Chapter-Two
DIVIDE AND CONQUER METHOD

2. DIVIDE AND CONQUER

It's a very interesting algorithmic strategy.

General Method:

The method has following activities, for a given problem is

1. **Divide:** Divided into smaller sub problems,
2. **Conquer**: These sub problems are solved independently,
3. **Combine**: Combining all the solutions of sub problems into a solution of the whole

- If the sub problems are large enough then divide and conquer is reapplied.
- The general sub problems are usually of the same type as original problem. so, Recursive algorithms are used in divide and conquer strategy.
- A control abstraction for divide and conquer is as given below- using control abstraction a flow of control of a procedure is given.

The base case for the recursion is sub-problem of constant size.

Advantages of Divide & Conquer technique:

- For solving conceptually difficult problems like Tower of Hanoi, divide & conquer is a powerful tool
- Results in efficient algorithms
- Divide & Conquer algorithms are adapted for execution in multi-processor machines
- Results in algorithms that use memory cache efficiently.

Limitations of divide & conquer technique:

- Recursion is slow
- Very simple problem may be more complicated than an iterative approach.

Example: adding n numbers etc

Algorithm DC(P)

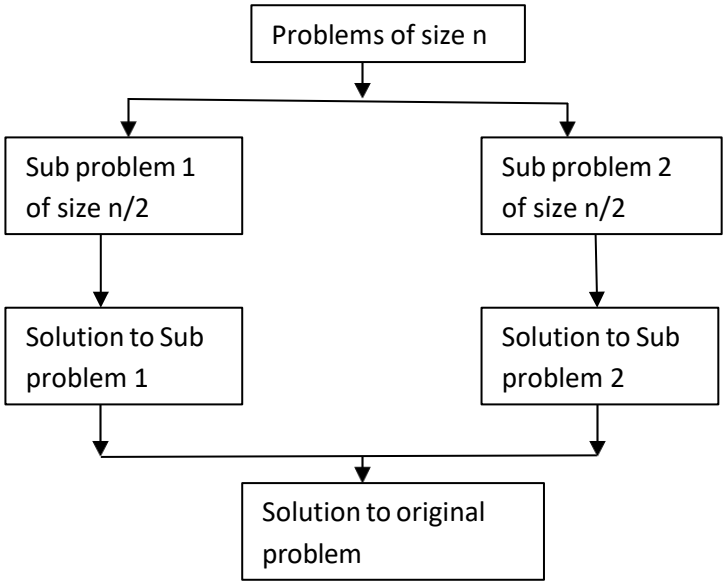
```
{
If P is too small then
Return solution of P
Else
{
Divide (P) and obtain P1,P2 .....Pn
where  n≥ 1
Apply DC to each sub problem
Return combines (DC(P1),DC(P2)…  DC(Pn));
}
}
```

The computing time of `the above procedure of divide and conquer is given by recurrence relation

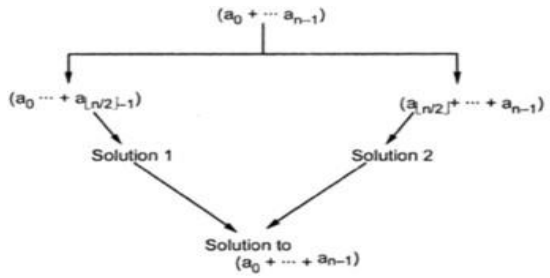
T(n)= {g(n) if n is small

T(n1)+T(n2)+…T(nr) +F(n) when n is sufficient.

Where T(n) is the time for divide and conquer of size n. the g(n) is the computation time required to solve small inputs. The F(n) is the time required required for dividing problem P and combining the solution to sub problems.
The divide and conquer method is given in the below figure.



The generated sub problems are usually of same type as the original problem. Hence sometimes recursive algorithms are used in divide and conquer method.
For example, if we want to compute sum of n numbers then by divide and conquer we can solve the problem as



If we want to divide a problem of size n into a size of n/b taking f(n) time to divide and combine, An instance of size n can be divided into b instances of size n/b, with "a" of them needing to be solved. [a ≥1, b > 1].Assume size n is a power of b.
The general divide and conquer recurrence relation is

T(n)=aT(n/b)+f(n)

T(n)- time for size of input n,
a= no.of instances,
T(n/b)= time for size n/b,
f(n)= time required for dividing the problem into sub problem. The order of growth of T(n) depends upon the constants a, b and order of growth function f(n).

Applications of divide and conquer method:

1. Binary search
2. Merge sort
3. Finding Maximum and Minimum

2.1 Binary search:

Binary Search is an efficient searching method. The most essential thing is that the elements in the array should be sorted one.

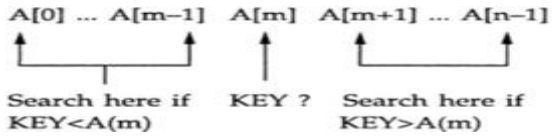
An element which is to be searched from the list of elements sorted in an array A[0,….,n-1] is called Key element.

Let A[m] be the mid element of the array A.

Then there are three conditions

1. If key=A[m] then desired element is present in the list.
2. Otherwise, if key <A[m] then search left sublist
3. Otherwise, if key >A[m] then search the right sublist.

This can be represented as



Algorithm:

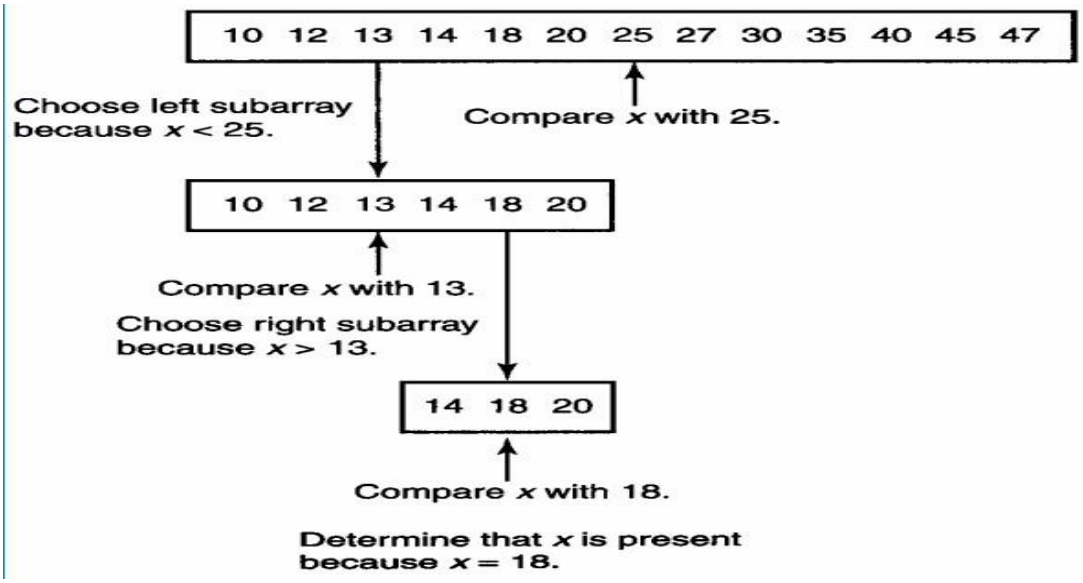
```
Algbinsearch(A[0,….,n-1],key)
//Problem description: this algorithm is for searching the
// element using binary search method.
//Input:an array A from which the KEY element to be searched
// Output:it retuns the index of the array element if it is equal to KEY otherwise it returns -1.
low← 0
high← n-1
while(low<high) do
{
m=(low+high)/2
if(key=A[m]) then
return m
else if(key <A[m]) then
high=m-1 // search left sublist
else
low=m+1   //search right sublist
}return -1  // if element is not present in the list.
```

Example:

Suppose the search element x=18 and the input array is 10,12,13,14,18,20,25,27,30,35,40,45,47

Here middle element A[m]=20

The middle element=0+12/2 = 6 th position of array is the middle element.



Analysis:

The basic operation in binary search is comparison of search key with the other array elements.

Depend on

Best – key matched with mid element

Worst – key not found or key sometimes in the list

To analyze efficiency of binary search, we must count the number of times the search key gets compared with the array elements. The comparison is also called a three-way comparison because algorithm makes the comparison to determine whether KEY is smaller, equal to or greater than A{m}.

In this algorithm, after one comparison the list of n elements is divided into n/2 sub list.

The worst-case efficiency is that the algorithm compares all the array elements for searching the desired element. In this method, one comparison is made and based on this comparison array is divided each time to n/2 sublist. Since after each comparison the algorithm divides the problem into half the size,hence the worst case complexity as

$C_{worst}(n) = C_{worst}(n/2) + 1$ for $n>1$

Also $C_{worst}(1)=1$.

Here $C_{worst}(n/2)$---------- time required to compare left or right sublist.

1------ one comparison made with middle element.

