Sorting Techniques :-

Let, A be a list of 'n' numbers. Sorting A refers to the operation of rearranging the elements of A so that they are in increasing order or decreasing order. i.e $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[n]$ or $A[1] \geq A[2] \geq \dots \geq A[n]$

Bubble Sort :-

A well known algorithm called bubble sort proceeds by scanning the list from left to right, and whenever a pair of adjacent keys is found to be out of order, then those items are swapped. This process repeats. The bubble sort algorithm works as follows :

Step-1 :-

Compare A[0] and A[1] and arrange them in the desired order, so that $A[0] \leq A[1]$, then compare A[1] and A[2] and arrange them so that $A[1] \leq A[2]$ continue until we compare A[n-2] with A[n-1] and arrange them so that $A[n-2] \leq A[n-1]$. This step involves n-1 comparisions. During this step the largest element is bubbled up to the (n-1)th position.

Step-2 :-

Repeat step 1 with one less comparision.

⋮

step n-1 :-

Compare A[0] and A[1] and arrange them so that $A[0] \leq A[1]$. After n-1 steps, the list will be sorted in non-decreasing order.

Note :-

Each of the above steps is called a pass. Thus the bubble sort algorithm requires (n-1) passes, where 'n' is the number of input items.

Ex :- Suppose the list A is as follows :

32	51	27	85	66	23	13	57

Pass-1

(i) Compare A[0] and A[1], since $32 \leq 51$, the list is not changed.

(ii) Compare A[1] and A[2], since $51 > 27$, interchange 51 and 27 as follows.

32, (27), (51), 85, 66, 23, 13, 57.

(iii) Compare A[2] and A[3] since $51 < 85$ not changed.

(iv) Compare A[3] and A[4] since $85 > 66$, interchange.

32, 27, 51, 66, 85, 23, 13, 57.

v) Compare A[4] & A[5] since 85 > 23. interchange 85 and 23 as:

32, 27, 51, 66, 23, 85, 13, 57.

vi) Compare A[5] and A[6] since 85 > 13, interchange 85 and 13 as

32, 27, 51, 66, 23, 13, 85, 57

vii) Compare A[6] and A[7], since 85 > 57, interchange

32, 27, 51, 66, 23, 13, 57, 85.

At the end of this 1st pass, the largest number, 85, has moved to the last position. However the rest of the numbers are not sorted, even though some of them have changed their positions.

```
i = 0, 1, ... n-1
j = 0, 1, 2, 3, 4, 5, 4, 4
outer loop
    i = 0, 1, 2, 3, 4, 5, 6, 7
inner loop j = 0 ..... 6
{
  if (A[j] > A[j+1])
  {
    temp = A[j];
    A[j] = A[j+1];
    A[j+1] = temp;
  }
}
where, i = no. of steps/pass
       j = no. of comparision
```

Pass-2

27, 32, 51, 66, 23, 13, 57, 85,

27, 32, 51, 66, 23, 13, 57, 85

27, 32, 51, 66, 23, 13, 57, 85

27, 32, 51, 23, 66, 13, 57, 85

27, 32, 51, 23, 13, 66, 57, 85

27, 32, 51, 23, 13, 57, 66, 85.

At the end of pass-2, the second largest number, 66, has moved its ways down to the next to last position.

Pass-3

27, 32, 51, 23, 13, 57, 66, 85.

27, 32, 23, 51, 13, 57, 66, 85

27, 32, 23, 13, 51, 57, 66, 85

Pass-4

27, 32, 23, 13, 51, 57, 66, 85 / 27, 23, 32, 13, 51, 57, 66, 85.

27, 23, 13, 32, 51, 57, 66, 85.

Pass-5 23, 27, 13, 32, 51, 57, 66, 85.

Pass-6 23, 13, 27, 32, 51, 57, 66, 85.

13, 23, 27, 32, 51, 57, 66, 85.

Pass-7

13, 23, 27, 32, 51, 57, 66, 85.

Note:- Since the list has 8 elements; it is sorted after the seventh pass.

```
/* Bubble Sort */
#include <stdio.h>
  void main().
  {
  int i, j, A[10], temp, n;
 printf ("enter the no. of elements: \n");
 scanf ("%d", &n);
 printf ("enter the elements: \n");
  for (i=0; i<n; i++)
 scanf ("%d", &A[i]);
 /* Sorting Starts */
 for (i=0; i<n; i++)
 for (j=0; j<n-i; j++)
   {
   if (A[j] > A[j+1])
     {
     temp = A[j];
     A[j] = A[j+1];
     A[j+1] = temp;
     {
   }
 printf ("The array after sorting is: \n");
   for (i=0; i<n; i++)
   printf ("%d", A[i]);
  }
```

Complexity Of Bubble Sort Algorithm :-

There are (n-1) comparisions during the 1st pass which places the largest element in the last position. There are (n-2) comparisions in the 2nd pass which places the 2nd largest element in the next to last position and soon. Then,

$$T(n) = (n-1) + (n-2) + \dots + 2 + 1$$

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

Selection Sort :-

This is known as push-down sort. The selection sort consists entirely of a selection phase in which the largest of the remaining elements, Large, is repeatedly placed in its proper position.

