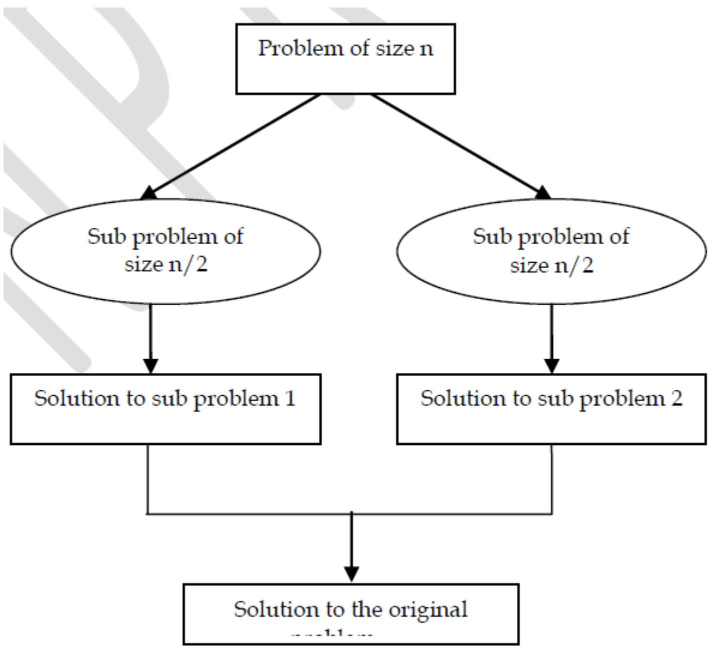
Unit II
DIVIDE AND CONQUER

General Method

Divide and Conquer is one of the best-known general algorithm design technique. Divide-and-conquer algorithms work according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original problem.

The divide-and-conquer technique is diagrammed, which depicts the case of dividing a problem into two smaller sub problems,



A function to compute on n inputs the D and C strategy suggests splitting the inputs into k distinct sub sets, 1<k ≤ n. These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. The control abstraction of the divide and conquer strategy is given below.

Control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures.

```
1.  Algorithm DAndC ( P)
2.  {
3.      if Small (P) then return S(P);
4.      else
5.      {
6.          Divide P into smaller instances P1, P2, ……………………Pk, $k \geq 1$;
7.          Apply DAndC to each of these sub problems;
8.          return Combine (DAndC (P1), DAndC (P2), …….. , DAndC(Pk));
9.      }
10. }
```

DAndC is initially invoked as DAndC (P) where P is the problem to be solved. Small (P) is a Boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem P is divided into smaller sub problems. These sub problems P1, P2, ……., Pk are solved by recursive applications of DAndC. Combine is the function that determines the solution to P using the solutions to the k sub problems. Computing time of DAndC is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n_1) + T(n_2) + \ldots\ldots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

Where T (n) is the time for DAndC on any input of size n and g(n) is the time to compute the answer directly for small inputs. The function f (n) is the time for dividing P and combing the solutions to sub problems.

One of the methods for solving any such recurrence relation is called the substitution method. This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.

ADVANTAGES:

- The time spent on executing the problem using divide and conquer is smaller than other methods.
- The divide and conquer approach provides an efficient algorithm in computer science.

- The divide and conquer technique is ideally suited for parallel computation in which each sub problem can be solved simultaneously by its own processor.

Binary Search

The binary search algorithm is one of the most efficient searching techniques which require the list to be sorted in ascending order. To search for an element in the list, the binary search algorithms split the list and locate the middle element of the list.

Let a_i, $1 \leq i \leq n$, be a list of elements that are sorted in increasing order. The problem is check whether the given element x is present in the list or not. If x is not in the list, then j is set to be 0.

Small (P) is true if n=1. In this case S(P) will take the value 1 if $x = a_i$ otherwise it will take the value zero.

Then $g(1) = \Theta(1)$. If P has more than one element, it can be divided or reduced into new sub problems. Pick an index q(middle element) and compare x with a_q. There are three possibilities.

1) $x= a_q$: in this case P is immediately solved.
2) $x< a_q$: x has to be searched for only in the sub list less than a_q
3) x> aq : sublist to be searched is greater than a_q

The resulting search algorithm is called binary search. There is no need for any combining. Recursive and iterative algorithms are given here. Recursive algorithm is invoked by Binsrch(a, 1, n, x).

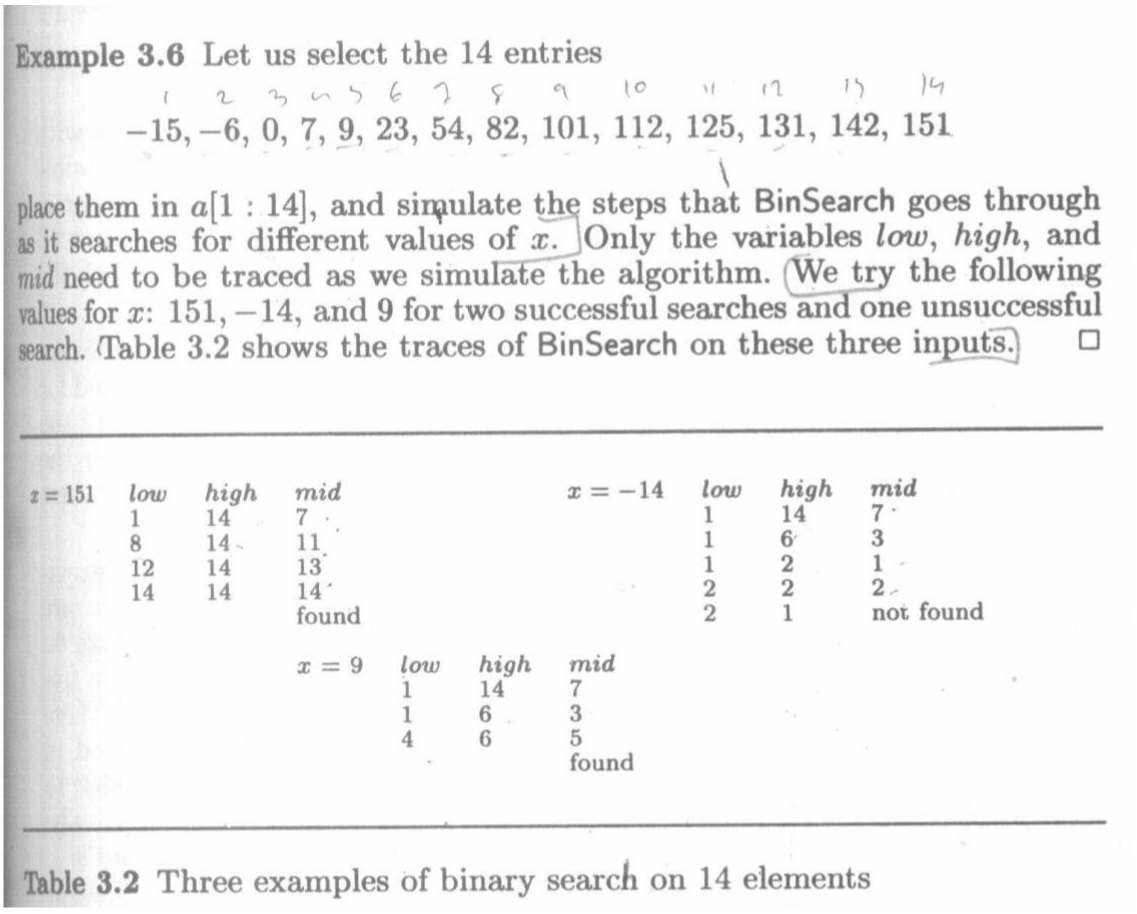
```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    if ($l = i$) then  // If Small($P$)
    {
        if ($x = a[i]$) then return $i$;
        else return 0;
    }
    else
    { // Reduce $P$ into a smaller subproblem.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                    return BinSrch($a, i, mid - 1, x$);
            else return BinSrch($a, mid + 1, l, x$);
    }
}
```

