Need help understanding the solution for the Jewelry Topcoder solution

Asked 2 years, 11 months ago Active 1 year, 3 months ago Viewed 960 times

3

3

I am fairly new to dynamic programming and don't yet understand most of the types of problems it can solve. Hence I am facing problems in understaing the solution of Jewelry topcoder problem.

Can someone at least give me some hints as to what the code is doing ?

Most importantly is this problem a variant of the subset-sum problem ? Because that's what I am studying to make sense of this problem.

What are these two functions actually counting ? **Why are we using actually two DP tables ?**

```
void cnk() {
  nk[0][0]=1;
  FOR(k,1,MAXN) {
    nk[0][k]=0;
  }

  FOR(n,1,MAXN) {
      nk[n][0]=1;
      FOR(k,1,MAXN)
        nk[n][k] = nk[n-1][k-1]+nk[n-1][k];
     }
}

void calc(LL T[MAXN+1][MAX+1]) {
  T[0][0] = 1;
  FOR(x,1,MAX) T[0][x]=0;
  FOR(ile,1,n) {
    int a = v[ile-1];
    FOR(x,0,MAX) {
      T[ile][x] = T[ile-1][x];
      if(x>=a) T[ile][x] +=T[ile-1][x-a];
    }
  }
}
```

How is the original solution constructed by using the following logic ?

```
FOR(u,1,c) {
  int uu = u * v[done];
  FOR(x,uu,MAX)
  res += B[done][x-uu] * F[n-done-u][x] * nk[c][u];
  }
done=p;
}
```

algorithm dynamic-programming

edited Jan 2 '18 at 12:01
Gijs Den Hollander
540 3 17

asked Jan 2 '18 at 10:41
ng.newbie
1,949 3 12 28

2 Answers

Active Oldest Votes

7

Let's consider the following task first:

- "Given a vector *V* of *N* positive integers less than *K*, find the number of subsets whose sum equals *S*".

This can be solved in polynomial time with dynamic programming using some extra-memory.

The dynamic programming approach goes like this: instead of solving the problem for *N* and *S*, we will solve all the problems of the following form:

- "Find the number of ways to write sum *s* (with $s \le S$) using only the first $n \le N$ of the numbers".

This is a **common characteristic** of the **dynamic programming solutions**: instead of only solving the original problem, you solve an entire family of related problems. The key idea is that solutions for more difficult problem settings (*i.e.* higher *n* and *s*) can efficiently be built up from the solutions of the easier settings.

Solving the problem for $n = 0$ is trivial (sum $s = 0$ can be expressed in one way -- using the empty set, while all other sums can't be expressed in any ways). Now consider that we have solved the problem for all values up to a certain *n* and that we have these solutions in a matrix *A* (*i.e.* A[n][s] is the number of ways to write sum *s* using the first *n* elements).

Then, we can find the solutions for *n*+1, using the following formula:

A[n+1][s] = A[n][s - V[n+1]] + A[n][s].

Indeed, when we write the sum *s* using the first *n+1* numbers we can either include or not V[n+1] (the n+1th term).

This is what the `calc` function computes. (the `cnk` function uses Pascal's rule to compute binomial coefficients)

Note: in general, if in the end we are only interested in answering the initial problem (*i.e.* for *N* and *S*), then the array `A` can be uni-dimensional (with length *S*) -- this is because whenever trying to construct solutions for $n + 1$ we only need the solutions for n and not for smaller values)

This problem (the one initially stated in this answer) is indeed related to the subset sum problem (finding a subset of elements with sum zero).

A similar type of dynamic programming approach can be applied if we have a reasonable limit on the absolute values of the integers used (we need to allocate an auxiliary array to represent all possible reachable sums).

In the zero-sum problem we are not actually interested in the count, thus the *A* array can be an array of booleans (indicating whether a sum is reachable or not).

In addition, another auxiliary array, *B* can be used to allow reconstructing the solution if one exists.

The recurrence would now look like this:

```
if (!A[s] && A[s - V[n+1]]) {
    A[s] = true;

    // the index of the last value used to reach sum _s_,
    // allows going backwards to reproduce the entire solution
    B[s] = n + 1;
}
```

Note: the actual implementation requires some additional care for handling the negative sums, which can not directly represent indices in the array (the indices can be shifted by taking into account the minimum reachable sum, or, if working in C/C++, a trick like the one described in this answer can be applied: https://stackoverflow.com/a/3473686/6184684).

I'll detail how the above ideas apply in the TopCoder problem and its solution linked in the question.

The B and F matrices.

First, note the meaning of the *B* and *F* matrices in the solution:

- B[i][s] represents the number of ways to reach sum *s* using only the **smallest** *i* items
- F[i][s] represents the number of ways to reach sum *s* using only the **largest** *i* items

Indeed, both matrices are computed using the `calc` function, after sorting the array of jewelry values in ascending order (for *B*) and descending order (for *F*).

Solution for the case with no duplicates.

Consider first the case with no duplicate jewelry values, using this example: `[5, 6, 7, 11, 15]`.

