DSA 4 - Interview Problems

Problem 1 Target Sum

You are given a set of non-negative integers and a target sum. The task is to determine whether there exists a subset of the given set whose sum is equal to the target sum.

Examples
Input: set = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
Explanation: There is a subset (4, 5) with a sum of 9.

:::warning
Please take some time to think about the brute force approach on your own before reading further.....
:::

Approach: Brute Force

The brute force approach involves exploring all possible subsets of the given set. This is achieved through a recursive function named isSubsetSum(set, n, sum). The function explores two possibilities for each element: including it in the subset or excluding it. The base cases check if the sum is zero or if there are no more elements to consider.

Code

```
boolean isSubsetSum(int[] set, int n, int sum) {
    if (sum == 0) return true;
    if (n == 0 && sum != 0) return false;

    // Explore two possibilities: include the current element or exclude it
    return isSubsetSum(set, n - 1, sum) || isSubsetSum(set, n - 1, sum - set[n - 1]);
}
```

Complexity Analysis

- **Time complexity**: O(2^n) - Exponential
- **Space complexity**: O(1)

Target Sum Dynamic Programming Approach

Approach:

To optimize the solution, dynamic programming is employed. A 2D array `dp` is used to store results of subproblems. The value `dp[i][j]` represents whether it's possible to obtain a sum of `j` using the first `i` elements of the set.

Code

```java
boolean isSubsetSum(int[] set, int n, int sum) {
    boolean[][] dp = new boolean[n + 1][sum + 1];

    // Initialize the first column as true, as sum 0 is always possible
    for (int i = 0; i <= n; i++) dp[i][0] = true;

    // Fill the dp array using a bottom-up approach
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            dp[i][j] = dp[i - 1][j];
            if (j >= set[i - 1]) dp[i][j] = dp[i][j] || dp[i - 1][j - set[i - 1]];
        }
    }

    return dp[n][sum];
}
```

Complexity Analysis

- **Time complexity**: O(n * sum)
- **Space complexity**: O(n * sum)

Problem 2 Minimum Jumps to Reach End

You are given a 0-indexed array of integers nums of length n. You are initially positioned at nums[0].

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j] where:

- 0 <= j <= nums[i]
- i + j < n

Return the minimum number of jumps to reach nums[n - 1]. The test cases are generated such that you can reach nums[n - 1].

Example 1

Input: nums = [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2

Input: nums = [2,3,0,1,4]
Output: 2

:::warning
Please take some time to think about the solution approach on your own before reading further.....
:::

Intuition

The code utilizes a greedy approach, aiming to minimize the number of jumps by selecting the jump that maximizes the range of reachable positions at each step.

Approach

- Initialize a counter to keep track of the number of jumps (counter).
- Use a while loop to iterate through the array until the end is reached.
- In each iteration:
    - Calculate the range of positions that can be reached from the current position (range).
    - If the calculated range includes the last position or exceeds it, exit the loop.
    - Find the position within the current range that maximizes the next reachable position (temp).

    - Update the current position to the selected position (temp).
    - Increment the jump counter (counter).
- Repeat the process until the end is reached.

Code

```java
class Solution {
    public int jump(int[] nums) {
        int i = 0;
        int counter = 0;

        // If the array has only one element, no jumps are needed
        if (nums.length == 1)
            return 0;

        while (i < nums.length) {
            counter++;
            int range = i + nums[i];

            // If the range includes the last position or exceeds it, exit the loop
            if (range >= nums.length - 1)
                break;

            int max = 0;
            int temp = 0;

            // Find the position within the current range that maximizes the next reach
            for (int k = i + 1; k <= range; k++) {
                if (nums[k] + k >= max) {
                    max = nums[k] + k;
                    temp = k;
                }
            }
            i = temp;
        }
        return counter;
    }
}
```

Complexity Analysis

Time complexity: O(n)

- The algorithm iterates through each element of the array once.
- The inner loop within each iteration also traverses a limited number of positions.

Space complexity: O(1)

- The algorithm uses a constant amount of extra space.
- The space requirements do not depend on the size of the input array.

Problem 3 N digit numbers

Find out the number of A digit positive numbers, whose digits on being added equals to a given number B.

Note that a valid number starts from digits 1-9 except the number 0 itself. i.e. leading zeroes are not allowed.

Since the answer can be large, output answer modulo 1000000007

:::warning
Please take some time to think about the brute force approach on your own before reading further.....
:::

Brute Force Approach

Objective

Count the number of A-digit positive numbers whose digit sum equals a given number B.

Idea

- Generate all possible A-digit numbers
- For each number, check if the sum of its digits equals B.
- Keep track of the count of valid numbers.

Code

```
int bruteForceCount(int A, int B) {
    int count = 0;
    for (int num = pow(10, A - 1); num < pow(10, A); ++num) {
        int digitSum = 0;
        int temp = num;
        while (temp > 0) {
            digitSum += temp % 10;
            temp /= 10;
        }
        if (digitSum == B) {
            count++;
        }
    }
    return count;
}
```

N digit numbers optimization using Recursion

Observation

The brute force approach involves iterating through all A digit numbers and checking the count of numbers whose digit sum equals B.