Algorithm 3.2 Recursive binary search

```
Algorithm BinSearch($a, n, x$)
// Given an array $a[1 : n]$ of elements in nondecreasing
// order, $n \geq 0$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    $low := 1$; $high := n$;
    while ($low \leq high$) do
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        if ($x < a[mid]$) then $high := mid - 1$;
        else if ($x > a[mid]$) then $low := mid + 1$;
            else return $mid$;
    }
    return 0;
}
```

Algorithm 3.3 Iterative binary search

Let us select 14 entries as an example which is given below:



Example 3.6 Let us select the 14 entries

1 2 3 4 5 6 7 8 9 10 11 12 13 14

$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$

place them in $a[1:14]$, and simulate the steps that BinSearch goes through as it searches for different values of x. Only the variables *low*, *high*, and *mid* need to be traced as we simulate the algorithm. We try the following values for x: $151, -14$, and 9 for two successful searches and one unsuccessful search. Table 3.2 shows the traces of BinSearch on these three inputs. □

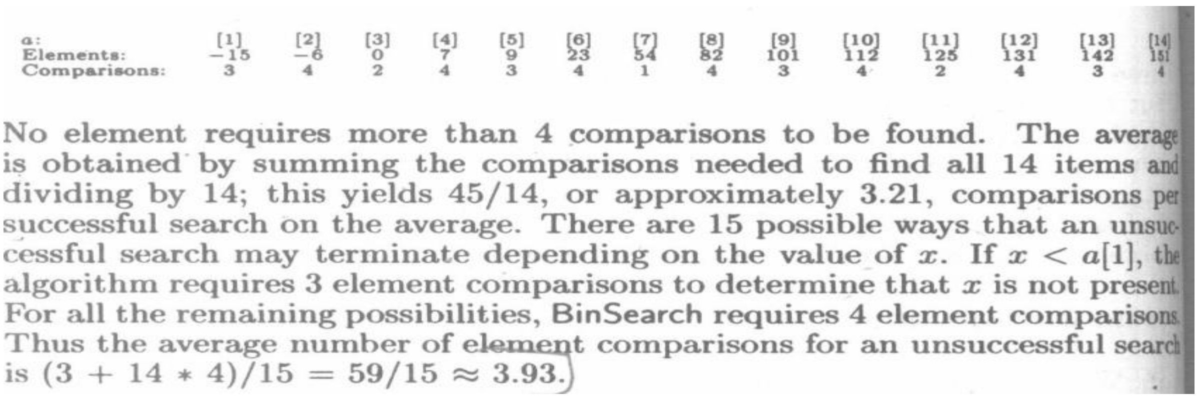
$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Table 3.2 Three examples of binary search on 14 elements

The number of element comparisons of 14 entries are represented by the following figure.



a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

No element requires more than 4 comparisons to be found. The averag is obtained by summing the comparisons needed to find all 14 items and dividing by 14; this yields 45/14, or approximately 3.21, comparisons per successful search on the average. There are 15 possible ways that an unsuc- cessful search may terminate depending on the value of x. If $x < a[1]$, the algorithm requires 3 element comparisons to determine that x is not present. For all the remaining possibilities, BinSearch requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is $(3 + 14 * 4)/15 = 59/15 \approx 3.93$.

Analysis of binary search

The better way to understand the algorithm is to consider the sequence of values for *mid* that are produced by BinSearch for all possible values of x. These values are nicely described using a binary decision tree in which the value in each node is the value of *mid*. For example, if $n = 14$, then Figure 3.1 contains a binary decision tree that traces the way in which these values are produced by BinSearch.

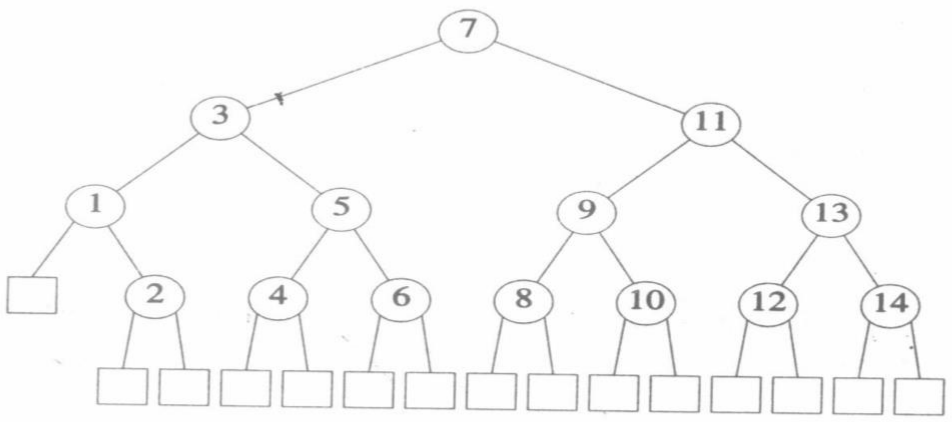


Figure 3.1 Binary decision tree for binary search, $n = 14$

The first comparison is x with $a[7]$. If $x < a[7]$, then the next comparison is with $a[3]$; similarly, if $x > a[7]$, then the next comparison is with $a[11]$. Each path through the tree represents a sequence of comparisons in the binary search method. If x is present, then the algorithm will end at one of the circular nodes that lists the index into the array where x was found. If x is not present, the algorithm will terminate at one of the square nodes. Circular nodes are called *internal* nodes, and square nodes are referred to as *external nodes*.

