IOI Training 2018 - Week 4
Dynamic Programming

Vernon Gutierrez

March 2018

1 Introduction

There is already plenty of excellent material on the internet about dynamic programming. If it's your first time encountering the concept, you should check out these four videos from MIT to help you get started. Then, check out this TopCoder tutorial and this TopCoder recipe for more examples. This tutorial will focus on how to come up with DP recurrences and implementation details.

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

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

That's all there is to it.

What makes DP so powerful is that with just a little bit of effort, exponential time algorithms are transformed into polynomial time algorithms. It is also an interesting combination of being formulaic (to memoize a recurrence) while requiring creativity (to figure out the recurrence in the first place).

Usually, DP is employed to solve one of the following types of problems:

1. **Optimization problems**: given some set of possible answers (called the **candidate solutions**), some conditions on what a valid answer can be (called the **constraints**), and some rule to determine how good an answer is (called the **objective function**), find the best among all the valid answers. A classic example of this type of problem is the knapsack problem. Another is Codeforces 2B - The least round way.

   Exercise 1.1 *Identify the set of candidate solutions, the constraints, and the objective function for the two given problems.*

2. **Feasibility problems**: Given a set of candidate solutions and some constraints, determine if there is a valid solution. Here's an example problem: given a rod of length n and an set S of permitted cutting lengths, is it possible to break the rod into pieces such that the length of each piece is in S? For example, if the rod is length 11 and $S = \{2, 3, 4\}$, then the answer is "yes," but the if the rod length is 9 and $S = \{2, 4, 6\}$, then the answer is "no." This is a slightly modified version of the classic rod cutting problem. Another good example is Codeforces 626B - Cards.

   Exercise 1.2 *Try to frame Codeforces 626B - Cards as one or more feasibility problems, as it's not immediately obvious how.*

3. **Counting problems**: Given a set of candidate solutions and some constraints, count the number of distinct valid solutions. An example of this type of problem is counting the number of binary strings with no consecutive 1's. Another is Codeforces 474D - Flowers.

To successfully apply DP, the problem at hand must have two conditions:

1. It must be solvable by some given recurrence.
2. The recurrence which solves the problem, when expanded all the way down to the base cases, must have significantly overlapping parts.

In CLRS and other popular algorithms books, the second condition is aptly called **overlapping subproblems**. The first condition is usually called **optimal substructure**, but I personally don't like this term, as DP can also be applied to counting problems.

2 How to Come Up with Recurrences

2.1 The Right Way to Think About Recursion

Often, the reason why people struggle with DP is not because they don't understand the concept of DP, which is really simple. The trouble seems to be that they have a broken idea of how recursion works. If you're having trouble coming up with recurrences, this section might help. Otherwise, skip to the next section.

Maybe when you think about recursion, you think about function calls getting pushed and popped off the stack. Or you might imagine a tree of recursive calls expanding from the initial call all the way to the base cases. Those are correct pictures of how function calls work when executed, but they don't necessarily help you come up with recursive algorithms. It's maybe useful to see how a recursive function actually gets executed a few times in your life, just so you can more easily believe that it actually works, but past a certain age, you have to stop looking at recursion from the point of view of the *computer* and to start seeing it from the point of view of a *programmer*[1].

So, how does one come up with a recursive solution in general? There are three key steps:

1. Imagine how to reduce your problem into a problem that is *one step* smaller.
2. *Believe* that you already know how to get the answer for any problem which is smaller than your original problem, that your function can already compute the answer for some smaller problem, *even before you've finished defining it.*
3. Assuming that the function already works for the smaller problem, supply the logic for computing the answer to the original problem, using the answer to the smaller problem.

It may seem magical at first that you can simply *believe* that your function already works while you are still writing it. But, if you understand the principle of mathematical induction[2], then this won't feel that weird.