Each item given to Bob has value less (or equal) to each item given to Frank, thus in every good solution there will be a separation point such that Bob receives only items before that separation point, and Frank receives only items after that point.

To count all solutions we would need to sum over all possible separation points.

When, for example, the separation point is between the 3rd and 4th item, Bob would pick items only from the `[5, 6, 7]` sub-array (smallest 3 items), and Frank would pick items from the remaining `[11, 12]` sub-array (largest 2 items). In this case there is a single sum (s = 11) that can be obtained by both of them. Each time a sum can be obtained by both, we need to multiply the number of ways that each of them can reach the respective sum (*e.g.* if Bob could reach a sum *s* in 4 ways and Frank could reach the same sum *s* in 5 ways, then we could get 20 = 4 * 5 valid solutions with that sum, because each combination is a valid solution).

Thus we would get the following code by considering all separation points and all possible sums:

```
res = 0;
for (int i = 0; i < n; i++) {
    for (int s = 0; s <= maxS; s++) {
        res += B[i][s] * F[n-i][s]
    }
}
```

However, there is a subtle issue here. This would often count the same combination multiple times (for various separation points). In the example provided above, the same solution with sum 11 would be counted both for the separation `[5, 6] - [7, 11, 15]`, as well as for the separation `[5, 6, 7] - [11, 15]`.

To alleviate this problem we can partition the solutions by "the largest value of an item picked by Bob" (or, equivalently, by always forcing Bob to include in his selection the largest valued item from the first sub-array under the current separation).

In order to count the number of ways to reach sum *s* when Bob's largest valued item is the *i*th one (sorted in ascending order), we can use B[i][s - v[i]]. This holds because using the v[i] valued item implies requiring the sum s - v[i] to be expressed using subsets from the first *i* items (indices 0, 1, ... *i* - 1).

This would be implemented as follows:

```
res = 0;
for (int i = 0; i < n; i++) {
    for (int s = v[i]; s <= maxS; s++) {
        res += B[i][s - v[i]] * F[n - 1 - i][s];
    }
}
```

Extension for the case when duplicates are allowed.

When duplicate values can appear in the array, it's no longer easy to directly count the number of solutions when Bob's most valuable item is v[i]. We need to also consider the number of such items picked by Bob.

If there are *c* items that have the same value as v[i], *i.e.* v[i] = v[i+1] = ... v[i + c - 1], and Bob picks *u* such items, then the number of ways for him to reach a certain sum *s* is equal to:

`comb(c, u) * B[i][s - u * v[i]]` (1)

Indeed, this holds because the *u* items can be picked from the total of *c* which have the same value in comb(c, u) ways. For each such choice of the *u* items, the remaining sum is s - u * v[i], and this should be expressed using a subset from the first *i* items (indices 0, 1, ... *i* - 1), thus it can be done in B[i][s - u * v[i]] ways.

For Frank, if Bob used *u* of the v[i] items, the number of ways to express sum *s* will be equal to:

`F[n - i - u][s]` (2)

Indeed, since Bob uses the smallest i + u values, Frank can use any of the largest n - i - u values to reach the sum *s*.

By combining relations (1) and (2) from above, we obtain that the number of solutions where both Frank and Bob have sum *s*, when Bob's most valued item is v[i] and he picks *u* such items is equal to:

`comb(c, u) * B[i][s - u * v[i]] * F[n - i - u][s] .`

This is precisely what the given solution implements.

Indeed, the variable `done` corresponds to variable *i* above, variable `x` corresponds to sums *s*, the index `p` is used to determine the `c` items with same value as v[done], and the loop over `u` is used in order to consider all possible numbers of such items picked by Bob.

edited Apr 1 '18 at 19:49
ng.newbie
1,949 3 12 28

answered Jan 2 '18 at 13:04
qwertyman
2,858 6 20

what is the use of pascal triangle here? – noman pouigt Jan 2 '18 at 18:01

@nomanpouigt it's related to the fact that values can be repeated. If, for example, we wanted to count how often we can reach sum 10 using values [5, 5, 5], the answer would be "3 take 2". – qwertyman Jan 2 '18 at 18:34

@qwertyman First of all thanks for all the help. You have really shed some light on a problem for which I was banging my head on the wall for 2 weeks. Now in the original subset sum problem, we use a boolean

@qwertyman Also a general question. In DP problems that involve combinatronics what benefit do I get from calculating the binomial coefficients? The coefficients don't actually show the combination it just gives us a count of how many combinations are possible? So how does it help? – ng.newbie Jan 3 '18 at 10:11

@ng.newbie glad it helps! Indeed, in the subset sum problem it is enough to use a bool array, whereas in the TC problem the int array counts the number of times a sum is obtainable. The binomial coefficients often come up in counting problems (but not always) because they represent an elementary quantity -- binomial(n, k) = number of ways to pick k distinct items from a set of n items -- but indeed, they only help with the count, and don't provide a way to retrieve specific combinations. If you feel it's important, I can detail their purpose in the TC problem (which is a counting problem). – qwertyman Jan 3 '18 at 16:49

