Date. _____
Page No. _____

Tutorial - 3

Q1. Write a linear search pseudo code to search an element in a sorted array with minimum comparison

-> Int linear search (int A[], int n, int t {
if (abs (A[0] - t) > abs (A[n-1] - t))
for (i = n-2 to 0; i--)
if (A[i] == t) { return i; }
else
for (i = 0 to n-1; i++)
if (A[i] == t)
return i;
}

Q2. Iterative Insertion Sort

```
void insertion (intA[], int n)
  {
    for (i = 1 to n)
      { t = A[i];
        j = i
    while (j >= 0 && t < A[j])
      {
        A[j+1] = A[j];
          j--;
        }
        A[j+1] = t;
      }
  }
```

Date.
Page No.

Recursive Insertion Sort

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

Insertion sort is also called online sorting algorithm it will if the elements to be sorted are provided one at a time with the understanding that the algorithm must keep the sequence sorted as more elements are added in other sorting algorithms like bubble sort, insertion sort, heap sort etc are considered external sorting technique as they need the data to be sorted in advance

Q3. Complexity of all sorting algorithms

Sorting	Best Case	Worst Case
Bubble sort	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$
Count sort	$O(n)$	$O(n+k)$
quick sort	$O(n \log n)$	$O(n^2)$
merge sort	$O(n \log n)$	$O(n \log n)$
heap sort	$O(n \log n)$	$O(n \log n)$

Q4.

Sort	Inplace	Stable	Online
Bubble	✓	✓	X
Selection	✓	X	X
Insertion	✓	✓	✓
Count	X	✓	X
quick	✓	X	X
merge	X	✓	X
Heap	✓	X	X

Q6 Recursive / Iterative pseudo code for binary search

Iterative

```
int binary search (int arr[], int x)
  {
    int l=0, r = arr.length -1;
    while (l<=r)
    {
```

Date.
Page No.

```
int m= l+ (r-l)/2;
if (arr[m] == x)
    return m;
if (arr[m] < x)
    l = m+1;
else
    r = m-1;
}
return -1;
}
```

Recursive

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

Linear search;

Date.
Page No.

Iterative:- Time complexity = O(n)
space complexity = O(1)

Recursive: Time complexity = O(n)
space complexity = O(n)

Binary search:
Iterative: Time Complexity: O(n log n)
space complexity: O(log n)

Q6.

T(n)
↓
T(n/2)
↓
T(n/4)
↓
⋮
↓
T(n/2^R)

Recurrence relation = T(n/2) + (0/1)

Q7.

```
int n;
int A[n];
int key;
int i=0, j=n-1;
while (i<j)
{
if ((A[i] + A[j]) = key)
    break;
  else if ((A[i] + A[j]) = key)
```

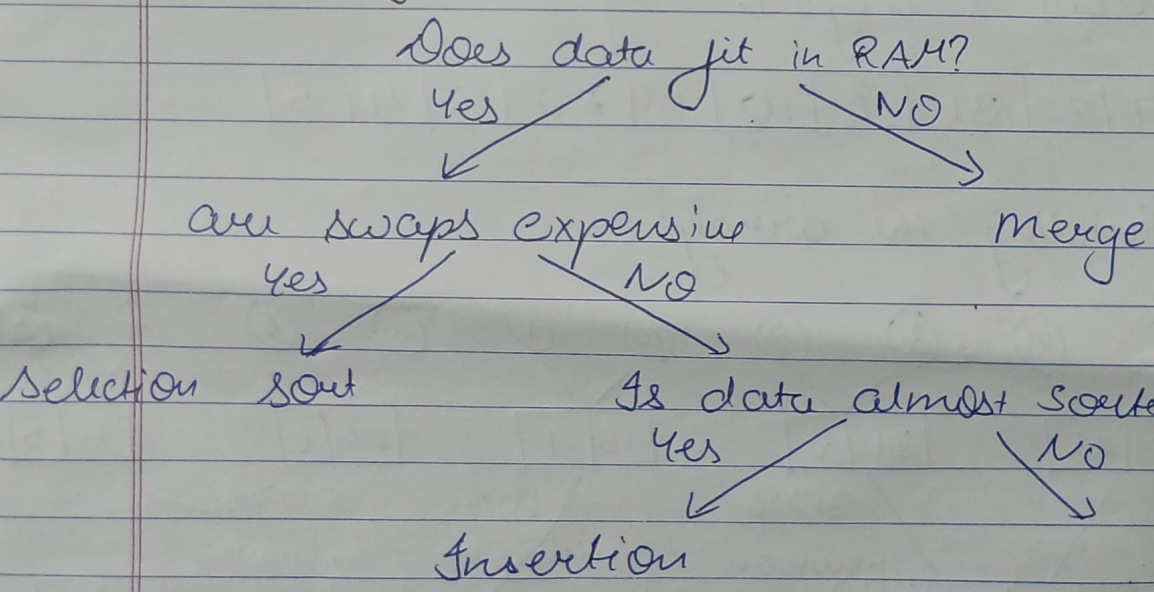
Date
Page No.

```
    else
        j--;
    }
    cout << i << " " << j;
```

