

Chapter 9

■ SORTING & SELECTION

9.1 Introduction

Sorting is one of the most fundamental processes in computing with applications in every field that uses computers, be it business, engineering, or science. We search every time we use a telephone book or a dictionary. We could not do these tasks efficiently unless the entries in each were sorted. Hence the main reason for sorting a set of data is to allow quick access to the elements of the set.

9.2 Lower Bounds for Sorting

A trivial lower bound on the number of steps needed to sort n items is $O(n)$. This is required just to read n items. Let us consider the very simple case of sorting 3 numbers a, b, c . Figure 1 shows the *decision tree* for deciding the permutation needed to transform a, b, c into ascending order.

It can be seen that there are 6 different leaf nodes. These correspond to the $3!$ different permutations of the 3 numbers. For example, if a, b, c are such that $c < a < b$ then any sorting algorithm must re-arrange these according to the permutation 3, 1, 2. Each permutation corresponds to the unique path from the root node to a leaf node. The internal nodes represent the tests or decision points in any sorting algorithm. In general, the decision tree for sorting n numbers will have $n!$ leaf nodes. The length of the longest path in the decision tree (the height of the tree) is the minimum number of comparisons required by any algorithm for the worst-case input. In effect any sorting algorithm must search this decision tree for the correct permutation. We note that algorithms may make more comparisons than the number on the longest path in the decision tree. In fact all $O(n^2)$ algorithms do this in the worst case.

Following [RND77] we let $C(n)$ denote the minimum number of comparisons required by any sorting algorithm for the worst-case input. That is,

$$C(n) = \min_{\text{All sorting algs}} \left[\max_{\text{All inputs}} (\text{No. Comparisons}) \right].$$

Now a binary tree of height h can have, at most, 2^h leaves. The decision tree for sorting n items has $n!$ leaves and so

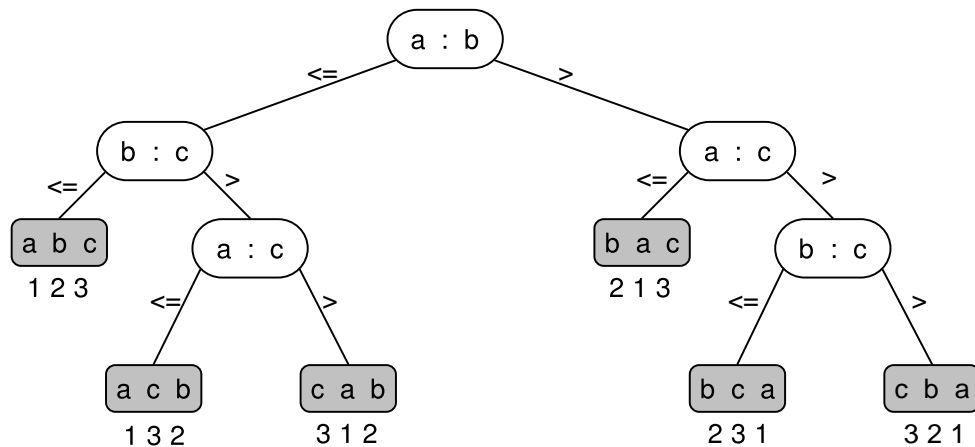


Figure 9.1. Decision Tree for Sorting 3 Numbers.

$$2^h = 2^{C(n)} \geq n!.$$

Taking logs of both sides we get

$$C(n) \geq \log_2 n! = \log_2 1 + \log_2 2 + \dots + \log_2 i + \dots + \log_2 n < O(n \log_2 n).$$

9.3 $O(n^2)$ Sorting Algorithms

In the algorithms and procedures that follow we will use the following Pascal data types.

```

CONST
    MaxArrSize = 30000;
TYPE
    ElemType   = INTEGER;
    IdxType    = 0..MaxArrType;
    ArrType    = ARRAY[IdxType] OF ElemType;
    
```

9.3.1 Selection Sort.

The main step in Selection sort is : scan the unsorted part of the array $T[1..i]$, locate the maximum element at j^* , say, and swap this with the last element of the unsorted array, i.e., $T[j^*] \leftrightarrow T[i]$.

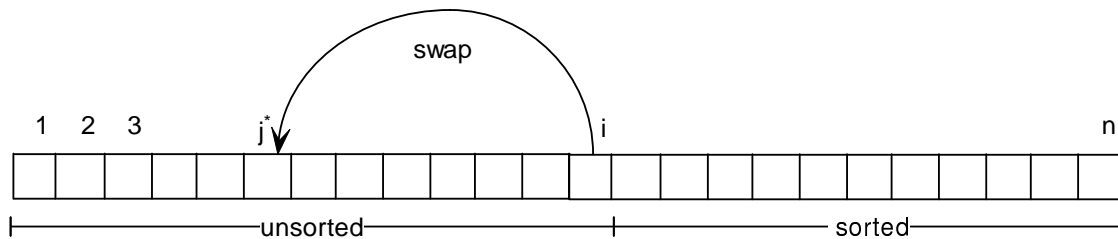


Figure 9.2. Selection Sorting.

```

procedure SelectSort(VAR T : ArrType; n : IndxType);
VAR
    i, j : IndxType;
    temp : ElemType;
BEGIN
    FOR i := n DOWNTO 2 DO BEGIN
        jstar := i;
        FOR j := 2 TO i DO BEGIN
            IF T[j] > T[jstar] THEN jstar := j;
        END; { for j }
        temp := T[i];      { Swap T[i] and T[jstar] }
        T[i] := T[jstar];
        T[jstar] := temp;
    END; { for i }
END; { proc SelectSort}