This excellent article reiterates these points and gives a nice example. In more complicated scenarios, there may not be only one sub-problem to your recurrence, but many, but the key thing to remember is that each of these sub-problems must be *one step* smaller. "One step" here does not necessarily just mean that the size of the input to the sub-problem is one smaller than the original problem. See, for example, the minimum coin change problem[3]. The important thing is that you can conceivably transform the answer to the smaller sub-problem into an answer for the original problem in *one step*. It may not even be meaningful to talk about the "size" of a sub-problem, as in the shortest paths problem.

Very often, if you have recurrences for optimization or counting problems, the recurrence is really capturing an exhaustive process of generating all possible candidate solutions, and either taking the minimum or maximum of them for optimization problems, or counting them via taking sums for counting problems. In these cases, I find it helpful to imagine that I have a super-parallel computer that can split into however many branches of computation as I want. This picture is helpful, because it allows me to just guess how to form the optimal solution for the original problem if I can, in parallel, solve for the optimal solution for all possible smaller sub-problems that are one step away from the original problem.

For example, take the minimum coin change problem. Let's say my goal is to produce the best way to make change for 30, and my denominations are 1, 10, and 25. I don't know what's the best way to produce 30, but I know that any solution must have a first coin. I don't know in advance what the first coin might be, so I try all possible first coins *in parallel*. I imagine the universe splitting into three. In the first branch of the universe, I started making change for 30 by taking a 1 coin first. In the second branch, I took a 10 coin as my first move instead. In the third branch, I took a 25 coin instead. In each of these initial branches, I need to make some *smaller* amount of change left. By the power of recursion, I believe that I can obtain the best way to make change for each of these remaining smaller amounts. I also know that the first coin needed for the best way to make change *has to be* one of 1, 10, 25, so to ensure I have the correct answer, I just try all of the possible choices and get the best one. There can be no other way because I've

[1] How do you know if you've successfully made this change of perspective? If you still debug your programs by simulating how your code runs, tweaking it a little bit if it doesn't run as expected, and repeating this several hundred times, then you're doing it wrong. This style of programming is akin to hill climbing. Not only is this a hopelessly inefficient way to debug recursive programs, but also doesn't work if in the first place you don't even have a clue what an almost correct solution which you hope to evolve into a correct solution looks like.

[2] If you're not comfortable with mathematical induction, then I recommend checking out this video from Khan Academy and these videos from MIT.

[3] You might want to watch this with subtitles on.

exhausted all the possibilities. I can express this as the following recurrence[4]:

$$C(i) = \min(C(i-1)+1, C(i-10)+1, C(i-25)+1) \tag{2}$$

If this seems hard to swallow, you can try to imagine the universe continuing to split off into three[5] branches every time you need to make a decision on what coin to choose next. Take a moment to convince yourself that this process does in fact generate all possible ways of producing 30 from 1, 10, and 25 coins. In all the universes, I did different things to make 30. In some of those universes, I may have made stupid decisions which led to a sub-optimal solution. But I am sure that in one of those universes, I did the right thing. I kill off the sub-optimal universes with the `min` function[6].

Of course, in reality, my computer doesn't really become super-parallel, and I'm not really splitting universes[7]. The computer executes each branch of my recurrence in sequence, but when trying to come up with the recurrence in the first place, 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.

For counting problems, the idea is similar. There are several ways to generate a combinatorial object. If I can break that generation process down into a series of stages, where at each stage I can make one of many *mutually exclusive* choices, I just try all those choices. If I know how to count the number of ways to generate a smaller version of the combinatorial object in question, by the Rule of Sum, the total number of ways to generate the original version of the combinatorial object is just the sum of these. In many cases, this is as simple as solving the optimization version of the problem first, and then converting the *min* or *max* function in the recurrence into a sum[8]. I know I'm not under-counting because I've exhaustively considered all the mutually exclusive choices. However, to avoid double-counting, we have to be careful to ensure that the choices we make are in fact mutually exclusive[9].