But as we consider the rounded down value when array gets divided the above equation can written as

$C_{worst}(n) = C_{worst}(n/2) + 1$ for $n>1$ ----------(1)

$C_{worst}(1)=1$ ---------- (2)

The above recurrence equation can be solved further. Assume $n=2^K$ the equation 1 becomes

$C_{worst}(2^K) = C_{worst}(2^K/2) + 1$ ---------- (3)

$C_{worst}(2^K) = C_{worst}(2^{K-1}) + 1$

Using backward substituiton method, we can substitute

$C_{worst}(2^{K-1}) = C_{worst}(2^{K-2}) + 1$
Then equation(3) becomes
$C_{worst}(2^{K}) = [C_{worst}(2^{K-2}) + 1]+1$
$= C_{worst}(2^{K-2}) + 2$
Then $C_{worst}(2^{K}) = [C_{worst}(2^{K-3}) + 1]+2$
$=[C_{worst}(2^{K-3}) + 3$
……
…..
…..
…..
$C_{worst}(2^{K})=[C_{worst}(2^{K-k}) + K]$
$= C_{worst}(2^{0}) + k$
$C_{worst}(2^{K})=[C_{worst}(1) + K]$
But as per equation(2),
$C_{worst}(1)= 1$ then we get equation(4),
$C_{worst}(2^{K})=[1+ K]$
As we have assumed,
$n=2^{K}$
taking logarithm(base 2) on both sides
$\log_2 n = \log_2 2^{K}$
$\log_2 n =K. \log_2 2$
$\log_2 n = K(1)$
therefore $\log_2 2=1$
$\therefore C_{worst}(n)= 1+ \log_2 n$
$\therefore C_{worst}(n)= \log_2 n$ for $n>1$
The worst case of complexity of binary search is $\Theta(\log n)$
As $C_{worst}(n)= 1+ \log_2 n$ we can verify equation(1) with this value
Considering equation (2),
$C_{worst}(n) =C_{worst}(n/2)+1$
For L.H.S
Assume n= 2!
$\therefore$ substitute
$C_{worst}(n) = \log_2 n+1$
$= \log_2(2i)+1$
$= \log_2 2 +\log_2 i+1$
$=\log_2 i+2$

For R.H.S
Assume n= 2!
$\therefore$ substitute
$C_{worst}(n/2)+1 = C_{worst}(2i/2)+1$
$= C_{worst}(i) +1$
As $C_{worst}(n) =\log_2 n+1$

In the same manner

$C_{worst}(i) + 1 = (\log_2 i + 1) + 1$

$= \log_2 i + 2$

Average case complexity:

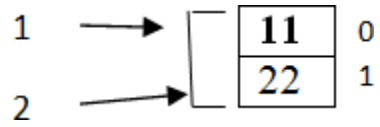
To obtain average case efficiency of binary search we will consider some samples of input n

If n=1

i.e there is only one element 11 is there

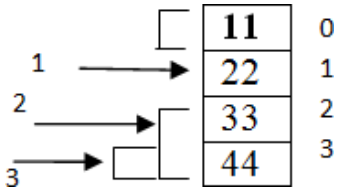
1 → [11] only one search is required to search some key

If n=2 and search key is 22



Two comparisons are made to search key 22

If n=4 and search key is 44



m=(0+3)/2 =1

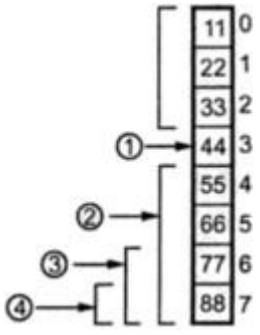
As A[1] = 22 and 22 < 44

Search right sublist again,

M=(2+3)/2 =2

Again A[2]=33 and 33< 44 we divide list.in right sublist A[4]=44 and key is 44. Thus, total 3 comparisons are made to search 44.

If n=8 and search key =88



m=(0+7)/2 =3

As A[3] =44 and 44 < 88 search right sublist, again

M=(4+7)/2 = 5

Again A[5] = 66 and 66< 88 search right sublist again

M=(6+7)/2 =6
But A[6] = 77 and 77 < 88
Then search right sublist
M= (7+7)/2 = 7
A[m]= A[7] = 88. Yes, the desired element is present .
Thus, total 4 comparisons are made to search 88

To summarize the above operations

N	Total comparison
1	1
2	2
4	3
8	4
16	5
…	….

Observing the above given table we can write as
$Log_2n+1 =c$
For instance, if n=2 then
$log_22 =1$
then c = 1+1 = 2
similarly n=8 then

$c = log_2n+1 = log_28+1 = 3+1 = 4$
thus, we can write as
average case complexity is $\Theta(\log n)$

The time complexity of binary search is

- ✓ Worst case: $\Theta(\log n)$
- ✓ Average case: $\Theta(\log n)$
- ✓ Best case: $\Theta(1)$

Applications of binary search:

- Number guessing game
- Word lists/search dictionary etc,

Advantages:

- Efficient on very big list
- Can be implemented iteratively/recursively

Limitations:

- Interacts poorly with the memory hierarchy
- Requires given list to be sorted
- Due to random access of list element, needs arrays instead of linked list.

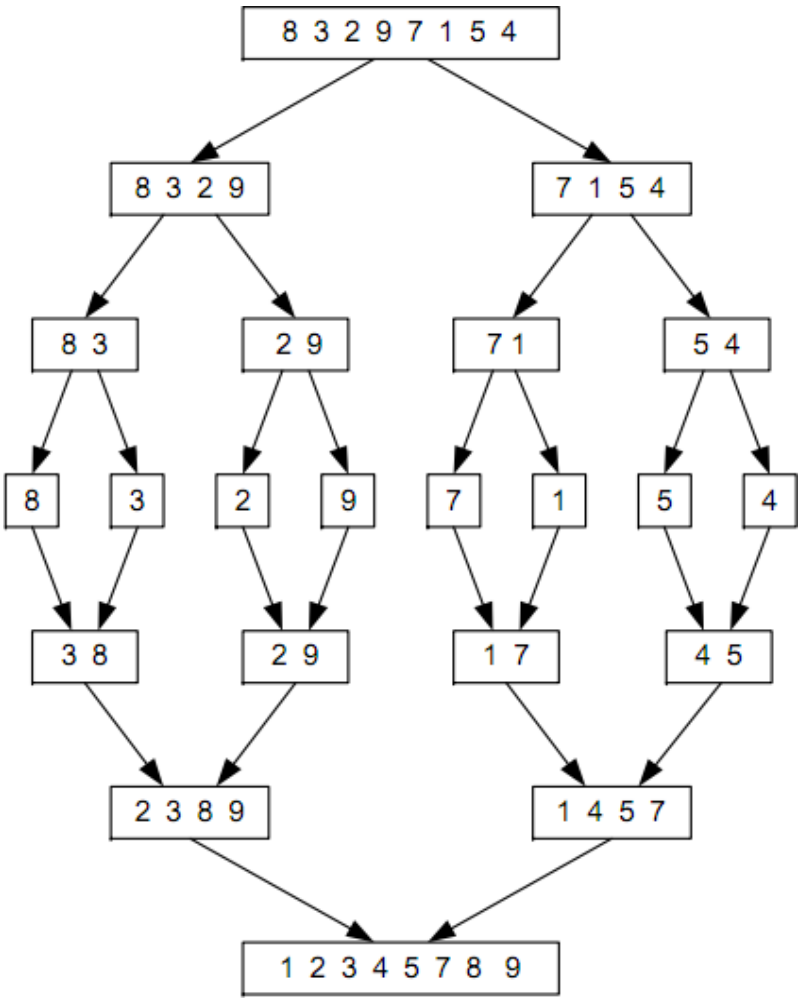
`

2.2 MERGE SORT:

Merge sort is a sort algorithm that splits the items to be sorted into two groups; recursively sorts each group, and merges them into a final sorted sequence. It has 3 steps

Divide: partition the array into two sublist s1 and s2 with n/2 element each
Conquer:then sort sublist s1&s2
Combine :Merge s1 and s2 into a unique group.
Features:

- It is a comparison based algorithm
- It is a stable algorithm
- It is a perfect example of divide & conquer algorithm design strategy



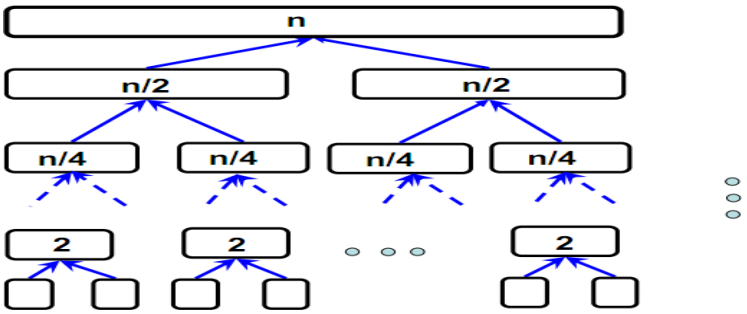
Algorithm Mergesort A[0,…n-1],low,high)
//Problem description: this algorithm is for sorting the elements using merge sort.
//Input: Array A of unsorted elements, low as beginning pointer of array A and high as end
//pointer of array A
//Output:Sorted array A[0,…n-1]
If (low < high) then
{
mid← (low+high)/2 //split the list at mid

```
Mergesort(A,low,mid)                 //first sublist
Mergesort(A,mid+1,high)              //second sublist
Combine(A,low,mid,high)              //Merging of two sublist

Algorithm Combine(low,mid,high)
{
K=low;     // k as index for array temporary
i=low;     // I as index for left sublist
j=mid+1;  // j as index for right sublist.
While(i<=mid and j<=high) do
{
If(A[i]<=A[j]) then     // if small element present in left sublist
{          // copy that smaller elements to temp array
Temp[k]=A[i];
i=i+1;
k=k+1;
}
Else       //smaller elements is present in right sublist
{
// copy that smaller elements to temp array
Temp[k]=A[j];  // if small element present in right sublist
j=j+1;
k=k+1;
}
}// copy remaining elements of left sublist to temp
While(i<=mid) do
{
Temp[k]=A[i];
 i=i+1;
k=k+1;
}
Copy remaining element right to temp
While(j<=high) do
{
Temp[K]=A[j];
j=j+1;
k=k+1;
}
```

Analysis:

In merge sort, two recursive calls are made. each recursive call focuses on n/2 elements of the list. After two recursive calls, one call is made to combine two sub list.



