Tutorial - 3

1. Write linear search code pseudocode to search an element in a sorted array with minimum comparisions

Sol:-

```
Void search (int arr[], int n, int x)
{
  if (arr[n-1] == x)
    cout << "Found" << endl;
  int t = arr[n-1];
  arr[n-1] = x;
  for(int i=0; i<n; i++)
    {
      if (arr[i] == x)
        {
          arr[n-1] = t;
          if (i < n-1)
            cout << "Found" << endl;
          else
            cout << "Not found" << endl;
        }
    }
}
```

② Write pseudo code for iterative and recursive insertion sort. Insertion sort is called online sorting. why? what about other sorting algorithms that has been discussed in lecutres.

Sol

Iterative

```
Void_insort(int a[], int n)
{
  for (int i = 1; i < n; i++)
    int t = a[i];
    int j = i;
    while (j > 0 && a[j-1] > t)
```

```
    { a[j] = a[j-1];
      j--;
    }
    a[j] = t
  }
}
```

Recursive

```
void i_sort(int a[], int n)
{
  if (n<=1)
    return;
    i_sort(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 called online sorting because it does not need to know anything about what values it will sort and the information is requested while the algorithm is running. simply it can grab new values at every iteration.

Only insertion is sort online sorting among all sorts.

(3) complexity of all sorting Algorithms that has been discussed in lectures

sl	sorting type	Worst case (T) Time / (S) space	Avg case	Best case
1.	Bubble sort	$O(n^2)/O(1)$	$O(n^2)$	$O(n)$
2.	Insertion	$O(n^2)/O(1)$	$O(n^2)$	$O(n)$
3.	selection	$O(n^2)/O(1)$	$O(n^2)$	$O(n^2)$
4.	Quick	$O(n^2)/O(\log n)$	$O(n\log n)$	$O(n\log n)$
5.	Merge	$O(n\log n)/O(n)$	$O(n\log n)$	$O(n\log n)$
6.	Count	$O(k)/O(k)$	$O(N+k)$	$O(n)$
7.	Randomized Quick	$O(n^2)/O(\log n)$	$O(n\log n)$	$O(n\log n)$
8.	Heap	$O(n\log n)/O(n)$	$O(n\log n)$	$O(n\log n)$

(4) Divide all the sorting algorithms into inplace/stable / online

sl	sorting type	Inplace	stable	online
1.	Bubble	✓	✓	
2.	Insertion	✓	✗	✓
3.	selection	✓	✓	✗
4.	Quick	✓	✗	
5.	Merge	✗	✓	✗
6.	Count	✗	✓	
7.	Randomised Quick	✓	✗	
8.	Heap	✓	✓	

(5) write Recursive / iterative pseudocode for binary search. what is the time and space complexity of linear and Binary search.

Recursive

```
int binary search (int [] A, int low, int high), int x)
{
  if (low > high)
   { return -1;
   }
   int mid = (low + high) / 2;
    if (x == A[mid] {
       return mid;
    }
    else if ( x < A[mid])
     {
       return binary search ( A, low, mid -1 , x);
     }
     else {
        return binary search (A, mid+1 , high, x);
     }
  }
```

iterative

```
int binary search (int [] A , int x)
 {
  int low =0 , high = A.length -1;
      while (low <= high)
       {
          int mid = (low + high) / 2;
          if (x == A[mid])
           {
             return mid;
           }
```

