## Set-5

3. For A[i][j]

B = Base address (A[0][0])

S = size of element

(i) Must skip i rows each containing c element

no. of elements = $i \times c$

(ii) At start of row i, we must be j more elements

no. of elements = $(i \times c) + j$

(iii) Address (A[i][j]) = $B + [(i \times c) + j] \times S$

4. Linear search - It checks every element in collection, one by one, until target is found or collection ends. It works on both sorted or unsorted array with time complexity is slow i.e. $O(n)$

Binary search - It uses divide & conquer algorithm but it requires data to be sorted

Example

(i) Set two pointers, low & high

(ii) Find middle mid = (high + low)/2

~~or mid = low + (hi~~

(iii) If target = middle, return middle

(iv) If target > mid,

set low = mid + 1

(v) If target < mid,
set high = mid - 1

(vi) Repeat process

Time complexity = O(log n)

5.

(i) Select pivot element

(ii) Array is rearranged so that all elements smaller than pivot are arranged to its left while others at its right

(iii) Algorithm then recursively applies same process to two sub-arrays.

Real life example - when sorting large list of exam scores for national test. If organisation needs to find 90th percentile cutoff (top 10%) quick sort is ideal

6. Algorithm

```
Function fibo(n)
    If n<=0 return 0
    If n==1 return 1
      a = 0
      b = 0
    For i = 2 to n
        current = a+b
          a = b
          b = current
      END For
        return b
```

8. reference to

a) Comparison with stack usage

(i) Recursion - It uses call stack implicitly. Each time function calls itself, new stack frame is pushed onto stack to store local variable. It consumes memory of O(n). If recursion is too deep, it can cause stack overflow error.

(ii) Iterative - It typically uses constant amount of stack space O(1) for its single stack frame. State is managed through local variable within that frame.

b) Recursive program for factorial

```
Factorial(n)
If n == 0 or n == 1 then
    return 1

EISE
Return n * factorial(n)
  END If
    END function
```

9.

MergeSort Function (Divide & conquer)

(i) Divide : if array has more than one element, find middle & split into two halves

(ii) Conquer : recursively call MergeSort on left half

(iii) Conquer : recursively call MergeSort on right half

(iv) Combine : Call Merge function to combine two sorted sub-arrays to one sorted array

Merge function (combine)

(i) Create two temporary arrays for each sorted half (L & R)

(ii) Use two pointers (i for L & j for R) & main pointer of original array

(iii) Compare L[i] & R[j]. Copy smaller element to A[k]

(iv) Increment pointer (i or j) from copied one & also increment k

(v) Repeat until one temporary array is empty

(vi) Copy remaining element from other temporary array back into A

Significance in Large Dataset Sorting -

(i) Guaranteed performance - It always runs in O(nlogn) time even in worst case.

(ii) External sorting - If dataset is too large to fit in RAM. It sorts "chunks" of file in RAM, saves them to disk then merges sorted files on disk

(iii) Stability - It is stable sort. If two items have same value, they will stay in their original relative order.