U N I T - 3

Algorithms – Priority Queues - Heaps – Heap Sort – Merge Sort – Quick Sort – Binary Search – Finding the Maximum and Minimum.

ALGORITHM

An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then forall cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

An algorithm is composed of a finite set of steps, each of which may require one or more operations.

Study of Algorithms involves

1. How to devise algorithms?
2. How to validate algorithms?
3. How to analyze algorithms?
4. How to test a program?
    - *Debugging* is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
    - *Profiling* or *performance measurement* is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Algorithm Specification

Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.
It’s simply an implementation of an algorithm written in plain English.
It has no syntax like any of the programming language and thus can’t be compiled or interpreted by the computer.

Pseudocode Convention:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces:{ and }.
3. An identifier begins with a letter.
4. Assignment of values to variables is done using the assignment statement

variable:=expression;

5. There are two Boolean values true and false. Logical operators **and or** and **not.**
   Relational operators <, >, ≥, ≤, and ≠ are provided.
6. The following looping statements are employed: for, while, and repeat- until

```
                    while (condition)do
                     {
                              (statement1)
                                   .
                                   .
                              (statement n) }
                     }
```

```
for variable:=valuel to value2 step step do
{
    (statement1)
    (statementn)
}
```

Foe eg:

```
               1  Algorithm Max (A, n)
               2  // A is an array of size n.
               3  {
               4       Result:=A[l];
               5       for i :=2 to n do
               6                  if A[i] >Result then Result:=A[i];
               7        returnResult;
               8   }
```

PERFORMANCE ANALYSIS

There are two criteria to analyze the performance of algorithms:
1. Space Complexity
2. Time Complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.
The time complexity of an algorithm is the amount of computer time it needs to run to completion.
Performance evaluation can be loosely divided into two major phases:

(1) A priori estimates / Performance analysis
(2) A posteriori testing / Performance measurement

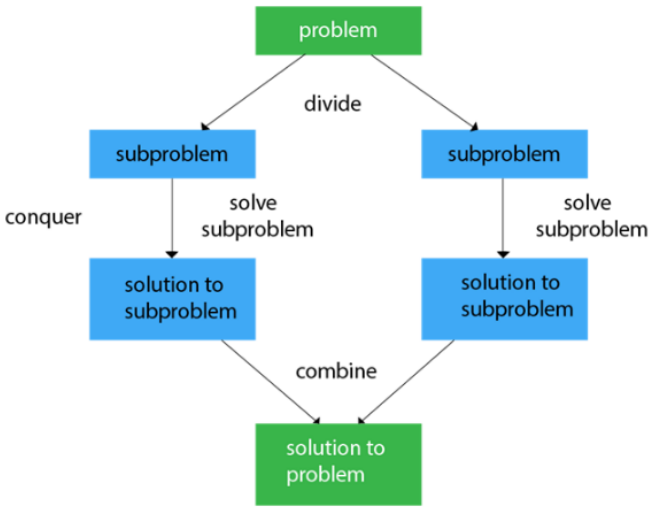
Algorithm Design TechniquesDivide - And –Conquer (Unit –III)

- The Greedy Method (Unit- IV)
- Dynamic Programming
- Backtracking (Unit –V)
- Branch-And-Bound

DIVIDE AND CONQUER

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. **Divide**:
   Break the given problem into subproblems of same type.
2. **Conquer**:
   Recursively solve these subproblems
3. **Combine**:
   Appropriately combine the answers



Examples:

The following computer algorithms are based on **divide-and-conquer** programming approach −

- Binary Search
- Finding the Maximum and Minimun
- Merge Sort
- Quick Sort

```
Algorithm Dand C(P)
{
    if small(P) then return s(p);
    else
     {
        Divide P into smaller instances P1,P2,P3,…..Pk. K>=1;
        Apply DandC to each of these subprograms.0
        return Combine(DandC(P1), DandC(P2), …………, DandC(Pk));
     }
}
```

PRIORITY QUEUES

Any data structure that supports the operations of search min (or max), insert., and delete min (or max, respectively) is called a *priority queue.*

Heaps

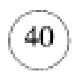
A *max (min) heap* is a complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist). Call this property the *heap property*.

