HW Questions AI Lab Homework

Task 1: Fabonacci Series:-

1) Simple Iterative:

Dry Run:- For n=5

- Initially a=0, b=1
- Step 1: a=1, b=1
- Step 2: a=1, b=2
- Step 3: a=2, b=3
- Step 4: a=3, b=5
- Return 5.

Time Complexity:-

O(n): We iterate through loop n-1 times, making the approach linear.

2) Simple Recursive:-

- Dry Run:- For n=5:
  - fabo_recursive = fabo_recursive(4) + fabo_recursive(3)
  - fabo_recursive = fabo_recursive(3) + fabo_recursive(2)
  - Repeating until base case reached
- Time Complexity:-
  - $O(2^n)$: Each function call spawns two more calls, resulting in an exponentially time complexity

3) Dp-Memoization:-

Dry Run:- (For n=5)

- Calls fabo_memo(5) → checks memo, not found

• Calls fibo_memo(4) → checks memo, not found
• Similarly for 3, 2 and 1. Once calculated, result store in memo.

Time Complexity:-

O(n): Each fibo number is calculated once and then stored in the memo dictionary.

4) Dp-Tabulation:-

Dry Run:- For n = 5.

• dp = [0, 1, 0, 0, 0, 0]
• After step 1: dp = [0, 1, 1, 0, 0, 0]
• After step 2: dp = [0, 1, 1, 2, 0, 0]
• After step 3: dp = [0, 1, 1, 2, 3, 0]
• After step 4: dp = [0, 1, 1, 2, 3, 5]

Time Complexity:-

O(n): We build a table dp from 0 to n, each value is computed in constant time.

Task 2: Minimum Coin Change:-

1) Simple Iterative:-

Dry Run:- For coins = [1, 3, 4], amount = 6:

• Coin 4 : 6 − 4 = 2
• Coin 1 : 2 − 1 = 1 , 1 − 1 = 0
• Return 3 (3 coins used)

Time Complexity:

$O(n \log n)$: Sorting the coins array takes $O(n \log n)$.

· In worst case, it goes through all the coins for each denomination.

2) Simple Recursive:-

Dry Run:- For coins = [1, 3, 4], amount = 6.

min_coins_recursive(6) checks 3 branches.

· For 4, min_coins_recursive(2) → leads to using 1 and 1

· For 3, min_coins_recursive(3) → similar process.

Time Complexity:-

$O(2^n)$: The recursive tree grows exponentially since each call spawns multiple recursive calls.

3) Dp-Memoization:-

Dry Run:- For coins = [1, 3, 4], amount = 6.

· Calls min_coins_memoization(6) and stores results for smaller amounts in the memo, avoiding redundant recalculations.

Time Complexity:-

$O(n * amount)$: Each subproblem is calculated only once and stored in the memo dictionary.

4) **Dp-Tabulation:-**

Dry Run:- For coins = [1,3,4], amount = 6.

- dp = [0, inf, inf, inf, inf, inf, inf]
- For amount 1 using coin 1 dp = [0, 1, inf, inf, inf, inf, inf]
- For amount 2 using coin 1 dp = [0, 1, 2, inf, inf, inf, inf]
- For amount 3 using coin 3 dp = [0, 1, 2, 1, inf, inf, inf]
- Continue until dp[6] = 2

Time Complexity.

O(n * amount): We iterate through each coin for each amount, which gives us a time complexity of O(n * amount).

Task 3:- Longest Common Subsequence:-

1) **Simple Iterative:-**

This is not possible we need to check all possible subsequence, so we neglect it.

2) **Simple Recursive:-**

Dry Run:-

For str1 = "ABC", str2 = "AC", m = 3, n = 2.

- lcs_recursive(3,2) compares C and C → 1 + lcs_recursive(2,1).

Time Complexity:-

$O(2^n)$: Each comparison branches into two recursive calls, leading to an exponential growth.

3) Dp-Memoization:-

Dry Run

For str1="ABC", str2="AC", m=3, n=2

Call lcs_memo(3,2) → check memo, avoids recalculating and stores values for each subproblem.

Time Complexity:-

O(m*n): Each subproblem (for every (m,n)) is calculated only once and stored in the memo.

4) Dp-Tabulation:-

Dry Run:-

For str1="ABC", str2="AC"

• Initially dp=[[0,0,0],[0,0,0],[0,0,0],[0,0,0]]

• After comparing each character pair and filling in table dp[3][2]=2 (the length of LCS is 2)

Time Complexity:-

O(m*n): We iterate through all possible pairs of characters in str1 and str2, resulting in O(m*n) time complexity.

CamScanner

Task 4: Climbing Stairs 1) Simple Iterative:

Dry Run (For n=5)

- Initially a=1, b=1
- Step 1: a=1, b=2
- Step 2: a=2, b=3
- Step 3: a=3, b=5
- Step 4: a=5, b=8
- Return 8 (8 distinct way to climb stairs)

Time Complexity:-

O(n): We compute each steps value once, resulting in linear.

2) Recursive:-

Dry Run- For n=5

- climb_recur(5) = climb_recur(4) + climb_recur(3)
- climb_recur(4) = climb_recur(3) + climb_recur(2)
- Repeating this until base case.

Time Complexity:-

O(2^n): Each function call splits into two recursive calls. leading Exponential time.

3) Dp-memoization:-

Dry Run:- For n=5:

- climb_memo(5) compute and store values for n=4,3,2, and so on, avoiding recomputation.

Time Complexity:-

O(n): Each subproblem is solved once, the result is stored in memo.

4) Dp-Tabulation:-

Dry Run:- For n=5

- dp = [1, 1, 0, 0, 0, 0]
- After step 2 dp = [1, 1, 2, 0, 0, 0]
- After " 3 dp = [1, 1, 2, 3, 0, 0]
- After " 4 dp = [1, 1, 2, 3, 5, 0]
- After " 5 dp = [1, 1, 2, 3, 5, 8]
- Return 8 (8 distinct ways to climb 5 stairs)

Time Complexity:-

O(n): We return n from 2 to n, filling in table with number of ways to reach each step.

Task 5:- Knapsack Problem

1) Simple Iterative:-

Greedy approach may fail to produce optimal results.

2) Recursive:-

Dry Run:-

For weights = [1, 3, 4, 5], values = [1, 4, 5, 7], w = 7, n = 4:
knap_recur(7,4) splits into include and exclude 4th item and so on recursively.

Time Complexity:-
O(2^n): For every item, it creates branches leads to exponential.

3) Dp-Memoization:-

Dry Run:-

. For weights = [1, 3, 4, 5], values = [1, 4, 5, 7], W = 7, n = 4

. knapsackmemo(7, 4) computes and stores for different weights and items indices in the memo

Time Complexity:-
O(n*W): Each subproblem solved only once and result is stored, so total subproblems is O(n*W).

4) Dp-Tabulation:-

Dry Run:-

For weights = [1, 3, 4, 5] values = [1, 4, 5, 7], W = 7, n = 4.

. Initial dp table filled with 0s

. After processing each item, the table is updated based on whether we include current item or not.

. Final result is in dp[4][7].

Time Complexity:-
O(n*W): We fill a table of size (n*W) where each entry calculated once.