IOI Training 2017 - Week 3
Dynamic Programming

Vernon Gutierrez

March 2017

1 Introduction

There is already plenty of excellent material on the internet about dynamic programming, and I believe many of our NOI participants are already familiar with DP. So instead, in this tutorial, I will simply give you some tips on implementation, and on how to make it easier for you to get divine inspiration to come up with a DP solution. If you still don't know what the fuss is all about, check out the following videos from MIT to get you started.

DP is seriously one of the coolest things in algorithms, so you have to learn it. What makes it so cool is that with just a little bit of effort, it is suddenly possible to transform exponential time algorithms into polynomial time algorithms. It is also an interesting combination of being formulaic while requiring creativity.

What DP is all about can be summarized in the following equation:

$$DP = bruteforce + memoization \tag{1}$$

That's all there is to it. First you need to see a few classic DP problems before what's written below will make sense. So here you go. Come back after you've read items 1-14, 27, and 55.

Finished? Ok, good. Now we can get to business.

2 The Brute Force Part

2.1 The right way to think about recursion for DP

Maybe when you think about recursion, you think about function calls getting pushed and popped off the stack. That is a correct picture of how function calls work when executed, but it is not necessarily the best picture to have when trying to come up with recursive algorithms.

Especially for brute force and DP problems, the way I like to think about recursion is to imagine myself having a super-parallel computer. If I write a recurrence like `F(n-1) + F(n-2)`, what I'm really doing is making the universe branch off into two. In one universe, I compute F_{n-1}. In another universe, I compute F_{n-2}. And then I am doing both computations in parallel and adding up the results when the two computations are done. Another example. For the knapsack problem, if I write `max(v[i] + OPT(i-1, W - w[i]), OPT(i-1, W))`, what I'm really doing is creating two universes. One in which I chose the ith item, another in which I did not. And then those two universes evolve in parallel, and may continue to split. I'm sure that in one of those universes, after n steps, I have the optimum set. The way I obtain this optimum set in order to print it out is by killing off the other universes using the `max` function. Of course, in reality, the computer doesn't really become super-parallel, and I'm not really splitting universes. The computer executes each branch of my recurrence in sequence, but when trying to come up with the DP solution, I don't need to think about that. It is more fruitful for me to imagine that I have a more powerful computer that can do an infinite number of things in parallel, and the language of recursion allows me to play that trick.

2.2 Brute Force vs. DP

Many classic DP problems are easy, because all that is required is brute force. And brute force requires almost no thinking. Right?

Wrong. Equation (1) above is not 100% accurate. In reality, the brute force required for DP needs to be a little bit smart so that subproblems can overlap. For example, if in the Knapsack problem, we used the set of items already chosen to represent our subproblem (shown in the code below), then even if we memoized our solution, the worst case running time would be $O(n^2 2^n)$ rather than $O(nW)$. (There would be $O(n2^n)$ subproblems, and each subproblem would require $O(n)$ time to copy the set.)

```
int OPT(int i, set<int> &s) {
    if(i == 0) {
        if(sum_weights(s) <= W) {
            return sum_values(s);
        } else {
            return -1;
        }
    } else {
        set<int> new_set;
        new_set.insert(s.begin(), s.end());
        new_set.insert(i);
        return max(OPT(i-1, new_set), OPT(i-1, s));
    }
}
```

The key conflict in designing DP solutions is this. You need your subproblem to contain enough information to be able to take into account all the constraints. But you also don't want to put so much information that subproblems don't overlap enough to make the running time go down to polynomial. In the knapsack problem, if the subproblem only has one parameter i, it is not enough to correctly capture the fact that the knapsack capacity is limited. On the other hand, all we really need to capture is the capacity we have left in order to solve our subproblems. We don't need to care about exactly what items have already been chosen.

2.3 DP patterns

One way to more easily come up with the DP recurrence is to recognize that most DP problems fall into certain patterns. This excellent TopCoder recipe explains what some of those patterns are.

I earlier stated that DP is formulaic. The reason is, when designing a DP solution, you often just need to follow these steps:

1. Represent the state or subproblem
2. Figure out the state transitions or "moves"
3. Analyze the running time
4. Implement the brute force recurrence
5. Memoize

Steps 3-5 are often easy and straightforward, once you've gotten a fair bit of practice with DP problems. Step 1 is usually the hardest. Step 2 sort of follows once you have Step 1, but it still requires some thinking. In some cases, Step 2 needs to be improved by clever means to reduce the degree of the running time of your solution by 1. But we will talk about that some other time.

