Tutorial Sheet-- 3.

Date: / /

1- Linear Search (pseudo code)

```
int lin_search (int * ar, int n, int key)
{ for (i>=0 to n-1)
    { if (ar[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 = ar[i]
        j = i-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
```

Date: / /

Why Insertion sort is called online sorting?
Because it doesn't need to know anything about what values it will sort & the information is requested while the algo. is running.

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 of $O(n^2)$
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	✓	✓	

Date: / /

5- Iterative Binary search

```
int bin_search (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 binary recursive search

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

7 -

8 - Which sort & why is used practically?

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

9- What is inversion count for an array?

How far (or close) the array is from being sorted. If the array 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/stdc++.h>
using namespace std;
int _mergeSort(int arr[], int temp[], int left, int right);
int merge(int arr[], int temp[], int left, int mid, int right);

int mergeSort(int arr[], int array_size) {
    int temp[array_size];
    return _mergeSort(arr, temp, 0, array_size-1);
}

int _mergeSort(int arr[], int temp[], int left, int right)
{   int mid, inv_count = 0;
    if (right > left) {
```

Date: / /

```
    mid = (right + left) / 2;
    inv_count += _mergeSort(ar, temp, left, mid);
    inv_count += _mergeSort(ar, temp, mid+1, right);
    inv_count += merge(ar, temp, left, 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++];
for (i = left; i <= right; i++)
        ar[i] = temp[i];
    return inv_count;
}
int main()
{   int ar[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5};
    int n = sizeof(ar)/sizeof(ar[0]);
    int ans = mergeSort(ar, n);
```

Date: / /

```
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 & 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) 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 data sets.

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

12 - 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];
```

Date: / /

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

13-

The easier 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 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 adjust in the main memory at once, it is called internal sorting.