Set 1

1). Define asymptotic notation and explain its importance in analyzing algorithm efficiency.

Asymptotic notation is a set of formal languages used to describe the limiting behaviour of a function, particularly in the context of an algorithm's running time or space requirements as the input size grows.

Its importance :-

As it provides a way to classify algorithms based on their growth rates, allowing for a comparison of their performance without getting bogged down in the specific details of a particular machine or programming language. It focuses on how the growth rate of the algorithm's complexity as the input size increase. This helps in choosing the most efficient algorithm for a given problem. especially for large datasets where the performance differences become significant.

Three notations are :

1). Big O notation - describes upper bound of an algo's running time. Worst-case scenario.

2) Big Omega notation - lower bound, best-case

3) Big Theta notation - a tight bound on an algo's running time, average-case scenario.

2). In tail recursion, the recursive call is the very last operation performed in the function. After this recursive call returns, there are no more operations to be executed. This allows some compilers to optimize the function by reusing the current stack frame, which can prevent stack overflow errors that might occur with deep recursion.

```
public static int factTa(int n, int a) {
  if (n == 0) {
    return a; }
  else {
  return factTa(n-1, n * a); } }
```

In Head recursion, the recursion call is the first operation performed in the function. The function performs operations on the return value of the recursive call. This means that the function must wait for the recursive call to return before it can lead to a growing stack of function calls.

```
public static int factHe(int n) {
    if (n == 0) {
  return 1; }
    else {
  return n * factHe(n-1); }
  }
```

3). The Index formula for a 2-D array A[i][j] in row-major order is :-

Address (A[i][j]) = Base Address + W X (i x n + j)

row index column index size in bytes no. of colum

A[i][j] = B + W * ((I - LR) * N + (J - LC))

For column-Major Order :-

A[i][j] = B + W * (J - LC) * M + (I - LR))

4). Linear Search	Binary Search
Time Complexities :-	TC :-
Best Case O(1) - The target element is found at 1st pos".	Best Case O(1) - The target element is found at the middle pos".
Worst Case O(n) - Target is found at last pos", or not present at all in the list.	The target element is at extreme end or not present. O(log n)
Average case : O(~~log~~ n)	Average Case : O(log n).
It is simple to implement.	Slightly more complex due to sorting requirement and the divide-and-conquer logic.

5) Insertion sort :- It is a simple, in-place and stable sorting algorithm that builds a sorted array one element at a time. It is efficient for small datasets and those that are already partially sorted.

Algorithm :-

1) Start with 1st element (consider it to be sorted).
2) Iterate through, starting from 2nd arr[1] to the last element arr[n-1].
3) Store the current element (the one to be inserted) in a temporary variable, often called key.
4) Compare key with each element in sorted array moving backward from the current posn.
5) If sorted element is > key, shift it one posn to the right to make space.
6) Continue shifting elements until you find a posn where the sorted element is less or equal to key.
7) Insert key into final vacant posn.
8) Repeat the process for all elements until the entire array is sorted.

6) Sparse Matrices are represented by :-
1) Triplet Representation 2) Compressed Sparse Row
3) Compressed Sparse Column
4) Linked List Representation - A LL is used, where each node stores the row, column, and value of a non-zero element. Useful for dynamic matrices where elements are frequently inserted or deleted.

7). Reversing a single Linked List :-

It involves changing the direction of the next pointers of each node to point to the previous node instead of the next using three pointers - prev, curr, next_node.

Initialization - prev is initialised to NULL and its next pointer will point to NULL.
curr is initialized to the head of L.L.
next_node is initialized to NULL.
Iteration - Traverse the L.L using a while loop until curr becomes NULL. Store next node of curr in next_node to avoid losing the rest of the list. Reverse the next ptr. of curr to point to prev. Move prev one step forward Move curr one step forward.
curr = next_node.
Return - After the loop finishes, prev will point to new head of the reversed linked list. Return prev.

8). Tower of Hanoi :-

Base case : If (disk == 1)
Recursive step : ((n > 1) disks)
1. Move the top (n-1) disks from the S to the A, using destination D as a temp. This frees up the largest one on the S.
2. Move the largest one (n th) disk from the S to the D. This is a direct move as the

D is clear now for the largest disk

3. Recursively move (n-1) disks that are now on the A to D, using S as a temp. helper. This places n-1 disks on top of the largest disk on the D, completing transfer.

9.

Recursion	Iteration
Code elegance and simplicity.	Higher efficiency
Fewer lines of code	More verbose code
Higher memory usage	lower memory usage
Difficult to debug	Easier to debug.
High risk of stack overflow with deep or infinite recursion.	lower risk of overflow issues.
Slower due to the overhead of function calls and stack management	Faster, as it avoids function call overhead.