The standard way to analyze the efficiency is to count number of times search key is compared with an element of the array. The best case analysis is easy. For successful search only one element comparison is needed. For an unsuccessful search, log n element comparisons are needed in the best case. The computing time of binary search that describe the best, average and worst cases are

Successful searches	Unsuccessful searches
Θ(1), Θ(log n), Θ(log n)	Θ(log n)
Best average worst	best, average, worst

ADVANTAGES

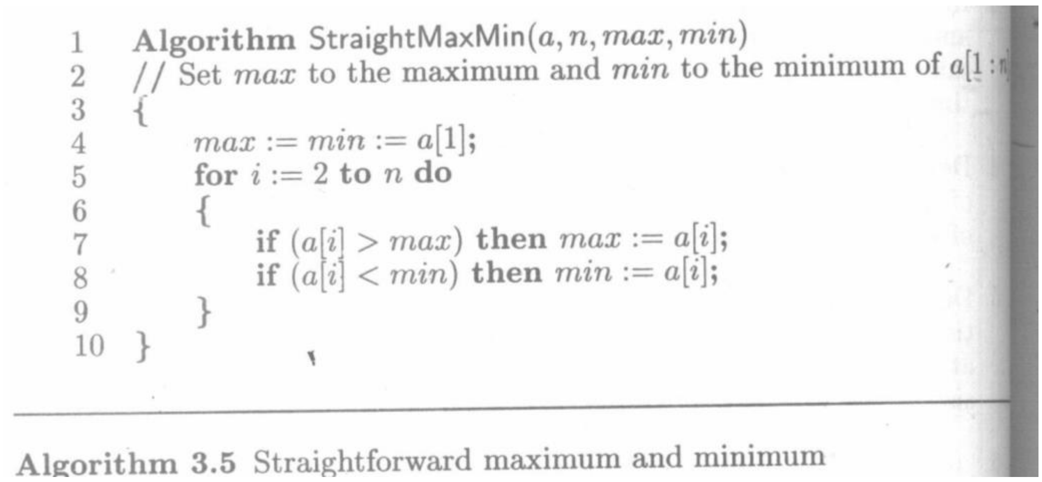
1. In this method elements are eliminated by half each time .So it is very faster than the sequential search.
2. It requires less number of comparisons than sequential search to locate the search key element.

DISADVANTAGES

1. An insertion and deletion of a record requires many records in the existing table be physically moved in order to maintain the records in sequential order.
2. The ratio between insertion/deletion and search time is very high.

Finding the maximum and minimum

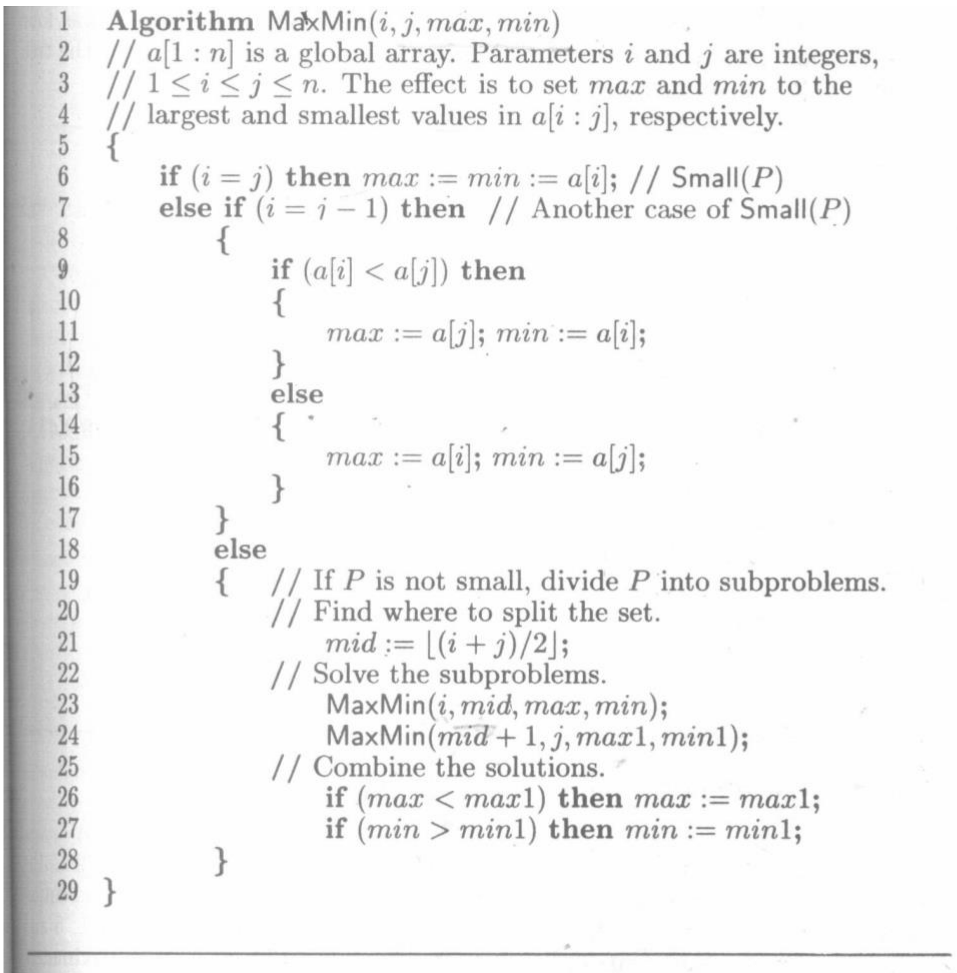
The problem is to find the maximum and minimum items in a set of n elements. Following algorithm shows the straight forward algorithm to find maximum and minimum.



```
Algorithm StraightMaxMin($a, n, max, min$)
// Set $max$ to the maximum and $min$ to the minimum of $a[1:n$
{
    $max := min := a[1]$;
    for $i := 2$ to $n$ do
    {
        if ($a[i] > max$) then $max := a[i]$;
        if ($a[i] < min$) then $min := a[i]$;
    }
}
```

Algorithm 3.5 Straightforward maximum and minimum

The divide and conquer algorithm for this problem is given below:



```
Algorithm MaxMin($i, j, max, min$)
// $a[1:n]$ is a global array. Parameters $i$ and $j$ are integers,
// $1 \leq i \leq j \leq n$. The effect is to set $max$ and $min$ to the
// largest and smallest values in $a[i:j]$, respectively.
{
    if ($i = j$) then $max := min := a[i]$; // Small($P$)
    else if ($i = j - 1$) then // Another case of Small($P$)
        {
            if ($a[i] < a[j]$) then
            {
                $max := a[j]$; $min := a[i]$;
            }
            else
            {
                $max := a[i]$; $min := a[j]$;
            }
        }
        else
        {   // If $P$ is not small, divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (i+j)/2 \rfloor$;
            // Solve the subproblems.
                MaxMin($i, mid, max, min$);
                MaxMin($mid + 1, j, max1, min1$);
            // Combine the solutions.
                if ($max < max1$) then $max := max1$;
                if ($min > min1$) then $min := min1$;
        }
}
```

Algorithm 3.6 Recursively finding the maximum and minimum

The procedure is initially invoked by the statement MaxMin (1, n, x, y).

Complexity Analysis of Finding Maximum & Minimum

Suppose we simulate MaxMin on the following nine elements:

uppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

good way of keeping track of recursive calls is to build a tree by adding ode each time a new call is made. For this algorithm each node has fo ems of information: i, j, max, and min. On the array $a[\,]$ above, the tr f Figure 3.2 is produced.