```

Analysis of SelectSort. The outer FOR-loop is performed $n - 1$ times and for each of these iterations the inner loop is performed $i - 1$ times. Hence the number of comparisons is

$$C_{ss}(n) = \sum_{i=2}^n \sum_{j=2}^i 1 \text{ comp} = \sum_{i=2}^n (i - 1) = \sum_{i=1}^{n-1} i = n(n - 1)/2.$$

The data movements only occur in the outer loop and so $M_{ss}(n) = n - 1 = O(n)$.

9.3.2 Insertion Sort.

This is another $O(n^2)$ method which is probably the best of the $O(n^2)$ sorting algorithms. Figure 8.3 shows the basic step at each iteration : (1), the next unsorted element $T[i]$ is put into a temporary location, (2), its correct position in the sorted part of the array is found at j , say, the elements $T[j..i - 1]$ are moved up one place, and (3), $T[i]$, stored in $temp$, is inserted into position j .

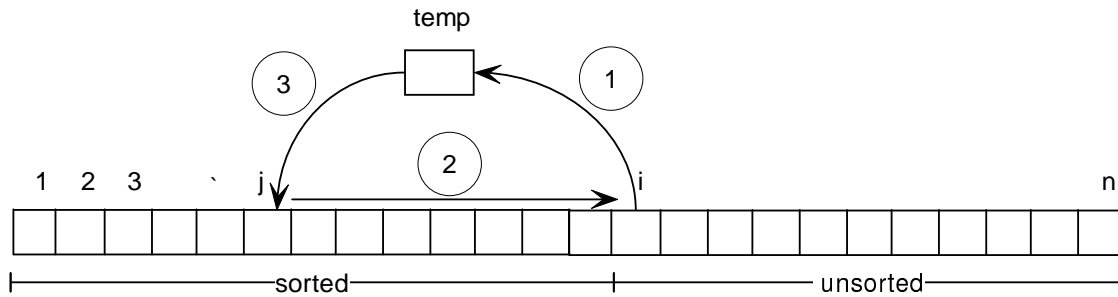


Figure 9.3. Insertion Sorting.

```

procedure InsertSort(VAR T : ArrType; n : IndxType);
VAR
  i, j : IndxType;
  temp : ElemType;
BEGIN
  FOR i := 2 TO n DO BEGIN
    temp := T[i];
    j := i;
    WHILE (j > 1) AND (T[j-1] > temp) DO BEGIN
      T[j] := T[j-1];
      j := j - 1;
    END; { while }
    T[j] := temp;
  END; { for }
END; { proc InsertSort}

```

Analysis of InsertSort. Consider the WHILE-loop first : for any i we move at most $i - 1$ elements up one place. Now this is done for $i = 2, 3, \dots, n$ and so the total number of moves is

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2).$$

The data movements occur in the inner loop and so $M_{is}(n) = O(n^2)$.

Modifications. Insertion Sort can be speeded up by a constant factor if we put a *sentinel* value of $-\infty$ in $T[0]$. This simplifies the WHILE-loop test to

```

WHILE T[j-1] > temp DO BEGIN

```

The sentinel ensures the new WHILE-loop test is false for $j = 1$. Thus j never goes below 1 and so the test for j can be eliminated. This simple trick can speed up Insertion Sort by 33% depending

on the machine-compiler combination. The sentinel trick can be used to great effect in other simple programs.

Exercise 9.3.1 : Show how a sentinel can be used to speed up the *Sequential Search* of a randomly ordered table $T[0..n]$.

Exercise 9.3.1 : Write a Pascal procedure that performs *Insertion Sort* on a doubly-linked list with header node and avoids the $O(n)$ data movement when using an array.

9.3.3 Bubble Sort.

This method appears in most elementary programming books. It is probably the slowest of the $O(n^2)$ algorithms. The main step is : scan the array and interchange any adjacent pair of elements that are out of order. It can be shown that each scan 'bubbles' the maximum of the current sub-array up to its correct position.

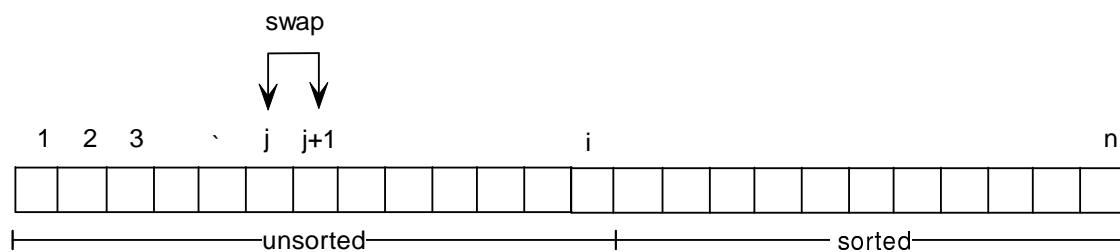


Figure 9.4. Bubble Sorting.

```

procedure BubbleSort(var T : ArrType; n : IndxType);
VAR
    i, j : IndxType;
    temp : ElemType;
BEGIN
    FOR i := n DOWNTO 2 DO BEGIN
        FOR j := 2 TO i DO BEGIN
            IF T[j-1] > T[j] THEN BEGIN
                temp := T[j];
                T[j] := T[j-1];
                T[j-1] := temp;
            END; { } if
        END; { for j }
    END; { for i }
END; { proc BubbleSort}

