each branch represents the outcome of each comparison. By convention, we will use the left branch to represent True and the right branch to represent False. A leaf node (rectangle) represents the possible sorted sequences.

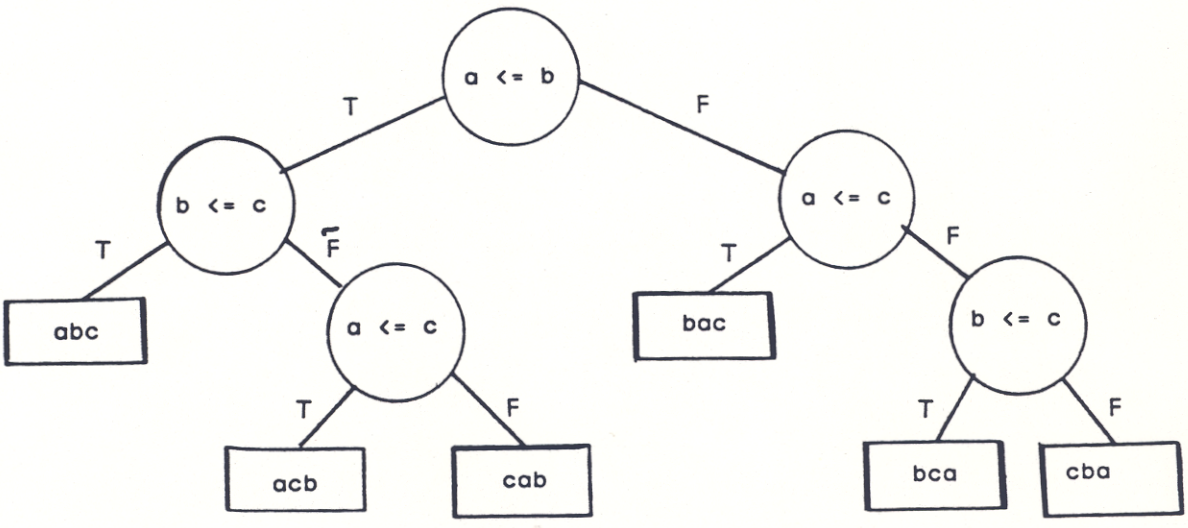



FIGURE 1

Each path through the tree represents a possible sequence of comparisons that an algorithm could take in order to arrive at a solution. There are 3! = 6 permutations of a, b, c thus there are 6 leaf nodes and 6 different paths in the tree. One comparison is made per level of the tree, thus we can determine the number of comparisons required by determining the height of the tree. The question arises: what is the height of a decision tree that sorts n distinct elements? The height of the tree must be at least log(n!) + 1. Proof: If n elements are to be sorted there must be n! different possible results. Thus any decision tree must have n! leaves. A decision tree is also a binary tree, and any binary tree can have at most $2^{k-1}$ leaves if its height is k. Thus the height must be at least log(n!) + 1 [*30*]. The expression log(n!) is approximately n + 1/2) × (logn − 3/2) + 2 or simply nlog $n + O(n)$. Therefore the lower bound running time for a sorting algorithm based on comparisons of items is given by $O(n\log n)$. Next we will present several sorting algorithms that have $O(n\log n)$ running times.

## Quicksort

The Quicksort algorithm developed by C. A. R. Hoare [*11, 12*] is one of the most efficient and one of the fastest internal sorting algorithms. This algorithm is sometimes called partition exchange sort. Quicksort sorts a file of keys of size n recursively by choosing a key, p, in the file (called the pivot key) around which to rearrange the other keys in the file. Ideally, the pivot key is near the median key value so that it is preceded by about half of the keys and followed by the other half. The keys are now rearranged such that for some position k in the array, A[1], A[2],..., A[k] contain keys less than or equal to p and A[k+1], A[k+2],..., A[n] contain the keys greater than or equal to p. The elements A[1...k] are called the left subfile and elements A[k+1...n] are called the right subfile. Thus depending on some value, p, the original file is partitioned into two subfiles where none of the keys in the left subfile are greater than any of the keys in the right subfile. Clearly, if the left and right subfiles are now sorted separately, the original file will be sorted. Thus the original problem of sorting A[1..n] is now reduced to two subproblems of sorting A[1..k] and A[k+1..n] independently. The Quicksort algorithm based on a recursive approach is given below:

```
Procedure Quicksort (L,R:integer);
/* Sort array elements A[L..R] in global A */
VAR k:integer; /* index of the partition */
    p:integer; /* pivot value */
if L < R then
  BEGIN
  FINDPIVOT(L,R,p);
  PARTITION(L,R,k);
  Quicksort(L,k-1);
  Quicksort(k,R);
  END;
```

Consider the following file of n = 13 keys with the middle key, 40, as the pivot key controlling the partitioning process. The process starts with two pointers, i and j initially pointing to the first and last elements in the file:

| 34 | 47 | 23 | 12 | 77 | 65 | 40 | 97 | 18 | 71 | 31 | 81 | 74 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i | | | | | | | | | | | | j |

We move i right until a value greater than or equal to the pivot is found. In this case i moves to 47. Now j moves left until a value less than or equal to the pivot is found. In this case j moves to 31. This yields the following:

| 34 | 47 | 23 | 12 | 77 | 65 | 40 | 97 | 18 | 71 | 31 | 81 | 74 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | i | | | | | | | | | j | | |

We now have the case where 47 belongs in the right subfile and 31 belongs in the left subfile. Thus the values are exchanged, hence the name partition exchange sort. The invariant states that elements A[1..i] are all less than or equal to the pivot, p and elements A[j..n] are all greater than or equal to p as shown below. The elements A[i+1..j−1] are yet to be determined.

| <= p | | | | | | | | | >= p | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 31 | 23 | 12 | 77 | 65 | 40 | 97 | 18 | 71 | 47 | 81 | 74 |
| | i | | | | | | | | | j | | |

We now move i to the right again until it rests on a value larger than or equal to the pivot; the value in this case is 77. We move j left again until it rests on a value less than or equal to the pivot, in this case 18. The values are exchanged yielding:

| <= p | | | | | >= p | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 31 | 23 | 12 | 18 | 65 | 40 | 97 | 77 | 71 | 47 | 81 | 74 |
| | | | | i | | | | j | | | | |

Next we move i right to 65 and move j left to 40. The values are exchanged yielding:

| <= p | | | | | | >= p | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 31 | 23 | 12 | 18 | 40 | 65 | 97 | 77 | 71 | 47 | 81 | 74 |
| | | | | | i | j | | | | | | |

Finally, move i right to 65 and move j left to 40 as shown below:

| <= p | | | | | | >= p | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 31 | 23 | 12 | 18 | 40 | 65 | 97 | 77 | 71 | 47 | 81 | 74 |
| | | | | | j | i | | | | | | |

The partitioning process terminates when i > j. We have two subfiles as shown above where no element in the left subfile is larger than any element in the right subfile. In other words, every element in the left subfile is less than every element in the right subfile except that values equal to the pivot may appear anywhere in either file. Each subfile is sorted independently.

The program for Quicksort using recursion is given below. The reader is referred to [7] for a program listing of a nonrecursive version of quicksort.

```
Procedure Quicksort(L,R:integer);
  var i,j: integer;
      p,t: integer;
  begin
    i := L; j := R;
    p := A[(L+R) div 2];
    /* p is the pivot--the middle value */
    repeat
      while A[i] < p do i := i+1;
      while A[j] > p do j := j-1;
      if i <= j then
        begin /* exchange A[i] and A[j] */
        t := A[i]; A[i] := A[j]; A[j] := t;
        i := i+1; j := j-1;
        end
    until i > j;
    if L<j then quicksort(L,j);
    if i<R then quicksort(i,R);
  end;
```

Looking at the details of the program given above the invariant can be stated in more detail as follows: the result of this partitioning scheme is such that keys in A[1..i−1] are all less than or equal to the pivot value; keys A[j+1..n] are all greated than or equal to the pivot value and therefore elements A[j+1..i−1] (if any) are all equal to the pivot value.

*Analysis of Quicksort*

Consider the following list of values to sort using the above algorithm. We present a trace for each successive partition and the indicated pivot value, p.

```
[2 4 6 7 1 5 3]          original file (p = 7)
[2 4 6 3 1 5] [7]        after first partition.    (Next p = 6)
[2 4 5 3 1] [6][7]       after second partition.   (Next p = 5)
[2 4 1 3] [5][6][7]      after third partition.    (Next p = 4)
[2 3 1] [4][5][6][7]     after fourth partition.   (Next p = 3)
[2 1] [3][4][5][6][7]    after fifth partition.    (Next p = 2)
[1][2][3][4][5][6][7]    after final partition.
```

This file was specially constructed such that in each partition the pivot value turned out to be the largest element in the file. This represents the worst possible case where at each partition step a file of size n is divided into two files of size 1 and n−1. The cost of each partition is $O(n)$ comparisons and in the worst case n−1 partition steps are required. Thus quicksort has a worst case running time of $O(n^2)$. In general, the worst case running time results whenever the pivot value is the smallest or largest value at every partition step.

The best case running time occurs when the partitioning scheme splits the file into two files of as equal length as possible every time. Initially

the entire file of size n is partitioned at a cost of $O(n)$ into 2 files of size n/2 each. Next the two files are partitioned into four files of size n/4 each at a cost of 2 × $O(n/2)$ or $O(n)$ also. Next 8 files of size n/8 each is generated at a total cost of $O(n)$. In general 2**k files are generated each of size $n/(2^k)$ at a cost of $O(n)$. This continues until each file is size one. Thus the total cost is $O(n)$ comparisons for each of the log n steps for a total running time of $O(n\log n)$. It can be shown that the average running time for quicksort is also O(n log n).

Quicksort has been studied extensively, probably more than any other single sorting algorithm. The improvements to quicksort have been in four general areas: (1) various schemes used to partition the file, (2) methods for determining a better pivot value, (3) algorithms that consider the size of the subfiles, and (4) algorithms that check for sorted subfiles. We will briefly review each of these areas. For a more extensive review of the enhancements to quicksort see Wainwright [*13*]. There have been many partitioning methods developed for quicksort. One scheme uses a method in which the pivot value is involved in every exchange [*14*]. Another method due to Lomoto described in [*15*] has both indices begin at the left end of the file and move toward the right end. The most commonly used method is due to Sedgewick and is based on two approaching indices. The scheme presented here is based on this technique. There seems to be no end to the variety of partitioning schemes found in the literature. Quicksort is the type of algorithm where simple modifications can be easily made in a variety of places in the algorithm to slightly improve the performance. Thus one rarely finds quicksort algorithms presented in exactly the same way in the literature. Most are slight variations of each other with little difference in performance. Some modifications, however, greatly improve the performance of quicksort.