2.4 Figuring out the state

The most common states for DP are DP on prefixes, DP on substrings, and DP on two (or more) pointers.

DP on prefixes means, that a subproblem looks something like $OPT(i, ...)$, where we are considering the prefix of the input that spans from item 1 to item i. The answer to a subproblem depends on subproblems for a "prefix" of the given input. An example of a problem that fits this pattern is weighted interval (or job) scheduling. To solve weighted interval scheduling, we needed to consider either the prefix of intervals without the last interval, or the prefix of intervals without all of the intervals which overlap with the last interval.

DP on substrings means, that a subproblem looks something like $OPT(i, j, ...)$, where we are considering the substring of the input that spans from item i to item j. An example of a problem that fits this pattern is matrix chain multiplication. To solve matrix chain multiplication, we guess a splitting point k, and then recursively solve the substrings of matrices $OPT(i, k)$ and $OPT(k + 1, j)$.

DP on two (or more) pointers is like DP on prefixes, but this time, there are multiple strings, and not all of the pointers have to move backward to constitute a prefix. An example of a problem that fits this pattern is longest common subsequence.

2.5 Figuring out the transitions

DP is usually about choice. Choose a subset or combination that maximizes this. Or minimizes that. It is too difficult to think about choosing an entire subset or combination. So what we do in DP solutions is to break up the choice into *stages*. To choose a subset, I break it down into smaller choices of either choosing one item or not. Once you think about choice in DP in this way, as making step-by-step choices, it becomes much clearer what you need to do to come up with the recurrence.

There are only two basic transition patterns for DP: binary choice or multiway choice.

Binary choice means, for each state of your problem, you have two *mutually exclusive* options on how to proceed. Usually, this is because your subproblems represent subsets, and you are choosing on whether or not to include the last item. Examples of binary choice DP problems are knapsack problem and weighted interval scheduling.

Multiway choice means, for each state of your problem, you have multiple *mutually exclusive* options on how to proceed. An example of this is the coin change problem, where starting from some amount, you have multiple possible single moves or choices on how to reduce the amount by using just one coin.

2.6 Analyze the running time

Analyzing the running time of DP solutions is usually straightforward. Just multiply the the number of subproblems with the amount of time spent on each subproblem, counting recursive calls as free due to memoization. The number of subproblems is just the product of the maximum sizes of the parameters to your recurrence. This is usually easy to compute from the appearance of your recursive function. If your recursive function is $OPT(i)$, then the number of subproblems is $O(n)$. If the recursive function is $OPT(i, j)$, then it is something like $O(n^2)$. $OPT(i, j, k)$? $O(n^3)$. Then you may need to multiply with an extra $O(n)$ factor to get your final running if you did a for-loop inside the recursive function. For example, there are $O(n)$ states for coin change, where n is the required amount. If there are d denominations, then the recurrence would be a multiway choice which requires a loop over d items (or d if statements). This means the running time is $O(nd)$. Don't forget to do this step before implementing, or you might waste time implementing a TLE solution.

3 The Memoization Part

It helps to have a mental template for implementing DP problems. That way, during contest time, all you have to focus on is finding the recurrence, and the implementation follows automatically. The personal template that I follow is the following (feel free to deviate from this):

```cpp
#include <iostream>
#include <cstring>
#define N 1001 // 1 + whatever the maximum size of the first parameter is.
#define M 1001 // 1 + whatever the maximum size of the second parameter is.
// .. and more if there are more parameters

using namespace std;

int memo[N][M]; // Change the type to long long or another type if needed.
                // Add more dimensions if needed.

int OPT(int n, int m) {
    if(memo[n][m] == -1) {
        int ans;
        // solve the recurrence here and save the answer to ans
        memo[n][m] = ans;
    }
    return memo[n][m];
}

int main() {
    for(int t = 0; t < n_testcases; t++) {
        memset(memo, -1, sizeof memo);
        int x, y;
        // read inputs
        cout << OPT(x, y) << endl;
    }
    return 0;
}
```

For example, here's how I would write a recursive, memoized Fibonacci function:

```cpp
#include <iostream>
#include <cstring>
#define N 1001
using namespace std;

long long memo[N];

long long F(int n) {
    if(memo[n] == -1) {
        int ans;
        if(n == 0) ans = 0;
        else if(n == 1) ans = 1;
        else ans = F(n-1) + F(n-2);
        memo[n] = ans;
    }
    return memo[n];
}

int main() {
    memset(memo, -1, sizeof memo);
    cout << F(1000) << endl;
    return 0;
}
```