For example, let's consider the counting version of the coin change problem, where we now need to count how many ways there are to make change for some amount of money. Let's use the same amount and denominations as above. We might naively reason as follows.

1. I need the number of ways to produce 30.
2. To produce 30, I can take one of three initial choices to generate a particular combination: add 1 to my combination, and generate the rest recursively; add 10 and generate the rest recursively; or add 25 and generate the rest recursively.
3. All of these are mutually exclusive choices, so by the Rule of Sum, the total number of ways is therefore[10]

$$C(i) = C(i-1) + C(i-10) + C(i-25) \tag{3}$$

But this is actually wrong! The problem here is assuming that the choices are mutually exclusive when they are not. The sequences $25, 1, 1, 1, 1$ and $1, 25, 1, 1, 1$ for example, are each generated and counted separately, even though they are technically the same combination.

Exercise 2.1 *Come up with the correct recurrence for UVa 357 - Let Me Count The Ways.*

Exercise 2.2 *Why didn't we have this problem for the optimization version of the problem?*

I leave it to you to figure out how to apply the same ideas for feasibility problems. It shouldn't be that hard once you understand how to do it for optimization and counting problems[11].

2.2 Pure Brute Force Recurrences vs. DP-Able Recurrences

DP is easy, because basically all that is required is brute force. And brute force requires almost no thinking. Right?

[4] I've intentionally left out some details on what happens when $i < 25$, as they can be safely ignored for the current discussion. If you haven't seen this before, it should be easy to work out the $i < 25$, $i < 10$, and $i = 0$ cases on your own.

[5] Or fewer with smaller amounts of money

[6] Wouldn't it be nice if real life could operate this way?

[7] Or maybe I am, but I'm not going down that rabbit hole now.

[8] The idea of having a general algorithmic design technique where you can plug in many different kinds of binary operations so that the same technique solves whole, different categories of problems is somewhat pervasive in computer science and in competitive programming, and you will encounter this idea again in future weeks.

[9] Or, apply the Principle of Inclusion-Exclusion

[10] Again, ignoring the cases when $i < 25$

[11] Again, the key here is to just plug-in a different binary operation into the same framework.

Wrong. DP recurrences need to be a little bit smart and carefully designed so that subproblems can overlap.

For example, if in the knapsack problem[12], we include the actual set of chosen items as a parameter to our subproblem (shown in the code below), then even if we memoized our solution, the worst case running time would be $O(n^2 2^n)$[13] rather than $O(nW)$, which is bad if we have a lot of items but the weights are small[14].

```
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> set_copy(s.begin(), s.end());
        set_copy.insert(i);
        return max(OPT(i-1, set_copy), OPT(i-1, s));
    }
}
```

There is a conflict you face when designing DP recurrences. You need your subproblem to contain enough parameters to be able to take into account all the constraints. But you also don't want to put so much 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 remaining capacity. We don't need to care about exactly what items are chosen.

2.3 Common DP Patterns

One way to more easily come up with the DP recurrence is to recognize that most of them fall into certain patterns. There are some common types of sets of candidate solutions, and there are somewhat standard ways of generating them. This TopCoder recipe explains some of them. Here's a more complete list (minus DP on trees and graphs, which will be covered in a future week).

2.3.1 Explicit state

This type of DP state is often the easiest to figure out. You can almost literally think subproblem = state of the game.

The best example of this is Minimum Path in a Grid, where the goal of your game is to get from the upper-left corner of the grid to the lower-right corner of the grid, the state of your game is your position in the grid, and the transitions are the moves you can make, already stated in the problem statement. Another good example of this is Codeforces 626B - Cards, where the state is literally the state of the deck of cards, and again, the transitions you can make are already given in the problem statement.

2.3.2 Implicit state, but explicit choices

This is a little bit less easier than *explicit state*, but still quite easy. Now, there might be no obvious game that you can easily gleam from the problem statement. You must use a little bit more imagination to see the problem as a game. Typically, this means that you will be doing either *DP on prefixes* or *DP on intervals* (explained below). However, it's still quite obvious what your moves should be.