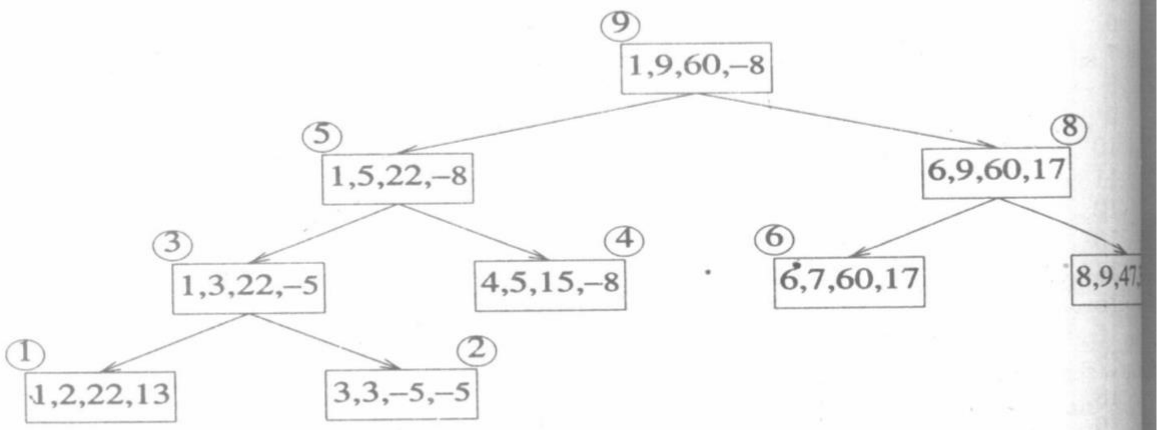


Figure 3.2 Trees of recursive calls of MaxMin

Examining Figure 3.2, we see that the root node contains 1 and 9 as t values of i and j corresponding to the initial call to MaxMin. This executi produces two new calls to MaxMin, where i and j have the values 1, 5 a 6, 9, respectively, and thus split the set into two subsets of approximat the same size. From the tree we can immediately see that the maxim depth of recursion is four (including the first call). The circled numbers the upper left corner of each node represent the orders in which max a min are assigned values.

The recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of 2, n=2k for some positive integer k, then

T (n) = 2T (n/2) + 2

= 2(2T (n/4) +2) +2

= 4T (n/4) +4+2

:

:

= 2k-1T (2) +Σ1<=i<=k-1 2i

= 2k-1+2k-2

=3n/2-2

Therefore 3n/2 is the best, average and worst case number of comparisons when n is a power of two.

Consider the count when element comparisons have the same cost as comparisons between i and j. Let C (n) be this number. Assuming n=2k for some positive integer k, we get

C(n)	= 2C(n/2) + 3	n>2
	2	n=2

C (n) = 2C (n/2) + 3

= 4 C (n/4) + 6 + 3

:

:

$= 2^{k-1}C\ (2) + 3\Sigma_0^{k-2}\ 2i$

$= 2^k+3*2^{k-1}-3$

= 5n/2-3

If comparisons between array elements are costlier than comparisons of integer variable, then divide and conquer method is more efficient. In both cases mentioned above the best case,average case and worst case complexity is θ(n)

Merge Sort

Merge sort is another example of divide and conquer approach. A sorting algorithm that has the property in the worst case its complexity is O (n log n) is called merge sort. Elements are sorted in non decreasing order and that are called as keys.

STEPS TO BE FOLLOWED

♦ The first step of the merge sort is to chop the list into two.

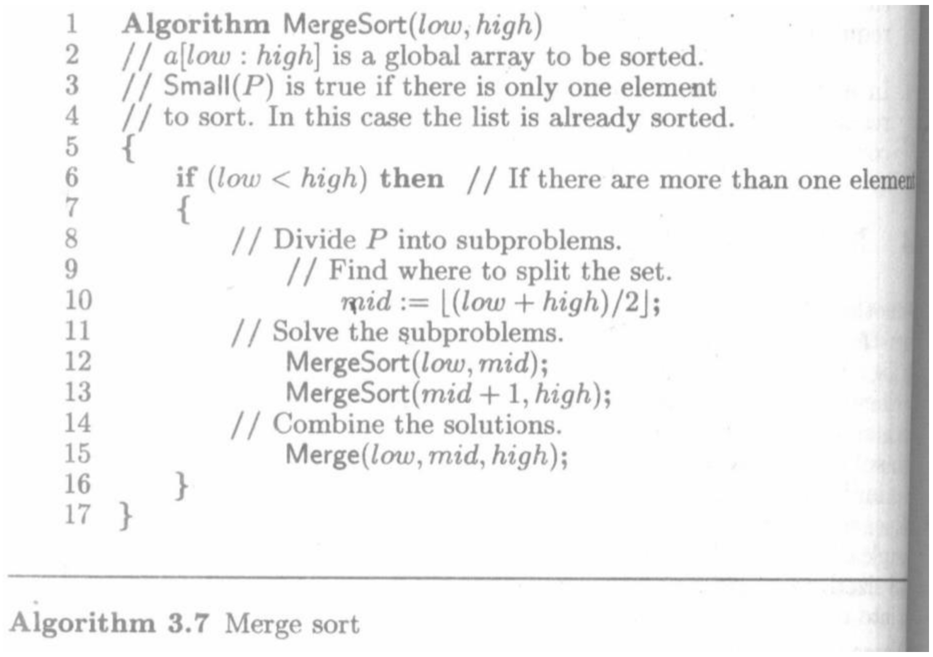
♦ If the list has even length, split the list into two equal sub lists.

♦ If the list has odd length, divide the list into two by making the first sub list one entry greater than the second sub list.

♦ then split both the sub list into two and go on until each of the sub lists are of size one.

♦ finally, start merging the individual sub lists to obtain a sorted list.