There have been several techniques developed over the years for determining a better pivot value. Remember the goal is to find some value that will divide the file into two subfiles of as equal length as possible. Hoare originally suggested using a key chosen at random from the file. Many people find it convenient simply to use the first key in the file. Notice in this case that if the file is already sorted then quicksort sorts the file in worst case time of $O(n^2)$ since the pivot value in every case is the smallest value in the file. This also happens for files sorted in reverse; the pivot value is the largest value in the file every time. It is ironic that quicksort, which is considered the best internal sorting algorithm, has a worst case condition for sorted or nearly sorted files! One solution is to use the middle value as we did in the above algorithm. A quicksort algorithm which uses the middle key as the pivot is known as quickersort. Another suggestion is the median-of-three method. Here the left, middle, and right keys are sorted and placed back into the same positions only in sorted order. The middle (median-of-three) value is used as the pivot. The array elements A[2..n−1] need only be considered since A[1] and A[n] are known to be in their proper subfile. Furthermore, this method guarantees that after splitting a file of size n the longest possible subfile is n−2 rather than n−1. In Meansort [*16*], the mean value of the file is calculated and used as the pivot value. This is one of the few variations that does not use one of the key values as the pivot. For most distributions of keys this method is an excellent estimate of the median key value and partitions tend to be divided into nearly equal sizes, which is the desired goal. However, the extra calculations needed to determine the mean each time need to be considered when evaluating this method.

Several improvements have been made to quicksort by considering the size of the subfiles. First, if a subfile is size two, then it is best to make one comparison and perhaps one exchange rather than recursively calling quicksort again. Several investigators have noticed that quicksort is not very efficient for small subfiles. This is unfortunate since the nature of quicksort guarantees many small subfiles. Hoare [*12*] was the first to suggest smaller subfiles be sorted by some other method. He suggested insertion sort. Knuth [*5*] suggests nine as the size of the subfile in which quicksort should invoke a simpler sorting algorithm. In other words, subfiles of size m or less are ignored and not partitioned further. When quicksort finishes, the file is not in sorted order, of course. Instead, the file contains small groups of randomly ordered values such that elements in one group will all be less then elements in any group to the right. Thus we simply sort the entire array by insertion sort. Suggested values for m range from 8 to 15 depending on the hardware used and the implementation language and compiler.

Another improvement to quicksort is to determine if any one of the subfiles generated by the partitioning process is in sorted order. If so the subfile need not be partitioned further. This would allow quicksort to sort sorted or nearly sorted subfiles in $O(n)$ running time regardless of how the pivot value was chosen. Bsort [*16a*] and Qsort [*13*] are two suggested algorithms for detecting sorted subfiles. The partitioning process builds the left and right subfiles one element at a time. These algorithms simply keep track of whether the new item placed into the left subfile (which is always done at the right end) has a value greater than or equal to its left neighbor each time. A check is also made if the new value placed into the right subfile (which is always done at the left end) is always less than or equal to the right neighbor. If so, the subfile is in sorted order. Qsort has been shown to be three times faster than Bsort. We did not implement any of these improvements in the program we presented above. Most of the enhancements show only slightly improved running time but, in doing so, greatly increase the size and the complexity of the quicksort program. In this way the algorithm we presented was kept as simple and as easy to understand as possible. There are many other references to quicksort that we have not cited [*8, 17—22*].

## Heapsort

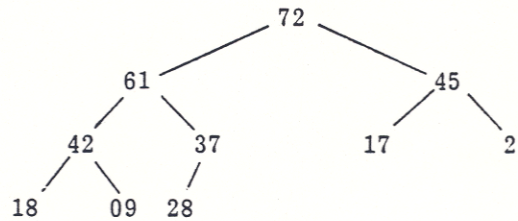
A heap is a data structure for representing a collection of items. A heap is defined as a binary tree where each node in the tree contains a key and

1. All of the leaves of the tree are either on the same level or are on two adjacent levels.
2. All levels are completely filled with nodes except possibly the lowest level and all of the leaves on the lowest level occur to the left.
3. The key of the root is greater than or equal to the key values of its children. Furthermore, the left and right subtrees (if they exist) are heaps.

The first two properties define the "heap shape" as shown in Figure 2. This shape ensures that the tree can be represented in contiguous memory such as an array. There can be no "holes" or missing nodes in the tree.

FIGURE 2

The third property defines the "heap order." The root node of the heap contains the largest value. This is called a max heap. A heap can be defined such that the key value of the root is less than or equal to the key values of its children. This is called a min heap. Our discussion will be concerned with max heaps only. Furthermore, for a max heap, the third property guarantees that each path in the heap from any node to a leaf defines a nonincreasing sequence of values. For this reason a heap is also called a "partially ordered" tree. Note an area of memory that is used for dynamic allocation is also called a "heap." This is a different use of the word "heap" and should not be confused with the heap we have defined here. The following binary tree is a max heap.




Because of the contiguous memory requirement of a heap, all heaps can be represented by an array where the values in the array correspond to the keys in the heap arranged in top-down, left-to-right order by levels. The following ten-element array represents the above heap.

| 72 | 61 | 45 | 42 | 37 | 17 | 23 | 18 | 09 | 28 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Any binary tree that has the heap shape can be sorted in a contiguous array as described above. An array A[1..n] representing such a binary tree has the following properties:

1. The left and right children of the node represented by index i in the array are located in positions 2×i and 2×i+1 in the array, respectively. If a position is beyond the bounds of the array (i.e., larger than n) then the child node does not exist.
2. The parent of the node represented by index i in the array is located in position i div 2. If this results in zero then the node is the root and has no parent.

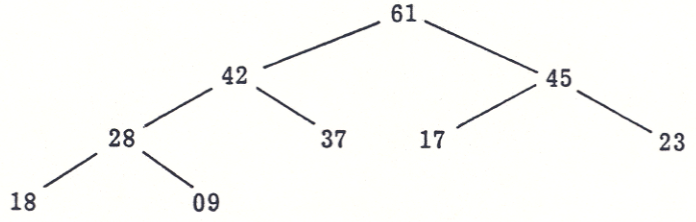
For example, for i = 5 (node 37) the parent is in A[2], the left child is located in A[10] and it has no right child.

### *Sorting Using a Heap*

Suppose we are given an array A[1..n] representing a heap. How can we use this heap to sort the values in the array? Clearly, the root of the heap (the first element in the array) is the largest value. This could be removed from the heap and set aside somewhere. Now we have a heap without a root. In order to maintain the heap shape and preserve all of the remaining values in the heap the logical solution is to remove the rightmost leaf node on the lowest level (the last element in the array) and place it into the root position. Now the heap has one less value in it and the heap shape is preserved. However the heap order may be in violation at the root position. This can be corrected by a process called sift-down as described below: Consider the above example. Key 72 is removed and set aside. Key 28 is moved into the root position. Now the sift-down process begins. The parent key 28 is compared against the larger of its children, 61 in this case. This is in violation of the heap order and clearly 61 should be the new root so 61 and 28 are exchanged. The contents of the array to this point is shown below:

| 61 | 28 | 45 | 42 | 37 | 17 | 23 | 18 | 09 | -- |
|---|---|---|---|---|---|---|---|---|---|

This process is repeated again with 28 until it is sifted down the tree to its proper heap position. That is, 28 is next compared against its children, 42 and 37. Key 42 is the larger child and greater than 28 so keys 28 and 42 are exchanged. Now key 28 is compared to its children, 18 and 09, and since this is in the proper heap order the sift-down procedure is finished. Notice that the last element of the array is not being used. This is the logical location for key 72 which we had earlier set aside. In fact we can use the nonheap portion of the array to begin to build the sorted list. The resulting heap and associated array after exchanging 72 and 28, then sifting 28 down to its proper position is given below:

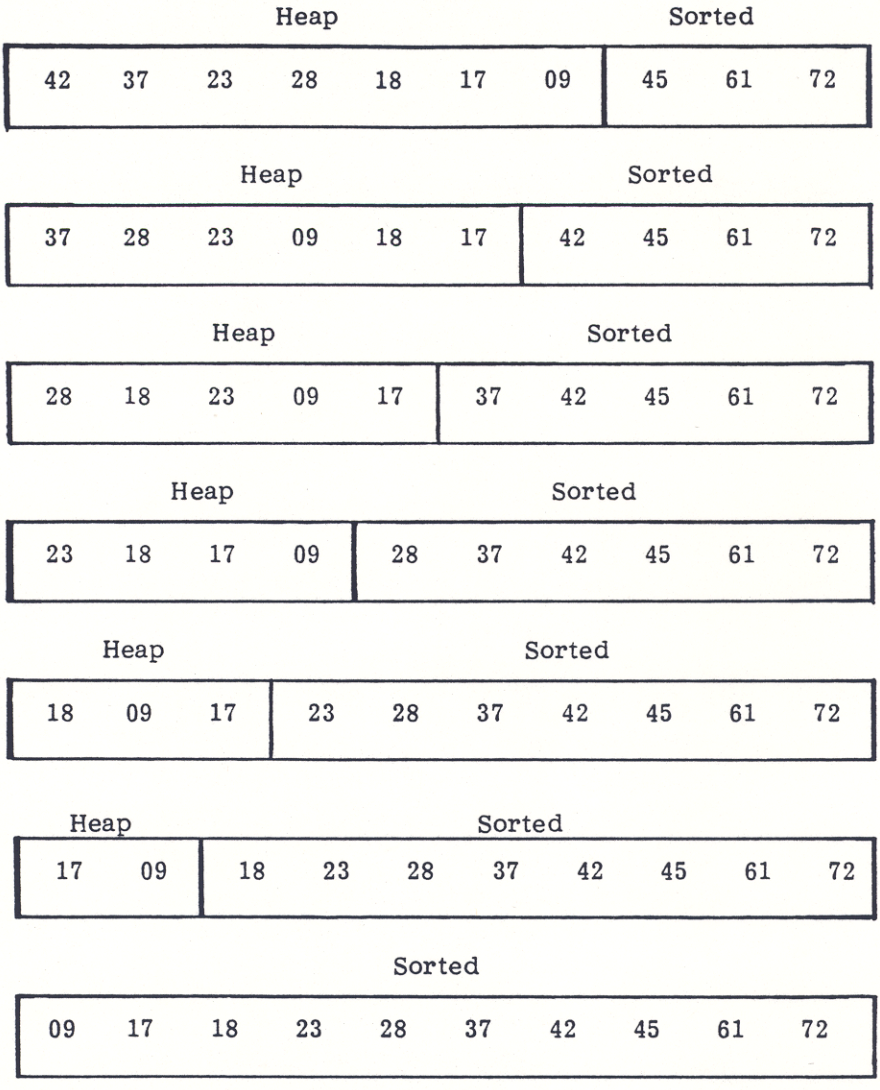



| Heap | | | | | | | | | Sorted |
|---|---|---|---|---|---|---|---|---|---|
| 61 | 42 | 45 | 28 | 37 | 17 | 23 | 18 | 09 | 71 |

This process is repeated. That is, keys 09 and 61 are exchanged and key 09 is sifted down the heap. The results are shown below:

| Heap | | | | | | | | Sorted | |
|---|---|---|---|---|---|---|---|---|---|
| 45 | 42 | 23 | 28 | 37 | 17 | 09 | 18 | 61 | 72 |

The following sequence of arrays shows the trace of this process as it is applied to the rest of the heap. Each time the last element and the root are exchanged, and the sift down procedure is performed.