Ex :-

	0	1	2	3	4	5	6
A	5	3	8	10	2	18	1

i = 0, 1, 2, 3, 4, 5

j = i+1, ... 6

Step-1

A 5, 3, 8, 10, 2, 18, 1

3, 5, 8, 10, 2, 18, 1

2, 5, 8, 10, 3, 18, 1

1, 5, 8, 10, 3, 18, 2

Step-2

i = 1

j = 2, ... 6

1	5	8	10	3	18	2

1 3 8 10 5 18 2

1 2 8 10 5 18 3

Step-3

i = 2

j = 3, ... 6

1 2 8 10 5 18 3

1 2 5 10 8 18 3

1 2 3 10 8 18 5

Step-4

i = 3

j = 4, 5, 6

1 2 3 10 8 18 5

1 2 3 8 10 18 5

1 2 3 5 10 18 8

Step-5

i = 4

j = 5, 6

1 2 3 5 10 18 8

1 2 3 5 8 18 10

Step-6

i = 5

j = 6

1 2 3 5 8 18 10

1	2	3	5	8	10	18

Complexity Of Selection Sort Algorithm :-

$$T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

```
/* Selection Sort */
/* To sort an integer array */
#include <stdio.h>
void main()
{
 int i, j, n, A[20], temp;
 clrscr();
 printf("How many numbers to be sorted \n");
 scanf("%d", &n);
 printf("Now type the numbers \n");
 for(i=0; i<n; i++)
 scanf("%d", &A[i]);
 for(i=0; i<n-1; i++)
   for(j=i+1; j<n; j++)
     {
     if(A[i] > A[j])
       {
       temp = A[i];
       A[i] = A[j];
       A[j] = temp;
       }
     }
 printf("The numbers after sorting are :");
   for(i=0; i<n; i++)
   printf("%d \n", A[i]);
 getch();
}
```

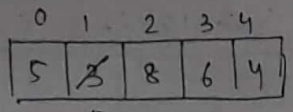
Insertion Sort :-

It is an efficient algorithm for sorting a small no. of elements. It sorts a set of values by inserting values into an existing sorted file.

```
/* Insertion sort */
#include<stdio.h>
void main()
{
int n,i,j,key,a[20];
printf("enter how many no.s to be sorted\n");
scanf("%d",&n);
printf("Enter the no.s\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(j=1;j<n;j++)
 {
 key=a[j];
  i=j-1;
 while((i>-1)&&(a[i]>key))
  {
  a[i+1]=a[i];
    i=i-1;
  }
 a[i+1]=key;
 }
printf("The sorted list is\n");
 for(i=0;i<n;i++)
 printf("%d",a[i]);
 }
```

$T(n) = O(n^2)$

Ex :-

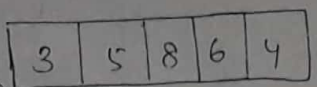


j = 1

Key = 3.

i = 0

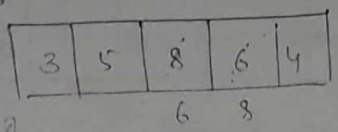
~~i~~ = 0 - 1 = -1



j = 2

Key = 8

i = 1

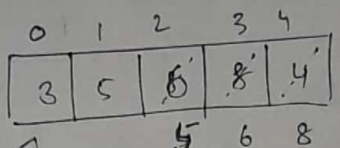


j = 3

Key = 6

i = 2

i = 2 - 1 = 1

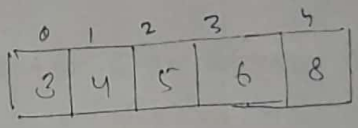


j = 4

Key = 4

i = 3

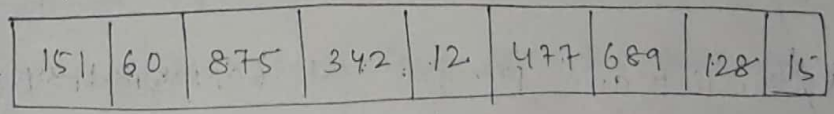
i = 3 - 1 = 2 - 1 = 1 - 1 = 0



Radix Sort :- (Bucket Sort)

To sort decimal numbers, we need ten buckets, since the base or radix is ten. These buckets are numbered as 0, 1, ..., 9.