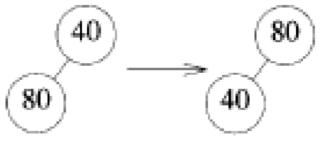
The definition of a max heap implies that one of the largest elements is at the root of the heap.

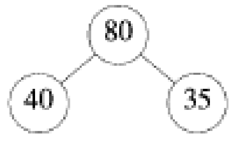
If the elements are distinct, then the root contains the largest item. A max heap can be implemented using an array a[].

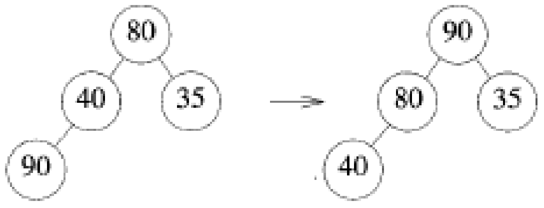
To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, great grandparent, and so on, until it is less than or equal to one of these values.

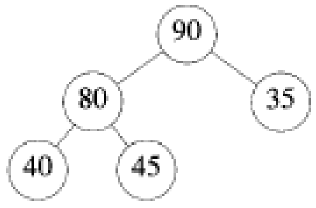
Forming a heap from the set {40, 80, 35, 90, 45, 50, 70}

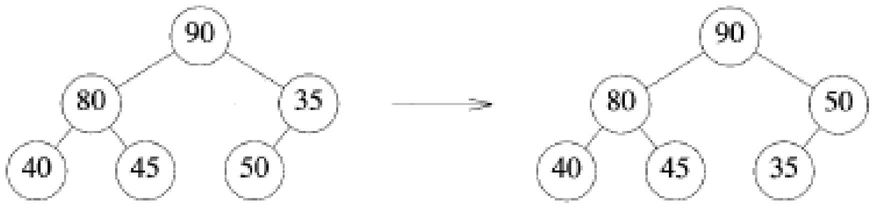

(a)

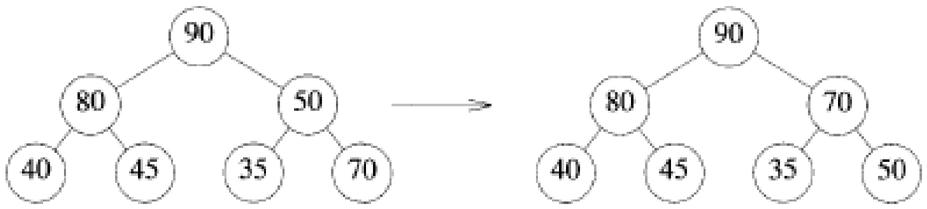

(b)


(c)


(d)


(e)


(f)


(g)

HEAP SORT

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

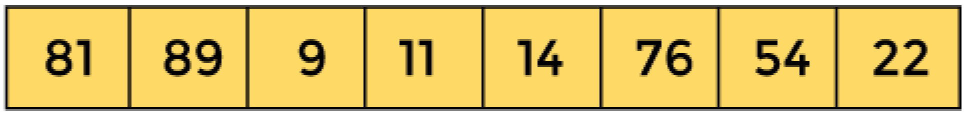
Now, let's see the working of the Heapsort Algorithm.