One common mistake is to create an array of size one too small than you need. For example, if we had defined N to be 1000 above, the program wouldn't have worked. Or worse, because of the way C++ behaves, it would've only probabilistically worked. Or it might consistently work on your computer but always fail to work when you submit to UVa or Codeforces. That is a debugging nightmare! To avoid this, always remember to make your memo the right size. Usually, it is one larger than the maximum expected input size. Some competitive programmers prefer to just add a lot of allowance to the memo size to avoid the pain, so they would instead define N to be 1010 above. I personally have not found enough reason to resort to such a tactic, but if you find yourself wasting time dealing with off-by-one errors too often, this tactic might be of benefit to you.

Warning: the `memset` function does not really set each individual element to some specified number. Instead, it sets each individual byte of memory spanned by the specified addresses to the specified byte. It is too time consuming and annoying to explain how it really works. The bottom line is that it happens to work for setting all elements to 0 or to -1, because of how binary numbers work. It does not in general work for any arbitrary value. For these cases, you should write a for loop to set the elements manually. You will rarely need to do so because -1 is the most common valid dummy value we use for DP problems. But if -1 happens to be a legitimate answer for one of your DP subproblems, then you need to use a different dummy value, and you also need to use a for-loop to initialize your memo with this other dummy value. There is a safer, more C++-ish alternative for 1D arrays, `fill_n`, but it only really works well with 1D arrays. If you want to understand what can go wrong with memset, try to compile and run the code below:

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
#define N 10
using namespace std;

int memo1[N];
int memo2[N][N];

int main() {
    memset(memo1, 2, sizeof memo1);
    for(int i = 0; i < N; i++)
        cout << memo1[i] << " ";
    cout << endl;

    fill_n(memo1, sizeof memo1, 2);
    for(int i = 0; i < N; i++)
        cout << memo1[i] << " ";
    cout << endl;

    memset(memo2, 2, sizeof memo2);
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            cout << memo2[i][j] << " ";
        cout << endl;

    fill_n(*memo2, sizeof memo2, 2);
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            cout << memo2[i][j] << " ";
        cout << endl;

    return 0;
}
```

You also should not use `memset` for non-integer arrays. Try changing the types of the arrays above to hold `double` values instead. What happens when you run it?

Here's a bottom-up implementation for Fibonacci:

```cpp
#include <iostream>
#define N 1001
using namespace std;

long long memo[N];