```

Analysis of Bubble Sort. The IF-test is performed $i - 1$ times for each iteration of the outer loop. Hence

$$C_{bs}(n) = \sum_{i=2}^n (i-1) = \frac{n(n-1)}{2}.$$

The number of data movements is $M_{bs}(n) = O(n^2)$.

Modifications. The procedure above is a very crude version of Bubble Sort. The first improvement that can be made is to stop if no exchanges have occurred in any scan : this means the array is sorted.

```

procedure BubbleSort2(var T : ArrType; n : IndxType);

  VAR
    i, j   : IndxType;
    temp   : ElemType;
    Sorted : BOOLEAN;
  BEGIN
    Sorted := false;
    i      := n
    WHILE (NOT Sorted) AND (i>1) DO BEGIN
      Sorted := True;
      FOR j := 2 TO i DO BEGIN
        IF T[j-1] > T[j] THEN BEGIN
          temp := T[j];
          T[j] := T[j-1];
          T[j-1] := temp;
          Sorted := false;
        END; { if }
      END; { for j }
      Dec(i);
    END; { while }
  END; { proc BubbleSort2 }

```

The algorithm can be improved further by keeping track of where the last exchange occurred in the current scan. The next scan need go no further than where the last exchange occurred. This modification includes the first because if no exchange occurs then last exchange occurred at 0.

```

procedure BubbleSort3(var T : ArrType; n : IndxType);
VAR
    i, j, LastEx : IndxType;
    temp          : ElemType;
BEGIN
    LastEx := n;
    WHILE LastEx > 1 DO BEGIN
        FOR j := 2 TO LastEx DO BEGIN
            IF T[j-1] > T[j] THEN BEGIN
                temp := T[j];
                T[j] := T[j-1];
                T[j-1] := temp;
                LastEx := j;
            END; { if }
        END; { for j }
    END; { while }
END; { proc BubbleSort3}

```

Although these modifications improve Bubble Sort they still don't make it competitive with Insertion Sort.

9.4 Divide and Conquer

We saw above that sorting large sets of elements can take a long time using InsertSort. The same is true of SelectSort and BubbleSort : they all are $O(n^2)$ algorithms. We say that these algorithms have $O(n^2)$ *complexity*. We now explain a principle that will help us devise much faster algorithms for many types of problem. This is the *Divide and Conquer* principle.

The divide and conquer principle can be stated algorithmically as follows:

algorithm DaCSolve (*Problem P of size n*)

```

1. Divide P into two parts :
    P1 of size n1 and P2 of size n2.
2(a). DaCSolve ( Problem P1 of size n1 ) → S1
2(b). DaCSolve ( Problem P2 of size n2 ) → S2
3.      Combine(S1, S2) → S, the solution of P
endalg{DaCSolve}

```

Notice that this is a *recursive* algorithm : it calls or invokes itself in steps 2(a) and 2(b). The divide and conquer principle is based on the fact that it is (usually) easier to solve two smaller problems than one larger problem.

9.5 $O(n \log n)$ Sorting Algorithms

9.5.1 Merge Sort.

This algorithm uses the divide and conquer in the following way : Divide $T[1..n]$ into $T[1..n/2]$ and $T[n/2 + 1..n]$, sort each half (recursively), and then merge the two sorted sub-arrays into one sorted array.

```

procedure MergeSort(VAR T : ArrType; L,R : IndxType);

VAR
    m : IndxType;
    temp : ElemType;
BEGIN
    IF R > L THEN BEGIN
        m := (L+R) DIV 2;    { Divide Step }
        MergeSort(T, L, m); { Solve sub-problems }
        MergeSort(T,m+1,R);
        Merge(T,L,m,R);     { Combine sub-problems }
    END; { proc MergeSort}

```

Exercise 9.5.1 : Write a procedure to merge two sorted arrays of size m and n and show that it works in $O(m + n)$ time.

Analysis of MergeSort. The divide step is trivial and takes $O(1)$ time. The two recursive steps require $T_{ms}(n/2)$ each. The combine step is $O(n)$. Hence

$$T_{ms}(n) = 2T_{ms}(n/2) + cn.$$

The solution of this recurrence is $T_{ms}(n) = c_{ms}n \log_2 n$. (see the analysis of QuickSort below).

Exercise 9.5.1 : Show how *Merge Sort* could be used to sort a file so large that it will not fit in memory.

