TUTORIAL-3 DAA.

Date

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;
  }
}.
```

<u>Recursive</u>

```
void InsertionSort (int arr[], int n) {
    if (n <= 1)
        return;
  insertion sort (arr, n-1);
    last = arr[n-1];
    j = n-2;
while (j >= 0 && arr [j] > last) {
```

Date

```
    arr [j+1] = arr [j];
        j--;
    }
  arr [j+1] = last;
}.
```

Insertion sort is also called online sort because it does not need to know anything about what values 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 Comp. = $O(1)$.

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 - $O(n \log n)$
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 - $O(n)$

vi) Heap Sort
Time Complexity Best Case - $O(n \log n)$
Worst Case - $O(n \log n)$
Space Complexity - $O(1)$.

4	Sorting	Inplace	Stable	Online
1	Selection	✓		
2	Insertion	✓		✓
3	Merge		✓	
4	Quick	✓	✓	
5	Heap	✓		
6	Bubble	✓	✓	

5. Iterative Binary Search.

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

Date

Time Complexity Best Case - O(1). Avg Case - O(logn)
Worst Case = O(n log n).

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;
}
```

Time Complexity Best Case = O(1) Avg. Case - O(log n)
Worst Case - O(log n).

Linear Search
Time Complexity Best Case : O(1) Avg. Case O(n).
Worst Case : O(n).

6. Recurrence Relation for Binary Search

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

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

Date

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

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

```
# include <bits / stdc ++. h>
using namespace std;
 int mergeSort (int arr [], int temp[], int left
                  int right);
int mergeSort (int arr[], int temp[], int left, int mid
                  int right);
int mergesort (int arr [], int array. size) {
       int temp [array-size];
return merge_sort (arr, temp, 0, array-size -1);
}
int mergeSort (int arr [], int temp[], int left, int
          right) {
  int mid, inv_count = 0;
     if (right > left) {
mid = left + [right - left]/2;
inv_count + = mergeSort (arr, temp, left, mid);
inv_count + = merge Sort (arr, temp, mid+1, right);
inv_count + = merge Sort (arr, temp, left, mid +1, right)
}
return inv-count;
}.
int merge (int arr [], int temp [], int left, int mid
             int right) {
 int i, j, k, inv_count = 0;
```

Date

```
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++];
      inv_count = inv_count + (mid-i);
  }}.
while (i<=mid-1)
   temp[k++] = arr[i++];
   while (j<= right)
      temp[k++]= arr[j++];
 for (i= left; j<= right; j++)
      arr[i]= temp[i];
  return inv_count;
}
int main() {
   int arr [] = {7, 21, 51, 8, 10, 1, 20, 6, 7, 5};
   int n = sizeof (arr) / sizeof (arr [0]);
   int ans = mergesort (arr, n);
   cout <<"no. of inversions are "<< ans;
     return 0;
}.
```

10. Worst 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 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.

Date

11. Reccurrence Relation

Merge Sort => $T(n) = 2T(n/2) + n$

Quick Sort => $T(n) = 2T(n/2) + n$.

→ Merge Sort is more effecient and work faster than quick Sort in case of large array size or datasets.

→ Worst case complexity for quick sort is $O(n^2)$ whereas $O(n \log n)$ for merge Sort.

12 Stable Selection Sort

```
void StableSelection (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] = a[min-1];
     min--;
 }
  arr [i] = Key;
}}
   int main () {
int arr [] = {4, 5, 3, 2, 4, 1};
int n = sizeof (arr) / sizeof (arr[0]);
StableSelection (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 and 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

- Internal Sorting - If the input data is such that it can adjusted in this main memory at once.