int main() {
    memo[0] = 0;
    memo[1] = 1;
    for(int n = 2; n < N; n++) {
        memo[n] = memo[n-1] + memo[n-2];
    }
    cout << memo[1000] << endl;
    return 0;
}
```

The advantage of writing the solution this way is that we don't have function calls, leading to a slightly faster program. We also never need to deal with stack overflow errors if the recursion becomes too deep. We also don't need to initialize the array to hold some dummy value. It's also now much clearer that the running time of memoized Fibonacci is $O(n)$. There's also a DP optimization which you will see in the future, which requires that the DP has already been written in a bottom-up style.

The disadvantage of writing the solution this way is that we needed to carefully think about the order in which to put elements in the memo, so that by the time we need answers to the smaller subproblems, they are already in the memo. This is something we didn't have to think about when doing the top-down version. This may not be an obvious disadvantage for Fibonacci, but for some problems, like matrix chain multiplication, thinking about the order is wasted contest time.

4 A Worked Example

Let's try applying all of the ideas above to solve an actual problem, Codeforces 118D - Caesar's Legions. Before you read the explanation below, I recommend trying to solve it yourself first.

4.1 Figuring out the state, attempt 1

In this problem, we need to form sequences of n_1 footmen and n_2 horsemen, subject to certain restrictions. What makes this problem interesting is that there are several variables here. There are many possible ways to form a DP state/subproblem, and it's not immediately clear what our states/subproblems should be.

One very useful problem strategy that applies to many kinds of problems, and not just DP, is to simplify the problem. There are too many things we have to do in this problem: form a sequence of footmen and horsemen, make sure not too many footmen are consecutive to each other, and make sure not too many horsemen are consecutive to each other. Maybe that's a little bit to difficult to think about all at once. So what we will do is first to consider an easier version of the problem. Let's say we just need to count the total number of sequences of n_1 footmen and n_2 horsemen, without the additional restrictions. How would we solve this problem? Think about it for a bit before moving on to the next paragraph.

A non-brute force solution would be to realize that all we are really choosing is where to put the n_1 footmen among the $n_1 + n_2$ soliders. This is $\binom{n_1+n_2}{n_1}$. We could have also chosen where to put the n_2 horsemen instead, giving us $\binom{n_1+n_2}{n_2}$, which happens to be equal to $\binom{n_1+n_2}{n_1}$.

But this is too smart! It's so smart, that it's now quite difficult to add into our holy, pristine formula the constraints we initially decided to ignore. Remember, $DP = bruteforce + memoization$. So we need to think brute force. Let's try to be dumber. Thinking about producing entire sequences at once is too hard. So, we will break down the choice for the entire sequence into a series of smaller choices or *stages*, and consider building sequences step-by-step.

This naturally leads us into thinking about doing DP on prefixes. A first attempt at the state might be $C(i)$, where i is the number of soldiers we have yet to pick, $0 \leq i \leq n_1 + n_2$.

4.2 Figuring out the transitions, attempt 1

The number of sequences we can form with 0 soldiers is just 1, the empty sequence. Now, for some position $i > 0$, we have two choices: we can either let a footman stand there, or a horseman. This naturally leads to the following recurrence:

$$C(i) = \begin{cases} 1 & i = 0 \\ C(i-1) + C(i-1) & i > 0 \end{cases} \tag{2}$$

But woops, $C(n_1+n_2)$ is $2^{n_1+n_2}$, and does not agree with the non-brute force solution. This tells us something is wrong. Indeed, we failed to consider the restriction that we only have n_1 footmen and n_2 horsemen.

4.3 Figuring out the state, attempt 2

In order to take this restriction into account, our state representation is not enough. We have to keep track of how many footmen and horsemen we have remaining at our disposal. The most natural way to do that is to add two new parameters to our state: f representing the number of footmen remaining, and h representing the number of horsemen remaining. This should be reminiscent of the knapsack problem, where we added an extra parameter to keep track of the remaining capacity. Note that instead of doing this, one other obvious way to change our state would be to add s, the sequence of footmen and horsemen we have made so far, and to determine f and h from there. But note that this is too much information! We already saw how this led to a bad solution for knapsack, so let's not make that mistake again. What really matters for a subproblem is only the number of footmen and horsemen remaining, not the entire sequence of footmen and horsemen we have chosen so far. Summarizing, our state would then be $C(i, f, h)$. Stop and think about what the recurrence relation should look like if we have this kind of state representation.

4.4 Figuring out the transitions, attempt 2

Like before, the number of sequences we can form with 0 soldiers is just 1. Again, for some position $i > 0$, we have two choices: we can either let a footman stand there, or a horseman. But we can only do either move if we still have footmen (or respectively, horsemen) remaining:

$$C(i,f,h) = \begin{cases} 1 & i = 0 \\ C(i-1,f-1,h) + C(i-1,f,h-1) & i > 0, f > 0, h > 0 \\ C(i-1,f-1,h) & i > 0, f > 0, h = 0 \\ C(i-1,f,h-1) & i > 0, f = 0, h > 0 \\ 0 & i > 0, f = 0, h = 0 \end{cases} \tag{3}$$

The last case can't really happen (why?), but we put it up there just as a sanity check. The answer we want is $C(n_1 + n_2, n_1, n_2)$. This now correctly computes the number of sequences, without the restrictions about footmen or horsemen standing consecutively. But because we've now set up a brute force solution, we're ready to *extend* it to handle the additional constraints.

4.5 Figuring out the state, attempt 3

Let's first figure out how to handle the restriction that at most k_1 footmen may stand consecutive to each other. The most natural way to do this is to again add a parameter to the state, k_f denoting the number of consecutive footmen we have, thus $C(i, f, h, k_f)$. Again, stop and think about what the recurrence should look like given this state representation.

4.6 Figuring out the transitions, attempt 3

How do the moves we have available change with this extra restriction? Every time we choose to add a footman to the sequence, we have to increase k_f. If k_f is already k_1, we can no longer add a footman, or we will have too many consecutive footmen. We're forced to add a horseman. Adding a horseman resets the number of consecutive footmen to 0:

$$C(i,f,h,k_f) = \begin{cases} 1 & i = 0 \\ Foot(i,f,h,k_f) + Horse(i,f,h,k_f) & i > 0 \end{cases} \tag{4}$$

where:

$$Foot(i,f,h,k_f) = \begin{cases} C(i-1,f-1,h,k_f+1) & f > 0, k_f < k_1 \\ 0 & otherwise \end{cases} \tag{5}$$

and:

$$Horse(i,f,h,k_f) = \begin{cases} C(i-1,f,h-1,0) & h > 0 \\ 0 & otherwise \end{cases} \tag{6}$$

The answer we want is $C(n_1+n_2,n_1,n_2,0)$.

We can actually write this a little bit better by letting k_f denote the number of footmen we can *still* add consecutively, rather than the number of footmen that we have *already* added. Adding a footman to the sequence decreases this number by one. If k_f is zero, then we know we have already added k_1 footmen consecutively, and we shouldn't add another one. On the other hand, adding a horseman resets k_f to k_1, since we are free to add up to k_1 consecutive horsemen again.

$$C(i,f,h,k_f) = \begin{cases} 1 & i = 0 \\ Foot(i,f,h,k_f) + Horse(i,f,h,k_f) & i > 0 \end{cases} \tag{7}$$

where:

$$Foot(i,f,h,k_f) = \begin{cases} C(i-1,f-1,h,k_f-1) & f > 0, k_f > 0 \\ 0 & otherwise \end{cases} \tag{8}$$

and:

$$Horse(i,f,h,k_f) = \begin{cases} C(i-1,f,h-1,k_1) & h > 0 \\ 0 & otherwise \end{cases} \tag{9}$$

This formulation is slightly better than the first one, because it more easily lends itself to a bottom-up implementation. The answer we want is $C(n_1+n_2,n_1,n_2,k_1)$.

4.7 Figuring out the state and transitions, attempt 4

After we've figured this out, adding in the restriction for the horsemen is now easy, since it's just symmetric to the footmen case. Our new state will be $C(i,f,h,k_f,k_h)$. Before reading the recurrence below, again, try to figure it out on your own first.

Done? Ok. Here's the recurrence:

$$C(i, f, h, k_f, k_h) = \begin{cases} 1 & i = 0 \\ Foot(i, f, h, k_f, k_h) + Horse(i, f, h, k_f, k_h) & i > 0 \end{cases} \tag{10}$$

where:

$$Foot(i, f, h, k_f, k_h) = \begin{cases} C(i-1, f-1, h, k_f - 1, k_2) & f > 0, k_f > 0 \\ 0 & otherwise \end{cases} \tag{11}$$

and:

$$Horse(i, f, h, k_f, k_h) = \begin{cases} C(i-1, f, h-1, k_1, k_h - 1) & h > 0, k_h > 0 \\ 0 & otherwise \end{cases} \tag{12}$$

The answer we need is $C(n_1 + n_2, n_1, n_2, k_1, k_2)$. Before reading the next section, try to figure out the running time of our algorithm, assuming we properly memoized it.

4.8 Analyzing the running time, attempt 4

The running time of our current solution is $O((n_1 + n_2)n_1 n_2 k_1 k_2)$. Depending on our implementation, this is a risky AC. It appears that we have too many parameters in our state here. Perhaps we can safely reduce them?

4.9 Attempt 5

Notice that we don't really need to keep track of how many soldiers we still need to put in line. That number can be easily deduced from two other parameters: the number of footmen and the number of horsemen. We can thus safely eliminate the first parameter from our DP state and get a faster solution. This technique works well for plenty of DP problems. This is one of the things you should try when your DP solution is too slow: try to eliminate parameters that are not really independent, but can be derived, from the others. Before reading the recurrence below, try to figure it out yourself first.

$$C(f, h, k_f, k_h) = \begin{cases} 1 & f + h = 0 \\ Foot(f, h, k_f, k_h) + Horse(f, h, k_f, k_h) & f + h > 0 \end{cases} \tag{13}$$

where:

$$Foot(f, h, k_f, k_h) = \begin{cases} C(f-1, h, k_f - 1, k_2) & f > 0, k_f > 0 \\ 0 & otherwise \end{cases} \tag{14}$$

and:

$$Horse(f, h, k_f, k_h) = \begin{cases} C(f, h-1, k_1, k_h - 1) & h > 0, k_h > 0 \\ 0 & otherwise \end{cases} \tag{15}$$

The answer we need is $C(n_1, n_2, k_1, k_2)$. The running time reduces to $O(n_1 n_2 k_1 k_2)$, and our solution is now a sure AC solution.

4.10 Implement the brute force recurrence

Before reading the code below, I invite you once again to try it out yourself first. Don't forget to take answers modulo 10^8.

Finished? Good. See how your implementation compares with mine:

```cpp
#include <iostream>
#define MOD 100000000
using namespace std;

int n1, n2, k1, k2;
long long C(int f, int h, int kf, int kh) {
    if(f + h == 0) return 1;
    else {
        long long ans = 0;
        if(f > 0 && kf > 0) ans = (ans + C(f - 1, h, kf - 1, k2)) % MOD;
        if(h > 0 && kh > 0) ans = (ans + C(f, h - 1, k1, kh - 1)) % MOD;
        return ans;
    }
}

int main() {
    cin >> n1 >> n2 >> k1 >> k2;
    cout << C(n1, n2, k1, k2) << endl;
    return 0;
}
```

4.11 Memoize

Finally! This is the easiest step. I'm sure you can do it on your own, before comparing with mine. I deviated from my template a little bit, just to make the code fit nicely on this page, but it is a good idea to use `#define`'s for the sizes, just so you can more quickly spot an off-by-one-error, a bug due misread input constraints, or whatever.

```cpp
#include <iostream>
#include <cstring>
#define MOD 100000000
using namespace std;

int n1, n2, k1, k2;
long long memo[101][101][11][11];

long long C(int f, int h, int kf, int kh) {
    if(memo[f][h][kf][kh] == -1) {
        long long ans;
        if(f + h == 0) ans = 1;
        else {
            ans = 0;
            if(f > 0 && kf > 0) ans = (ans + C(f - 1, h, kf - 1, k2)) % MOD;
            if(h > 0 && kh > 0) ans = (ans + C(f, h - 1, k1, kh - 1)) % MOD;
        }
        memo[f][h][kf][kh] = ans;
    }
    return memo[f][h][kf][kh];
}

int main() {
    memset(memo, -1, sizeof memo);
    cin >> n1 >> n2 >> k1 >> k2;
    cout << C(n1, n2, k1, k2) << endl;
    return 0;
}
```

It is not too hard to convert this into bottom-up style:

```
#include <iostream>
#include <cstring>
#define N1 101
#define N2 101
#define K1 11
#define K2 11
#define MOD 100000000
using namespace std;

int n1, n2, k1, k2;
long long C[N1][N2][K1][K2];

int main() {
    memset(C, 0, sizeof C);
    cin >> n1 >> n2 >> k1 >> k2;

    for(int f = 0; f <= n1; f++) {
        for(int h = 0; h <= n2; h++) {
            for(int kf = 0; kf <= k1; kf++) {
                for(int kh = 0; kh <= k2; kh++) {
                    long long ans;
                    if(f + h == 0) ans = 1;
                    else {
                        ans = 0;
                        if(f > 0 && kf > 0) ans = (ans + C[f - 1][h][kf - 1][k2]) % MOD;
                        if(h > 0 && kh > 0) ans = (ans + C[f][h - 1][k1][kh - 1]) % MOD;
                    }
                    C[f][h][kf][kh] = ans;
                }
            }
        }
    }
    cout << C[n1][n2][k1][k2] << endl;
    return 0;
}
```

4.12 Attempt 6?

This problem can be solved in $O(n_1 n_2 \max(k_1, k_2))$. I leave it as a challenge for you to figure out how.

5 Subtask 1

These are warmup problems. If you are having difficulty solving the subtask 2 problems, try these first. You are not required to submit your solutions to these.

Codeforces 313B - Ilya and Queries

UVa 10130 - SuperSale

UVa 10684 - The jackpot

UVa 357 - Let Me Count The Ways

UVa 108 - Maximum Sum

6 Subtask 2

These problems are required.

Codeforces 368B - Sereja and Suffixes

Codeforces 698A - Vacations

Codeforces 626B - Cards

Codeforces 706C - Hard Problem

Codeforces 711C - Coloring Trees

Codeforces 455A - Boredom

Codeforces 474D - Flowers

7 Subtask 3

Submitting 2 out of 4 of these exempts you from submitting the others. Warning: they are hard!

Codeforces 538E - Demiurges Play Again

Codeforces 427D - Match & Catch

Codeforces 677E - Vanya and Balloons

Codeforces 543C - Remembering Strings