1.3 Binary Search

Henceforth, in the context of searching and sorting problems, we will assume that the elements are drawn from a linearly ordered set, for example the set of integers. This will also be the case for similar problems, like finding the median, the kth smallest element, and so forth. Let $A[1..n]$ be a sequence of n elements. Consider the problem of determining whether a given element x is in A. This problem can be rephrased as follows. Find an index $j, 1 \leq j \leq n$, such that $x = A[j]$ if x is in A, and $j = 0$ otherwise. A straightforward approach is to scan the entries in A and compare each entry with x. If after j comparisons, $1 \leq j \leq n$, the search is *successful*, i.e., $x = A[j]$, j is returned; otherwise a value of 0 is returned indicating an *unsuccessful* search. This method is referred to as *sequential search*. It is also called *linear search*, as the maximum number of element comparisons grows linearly with the size of the sequence. The algorithm is shown as Algorithm LINEARSEARCH.

Algorithm 1.1 LINEARSEARCH
Input: An array $A[1..n]$ of n elements and an element x.
Output: j if $x = A[j], 1 \leq j \leq n$, and 0 otherwise.

```
1. j ← 1
2. while (j < n) and (x ≠ A[j])
3.     j ← j + 1
4. end while
5. if x = A[j] then return j else return 0
```

Intuitively, scanning all entries of A is inevitable if no more information about the ordering of the elements in A is given. If we are also given that the elements in A are sorted, say in nondecreasing order, then there is a much more efficient algorithm. The following example illustrates this efficient search method.

Example 1.1 Consider searching the array

$$A[1..14] = \boxed{1}\boxed{4}\boxed{5}\boxed{7}\boxed{8}\boxed{9}\boxed{10}\boxed{12}\boxed{15}\boxed{22}\boxed{23}\boxed{27}\boxed{32}\boxed{35}.$$

In this instance, we want to search for element $x = 22$. First, we compare x with the middle element $A[\lfloor (1 + 14)/2 \rfloor] = A[7] = 10$. Since $22 > A[7]$, and since it is

known that $A[i] \leq A[i+1], 1 \leq i < 14$, x cannot be in $A[1..7]$, and therefore this portion of the array can be discarded. So, we are left with the subarray

$$A[8..14] = \boxed{12}\boxed{15}\boxed{22}\boxed{23}\boxed{27}\boxed{32}\boxed{35}.$$

Next, we compare x with the middle of the remaining elements $A[\lfloor (8+14)/2 \rfloor] = A[11] = 23$. Since $22 < A[11]$, and since $A[i] \leq A[i+1], 11 \leq i < 14$, x cannot be in $A[11..14]$, and therefore this portion of the array can also be discarded. Thus, the remaining portion of the array to be searched is now reduced to

$$A[8..10] = \boxed{12}\boxed{15}\boxed{22}.$$

Repeating this procedure, we discard $A[8..9]$, which leaves only one entry in the array to be searched, that is $A[10] = 22$. Finally, we find that $x = A[10]$, and the search is successfully completed.

In general, let $A[low..high]$ be a nonempty array of elements sorted in nondecreasing order. Let $A[mid]$ be the middle element, and suppose that $x > A[mid]$. We observe that if x is in A, then it must be one of the elements $A[mid+1], A[mid+2], \ldots, A[high]$. It follows that we only need to search for x in $A[mid+1..high]$. In other words, the entries in $A[low..mid]$ are discarded in subsequent comparisons since, by assumption, A is sorted in nondecreasing order, which implies that x cannot be in this half of the array. Similarly, if $x < A[mid]$, then we only need to search for x in $A[low..mid-1]$. This results in an efficient strategy which, because of its repetitive halving, is referred to as *binary search*. Algorithm BINARYSEARCH gives a more formal description of this method.