The Sift-down procedure is given below:

```
Procedure Siftdown (N);
var C,P,T : integer;
  DONE : boolean;
  /* P is the parent node */
  /* C and C+1 are the children of P */
  /* DONE tells when the sifting is finished */
  /* Invariant: A[2..N] is a heap, A[1] is not */
  /* Post condition (after the procedure is finished): */
  /* A[1..N] is a heap */
begin
  P := 1; DONE := false;
  while not DONE do
```

```
    begin
    C := 2 * P; /* left child */
    if C > N then DONE := true
      else
      begin
      if C + 1 <= N then
        if A[C+1] > A[C] then C := C + 1;
      /* C is the position of the larger child */
      if A[P] >= A[C] then DONE := true
        else /* swap parent and larger child */
        begin T := A[P]; A[P] := A[C]; A[C] := T;
        P := C; /* update parent index */
        end /* of swap */
      end /* of comparison */
    end /* of while not DONE */
  end /* of siftdown */
```

Thus if we are given an array, A[1..n], representing a heap one can sort the elements by applying the following process:

```
for I := N downto 2 do
begin
Swap (A[1] with A[I])
Siftdown(I-1)
end
```

This is the second phase of the heapsort algorithm. The first phase is to take an array of unordered keys which represents a contiguous binary tree and construct a heap. A heap can be constructed as follows. A single node tree is by definition a heap. Now the problem is how to add additional nodes to the heap. That is, given a heap with n nodes, how can an additional node be "added" to the heap? In order to maintain the heap shape a new node can be added in only one place: as the next leaf on the lowest level of the heap. This corresponds to position A[n+1] in the array representation of the heap. The heap shape is maintained, but the heap order may be in violation. To correct this the new leaf value needs to be compared to its parent to determine if they need to be exchanged. If not, the heap is in proper order and the process is finished. If so, the new value is repeatedly exchanged with its parent up the tree as far as it should go until its proper heap position is located. This process is called sifting up and is similar to the sift down process described earlier. The procedure for sift-up is given below:

```
Procedure Siftup (N);
var C,P,T : integer;
  /* P is the parent node */
  /* C is the child of P */
  /* This procedure adds node N to the heap */
  /* Invariant: A[1..N-1] is a heap, A[N] is not */
  /* Post condition (after the procedure is finished): */
  /* A[1..N] is a heap */
begin
```

```
    C := N; /* C is the new node */
    while C <> 1 do
      begin
      P := C div 2;
      if A[P] >= A[C] then C := 1; /* exit */
        else /* swap parent and child */
        begin T := A[P]; A[P] := A[C]; A[C] := T;
        C := P; /* update index of the new node */
        end /* of swap */
      end /* while */
  end /* Siftup */
```

We can construct a heap by applying the sift-up procedure first to A[2] constructing A[1..2] as a heap. Next the sift-up procedure is applied to A[3] constructing A[1..3] as a heap and so on until A[1..n] is a heap.

Therefore, the complete heapsort algorithm to sort an array A[1..n] of unordered key is given below:

```
for I := 2 to N do Siftup(I); /* construct a heap */
for I := N downto 2 do
  begin
  Swap A[1] with A[I]
  Siftdown(I-1)
  end
```

*Analysis of Heapsort*

The heap shape guarantees that no node is more than $\log_2 n$ distance from the root. Therefore, the sift-up routine used to build a heap has running time of at most $\log_2 n$ for each node. The sift-down routine also has at most $\log_2 n$ running time for each node in reorganizing the heap. Thus the above six-line heapsort program has running time of $((n-1)\log_2 n) + ((n-1)\log_2 n)$ comparisons at most. This argument shows that heapsort takes O(n log n) steps even in the worst case. Unlike other methods we have discussed there is no worst case input that will cause heapsort to run slower. This excellent worst case performance for heapsort is one of its most attractive qualities. Very few sorting algorithms for contiguous lists are guaranteed to run in $O(n\log n)$ time with minimal storage requirements. The worst case for heapsort is slightly poorer than the average case for quicksort and quicksort has a worst case running time of $O(n^2)$. The average case analysis for heapsort is rather complicated. Empirical studies have shown that heapsort takes about twice as long as quicksort on the average. Heapsort, however, guarantees predictable performance times and is sometimes used instead of quicksort to avoid the possibility of $O(n^2)$ degradation of performance.

The heapsort algorithm was invented by Williams [*23*]. Floyd [*24*] developed a method for constructing a heap in $O(n)$ time. This algorithm uses the sift-down routine rather than the sift-up routine to construct the heap. A description and analysis of this algorithm can be found elsewhere [*7, 14, 18*]. However this does not affect the overall $O(n\log n)$ running time of heapsort. An excellent overview of heaps and the heapsort algorithm is given by Bentley [*25*]. There are several other excellent references on heaps [*5, 6, 8, 22*].

## Merge Sort

Merging is the process of combining two or more sorted lists into one sorted list. For example, consider merging the following two sorted lists of numbers:

List 1: [12, 34, 41, 56, 89]
List 2: [15, 17, 39, 45, 67, 82, 97]

Since both lists are in sorted order, the first number in the resultant merged list is determined by comparing the first numbers in each list. In this case 12 is smaller than 15, so 12 will be the first number in the merged list. Next 15 and 34 are compared and 15 is placed onto the merged list. This process of comparing the next two values in each list and placing the smaller onto the merged list continues until one list is exhausted. The remainder of the other list is then appended onto the end of the merged list. The resultant merged list is given below:

Merged List: [12, 15, 17, 34, 39, 41, 45, 56, 67, 82, 89, 97]

The sorted lists need not be of the same length. If the size of the merged list is n then the number of comparisons required to form the merged list from two sorted lists is $O(n)$. Notice also that this method requires $O(n)$ extra memory to hold the merged list.

Suppose we want to sort an array of n values. One method is to divide the array into two equal parts of size n/2 and sort each half by some fast sorting algorithm like quicksort. Once each half is sorted they can be merged together into one sorted array as described above. Suppose, however, we decide to use merge sort to sort each half of the array rather than some other method. This defines merge sort as a recursive algorithm. The recursive procedure for merge sort is given below:

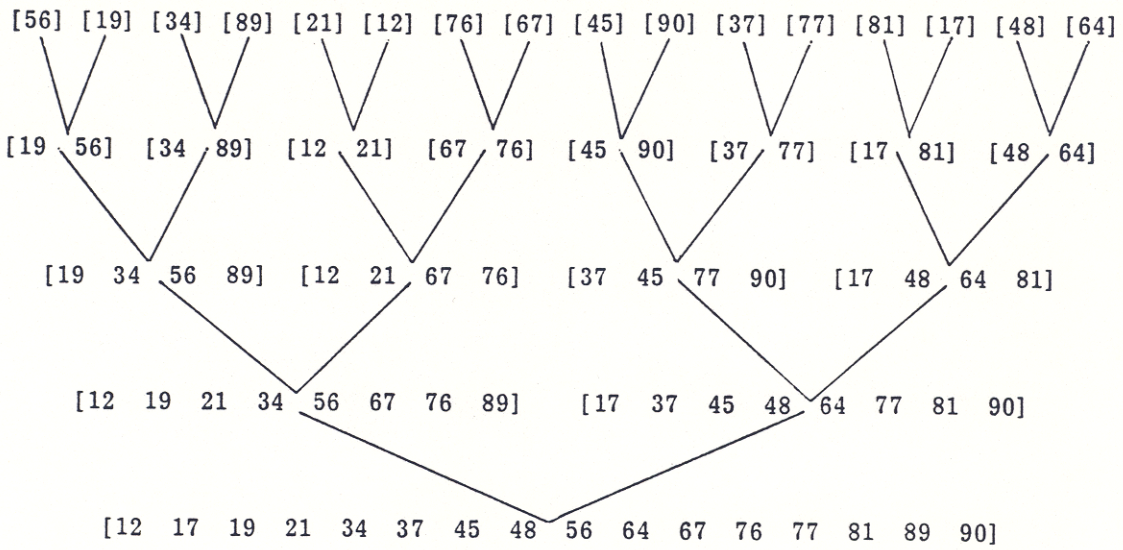
```
Procedure mergesort(A, left, right);
/* This procedure sorts elements A[left..right] */
VAR middle, size: integer;
  BEGIN
  size := right-left+1;
  /* By definition: a single value is sorted */
  if size = 1 then return;
  middle := (left + right) div 2;
  /* Recursively sort the first subarray */
  mergesort(A,left,middle);
  /* Recursively sort the second subarray */
  mergesort(A,middle+1,right);
  /* Now merge the two sorted subarrays */
  mergetwolists(A,left,middle+1,right);
  END
```

The procedure to merge two sorted lists is given below:

```
Procedure mergetwolists(A,start1,start2,end2);
/* Invariant: A[start1..start2-1] defines the first list. */
/*            A[start2..end2] defines the second list. */
/*            both lists are in sorted order */
```

```
/* Result: A[start1..end2] will be in sorted order */
VAR I,J,K:integer;
  /* I indexes over the first list */
  /* J indexes over the second list */
  /* K indexes over B which is a temporary */
  /* array used in the merging process */
  BEGIN
  /* Initialize */
  I := start1; J := start2; K := 0;
  /* Compare lists element by element */
  /* and place the smaller value into B */
  while I < start2 and J <= end2 do
    if A[I] <= A[J] then
      BEGIN
      K := K + 1;
      B[K] := A[I];
      I := I + 1;
      END
  else
      BEGIN
      K := K + 1;
      B[K] := A[J];
      J := J + 1;
      END;
  /* One of the lists has been exhausted. */
  /* Append the other list onto the end of B */
  if I >= start2
    then /* Append what is left of second list */
      while J <= end2 do
      BEGIN
      K := K + 1;
      B[K] := A[J];
      J := J + 1;
      END
    else /* Append what is left of first list */
      while I < start2 do
      BEGIN
      K := K + 1;
      B[K] := A[I];
      I := I + 1;
      END
  /* Copy the merged list in B back */
  /* into the original area in A */
  for I := 1 to K do A[start1 - 1 + I] := B[I];
  END;
```

The following diagram illustrates how the merge sort algorithm works on a list of 16 elements. The mergesort procedure recursively breaks the list of 16 elements into 16 lists of size one then uses mergetwolists as follows:




*Analysis of Merge Sort*

All of the comparisons in the merge sort algorithm take place in the merge-twolists procedure. In this procedure if the total size of both lists is n then the maximum number of comparisons required is n−1. This happens when the lists are the same size, n/2. The least number of comparisons is one whenever the size of one of the lists is one. It is clear from the above diagram that the total length of all of the lists on each level is n, which is the original size of the array to be sorted. Thus the total number of comparisons performed on each level cannot exceed n. The number of levels (excluding the first level where no comparisons occur) is $\log_2 n$. Therefore, the number of comparisons performed by merge sort on an original list of n items is no more than $n \times \log_2 n$ and hence the algorithm is $O(n \log n)$.