Ex :-

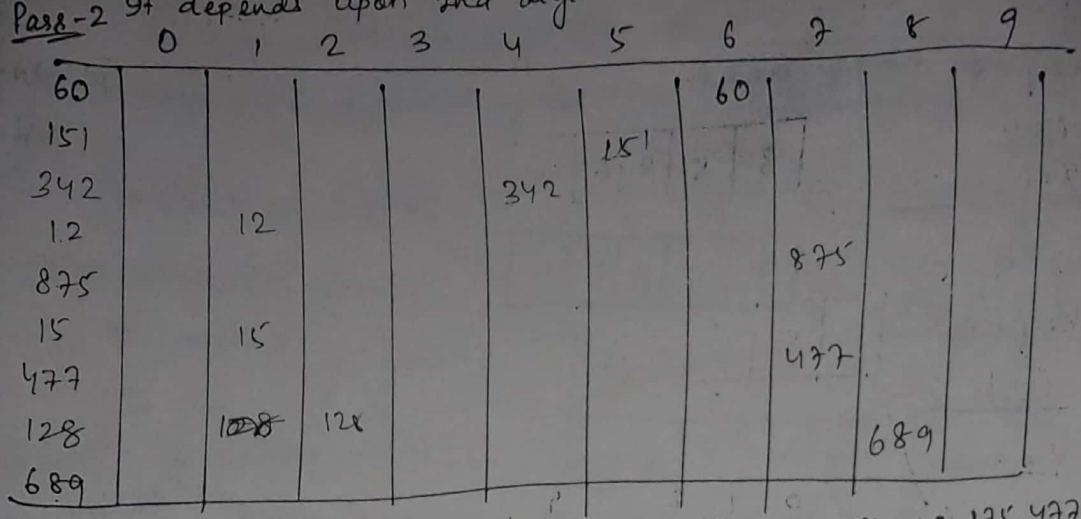


Pass-1 ~~9~~ depends upon the 1st digit.

	0	1	2	3	4	5	6	7	8	9
151		151								
60	60									
875						875				
342			342							
12			12							
477								477		
689										689
128									128	
15						15				

After pass-1 the numbers are - 60, 151, 342, 12, 875, 15, 477, 128, 689

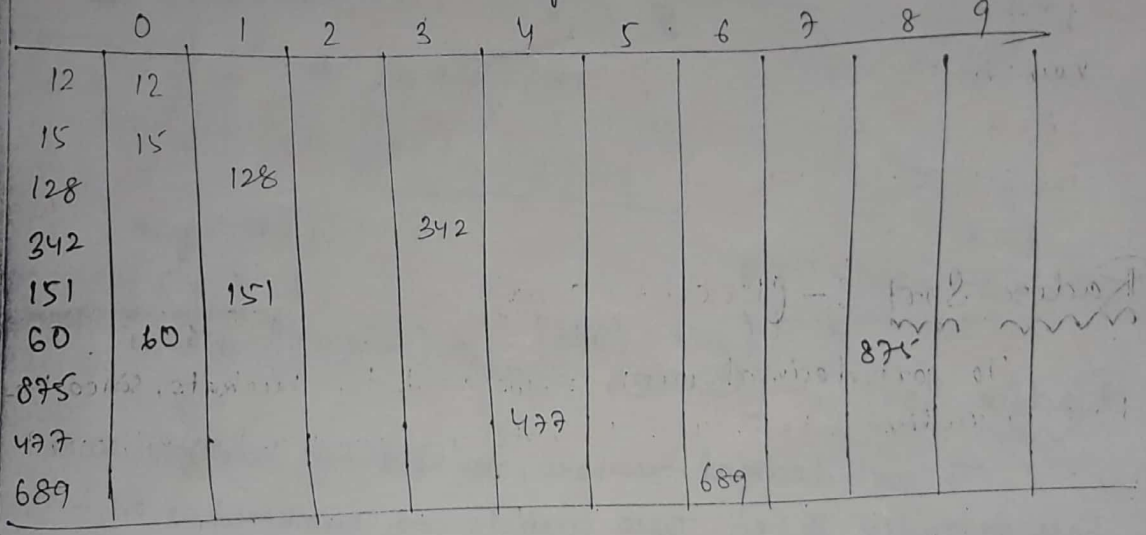
Pass-2 It depends upon 2nd digit.



The output after pass-2 is 12, 15, 128, 342, 151, 60, 875, 477, 689.

Pass-3

It depends upon 3rd digit



The output / the list after pass 3 - 12, 15, 60, 128, 151, 342, 477, 689, 875

Algorithm :-

Radix (a, n)

1. Set large = largest element in the array
2. Set num = total no. of digits in the largest element.
3. Set digit = num
4. Set pass = 1
5. Repeat steps 6 to 12 while pass <= num.
6. Initialise buckets.
7. Set i = 0
8. Repeat steps 9 to 11 while i <= n-1
9. Set k = Pass-1 position of number a[i].