Algorithm 1.2 BINARYSEARCH
Input: An array $A[1..n]$ of n elements sorted in nondecreasing order and an element x.
Output: j if $x = A[j], 1 \leq j \leq n$, and 0 otherwise.

```
1. low ← 1;   high ← n;   j ← 0
2. while (low ≤ high) and (j = 0)
3.     mid ← ⌊(low + high)/2⌋
4.     if x = A[mid] then j ← mid
5.     else if x < A[mid] then high ← mid − 1
6.     else low ← mid + 1
7. end while
8. return j
```

1.3.1 *Analysis of the binary search algorithm*

Henceforth, we will assume that each three-way comparison (if-then-else) counts as one comparison. Obviously, the minimum number of comparisons is 1, and it is achievable when the element being searched for, x, is in the middle position of the array. To find the maximum number of comparisons, let us first consider applying binary search on the array $\boxed{2}\boxed{3}\boxed{5}\boxed{8}$. If we search for 2 or 5, we need two comparisons, whereas searching for 8 costs three comparisons. Now, in the case of unsuccessful search, it is easy to see that searching for elements such as 1, 4, 7 or 9 takes 2, 2, 3 and 3 comparisons, respectively. It is not hard to see that, in general, the algorithm always performs the maximum number of comparisons whenever x is greater than or equal to the maximum element in the array In this example, searching for any element greater than or equal to 8 costs three comparisons. Thus, to find the maximum number of comparisons, we may assume without loss of generality that x is greater than or equal to $A[n]$.

Example 1.2 Suppose that we want to search for $x = 35$ or $x = 100$ in

$$A[1..14] = \boxed{1}\boxed{4}\boxed{5}\boxed{7}\boxed{8}\boxed{9}\boxed{10}\boxed{12}\boxed{15}\boxed{22}\boxed{23}\boxed{27}\boxed{32}\boxed{35}.$$

In each iteration of the algorithm, the bottom half of the array is discarded until there is only one element:

$$\boxed{12}\boxed{15}\boxed{22}\boxed{23}\boxed{27}\boxed{32}\boxed{35} \rightarrow \boxed{27}\boxed{32}\boxed{35} \rightarrow \boxed{35}.$$

Therefore, to compute the *maximum* number of element comparisons performed by Algorithm BINARYSEARCH, we may assume that x is greater than or equal to all elements in the array to be searched. To compute the number of remaining elements in $A[1..n]$ in the second iteration, there are two cases to consider according to whether n is even or odd. If n is even, then the number of entries in $A[mid + 1..n]$ is $n/2$; otherwise it is $(n-1)/2$. Thus, in both cases, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2 \rfloor$.

Similarly, the number of remaining elements to be searched in the third iteration is $\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/4 \rfloor$ (see Eq. 2.3 on page 71).

In general, in the jth pass through the while loop, the number of remaining elements is $\lfloor n/2^{j-1} \rfloor$. The iteration is continued until either x is found or the size of the subsequence being searched reaches 1, whichever

occurs first. As a result, the maximum number of iterations needed to search for x is that value of j satisfying the condition

$$\lfloor n/2^{j-1} \rfloor = 1.$$

By the definition of the floor function, this happens exactly when

$$1 \leq n/2^{j-1} < 2,$$

or

$$2^{j-1} \leq n < 2^j,$$

or

$$j - 1 \leq \log n < j.^{\dagger}$$

Since j is integer, we conclude that

$$j = \lfloor \log n \rfloor + 1.$$

Alternatively, the performance of the binary search algorithm can be described in terms of a *decision tree*, which is a binary tree that exhibits the behavior of the algorithm. Figure 1.1 shows the decision tree corresponding to the array given in Examples 1.1. The darkened nodes are those compared against x in Examples 1.1.

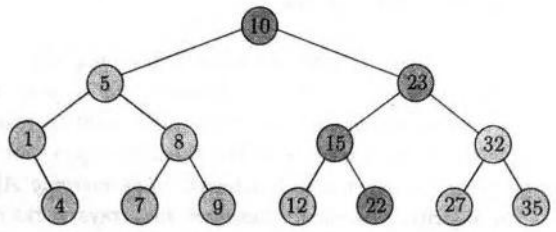


Fig. 1.1 A decision tree that shows the behavior of binary search.

Note that the decision tree is a function of the *number* of the elements in the array only. Figure 1.2 shows two decision trees corresponding to two arrays of sizes 10 and 14, respectively. As implied by the two figures, the

[†]Unless otherwise stated, all logarithms in this book are to the base 2. The natural logarithm of x will be denoted by $\ln x$.

maximum number of comparisons in both trees is 4. In general, the maximum number of comparisons is one plus the height of the corresponding decision tree (see Sec. 3.5 for the definition of height). Since the height of such a tree is $\lfloor \log n \rfloor$, we conclude that the maximum number of comparisons is $\lfloor \log n \rfloor + 1$. We have in effect given two proofs of the following theorem:

Theorem 1.1 The number of comparisons performed by Algorithm BINARYSEARCH on a sorted array of size n is at most $\lfloor \log n \rfloor + 1$.

