Name - Abhiraj Gautam
Section - CST SPL - 2
Roll-No - 05

Tutorial - 3

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

→
```
Int linear_search(int A[], int n, int t
{ if(abs(A[0]-t) > abs(A[n-1]-t))
    for(i=n-1 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(int A[], 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;
}
```

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;
}
```

Abhirai

PAGE 3
DATE

Insertion sort is also called online sorting algorithm because it will work 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	$O(n \log n)$	$O(n \log n)$

Abhiraj

PAGE
DATE

Q4.

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

Q5. Recursive / Iterative pseudo code for binary search.

Iterative:

```
int binarySearch(int arr[], int x)
{
    int l=0, r=arr.length-1;
    While (l<=r)
    {
      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 binarySearch(int arr[], int l, int r,
int x)
{
  if (r >= l)
  {
    int mid = l + (r - l)/2;
    if (arr[mid] == x)
      return mid;
    else if (arr[mid] > x)
    return binarySearch(arr, l, mid-1, x
    else
    return binarySearch(arr, mid+1, r, x);
  }
  return (-1);
}
```

Linear search:

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: $O(1)$

Recursive: Time Complexity: $O(n \log n)$
Space Complexity: $O(\log n)$

Q6.

$T(n)$
↓
$T(n/2)$
↓
$T(n/4)$
↓
⋮
↓
$T(n/2^k)$

recurrance relation = $T(n/2) + (0/1)$.

Q7.

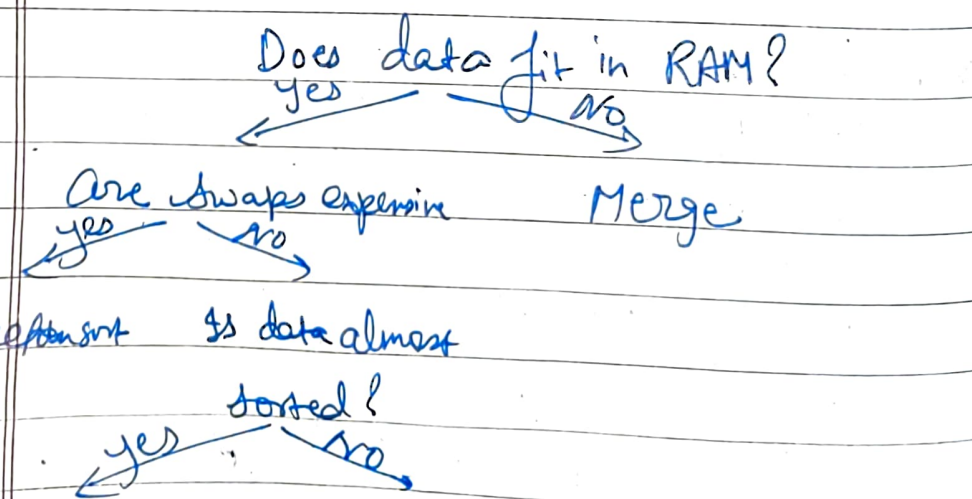
```
int n;
int A[n];
int key;
int i = 0, j = n-1;
while (i<j)
{
  if ((A[i] + A[j]) == key)
    break;
```

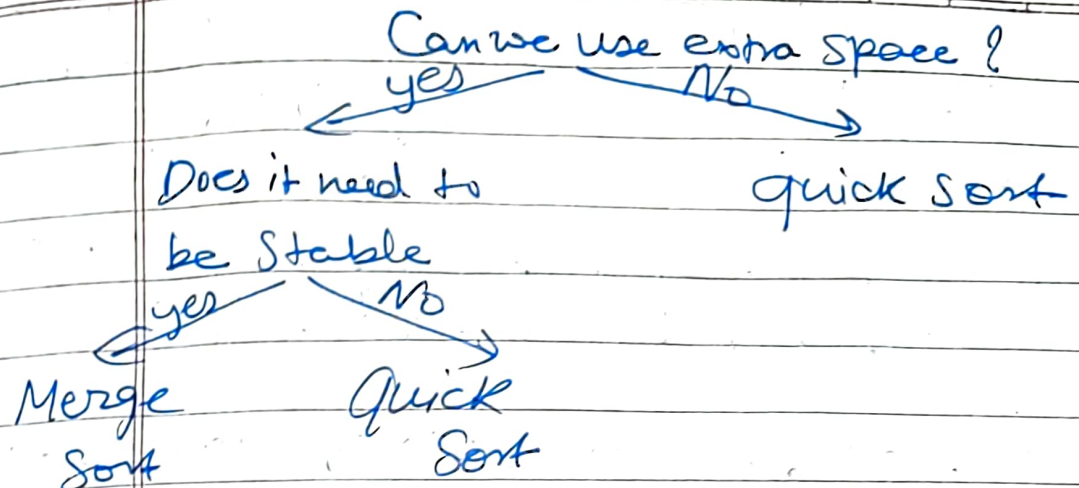
```
else if (A[i] + A[j] > Key)
        j--;
else
        i++;
}
Cout << i << " " << j;
```

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 algorithm. It depends on the situation or the type of array provided.





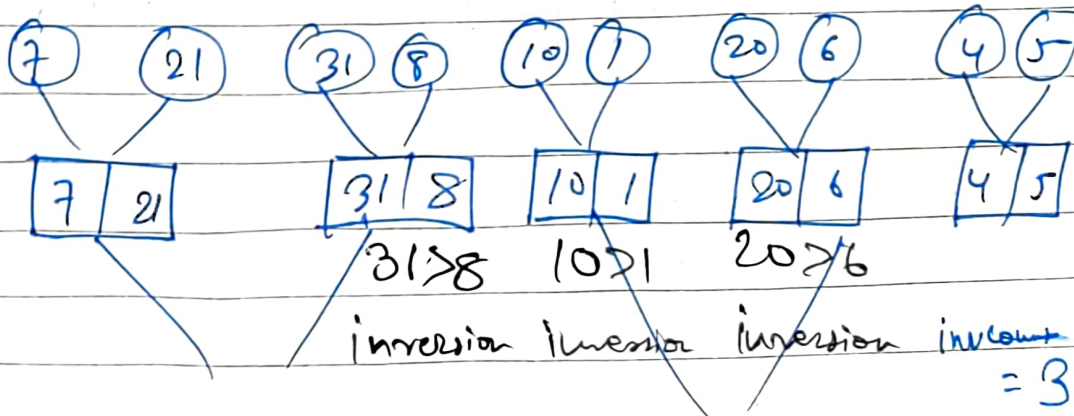
Q9. Inversion in an array indicates how far the array is from being sorted. if the array is already sorted, the inversion count is 0, but if the array is sorted in reverse order, then the inversion count is maximum.

Condition for inversion,