9.5.2 QuickSort.

In 1962 one of the first sorting algorithms with a complexity less than $O(n^2)$ was invented by C.A.R. Hoare, now professor of Computer Science at Oxford University. He used the *Divide and Conquer* principle to construct a sorting algorithm whose complexity is $O(n \log_2 n)$. Hoare used this principle in the QuickSort algorithm as follows :

```

algorithm QuickSort( Array  $T[1 \dots n]$  of size  $n$  )


---


if  $n > 1$  do
    1. Divide step : Re-arrange the elements of  $T$  about
        a pivot element  $T[p]$ , so that

        
$$T[1 \dots p - 1] \leq T[p] < T[p + 1 \dots n].$$


    2(a). QuickSort( Array  $T[1 \dots p - 1]$  of size  $p - 1$  )
    2(b). QuickSort( Array  $T[p + 1 \dots n]$  of size  $n - p$  )
    3. No Combine step needed.
endif
endalg{QuickSort}
    
```

This algorithm repeatedly divides the array T into smaller and smaller sub-arrays until we are left with trivial sub-problems : sorting sub-arrays of size 1.

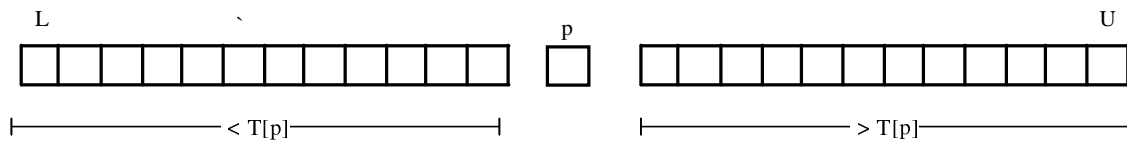


Figure 9.5. QuickSort.

The main problem remaining is the re-arrangement or *partition* step.

Partitioning. This is at the heart of the QuickSort algorithm. Indeed the algorithm is essentially a sequence of partitioning steps. There are two methods available : Hoare’s original method, which is faster but tricky to implement, and Lomuto’s method which is slower but easier to implement. We now show how both work and the we will implement Lomuto’s method.

We wish to re-arrange the array T about a pivot element so that

$$T[1 \dots p - 1] \leq T[p] < T[p + 1 \dots n].$$

We are free to choose any $T[i]$ about which to re-arrange the table. We will choose $T[L]$ and then shift the values of T around so that the value of $T[L]$ winds up in position p of T .

Hoare’s method move a left and a right i and j pointer towards each other until $T[i] > T[L]$ and $T[j] < T[L]$.

The picture of this process is shown in Figure ??.

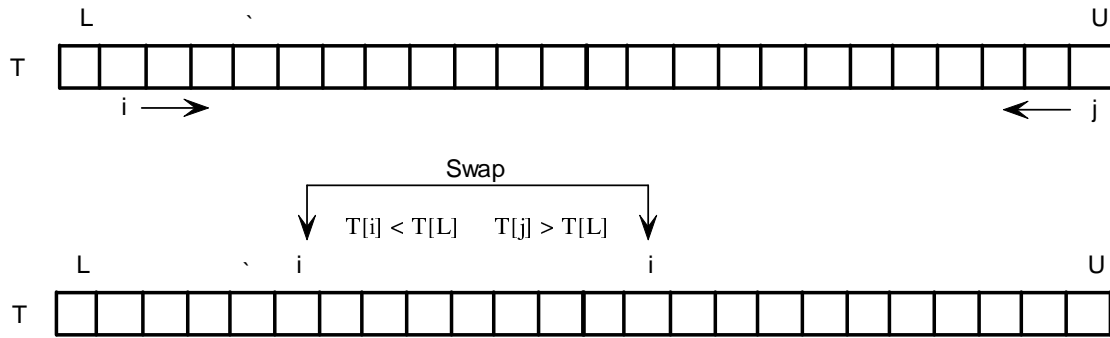


Figure 9.6. Hoare's Partitioning.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	swap
3		1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
		i													j	
3		1	4	1	5	9	2	6	5	3	5	8	9	7	9	3 (3,16)
		i													j	
3		1	3	1	5	9	2	6	5	3	5	8	9	7	9	4 (5,10)
				i						j						
3		1	3	1	3	9	2	6	5	5	5	8	9	7	9	4 (6,7)
				i	j											
3		1	3	1	3	2	9	6	5	5	5	8	9	7	9	4 cross
					j	i										
<hr/>																
2	1	3	1	3	3	9	6	5	5	5	8	9	7	9	4 (1,6)	
					p										$p=6$	