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

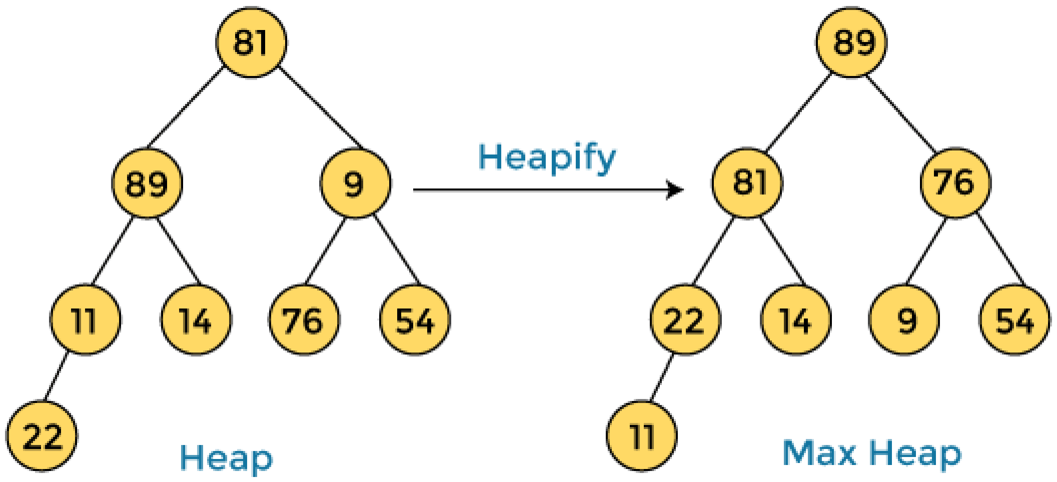
- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Example:

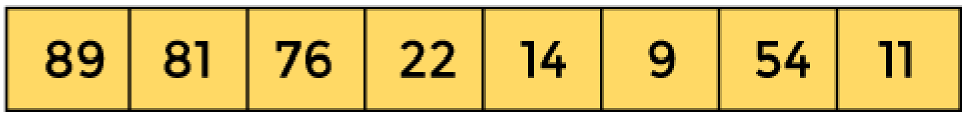
First, we have to construct a heap from the given array and convert it into max heap.



81	89	9	11	14	76	54	22

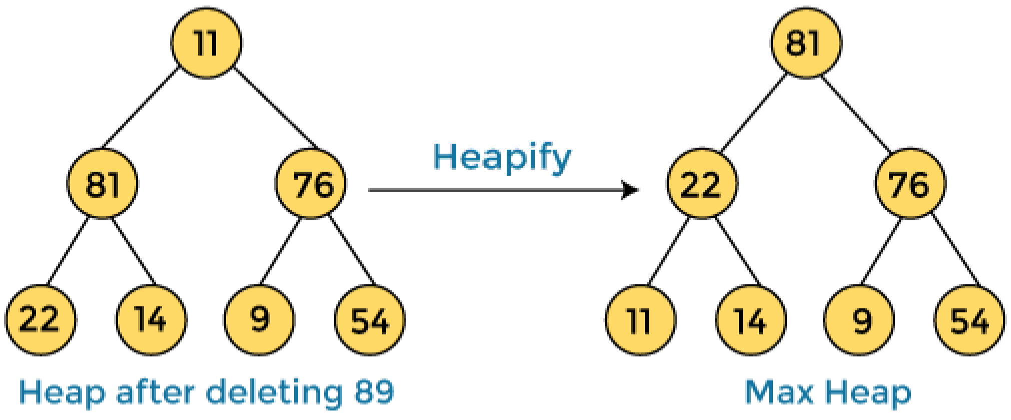


After converting the given heap into max heap, the array elements are -

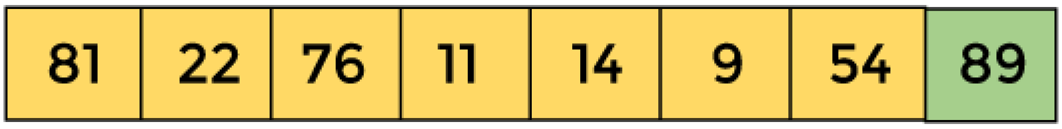


89	81	76	22	14	9	54	11

Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.

