Date.
Page No.

Tutorial - 3

1. Int LinearSearch(int Arr[], int n, int key)
```
for(int i=0; i<n; i++)
{
    if(Arr[i] == key)
        return i;
}
    return -1;
}
```

2. Iterative insertion sort
```
Void Insertionsort(int Arr[], int n)
{  int i, j, t=0;
   for(i=1; i<n; i++){
      t = Arr[i];
      j = i-1;
      while(j>=0 && t < Arr[j])
{  Arr[j+1] = Arr[j];
      j--;
   }
   Arr[j+1] = t;
}
}
```

Recursive insertion Sort.
```
Void InsertionSort(int Arr[], int n)
{if (n<=1)
    return;
 InsertionSort(Arr, n-1);
 int last = Arr[n-1];
   j = n-2;
   while(j>=0 && Arr[j]>last){
      Arr[j+1] = Arr[j];
      j--; }  Arr[j+1] = last; }
```

Date:
Page No.

Insertion sort is called online sorting because it does not need to know anything about what value it will sort and the information is requested while the algorithm is running.

3(i) Bubble Sort:-
Time Complexity:- Best Case = $O(n^2)$
Worst Case = $O(n^2)$
Space Complexity = $O(1)$

(ii) Selection Sort:
Time Complexity:- Best Case = $O(n^2)$
Worst Case = $O(n^2)$
Space Complexity: $O(1)$

(iii) Merge Sort:-
Time Complexity:- Best Case = $O(n \log n)$
Worst Case = $O(n \log n)$
Space Complexity = $O(n)$

(iv) Insertion Sort:
Time Complexity:- Best Case = $O(n)$
Worst Case = $O(n^2)$
Space Complexity = $O(1)$

(v) Quick Sort:-
Time Complexity:- Best Case = $O(n \log n)$
Worst Case = $O(n^2)$
Space Complexity = $O(n)$

5	1	3	2	6
0	1	2	3	4

(m) ① key ← 3

Date
Page No.

(vi) Heap Sort:-
Time complexity: Best case = O(nlogn)
Worst case = O(nlogn)
Space complexity = O(1)

(4)

Sorting Algo	inplace	Stable	Online
Selection	✓		
Insertion	✓	✓	✓
Merge		✓	
Quick	✓		
Heap	✓		
Bubble	✓	✓	

(5). Iterative Binary Search:-

```
int BinarySearch(int Arr[], int l, int r, int key)
{ while(l <= r)
(m)→{ int m = (l+r)/2;
  if(Arr[m] == key)
      return m;
  if(Arr[m] < key)   // Right the search
      l = m+1;
  else
      r = m-1;
}
return -1;   // Not found case.
}
```

Note: Time complexity: ① Best case = O(1)
② Avg. Case = O(logn)
③ Worst Case = O(logn)

Recursive Binary Search :-

```
int BinarySearch (int Arr[], int l, int r, int key)
{ if ( r >= l)
{   int m = (l + r) / 2;
    if ( Arr[m] == key)
        return m;
    else if ( Arr[m] > key)
    return Binarysearch ( Arr, l, mid-1, key );
    else
    return Binarysearch ( Arr, mid+1, r, key );
}
return -1;
}
```

T.C :- Best Case = O(1)
Worst Case = O(log n)
Avg. Case = O(log n)

Linear Search T.C :-
Best Case = O(1)
Avg. Case = O(n)
Worst Case = O(n)

16) Recurrence relation for binary recursive search :-

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

7. $A[i] \neq A[j] = K$

Date
Page No.

(8). Quick Sort is the fastest general purpose sort. In most practical situations, quicksort is the method of choice. If stability is important & space is available, merge sort might be best.

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

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

```
#include <bits/stdc++.h>
using namespace std;
int merge_sort(int Arr[], int temp[], int left, int rig
int merge(int Arr[], int temp[], int left, int mid, int right
int merge_Sort(int Arr[], int Arr_size)
{ int temp[Arrsize];
  return merge_Sort[Arr, temp, 0, Arrsize-1);
} int merge_sort(int Arr[], int temp[], int left, int righ
{ int mid, invCount = 0;
  if (right > left){
    mid = left + (right - left)/2;
    invCount += merge_Sort(Arr, temp, left, mid);
    invCount += merge_Sort(Arr, temp, mid+1, right);
  invCount += merge(Arr, temp, left, mid+1, right);
}
  return invcount;
}
```

Page No.

```
int merge(int Arr[], int temp[], int left, int mid,
          int right)
{ int i, j, k, invCount = 0;
  i = left;
  j = mid;
  k = left;
while( (i <= mid-1) && (j <= right) ){
if( Arr[i] <= Arr[j] )
  temp[k++] = Arr[i++];
else{
  temp[k++] = Arr[j++];
invCount = invCount + (mid - i); } }
while (i <= mid-1)
 temp[k++] = Arr[i++];
while (j <= right)
temp[k++] = Arr[j++];
for(i = left; j <= right; i++)
 Arr[i] = temp[i];
return invCount;
}
int main() {
int Arr[] = {7, 21, 57, 8, 10, 1, 20, 6, 4, 5};
int n = sizeof(Arr)/sizeof(Arr[0]);
int ans = mergeSort(Arr, n);
Cout << "no of inversion arr" << ans;
return 0;
}
```

Date
Page No.

10. The worst case time complexity of quick sort is $O(n^2)$. The worst case occur when the picked pivot is always an extreme (smallest or largest) element. This happen when input array is sorted or reverse sorted and either first or last element is picked as pivat.
The best case of quick sort is when we will select pivot as a mean element.

11. Recurrence relation of:
(a) Merge Sort: $T(n) \Rightarrow 2T(n/2) + n$
(b) Quick Sort: $T(n) \Rightarrow 2T(n/2) + n$
Merge sort is more efficient & works faster than quicksort 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.

(12) Stable Selection Sort →

```
Void StableSelectionSort(int Arr[], int n)
{ for (int i=0; i<n-1; i++) {
    int min = i;
  for (int j=i+1; j<n; j++) {
  if (Arr[min] > Arr[j])
      min = j;
  }
  int key = Arr[min];
  while (min > i) {
     Arr[min] = Arr[min-1];
     min--;
  } Arr[i] = key; } }
```

```
int main()
{ int Arr[] = {4, 5, 3, 2, 4, 1};
  int n = sizeof(Arr)/ sizeof(Arr[0]);
  StablesselectionSort(Arr, n);
  for (int i=0; i<n; i++)
      cout << Arr[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 these files.

External Sorting:- If the input data is such that is cannot adjusted in the memory entirely at once it needs to be sorted in a hard disk, floppy disk or any 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.