## Radix Sort

All of the sorting algorithms we have presented so far are based on the comparison of key values. This method is not. Instead this sort is based on the values of the actual digits in the positional representation of the keys being sorted. In other words, this method treats the keys as numbers represented in a base-R number system and works with the individual digits within the number. This is exactly the way a mechanical card sorter works. Consider the following list of 20 keys composed of three decimal digits each (R = 10). The algorithm works as follows:

Original list: [171, 805, 624, 562, 476, 409, 624, 437, 248, 003,
669, 552, 437, 743, 797, 020, 109, 217, 317, 816]

*Pass 1:* Each key in the list is examined one by one and placed into one of 10 pockets based on the value of the least significant (rightmost) digit of the key. The results are shown below:

Pockets

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 020 | 171 | 562 | 003 | 624 | 805 | 476 | 437 | 248 | 409 |
| | | 552 | 743 | 624 | | 816 | 437 | | 669 |
| | | | | | | | 797 | | 109 |
| | | | | | | | 217 | | |
| | | | | | | | 317 | | |

Next starting with the keys in the 0 digit pocket and ending with the keys in the 9-digit pocket a single list is formed as follows:

[020, 171, 562, 552, 003, 743, 624, 624, 805, 476,
816, 437, 437, 797, 217, 317, 248, 409, 669, 109]

*Pass 2:* Each key in the above list from pass 1 is examined and placed into one of 10 pockets based on the value of the middle digit of the key. The results are shown below:

Pockets

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 003 | 816 | 020 | 437 | 743 | 552 | 562 | 171 | | 797 |
| 805 | 217 | 624 | 437 | 248 | | 669 | 476 | | |
| 409 | 317 | 624 | | | | | | | |
| 109 | | | | | | | | | |

Starting with the keys in the 0 digit pocket and ending with the keys in the 9 digit pocket a single list is again formed as follows:

[003, 805, 409, 109, 816, 217, 317, 020, 624, 624,
437, 437, 743, 248, 552, 562, 669, 171, 476, 797]

*Pass 3:* Each key in the above list from pass 2 is examined and placed into one of 10 pockets based on the value of the most significant digit of the key. The results are shown below:

Pockets

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 003 | 109 | 217 | 317 | 409 | 552 | 624 | 743 | 805 | |
| 020 | 171 | 248 | | 437 | 562 | 624 | 797 | 816 | |
| | | | | 437 | | 669 | | | |
| | | | | 476 | | | | | |

Starting with the keys in the 0 digit pocket and ending with the keys in the 9 digit pocket the final sorted list is formed as follows:

[003, 020, 109, 171, 217, 248, 317, 409, 437, 437,
476, 552, 562, 624, 624, 669, 743, 797, 805, 816]

Since there are three digits in the keys, three passes are required in order to sort the original list. The radix sort algorithm to sort an array, A, of n keys expressed in pseudo code is given below. We will not present a program for radix sort; the reader is referred elsewhere [*6, 8*] for actual program code.

```
FOR k := least-significant-digit TO most-significant-digit DO
  BEGIN
    FOR i := 1 to n DO
      BEGIN
      b := A[i];
      j := kth digit of b
      place b at the rear of pocket[j].
      END
    FOR p := 0 to 9 DO
      place elements of pocket[p] into the
      next sequential positions of A.
  END
```

*Analysis of Radix Sort*

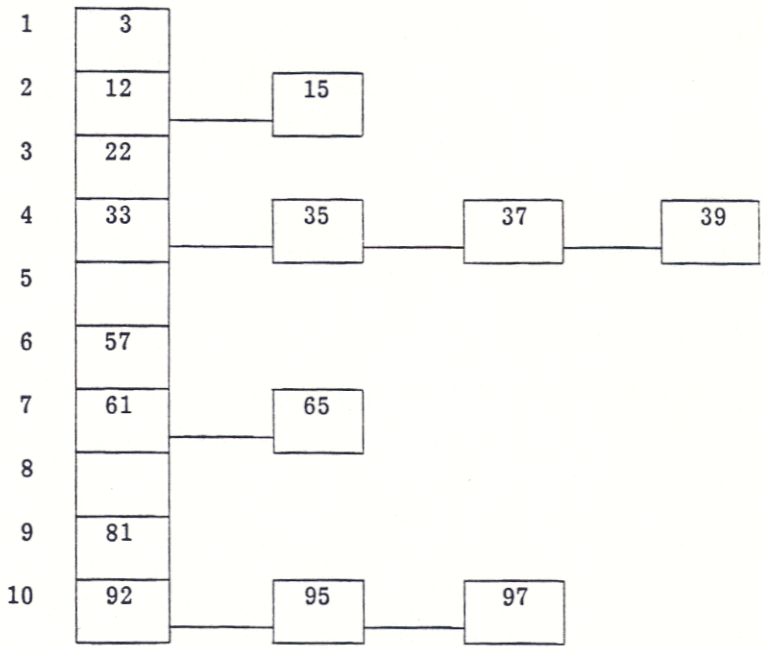
Sequential allocation of memory is not efficient when implementing the radix sort. The size of the pockets cannot be predicted during each pass. One can always allow for the worst case and allocate 10 pockets of size n each, but this is extremely wasteful and may not be possible for large n. The pockets are usually implemented with a linked FIFO queue requiring a total of $O(n)$ additional memory locations. It is possible to consider the keys as binary numbers (R = 2) and perform a radix sort on that basis. Radix sort is best suited for integer keys with small values. Real key values or negative values will pose some additional problems to take into consideration. The time requirements for the radix sort are dependent on both the size of the file, n, and the number of digits (passes), m. Clearly, during each pass $O(n)$ work is performed since each key is processed once. Thus the running time for radix sort is $O(m \times n)$. This is very efficient if the number of digits in the keys is not too large. However, for large number of digits in the keys m could be in the range of the log of n. Furthermore, if m is dependent on n and increases as n gets larger then the radix sort becomes an $O(n \log n)$ algorithm. An excellent analysis of radix sort can be found in [*1*].

## Address Calculation Sort

The address calculation sort is similar to the radix sort. Both algorithms are distributive sorts in that the keys are distributed into buckets or equivalence classes according to their values. This method depends on a hashing function to determine the bucket or address location of each of the keys. A particular kind of hashing function is required in order to sort properly. The hashing function, H, must have the property that k1 < k2 implies H(k1) <= H(k2), where k1 and k2 are any two keys. This hashing function is called a nondecreasing function. The importance of this property will be seen shortly. The hashing function distributes all of the keys into k buckets. If two or more keys are hashed into the same bucket then an ordered linked list is maintained for each bucket. Consider the following unordered list of keys:

22 81 37 15 95 61 33 3 92 12 97 65 39 57 35

Suppose we wish to define 10 buckets. Suppose the range of the keys is known to be between 1...100, then a hashing function can be defined as H(key) = ⌈key/10⌉, where ⌈ ⌉ is the ceiling function. Thus all keys in the range 1..10 are hashed into the first bucket, keys in the ranges of 11..20, 21..30, 31..40 and so on are hashed in successive buckets. The results are shown below after which it is a straightforward process to collect all of the keys in sorted order:




Instead of using linked lists to keep all of the keys within the same bucket, an alternative method is to use an array. The buckets are defined to be of size one, that is each array element is one bucket. Whenever a key falls into a bucket that is already full (collision) it is placed into an adjacent bucket. The key is moved up or dowm in the array from the "home" bucket or address such that it finds its place in the sorted order. All other keys are shifted as needed to make room for the insertion. An example is shown below. Consider the following 12 keys to be sorted:

24 14 42 33 17 6 25 4 22 18 30 15

The keys are in the range of 1..45 and suppose we have a 15 element array. We can then define a hashing function to be H(key) = ⌈key/3⌉. We process keys 24, 14, 42, 33, 17, 6, and 25 and place them into the array without collision as shown below:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 6 | | | 14 | 17 | | 24 | 25 | | 33 | | | 42 | |

Key 4 is now hashed into address 2 and collides with key 6. Since 4 < 6, key 4 is placed into address 1 of the array. Similarly, key 22 collides with key 24 at address 8 and key 22 is placed into address 7 since it is less than 24. This yields the following arrangement:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | | | 14 | 17 | 22 | 24 | 25 | | 33 | | | 42 | |

Now key 18 is hashed into address 6 and collides with key 17. Since the new key is larger than 17 it is placed in its proper sorted position to the right of address 6. Keys 22, 24, and 25 slide to the right to make room:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | | | 14 | 17 | 18 | 22 | 24 | 25 | 33 | | | 42 | |

Next key 30 is hashed into address 10 and is placed into address 11 displacing key 33 to address 12 as follows:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | | | 14 | 17 | 18 | 22 | 24 | 25 | 30 | 33 | | 42 | |

Finally, key 15 collides with key 14 at address 5. Key 15 is placed into address 6 and all of the keys to right are shifted one position to make room. The final sorted results are given below:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | | | 14 | 15 | 17 | 18 | 22 | 24 | 25 | 30 | 33 | 42 | |

*Analysis of Address Calculation Sort*

If all of the keys fall into different buckets then the cost is $O(1)$ work for each of the n keys yielding an $O(n)$ algorithm. If every key is mapped into the same bucket then we have a straight insertion sort which is $O(n^2)$. Thus the total sort time ranges from very good to very bad depending on the distribution of the keys. If one knows the probability distribution of the keys then it may be possible to design a hashing function to minimize the number of collisions. The interested reader can find more information about address calculation sort in [*5, 22, 26*]. Additional sources on hashing functions, hash tables, and collison resolution policies are listed in the references [*1, 5, 8, 14, 18, 22*].

## Summary of Internal Sorting Algorithms

The internal sorting algorithms we have discussed are summarized in Tables 1 and 2. Note the entries in the tables are approximate. The parameter m used in the radix sort denotes the number of digits in the key, N/A

TABLE 1 Comparison of Internal Sorting Algorithms (Entries are Approximate)

| Comparison parameters or classification | Algorithm | | | | |
|---|---|---|---|---|---|
| | Straight selection sort | Straight insertion sort | Straight exchange (bubble) | Shell sort | Quicksort |
| Average case running time | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n^{1.25})$ | $O(n\log n)$ |
| Best case running time | $O(n^2)$ | $O(n)$ (sorted) | $O(n)$ (sorted) | $O(n)$ (sorted) | $O(n\log n)$ |
| Worst case running time | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n^{1.5})$ | $O(n^2)$ |
| Additional storage required | None | None | None | $O(\log n)$ | $O(\log n)$ |
| Sorted in situ | Yes | Yes | Yes | Yes | Yes |
| Sort dependent on distribution of values | No | Yes | Yes | Yes | Yes |
| Key comparison sort | Yes | Yes | Yes | Yes | Yes |
| Stable algorithm | Yes | Yes | Yes | No | No |
| A straight sorting method | Yes | Yes | Yes | No | No |
| An insertion method | No | Yes | No | Yes | No |
| A selection method | Yes | No | No | No | No |
| An exchange method | No | No | Yes | No | Yes |
| A distributive method | No | No | No | No | Yes |
| Hard-split/ easy-join method | Yes | No | Yes | No | Yes |
| Easy-split/ hard-join method | No | Yes | No | Yes | No |

TABLE 2 Comparison of Internal Sorting Algorithms (Entries are Approximate)

| Comparison parameters or classification | Algorithm | | | |
|---|---|---|---|---|
| | Heap sort | Merge sort | Radix sort | Address calculation sort |
| Average case running time | $O(n\log n)$ | $O(n\log n)$ | $O(n+m)$ | $O(n)$ |
| Best case running time | $O(n\log n)$ | $O(n\log n)$ | $O(n+m)$ | $O(n)$ |
| Worst case running time | $O(n\log n)$ | $O(n\log n)$ | $O(n+m)$ | $O(n^2)$ |
| Additional storage required | None | n | $O(n)$ for pockets | $O(n)$ for links |
| Sorted in situ | Yes | No | No | No |
| Sort dependent on distribution of values | No | No | No | Yes |
| Key comparison sort | Yes | Yes | No | No |
| Stable algorithm | No | Yes | Yes | Yes |
| A straight sorting method | No | No | No | No |
| An insertion method | No | No | No | No |
| A selection method | Yes | No | No | No |
| An exchange method | No | No | No | No |
| A distributive method | No | No | Yes | Yes |
| Hard-split/ easy-join method | Yes | No | N/A | N/A |
| Easy-split/ hard-join method | No | Yes | N/A | N/A |

denotes not applicable. We have summarized each algorithm according to its best, worst, and average running times. We have also classified each algorithm according to several definitions and modes of operation. Other tables summarizing sorting algorithms are found in the literature [5, 7, 22]. In conclusion, we elaborate on some of the classifications given in Tables 1 and 2 and make several noteworthy comments.