[0th position of number 123 is 3]

10. Put the number a[i] into bucket k.

11. Set i = i + 1

[End of step 8 loop]

12. Set pass = pass + 1

[End of step 5 loop]

13. Write all the numbers from the bucket in order.

14. Exit.

$T(n) = O(n^2)$ → Worst case $T(n) = O(n \log n)$ → Best case.

Heap

1) Max Heap

2) Min Heap.

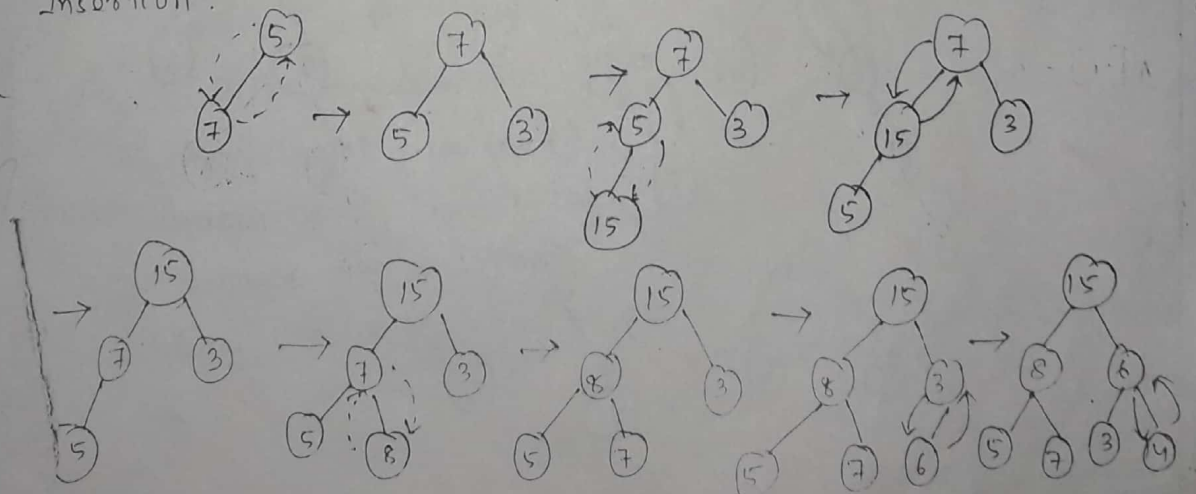
The heap data structures is an array object that can be viewed as an almost complete binary tree with n elements. Then H is called a heap (or max heap) if each node N of H has the following property: The value at N is greater than or equal to the value at each of the children of N.

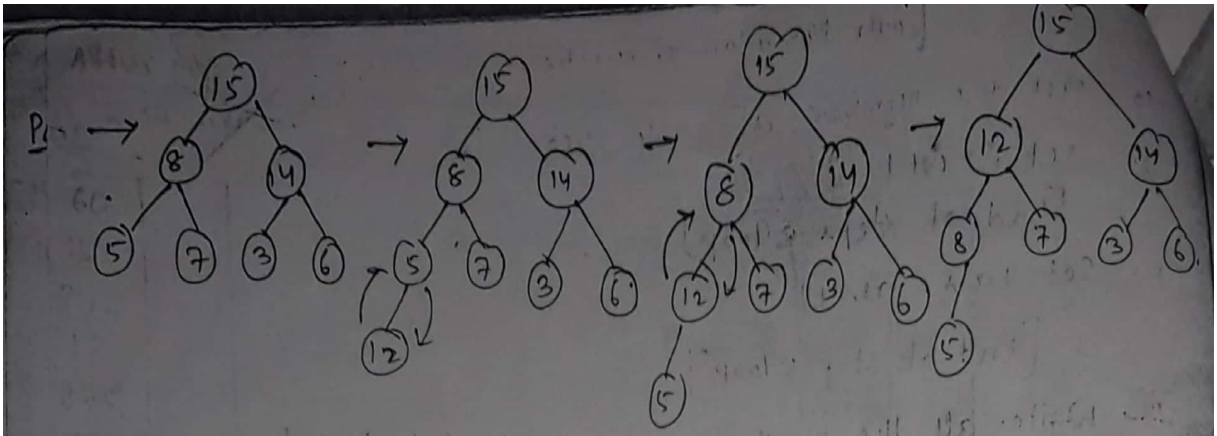
Similarly, H is called a min heap if each node N of H has the following property: The value at N is less than or equal to the value at each of the children of N.

Ex:- Using Heap sort technique sort the following.