[12] Here are some additional good resources on the knapsack problem from MIT and Stanford. Watch them in exactly this order for humorous effect.

[13] There would be $O(n2^n)$ subproblems, and each subproblem would require $O(n)$ time to copy the set.

[14] Theoretically, $O(nW)$ is not considered polynomial time but only pseudo-polynomial. The running time can still be bad if the weights are large integers. The distinction is explained a little bit in the MIT video for those who are curious.

A classic example is the minimum coin change problem, where with some imagination, you can see the problem as a game of eliminating some amount money in the fewest number of moves. The state of the game would be the amount of money left. It's easy to see that the moves in this game would be: removing the amount of the first denomination, removing the amount of the second denomination, etc. Another nice example is Codeforces 698A - Vacations, where the moves are literally the actions that Vasya can do on a given day.

2.3.3 DP on prefixes

Another common type of problem has an array of numbers as input, and each subproblem typically represents a *prefix* of that input. Very commonly, the recursive function is of the type $OPT(i, ...)$, where i represents the "position" in the array, or perhaps more correctly, that we are only considering the first i items of the input array.

A classic example would be the longest increasing subsequence problem (LIS).

When the input to a problem is an integer, as in the case of coin change problem or in the integer partitions problem, you can sort of view the integer as an array of ones, and "moving" from a larger integer to a smaller integer is equivalent to "moving" from one position of the array to another. Arguably the term "DP on prefixes" can apply here as well, but it's a stretch. Think about it this way if it helps you, and don't if it doesn't.

2.3.4 Subsets

This is a special type of DP on prefixes, where you want to brute-force over all possible subsets of a given input set. Sets and subsets don't have a notion of order, but to be able to generate subsets systematically, it helps to impose an order on the input set and turn it into a list. Typically, the input is already given as an array anyway so this is almost trivial. To generate subsets, consider the items from right to left (or left to right) and consider the choice of including each element "independently" from the others. $OPT(i, \dots)$ will depend only on two branches of $OPT(i-1, \dots)$, where the first branch represents that the ith item is included in the subset, and the other branch represents that the ith item is excluded from the subset.

The canonical example is the 0-1 knapsack problem. A slightly less obvious example is Codeforces 455A - Boredom. Here, depending on your approach, the subproblem $OPT(i, \dots)$ may not literally just depend on $OPT(i-1, \dots)$, but thinking about the binary choice of including or excluding an item in the optimal set of chosen items will help you figure out the correct recurrence.

2.3.5 Multisets

This is a direct extension of the above, where now, you are allowed to take more than one instance of a particular type of item in the given input set. For example, consider Codeforces 106C - Buns.

One way to "brute force" over all possible multisets is to simply apply the same strategy as we did for subsets, but this time, instead of having two branches per subproblem representing the binary choice of taking an item or not, we now have multiple branches where the jth branch (counting from 0) represents that we take j instances of the current item. This would look something like

$$OPT(i, \dots) = \bigoplus_j OPT(i-1, \dots) + c(i, j) \tag{4}$$

where $\oplus$ is min, max, or $\sum$ depending on our problem, and $c(i, j)$ represents the cost or reward of taking j instances of item i[15].

This is easy to do with a loop inside the recursive function definition. However, if in the worst case, the maximum number of times an item can be taken is k, then this straightforward approach incurs an extra (typically linear) factor of k in the time complexity.

To avoid this extra factor, we can convert the *multiway* choice of taking between 0 to k items into a binary choice of either moving on to the next item without taking the current item, or taking the current item *without moving on to the next item*. The recurrence would then look something like

$$OPT(i, \dots) = \oplus(OPT(i-1, \dots), OPT(i, \dots) + c(i)) \tag{5}$$

[15] Note that $c(i, j) = 0$ for counting problems.

where again $\oplus$ is min, max, or $\sum$ depending on our problem, and $c(i)$ represents the cost or reward of taking a single instance of item i[16].