1. The straight exchange sort reported in the tables is the bubble sort with flag. Bubble sort is still the worst method by far. Even the suggested improvement, Shakersort, does not add much to its efficiency. We include this method for the sake of completeness.
2. The improvement of binary insertion over straight insertion is questionable. Any improvement is marginal and in some instances it could lead to worse performance.
3. Sorting in situ means the values are sorted in place. Generally, if an algorithm is sorted in situ it does not require any additional storage. Shellsort and quicksort are exceptions.
4. A "straight" sorting method moves each item one position at a time. An "insertion" sort places an item into its proper sorted position within a sorted array. A "selection" sort selects the minimum (or maximum) item from the array. An "exchange" algorithm will exchange (swap) pairs of items such that they are placed in pairwise sorted order. A "distributive" sorting algorithm distributes items into subsets such that all the items in one subset are greater than all items in the other subset.
5. Enumeration sort is an algorithm we did not discuss. The idea behind this algorithm is that each item is compared with each of the others. Counting the number of smaller keys will determine its final position in the sorted array. This method is also known as "comparison counting" or "sorting by counting." As expected it has $O(n^2)$ running time and is very inefficient. The algorithm works much better in a parallel-processing environment. See Kunth [5] for more details on this algorithm.
6. A key comparison algorithm is an algorithm that sorts by comparing the keys with each other. The lowest possible running time for such an algorithm has been shown to be $O(n\log n)$. Radix sort and address calculation sort are not based on key comparisons and hence may obtain $O(n)$ running times. The lowest possible running time for an algorithm that sorts n items on a sequential machine is $O(n)$. This is because, at the very least, each item must be visited once, and thus the amount of work involved is proportional to n.
7. A sorting algorithm is called "stable" if the relative order of items with equal keys remains unchanged by the sorting process. This can be a desirable feature in a sorting routine. Perhaps records with equal keys are ordered among themselves based on some secondary item that must be preserved during sorting. Radix sort and address calculation sort can easily be made into stable algorithms by carefully ordering the equal keys that are mapped into the same pocket. Algorithms such as quicksort are more difficult to make stable. However, stable versions of quicksort do exist, see [21, 27].

8. Rather than classifying sorting algorithms based on elemental operations such as "insertion," "selection," or "exchange," Merritt [*28*] suggests a classification based on the top-down design of the algorithm. She suggests the categories of easy-split/hard-join and hard-split/easy-join. For example, if we consider quicksort from a procedural or high-level point of view, we see the algorithm spends most of its time splitting an array into two parts. Once each part is sorted it is trivial to join them back together. Thus quicksort is a hard-split/easy-join sorting algorithm. An analysis of straight selection sort, straight exchange sort and heap sort show they also fall into this category. If we consider merging two sorted arrays, the "split" has already been determined and is trivial. Most of the work involves joining the two parts. Hence from a procedural view, mergesort is an easy-split/hard-join algorithm. Straight insertion sort and shellsort also fall into this category. It can be argued that all comparison-based sorting algorithms can be classified as either easy-split/hard-join or hard-split/easy-join algorithms [*28*].
9. Quicksort is generally regarded as the best overall internal sorting algorithm considering all distributions of keys. It is generally 2 to 3 times faster than heapsort. Table 1 indicates the best case running time for quicksort to be $O(n\log n)$. This is true for the version of quicksort that we presented. However, several recent versions of quicksort have been developed in which sorted subfiles are detected and not processed further [*13, 16a*]. Hence sorted or nearly sorted files or sorted in reverse or nearly sorted in reverse files can be sorted using these algorithms in O(n) running time.
10. It is difficult to determine which sorting algorithm is the best one to use. Factors to consider are the distribution of keys, the size of the file, the range and the size of the key values, and the amount of main memory available. If the number of items is small straight selection or straight insertion sort may be appropriate. If the number of keys is large but the size of the key is small, then radix sort will probably work well. For a large number of long keys mergesort, heap sort, or quicksort should be used. If the keys are uniformly distributed over a rather small range, then address calculation sort is a good method. If the keys are known to be sorted or nearly sorted then straight insertion sort or a special version of quicksort is a good algorithm. Cook and Kim [*29*] have developed an excellent algorithm for files that are known to be nearly sorted.
11. Knuth [*5*] is the best reference for sorting algorithms. He covers internal, external, and parallel algorithms. He discusses 25 sorting algorithms in his text, which he states, is still only a fraction of the sorting algorithms that have been invented.

## EXTERNAL SORTING ALGORITHMS

In many cases, a file is so large that it will not fit into main memory. In this case internal sorting algorithms are not appropriate. In this section we will present some common external sort-merge algorithms for sorting

files that cannot fit in main memory. We will present the two-way sort-merge, two-way balanced sort-merge, k-way balanced sort-merge, and the polyphase sort-merge algorithms. Each of these algorithms has a sort phase and a merge phase and use the following general strategy: (1) Sort phase: Suppose the size of the available main memory is M. The first pass through the external file to be sorted will read in M records at a time into main memory and sort them by some internal sorting algorithm. Each of the sorted blocks of records is called a run. The sorted runs are distributed to other external files to be merged later. (2) Merge phase: Runs, one at a time, from each of the external files generated from the sort phase, are merged together to form larger runs. This is stored on another external file. Several merge passes through the file are made making successively larger runs until finally the entire file is in sorted order.

Most often, the data is accessed in a sequential manner so it does not matter if the storage device being used is disk or tape; all of these sort-merge algorithms will still be appropriate. Since most of the cost of external sorting is input-output, a rough measure of the cost of a sort-merge algorithm is the number of times each record in the file is read or written. This is also the number of passes over all of the data. Therefore, it is very important to minimize the number of passes required during the merge phase in any one of these algorithms.

### Two-Way Sort-Merge Algorithm

In a two-way sort-merge algorithm two files are merged during each merge phase onto a third file. Suppose a file of records to be sorted contains the following keys in the order given and the size of available main memory is M = 3.

File 1 (original data):

| 32 | 89 | 02 | 45 | 23 | 90 | 62 | 18 | 76 | 39 | 71 | 29 | 54 | 97 | 07 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

The size of each run is three. Thus records, three at a time are brought into main memory, sorted, then written alternately onto two external files (files 2 and 3) as shown below:

File 1: Empty

| File 2: | 02 | 32 | 89 | 18 | 62 | 76 | 07 | 54 | 97 |
|---|---|---|---|---|---|---|---|---|---|

| File 3: | 23 | 45 | 90 | 29 | 39 | 71 |
|---|---|---|---|---|---|---|

This is the end of the sort phase. Now the first merge phase is performed where corresponding runs on files 2 and 3 are merged and written back to file 1 as shown below:

File 1:

| 02 | 23 | 32 | 45 | 89 | 90 | 18 | 29 | 39 | 62 | 71 | 76 | 07 | 54 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

File 2: Empty

File 3: Empty

Now the runs in file 1 are redistributed alternately to files 2 and 3:

File 1: Empty

File 2:

| 02 | 23 | 32 | 45 | 89 | 90 | 07 | 54 | 97 |
|---|---|---|---|---|---|---|---|---|

File 3:

| 18 | 29 | 39 | 62 | 71 | 76 |
|---|---|---|---|---|---|

The second merge step is performed yielding:

File 1:

| 02 | 18 | 23 | 29 | 32 | 39 | 45 | 62 | 71 | 76 | 89 | 90 | 07 | 54 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

File 2: Empty

File 3: Empty

Again the runs in file 1 are redistributed alternately to files 2 and 3:

File 1: Empty

File 2:

| 02 | 18 | 23 | 29 | 32 | 39 | 45 | 62 | 71 | 76 | 89 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|---|

File 3:

| 07 | 54 | 97 |
|---|---|---|

Now the third and final merge step is performed yielding the sorted file:

File 1:

| 02 | 07 | 18 | 23 | 29 | 32 | 39 | 45 | 54 | 62 | 71 | 76 | 89 | 90 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

File 2: Empty

File 3: Empty

Each merge step of the two-way sort-merge algorithm doubles the size of each run and hence reduces the number of runs by approximately half. In the above case the first merge step merged five runs of length 3 to two runs of length 6 and one run of length 3. The second merge yielded one run of length 12 and one run of length 3. Finally a run of 15 was generated. Suppose in the above example the original file has only 12 records. The sort phase would generate four runs of length 3. The merge steps would take four runs of length 3 and generate two runs of length 6 and finally one run of length 12, requiring two merge steps. Thus if the file contains N records and main memory size is M (also the size of

the runs generated during the sort phase) then the number of required merge steps is given by $\lceil \log_2 (N/M) \rceil$.

### Balanced Two-Way Sort-Merge Algorithm

In the above two-way sort-merge algorithm, a redistribution of the runs from one file to two other files was needed after each merge phase before another merge sould be performed. A simple improvement to the two-way sort-merge algorithm is to increase the number of files. Instead of merging two input files into one output file, the balanced two-way sort-merge algorithm merges two input files storing the merged runs alternately on two output files. Increasing the number of output files to match the number of input files will eliminate the costly redistribution of the runs before another merge step can be performed. Consider the same file we used in the previous algorithm only with four files available instead of three.