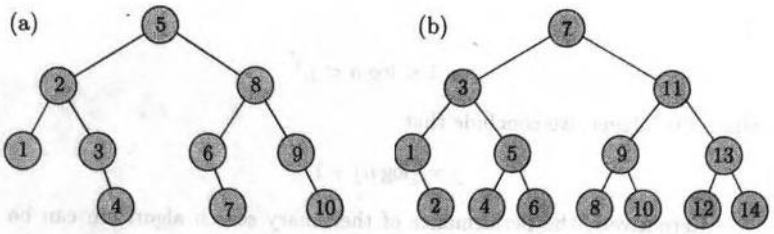


Fig. 1.2 Two decision trees corresponding to two arrays of sizes 10 and 14.

1.4 Merging Two Sorted Lists

Suppose we have an array $A[1..m]$ and three indices p, q and r, with $1 \leq p \leq q < r \leq m$, such that both the subarrays $A[p..q]$ and $A[q+1..r]$ are individually sorted in nondecreasing order. We want to rearrange the elements in A so that the elements in the subarray $A[p..r]$ are sorted in nondecreasing order. This process is referred to as *merging* $A[p..q]$ with $A[q+1..r]$. An algorithm to merge these two subarrays works as follows. We maintain two pointers s and t that initially point to $A[p]$ and $A[q+1]$, respectively. We prepare an empty array $B[p..r]$ which will be used as a temporary storage. Each time, we compare the elements $A[s]$ and $A[t]$ and append the smaller of the two to the auxiliary array B; if they are equal we will choose to append $A[s]$. Next, we update the pointers: If $A[s] \leq A[t]$, then we increment s, otherwise we increment t. This process ends when $s = q+1$ or $t = r+1$. In the first case, we append the remaining elements $A[t..r]$ to B, and in the second case, we append $A[s..q]$ to B. Finally, the

array $B[p..r]$ is copied back to $A[p..r]$. This procedure is given in Algorithm MERGE.

Algorithm 1.3 MERGE
Input: An array $A[1..m]$ of elements and three indices p, q and r, with $1 \leq p \leq q < r \leq m$, such that both the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted individually in nondecreasing order.
Output: $A[p..r]$ contains the result of merging the two subarrays $A[p..q]$ and $A[q+1..r]$.

```
1.  comment: B[p..r] is an auxiliary array.
2.  s ← p;   t ← q + 1;   k ← p
3.  while s ≤ q and t ≤ r
4.      if A[s] ≤ A[t] then
5.          B[k] ← A[s]
6.          s ← s + 1
7.      else
8.          B[k] ← A[t]
9.          t ← t + 1
10.     end if
11.     k ← k + 1
12. end while
13. if s = q + 1 then B[k..r] ← A[t..r]
14. else B[k..r] ← A[s..q]
15. end if
16. A[p..r] ← B[p..r]
```

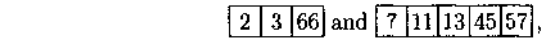
Let n denote the size of the array $A[p..r]$ in the input to Algorithm MERGE, i.e., $n = r - p + 1$. We want to find the number of comparisons that are needed to rearrange the entries of $A[p..r]$. It should be emphasized that from now on when we talk about the number of comparisons performed by an algorithm, we mean *element comparisons*, i.e., the comparisons involving objects in the input data. Thus, all other comparisons, e.g. those needed for the implementation of the **while** loop, will be excluded.

Let the two subarrays be of sizes n_1 and n_2, where $n_1 + n_2 = n$. The least number of comparisons happens if each entry in the smaller subarray is less than all entries in the larger subarray. For example, to merge the two subarrays

| 2 | 3 | 6 | and | 7 | 11 | 13 | 45 | 57 |,

the algorithm performs only three comparisons. On the other hand, the

number of comparisons may be as high as $n-1$. For example, to merge the two subarrrays



seven comparisons are needed. It follows that the number of comparisons done by Algorithm MERGE is at least n_1 and at most $n-1$.

