Operating Systems and Algorithms - Assignment 2

Analysis Report

Sreeja Guduri (2021102007)

1. Time to Judge

ALGORITHM:

We use dynamic programming, or more specifically, memoization to solve this problem. We make a DP 2-D array to store the answers to all operations that we have already computed to reduce time complexity.

In my function, 'sum' is the variable that stores the most optimal time we get until then. Total time is the sum of the processing time of each problem (which is the maximum time - one problem at a time). So, we look at every possibility and see which one gives us the least processing time by either including a particular problem with judge 1 ('include' recursion) or leaving it for judge 2 ('exclude' recursion).

We then store the minimum of those two values in the DP array and keep checking if the problem we are dealing with has already been solved.

Thus, the DP formula is:

dp[i][j] = min{including T[i], excluding T[I]}

TIME COMPLEXITY:

The time complexity will be O(n*totalSum), where n is the number of problems to judge and totalSum is the sum of all the processing times of the problems.

This is because when calling the recursion function, we call it for all problems (n) and we could go until the maximum possible answer for minimum time. Thus, time complexity is directly proportional to the size of DP table.

SPACE COMPLEXITY:

Since we have a 2-D DP table of size (n*totalSum), the space taken up will also be (n*totalSum).

2. Gamers Dilemma

ALGORITHM:

The logic for this problem is a direct application of the knapsack problem. We make a DP table of size (n+1) x (b+1) where n is the number of game disks we can pick from and b is the budget available.

DP[i][j] represents the maximum number of games we could get with ‘i’ game disks and ‘j’ budget. Thus, we choose to include or exclude a particular game disk and store the maximum in the dp array.

The DP formula is given by:

dp[i][j] = max{dp[i-1][j], value[i-1] + dp[i-1][j - price[i-1]}

Here,

dp[i-1][j] represents the maximum value of games we would have if game disk ‘i’ was excluded.

value[i-1] + dp[i-1][j - price[i-1] represents the maximum value of games we would have if game disk ‘i’ was included. (dp[i-1][j - price[i-1] gives us the maximum number of games we could get with the remaining budget)

TIME COMPLEXITY:

The time complexity of the given question is O(n*b) where n is the number of game disks and b is the budget available. This is because we have to loop through the DP array to figure out every possible combination.

SPACE COMPLEXITY:

The space complexity will similarly be (n*b) because this is the size of the DP 2-d array.

3. Beautiful Arrays

ALGORITHM:

The logic here is to again, use dynamic programming to figure out how to split the array such that it becomes divisible by ‘I’.

To do this we again use a DP table dp[I][j] which represents the number of ways to split the first ‘i’ elements such that it is divisible by j.

We start by forming the prefix sum array for the original array so that we can figure out the sums of subarrays quickly. Then to update the DP table, we start by considering a specific number of elements and looping through different subarray lengths and storing the number of splits.

The DP formula for this is:

dp[j+1][prefix[i] % (j+1)] += dp[j][prefix[i] % j]

Here,

prefix[i] % (j+1) calculates if the sum of current subarray is divisible by its index

We calculate the number of splits required by using the efficient number of splits calculated for first ‘i’ elements divisible by ‘j’.

Then the final answer is calculated by considering all N elements of the original array and adding up all the ways we could split the array.

TIME COMPLEXITY:

The time complexity of this is O(N^2) because we loop through the DP array of size (N*N) to store all the ways to split the array.

SPACE COMPLEXITY:

The space complexity is N^2 because of the fact that we use a DP array of size (N*N).