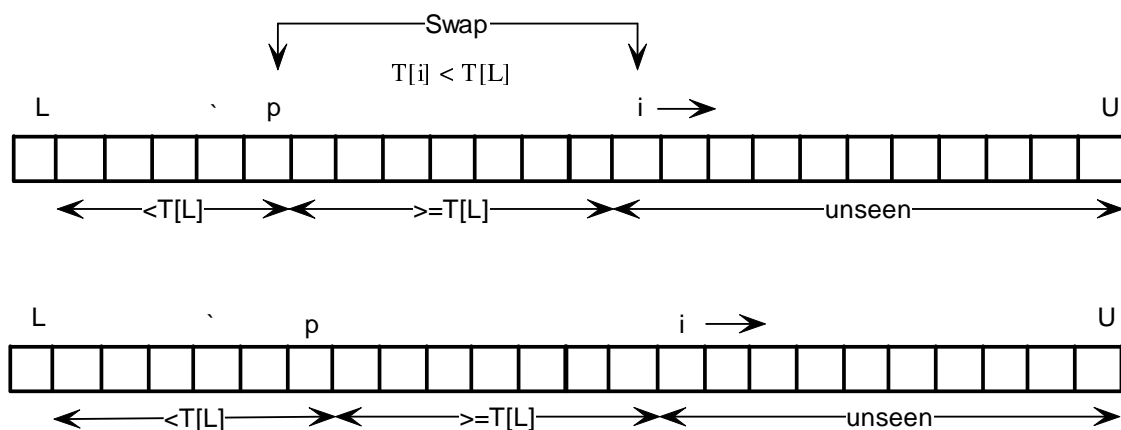


Figure 9.7. Lomuto's Partitioning.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	swap	
3		1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	(2,2)
p		i															
3		1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	(3,4)
		p		i													
3		1	1	4	5	9	2	6	5	3	5	8	9	7	9	3	(4,7)
			p			i											
3		1	1	2	5	9	4	6	5	3	5	8	9	7	9	3	(end)
				p												i	
2		1	1	3	5	9	4	6	5	3	5	8	9	7	9	3	(1,4)
				p													(p=4)

At each stage we have the array partitioned into 3 parts : $< T[L]$, $\geq T[L]$, and 'unseen'. The index p marks the end of the $< T[L]$ part and the index i marks the start of the 'unseen' part. The Pascal function below implements this method.

```
function PartitionL ( VAR T:ArrNType; L,U:IndxType):IndxType;
```

```
VAR
    i, p : IndxType;
BEGIN
    p := L;
    FOR i := L TO U DO BEGIN
        IF (T[i] < T[L]) THEN BEGIN
            Inc(p);
            Swap(T[p],T[i]);
        END{ if }
    END;{ for }
    PartitionL := p;
END;{ func PartitionL }
```

Analysis of Lomuto's Partitioning Method. There is a single FOR-loop so the number of steps is $U - L + 1 = O(n)$ for $L = 1$ and $U = n$.

Exercise 9.5.1 : *Comparing Apples and Oranges (and Pears)* Given an array which contains apples, oranges, and pears in random order, how can we sort this array so that the apples are first, oranges second and pears last, in $O(n)$ time, using comparisons and exchanges only.

We can now write the complete QuickSort procedure.

```

procedure QuickSort(var T : ArrType; L, U : IndxType)
VAR
    p : IndxType;
BEGIN
    IF L > U DO BEGIN
        p := PartitionL(T, L, U);
        QuickSort(T, L, p-1);
        QuickSort(T, p+1, U);
    END{ if }
END; { proc QuickSort}

```

Analysis of QuickSort. The table length is reduced by a factor of 2 at each stage. How many times can we divide n by 2 before we have a table of size 1? $\log_2 n$ times. If we let $T(n)$ be the time to QuickSort n elements and cn the time to partition n elements then we have the equation

$$T(n) = cn + 2T(n/2),$$

assuming the table is split evenly at each partition. Successively substituting we get

$$\begin{aligned}
 T(n) &= 1cn + 2T(n/2) = 1cn + 2(cn/2 + 2T(n/4)) \\
 &= 2cn + 4T(n/4) = 2cn + 4(cn/4 + 2T(n/8)) \\
 &= 3cn + 8T(n/8)
 \end{aligned}$$

·
·
·

$$T(n) = kcn + 2^k T(n/2^k) \quad \text{for any } k.$$

Now $k = \log_2 n$ is the number of times we halve the table until it is reduced to sub-tables of size 1. $T(1) = 0$, the time to sort 1 element, so we have

$$T_{qs}(n) = c_{qs} n \log_2 n$$

The assumption that the table is evenly partitioned is important : if the table is already sorted then the partition is $1 : n - 1$ and $T_{qs}(n) = cn + T_{qs}(1) + T_{qs}(n - 1)$. The solution of this recurrence is $T_{qs}(n) = O(n^2)$.

Exercise 9.5.1 : If the table is partitioned in the ratio 1:9 at each step what is $T_{qs}(n)$? If it is partitioned in the ratio $\alpha : 1 - \alpha$, $0 < \alpha < 1$, what is $T_{qs}(n)$?

Modifications of QuickSort.

1. Use a random partitioning element.
2. Use a median-of-three partitioning element.
3. Use *InsertSort* on sub-problems of size less than m .
4. Ignore sub-problems of size less than m . Do one *InsertSort* on the whole table at the end.

9.5.3 Heap Sort.

This is Select Sort with the table $T[1..n]$ arranged as a *Heap*. A heap is a particular implementation of the the more general ADT *Priority Queue*.

ADT Priority Queue. This is a set of elements S totally ordered by some *key* value. The operations are $Insert(x, S)$ and $DeleteMin(S)$ or $DeleteMax(S)$.

If we represent the set S as an unordered array then $Insert$ is $O(1)$ and $DeleteMin$ is $O(n)$. If the array is sorted then these become $O(n)$ and $O(1)$, respectively.

