Page No
Date

Tutorial - 3

Sol 1 -

```
int linear( int *as, int n, int key)
    for i >=0 to n-1
        if as[i] = key
            return i
    return -1.
```

Sol 2 - Iterative insertion sort -

```
void insertion( int a[], int n)
  int i, temp, j;
  for i ← 1 to n
      temp ← a[i]
        j ← i-1
      while( j>=0 AND a[j] >temp)
          a[j+1] ← a[j]
            j ← j-1
          a[j+1] ← temp
```

insertion

Recursive sort -

```
void insertion (int a[], int n)
    if(n<=1)
      return
  insertion (a, n-1)
  last = a[n-1]
    j = n-2
  While ( j>=0 && a[j] > last)
      a[j+1] = a[j]
        j--
  as[ j+1] = last
```

Insertion sort is called online sorting because it does not need to know anything about values it will sort and information is required while algorithm

is running-

Sol 3- 1) Selection Sort -
T.C = $O(n^2)$ (Best), $O(n^2)$ (Worst)
S.C = $O(1)$

2) Insertion Sort -
T.C = $O(n^2)$ Best, $O(n^2)$ (Worst)
S.C = $O(1)$

3) Merge Sort -
T.C = $O(n \log n)$ (Best), $O(n \log n)$ (Worst)
S.C = $O(n)$

4) Quick Sort -
T.C = $O(n \log n)$ (Best), $O(n^2)$ (Worst)
S.C = $O(n)$

5) Heap Sort -
T.C = $O(n \log n)$, $O(n \log n)$ (Worst)
S.C = $O(1)$

6) Bubble Sort -
T.C = $O(n^2)$ (Best), $O(n^2)$ Worst)
S.C = $O(1)$

Sol 4-

Sorting	Inplace	stable	online
Selection	✓		
Insertion	✓	✓	✓
Merge		✓	
quick	✓		
heap	✓		
Bubble	✓	✓	

Sol 5- • iterative binary search -

```
int binary( int a[], int l, int r, int x)
{ while (l<=r)
  { int m = (l+r)/2;
```

```
if (a[m] == x)
    return m;
if (a[m] < x)
    l = m+1;
else
    r = m-1;
}
return -1
}
```

T.C
Best = O(1)
Average = O(log n)
Worst = O(log n)

- Recursion Binary Search -

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

T.C
Best = O(1)
Average = O(log n)
Worst = O(log n)

Sol 6 - Recurrence relation for binary recursive search

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

Sol 7 - $A[i] + A[j] = k$

Sol 8 - Quick sort is ~~fastest~~ general-purpose sort. Mostly quick sort is used. If stability is important and space is available, merge sort is best.

Page No.
Date

Sol 9- Inversion count for an array indicate, how far (or close) the array is from being sorted. If array is already sorted, then the inversion count is 0, but if array is in reverse order, the inversion count is maximum.

a[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

```
#include <bits/stdc++.h>
using namespace std;
int _mergesort (int a[], int temp[], int l, int r)
{
 int mid, c=0;
 if (r>l)
 { mid = (r+l)/2;
  int c+ = _mergesort (a, temp, l, mid);
     c+ = _mergesort (a, temp, mid+1, r);
     c+ = merge (a, temp, l, mid+1, r);
 }
 return c;
}
int mergesort (int a[], int array_size)
{
 int temp [array_size];
  return _mergesort (a, temp, 0, array_size-1);
}

int merge (int a[], int temp[], int l, int mid, int r)
{ int i, j, k;
  int c=0;
   i=left;
   j=mid;
    k=left;
```

```
while ((i<=mid-1) && (j<=r))
 { if (a[i]<=a[j])
    temp[k++] = a[i++];
   else
    {
    temp[k++] = a[j++];
     c += (mid-i);
     }
  }
While (i<mid-1)
 { temp[k++] = a[i++];
  While (j<=right)
    temp[k++] = a[j++];
for(i=l , i<=r ; i++)
    a[i] = temp[i]
    return c;
  }
int main ()
 { int a[] = {7,21,31,8,10,1,20,6,4,5}
    int n = 10;
    int ans = mergesort(a,n);
    Cout<<"Inversion "<<ans;
    return 0;
 }
```

Sol 10- The worst case time complexity of quick sort is $O(n^2)$. The worst case occurs when the picked pivot is an extreme (smallest/largest) element. This happen when input array is sorted or reverse sorted and 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.

Page No.
Date

Sol 11- Recurrence 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 & works faster than quick sort in case of larger array size or dataset.
- Worst case complexity for quick sort is $O(n^2)$ whereas $O(n \log n)$ for merge sort.

Sol 12. Stable selection sort -

```
using namespace std;
void stableselection(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[]= {4,5,3,2,4,1}
 int n = 6;
 stableselection (a,n)
 for (int i=0; i<n; i++)
   cout<<a[i]<<" ";
 return 0;
}
```

Page No.
Date

Sol 13 - The easiest way to do this is to use external sorting. We divide our source file into temporary file of size equal to size of RAM and first sort those files.

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

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