@qwertyman It will be better if you add it to the answer. These comments are getting long. And thanks once again. – ng.newbie Jan 4 '18 at 11:13

@ng.newbie yes, indeed that part is a little longer -- I will add it to the answer (later today, or tomorrow), and send a comment here to let you know when it's done. – qwertyman Jan 4 '18 at 11:24

1 Let's say that your array only contains two 5s: `[5, 5]` . There is a single allocation of jewels: Frank gets a `5` and Bob gets the other `5` . But you have `c(1, 2)` ways of choosing the `5` you give to Frank. – fjardon Jan 4 '18 at 12:21

@fjardon Actually no you have 2 allocations of jewels according to the question. Frank gets the first 5 and Bob gets the second 5, that's one allocation. The second allocation is that Frank gets the second 5 and Bob gets the first 5. This logic is taken from the question's example. – ng.newbie Jan 5 '18 at 4:58

@ng.newbie Sorry, I meant `c(2, 1)` the french convention is opposite to the english one. And `c(2,1) = 2` as expected. wolframalpha.com/input/?i=choose%5B2,+1%5D – fjardon Jan 5 '18 at 9:26

@ng.newbie I detalied the solution for the TC problem and the use of binomial coefficients -- the key idea is also contained in fjardon's comment above. – qwertyman Jan 5 '18 at 19:03

@nomanpouigt I added more details which also describe the use of the binomial coefficients (which are computed using pascal triangle) in the solution for the topcoder problem. – qwertyman Jan 5 '18 at 19:05

1 @qwertyman I wish I got this guidance when I was starting out competitive programming. I can't thank you enough. – ng.newbie Jan 7 '18 at 8:13

@qwertyman why are you multiplying by v[i]?? Look at equation (1) in your extension foŕ equal elements allowed. – ng.newbie Apr 1 '18 at 19:37

@ng.newbie the "s - u * v[i]" value is what is the sum which remains after picking *u* items, each with value v[i]. – qwertyman Apr 1 '18 at 19:53

@qwertyman If we are counting combinations with respect to Bob then why aren't we considering it in case of Frank? Take the input {5,5,5,5,5,10}, when Bob has taken {5,5} we count the combinations of it with respect to the other 5s, later in the input. But why aren't we doing such a thing for Frank? When Frank takes {5,5,5,10} we are not counting the combinations for him, for example for Sum = 10 Frank has 3 ways according to the algorithm. Why not consider combinations for him? – ng.newbie Jun 18 '18 at 9:05

@qwertyman The algorithm counts the number of equal items picked by Bob but not by Frank, why is that? – ng.newbie Jun 18 '18 at 11:13

@ng.newbie at any given point, we count the number of valid solutions where Bob's largest item is valued v[i], and he picks exactly *u* such items (and we do this for all possible values of v[i] and *u*). After this is fixed for Bob, we do not actually care how many items with value v[i] Frank receives -- we need to count all possibilities (would be valid but less efficient to also sum over all his choices). Now a possible question is "but why do we care how many such items Bob receives?" -- we need to fix this because otherwise we wouldn't know how many such items are available to Frank. – qwertyman Jun 18 '18 at 21:04

1

Here's some Java code for this that references the original solution. It also incorporates qwertyman's fantastic explanations (to the extent feasible). I've added some of my comments along the way.

```
import java.util.*;
public class Jewelry {
    int MAX_SUM=30005;
    int MAX_N=30;
    long[][] C;

    // Generate all possible sums
    // ret[i][sum] = number of ways to compute sum using the first i numbers from val[]
    public long[][] genDP(int[] val) {
        int i, sum, n=val.length;
        long[][] ret = new long[MAX_N+1][MAX_SUM];
        ret[0][0] = 1;
        for(i=0; i+1<=n; i++) {
            for(sum=0; sum<MAX_SUM; sum++) {
                // Carry over the sum from i to i+1 for each sum
                // Problem definition allows excluding numbers from calculating sums
                // So we are essentially excluding the last number for this calculation
                ret[i+1][sum] = ret[i][sum];

                // DP: (Number of ways to generate sum using i+1 numbers =
                //         Number of ways to generate sum-val[i] using i numbers)
                if(sum>=val[i])
                    ret[i+1][sum] += ret[i][sum-val[i]];
            }
        }
        return ret;
    }

    // C(n, r) - all possible combinations of choosing r numbers from n numbers
    // Leverage Pascal's polynomial co-efficients for an n-degree polynomial
    // Leverage Dynamic Programming to build this upfront
    public void nCr() {
        C = new long[MAX_N+1][MAX_N+1];
        int n, r;
        C[0][0] = 1;
        for(n=1; n<=MAX_N; n++) {
            C[n][0] = 1;
            for(r=1; r<=MAX_N; r++)
                C[n][r] = C[n-1][r-1] + C[n-1][r];
```

answered Jul 31 '19 at 19:40

S D
66 1 4