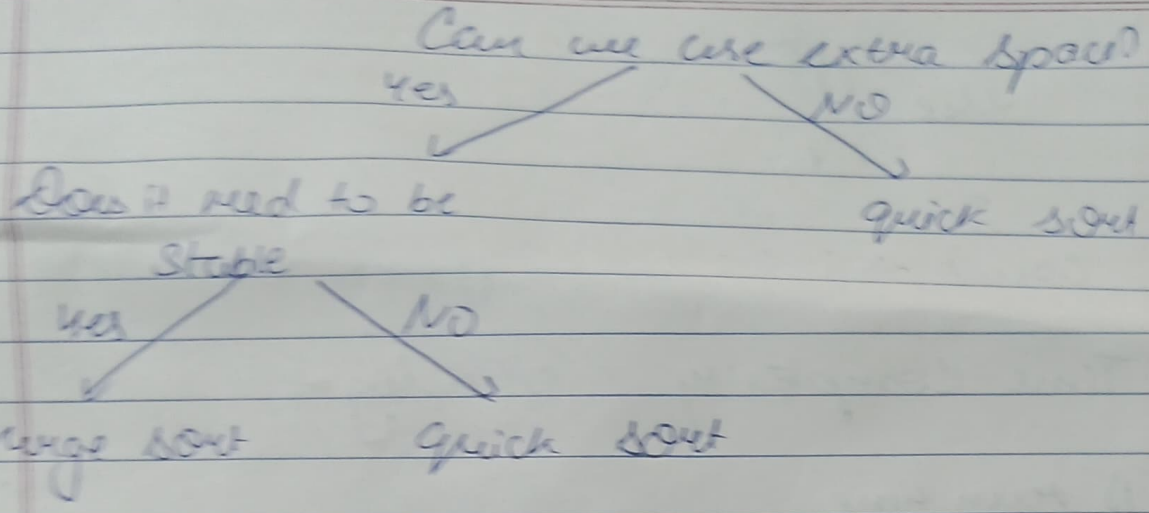
Time complexity = $O(n \log n)$

Q8. i) run time
ii) space
iii) stable
iv) No. of swaps
v) will the data fit in the RAM

→ There is no best sorting algorithms. It depends on the situation or the type of array provided



Date. _____
Page No. _____



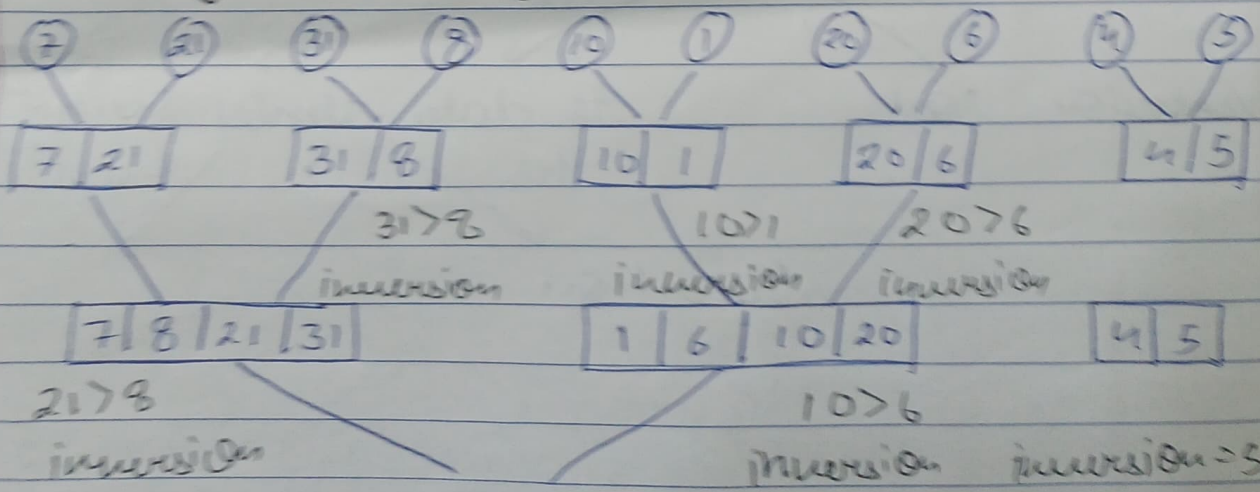
Q. Inversion in an array indicates how far the array is from being sorted. If the array is already sorted, the inversion count is 0, but if the array is sorted in reverse order, then the inversion count is maximum.