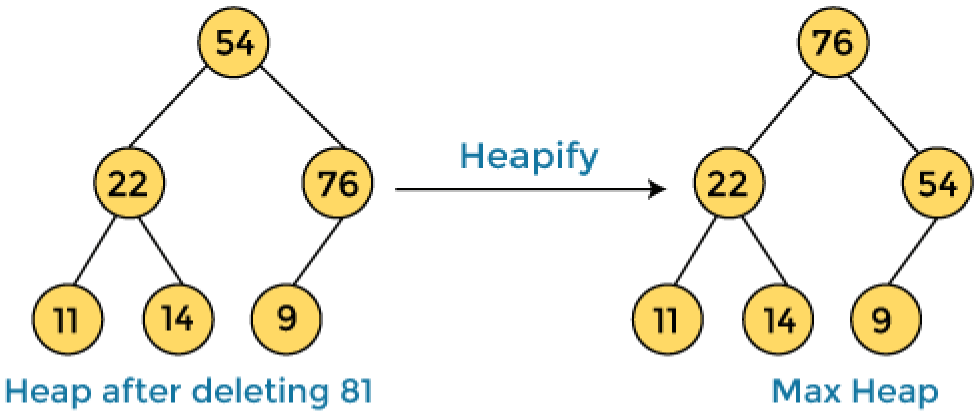


After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are -

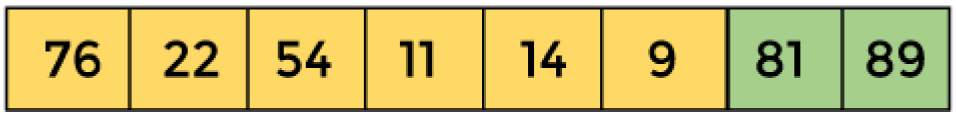


81	22	76	11	14	9	54	89

In the next step, again, we have to delete the root element (81) from the max heap. To delete this node, we have to swap it with the last node, i.e. (54). After deleting the root element, we again have to heapify it to convert it into max heap.

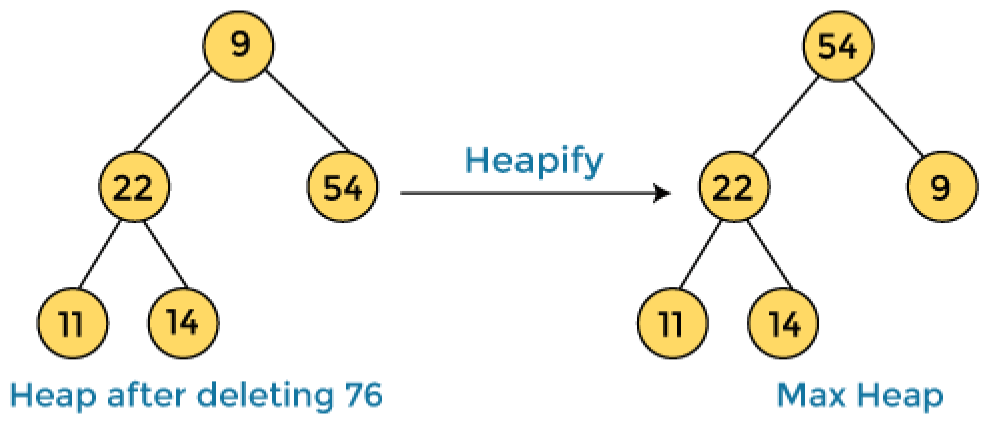


After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are -

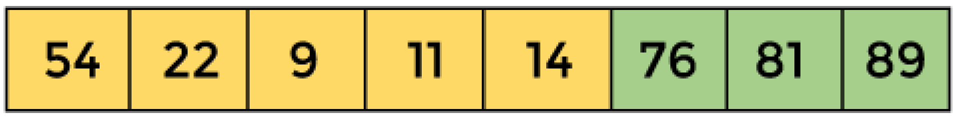


76	22	54	11	14	9	81	89

In the next step, we have to delete the root element (76) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.