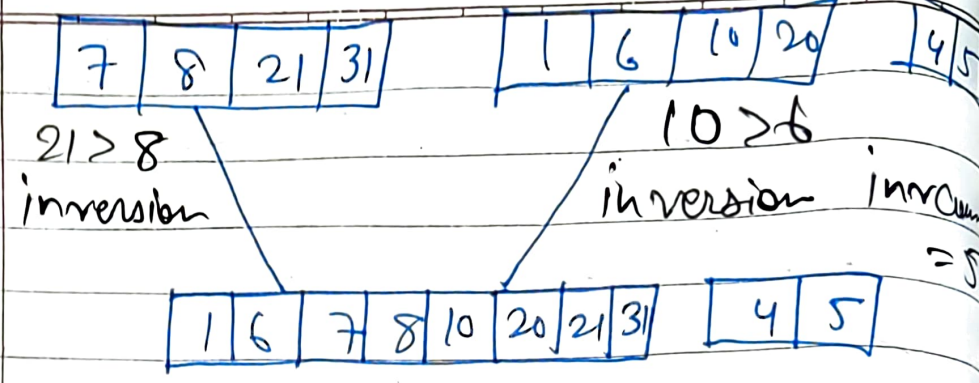
$a[i] > a[j]$ & $i < j$

7	21	31	8	10	1	20	6	4	5

Dividing the array!

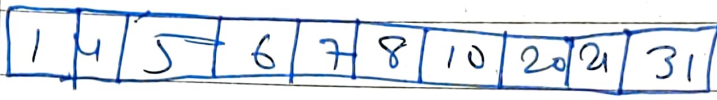


PAGE
DATE



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

Total inversion in this Step = 12



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.

Q10. Best Case:
Time Complexity = $O(n \log n)$

The best Case occurs when the partition process always picks the middle element as pivot.

Worst Case:
Time Complexity: $O(n^2)$
When the array is sorted in ascending or descending order.

Q11. Best Cases:

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 algorithm
Both have best case complixity of $O(n \log n)$

Differences:

	Merge sort	quick sort
(i)	The array is divided into just 2 half	(i) The array is divided in any ratio.
(ii)	worst case complexity is $O(n \log n)$	(ii) worst case complexity $O(n^2)$.

Abhiraj

PAGE
DATE

Merge Sort	Quick Sort
(iii) It requires extra space i.e NOT inplace	(iii) It doesnot requires extra space i.e inplace
(iv) It is external Sorting algorithm & it is Stable	(iv) It is internal Sorting algorithm & not Stable
(v) Works Consistently on any size of data set	(v) Works fast on small data set.

Q.2. 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[Min] > A[j])
      min = j;
int key = A[min];
while (min > i)
{
  A[min] = A[min-1]
     min--;
}
A[i] = key;
}
}
```

Q.3. void bubblesort(int A[], int n)

```
{
    int i, j;
    int flag = 0;
for(i=0; i<n; i++)
{
  for(j=0; j<n-1; j++)
  if(A[j] > [j+1])
  {
  swap(A[j], A[j+1])
  flag = 1;
}
}
```

```
if (j--==0)
{ break;
}
```

Q(B14.) 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 massive amount of data. It is required when the data doesn't fit inside the RAM & instead they must reside in the slower external memory.

During Sorting, Chunks of small data that can fit in main memory are read, sorted & written out to a temperory file.

During Merging, the Sorted subfiles are Combines Into a single large files.

Internal Sorting:

It is a type of Sorting which is used when the intire 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.

—x—