1. [] Linear Search (Pseudo Code)

```
int lin-search (int* ar, int n, int key)
{
    for (i=0  to n-1)
    { if (arr[i] = key)
          return i;
    }
  return -1;
}
```

2) Iterative insertion sort

```
void insert-sort (int ar[], int n)
 int i, temp, j;
 for  i=1 to n
     temp = arr[i]
      j = j-1;
     while (j>=0  AND ar[j] > temp)
         ar[j+1] = ar[j]
          j = j-1
        ar[j+1] = temp
```

Recursive insertion sort

```
void insert-sort (int ar[], int n)
 if (n <=1)
     return
 insert.sort (ar, n-1)
   last = ar[n-1]
   j = n-2
 while (j>=0 && ar[j] > last)
 ar[j+1] = ar[j]
    j--
 ar[j+1] = last
```

doesn't need to know anything about what values it will sort and the information is requested while the algorithm is running. sorting Because it

3] i] selection sort:
T.C. = Best Case = $O(n^2)$; Worst Case = $O(n^2)$
S.C. = $O(1)$

ii) Insertion Sort:
T.C. = Best Case = $O(n)$; worst case $O(n^2)$
S.C. = $O(1)$

iii) Merge sort:
T.C. = Best case = $O(n \log n)$; worst case = $O(n \log n)$
S.C. = $O(n)$

iv) Quick sort:
T.C. = Best Case = $O(n \log n)$; worst case = $O(n^2)$
S.C. = $O(n)$

v) Heap Sort
T.C. = Best case = $O(n \log n)$; worst case = $O(n \log n)$
S.C. = $O(1)$

vi) Bubble sort
T.C. = Best case = $O(n^2)$; worst case = $O(n^2)$
S.C. = $O(1)$

4]

Sorting	Inplace	Stable	Online
Selection	✓		
Insertion	✓	✓	✓
Merge		✓	
Quick	✓		
Heap	✓		
Bubble	✓	✓	

5] Iterative Binary Search

```
int binsearch(int ar[], int l, int r, int x)
{
  while(l <= r){
    int m = (l+r)/2;
    if(ar[m] == x)
      return m;
    if(ar[m] < x)
      l = m+1;
    else
      r = m-1;
  }
  return -1;
}
```

T.C.

Best case :- O(1)

Avg case :- $O(\log_2 n)$

worst case :- O(log n)

Recursive Binary search

```
int bin_search(int ar[], int l, int r, int x)
{ if(r >= l){
    int mid = (l+r)/2;
    if(ar[mid] == x)
      return mid;
    else if(ar[mid] > x)
      return bin_search(ar, l, mid-1, x);
    else
      return bin_search(ar, mid+1, r, x);
  }
  return -1;
}
```

T.C.

Best case = O(1)

Avg. case = O(log n)

worst case = O(log n)

6) Recurrence Relation for binaoy recursive search

$T(n) = T(n/2) + 1$

Quick sort is the fastest general purpose sort. In most practical situations, quick sort is the method of choice. If stability is important and space is available, merge sort might be best.

How far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0. but if array is sorted in reverse order, the inversion count is max.

For following array arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

```
#include <bits/stdtt.h>
using namespace std;
int merge sort (int ar[], int temp [] , int left, int right);
int merge ( int ar[], int temp[ ], int left, int mid, int right);

int merge sort (int ar[], int array-size){
    int temp[array - size];
    return merge sort ( ar, temp, 0, array-size-1);
    }
int merge sort (int ar[], int temp[], int left, int right)
{ int mid, int count=0;
   if ( right > left){
   mid = (right + left) /2;
   inv-count = merge sort (ar, temp, left, mid);
   inv-count = -merge sort (ar, temp, mid+1, right);
   inv-count = merge (ar, temp, mid+1, right);
   }
   return inv-count;
}
int merge (int ar[], int temp[], int left, int mid, int right
{ int i, j, k;
   int inv-count =0;
     i = left;
     j = mid;
     k = left)
```

```
while (i <= mid-1) && (j <= right))
{ if (ar[i] < ar[j])
    temp [k++] = ar[i++];
  else
  {
    temp [k++] = ar[j++];
    inv_count = inv_count + (mid-i);
  }
}
while (i <= mid-1)
  temp[k++] = ar[i++];
while (j <= right)
  temp[k++] = ar[j++];
  ar[i] = temp[i];
  return inv_count;
}
int main()
{ int ar[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}
  int n = sizeof(arr) / sizeof(ar[0]);
  int ar = mergesort (ar, n);
  cout << "No. of inversion are" << cans;
  return 0;
}
```

6] → The worst case time complexity of quick sort is $O(n^2)$ the worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happen when input array is sorted on reverse sorted & either first or last element is picked as pivot.

The best case of quick sort is when we will select pivot as a mean element.

relation of:

a) merge sort → $T(n) = 2T(n/2) + n$

b) Quick sort → $T(n) = 2T(n/2) + n$

merge sort is more efficient and works faster than quick sort in case of larger array size or data sets.

worst case complexity for quick sort is $O(n^2)$ whereas as $O(n \log n)$ for merge sort.

(2)→ stable selection sort:-

```
using namespace;
void stab_selsort(int a[], int n){
  for(int i=0; i<n-1; i++){
     int min=i;
     for(int j=i+1; j<n; j++)
       { if(a[min]>a[j])
           min=j;
       }
     int key=a[min];
     while(min>i)
     {
       a[min]=a[min-1];
       min--;
     }
     a[i]=key;
  }
}
int main(){
  int a[]={4,5,3,2,8,1};
  int n=sizeof(a)/sizeof(a[0]);
  Stab_selsort(a,n);
  for(int i=0;i<n;i++)
     cout<<a[i]<<" ";
     cout<<endl;
     return 0;
}
```

13] :
The easiest way to do this is to use external sorting we divide our source file into temporary files of size equal to the size of the RAM & first sort in files

External sorting :- If the input data is such that it cannot adjust in the memory entirely at once, it needs to be stored in a hard disk, floppy disk or any other storage device This is external sorting.

Internal Sorting :- If the input data is such that it can be in the main memory at once, it is called internal sorting.