The recurrence relation is
T(n)=T(n/2)+T(n/2)+n
Where T(n/2)= time taken for the left sublist to be sorted.
T(n/2)=time taken for the right sublist to be sorted.
n=time taken for combining right and left sublist.Where n>1 i.e T(1)=0
Using Master's theorem,
As per theorem $T(n)=\Theta(n^d \log n)$ if a=b
Here T(n)=2T(n/2)+ n, Here a=2,b=2 with d=1
So,$T(n)=\Theta(n \log n)$
Here most of the work done in combining the solutions,

- **Best case: Θ(nlogn)**
- **Worst case: Θ(nlog n)**
- **Average case: Θ(nlogn)**

Applications of Merge Sort:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

Advantages of Merge Sort:

- **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(nlogn), which means it performs well even on large datasets.
- **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.

- **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

Finding Maximum and minimum

Algorithm for finding minimum element from the set of n numbers.

Algorithm Minimum_val(A[1,….n])
{
// Problem description:this algorithm is to find minimum value from array A to n elements.
min← A[1] //Assuming first element as min.
for(i← 2 to n) do
{
If(min>A[i]) then
min←A[i] //set new min value.
} return min;
}

Thus we require total(n-1) comparison to obtain minimum value . similarly we can obtain the maximum value from an array Ain(n-1) comparison.

The algorithm for finding maximum element:

Algorithm **Maximum_val**(A[1,…n])
{
//Problem description:This algorithm is to find maximum value from array A of n elements.
max← A[1]

```
for(i← 2 to n) do
{ if[A[i]>max) then
max← A[i]
}          Return max          }
```

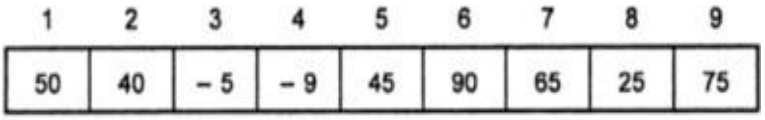
We can obtain maximum andminimum values from an array simultaneously following algorithm.

```
Algorithm Max_minA[1,..n],Max,Min)
{
```

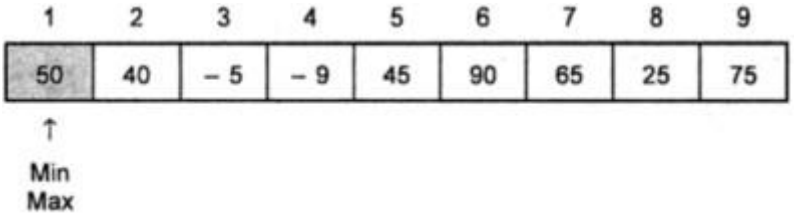
Problem description:This algorithm obtains min and max value simultaneously

```
Max← Min←A[1]
For( i← 2 to n) do
{
If(A[i]>Max) then
Max←A[i]      //Obtaining maximum value
If(A[i]<Min) then
Min←A[i]      //Obtaining minimum value
} }
```

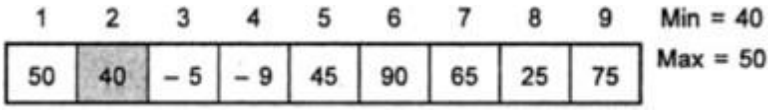
Thus we can get maximum element in max and minimum element in min variable.\
Example: consider the array



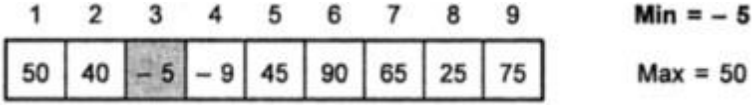
Step 1:



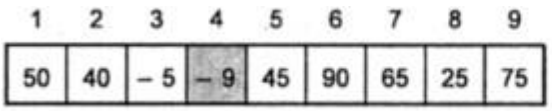
Step 2: now from index 2 to n we will compare an array element with Min and Max value.



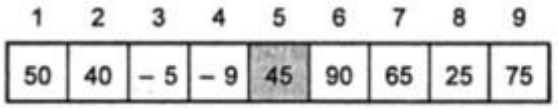
Step 3:



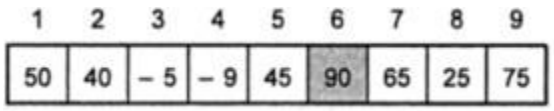
Step 4:



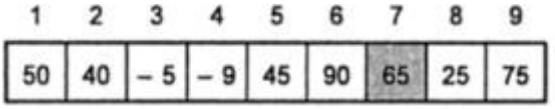
Step 5:



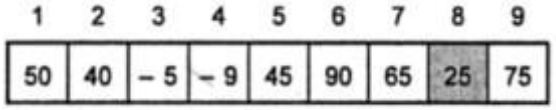
Step 6:



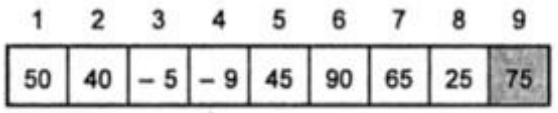
Step 7:



Step 8:



Step 9:



Step 10:

As we have reached at nth location of the array we will terminate the procedure of finding maximum and minimum values and print minimum values as -9 and maximum value as 90.

Analysis:

The above algorithm takes Θ(n) run n time. This is because the basic operation comparing array element with min or max value is done within a for loop.

The above algorithm is a straightforward algorithm of finding the maximum and minimum but we can obtain them using recursive method. The recursive algorithm makes use of divide and conquer strategy.

In this algorithm, the list is divided at the mid in order to obtain two sub lists. From both the sublist maximum and minimum elements are chosen. Two maxima and minima are compared and from then real maximum and minimum elements are determined. This process is carried out for entire list.

The algorithm is given below:

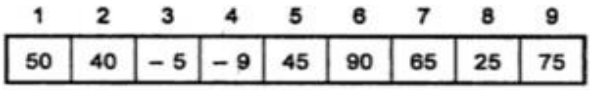
Algorithm Max_val(i,j,max,min)

//Problem description:Finding min, max elements recursively.

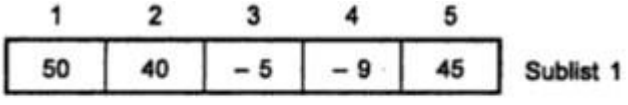
//input:I,j are integers used as index to an array A. the max and min will contain maximum and minimum value elements.

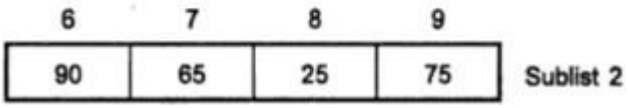
```
//output:None
If(i==j) then
{
max←A[i];
min←A[j];
}
Else if(i= j-1) then
{ if(A[i]< A[j])then
{
max←A[j];
min←A[i]
}else
{max←A[i]
min←A[j]
}          //end of els
}          //end of if.
Else
{
mid←(i+j)/2    // divide the list handling two lists separately.
Max_Min_val(i ,mid,max,min)
Max_Min_val(mid+1,j,max_new.min_new)
If(Max<max_new) then
max←max_new                    //combine solution
if(min<min_new) then
min←min_new                    // combine solution
}
```

Example: consider a list of some elements from which maximum and minimum element can be found out.

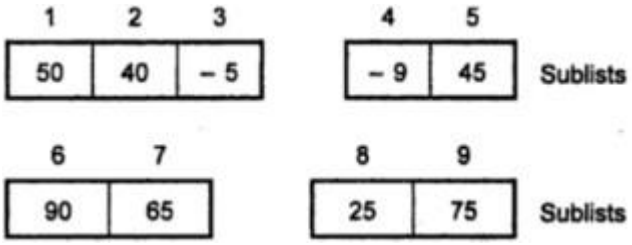


Step 1:





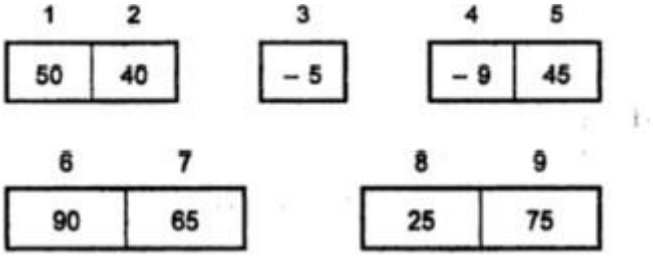
We have divided the original list at mid and two sub lists: sublist1 and sub list 2 are created. We will find min and max values respectively from each sub list.



Step 2:

Again divide each sub list and create further sub lists. then from each sub list obtain.

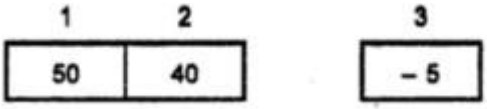
Step 3:



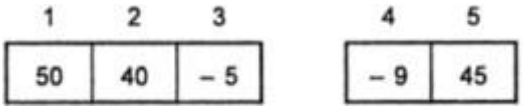
It is possible to divide the list(50,40,-5) further. Hence, we have divided the list into sub lists. And min, max values are obtained.

Step 4:

Now further division of the list is not possible. Hence, we start combining the solutions of min and max values from each sub list.



Combine (1,2) and(3) Min= -5 , Max=50.



Combine (1,..3) and(4,5) Min =-9 , Max =50

Now we will combine (1,..3) and (4,5) and the min and max values among them are obtained. Hence,

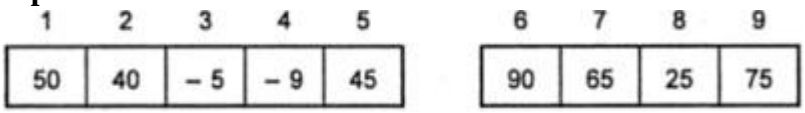
Min value= -9

Max value = 50

Step 5:

Combine (6,7) and (8,9) Min=25, Max=90

Step 6:



Combine th sublists_1,…5) and (6,..9).find out min mad max values which are min = -9 and max=90.

Thus, the complete list is formed from which the min and max values are obtained. Hence final min and max values min=-9 and max=90

.