File 3 (original data):

| 32 | 89 | 02 | 45 | 23 | 90 | 62 | 18 | 76 | 39 | 71 | 29 | 54 | 97 | 07 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Records three at a time are brought into main memory, sorted then written alternately on two external files (files 1 and 2) as shown below:

File 1:

| 02 | 32 | 89 | 18 | 62 | 76 | 07 | 54 | 97 |
|---|---|---|---|---|---|---|---|---|

File 2:

| 23 | 45 | 90 | 29 | 39 | 71 |
|---|---|---|---|---|---|

File 3: Empty

File 4: Empty

The sort phase is the same for both the two-way sort-merge and the balanced two-way sort-merge algorithms. Now corresponding runs on files 1 and 2 are merged and written alternately to files 3 and 4 as shown below:

File 1: Empty

File 2: Empty

File 3:

| 02 | 23 | 32 | 45 | 89 | 90 | 07 | 54 | 97 |
|---|---|---|---|---|---|---|---|---|

File 4:

| 18 | 29 | 39 | 62 | 71 | 76 |
|---|---|---|---|---|---|

The second merge step is performed yielding:

File 1:

| 02 | 18 | 23 | 29 | 32 | 39 | 45 | 62 | 71 | 76 | 89 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|---|

File 2:

| 07 | 54 | 97 |
|---|---|---|

File 3: Empty

File 4: Empty

Finally, a third merge step is performed yielding the sorted file:

File 1: Empty

File 2: Empty

File 3:

| 02 | 07 | 18 | 23 | 29 | 32 | 39 | 45 | 54 | 62 | 71 | 76 | 89 | 90 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

File 4: Empty

The same number of merge steps are required in both the two-way sort-merge and the balanced two-way sort-merge algorithms, $\lceil \log_2 (N/M) \rceil$. However, the number of passes through the file for the balanced two-way sort-merge is about half that of the two-way sort-merge. The two-way sort-merge algorithm requires two passes through the file per merge step due to the redistribution of the runs from one file to two other files after each merge. The balanced two-way sort-merge algorithm requires only one pass through the file per merge step and thus is approximately twice as fast as the two-way merge-sort algorithm. A natural extension of the balanced two-way sort-merge algorithm is to increase the number of available files to k input files and k output files. This is in general called the balanced k-way sort-merge algorithm. With additional input and output files the number of runs contained on each file will be less which results in fewer merge steps and a faster algorithm.

## Polyphase Sort-Merge

The balanced two-way sort-merge algorithm is in need of two improvements: (1) a reduction in the number of files required and (2) avoiding copying runs when the number of runs is not a multiple of the number of files. Note in each case for both the two-way sort-merge and the balanced two-way sort-merge the "odd" run (07, 54, 97) was copied during each merge phase and was not involved until the very last merge. The polyphase sort-merge algorithm has incorporated both these improvements. The basic idea of the polyphase sort-merge is to merge the runs somewhat unevenly among all of the files always leaving one file empty and then apply the "merge until empty" strategy. As soon as one of the input files is empty, one of the output files becomes an input file and the empty input file becomes one of the output files. Consider the same file we used in the previous examples with three files available and M = 3.

File 3:

| 32 | 89 | 02 | 45 | 23 | 90 | 62 | 18 | 76 | 39 | 71 | 29 | 54 | 97 | 07 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Records three at a time are brought into main memory, sorted then written alternately on two external files (files 1 and 2) as shown below: (The sort phase is the same for the two-way sort-merge, balanced two-way sort-merge and the polyphase sort-merge algorithms.)

File 1: | 02 32 89 | 18 62 76 | 07 54 97 |

File 2: | 23 45 90 | 29 39 71 |

File 3: Empty

Now the runs on files 1 and 2 are merged onto file 3 until file 2 becomes empty, yielding:

File 1: | 07 54 97 |

File 2: Empty

File 3: | 02 23 32 45 89 90 | 18 29 39 62 71 76 |

Next, file 2 becomes the output file and runs on files 1 and 3 are merged onto file 2 until file 1 becomes empty:

File 2: | 02 07 23 32 45 54 89 90 97 |

File 1: Empty

File 3: | 18 29 39 62 71 76 |

Finally, files 2 and 3 merge and yield the desired result in file 2:

File 2: Empty

File 1:

| 02 07 18 23 29 32 39 45 54 62 71 76 89 90 97 |

File 3: Empty

The polyphase sort-merge algorithm with k input files requires a total of k + 1 files where the balanced k-way sort-merge algorithm requires 2k files. Furthermore, the polyphase merge is broken up into many phases which do not necessarily involve all of the data each time. There is also no direct copying of data. The polyphase sort-merge algorithm can be extended to work on any number of files. For example suppose there are four files and the results of the sort phase placed 12 initial runs onto file 1, 8 initial runs onto file 2, and 5 initial runs onto file 3. File 4 is the initial output file. A three-way merge of files 1, 2, and 3 onto file 4 until file 3 is empty yields 7, 3, and 5 runs on files 1, 2, and 4, respectively. The following table summarizes the merge phases for this example until finally 3 contains the desired results.

| | The number of runs on file | | | |
|---|---|---|---|---|
| | F1 | F2 | F3 | F4 |
| Sort phase | 12 | 8 | 5 | 0 |
| Merge pass 1 | 7 | 3 | 0 | 5 |
| Merge pass 2 | 4 | 0 | 3 | 2 |
| Merge pass 3 | 2 | 2 | 1 | 0 |
| Merge pass 4 | 1 | 1 | 0 | 1 |
| Merge pass 5 | 0 | 0 | 1 | 0 |

The most difficult task involved in the polyphase sort-merge algorithm is how to distribute the initial runs. Given the initial number of runs on each file it is not difficult to build the table. To get the next line in the table, take the smallest number of runs among the input files and subtract that value from each of the input files. Also place that value into the output file and repeat the process. At first glance it may appear that the initial runs should be distributed as evenly as possible. Consider what happens if the number of runs is the same on each input file. The first merge will leave one file with all of the runs and all the other files empty. In order to continue, a redistribution of the runs will be needed. Even when the number of runs is about equal, the performance is poor. The reader is encouraged to build a table when the initial number of runs for four files are, respectively, 10, 10, 11, 0. This results in extremely poor performance.

For more details concerning the two-way sort-merge, balanced two-way sort-merge, balanced k-way sort merge, and polyphase sort-merge algorithms the reader is referred elsewhere [*6, 7, 22, 30—34*]. Programs detailing each algorithm can be found in [*35*]. For an exhaustive treatment on external sorting techniques, including analysis of how to initially distribute runs for the polyphase merge see Knuth [*5*].

## An Introduction to Parallel Sorting Algorithms

There has always been a demand for more computing power than what today's computers can deliver. It is clear that significant increases in speed due to faster electronic devices is reaching the limit. Thus the current trend in attaining high computational speeds is to use a parallel computer. In this way problems can be partitioned into several smaller pieces and solved simultaneously on separate processors. With the steady decline of hardware costs it is now possible to have hundreds even tens of thousands of processors available to solve a problem. This can greatly reduce the solution time for the problem but it may add to the complexity of the algorithm.

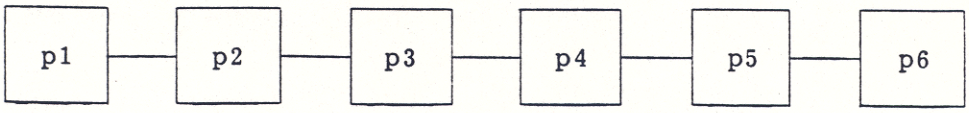
In order to develop an algorithm to run on a parallel computer it is important to know the architecture of the parallel machine. For our purposes, we will classify parallel architectures into two broad classifications: single instruction stream, multiple data stream (SIMD) and multiple instruction stream, multiple data stream (MIMD). An SIMD computer consists of several processors operating under the control of a single instruction issued

by a central control unit. Each processor has its own local memory (data) and during a given time unit those processors that are active will execute the same instruction but on different data. Each processor has the ability to exchange or pass data to each of the other processors by some interconnection scheme or via shared memory. There are several configurations that can be used for connecting the processors together, and depending on this scheme, different sorting algorithms have been developed to take advantage of the architecture. Akl [36] discusses various parallel sorting algorithms based on linear, mesh, cube, tree, and perfect shuffle interconnection architectures. This is an excellent survey text on parallel sorting algorithms. Bitton and others have an excellent article on the taxonomy of parallel sorting [37]. An MIMD computer has each processor execute independent instructions on separate data items in an asychronous environment. Akl also reviews several sorting algorithms for this type of computer as well. Other excellent references for parallel sorting algorithms include [5, 38—47].

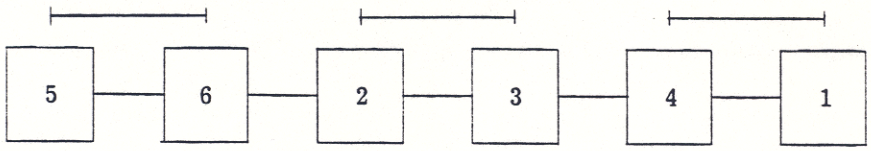
Before we present some parallel sorting algorithms, it is necessary to define some measurements used in evaluating parallel algorithms. The speedup produced by a parallel algorithm is a good indication of how good the algorithm is. The speedup of a parallel algorithm running on p processors is defined as the ratio of executing the fastest known sequential algorithm for a problem by the same computer executing the parallel algorithm using p processors. Ideally, one would hope to achieve a speedup of p when using p parallel processors. In practice this is not achieved. For one thing it is difficult to partition a problem into p tasks each taking 1/p of the time and second, communication times between processors and other restrictions must be considered in the overall performance. When evaluating a parallel algorithm, the number of processors used must be taken into consideration as well. We define cost of a parallel algorithm as the parallel running time times the number of processors used. The best known running time on a sequential processor for a key-comparison sorting algorithm is $O(n\log n)$. If we have a parallel sorting algorithm using n processors then we could get no better running time than $O(\log n)$, that is $(n\log n)/n$. Thus this is a lower bound execution time to sort n items in parallel using n processors. Therefore, a cost-optimal parallel sorting algorithm is $O(n\log n)$. Another measurement is the efficiency of a parallel algorithm which is defined as the running time of the fastest known sequential algorithm divided by the cost of the parallel algorithm. Consider the following example illustrating these measurements. Suppose we are working on a parallel algorithm for quicksort. The sequential quicksort algorithm executes in 8 seconds on a single parallel processor. Our parallel version of quicksort executes in 2 seconds while using five processors. Therefore, our parallel algorithm for quicksort has a speedup of 4, a cost of 10, and an efficiency of 0.8 with five processors.

### The Odd-Even Transposition Sort

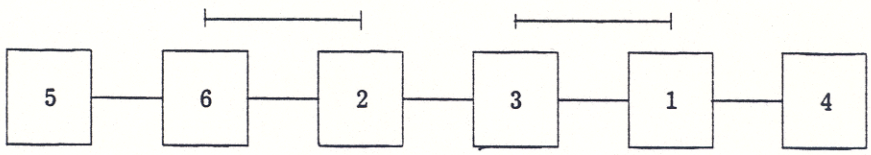
The odd-even transposition sort is perhaps the most well-known sorting algorithm using a one-dimensional linear array of parallel processors [5, 38, 42, 46]. A linear array of six processors is illustrated below with communications lines shown between the processors:



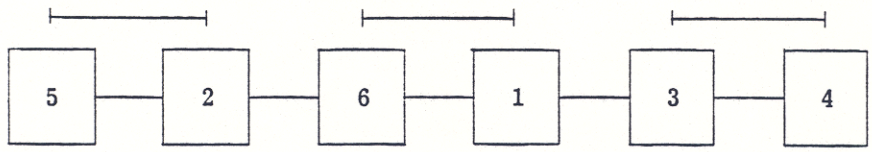

In this method the original keys to be sorted are first placed into the linear array of processors, one key per processor. In each step of the sorting process odd and even-numbered processors are activated alternately. In each step the key in an activated processor is compared to the key in the adjacent processor to the right. An exchange is made if the keys are out or order. We illustrate this algorithm by sorting keys 5, 6, 2, 3, 4, 1 which are initially placed into six processors.