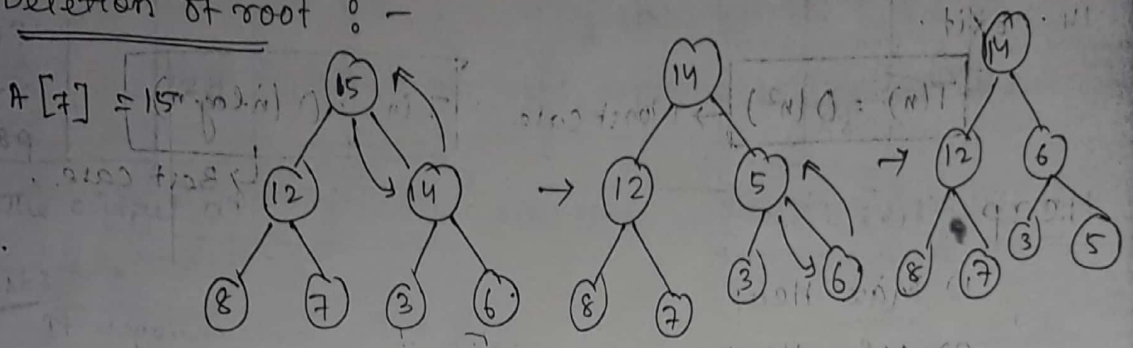
5 7 3 15 8 6 14 12

Insertion:

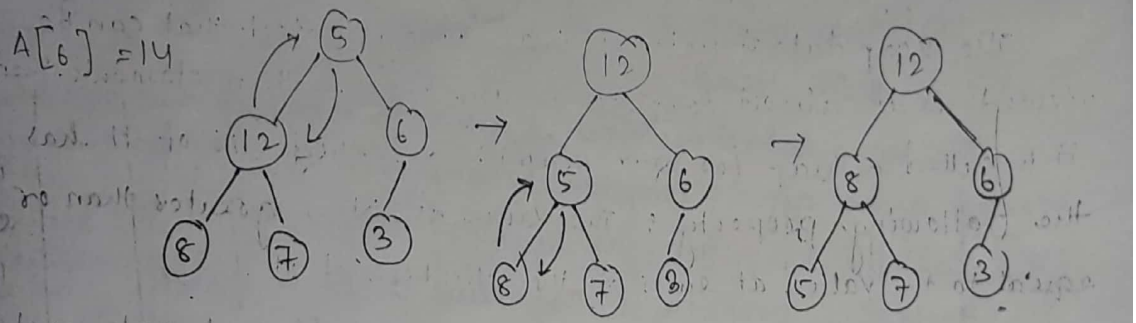




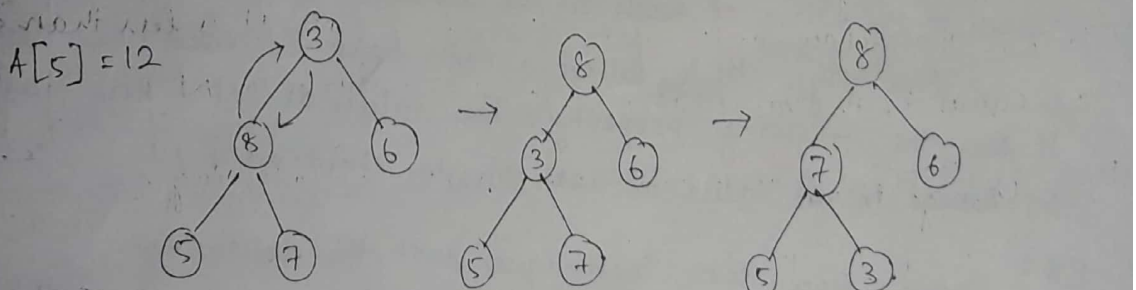
Deletion of root :-



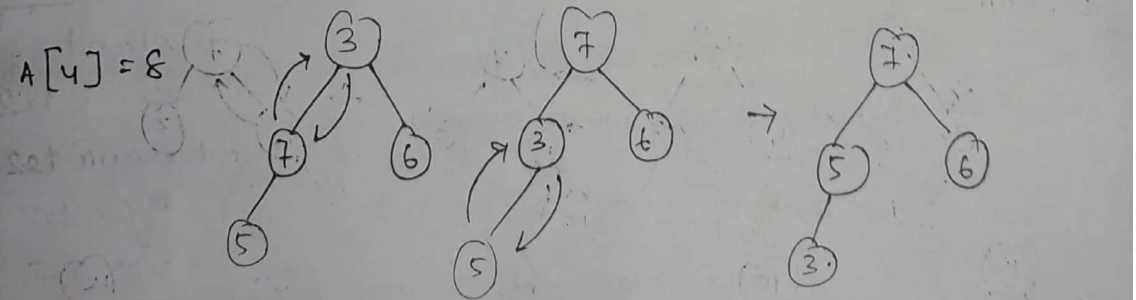
A[7] = 15



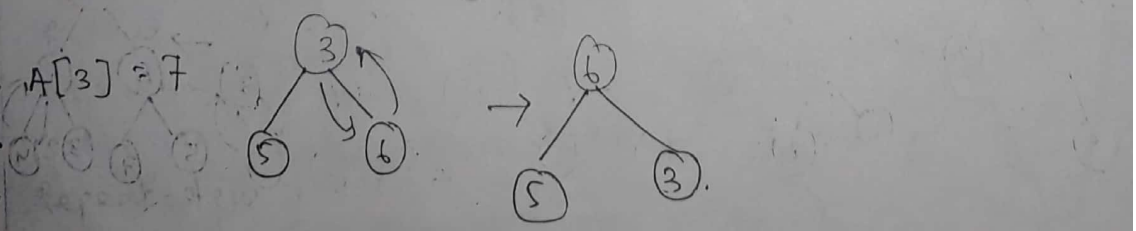
A[6] = 14



A[5] = 12

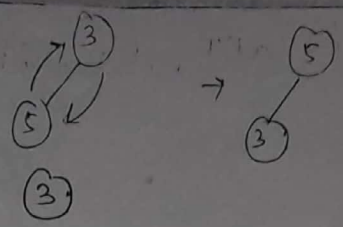


A[4] = 8



A[3] = 7

A[2] = 6 :



A[1] = 5

A[0] = 3

empt

After sorting we get 3, 5, 6, 7, 8, 12, 14, 15.

Merge Sort :-

Given a sequence of n elements a[1], ..., a[n], the general idea is to imagine them split into two sets a[1], ... a[⌈n/2⌉] and a[⌊n/2⌋+1], ... a[n].

Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n-elements.

Algorithm :-

→ Algorithm Merge sort (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.
    Merge sort (low, mid);
    Merge sort (mid+1, high);
   // combine the solutions.
    Merge (low, mid, high);
   }
}
```

```
void Merge sort (int a[], int p, int r)
  {
  int q;
  if (p<r)
   {
    q=(p+r)/2;
  Merge sort (a,p,q);
  Merge sort (a,q+1,r);
  Merge (a,p,q,r);
   }
  }
  void merge (int a[], intp, inta, int r)
   {
   int b[20], l1, r1, i;
   l1 = P;
   r1 = q+1;
   i = P;
   while ((l1<=q) && (r1<=r))
    {
    if (a[l1] <a[r1])
      {
      b[i] = a[l1];
      l1 = l1+1;
      i = i+1;
      }
     else
      {
       b[i] = a[r1];
       r1 = r1+1;
       i = i+1;
      }
    }
```