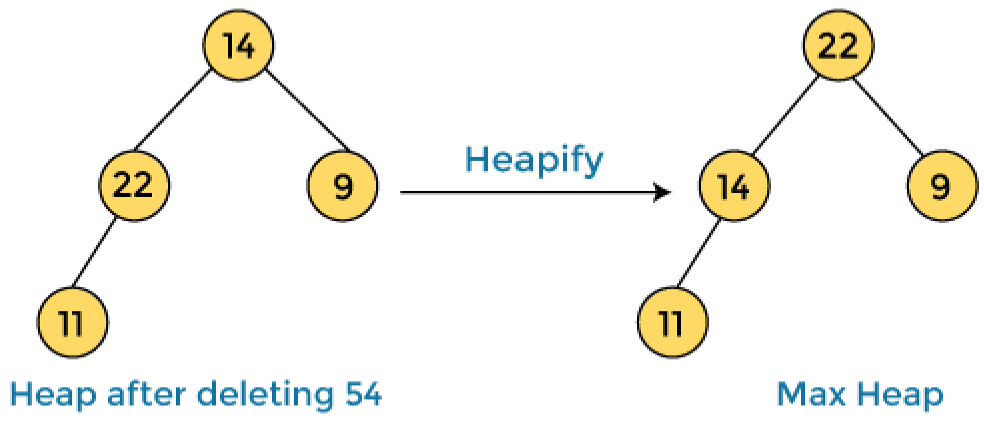


After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are -

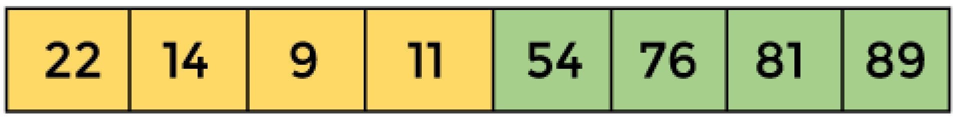


54	22	9	11	14	76	81	89

In the next step, again we have to delete the root element (54) from the max heap. To delete this node, we have to swap it with the last node, i.e. (14). After deleting the root element, we again have to heapify it to convert it into max heap.

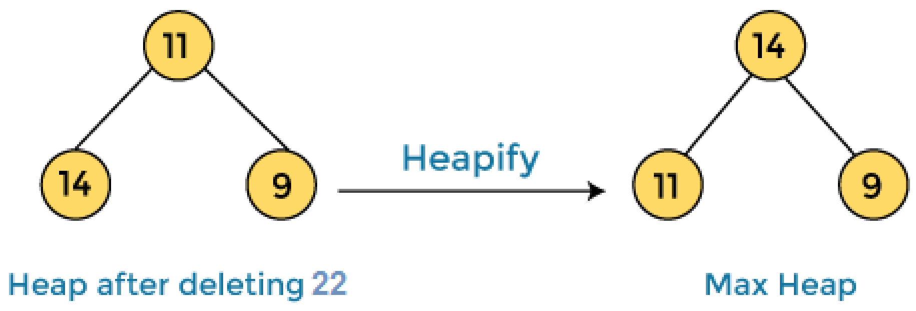


After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are -



22	14	9	11	54	76	81	89

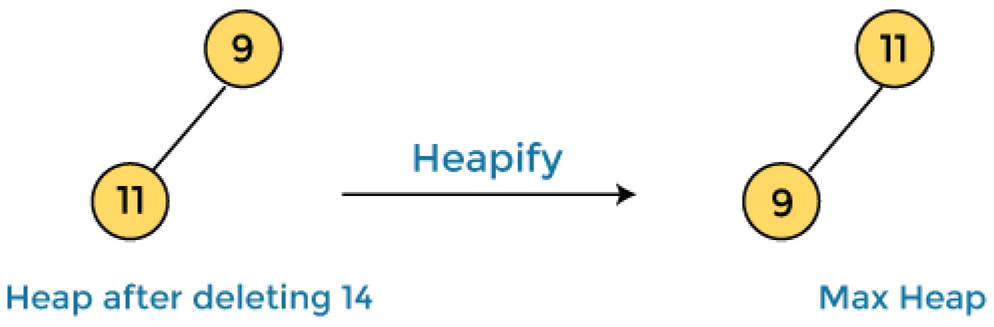
In the next step, again we have to delete the root element (22) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89

In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.

