DYNAMIC PROGRAMMING

MNNIT COMPUTER CODING CLUB

KNAPSACK PROBLEMS

0/1 Knapsack

Given **N** items each having its corresponding weight (**w[i]**) and price value (**val[i]**), we need to find out the **maximum price** of the items we can take such that those items fit into a knapsack of size **W**.

Solution

Let’s say we have a function fun(i , j) where i refers to the fact that we have to process i items and j means we currently have j capacity in the knapsack.

fun(i , j) returns the maximum price that we can accumulate.

For the transitions we can do the following if we are currently at state (i , j):

- We take the current item and put it into the knapsack that means the price we’ll get from this transition is val[i] + dp(i-1 , j - w[i])
- We ignore the current item i.e., price in this case would be dp(i-1 , j)

So overall **dp(i , j) = max(dp(i-1 , j) , dp(i-1 , j-w[i]) + val[i])**

0/oo Knapsack

The problem is similar to the 0/1 knapsack. The only difference is that the frequency of every item is infinity i.e., unlike the 0/1 knapsack where we can choose each items at most once, in this variant we can choose every item any number of times.

Will the solution of 0/1 knapsack work?

The initial thoughts that come to our mind is to re-use the solution of 0/1 knapsack problem. Can we do so?

Answer is NO.

Because in the standard 0/1 knapsack problem after considering ith item we always move on to process the next item, regardless of whether we take it or not. This accounts for the restriction that only 1 occurrence of each item can be considered.

How can the 0/1 knapsack solution be modified?

To be able to modify the solution of 0/1 knapsack to fit the needs of the current problem we need to make some observations:

- The optimal solution can always be achieved if we process the items in the given order. In other words it isn't possible that in the optimal solution we have to re visit a particular item after fetching another item. This observation can also be expressed as if we decide to not take the current item then we can simply move to the next item. I.e., dp(i-1 , j)
- If we decide to take the current item then there is no need to move to the next item as we can take the current item again. Ie., dp(i , j-w[i]) + price[i]

Final Solution

dp(i , j) = max(dp(i-1 , j) , dp(i , j-w[i]) + val[i])

Where dp(0 , 0) = 0

Space Optimizations

Currently for both 0/1 and 0/oo knapsack the space complexity of the solutions we are using is **O(N * W).**

Can we reduce this complexity? If yes then How?

Basic Idea for space optimizations

The most important observation for space optimization is that at any particular point of time we might not need all the possible states but only a handful of them.

So if instead of storing all the states we only memoise the required states, we can considerably bring down the space required.

How can we use this idea for knapsack problem

For knapsack problem what we see is that when calculating dp(i , j) we always need something of the form dp(i-1 , k) or dp(i , k) (in case of 0/oo) where 0 <= k <= W.

So there is no need to store the values like dp(i-2 , k) , dp(i-3 , k) and so on. So we only need **O(W)** space for our algorithm.

Simple trick

- For 0/1 knapsack process j from W to 0.
- For 0/oo knapsack process j from 0 to W.

KADANE'S ALGORITHM

Purpose of the Algorithm

To find the maximum subarray sum for an array A, which allows negative elements also.

Working of the algorithm

Let fun(i) return the maximum subarray sum of the array A[0...i]. Thus now using fun(i) let's try to find fun(i+1)

Then there are two possibilities:

- The optimal subarray has A[i+1] as one of its element. Since fun(i+1) only considers array A[0 … i+1] thus for the current scenario if we decide to include A[i+1] the fun(i+1) would be the maximum suffix value.
- If we decide not to include A[i+1] then fun(i+1) is simply fun(i)

So **fun(i+1) = max(fun(i) , max(sum(j … i+1))** for some j.

What is the time complexity of the solution?

Answer : **O(N * N)**.

Wait but isn’t it the same as the optimized brute force solution??

So why take all the pain in deriving a new solution??

The answer is because the new solution can be easily optimized to linear time.

How to optimize to linear Complexity?

One observation which we can make is if we are somehow able to calculate sum(j … i+1) in O(1) then the overall complexity will be O(N).

So how to solve this problem (To find maximum value of a subarray of the given array such that the right bound is given in O(1))?

Solution

We know sum(j …. i+1) = pref(i+1) - pref(j-1).

So all we need to do is minimize pref(j-1).

That can done by maintaining a prefix minimum array for the prefix sum.

So the Kadane's algorithm would reduce to

fun(i+1) = max(fun(i) , pref[i+1] - pref_min[i])

where , **pref_min[i] = min(pref_min[i-1] , pref[i]) and pref[i] = pref[i-1] + A[i]**

Complexity of the optimized solution?

O(N)

This is indeed the entire Kadane’s algorithm. Seems intriguing?

Is it really this hard??

Actually No. The space efficient implementation is really simple. Let’s have a look.