We can do much better if we represent S as a *partially ordered tree*. This is a tree in which *the key value of any node is less than or equal to those of its children*. This implies the *Partially Ordered Tree Property* : the minimum of any subtree is at the root of the subtree. This means that the minimum of S is at the root of the partially ordered tree representing S .

The most efficient way to represent a partially ordered tree is a *Heap*. A heap is a complete binary tree arranged so that

$$key(i) \leq key(LChild(i)) \text{ and } key(RChild(i)).$$

Example : $S = \{50, 26, 49, 21, 71, 21, 20, 48, 11, 56\}..$

Before we implement the $Insert$ and $DeleteMin$ operations we need two ancilliary operations : $SiftUp(i, S, size)$ moves the contents of node i up to it correct position in the heap and $SiftDown(i, S, size)$ moves the contents of node i down to its correct position. These operations affect only one path in the heap. Hence they require $O(\log_2 n)$ time.

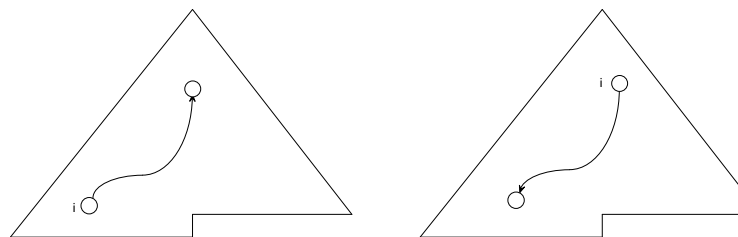


Figure 9.8. SiftUp and SiftDown Operations.

Exercise 9.5.1 : Implement the $SiftUp$ and $SiftDown$ operations in some programming language.

We are now in a position to implement the $Insert$ and $DeleteMin$ operations.

$Insert(x, S)$ is easily implemented by inserting x in the next available space in the heap array S . $SiftUp$ then moves x up to its correct position. The algorithm is as follows :

```

ALGORITHM Insert (x, S, size)
    Inc(size)
    S[size] := x
    SiftUp(size, S)
ENDALG Insert

```

```

ALGORITHM DeleteMin (S, size) : element
  DeleteMin := S[1]
  S[1] := S[size]
  Dec(size)
  SiftDown(1,S,size)
ENDALG DeleteMin

```

It is obvious that both of these operations are $O(\log_2 n)$.

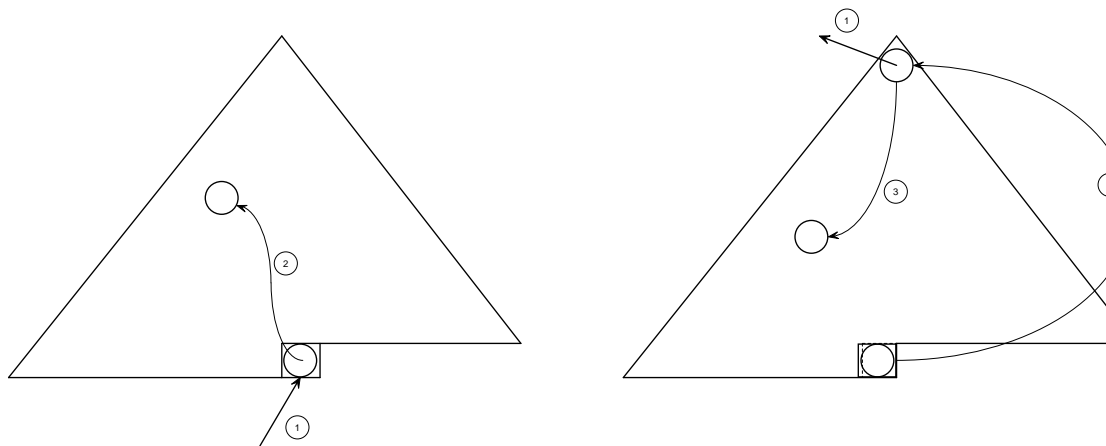


Figure 9.9. The *Insert* and *DeleteMin* Operations.

We may create the initial heap in two ways. The first is straight-forward : Read in each x and use $Insert(x, S, size)$, incrementing $size$ for each x read in. This method is $O(n \log_2 n)$. A more elegant and efficient method is to assume that all elements are in the array S , in any order, and then apply the following algorithm :

```

algorithm MakeHeap (S, size)
  FOR i := size div 2 DOWNTO 1 DO
    SiftDown(i,S,size)
  ENDFOR
ENDALG MakeHeap

```

This algorithm requires $n/2$ SiftDown operations and so the complexity is, on the face of it, $O(n \log_2 n)$. Surprisingly, a more careful analysis shows that this algorithm is $O(n)$. We now do this analysis.