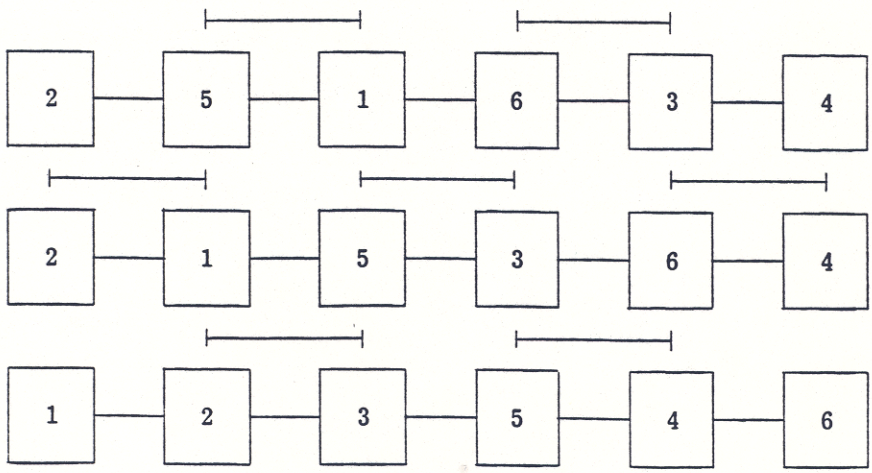

Suppose the odd processors are activated at step one. This means processors (p1, p2), (p3, p4), and (p5, p6) as indicated above compare and exchange keys. This results in the following:

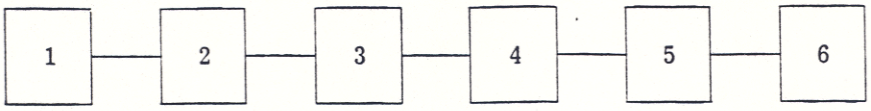



Now at step two the even processors are activated and processors (p2, p3), (p4, p5) as indicated above compare and exchange keys. This yields the following:




Successive steps of the algorithm are shown below:



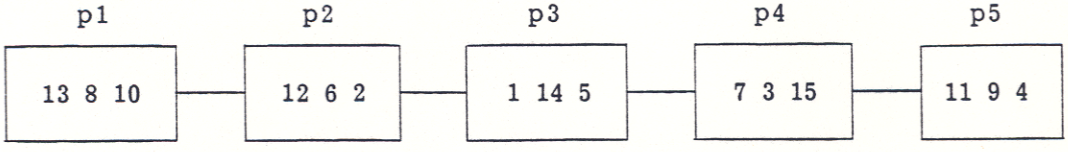



This method sorts n keys using n processors in n steps. Thus the parallel running time is $O(n)$. The cost is $n \times O(n) = O(n^2)$, which is not cost-optimal. The algorithm for the odd-even transposition sort is given below in pseudo code. Let p[i] denote the value of the key in processor i.

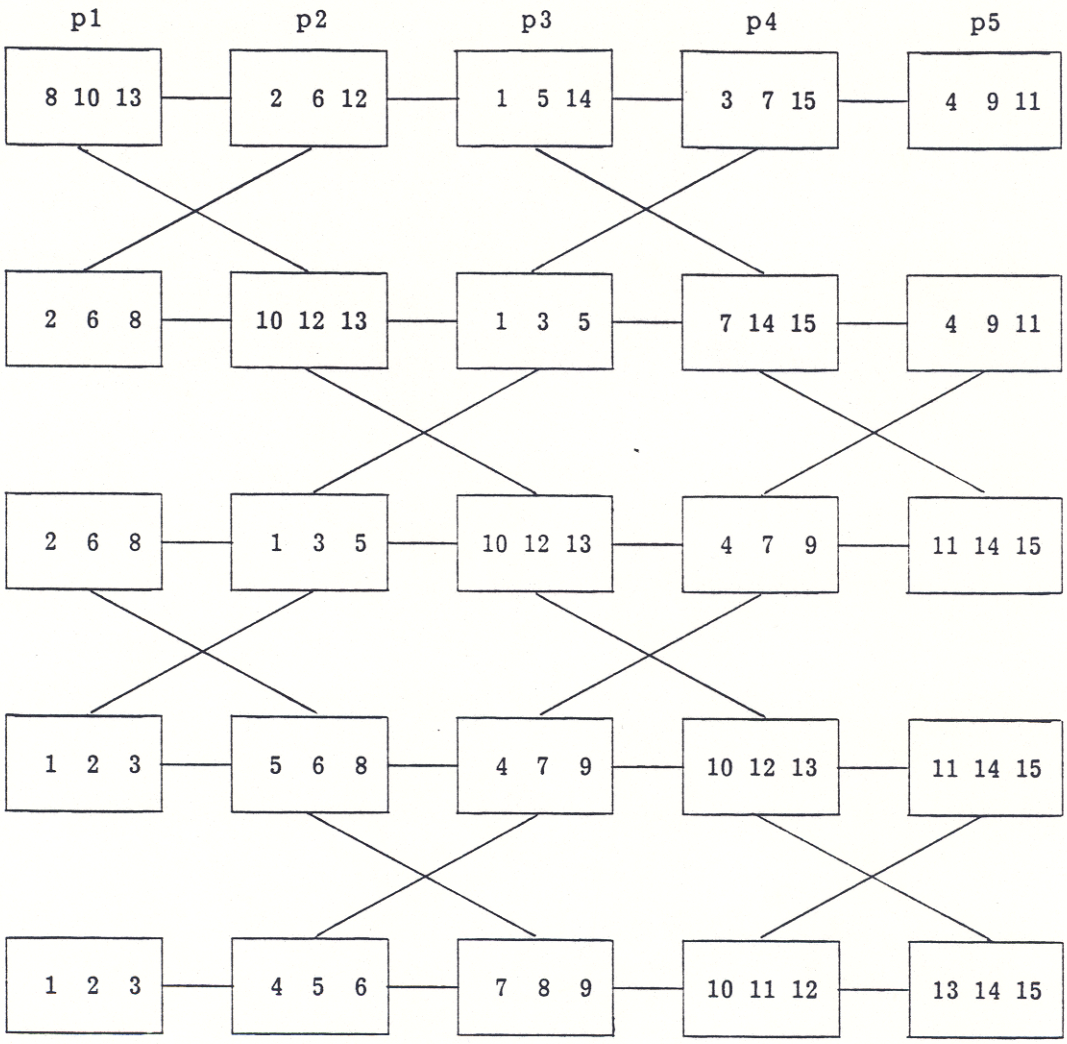
```
for j = 1 to ⌈n/2⌉ do
  begin
    do i in parallel
    for i = 1,3,5,...,2⌈n/2⌉ − 1
      if p[i] > p[i+1] then swap (p[i],p[i+1])
    end parallel

    do i in parallel
    for i = 2,4,6,...,2⌊n/2⌋
      if p[i] > p[i+1] then swap (p[i],p[i+1])
    end parallel
  end
```

### Merge-Splitting Sort

It is possible to generalize the odd-even transposition sort where each processor holds more than one element. In this case, the comparison-exchange operation is replaced with a merge-split operation. Suppose we have n elements to sort with p processors ($p < n$). Each processor is initially given $n/p$ items. If $n/p$ is not an integer, then dummy elements are padded to the list of items such that the number of items to sort is a multiple of p. The dummy items must be larger than all of the other items so as not to affect the sorting process. Consider the following example with $n = 15$ and $p = 5$:



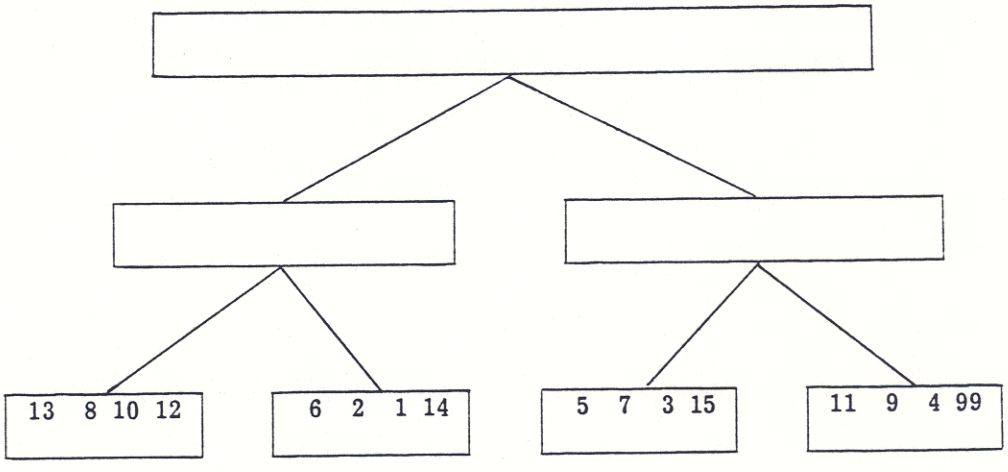

During the first step, each processor sorts its elements using some fast internal sorting algorithm. Next each odd-numbered processor merges its elements with the elements in its neighbor to the right. The odd-numbered processor then keeps the first half of the merged list and gives the second half of the merged list back to its neighbor on the right. During the next step the even-numbered processors are activated and perform the same merge-split operation. These steps alternate for p iterations, after which the elements are in sorted order within the p processors. The sort step and all of the merge-split steps for this example are shown below.



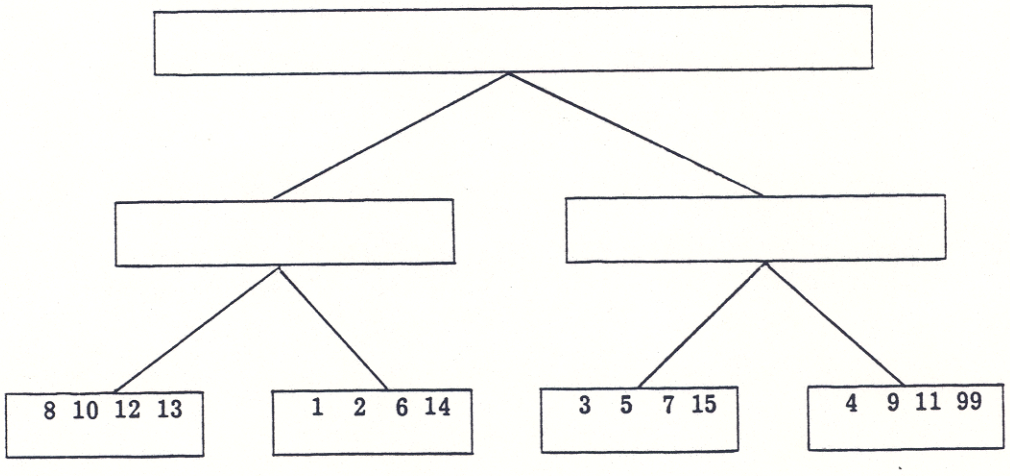

The first step in this algorithm requires a sequential sort of n/p items on each processor. Using an efficient algorithm such as heapsort or quicksort this can be done in $O((n/p) \log (n/p))$ running time. Each of the p merge-split steps that follow take $O(n/p)$ calculations each for a total running time of $O((n/p) \log (n/p)) + O(n)$. The cost of this algorithm is $O(n \log n) + O(np)$, which is cost-optimal for p <= log n [36].