Optimized Recursive Approach

Recursive Function Definition

- The recursive function `countWays(id, sum)` is defined to represent the count of A-digit numbers with a digit sum equal to sum.
- The base cases are as follows:
    - If sum becomes negative, it implies that the digit sum has exceeded the target, so the count is 0.
    - If id becomes 0, meaning all digits have been considered, the count is 1 if sum is 0 (indicating the target sum has been achieved), and 0 otherwise.
    - Memoization is used to store and retrieve previously calculated values, preventing redundant computations.

Memoization Table

- The memo vector is a 2D table of size (A + 1) x (B + 1) initialized with -1 to represent uncalculated states.
- The value `memo[id][sum]` stores the count of A-digit numbers with a digit sum of sum that has already been calculated.

Recursive Part

- The function explores all possible digits from 0 to 9 in a loop.
- For each digit, it recursively calls `countWays(id - 1, sum - digit, memo)` to calculate the count of (A-1)-digit numbers with the updated digit sum.
- The results are summed up, and the modulus operation is applied to avoid integer overflow.

Optimized Recursive Code

```
int memoizationCount(int A, int B) {
    vector<vector<int>> memo(A + 1, vector<int>(B + 1, -1));
    return countWays(A, B, memo);
}

int countWays(int id, int sum, vector<vector<int>>& memo) {
    if (sum < 0) return 0;
    if (id == 0) return (sum == 0) ? 1 : 0;
    if (memo[id][sum] != -1) return memo[id][sum];

    int ways = 0;
    for (int digit = 0; digit <= 9; ++digit) {
        ways += countWays(id - 1, sum - digit, memo);
        ways %= 1000000007;
    }
    return memo[id][sum] = ways;
}
```

Optimized Iterative Code

```
int iterativeCount(int A, int B) {
    vector<vector<int>> dp(A + 1, vector<int>(B + 1, 0));

    // Base case
    for (int digit = 1; digit <= 9; ++digit) {
        if (digit <= B) dp[1][digit] = 1;
    }

    // Build DP table
    for (int id = 2; id <= A; ++id) {
        for (int sum = 1; sum <= B; ++sum) {
            for (int digit = 1; digit <= 9; ++digit) {
                if (sum - digit >= 0) {
                    dp[id][sum] += dp[id - 1][sum - digit];
                    dp[id][sum] %= 1000000007;
                }
            }
        }
    }

    return dp[A][B];
}
```

Time Complexity : O(A * B)

Space Complexity : O(A * B)

Problem 4 Maximum Profit from Stock Prices

Given an array A where the i-th element represents the price of a stock on day i, the objective is to find the maximum profit. We're allowed to complete as many transactions as desired, but engaging in multiple transactions simultaneously is not allowed.

:::warning
Please take some time to think about the solution approach on your own before reading further.....
:::

Approach

Let's start with some key observations:

Note 1: It's never beneficial to buy a stock and sell it at a loss. This intuitive insight guides our decision-making process.

Note 2: If the stock price on day i is less than the price on day i+1, it's always advantageous to buy on day i and sell on day i+1.

Now, let's delve into the rationale behind Note 2:

Proof: If the price on day i+1 is higher than the price on day i, buying on day i and selling on day i+1 ensures a profit. If we didn't sell on day i+1 and waited for day i+2 to sell, the profit would still be the same. Thus, it's optimal to sell whenever there's a price increase.

DP-Based Solution

Now, let's transition to a dynamic programming solution based on the following recurrence relation:

Let Dp[i] represent the maximum profit you can gain in the region (i, i+1, ..., n).

Recurrence Relation: `Dp[i] = max(Dp[i+1], -A[i] + max(A[j] + Dp[j] for j > i))`

Here, Dp[i] considers either continuing with the profit from the next day (Dp[i+1]) or selling on day i and adding the profit from subsequent days.

Base Cases

When i is the last day (i == n-1), Dp[i] = 0 since there are no more future days.
When i is not the last day, Dp[i] needs to be computed using the recurrence relation.

Direction of Computation

We start computing Dp[i] from the last day and move towards the first day.

Code

```
int max_profit(vector<int>& A) {
    int n = A.size();
    vector<int> dp(n, 0);

    for (int i = n - 2; i >= 0; --i) {
        dp[i] = dp[i + 1];

        for (int j = i + 1; j < n; ++j) {
            if (j + 1 < n) {
                dp[i] = max(dp[i], -A[i] + A[j] +  dp[j + 1]);
            } else {
                dp[i] = max(dp[i], -A[i] + A[j]);
            }
        }
    }

    return dp[0];
}
```

Optimized Code

The provided code snippet in C++ contains this observation-based solution. It iterates through the array, checks for price increases, and accumulates the profits accordingly.

```
int Solution::maxProfit(const vector<int> &A) {
    int total = 0, sz = A.size();
    for (int i = 0; i < sz - 1; i++) {
        if (A[i + 1] > A[i])
            total += A[i + 1] - A[i];
    }
    return total;
}
```

Time Complexity : O(|A|)
Space Complexity : O(1)