```
while ( l1 <= q)
  {
  b[i] = a[l1];
  l1 = l1+1;
  i = i+1;
  }
while ( r1 <= r)
  {
  b[i] = a[r1];
  r1 = r1+1;
  i = i+1;
  }
for (i=p; i<=r; i++)
  a[i] = b[i];
  }
```

Ex:-

a[1:10] = [310, 285, 179, 652, 351, 423, 861, 254, 450, 520]

split a[ ] into two subarrays each of size 5 (a[1:5], a[6,10])

a[1:5] → a[1:3] & a[4,5]

a[1:3] → a[1:2] & a[3:3]

The two values in a[1:2] are split a final time into one element subarrays and now the merging begins.

The array is viewed as

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

285, 310|179, 652, 351|423, 861, 254, 450, 520

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

Next elements a[4] and a[5] are merged.

179, 285, 310, 351, 652, 423, 861, 254, 450, 520.

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

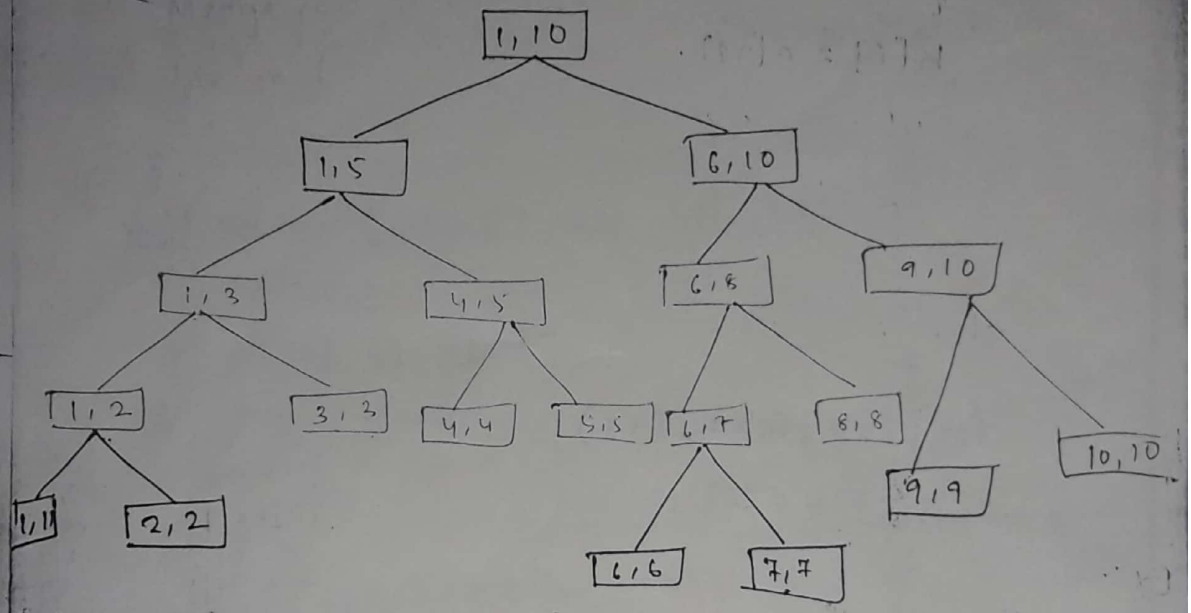
(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)

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

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

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

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



Quick Sort :-

It is an sorting algorithm for sorting the numbers. Here, input is array of numbers.

Input = Array of numbers $\{n_1, n_2, n_3 \cdots n_n\}$

Output = $n_1 \leq n_2 \leq n_3 \leq \cdots \leq n_n$

or, $n_1 \geq n_2 \geq n_3 \cdots \geq n_n$

Description :-

1) Divide : The array $A[p \ldots r]$ is partitioned into two non-empty subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ such that each element of $A[p \ldots q]$ is less than or equal to each element of $A[q+1 \ldots r]$. The index 'q' is computed as part of this partitioning procedure.

Conquer : The two subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are sorted by recursive calls to quick sort.

Combine : Since the subarrays are sorted in place, the entire array $A[p \ldots r]$ is now sorted.

Algorithm :-

Quick sort (A, P, r)

1. if P < r.
2. Then q ← PARTITION (A, P, r)
3. Quicksort (A, P, q)
4. Quicksort (A, q+1, r)

* Partitioning the array

PARTITION (A, P, r)

1. x ← A[p]
2. i ← P-1
3. j ← r+1
4. while ~~true~~ (1)
5. do ~~repeat~~ {j ← j-1}
6. ~~while~~ ~~(A[j] < x)~~ while (A[j] ≥ x);
7. ~~Repeat~~ do {i ← i+1}
8. ~~until A[i] ≥ x~~ while (A[i] < x);
9. if i < j
10. then exchange A[i] ↔ A[j]
11. else return j.

EX :-

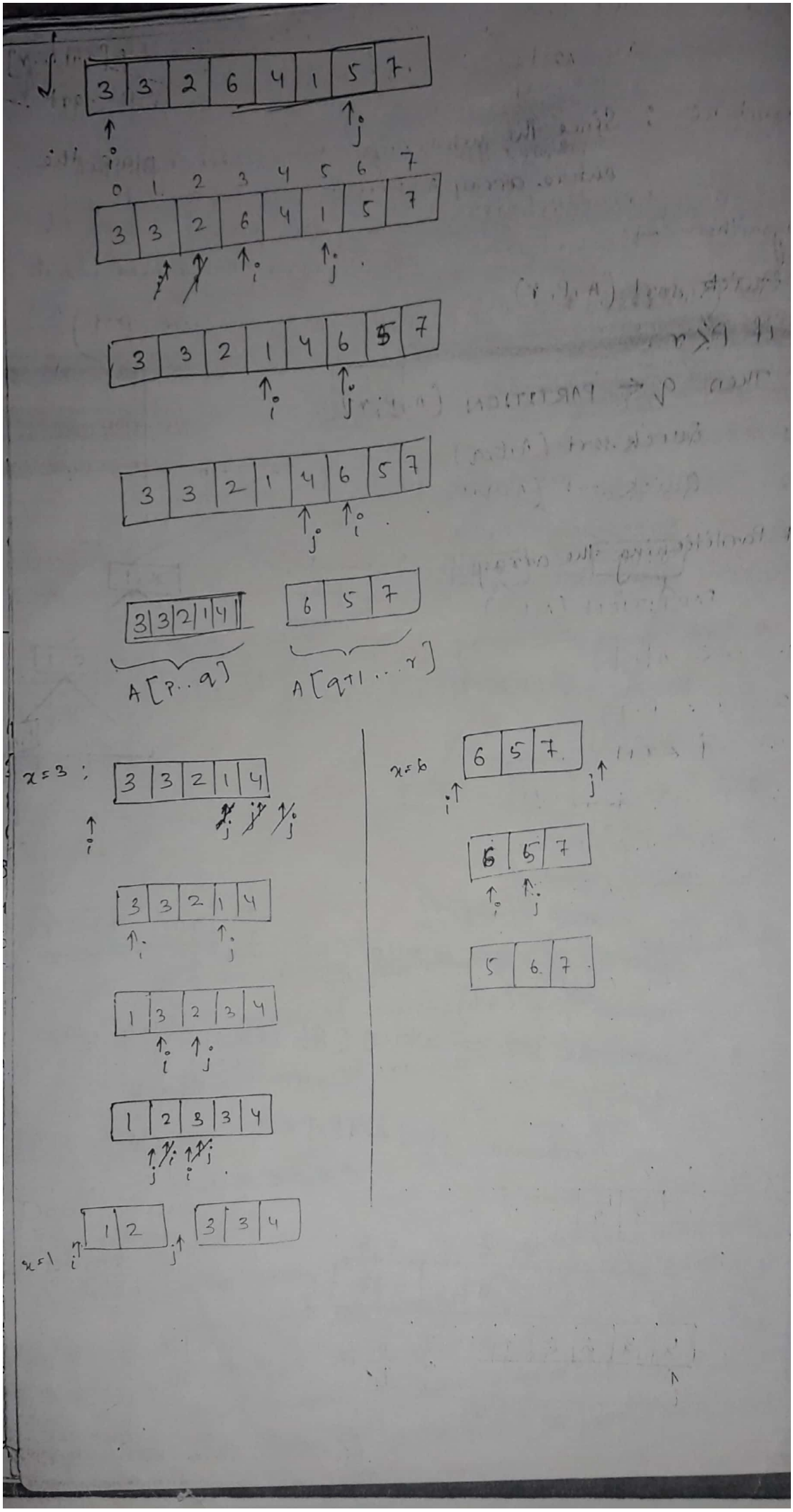
x = 5

1	2	3	4	5	6	7	8
5	3	2	6	4	1	3	7

i ↑ j ↑

5	3	2	6	4	1	3	7

i ↑ j ↑ j ↑



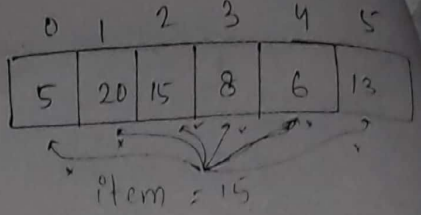
Program :-