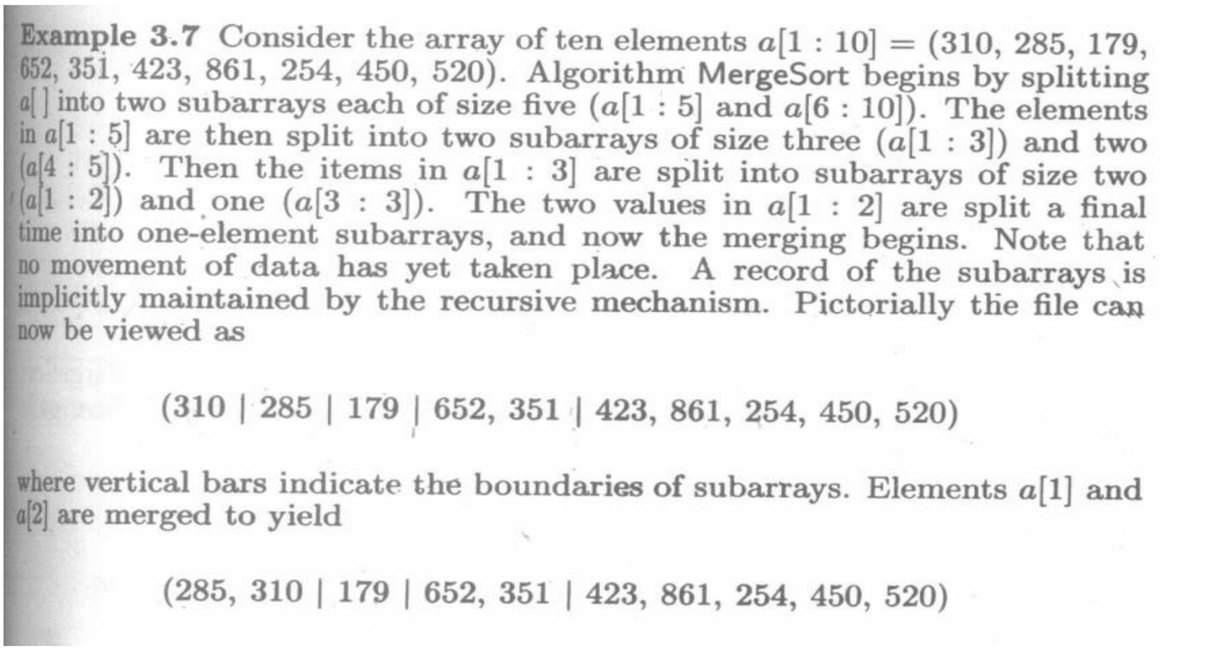
MergeSort algorithm describes this process very succinctly using recursion and a function Merge which merges two sorted sets.



```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one elemen
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Algorithm 3.7 Merge sort

Complexity Analysis of Merge sort

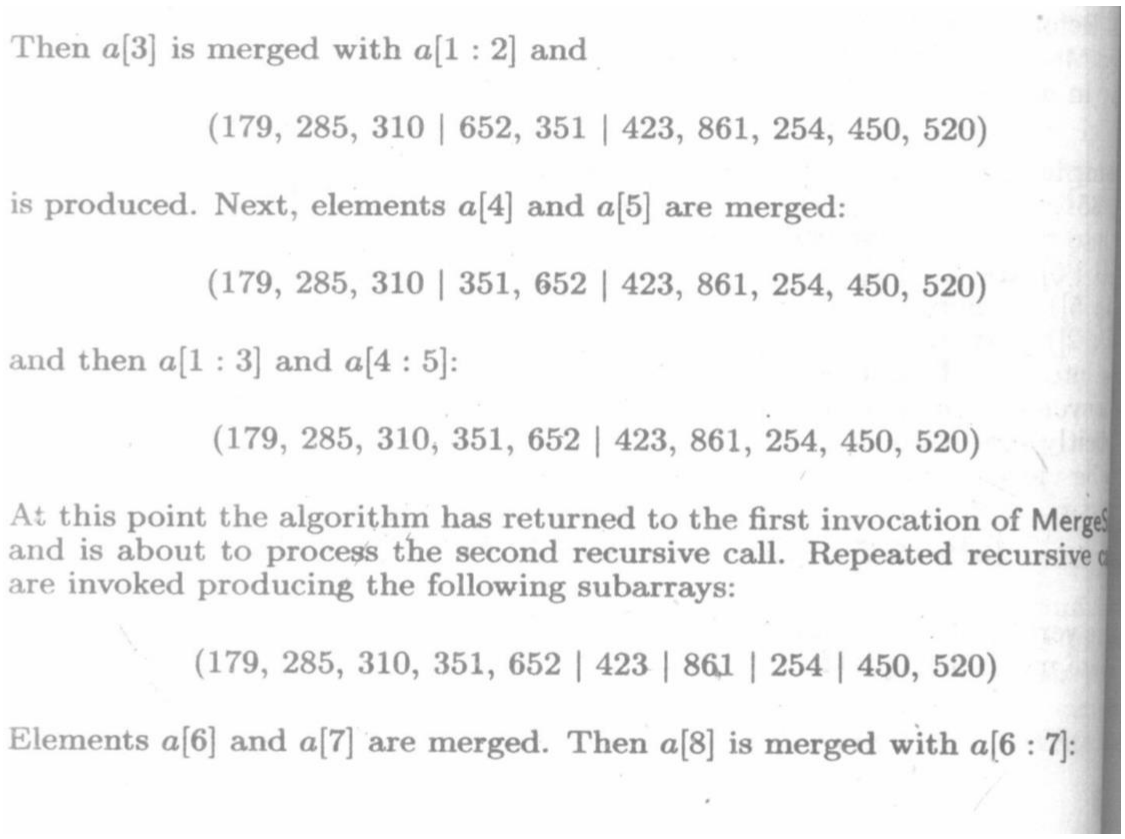


Example 3.7 Consider the array of ten elements $a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$. Algorithm MergeSort begins by splitting $a[\,]$ into two subarrays each of size five ($a[1:5]$ and $a[6:10]$). The elements in $a[1:5]$ are then split into two subarrays of size three ($a[1:3]$) and two ($a[4:5]$). Then the items in $a[1:3]$ are split into subarrays of size two ($a[1:2]$) and one ($a[3:3]$). The two values in $a[1:2]$ are split a final time into one-element subarrays, and now the merging begins. Note that no movement of data has yet taken place. A record of the subarrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

$$(310 \mid 285 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

$$(285, 310 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$



Then $a[3]$ is merged with $a[1:2]$ and

$$(179,\ 285,\ 310 \mid 652,\ 351 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

$$(179,\ 285,\ 310 \mid 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

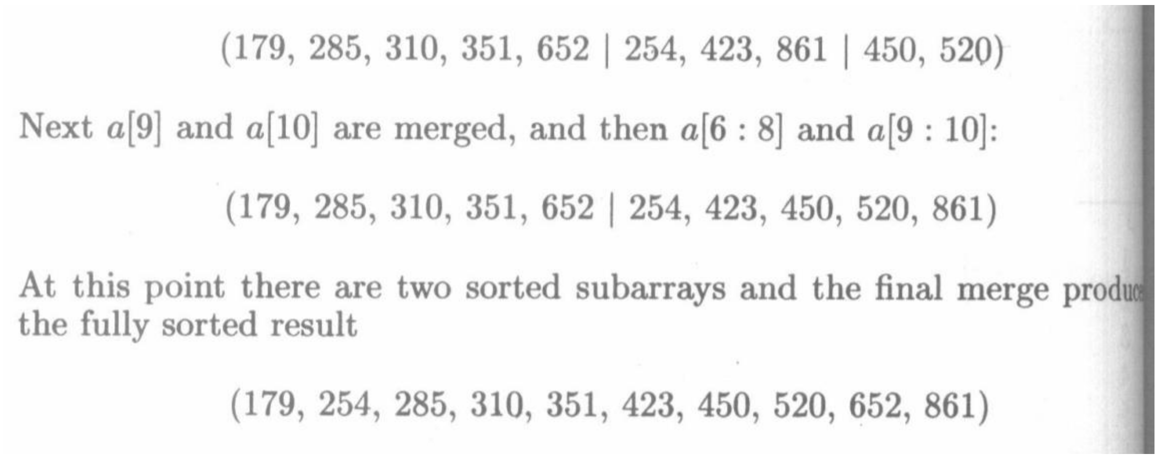
and then $a[1:3]$ and $a[4:5]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

At this point the algorithm has returned to the first invocation of MergeS
and is about to process the second recursive call. Repeated recursive c
are invoked producing the following subarrays:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423 \mid 861 \mid 254 \mid 450,\ 520)$$

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6:7]$:



$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 861 \mid 450,\ 520)$$

Next $a[9]$ and $a[10]$ are merged, and then $a[6:8]$ and $a[9:10]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 450,\ 520,\ 861)$$

At this point there are two sorted subarrays and the final merge produ
the fully sorted result

$$(179,\ 254,\ 285,\ 310,\ 351,\ 423,\ 450,\ 520,\ 652,\ 861)$$

Following tree represents the sequence of recursive calls that are produced by MergeSort when it is applied to ten elements.

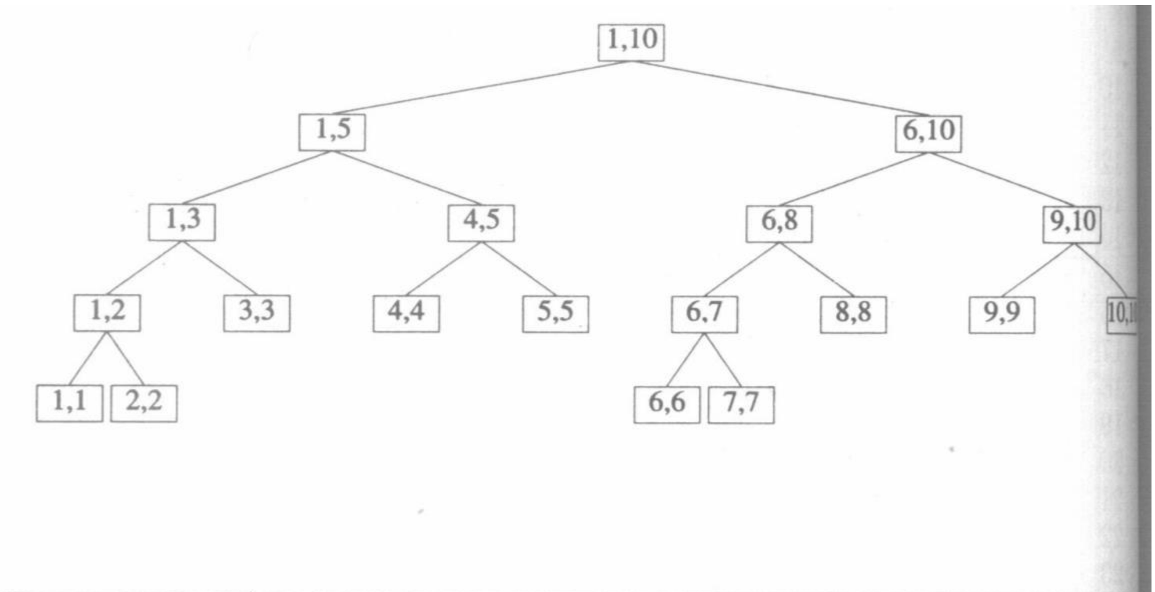


Figure 3.3 Tree of calls of MergeSort(1, 10)

The given figure is a tree representing the calls to procedure Merge by MergeSort

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

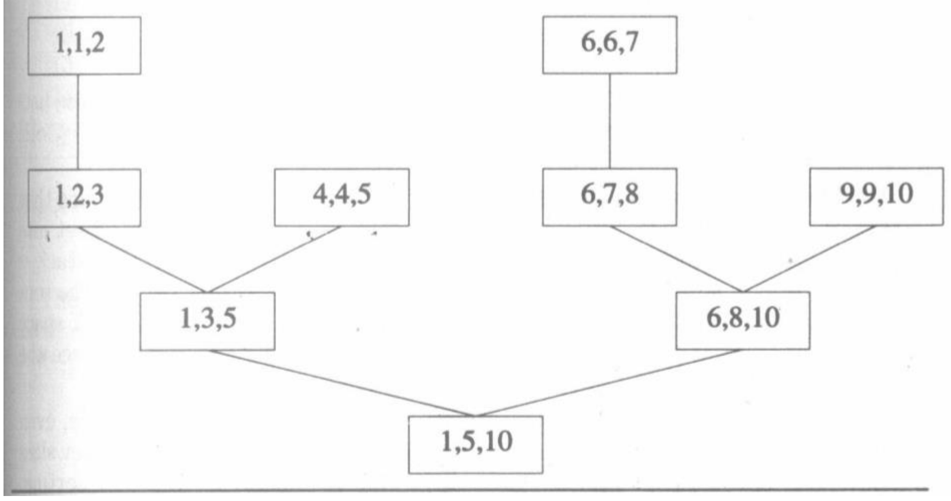
$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$



Limitations

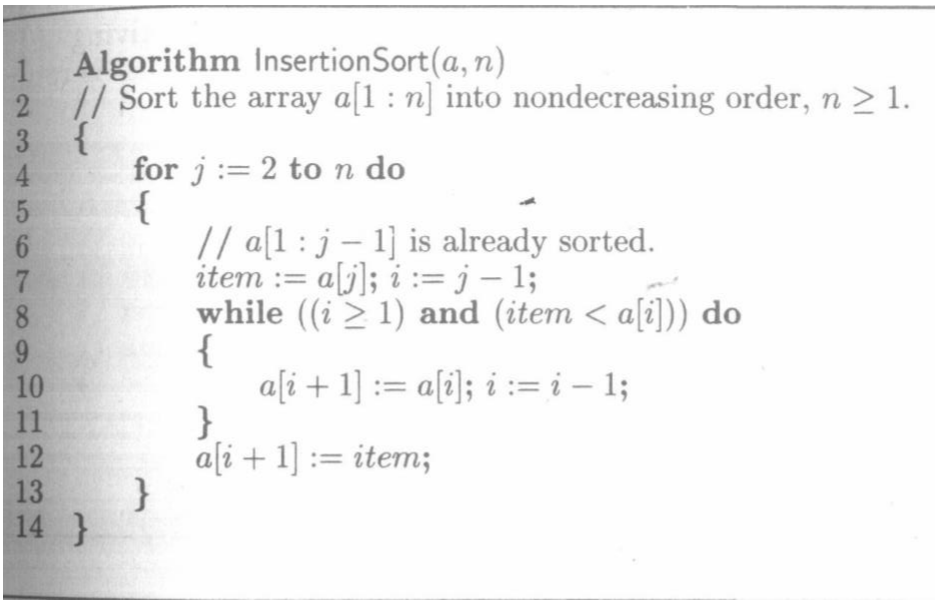
- It uses 2n locations.
- Additional stack space for recursion

For small set sizes most of the time will be spent processing recursion instead of sorting. Solution to this problem is not allowing the recursion to go to the lowest level. So there is a second sorting algorithm Insertion sort for small sizes. Insertion sort works fast on arrays of less than 16 elements. The basic ideas of sorting is

