DSA practice Questions

Set - 1

1. Asymptotic notation is a tool used to describe limiting behaviour of function's runtime or space usage as input size grows to infinity. It ignores constant factors and focus on dominant terms.

Types -

(i) Big - O - represents upper bound (worse case)
(ii) Big - omega - lower bound (best case)
(iii) Big - theta - tight bound (average case)

It provides hardware - independent way to compare efficienty and scalability of algo.

2. Head recursion - recursive call is made before main operation. This builds up stack consuming O(n) space.

Example

```
void head (int n) {
 if (n ==0) return;
 head (n-1);
 SOUT (n); }
```

OUTPUT → 1 2 3

Tail recursion - Recursive call is very last operation in function. No computation after call returns. It can be optimised by compilers to run O(1) space converting them into iterative loop.

Example

```
void tail (int n) {
 if (n == 0) return;
 SOUT (n);
 tail (n-1); }
```

OUTPUT → 3 2 1

3. For Array $A[i][j]$ where base add. is $A[0][0]$ with M rows & N columns

(i) Get to correct row - To reach i^{th} row, we need to skip row 0 to row $i-1$.

(ii) Count skipped elements - row x element in each row (column no.) $i \times N$

(iii) Get to correct column - move to j^{th} element of row i, skip j elements

(iv) Total skipped elements - $(i \times N) + j$

(v) Byte offset - Multiply skipped elements with element size

$$\text{offset} - ((i \times N) + j) \times w$$

(vi) Final address

$$A[i][j] = ((i \times N) + j) \times w + \text{Base}$$

4.

Linear Search	Binary Search
(i) Can be unsorted or sorted	(i) Must be sorted
(ii) Check each element from start until target is found or end of array is reached.	(ii) Checks middle element. If target is smaller, repeats process on left half otherwise on right half
(iii) Best case - O(1) Target is first element	(iii) Best case - O(1) Target is middle element
(iv) Average case - O(n)	(iv) Average case - O(logn)

5. Algorithm

(i) Start from second element (index 1)

for i from 1 to length (A) - 1

(ii) Store element to inserted

key = A[i]

(iii) Initialise j to index before key

j = i - 1

(iv) Move elements of A[0 .. i-1] that are greater than key to one position ahead of current position

while j >= 0 && A[i] > key

A[j+1] = A[j]

j = j - 1

(v) Place key in corrected sorted position

A[j+1] = key

6. Sparse Matrix is matrix in which most elements are zero & storing them in standard 2-D array is highly inefficient.

To save space we represent them in only non-zero elements

Triplet (coordinate list)

It uses 2-D array with 3 coloumns to store all information about non-zero elements

Coloumn 0 - stores row index

column 1 - stores column index

column 2 - stores value of non-zero elements

7. Algorithm

(i) Initialise

prev = NULL , curr = head

(ii) while

curr != NULL : next = curr → next;

curr → next = prev ; prev = curr; curr = next

(iii) At end prev is new head

Example 1 → 2 → 3 → NULL becomes

3 → 2 → 1 → NULL

8. a) Algorithm

TowerofHanoi (n, source, desti, temp)

// Base Case

If n == 1 then

Move disk 1 from source to desti

Return

END IF

// Recursive Step 1

// Move n-1 disks from source to temp,

using desti as helper

TowerofHanoi (n-1, source, temp, desti)

// Recursive step 2

// Move n-1 disk from temp to desti, using source

as helper

TowerofHanoi (n-1, temp, desti, source)

For recursive steps
(i) First, recursively move top n-1 from source rod to temp rod
(ii) Second, move n^{th} disk from source to desti
(iii) recursively move n-1 disks from temp rod to desti rod

b) Trade-offs

Recursion	Iteration
(i) Results in shorter, simpler & readable code	(i) Can be longer & more complex to write especially for nested structur
(ii) Uses call stack. Can lead to stack overflow	(ii) Uses heap memory. Memory usage is generally O(1) constant or explicitly managed.
(iii) Slower due to overhead of function calls	(iii) Faster because it avoid function calls

9. Merge sort

Divide → Sort → Merge

Divide ↓ Every array divides into two halves

Merge ↓ Each element gets merged with other element after sorting

Example [8, 3, 1, 7, 0, 10, 2]

1. Divide
[8, 3, 1, 7] & [0, 10, 2]
[8, 3] & [1, 7] & [0, 10] & [2]
[8] & [3] & [1] & [7] & [0] & [10] & [2]

2. Merge

[3,8] & [1,7] & [0,10] & [2]

[1,3,7,8] & [0,2,10]

[0,1,2,3,7,8,10]

Time complexity for merge sort in all best, average, worst cases is O(nlogn)

(i) logn - divide step, logn level of recursion

(ii) n - Every element in conquer stage must be processed during merge phase. This takes O(n) time per level.

(iii) Total time complexity - O(nlogn)

Benefits over bubble sort

(i) Effeciency - Merge's sort O(nlogn) is superior than bubble's sort average & worst case $O(n^2)$

(ii) ~~Scalability~~ Stability - Merge sort is stable, it preserves original relative order of elements of equal values.

Set-2