The two recursive calls made in this algorithm for each half divided sublist. Hence time required for computing min nad max will be\

$T(n) = T(n/2) + T(n/2) + 2$ when $n>2$

$T(n) = 1$ when $n=2$

If single element is present in the list then $T(n)=0$

Now, time required for computing min and max will be

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

$= 2\{2T(n/4)+2]+2$

$=2(2[2T(n/8)+2]+2)$

$=8T(n/8) +10$

Continuing in this fashion a recursive equation can be obtained. If we put $n=2^k$ then

$T(n) =2^{K-1} T(2) +\sum_{i=1}^{k-1} 2^i$

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

$=3n/2 -2$

So the time complexity **O(n).**

Quick Sort:

Quick sort is a sorting alogorithm that uses the divide and conquer strategy,In this method, division is dynamically carried out.

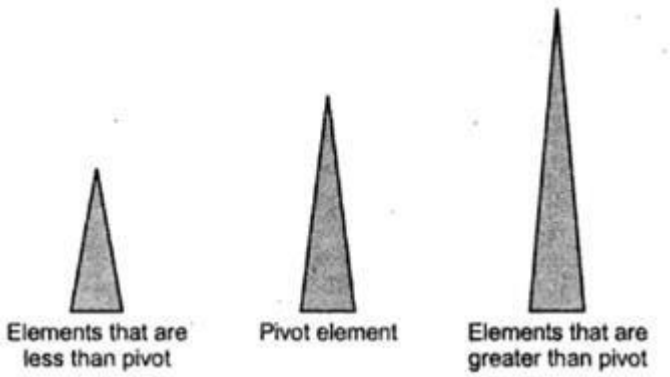
The **three steps of Quick sort** are as follows:

Divide: Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element.The splitting of the array into two sub arrays is based on the pivot element .All the lements that are less than pivot should be in the left sub array and all the elements that are more than pivot should be in the right subarray.

Conquer: Recursively sort the two sub arrays.

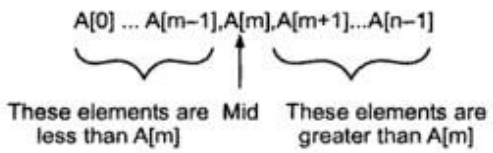
Combine: Combine all the sorted elements in a group to form a list of sorted elements.

In this,the division is based on actual value of the element.Consider an array A[i] where I is ranging fron 0 to n-1 .



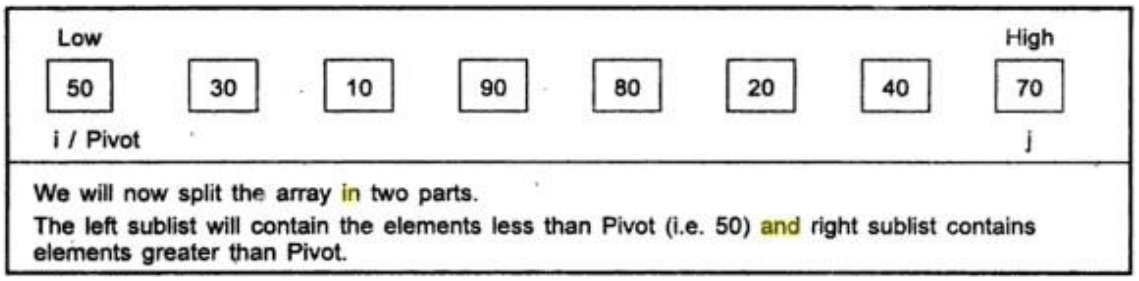
then we can formulize the division of array elements as

Let us understand this algorithm with the help of some example.



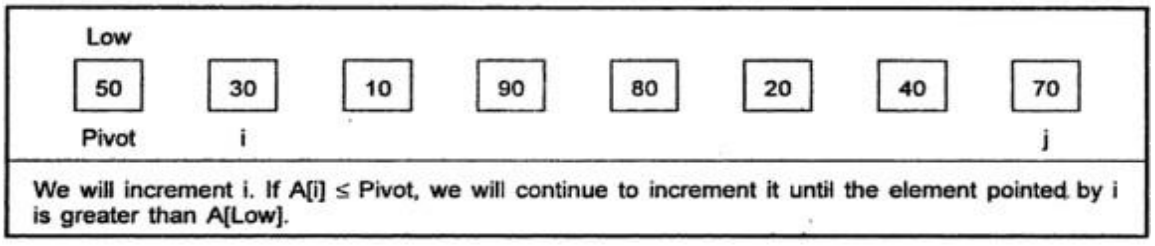
Example :

Step 1 :



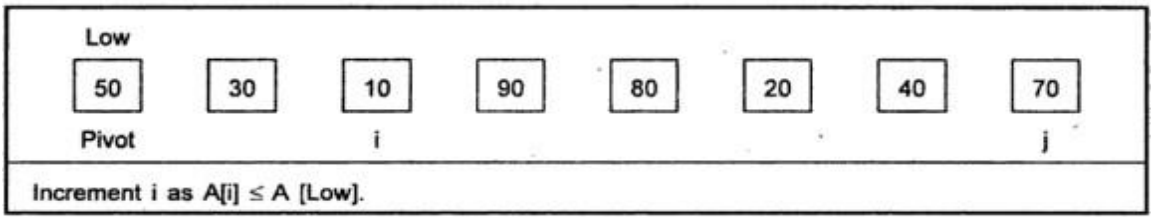
We will now split the array in two parts.
The left sublist will contain the elements less than Pivot (i.e. 50) and right sublist contains elements greater than Pivot.

Step 2 :



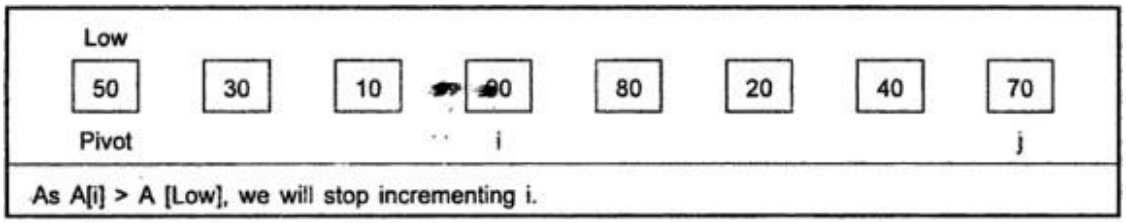
We will increment i. If A[i] ≤ Pivot, we will continue to increment it until the element pointed by i is greater than A[Low].

Step 3 :



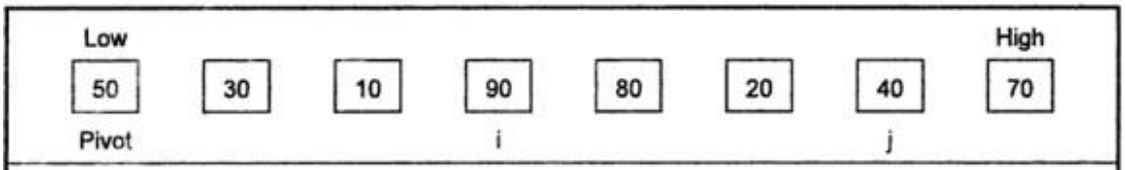
Increment i as A[i] ≤ A [Low].

Step 4 :



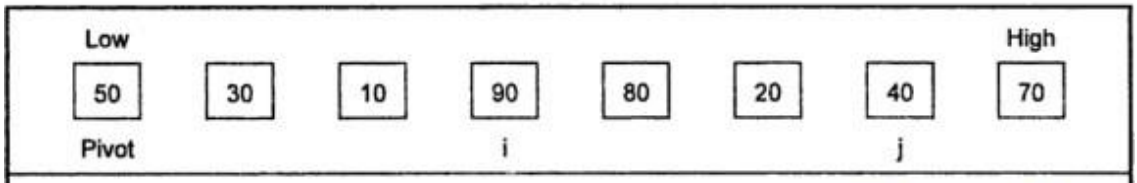
As A[i] > A [Low], we will stop incrementing i.

Step 5 :



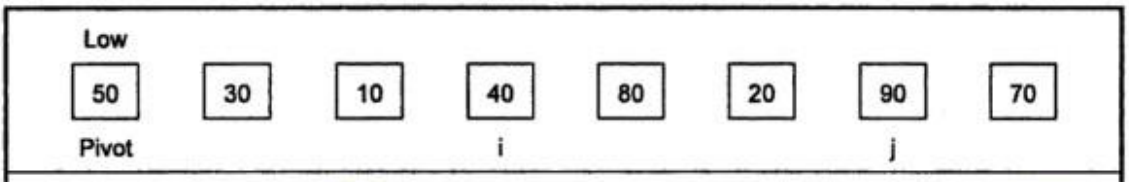
As A[j] > Pivot (i.e. 70 > 50). We will decrement j. We will continue to decrement j until the element pointed by j is less than A [Low].

Step 6 :



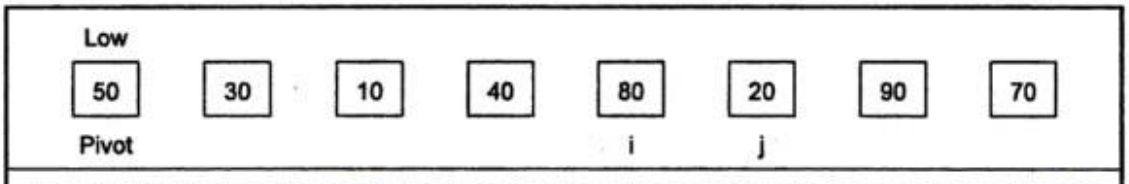
Now we can not decrement j because 40 < 50. Hence we will swap A[i] and A[j] i.e. 90 and 40.