Take a moment to convince yourself that this does the same thing as the earlier recurrence, except it is much more efficient when memoized[17]. Notice that i is not decreased in the second branch of the recurrence, so to avoid infinite recursion, ensure that there are other parameters capturing the fact that item i was chosen and that in fact, the second branch is a smaller subproblem.

If this feels too abstract and you are confused by now, I recommend trying out Codeforces 106C - Buns. Then, see if you can find a solution for Codeforces 543A - Writing Code that runs in time (forget about space complexity first).

2.3.6 Subsequences with constraints on consecutive elements of the subsequence

This is almost like subsets, but since the ability to choose a new element of the array is dependent on the most recent element we picked, we need to slightly modify our function.

Take for example LIS. One way to deal with the constraint in LIS is to use a different strategy for generating subsets. Instead of going through every item of the input sequence and consider whether or not to include each item, we decide which of the n items be the last element of the subsequence and "jump" to that position in the array, decide which of the remaining items be the second to the last element and again "jump" to that position in the array, and so on. The index in our subproblem denotes both that we are considering the prefix of the input sequence covering the first i items, and that we choose to include item i most recently. This is what is done in the standard solution for LIS.

Another way is to add another parameter to our state, representing the most recent item chosen, while still proceeding with binary choice, knapsack-style transitions.

Both incur an extra linear factor in time: the first method due to the extra loop we need for "brute-forcing" over all possible next elements of our subsequence, the second method due to the additional state parameter. The second method also incurs and extra linear factor in space. The first method is usually better because of the space advantage[18], and because the for-loop is often much easier to optimize away than the extra parameter. See, for example, the $O(n \lg n)$ algorithm for LIS. However, the idea of adding an extra parameter representing the most recent move is quite useful for certain types of problems, like UVa 357 - Let Me Count The Ways.

2.3.7 Partitions

Another common type of problem is to figure out the best way to partition a list according to some constraints. Again, we try DP on prefixes. To partition some prefix, we "brute-force" over all possible ways of cutting out some suffix of our current prefix to make the last part of our partition. The recurrence typically looks like

$$OPT(i, \dots) = \bigoplus_{j<i} OPT(j, \dots) + c(j+1, i) \tag{8}$$

where $\oplus$ is min, max, or $\sum$ depending on our problem, and $c(x, y)$ represents the cost or reward of putting into a single partition all elements of the subarray of the input between indices x and y[19].

[16] Again, $c(i) = 0$ for counting problems.

[17] Note however, that this only works if the cost of taking a single instance of an item can be determined independently from the cost of taking more instances. That is, if $c(i, j)$ is something like $c(i, j) = j \times c(i)$ then it works, but if is something weirder like $c(i, j) = c(i)^j$ then this doesn't work. Another requirement is that there should be no local constraints on the number of instances of each item, only global constraints. That is, we can have

$$OPT(i, W, \dots) = \bigoplus_{j} OPT(i-1, W - w(i, j), \dots) + c(i, j) \tag{6}$$

where $w(i, j)$ represents the effect of taking j instances of item i on the global constraint W, but we cannot have

$$OPT(i, \dots) = \bigoplus_{j=0}^{k(i)} OPT(i-1, \dots) + c(i, j) \tag{7}$$

where $k(i)$ is the maximum number of instances of item i which we can take.

[18] Note that it is possible to avoid needing the extra space for the second method, but it is trickier to achieve.

[19] As usual, there are no associated costs for counting problems.

A classic example is text justification[20]. A nice Codeforces problem where this idea can be used is Codeforces 711C - Coloring Trees.

2.3.8 Subarrays

Here, we want to brute force over all possible subarrays, but typically the problem input size is too large for even quadratic time solutions. Instead, we to try compute the best subarray ending at every possible position in linear time. For this to work nicely, the best subarray ending at i must depend in some clever way only on the best subarray ending at $i-1$[21].

