DYNAMIC PROGRAMMING - 3

MNNIT COMPUTER CODING CLUB

0/k Knapsack

Similar to the regular knapsack with the fact that each item can be selected at most k times.

Problem Description

Given an array A of size N, where 1 <= N <= 100. You need to print an array B of size 1 <= S <= 1e5, where B[i] = 1 if it is possible to achieve a sum of i considering each element of A can be used at most 1 <= k <= 100 times.

Trivial Way

We can simply try to convert this problem into 0/1 knapsack by simply creating a new array A where we duplicate each A[i] k times.

But the complexity would be **O(nkS)**

Let’s try to do it in **O(nS).**

Pre-requisites

- Space optimized implementation of 0/1 and 0/oo knapsack.
- Equivalence classes.
- Sliding window problem

Observations

Consider finding answer on the basis of A[i]. Notice that the entire space of size S can be divided into A[i] different equivalence classes where elements in each class can be independently reduced.

Then using **sliding window** technique, each equivalence class can be independently solved in O**(n * S)** amortised complexity.

TRACING THE ACTUAL ANSWER BACK

Sometimes the problem expects us to trace the exact choices which led us to the optimal answer. For example in case of LIS problem, we may be asked to trace the actual subsequence of maximum length or in case of knapsack we may be asked to specify the exact items which we selected.

How to approach these problems? We'll understand this by taking an example.

Problem Description

Given N items each having a weight and price associated with it. Consider we have a knapsack of size W, find the maximum price we can get if we can taken only those elements whose cumulative weights <= W (0/1 knapsack). Besides finding the optimal answer find the exact items which you selected.

Approach

To find the optimal answer i.e., the maximum price is trivial knapsack dp. We'll get the following transitions.

fun(index , weight_rem) = max(fun(index - 1 , weight_rem) , fun(index - 1 , weight_rem - weight[index]) + price[index]).

Notice for getting the optimal answer for the state (index , weight_rem) we use only one of the possible two transitions. Thus if we start from the final state (N , W) and keep tracing back the transitions we took for each state for our optimal answer then we can easily get the exact choice which we took.

Pseudo Code

```
void trace (vector<vector<int>>& dp) {
    int N = dp.size();
    int W = dp[0].size()-1;
    int curr_index = N-1 , curr_weight = W;

    while(curr_index >= 0 && curr_weight > 0) {
        /*
            possible prev states
        */
        int prev_index = curr_index - 1 , prev_weight1 = curr_weight , prev_weight2 = curr_weight - weight[curr_index];
        if(prev_weight2 < 0) {
            curr_index = prev_index;
            curr_weight = prev_weight1;
            // The item[index] isn't taken
        }
        else {
            if(dp[prev_index][prev_weight1] >= dp[prev_index][prev_weight2] + price[curr_index]) {
                // item[index] not taken
                curr_index = prev_index;
                curr_weight = prev_weight1;
            }
            else {
                //item index is taken
                curr_index = prev_index;
                curr_weight = prev_weight2;
            }
        }
    }
}
```

prev_weight2 int prev_weight2

So in short the each choices can be traced by knowing the exact transition which led to the optimal answer for the current state and moving to the corresponding state.

This is continued until we reach the base case.

BITSET TRICK

Bitset Definition

std::bitset - C++ Reference

In simpler words bitset is a DS where:

- Where we can define the size of it like boolean arrays (size should be static).
- We can apply bit operations on it like bitmasks.

Problem Description

Given an array A of size N, where 1 <= N <= 1e5. You need to print an array B of size 1 <= S <= 1e5, where B[i] = 1 if it is possible to achieve a sum of i considering each element of A can be used at most 1 time.

Trivial Way

We can simply use a 0/1 knapsack DP for it.

The complexity of it would be O(N * S).

Optimisation using bitsets.

The same thing could be done using bitsets.

Let b be the bitset representing the dp array.

Then for a particular A[i] we can update b as follows:

```
b = b | (b << A[i])
```

Does bitset implementation improves complexity?

A big NOOOOO.

The complexity is still O(N*S)

So how does it help?

It improves the constant factor of the actual time which isn’t accounted in the complexity.

In particular if W = 32 bits is the word size then the complexity is O(N * S / W) and this W helps in squeezing the O(N * S) algorithm under TL.

BITMASK DP

What is a Bitmask

A bitmask is set of bits (a binary number) which is used to represent something.

Eg. The number 5 (101) can represent a subset of an array of size 3, where the first and the 3rd elements are taken.

Need of Bitmask DP

Bitmasks can be used to represent states which are not possible to represent using simple variables.

Solving a couple of problems would help us in understanding the concept better.

Problem 1

The are N men and N women in a prom. You are given a N * N matrix A , where A[i][j] = 1 if the man i and woman j are compatible to dance together, otherwise A[i][j] = 0. You need to find out the number of ways in which pairing can be done such that all the couples formed are compatible with each other. Output modulo 1e9 + 7;

Problem Link : https://atcoder.jp/contests/dp/tasks/dp_o

Thoughts?

Let’s consider all the women in a sequential order and select a man from the remaining men to form a couple depending on their compatibility factor. Now after one man has been selected how do we acknowledge this fact in the upcoming states?

Unlike all the problems we have done so far **we not only need to know how many couples have been formed till now, but we also want to know exactly which men have been selected and which are remaining.**

How do we acknowledge this fact as a state?

Solution

This information can be represented using a selected array kind of thing. We can maintain an array selected where selected[i] = 0 if the ith man hasn’t been paired yet.

However using an array as a state variable is very costly. That’s where bitmask comes into play. Instead of maintaining a boolean array we can maintain a bitmask representing the same thing.

Problem 2

You are given a room of size N * M where the floor is divided into cells of size 1*1. Some cells have pillars on them. You want to lay floorboards on the floor. Floorboards are rectangles of size 1 * k where k can be anything.You can also rotate the floorboards. What is the minimum number of floorboards required?

Constraints : 1 <= N <= 10 , 1 <= M <= 10

Will Trivial Bitmask DP work?

If we consider the entire matrix as a mask, then complexity would be O (2 ^ (NM)) which is way too high to pass.

So trivial implementation won’t help.

Solution

Since it is obvious that every cell has to be filled with a floorboard, so there are basically 4 ways in which the cell (i , j) could be filled:

- The horizontal floorboard currently running can be extended to the right (If possible)
- The vertical floorboard just above can be extended downwards (If possible).
- A new horizontal floorboard can be started from the current cell.
- A new vertical floorboard can be started from the current cell.

To maintain whether a horizontal floorboard can be extended to (i , j) we just need one boolean variable.

To maintain whether a vertical floorboard can be extended we can maintain a mask for the upper row where jth bit is 1 if a vertical floorboard can be extended from jth column .