Step 7 :



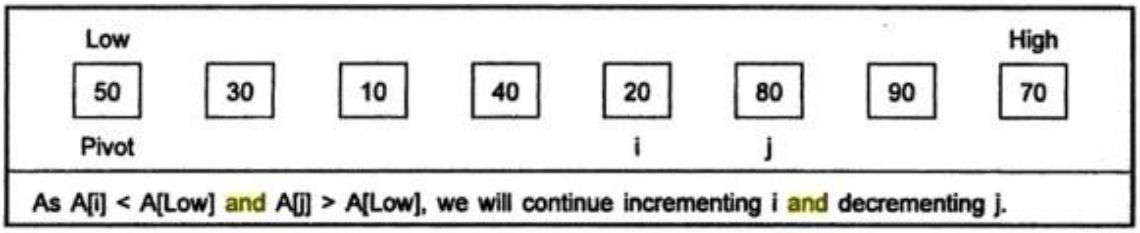
As A[i] is less than A[Low] and A[j] is greater than A[Low] we will continue incrementing i and decrementing j, until the false conditions are obtained.

Step 8 :



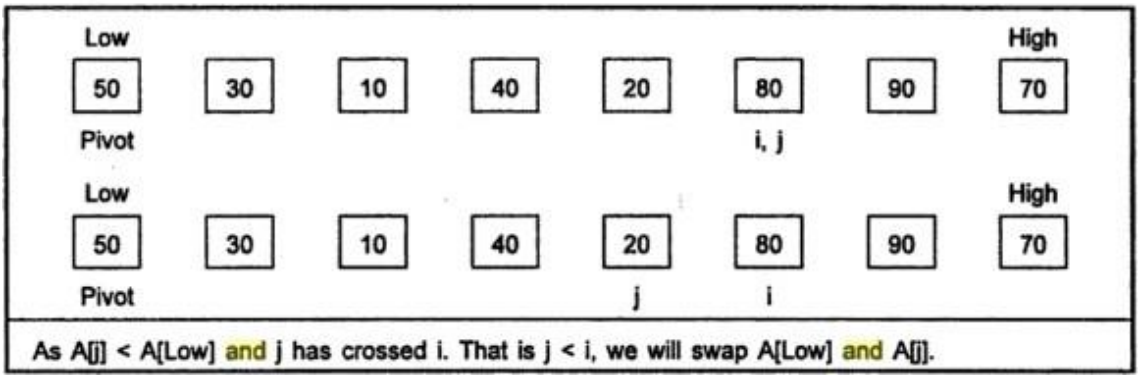
We will stop incrementing i and stop decrementing j. As i is smaller than j we will swap 80 and 20.

Step 9 :



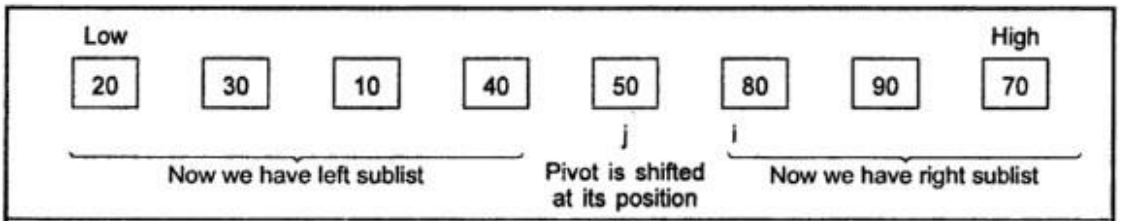
As A[i] < A[Low] and A[j] > A[Low], we will continue incrementing i and decrementing j.

Step 10 :



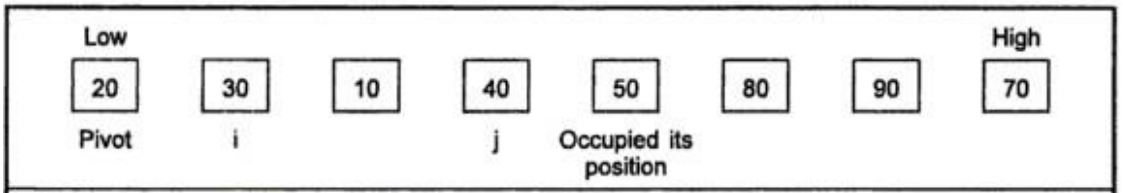
As A[j] < A[Low] and j has crossed i. That is j < i, we will swap A[Low] and A[j].

Step 11 :



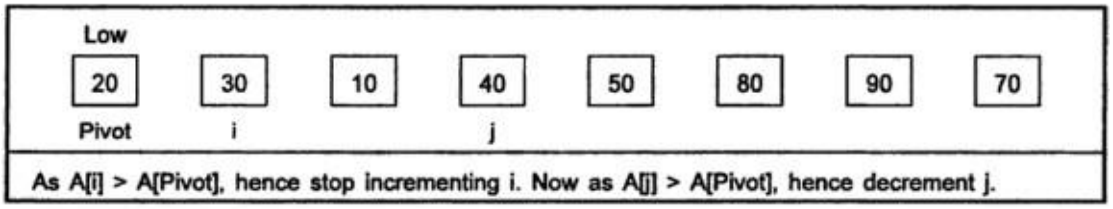
We will now start sorting left sublist, assuming the first element of left sublist as pivot element. Thus now new pivot = 20.

Step 12 :

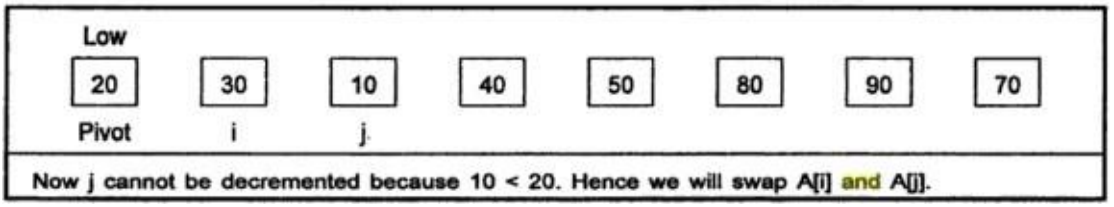


Now we will set i and j pointer and then we will start comparing A[i] with A[Low] or A[Pivot]. Similarly comparison with A[j] and A[Pivot].

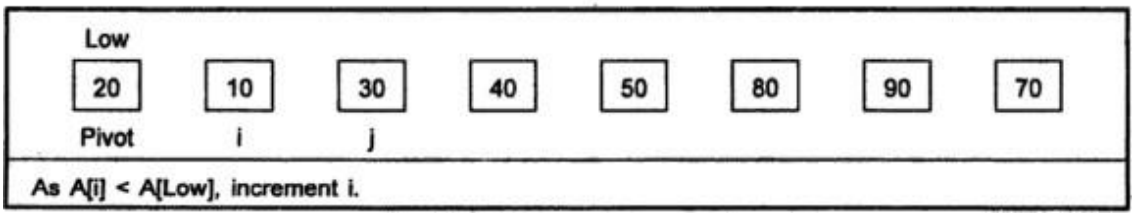
Step 13 :



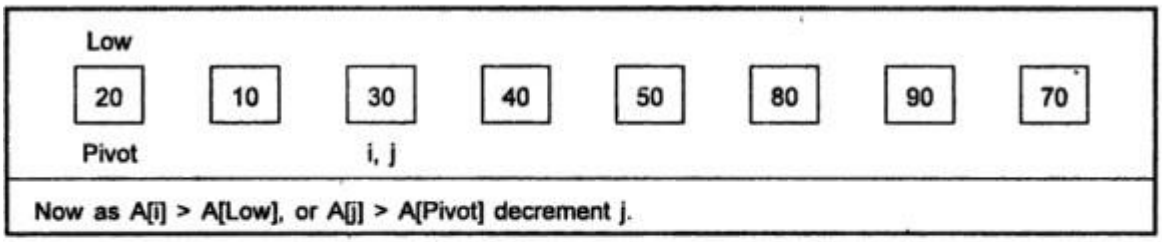
Step 14 :



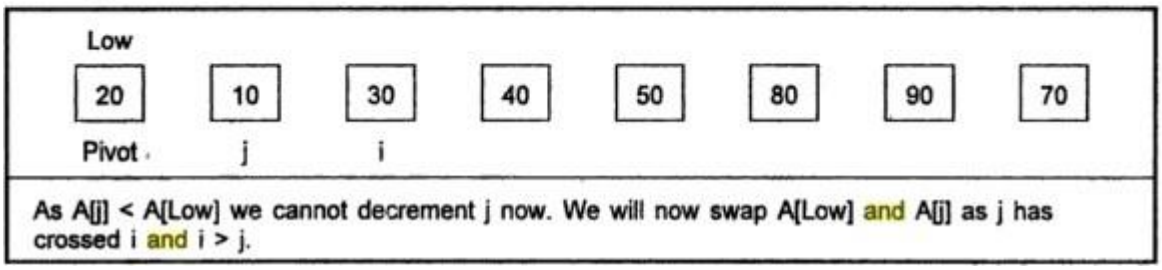
Step 15 :



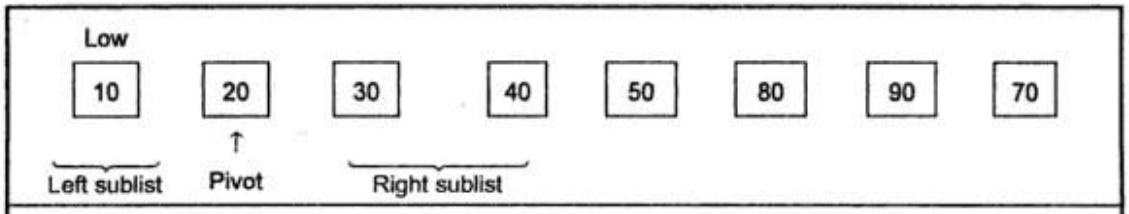
Step 16 :



Step 17 :