```
else if (x < A[mid])
    { high = mid - 1;
    }
  else {
      low = mid + 1
    }
}
return -1;
```

} Binary search

Time complexity of Binary search recursive is O(log n)

Time complexity - iterative - O(log n)

space complexity - recursive - O(log n)

space complexity - iterative - O(1) / O(log n)
best Avg/worst

Linear search

Time complexity recursive - O(n)

Time complexity iterative - O(n)

space complexity recursive - O(nm)

space complexity iterative - O(1)

(6) write recurrence relation for binary recursive search

Recurrence relation is $T(n) = T(n/2) + 1$,

where t(n) is the required time for binary search in an array of size n.

(7) Find two indexes such that $A[i] + A[j] = k$ in minimum time complexity. int find(A[], n, k)

{ sort(A, n); for (i=0 to n-1) { n = binary search (A, 0, n-1, k-A[i]); if (n) return 1; } return -1; }

Time complexity O(n log n)

(8) which sorting is best for practical uses? Explain.

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

(9) what do you mean by no of inversions in an array? Count the number of inversions in Array arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5} using merge sort.

Sol:- Inversion in array & numbers indicate

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 the array is sorted in the reverse order the inversion count is maximum.

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

inversions - {(7,21) (7,31) (7,8) (7,10) (7,20)
(21,31), (8,10), (8,20), (10,20), (1,20), (1,6)
(1,4), (1,5) (4,5)}

Total No of inversions - 14

(10) In which cases Quick sort will give the best and worst case time complexity.

Sol The worst case time complexity of Quick sort is O(n²). The worst case occurs when the picked pivot is always 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 time complexity of Quick sort is $O(n \log n)$. The best case occurs when we select pivot as a mean element.

(11) write recurrence relation of merge sort & Quick sort in best and worst case? what are the similarities and differences between complexities of two algorithms and why?

Sol Recurrence relation of

Merge sort → $T(n) = 2T(n/2) + n$

Quick sort → $T(n) = 2T(n/2) + n$

merge sort is more efficient and works faster than quick sort in case of large array size or data sets.

worst case complexity for Quick sort is $O(n^2)$ where as $O(n \log n)$ for merge sort.

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

Sol stable selection sort

```
Void stable_sel (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[p] = key;
}
```

(13) Bubble sort scans whole array even when array is sorted. can you modify the bubble sort so that it doesn't scan the whole array once it stored

Sol:- Modified bubble sorting

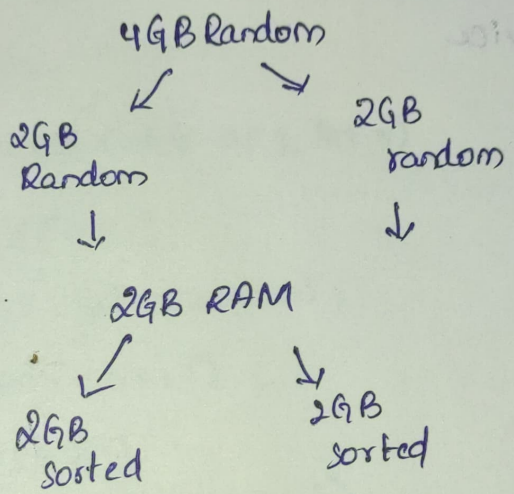
```
void bubble(int a[], int n)
{
  for(int i=0; i<n; i++)
  {
    int swaps=0;
    for(int j=0; j<n-1-i; j++)
    {
      if(a[j] > a[j+1])
      {
        int t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
        swaps++;
      }
    }
    if(swaps==0)
      break;
  }
}
```

(14) your computer has a RAM (physical memory) of 2GB and you are given of array of 4 GB for sorting which algorithm you are going to use for this purpose and why? Also explain the concept of external & internal sorting.

Sol The easiest way to do this is use external sorting ie divide our source file into temporary files of size equal to the size of the RAM and first sort these files

Assume 1GB = 1024 B,

1. Divide the source file into 2 small temporary files of each size 2 GB (equal to the size of RAM)
2. Sort these temporary files one by one using the ram individually (Any sorting algorithm : Quick / Merge)



Now we have sorted temporary files

1. pointers are initilized in each file.
2. A new file of size 4 GB (size of soure file) is created
3. First element is compared from each file with the pointer
4. smallest element is copied into the new 1GB file and pointer gets incremented in the file which pointed to this smallest element.
5. same process is followed till all pointers have traversed their respective files
6. when all the pointers have traversed we have

a new file which has 1GB of sorted integers

This is how any larger file can be sorted when there is a limitation on the size of RAM.

Internal sorting

If the input data is such that it can be adjusted in the main memory at once it is called internal sorting

External sorting - If the input data is such that it cannot be adjusted in the memory entirely at once it needs to be sorted in a hard disk or floppy disk or any storage device