Complexity of MakeHeap. We wish to find a formula for $T_{mh}(n)$, the number of exchanges required to make a heap of n nodes. Assume, for convenience, that all levels of the heap are full. There is 1 node at level 0, 2 at level 1, 4 at level 2, and 2^i at level i . Hence, for a heap with n nodes and l levels we have the formula

$$n = \sum_{i=0}^l 2^i = 2^{l+1} - 1.$$

Thus we see that a heap with n nodes has

$$l = \log_2(n + 1) - 1.$$

Now, in MakeHeap, SiftDown operates at all levels except the lowest, l , exchanging node values until each finds its proper place. This means that

nodes at level l require 0 exchanges
 nodes at level $l - 1$ require 1 exchanges
 nodes at level $l - 2$ require 2 exchanges
 nodes at level $l - i$ require i exchanges
 nodes at level 0 require l exchanges

There are 2^{l-i} nodes at level $l - i$ and each of these requires i interchanges, at most. Thus there are, at most, $i2^{l-i}$ interchanges at level i . Hence, the total number of interchanges is

$$T_{mh}(n) = \sum_{i=0}^l i2^{l-i} = \sum_{i=0}^l i2^l 2^{-i} = 2^l \sum_{i=0}^l i2^{-i}.$$

Consider the expression $\sum_{i=0}^l i2^{-i}$ in the formula above. This is bounded above by $\sum_{i=0}^{\infty} i2^{-i}$. This is a standard summation whose value is obtained as follows :

$$\begin{aligned} S &= \sum_{i=0}^{\infty} i \frac{1}{2^i} = \sum_{i=0}^{\infty} (i+1) \frac{1}{2^{i+1}} \\ &= \frac{1}{2} \sum_{i=0}^{\infty} i \frac{1}{2^i} + \frac{1}{2} \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= \frac{1}{2} S + \frac{1}{2} \left(\frac{1}{1 - 1/2} \right) \\ &= \frac{1}{2} S + 1. \end{aligned}$$

Thus

$$S = \sum_{i=0}^{\infty} i2^{-i} = 2.$$

We now plug this into the main formula to get the following upper bound :

$$T_{mh}(n) = 2^l \sum_{i=0}^l i2^{-i} < 2^l S = 2^{l+1} = n + 1.$$

Thus we have shown that MakeHeap can be performed in (usually less than) n exchanges.

```

procedure HeapSort(VAR T : ArrType; n : IndxType);
VAR
    i, j : IndxType;
    temp : ElemType;
BEGIN
    FOR i := size DIV 2 DOWNTO 1 DO
        SiftDown(i,T,size)
    END;
    FOR i := n DOWNTO 2 DO BEGIN
        temp := T[i];
        T[i] := T[1];
        T[1] := temp
        SiftDown(1,T,i-1)
    END; { for }
END; { proc HeapSort}

```

Analysis of HeapSort. *MakeHeap* is $O(n)$ and each *SiftDown* is $O(\log_2 n)$, and is called $n - 1$ times. Hence

$$T_{hs}(n) = c_{hs}n \log_2 n$$

It should be noted that, unlike *QuickSort*, this complexity holds for the worst case. As such *HeapSort* is a more robust algorithm than *QuickSort*.

9.6 Average Complexities of Sorting Algorithms

The analyses of the sorting algorithms above have been for the worst case. Worst-case analyses are relatively easy, as are best-case analyses. Deriving the average complexity of algorithms is much more difficult and often a level of mathematics and probability theory that is beyond the average computer scientist or engineer. Another, and perhaps more important difficulty is : what set of inputs should we average over? That is, what sample space of inputs should we choose and what is its probability distribution function? The average case analyses of sorting algorithms usually assume that the space is all permutations of the numbers $1 \dots n$ and that each permutation is equally likely, i.e., the probability of each permutation is $1/n!$. While this is a mathematically and empirically convenient assumption (random permutations are easy to generate), real sorting problems are probably not random. Indeed much sorting in business applications is done on nearly sorted inputs. While this may not be a major problem in analysing and testing sorting algorithms, it certainly is a problem when considering graph algorithms. For example, testing Shortest Path algorithms on random networks is misleading : real networks are not random. (see the chapter on Shortest Path algorithms).

Despite all of the above we now list the average-case complexity of the algorithms we have discussed so far. We also give the best- and worst-case complexities for comparison.

Algorithm	Best $C(n)$	Average $C(n)$	Average $M(n)$	Worst $C(n)$
Selection	$n^2/2$	$n^2/2$	n	$n^2/2$
Bubble	n	$n^2/2$	$n^2/2$	$n^2/2$
Insertion		$n^2/4$	$n^2/8$	$n^2/2$
Merge		$n \log_2 n$	$n \log_2 n$	n
Quick		$1.38n \log_2 n$		$O(n^2)$
Heap		$< 2n \log_2 n$		$O(n \log_2 n)$

Although *Merge Sort* appears best in the table above, there is a catch : it requires $O(n)$ extra space. Likewise, *Quick Sort* requires a stack of $O(n)$ extra space in the worst case, but this can be reduced to $O(\log_2 n)$ on average. A conservative best choice would be *Heap Sort* : it has a guaranteed running time of $O(n \log_2 n)$ for all inputs and requires no extra space. It can be seen that *Insertion Sort* is the best of the $O(n^2)$ algorithms on average. It is also faster than all other algorithms for small tables with $n = [15..30]$.

Exercise 9.6.1 : Write an algorithm to sort in $O(n \log_2 n)$ time using a *Binary Search Tree*.