9
Heapify
9
Heap after deleting 11
Max Heap

After swapping the array element 11 with 9, the elements of array are -

9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty

```
Algorithm Heapify(a, n)
// Readjust the elements in a[1 : n] to form a heap.
{
    for i := ⌊n/2⌋ to 1 step −1 do Adjust(a, i, n);
}
```

```
Algorithm HeapSort(a, n)
// a[1 : n] contains n elements to be sorted. HeapSort
// rearranges them inplace into nondecreasing order.
{
    Heapify(a, n); // Transform the array into a heap.
    // Interchange the new maximum with the element
    // at the end of the array.
    for i := n to 2 step −1 do
    {
        t := a[i]; a[i] := a[1]; a[1] := t;
        Adjust(a, 1, i − 1);
    }
}
```

BINARY SEARCH

Let a_i, $1 \leq i \leq n$, be a list of elements that are sorted in nondecreasing order.

Consider the problem of determining whether a given element x is present in the list.

If x is present, we are to determine a value j such that $a_j = x$. If x is not in the list, then j is to be set to zero.

Let $P = (n, a_i, \ldots, a_l, x)$ denote an arbitrary instance of this search problem where n is the number of elements in the list, $a_i, \ldots, a_l$ is the list of elements, and x is the element searched for.

Divide-and-conquer can be used to solve this problem. Let Small(P) be true if $n = 1$. In this case, S(P) will take the value i if $x = a_i$; otherwise it will take the value 0.

If P has more than one element, it can be divided (or reduced) into a new subproblem as follows. Pick an index q (in the range [i, l]) and compare x with a_q.

There are three possibilities:

(1) $x = a_q$: In this case the problem *P* is immediately solved.

(2) $x < a_q$: In this case *x* has to be searched for only in the first half of the list.

(3) $x > a_q$: In this case *x* has to be searched for only in the second half of the list.

```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \le i \le l$, 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($a, mid + 1, l, x$);
    }
}
```

Example: Let us select the 14 entries

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

$x = 151$	*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

FINDING THE MAXIMUM AND MINIMUM

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array..

To find the maximum and minimum numbers in a given array numbers[] of size n, the following algorithm can be used. First we are representing the naive method and then we will present divide and conquer approach.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
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 $min := a[i]$;
    }
}
```

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

A divide-and-conquer algorithm for this problem would proceed as fol- lows: Let P = (n, a[i], ... , a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list a[i], ... , a[j] and we are interested in finding the maximum and minimum of this list. Let Small(P)/ be true when n≤2. In this case, the maximum and minimum are a[i] if n = 1.

If n = 2, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances A = $(\lfloor n/2 \rfloor, a[1], \ldots, a[\lfloor n/2 \rfloor])$ and P2 = $(n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], \ldots, a[n])$ .After having divided P into two smaller subproblems, we can solve them by recursively invoking the same divide-and-conquer algorithm. If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then MAX(P) is the larger of MAX(A) and MAX(P2)- Also, MIN(P) is the smaller of MIN (A) and MIN (P2).

The algorithm for the recursive approach to find the maximum and minimum of an array using divide-and-conquer technique is shown below:

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers,
// 1 ≤ i ≤ j ≤ 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 := ⌊(i + j)/2⌋;
            // 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;
        }
}
```

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

Example:

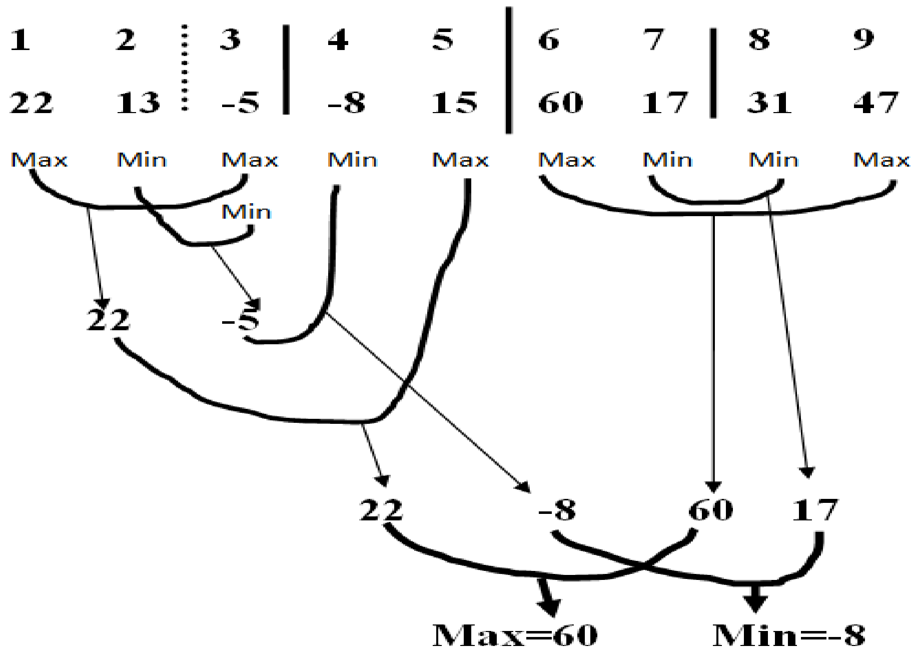
a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