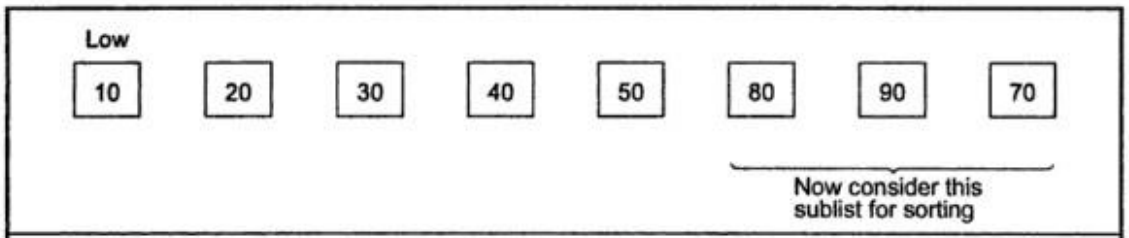


Step 18 :



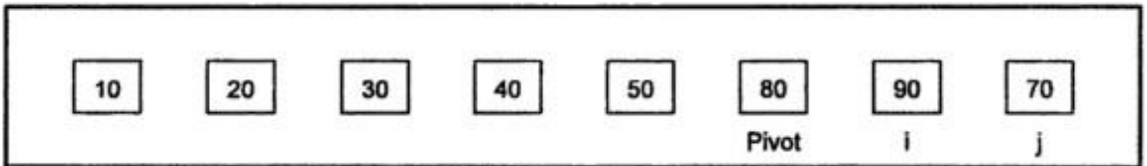
As there is only one element in left sublist hence we will sort right sublist.

Step 19 :



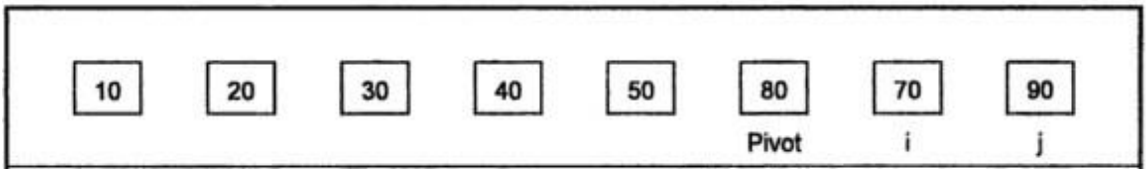
As left sublist is sorted completely we will sort right sublist, assuming first element of right sublist as Pivot.

Step 20 :



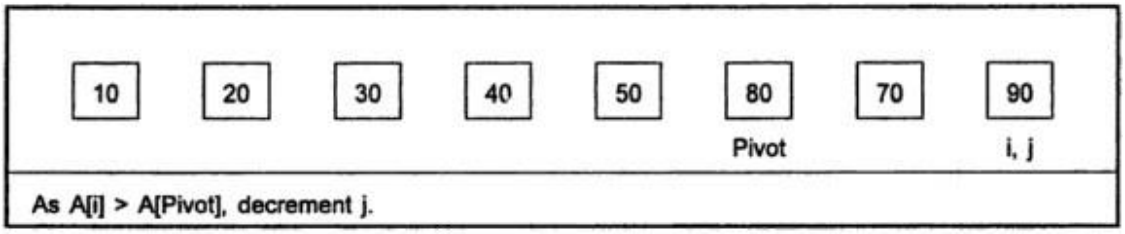
As A[i] > A[Pivot], hence we will stop incrementing i. Similarly A[j] < A[Pivot]. Hence we stop decrementing j. Swap A[i] and A[j].

Step 21 :

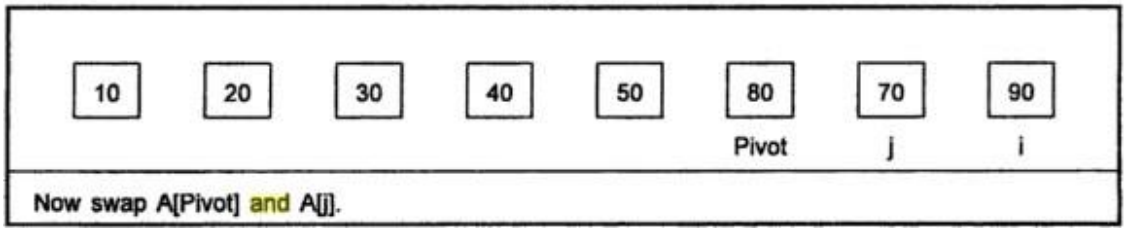


As A[i] < A[Pivot], increment i.

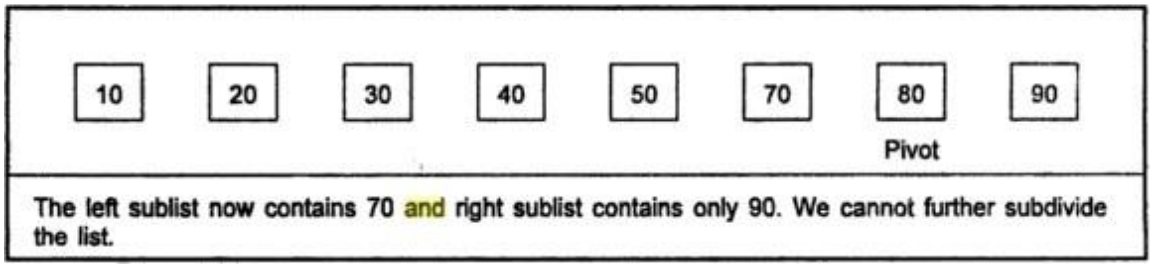
Step 22 :



Step 23 :



Step 24 :



Hence list is

This is a sorted list.

Algorithm:

The quick sort algorithm is performed using following two important functions –Quick and partition.

AlgorithmQuick(A[0,..n-1],low,high)
//Problem Description: This algorithm performs sorting of the elemnts given in Array a[0,…,n-1]

//Input: An Array A[0,..n-1] in which unsorted elements are given. The low indicated the leftmost element in the list and high indicates the rightmost element in the list.

//Output:Creates a sub array which is sorted in ascending order

If(low<high) then

//Split the array into two sub arrays

M ← partition (A[low….high]) // m is the mid array

Quick(A[low…m-1])

Quick(A[mid+1…high])

In above algorithm call to partition algorithm is given. The partition performs arrangement of the elements in ascending order. The recursive quick routine is for dividing the list in two sub lists.

The pseudo code for partition is as given below:

Algorithm Partition (A[low…high])

//Problem Description: The algorithm partitions the sub array using the first element as pivot element.

//Input: A sub array A with low as left most index of the array and high as the rightmost index of the array

//Output: The partitioning of array A is done and pivot occupies its proper position and the rightmost index of the list is returned.

Pivot← A[low]

i← low

j← high +1

While (i<=j) do

{

While (A[i] <= pivot) do

i← i+1

while(A[j]>=pivot) do

j ←j-1;

if(i<=j) then

swap(A[i],A[j]) //swap A[i] and A[j]

}

Swap (A[low,A[j]) //when I crosses j swap A[low] and A[j]

Return j //rightmost index of the list

The Partition function is called to arrange the elements such that all the elements that are less than pivot are at the left side of pivot and all the elements that are greater than pivot are all at the right of pivot.

Analysis:

Best Case(Split in the middle)

If the array is always partitioned at the mid, then it brings the best case efficiency of an algorithm.

The recurrence relation for quick sort for obtaining best case time complexity is

$$C(n) = \underbrace{C(n/2)}_{\text{Time required to sort left sub array}} + \underbrace{C(n/2)}_{\text{Time required to sort right sub array}} + \underbrace{n}_{\text{Time required for partitioning the sub array}} \quad \text{... equation (1)}$$

and $C(1) = 0$

Method 1 :Using Masters Theorem

The master theorem is

If $f(n) \in \Theta(n^d)$ then

1. $T(n) = \Theta(n^d)$ if $a < b^d$
2. $T(n) = \Theta(n^d \log n)$ if $a = b^d$
3. $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

We get,

$C(n) = 2\,C(n/2) + n$

Here $f(n) \in n^1$ i.e $d=1$

Now $a=2$ and $b=2$

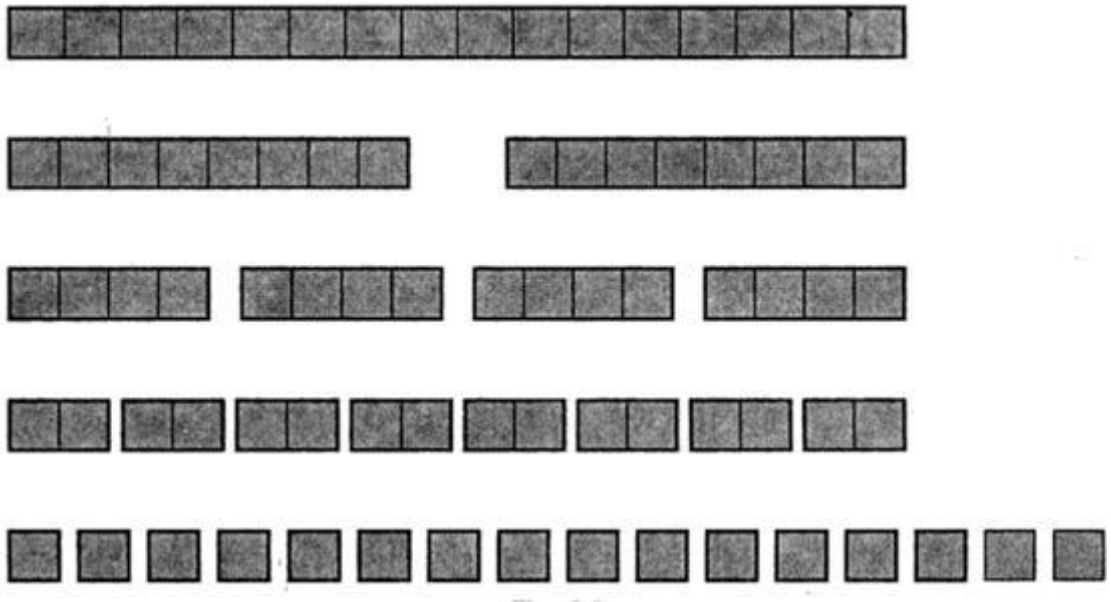
As from case 2, we get $a=b^d$ i.e $2=2^1$,we get