9.7 $O(n)$ Sorting Algorithms

We have seen that any sorting algorithm using comparisons only must make at least $O(n \log_2 n)$ comparisons in the worst case. However, if we relax the ‘comparisons only’ rule or if the input data is special then we may be able to sort in $O(n)$ time—this is the very least any algorithm must do because just to read n elements requires $O(n)$ time.

Example : Given. a random permutation of the integers $1 \dots n$, sort these into ascending order. The following algorithm sorts the random permutation in exactly n steps :

algorithm <i>LinearSort</i> (T, S, n)
<pre> FOR i := 1 TO n DO S[T[i]] := T[i] ENDFOR endalg LinearSort </pre>

To see that this works correctly consider the integer $T[i]$ whose value is j . When this algorithm finishes j is in $S[j]$ and so $S[1..n]$ is the identity permutation $\langle 1, 2, \dots, n \rangle$. Notice that we sorted the random permutation in n steps with no comparisons. What we did use was *address arithmetic* in the form of *indirect indexing*.

The example above shows that if there is something special about the input data then we may be able to exploit this fact. Can we generalize the *Linear Sort* algorithm? Let's see.

9.7.1 Bucket Sorting.

Before we discuss the various methods of bucket sorting let us consider a related problem :

Histogram Construction. Given a set of N numbers in the range $[N_{min}..N_{max}]$, divide this interval into k equal sub-ranges and count the number of numbers that fall into each interval. This is called a *frequency count* and a histogram is a graphical representation of this. A typical and topical example is constructing a histogram of student marks on a given exam. The program below shows how to do this.

```

program Histogram;
  {data stucture declarations}
  BEGIN
    for i := 0 to 9 do begin
      Hist[i] := 0;
    end { FOR i }

    for i := 1 to NoStuds do begin
      BoxNo := (Mark[i] div 10);
      Hist[BoxNo] := Hist[BoxNo] + 1;
    end;{FOR}

    {'Graphical Output'}
    for i := 0 to 9 do begin
      WriteLn;
      Write(i); Write('|');
      for j := 1 to Hist[i] do begin
        Write('*');
      end;{for j}
      WriteLn(Hist[i]);
    end;{for i}
  end. {Prog Histogram}

```

The output from this program should look like this :

```

0|***3
1|*****5
2|*****9
3|****4
4|****4
5|*****12
6|*****16
7|*****6
8|****4
9|*1

```

We can generalize this easily so that the program works for any range of numbers and any number of sub-intervals. The algorithm *Histogram* below does this.

```
algorithm Histogram( $T, n, k, H$ )
```

```
 $N_{min} := Min(T, n);$   
 $N_{max} := Max(T, n);$   
 $IntWidth := (N_{max} - N_{min}) div k;$   
for  $i := 1$  to  $n$  do begin  
     $BoxNo := Mark[i] div IntWidth;$   
     $Hist[BoxNo] := Hist[BoxNo] + 1;$   
end;{FOR}  
endalg{ Histogram }
```

We now consider bucket sorting. The general idea is to divide the numbers to be sorted into k buckets, sort each bucket, and then concatenate the buckets into one big sorted bucket.

Assume that the input is a set of n numbers that is uniformly distributed over the range $[N_{min}..N_{max}]$.

We do not assume the numbers are distinct. Now devise a function that *monotonely* maps numbers in the range $[N_{min}..N_{max}]$ into numbers in the range $[1..n]$. That is, *hash* $x \in [N_{min}..N_{max}] \rightarrow i \in [1..n]$.

Bucket Sort with Linked Lists.

Lomuto, and Apples, Oranges & Pears, and Dijkstra .

Bucket Sort in place.

Radix Sorting.

9.8 Selection and Order Statistics

We wish to select the k th smallest element in an array T , i.e., find the element that would be in the k th position if T were sorted in ascending order. $T[k]$ is called the k th *order statistic* of T .

There are various ways to do this.

1. The first and most obvious way is to sort $T[1..n]$ and then the k th smallest is $T[k]$. The complexity of this method is $O(n \log n)$.
2. Make T into a min-heap and do k *DeleteMin* s. This has complexity $O(n) + kO(\log n) = O(k \log n)$.
3. Use a modification of Hoare's *QuickSort* algorithm, as shown below. This method repeatedly partitions until the k th smallest element is in its correct sorted position, i.e., k .

algorithm *Select* (L, U, T, k)

```

p := Partition(L, U, T)
if p = k then return(T[p])
else if p < k then Select(p+1, U, T, k-p)
else Select(L, p-1, T, k)
endalg Select

```

Analysis of Select. Note that unlike *QuickSort* only one recursive call is made in this algorithm. Hence

$$\begin{aligned}
T_s(n) &= cn + T_s(n/2) \\
&= cn + (cn/2 + T_s(n/4)) \\
&= cn + cn/2 + cn/4 + T_s(n/8) \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&= cn + cn/2 + cn/4 + \cdots + cn/2^{k-1} + T_s(n/2^k) \\
&= cn \sum_{i=0}^{k-1} 1/2^i + T_s(1), \quad k = \log_2 n \\
&< cn \sum_{i=0}^{\infty} 1/2^i \\
&= 2cn
\end{aligned}$$

Thus $T_s(n)$ is $O(n)$.