Set-3

1. Algorithm efficiency refers to how many computation resources an algorithm consumes in relation to its size of input. It is measure of its performance -

(i) Time efficiency - how fast algorithm runs
(ii) Space efficiency - how much memory it takes

2.

Time complexity	Space complexity
(i) Amount of time an algorithm takes to complete function of input size n	(i) Amount of memory an algorithm takes to complete function of input size n
(ii) Example - Linear search O(n) Time in worst case, it must check all element	(ii) Example - Linear search O(1) It only uses few extra variable. Memory doesn't grow in array's size.
(iii) Merge sort O(n log n) - Divide array log n time & performs n comparisons at each level	(iii) Merge sort O(n) - requires extra auxillary array of size n to merge sorted halves

3. Let 3D array is A[D1][D2][D3] where
D1 = No. of rows
D2 = No. of columns
D3 = No. of planes
B = Base address (A[0][0][0])
S = Size of one element (4 for int)

Find address of A[i][j][k]

(i) To find correct depth - $D_1 \times D_2$ elements

(ii) To get column j within slab
Elements to skip = $j \times D_1$

(iii) To get correct row within column
Element to skip = i

(iv) Total elements = $(k \times D_1 \times D_2) + (j \times D_1) + i$

(v) Final address
$A[i][j][k] = B + [(k \times D_1 \times D_2) + (j \times D_1) + i] \times S$

4. Binary search algorithm

```
BS(arr A, int target)
    low = 0
    high = length(A) - 1
while low <= high
    mid = (high + low) / 2
// 1. if target is at middle
    If A[mid] == target then
        Return mid;
// 2. if target is greater, search right half
    Else If A[mid] < target then
        low = mid + 1
// 3. if target is smaller, search left half
    else
        high = mid - 1
END while
Return -1
```

5. Bubble sort repeatedly steps through the lists, compares adjacent elements & swap them if they are in wrong order

Working - Algorithm uses two nested loops -

(i) Outer loop - This loop runs n-1. After each pass, next largest element is guaranteed to be in its final sorted position

(ii) Inner loop - Iterates from beginning of array up to unsorted portion. It compares A[j] to A[j+1] & swaps if A[j] > A[j+1]

Example- [3, 1, 2] → swap (3 & 1)
[1, 3, 2] → swap (3 & 2)
[1, 2, 3]

Example [3, 2, 1] → swap (3 & 2)
[2, 3, 1] → swap (3 & 1)
[2, 1, 3] → swap (2 & 1)
[1, 2, 3] → No swaping as a[2] < a[3]

6.
```
int factorial(int n){
    if (n == 0){
        return 1;}
    else{
        return n*factorial(n-1); } }
```

(i) Base case - If n = 0, stops recursing & returns 1

(ii) Recursive case - If n > 0, factorial is defined as n x (n-1)! where function calls itself with n-1 argument

8.

a) Recursion	Iteration
(i) It uses call stack. Each time function calls itself, a new stack frame is pushed to frame	(i) Uses loop. Memory required is typically just for loop control variables
(ii) Easy to write but can memory intensive. Space complexity is O(n)	(ii) More memory efficient with space complexity O(1)
(iii) If recursion is too deep, it consume all available stack memory resulting stack overflow error	(iii) Iteration avoid stack overflow due to heap memory which is much larger memory area

b)
```
int fibo (int n){
  if (n==0){
    return 0;}
  else if (n==1){
    return 1;}
  else {
    return fibo(n-1) + fibo(n-2);}
}
```

9. a) Divide

(i) Algorithm divides unsorted array into two halves.

(ii) It recursively call itself on each halves.

(iii) Halves broken into array of size 1. Each array containing one sorted element

(iv) Division takes log n time at each level

b) Merge

(i) Merge two sorted arrays into a larger sorted array

(ii) It is very efficient as it uses two pointers & compares elements at pointers

(iii) It is repeated until all elements get stored in temporary array

(iv) Merging process taking linear time, $O(n)$

(v) Overall efficiency - $O(n \log n)$

Real world applications

(i) External sorting

(ii) Stable sorting requirements

(iii) Parallel & concurrent programming

(iv) Language standard libraries