```
for j:= 2 to n do
{
place a[j] in its correct position
```

in the sorted set a[1: j-1];

}

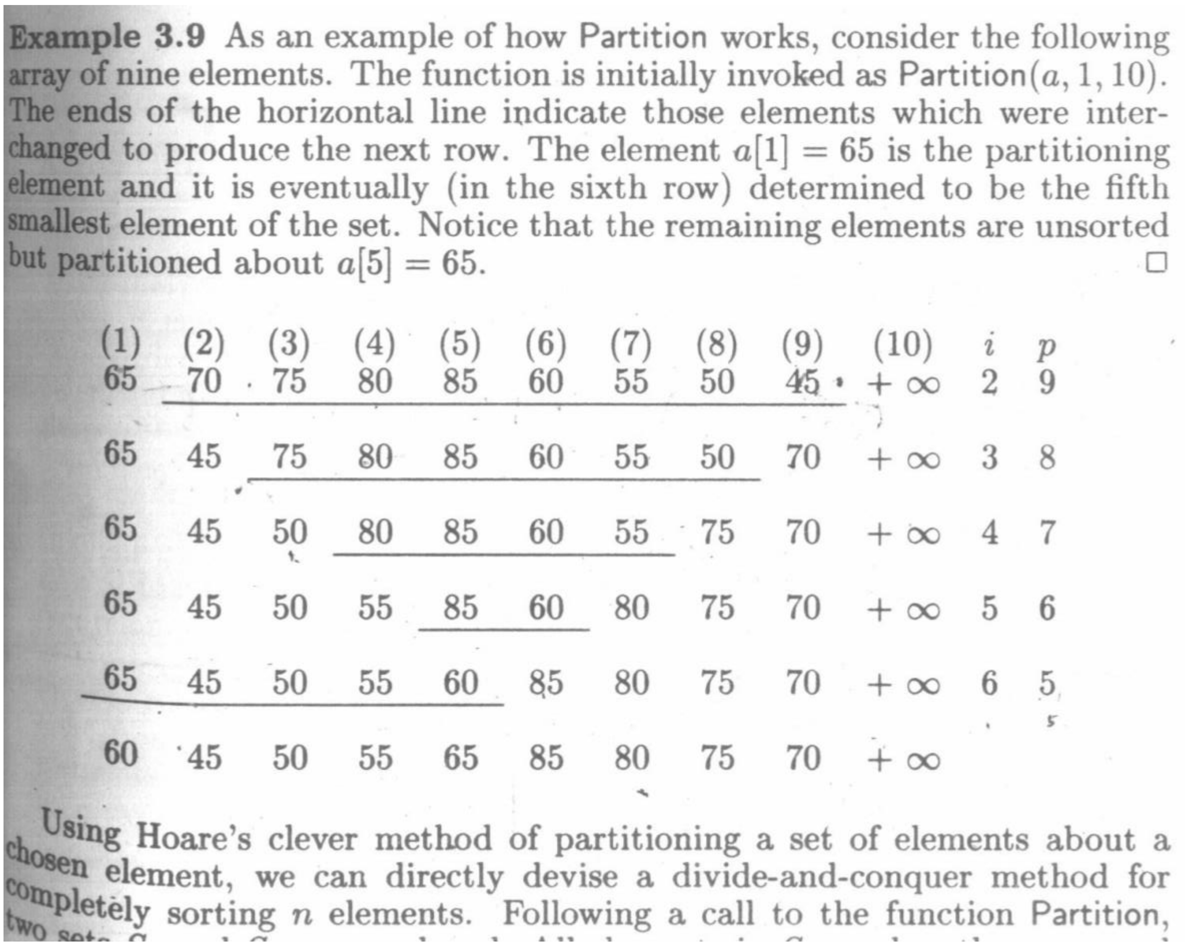