Suppose we have a SIMD computer where processors are interconnected to form a binary tree. This architecture is referred to as a tree machine. If the tree has d levels, then it has $2^d-1$ nodes (processors) in the tree. Each node is connected to a single parent (unless the node is the root) and is connected to two child nodes (unless the node is a leaf). Suppose the number of elements to be sorted is a power of 2, that is $n = 2^m$ for some positive integer m, which must also be a power of 2. Thus $m = 2^q$ for some positive integer, q. Suppose we have a tree machine with m leaf processors. The total number of processors in the tree machine is 2m - 1 or 2(log*n*) - 1. Notice that d = q + 1. During the first step of the algorithm, n/m elements are placed into each of the m leaf processors and sorted

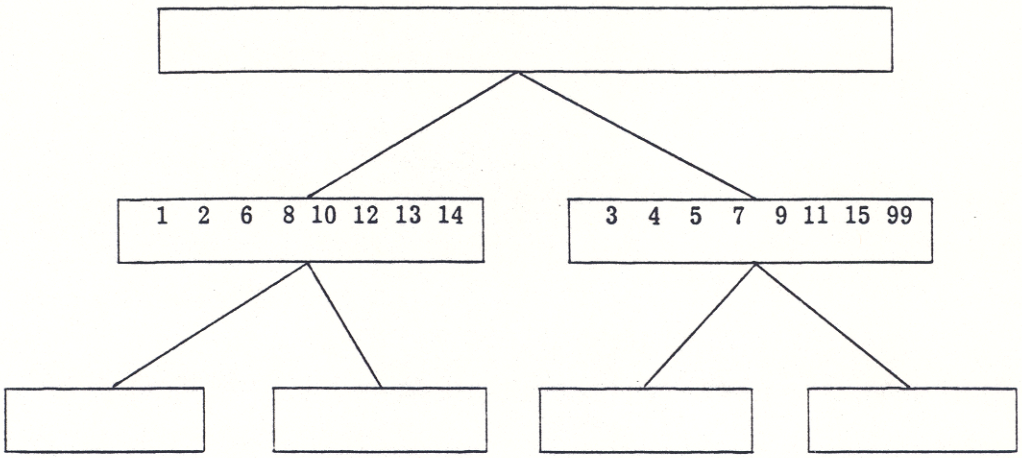
simultaneously, using an optimal sequential internal sorting algorithm. For example, consider sorting the following 15 values using this algorithm: (13, 8, 10, 12, 6, 2, 1, 14, 5, 7, 3, 15, 11, 9, 4). First we pad the list with a single value larger than all the others (99) in order for the number of values to be a power of two. Thus n = 16 which means that m = 4, q = 2 and d = 3, and the total number of processors required in the tree machine is $2^d - 1 = 7$. The tree machine is shown below with n/m = 4 values placed into each of the m leaf nodes:



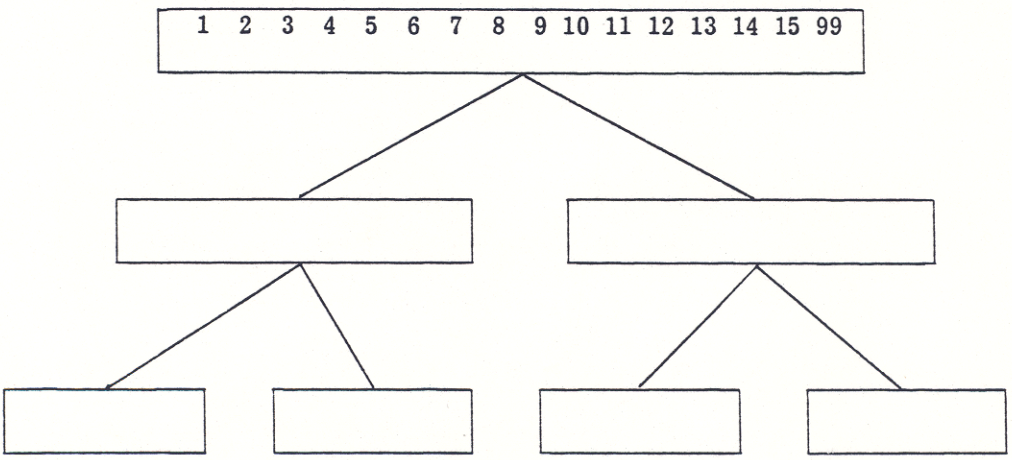

Each leaf processor sorts its values using an efficient internal sorting algorithm:




Now each processor at the next higher level takes the sorted input from each of its children processors and merges the two sequences. This results in the following:




Finally the root processor merges the sorted sequences from its children processors resulting in the desired sorted sequence in the root processor:




### Analysis of the Parallel Tree Sorting Algorithm

During the first step, m leaf processors read in n/m [i.e., n/(log n)] values each at $O(n/(\log n))$ time. The values are then sorted using heapsort or quicksort or some other fast algorithm taking $O((n/(\log n)\log(n/(\log n)))$ running time. This takes a total of $O(n)$ running time in the worst case. The running time for all of the merge steps is determined as follows. Suppose at a given level there are k processors. Each processor merges n/k items for a total running time of $O(n/k)$. The values of k for the various levels of a tree machine are 1, 2, 4, 8,...(i.e., the powers of 2). Thus the total running time for all of the merge steps is $O(n + n/2 + n/4 + n/8 + \ldots)$ which is $O(n)$. Therefore the total running time to sort n items using a parallel tree sorting algorithm is $O(n)$. The number of processors required is 2m − 1. Since m = log*n* the number of required processors is $O(\log n)$ making the cost of this algorithm $O(n\log n)$ which is cost-optimal [*36*].

## REFERENCES


1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.
2. J. L. Bentley, "Programming Pearls: Aha! Algorithms," *Commun. ACM*, *26*(9), 623–628 (September 1983).
3. J. L. Bentley, "Programming Pearls: Algorithm Design Techniques," *Commun. ACM*, *27*(9), 865–871 (September 1984).
4. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1984.
5. D. E. Knuth, *The Art of Computer Programming*, Vol 3, *Sorting and Searching*, Addison-Wesley, Reading, MA, 1972.
6. R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1984.
7. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
8. A. M. Tenenbaum and M. J. Augenstein, *Data Structures Using Pascal*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
9. D. L. Shell, "A High Speed Sorting Procedure," *Commun. ACM*, *2*(7), 1959.
10. V. Pratt, Shellsort and Sorting Networks, Outstanding Ph.D. dissertations in Computer Science, Stanford University, 1972.
11. C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, *4*(7), 321 (July 1961).
12. C. A. R. Hoare, "Quicksort," *Computer J.*, *5*(1), 10–15 (1962).
13. R. L. Wainwright, "Quicksort Algorithms With an Early Exit for Sorted Subfiles," in *Proceedings of the 1987 ACM Computer Science Conference*, St. Louis, February 1987, pp. 183–190.
14. T. A. Standish, *Data Structures Techniques*, Addison-Wesley, Reading, MA, 1980.
15. J. L. Bentley, "Programming Pearls: How to Sort," *Commun. ACM*, *27*(4), 287–291 (April 1984).
16. D. Motzkin, "Meansort," *Commun. ACM*, *26*(4) (April 1983).

16a. R. L. Wainwright, "A Class of Sorting Algorithms Based on Quicksort," *Commun. ACM*, *28*(4) (April 1985).

17. J. Amsterdam, "An Analysis of Sorts," *BYTE*, 105–112 (September 1985).
18. R. L. Kruse, *Data Structures and Program Design*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
19. R. Sedgewick, "The Analysis of Quicksort Programs," *Acta Informatica*, *7*, 327–355 (1977).
20. R. Sedgewick, "Implementing Quicksort Programs," *Commun. ACM*, *21*(10) (October 1978).
21. R. Sedgewick, "Quicksort with Equal Keys," *J. Comput. SIAM*, *6*(2) (June 1977).
22. J. P. Tremblay and P. G. Sorenson, *An Introduction to Data Structures and Applications*, McGraw-Hill, New York, 1984.
23. J. W. J. Williams, "Heapsort (Algorithm 232)," *Commun. ACM*, *7*(6), 347–348 (1964).
24. R. W. Floyd, "Treesort (Algorithms 113 and 243)," *Commun. ACM*, *5*(8), (1962); *Commun. ACM*, *7*(12) (1964).
25. J. L. Bentley, "Programming Pearls: Thanks, Heaps," *Commun. ACM*, *28*(3), 245–250 (March 1985).
26. W. A. Martin, "Sorting," *Comp. Surveys*, *3*(4), 147–174 (1971).

27. D. Motzkin, "A Stable Quicksort," *Software Pract. Exp.*, *11*, 607–611 (1981).
28. S. M. Merritt, "An Inverted Taxonomy of Sorting Algorithms," *Commun. ACM*, *28*(1) (January 1985).
29. C. R. Cook and D. J. Kim, "Best Sorting Algorithm for Nearly Sorted Lists," *Commun. ACM*, *23*, 11 (November 1980).
30. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, MD, 1982.
31. J. F. Korsh, *Data Structures, Algorithms, and Program Style*, PWS Publishers, Boston, 1986.
32. R. Sedgewick, "Data Movement in Odd-Even Merging," *J. Comput. SIAM*, *7*(3) (1978).
33. N. Wirth, *Algorithms & Data Structures*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
34. D. A. Zave, "Optimal Polyphase Sorting," *J. Comput. SIAM*, *6*(1) (March 1977).
35. N. E. Miller, *File Structures Using Pascal*, Benjamin/Cummings, Menlo Park, CA, 1987.
36. S. G. Akl, *Parallel Sorting Algorithms*, Academic Press, Orlando, FL, 1985.
37. D. Bitton, D. J. DeWitt, D. K. Hsaio, and J. Menon, "A Taxonomy of Parallel Sorting," *ACM Comput. Surv.*, *16*(3), 287–318 (1984).
38. G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. Comput.*, *C-27*(1) (January 1978).
39. R. Haggkvist and P. Hell, "Parallel Sorting with Constant Time for Comparisons," *J. Comput. SIAM*, *10*(3) (August 1981).
40. D. S. Hirschberg, "Fast Parallel Sorting Algorithms," *Commun. ACM*, *21*(8), 657–666 (August 1978).
41. M. Kumar and D. S. Hirschberg, "An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and its Application in Parallel Sorting Schemes," *IEEE Trans. Comput.*, *C-32* (March 1983).
42. H. T. Kung, *The Structure of Parallel Algorithms*, *Advances in Computers*, vol. 19, Academic Press, New York, 1980.
43. D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. Comput.*, *C-27*(1) (January 1979).
44. F. P. Preparata, "New Parallel Sorting Schemes," *IEEE Trans. Comput.*, *C-27*(7) (July 1978).
45. Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging and Sorting in a Parallel Computational Model," *J. Algorithms*, *2*(1) (March 1981).
46. C. D. Thompson and H. T. Kung, "Sorting on Mesh-Connected Parallel Computer," *Common. ACM*, *20*(4) (April 1977).
47. M. L. Warshauerm, "Conway's Parallel Sorting Algorithm," *J. Algorithms* (7) 270–276 (1986).

ROGER L. WAINWRIGHT