Condition for inversion

$a[i] > a[j]$ & $i < j$

7	21	31	8	10	1	20	6	4	5

Dividing the array:



Date.
Page No.

| 1 | 6 | 7 | 8 | 10 | 20 | 21 | 31 | 4 | 5 |

7>1, 7>6, 8>1, 8>6, 21>10, 21>20, 31>6, 31>6, 31>20, 21>1, 21>6

Total inversion in this step = 12

| 1 | 4 | 5 | 6 | 7 | 8 | 10 | 20 | 21 | 31 |

inv count = 0

6>4, 6>5, 7>4, 7>5, 8>4, 8>5, 10>4, 10>5, 20>4, 20>5, 21>4, 21>5, 31>4, 31>5

→ Total inversion in this step = 14

inversion count = 31

Q1c Best Case:
Time Complexity = O(n log n)

The best case occurs when the position process always picks the middle elements as pivot

Worst Case:
Time complexity : O(n²)
when the array is sorted in ascending and descending order

Date.
Page No.

Q1. Best Case:

Merge Sort: $2T(n/2) + n$
Quick Sort: $2T(n/2) + n$

Worst Case:

Merge Sort: $2T(n/2) + n$
Quick Sort: $T(n-1) + n$

Similarities: They both work on the concept of divide & conquer algorithms. Both have best case complexity of $O(n \log n)$

Differences:

Merge Sort	Quick Sort
i) The array is divided into just 2 halves	i) The array is divided in any ratio
ii) Worst case complexity is $O(n \log n)$	ii) Worst case complexity $O(n^2)$
iii) It requires extra space i.e NOT inplace	iii) It does not require extra space i.e inplace
iv) It is external sorting algorithm & it is stable	iv) It is internal sorting algorithm & not stable
v) Works consitantly on	v) Works fast on small

Date.
Page No.

any size of data set sel

Q12. Selection sort is not stable by default but you can write a version of stable selection sort

```
void selection (int A[], int n)
{
for (int i=0, i<n-1; i++)
{
   int num = i;
for (int j = i+1; j<n; j++)
{
   if (A[i] > A[j])
      min = j;
int key = A[min];
while (min > i)
{
   A[min] = A[min -1]
      min --;
}
   A[i] = key;
}
}
```

Q13 void bubble sort (int A[], int n)

```
{
   int i, j;
   int j = 0;
for (i=0; i<n; i++)
{
for (j=0; j<n-1; j++)
```

Date. _____
Page No. _____

```
    }
    if (A[j] > A[j+1])
    {
    swap (A[j], A[j+1])
    j = 1;
    }
}
    if (j == 0)
    break;
}
```

Q14. When the data set is large enough to fit inside RAM, we ought to use merge sort because it uses the divide & conquer approach in which it keeps dividing the array into smaller parts until it can no longer be splitted, it then merge the array divided in n parts. Therefore, at the time only a part of array is taken on RAM

External Sorting

It is used to sort maximum amount of data. It is required when the data doesn't fit inside the RAM & instead they must reside in the slower external memory

Date.
Page No.

During sorting, chunks of small data that can fit in main memory read & written out to a temporary file

During merging, the sorted subfiles are combined into a single large file

Internal sorting

It is a type of sorting which is used when the entire collection of data is small enough to reside within RAM. Then there is no need of external memory for program execution. It is used when input is small

Eg- Insertion sort, quick sort, heap sort etc.