Observation 1.1 The number of element comparisons performed by Algorithm MERGE to merge two nonempty arrays of sizes n_1 and n_2, respectively, where $n_1 \leq n_2$, into one sorted array of size $n = n_1 + n_2$ is between n_1 and $n-1$. In particular, if the two array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons needed is between $\lfloor n/2 \rfloor$ and $n-1$.

How about the number of *element assignments* (again here we mean assignments involving input data)? At first glance, one may start by looking at the **while** loop, the **if** statements, etc. in order to find out how the algorithm works and then compute the number of element assignments. However, it is easy to see that each entry of array B is assigned exactly once. Similarly, each entry of array A is assigned exactly once, when copying B back into A. As a result, we have the following observation:

Observation 1.2 The number of element assignments performed by Algorithm MERGE to merge two arrays into one sorted array of size n is exactly $2n$.

1.5 Selection Sort

Let $A[1..n]$ be an array of n elements. A simple and straightforward algorithm to sort the entries in A works as follows. First, we find the minimum element and store it in $A[1]$. Next, we find the minimum of the remaining $n-1$ elements and store it in $A[2]$. We continue this way until the second largest element is stored in $A[n-1]$. This method is described in Algorithm SELECTIONSORT.

It is easy to see that the number of element comparisons performed by the algorithm is exactly

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

Algorithm 1.4 SELECTIONSORT
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for i ← 1 to n − 1
2.     k ← i
3.     for j ← i + 1 to n      {Find the ith smallest element.}
4.         if A[j] < A[k] then k ← j
5.     end for
6.     if k ≠ i then interchange A[i] and A[k]
7. end for
```

It is also easy to see that the number of element interchanges is between 0 and $n - 1$. Since each interchange requires three element assignments, the number of element assignments is between 0 and $3(n - 1)$.

Observation 1.3 The number of element comparisons performed by Algorithm SELECTIONSORT is $n(n-1)/2$. The number of element assignments is between 0 and $3(n - 1)$.

1.6 Insertion Sort

As stated in Observation 1.3 above, the number of comparisons performed by Algorithm SELECTIONSORT is *exactly* $n(n - 1)/2$ regardless of how the elements of the input array are ordered. Another sorting method in which the number of comparisons depends on the order of the input elements is the so-called INSERTIONSORT. This algorithm, which is shown below, works as follows. We begin with the subarray of size 1, $A[1]$, which is already sorted. Next, $A[2]$ is inserted before or after $A[1]$ depending on whether it is smaller than $A[1]$ or not. Continuing this way, in the ith iteration, $A[i]$ is inserted in its proper position in the sorted subarray $A[1..i - 1]$. This is done by scanning the elements from index $i - 1$ down to 1, each time comparing $A[i]$ with the element at the current position. In each iteration of the scan, an element is shifted one position up to a higher index. This process of scanning, performing the comparison and shifting continues until an element less than or equal to $A[i]$ is found, or when all the sorted sequence so far is exhausted. At this point, $A[i]$ is inserted in its proper position, and the process of inserting element $A[i]$ in its proper

place is complete.

Algorithm 1.5 INSERTIONSORT
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for i ← 2 to n
2.     x ← A[i]
3.     j ← i − 1
4.     while (j > 0) and (A[j] > x)
5.         A[j + 1] ← A[j]
6.         j ← j − 1
7.     end while
8.     A[j + 1] ← x
9. end for
```

Unlike Algorithm SELECTIONSORT, the number of element comparisons done by Algorithm INSERTIONSORT depends on the order of the input elements. It is easy to see that the number of element comparisons is minimum when the array is already sorted in nondecreasing order. In this case, the number of element comparisons is exactly $n-1$, as each element $A[i]$, $2 \leq i \leq n$, is compared with $A[i-1]$ only. On the other hand, the maximum number of element comparisons occurs if the array is already sorted in decreasing order and all elements are distinct. In this case, the number of element comparisons is

$$\sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2},$$

as each element $A[i]$, $2 \leq i \leq n$, is compared with each entry in the subarray $A[1..i-1]$. This number coincides with that of Algorithm SELECTIONSORT.