The procedure works as follows:

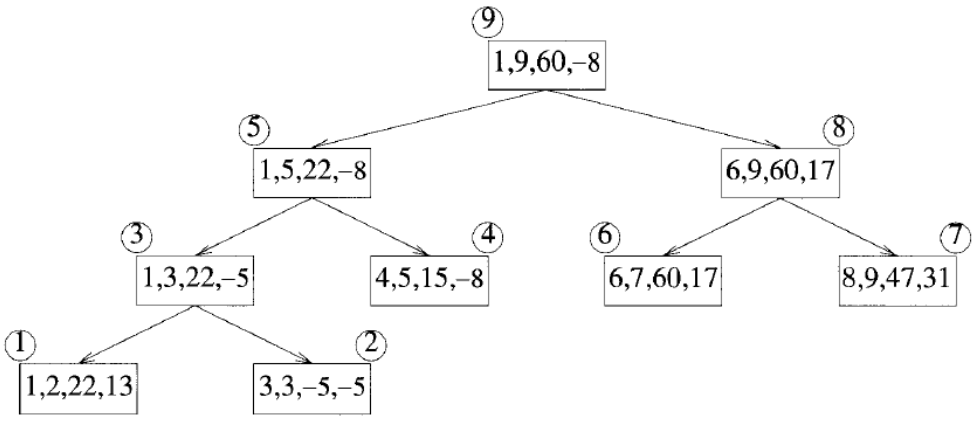
1	2	3	4	5	6	7	8	9	
22	**13**	**-5**	**-8**	**15**	**60**	**17**	**31**	**47**	**Mid=(9+1)/2=5**

1	2	3	4	5	6	7	8	9	
22	**13**	**-5**	**-8**	**15**	**60**	**17**	**31**	**47**	**Mid=(1+5)/2=3**

The above process continued until the size of the subproblem is either 1 or 2.



The tree of recursive calls of MaxMin is shown below:



MERGE SORT

Merge sort is the sorting technique that follows the divide and conquer approach. It is one of the most popular and efficient sorting algorithm.

It divides the given list into two equal halves, $a[1], \ldots, a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$ calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process.