Space efficient solution of Kadane's algorithm

We just use 2 variables:

- curr_sum : The current maximum suffix sum i.e., if we are processing ith element then curr_sum = max(sum(j … i)). So to find curr_sum when we process ith element curr_sum = max(curr_sum + A[i] , A[i]).
- max_sum : The global maximum subarray sum found till now.

And the implementation becomes really short with a wonderful linear time complexity and constant space usage.

Problem

Given an array A of size N <= 1e6.

To find : Maximum value of summation(a[k]) - (j - i + 1), i <= k <= j

Solution

Rewrite the problem in the following way:

To find : maximum value of (A[i] - 1) + (A[i+1] - 1) ……. (A[j]-1)

If we make a new array B such that B[i] = A[i] - 1, then all we need to do is run kadane’s algorithm on B to get our answer.

LONGEST INCREASING SUBSEQUENCE

Problem description

Given an array A of size N , find the longest increasing subsequence of the array. Ie., to find the **maximum value k** such that there exists 1 <= i1 < i2 < i3 …. < ik <= N and A[i1] < A[i2] < ……. < A[ik]

Solution

One observation we can simply derive from the problem itself is

len(1 ….. ik) = len(1….. i(k-1)) + 1

So if we take our dp state to be the number of elements processed then we can make the following transition

fun(i) = max(fun(j)) + 1 for all j such that j < i and a[j] < a[i]

Complexity analysis

Time complexity : O(N*N)

Space Complexity : O(N)

Can we achieve a better complexity?

Answer is Yes.

Another representation

Instead of taking the state as the index we are considering, we can take the state as the size of the increasing subsequence.

fun(i) would return the minimum ending number of the increasing subsequence of size i.

So how would the transitions look?

```
vector<int> d(n+1, INF);

dp[0] = -INF;


for (int i = 0; i < n; i++) {

    for (int j = 1; j <= n; j++) {

        if (dp[j-1] < a[i] && a[i] < d[j])

            dp[j] = a[i];

    }

}
```

Complexity Analysis

Answer : O(N * N)

It’s still quadratic. Nothing changed. So why use another representation?

Because it is easier to optimise this version.

Optimizing the quadratic solution

Observation : The dp array we maintain will always be sorted.

Proof : Let’s assume our observation is wrong. Then there should exit some i , j such that j < i and dp[j] > dp[i].

However from the very definition of the dp array we know dp[i] is last element of an INCREASING subsequence of size i. So the jth element of that sequence should be < dp[i]. Thus dp[j] should be less than dp[i]. This contradicts our assumption so our observation is correct.

Another observation: Every element we process from array A, we need to update at most one index of dp array.

So we can binary search the possible point of update and check whether update is needed.

Optimized solution

```
vector<int> d(n+1, INF);
     d[0] = -INF;
    for (int i = 0; i < n; i++) {
        int j = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[j-1] < a[i] && a[i] < d[j])
            d[j] = a[i];
    }
    int ans = 0;
    for (int i = 0; i <= n; i++) {
        if (d[i] < INF)
            ans = i;
    }
```

Complexity analysis

Time Complexity : O(NlogN)

Space Complexity : O(N)

Another representation of LIS

Problem : Problem - 977F

Yet another Representation

Let fun(i) mean the maximum subsequence length that ends at i (not the index but number i).

So fun(i) = max(fun(j)) + 1 such that j < i

Time Complexity : O(S * S) S -> maximum value of an array element.

Can be optimized using Range Query Data Structures. Would be discussed afterwards.

LONGEST COMMON SUBSEQUENCE

Problem Description

Given two arrays A and B of size N , M respectively, we are required to find the length of the maximum subsequence which is common to both the arrays.

Also determine the common subsequence.

Solution

Let fun(i , j) be the length of the longest common subsequence if we consider arrays A[0...i] and B[0...j].

Then **fun(i , j) = fun(i-1 , j-1) + 1 if A[i] == B[j]**

max(fun(i , j-1) , fun(i-1 , j)) if A[i] != B[j]

Figuring the actual subsequence

From the transitions we can observe that whenever we are including an element in the subsequence we move from state (i , j) to (i+1 ,j+1). Otherwise we just move either to (i , j+1) or(i+1 , j).

So we start with the final state (N , M) and keep keep moving back while figuring out whether we included the current element in our answer or not. If dp(i-1,j-1) + 1 == dp(i , j) and dp(i-1 , j) and dp(i , j-1) < dp(i , j) then we include current element in our answer and move to (i-1 , j-1) else move to (i-1 , j) or (i , j-1) depending on whichever is larger.

Complexity Analysis

Time Complexity : O(N * M)

Space Complexity : O(N * M)

Practice Problem

Given two strings a and b of digits, find a way to delete digits in the strings so that:

- The two strings are equal
- The sum of the digits deleted is minimized

And return this minimized sum.

Constraints

- $n \leq 500$ where n is the length of a
- $m \leq 500$ where m is the length of b

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 .