```
Algorithm InsertionSort($a, n$)
// Sort the array $a[1 : n]$ into nondecreasing order, $n \geq 1$.
{
    for $j := 2$ to $n$ do
    {
        // $a[1 : j - 1]$ is already sorted.
        $item := a[j]$; $i := j - 1$;
        while (($i \geq 1$) and ($item < a[i]$)) do
        {
            $a[i + 1] := a[i]$; $i := i - 1$;
        }
        $a[i + 1] := item$;
    }
}
```

The worst case computing time for this insertion sort is $\Theta(n^2)$ and best case time is $\Theta(n)$ because body of while loop is never entered in best case.

Quick Sort

In quick sort division into two sub arrays is made but sorted sub arrays do not need to be merged later. This is accomplished by rearranging a [1: n] such that a[i] ≤ a[j] for all i between 1 and m and all j between m+1 and n for some m, 1 ≤ m ≤ n. Rearrangement is performed by picking some elements of a[], t= a[s], then reorder the other elements, so that all elements appearing before t in a[1:n] are ≤ t and all elements appearing after t are ≥ t. This rearranging is referred to as **partitioning**.



Example 3.9 As an example of how Partition works, consider the following array of nine elements. The function is initially invoked as Partition$(a, 1, 10)$. The ends of the horizontal line indicate those elements which were interchanged to produce the next row. The element $a[1] = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about $a[5] = 65$. □

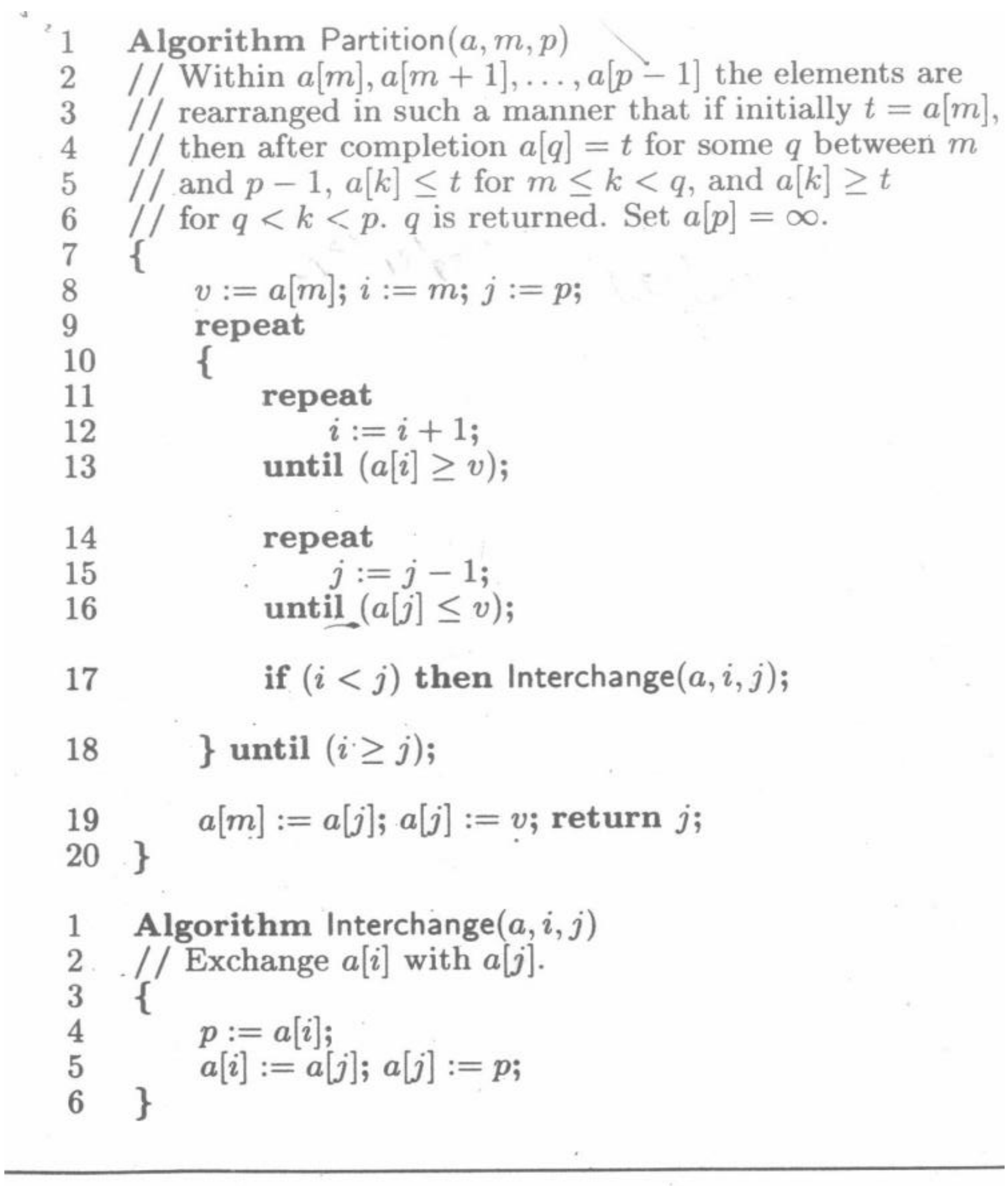
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Using Hoare's clever method of partitioning a set of elements about a chosen element, we can directly devise a divide-and-conquer method for completely sorting n elements. Following a call to the function Partition, two sets

Following a call to function partition, two sets S1 and S2 are produced. All elements in S1 are ≤ elements in S2. Each set is sorted by reusing function partition.