Classic example would be the maximum subarray problem[22]. A nice Codeforces problem where you can use this idea is Codeforces 814C - An impassioned circulation of affection. Thinking about the best subsegment ending at i and considering whether or not to extend the subsegment ending at $i-1$ to form the subsegment ending at i will help you solve this problem.

2.3.9 DP on multiple prefixes

This is really not that much different from DP on a single prefix. We're given more input sequences now, but the subproblems are still prefixes of the original sequences, and that each prefix may grow or shrink independently of the others.

The classic example is longest common subsequence problem.

2.3.10 DP on intervals

This deserves a whole category on its own, because the thought process involved in figuring out these kinds of recurrences is somewhat different than DP on prefixes. For DP on prefixes, the DP subproblem usually represents that we are trying to solve a prefix of the input. Here, we are instead solving a contiguous *substring* or *interval* of the input. There are two broad patterns for defining the transitions for interval-type states.

First, the answer for an interval can depend on shrunk versions of the original interval, shrunk by some constant amount from either or both ends of the interval.

This is very common for palindrome-type problems, such as in the longest palindromic subsequence problem.

The other way DP on intervals can work is as follows. To get the optimal answer for an interval spanning from i to j, we guess a splitting point $i \le k < j$, recursively solve the intervals from i to k and from $k+1$ to j, combine the answers from the two parts, and finally apply min, max, or $\sum$ across all guesses.

An example of this would be matrix chain multiplication. Another nice, classic problem that requires this idea is optimal binary search tree. This pattern should remind you of divide-and-conquer. The difference here is that we try *all* possible splitting points and solve the current subproblem by considering all possible bipartitions of the current interval, while in divide-and-conquer, we already have a single, fixed splitting point.

[20] Here, the strategy that Erik Demaine follows is DP on *suffixes*, which is basically the same thing as DP on prefixes, only left and right are reversed. You may sometimes find this more intuitive because you are moving "forward" through the array. You can try figuring out an initial recurrence this way if it feels easier. However, you should learn how to "think in reverse" and write recurrences that depend on literally *smaller* subproblems. I.e., $OPT(i, \dots)$ depends on some $OPT(i-k, \dots)$ subproblems rather than depending on some $OPT(i+k, \dots)$. Usually, this makes the math nicer. It will also make it easier for you to apply DP optimizations.

[21] Or, some constant number of earlier subarrays

[22] Arguably, this is not really a dynamic programming algorithm because it has some greedy component. See this answer on StackOverflow. But ehh.

3 Implementing DP Solutions in C++

The easiest way to implement a DP solution is by almost-mechanically converting the recurrence into a recursive function, and then almost-mechanically applying memoization to the function. It is important to practice this several times, until you can do these steps by habit. That way, during contest time, all you have to focus on is finding the recurrence, and the implementation follows automatically.

The most straightforward way to do this is to have a `map` mapping from `input_type` to `output_type`, where `input_type` is a `tuple` or `struct`[23] representing the aggregate of all the input types of the recursive function and `output_type` is the same as the return type of the recursive function.

For example, if this were my original recursive function:

```
#include <bits/stdc++.h>
using namespace std;

int OPT(int n, int m) {
    int ans;
    // solve the recurrence here and save the answer to ans
    return ans;
}
```

I can memoize it this way:

```
#include <bits/stdc++.h>
using namespace std;

typedef tuple<int, int> input_type;
map<input_type, int> memo;

int OPT(int n, int m) {
    input_type params = {n, m};
    if(!memo.count(params)) {
        int ans;
        // solve the recurrence here and save the answer to ans
        memo[params] = ans;
    }
    return memo[params];
}
```

Of course, I could've used a `pair` here instead, but `tuple` is generalizable to any number of input parameters.

