DATE : / 20
PAGE No

Tutorial - 3

① Pseudocode for linear search

→
```
for (i=0 to n)
{
    if (arr[j] == value)
        //element found.
}
```

②
```
Void insertion (int arr[], int n)  //recursive
{
    if (n<=1)
        return;
    insertion (arr, n-1)
    int nth = arr[n-1];
    int j = n-2;
    while (j>=0 && arr[j] > nth)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = nth;
}
for (i=1 to n)
{
    key ← A[i];
    j ← i-1;
    while (j>=0 and A[j] > key)
    {
        A[j+1] ← A[j]
        j ← j-1
    }
    A[j+1] ← key
}
```

DATE : / 20
PAGE No

Insertion sort is online sorting because it doesn't know the whole input, more input can be inserted with the insertion sorting is running

(8) Complexity

Name Best

(3) (1) Bubble sort :

Time complexity - Best case = $O(n^2)$
worst case = $O(n^2)$

space complexity - $O(1)$

(2) Selection sort -

Time complexity - Best case - $O(n^2)$
worst case - $O(n^2)$

space complexity - $O(1)$

(3) Merge sort -

Time complexity - Best case - $O(n \log n)$
Worst case - $O(n \log n)$

space complexity - $O(n)$

(4) Insertion sort -

Time complexity - Best case - $O(n)$
Worst case - $O(n^2)$

space complexity - $O(1)$

(5) Quick sort -

Time complexity - Best case - $O(n \log n)$
worst case - $O(n^2)$

DATE : / 20
PAGE No

⑥ Heap sort -

Time complexity - Best case - O(nlogn)
worst case - O(nlogn)
Space complexity = O(1)

④

Sorting	inplace	stable	online
Selection	✓		
Insertion	✓	✓	✓
Merge		✓	
Quick	✓		
heap	✓		
bubble	✓	✓	

⑤ Iterative Binary search.

```
int binarysearch(int arr[], int l, int r, 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;
}
```

Time complexity
Best Case - O(1)
Avg case - O(logn)
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;
}
```

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 recursive search

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

DATE : / 20
PAGE No

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

(9) Insertion count for an array indicates - how far (or close) the array is from being sorted. If the array is already sorted then the insertion 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 <iostream>
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 [arraysize];
    return mergesort (arr, temp, 0, array_size-1);
}
int mergesort (int arr[], int temp[], int left,
                int right)
{
        int mid, invcount = 0;
    if (right > left)
```

```
{
    mid = left + (right - left)/2;
    invcount += mergesort(arr, temp,
                          left, mid);
    invcount += mergesort(arr, temp, right,
                          mid+1);
    invcount += merge(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;
    i = left;
    j = mid;
    k = right;
    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; i++)
    arr[i]=temp[i];
return inv_count;
}

int main()
{
    int arr[] = {7, 21, 5, 8, 10, 1, 20, 6, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    int ans = mergesort(arr, n);
    cout << "no of inversion are" << ans;
    return 0;
}
```

(10) 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 happens 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.

(11) Recurrence relation of:
a) mergesort : $T(n) = 2T(n/2) + n$
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 datasets

→ 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]);
    stableselectionsort(arr, n);
    for(int i=0; i<n; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
}
```

DATE : / 20
PAGE No

(13) The easist way to do this is to use external sorting we divide our source file into temprory files of size equal to the size of the RAM & first sort these files

- External sorting : If input data is such that is cannot adjusted in the memory entirely at once it need to be stored 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