The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

The algorithms MergeSort and Merge are shown below:

```
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 element
    {
        // 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 Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h]; h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

Example:

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 further splitted into two subarrays. This process continues until the size of the subarray is **one**.

Then the merging begins. This process continues for the second half of the array.

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

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:

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

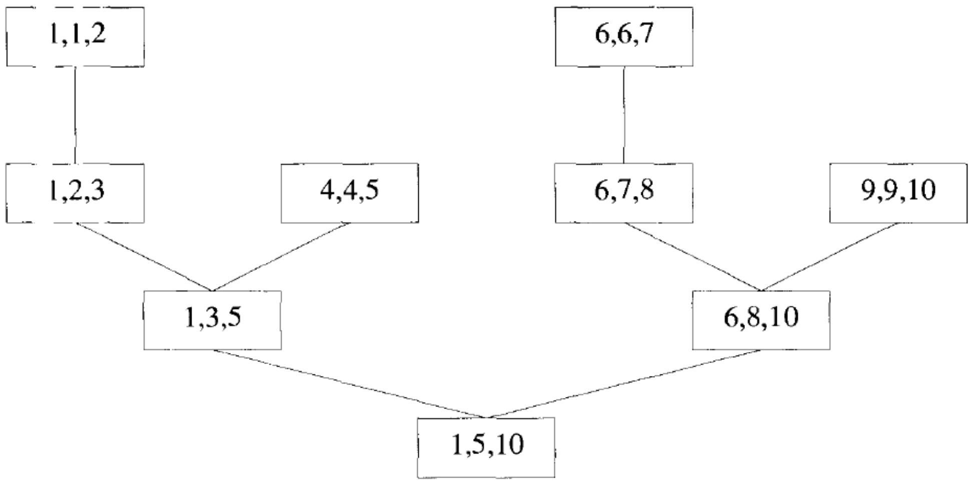
and then a[1:3] and a[4:5]:

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

The same procedure is followed for the second half of the array, and finally the array is sorted.

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

Trees of calls of Merge:



QUICK SORT

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.

A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

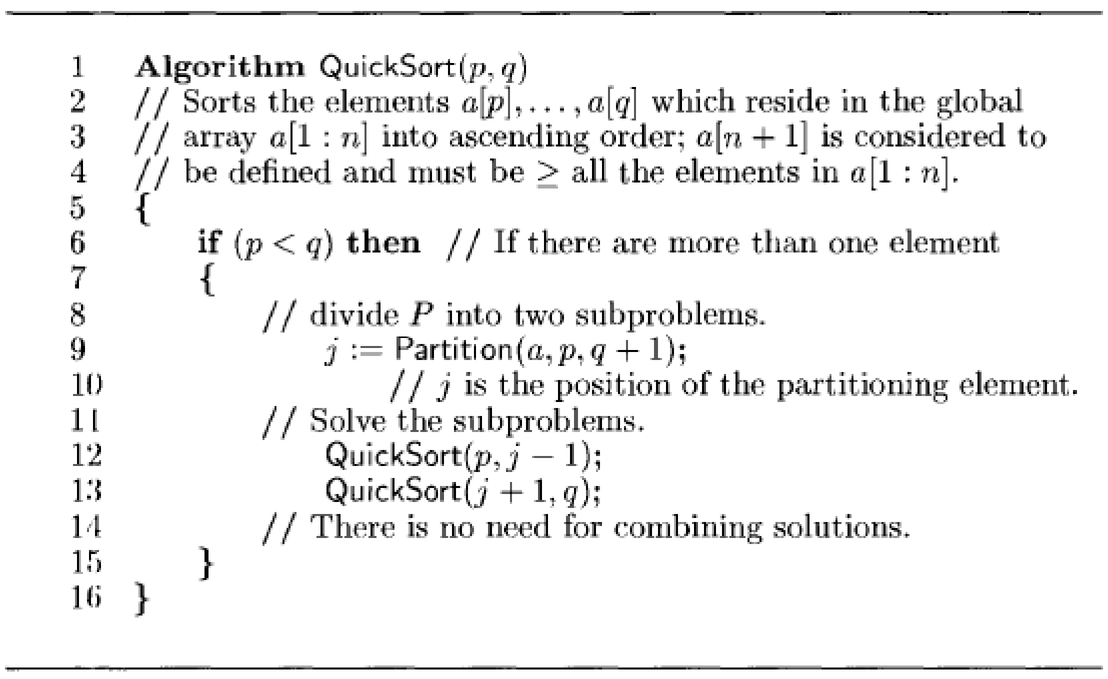
Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

The QuickSort and the partition algorithms are shown below:



```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \dots, a[q]$ 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 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 $p-1$, $a[k] \le t$ for $m \le k < q$, and $a[k] \ge t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \ge v$);

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

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

    } until ($i \ge j$);

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

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

Example: In this example, the pivot element is **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	$+\infty$		

Now, the list is divided into two halves. First half, the elements smaller than 65 and the second part, the elements are larger than 65. This procedure continues until the array will be sorted.