Observation 1.4 The number of element comparisons performed by Algorithm INSERTIONSORT is between $n-1$ and $n(n-1)/2$. The number of element assignments is equal to the number of element comparisons plus $n-1$.

Notice the correlation of element comparisons and assignments in Algorithm INSERTIONSORT. This is in contrast to the independence of the number of element comparisons in Algorithm SELECTIONSORT related to data arrangement.

1.7 Bottom-Up Merge Sorting

The two sorting methods discussed thus far are both inefficient in the sense that the number of operations required to sort n elements is proportional to n^2. In this section, we describe an efficient sorting algorithm that performs much fewer element comparisons. Suppose we have the following array of eight numbers that we wish to sort:

9	4	5	2	1	7	4	6

Consider the following method for sorting these numbers (see Fig. 1.3).

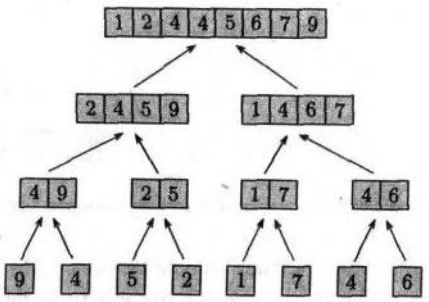


Fig. 1.3 Example of bottom-up merge sorting.

First, we divide the input elements into four pairs and merge each pair into one 2-element sorted sequence. Next, we merge each two consecutive 2-element sequences into one sorted sequence of size four. Finally, we merge the two resulting sequences into the final sorted sequence as shown in the figure.

In general, let A be an array of n elements that is to be sorted. We first merge $\lfloor n/2 \rfloor$ consecutive pairs of elements to yield $\lfloor n/2 \rfloor$ sorted sequences of size 2. If there is one remaining element, then it is passed to the next iteration. Next, we merge $\lfloor n/4 \rfloor$ pairs of consecutive 2-element sequences to yield $\lfloor n/4 \rfloor$ sorted sequences of size 4. If there are one or two remaining elements, then they are passed to the next iteration. If there are three elements left, then two (sorted) elements are merged with one element to form a 3-element sorted sequence. Continuing this way, in the jth iteration, we merge $\lfloor n/2^j \rfloor$ pairs of sorted sequences of size 2^{j-1} to yield $\lfloor n/2^j \rfloor$ sorted sequences of size 2^j. If there are k remaining elements, where $1 \leq k \leq 2^{j-1}$,

then they are passed to the next iteration. If there are k remaining elements, where $2^{j-1} < k < 2^j$, then these are merged to form a sorted sequence of size k.

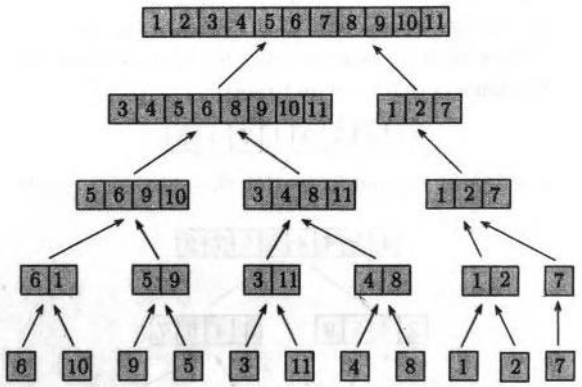


Fig. 1.4 Example of bottom-up merge sorting when n is not a power of 2.

Algorithm BOTTOMUPSORT implements this idea. The algorithm maintains the variable s which is the size of sequences to be merged. Initially, s is set to 1, and is doubled in each iteration of the outer **while** loop. $i + 1$, $i + s$ and $i + t$ define the boundaries of the two sequences to be merged. Step 8 is needed in the case when n is not a multiple of t. In this case, if the number of remaining elements, which is $n - i$, is greater than s, then one more merge is applied on a sequence of size s and the remaining elements.

Example 1.3 Figure 1.4 shows an example of the working of the algorithm when n is not a power of 2. The behavior of the algorithm can be described as follows.

(1) In the first iteration, $s = 1$ and $t = 2$. Five pairs of 1-element sequences are merged to produce five 2-element sorted sequences. After the end of the inner **while** loop, $i + s = 10 + 1 \not< n = 11$, and hence no more merging takes place.