$T(n)$ i.e $C(n) = \Theta(n^d \log n)$

$C(n) = \Theta(n \log n)$

Thus, the best case time complexity of quick sort is $\Theta(n \log n)$

The best case of quick sort can be represented by



Method 2 : Using Substitution Method:

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

$C(n) = 2\,C(n/2) + n$

We assume $n=2^K$ since each time the list is divided into two equal halves. Then the equation becomes,

$C(2^K) = 2\,C(2^K/2) + 2^K$

$= 2\,C(2^{k-1}) + 2^K$

Now substitute $C(2^{k-1}) = 2\,C(2^{K-1}) + 2^{k-1}$

We get $C(2^K) = 2[2\,C(2^{K-1}) + 2^{k-1}] + 2^K$

$C(2^K) = 2^2\,C(2^{(k-2)} + 2^{k-1}] + 2^K$

$= 2^2\,C(2^{(k-2)}) + 2.2^{K-1} + 2^K$

$= 2^2\,C(2^{(k-2)}) + 2^K + 2^K$

$C(2^K) = 2^2\,C(2^{(k-2)}) + 2.2^K$

If we substitute $C(2^{(k-2)})$ then,

$C(2^K) = 2^2\,C(2^{(k-2)}) + 2.2^K$

$= 2^2\,[2C(2^{k-3}) + 2^{k-2}] + 2.2^k$

$= 2^3\,C(2^{k-3}) + 2^2 \cdot 2^{k-2} + 2.2^k$

$= 2^3\,C(2^{k-3}) + 2^k + 2.2^k$

$C(2^K) = 2^3\ C(2^{k-3})\ + 3.2^k$

Similarly, we can write

$C(2^K) = 2^4\ C(2^{k-4})\ + 4.2^k$

……

…..

…..

$= 2^k\ C(2^{k-k})\ + k.2^k$

$= 2^k\ C(2^0)\ + k.2^k$

$C(2^K) = 2^k\ C(1)\ + k.2^k$

But C(1) =0 hence the above equation becomes

$C(2^K) = 2^k .0\ + k.2^k$

Now we assume $n=2^K$ we can also say\

$K= \log_2 n$ (By taking logarthim on both sides)

$C(n) = n.0 + \log_2 n\ .\ n$

$C(n) = n\ \log n$

Thus it is proved that best case complexity of quick sort is Θ (n log n)

Wosrt case (sorted array):

The worst case for quick sort when the pivot is a minimum or maximum of all the elements in the list.

We can write it as

$C(n) = C(n-1) + n$

Or $C(n) = n + (n-1) + (n-2) + \ldots 2 + 1$

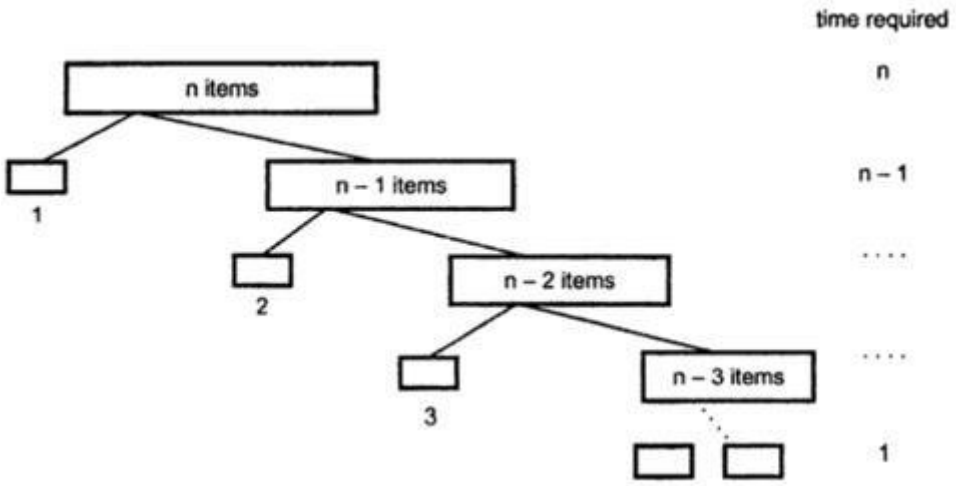
But as we know

$1 + 2 + 3 + \ldots n = n(n+1)/2 = \frac{1}{2} n^2$

$C(n) = \Theta (n^2)$

The time complexity of worst case of quick sort is $\Theta (n^2)$

This can be graphically representes as



Average case:

Let $C_{avg}(n)$ denotes the average time of quicksort.

C(n) =C(0) +C(n-1)+ n

Or C(n) =C(1) +C(n-2)+ n

Or C(n) =C(2) +C(n-3)+ n

….

…

….

C(n) =C(0) +C(n-1)+ n

The average value of C(n) is sum of all the possible values divided by n

i.e$C_{avg}(n)$ = 2/n (C(1) +C(2)+….+ C(n-1))+ n

Multiplying both sides by n we get,

n *$C_{avg}(n)$ = 2 ($C_{avg}(1)$ +$C_{avg}(2)$+….+ $C_{avg}(n-1)$)+ n^2---------- 1

Now we put n=n-1 then,

(n-1) *$C_{avg}(n-1)$ = 2 (C_{avg} (1) + C_{avg} (2)+….+ C_{avg} (n-2))+ (n-1) * (n-1) ----- 2

Subtract 1-2

n *$C_{avg}(n)$ -(n-1) * $C_{avg}(n-1)$

= 2 (C_{avg} (1) + C_{avg} (2)+….+ C_{avg} (n-2))+ (n-1) * (n-1)– (2 (C_{avg} (1)+C_{avg} (2)+….+ C_{avg} (n-1))+ n^2)

=2 C_{avg} (n-1)- n^2 +$(n-1)^2$ $(a-b)^2 = a^2 - b^2 + 2ab$

=2 C_{avg} (n-1)- n^2 + n^2 -1+2n

n *C_{avg}(n) -(n-1) * C_{avg}(n-1) =2 C_{avg} (n-1)+(2n-1)

n *C_{avg}(n) = (n-1) * C_{avg}(n-1) +2 C_{avg} (n-1)+(2n-1)

n *C_{avg}(n) = n * C_{avg}(n-1) - C_{avg}(n-1) +2 C_{avg} (n-1)+(2n-1)

n *C_{avg}(n) = n * C_{avg}(n-1) +C_{avg} (n-1)+(2n-1)

n *C_{avg}(n) =(n+1) * C_{avg}(n-1) +(2n-1)

$$n * C_{avg}(n) = (n+1) * C_{avg}(n-1) + (2n-1) < (n+1) * C_{avg}(n-1) + 2n$$

If we assume

$$(n+1) * C_{avg}(n-1) + 2n = (n+1) * C_{avg}(n-1) + C'n$$

Then,

$$(n+1) * C_{avg}(n-1) < (n+1) * C_{avg}(n-1) + C'n$$

Divide this equation by n (n + 1) then

$$n * C_{avg}(n) < (n+1) * C_{avg}(n-1) + C'n$$

$$\frac{C_{avg}(n)}{(n+1)} < \frac{C_{avg}(n-1)}{n} + \frac{C'}{(n+1)}$$

If we assume

$$A(n) = \frac{C_{avg}(n)}{(n+1)}$$ then

If we assume

$$A(n) = \frac{C_{avg}(n)}{(n+1)} \text{ then}$$

$$A(n) < A(n-1) + \frac{C'}{(n+1)}$$

$$A(n-1) < A(n-2) + \frac{C'}{n}$$

$$A(n-2) < A(n-3) + \frac{C'}{n-1}$$

$$\dots$$

$$\dots$$

$$\dots$$

$$A(1) < A(0) + \frac{C'}{2}$$

Adding and cancelling these equations we get,

$$A(n) < C' * \left[\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} + \frac{1}{n+1}\right]$$

$$\therefore C_{avg}(n) = (n+1) * A(n)$$

$$\text{i.e. } (n+1) * A(n) < C''(n+1)\log n$$

$$\therefore C_{avg}(n) = \Theta(n \log n)$$

Time complexity of quick sort

Best case	Average case	Worst case
$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n^2)$

Selection Sort:

Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element scan the entire list to find the smallest element and swap it with the second element. Then starting from the third element the entire list is scanned in order to find the next smallest element. Continuing in this fashion we can sort the entire list.

Generally on the pass i(0 < = i < = n-2) ,the smallest element is searched among lase n-1 elements and is swapped with A[i]

$$A[0] \le A[1] \le \dots \le A[i-1] \mid \overbrace{A[i], \dots, A[k], \dots, A[n-1]}^{\text{Last } n-i \text{ elements}}$$

A[k] is smallest element
so swap A[i] and A[k]

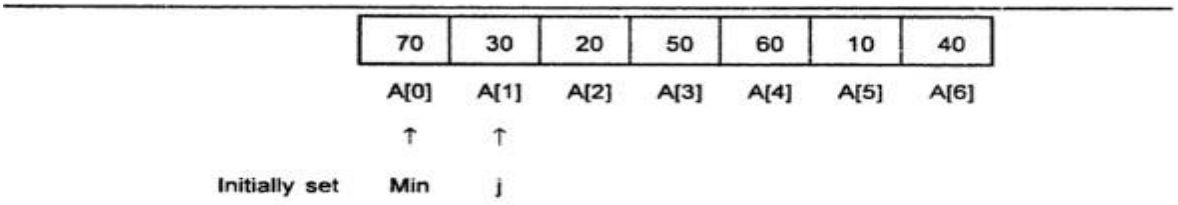
The list gets sorted after n–1 passes.