```
Void Quick Sort (int a[ ], int P, int r)
  {
  int q;
  if (P<r)
    {
    q = partition (a,p,r);
   quick sort (a,p,q);
   quick sort( a,q+1,r);
     }
  }
int partition ( int a[ ], int P, int r)
   {
  int x, i, j, temp;
  x = a[P];
  i = P-1;
  j = r+1;
while (1)
  {
  do
    {
     j = j-1;
     } ~~while (a[j]>x);~~ while (a[j]>=x);

  do
     { i = i+1;
     } while (a[i]<x);

  if (i<j)
      {
       temp = a[i];
       a[i] = a[j];
       a[j] = temp;
      }
    else
      return (j);
    }
}
```

Program for linear Search :-




```
#include<stdio.h>
void main()
{
int n, i, item, a[20], loc=0;
printf("enter the number of elements \n");
scanf("%d",&n);
printf("enter the elements of array \n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("enter the element to be searched \n");
scanf("%d",&item);
for(i=0;i<n;i++)
{
if(item==a[i])
{
loc=i+1;
printf("element %d is present in position %d", item, loc)
}
}
printf("element %d is not present in the array", item);
getch;
}
```

Program for binary search :-

```
#include <stdio.h>
void main()
{
int i, n, item, a[10];
int beg, end, mid;
printf("enter the no. of elements \n");
scanf("%d",&n);
printf("enter the elements of array \n");
for(i=0;i<n;i++)
```

```
scanf("%d", &a[i]);
printf("enter the elements to be searched \n");
scanf("%d", &item);
beg = 0;
end = n-1;
mid = (beg + end)/2;
while ((beg < end) && (a[mid] != item))
{
if (item < a[mid])
end = mid - 1;
else
beg = mid + 1;
mid = (beg + end)/2;
}
if (item == a[mid])
printf("the element %d is present in position %d",
                                    item, mid);
else
printf("element %d is not present in the array", item);
```

1", item, loc);

, item);

Binary Search :- let a_i, $1 \le i \le n$ be a list of elements that are sorted in non-decreasing order

Ex :-

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

a[1:14]

for x = 151

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	found

for x = -14

low	high	mid
1	14	7
1	6	3
1	2	1
2	2	Not found

for x = 9

low	high	mid
1	14	7
1	6	3
4	6	5
5	5	found