```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1 : n]$ into ascending order; $a[n + 1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1 : n]$.
{
    if ($p < q$) then // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j$ := Partition($a, p, q + 1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j - 1$);
            QuickSort($j + 1, q$);
        // There is no need for combining solutions.
    }
}
```

Algorithm 3.13 Sorting by partitioning



```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}
```

```
Algorithm Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

Algorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

Worst case time is O (n2) and average time is only O (nlog n).

Stack space for recursion is O (log n) because worst case maximum depth of recursion may be n-1.

Performance measurement

QuickSort and MergeSort were evaluated on a SUN workstation 10/30. Each data set consisted of random integers in the range (0, 1000). Following tables show the results in milliseconds.

Scanning the above tables, we immediately see that QuickSort is faster than MergeSort for all values. Even though both algorithms require O(n log n) time on the average, QuickSort usually performs well in practice.

Selection

From given n elements a[1:n] determine the k^{th} smallest element. Following algorithm shows the working of this.

```
Algorithm Select1(a,n, k)
//selects the kth smallest element in a[1:n] and places it
//in the kth position of a[]. The remaining elements are
//rarranged such that $a[m] \le a[k]$ for $1 \le m < k$ and
//$a[m] \ge a[k]$ for $k < m \le n$.
{
  low:=1; up := n +1;
  a[n+1] := $\infty$; // a[n+1] is set to infinity
  repeat
  {
  //each time the loop is entered,
  //$1 \le low \le k \le up \le n+1$
  j:= Partition (a, low, up);
  // j is such that a[j] is the jth smallest value in a[]
  If (k=j) then return;
  else if (k<j) then up := j; // j is the new upper limit
          else low := j +1; // j+1 is the new lower limit
  } until (false);
}
```

The above algorithm places the k^{th} smallest element into position a[k] and partitions the remaining elements so that $a[i] \le a[k]$, $1 \le i < k$, and $a[i] \ge a[k]$, $k < i \le n$.

Assumptions from Select1 that were made for QuickSort are:

1. The n elements are distinct
2. Partition element has equal probability to can be i^{th} smallest

Partition requires O(p – m) time m increases by at least one and j decreases by at least one.

Worst case complexity of Select1 is O(n2) .

Average computing time is O(n)

The space needed by Select1 O(1).

Strassen's Matrix Multiplication

Let A and B be two n x n matrices. To compute C(i, j) = $\sum_{1 \le k \le n} A(i, k)B(k, j)$

This equation needs n multiplications to compute C(i, j). The matrix C has n^2 elements, time for matrix multiplication is Θ (n^3) by conventional method. Divide and Conquer strategy suggests another way to compute the product. Assume n is the power of 2. When n is not a power of two, rows and columns of zeros can be added to both A and B, resulting dimensions are a power of two.

Imagine A and B are each partitioned into four square sub matrices, each have n/2 x n/2 dimensions. Product of AB can be calculated by the above formula for the product of 2 x 2 matrix.

That is, $C_{11} = A_{11} B_{11} + A_{12} B_{12}$

$C_{12} = A_{11} B_{12} + A_{12} B_{22}$

$C_{21} = A_{21} B_{11} + A_{22} B_{21}$

$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

To compute AB using above equations we need to perform 8 multiplications and four additions of n/2 x n/2 matrices. So over all computing time T(n) by divide and conquer algorithm is

$$T(n) = \begin{cases} b & n \le 2 \\ 8\,T(n/2) + Cn^2 & n > 2 \end{cases}$$

where b an

This recurrence can be solved the same way as earlier recurrence T(n)= O (n^3). No improvement over conventional method because matrix multiplications are more expensive than matrix addition. Reformulate the equation for Cij with fewer multiplications and more additions.

Volker Strassen discovered a way to compute Cij's of using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven n/2 x n/2 matrices.

P, Q, R, S, T, U and V.

P= (A11 + A22) (B11 + B22)

Q= (A21 + A22) B11

R= A11 (B12 – B22)

S= A22 (B21 – B11)

T= (A11 + A12)B22

U= (A21 – A11) (B11 + B12)

V= (A12 – A22) (B21 + B22)

$C11 = P + S - T + V$

$C12 = R + T$

$C21 = Q + S$

$C22 = P + R - Q + U$

The resulting recurrence relation for T (n) is

$$T(n)= \begin{cases} b & n \le 2 \\ 7\,T(n/2) + an^2 & n > 2 \end{cases}$$

where a and b are constants. Working with this formula, we get

$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$