4.3.1.1 Example

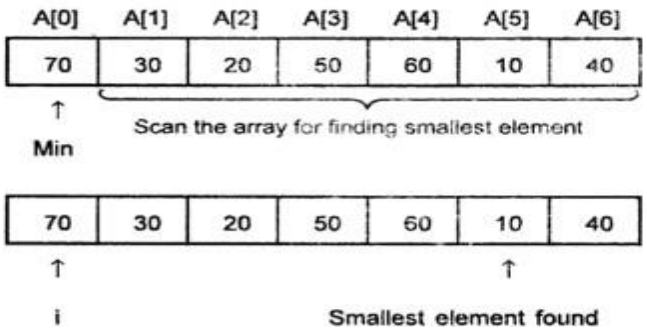
Consider the elements

70, 30, 20, 50, 60, 10, 40

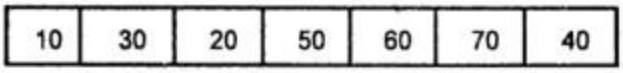
We can store these elements in array A as :



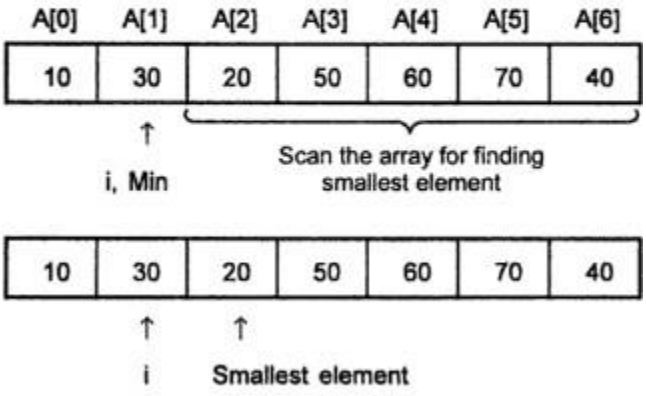
1st pass :



Now swap A[i] with smallest element. Then we get,



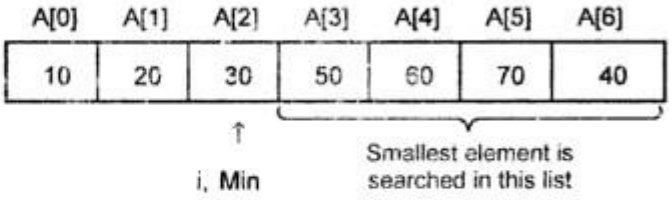
2nd pass :



Swap A[i] with smallest element. The array becomes,

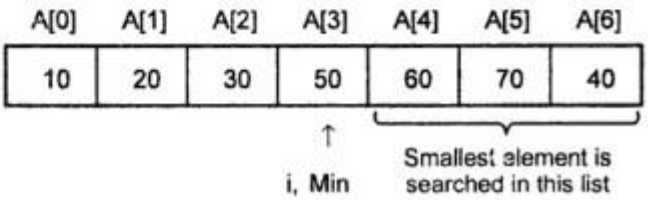
10	20	30	50	60	70	40

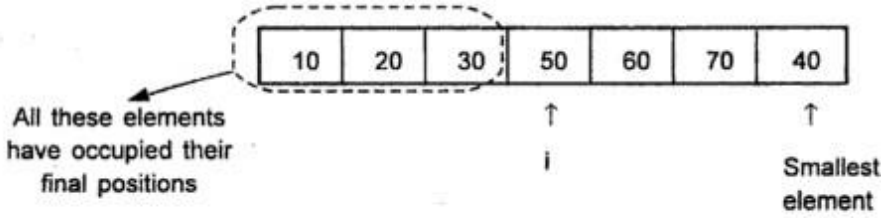
3rd pass :



As there is no smallest element than 30 we will increment i pointer.

4th pass :

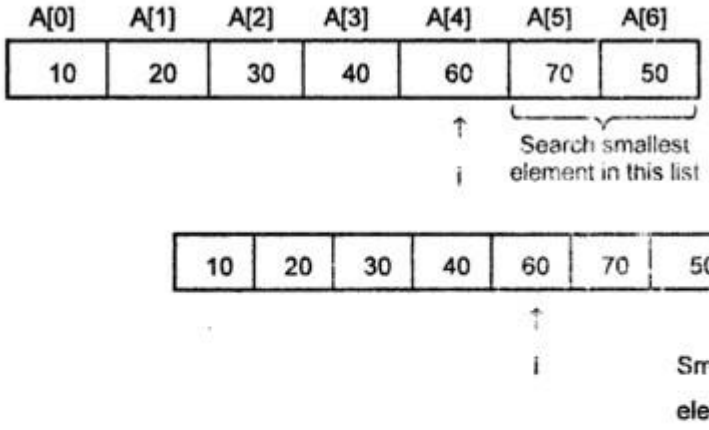




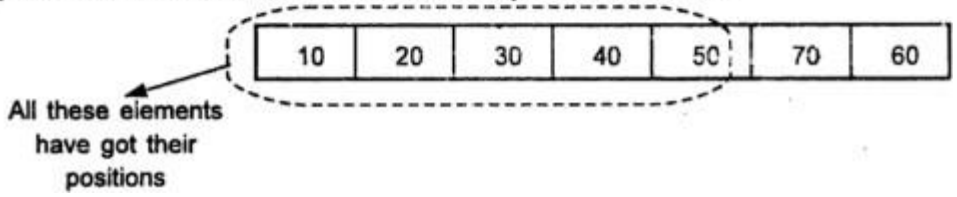
Swap A[i] with smallest element. The array then becomes,

10	20	30	40	60	70	50

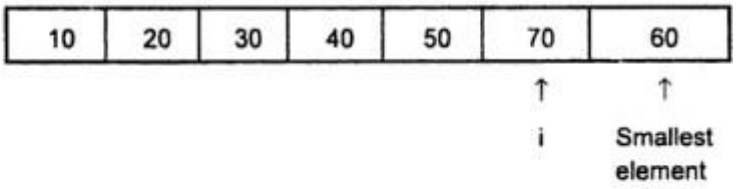
5th pass :



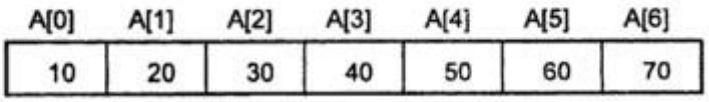
Swap A[i] with smallest element. The array then becomes,



6th pass :



Swap A[i] with smallest element. The array then becomes,



This is a sorted array.

Algorithm:

The Pseudo code for sorting the elements using selection sort is as given below:

Algorithm Selection(A[0….n-1])

//Problem Description : This algorithm sorts the elements using selection sort

//Input:An array of elements A[0..n-1] that is to be sorted

//Output:The sorted array A[0…n-1]

For i← 0 to n-2 do

{

Min ← i

For j← i+1 to n-1 do

{

If A[j] < A[min]

Min ← j

//swap a[i] and A[min]

}// end of the inner for loop

Temp ← A[i]

A[i] ← A[min]

A[min] ← Temp

}// end of the outer for loop

Analysis:

The above algorithm can be analyzed mathematically. We will apply a general plan for non-recursive mathematical analysis.

Step 1: The input size is n i.e Total number of elements in the list.

Step 2 : In the algorithm the basic operation is key comparison\

If A[j] < A[min]

Step 3: This basic operation depends upon on array size n. Hence we can find sum As

$$C(n) = \underset{\text{with variable i}}{\text{Outer for loop}} \times \underset{\text{with variable j}}{\text{Inner for loop}} \times \text{Basic operation}$$

$$\therefore \quad C(n) = \sum_{i=0}^{n-2} \left[\sum_{j=i+1}^{n-1} 1 \right]$$

$$C(n) = \sum_{i=0}^{n-2} (n - 1 - i)$$

If $\sum_{i=0}^{n} 1 = (n - 0 + 1)$ using this formula we get,

$$\sum_{j=i+1}^{n-1} 1 = [(n - 1) - (i + 1) + 1]$$

$$= (n - 1 - i)$$

Step 4 : Simplifying sum we get,

$$C(n) = \sum_{i=0}^{n-2} (n - 1 - i)$$

$$= \sum_{i=0}^{n-2} (n - 1) - \left[\sum_{i=0}^{n-2} i \right]$$

$$= \sum_{i=0}^{n-2} (n - 1) - \frac{(n-2)(n-1)}{2}$$

If $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ using this formula

$$\sum_{i=0}^{n-2} i = \frac{(n-2)(n-2+1)}{2}$$

$$= \frac{(n-2)(n-1)}{2}$$

Now taking (n – 1) as common factor we get,

$$C(n) = (n - 1) \left[\sum_{i=0}^{n-2} 1 \right] - \frac{(n-2)(n-1)}{2}$$

As $\sum_{i=1}^{n} 1 = (n - 1 + 1)$, we get,

$$\sum_{i=0}^{n-2} 1 = (n - 2 - 0 + 1) = (n - 1)$$

$$= (n - 1)(n - 1) - \frac{(n-2)(n-1)}{2}$$

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

Solving this equation we will get,

$$= \frac{2(n-1)(n-1) - (n-2)(n-1)}{2}$$

$$= \frac{2(n^2 - 2n + 1) - (n^2 - 3n + 2)}{2}$$

$$= \frac{n^2 - n}{2}$$

$$= \frac{n(n-1)}{2}$$

$$\approx \frac{1}{2}(n^2)$$

$= \Theta(n^2)$ for all input. But total number of key swaps is only $\Theta(n)$