(2) In the second iteration, $s = 2$ and $t = 4$. Two pairs of 2-element sequences are merged to produce two 4-element sorted sequences. After the end of the inner **while** loop, $i + s = 8 + 2 < n = 11$, and hence one sequence of size $s = 2$ is merged with the one remaining element to produce a 3-element sorted sequence.

(3) In the third iteration, $s = 4$ and $t = 8$. One pair of 4-element sequences are merged to produce one 8-element sorted sequence. After the end of the inner

Algorithm 1.6 BOTTOMUPSORT
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. t ← 1
2. while t < n
3.     s ← t;    t ← 2s;    i ← 0
4.     while i + t ≤ n
5.         MERGE(A, i + 1, i + s, i + t)
6.         i ← i + t
7.     end while
8.     if i + s < n then MERGE(A, i + 1, i + s, n)
9. end while
```

while loop, $i + s = 8 + 4 \not< n = 11$ and hence no more merging takes place.

(4) In the fourth iteration, $s = 8$ and $t = 16$. Since $i + t = 0 + 16 \not\leq n = 11$, the inner **while** loop is not executed. Since $i + s = 0 + 8 < n = 11$, the condition of the **if** statement is satisfied, and hence one merge of 8-element and 3-element sorted sequences takes place to produce a sorted sequence of size 11.

(5) Since now $t = 16 > n$, the condition of the outer **while** loop is not satisfied, and consequently the algorithm terminates.

1.7.1 *Analysis of bottom-up merge sorting*

Now, we compute the number of element comparisons performed by the algorithm for the special case when n is a power of 2. In this case, the outer **while** loop is executed $k = \log n$ times, once for each level in the sorting tree except the topmost level (see Fig. 1.3). Observe that since n is a power of 2, $i = n$ after the execution of the inner **while** loop, and hence Algorithm MERGE will never be invoked in Step 8. In the first iteration, there are $n/2$ comparisons. In the second iteration, $n/2$ sorted sequences of two elements each are merged in pairs. The number of comparisons needed to merge each pair is either 2 or 3. In the third iteration, $n/4$ sorted sequences of four elements each are merged in pairs. The number of comparisons needed to merge each pair is between 4 and 7. In general, in the jth iteration of the while loop, there are $n/2^j$ merge operations on two subarrays of size 2^{j-1} and it follows, by Observation 1.1, that the number of comparisons needed in the jth iteration is between $(n/2^j)2^{j-1}$ and $(n/2^j)(2^j - 1)$. Thus, if we

let $k = \log n$, then the number of element comparisons is at least

$$\sum_{j=1}^{k} \left(\frac{n}{2^j}\right) 2^{j-1} = \sum_{j=1}^{k} \frac{n}{2} = \frac{kn}{2} = \frac{n \log n}{2},$$

and is at most

$$\begin{aligned}
\sum_{j=1}^{k} \frac{n}{2^j} \left(2^j - 1\right) &= \sum_{j=1}^{k} \left(n - \frac{n}{2^j}\right) \\
&= kn - n \sum_{j=1}^{k} \frac{1}{2^j} \\
&= kn - n\left(1 - \frac{1}{2^k}\right) \qquad \text{(Eq. 2.11, page 78)} \\
&= kn - n\left(1 - \frac{1}{n}\right) \\
&= n \log n - n + 1.
\end{aligned}$$

As to the number of element assignments, there are, by Observation 1.2 applied to each merge operation, $2n$ element assignments in each iteration of the outer **while** loop for a total of $2n \log n$. As a result, we have the following observation:

Observation 1.5 The total number of element comparisons performed by Algorithm BOTTOMUPSORT to sort an array of n elements, where n is a power of 2, is between $(n \log n)/2$ and $n \log n - n + 1$. The total number of element assignments done by the algorithm is exactly $2n \log n$.

1.8 Time Complexity

In this section we study an essential component of algorithmic analysis, namely determining the running time of an algorithm. This theme belongs to an important area in the theory of computation, namely computational complexity, which evolved when the need for efficient algorithms arose in the 1960's and flourished in the 1970's and 1980's. The main objects of study in the field of computational complexity include the time and space needed by an algorithm in order to deliver its output when presented with