This approach, however, incurs an extra logarithmic factor for accessing the memo. Usually, the inputs to DP recurrences are integer types. In this case, you can use a (multi-dimensional) array instead to implement the memo[24], where each dimension of the array corresponds to a parameter of the recursive function. To determine whether or not the answer to a subproblem is already in the array, we can use a separate Boolean array.

[23] Using a `tuple` here is usually easier, as you won't have to define custom comparators for the key type.

[24] If the range of possible inputs is large but the actual number of subproblems are small, your DP solution might pass the time limit but fail the memory limit. In this case, you have to determine if you can afford the extra log factor incurred by using a `map`. If not, you should define a hash function for your input parameters and make your memo map from the hash of the `input_type` to the `output_type` instead. If you hash to a reasonable range of integers, then you should still be able to use an array as your memo. There is usually no advantage to using an `unordered_map` instead, as you would have to define a custom hash function anyway, and the constant factor incurred by `unordered_map` are typically comparable to the log factor incurred by `map`.

```cpp
#include <bits/stdc++.h>
#define N 1000 // set this to the maximum expected input size
#define M 1000
using namespace std;

int memo[N+1][M+1];
bool in_memo[N+1][M+1];

int OPT(int n, int m) {
    if(!in_memo[n][m]) {
        int ans;
        // solve the recurrence here and save the answer to ans
        memo[n][m] = ans;
        in_memo[n][m] = true;
    }
    return memo[n][m];
}
int main() {
    memset(in_memo, false, sizeof in_memo); // initially, all subproblems are not yet in the memo
    int n, m;
    cin >> n >> m;
    cout << OPT(n, m) << endl;
    return 0;
}
```

In many (but certainly not all) cases, the answers to our subproblems are limited to certain ranges only, like non-negative integers. So we can get rid of the Boolean array and populate the memo with a sentinel value instead to signal that the answer to a subproblem is not yet in the array.

```cpp
#include <bits/stdc++.h>
#define N 1000 // set this to the maximum expected input size
#define M 1000
using namespace std;

int memo[N+1][M+1];

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() {
    memset(memo, -1, sizeof memo); // initially, all subproblems are not yet in the memo
    int n, m;
    cin >> n >> m;
    cout << OPT(n, m) << endl;
    return 0;
}
```

For example, here's how I would write a program that solves the minimum coin change problem, with a recursive function.

```cpp
#include <bits/stdc++.h>
#define INF 1'000'000'000
using namespace std;

vector<int> denominations;

int C(int n) {
    int ans;
    if(n == 0) {
        ans = 0;
    } else {
        ans = INF;
        for(int d : denominations)
            if(n - d >= 0)
                ans = min(ans, C(n - d) + 1);
    }
    return ans;
}

int main() {
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    for(int i = 0; i < number_of_denominations; i++) {
        int denomination; cin >> denomination;
        denominations.push_back(denomination);
    }
    cout << C(amount) << endl;
    return 0;
}
```

And here's the memoized version.

```cpp
#include <bits/stdc++.h>
#define N 1000
#define INF 1'000'000'000
using namespace std;

int memo[N+1];
vector<int> denominations;

int C(int n) {
    if(memo[n] == -1) {
        int ans;
        if(n == 0) {
            ans = 0;
        } else {
            ans = INF;
            for(int d : denominations)
                if(n - d >= 0)
                    ans = min(ans, C(n - d) + 1);
        }
        memo[n] = ans;
    }
    return memo[n];
}

```

```
int main() {
    memset(memo, -1, sizeof memo);
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    for(int i = 0; i < number_of_denominations; i++) {
        int denomination; cin >> denomination;
        denominations.push_back(denomination);
    }
    cout << C(amount) << endl;
    return 0;
}
```

One common mistake is to create an array of size one too small than you need. For example, if we forgot the `+1` in the definition of `memo` above, and the maximum input amount were 1000, 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 happens to work for setting all elements to 0 or to -1, because of how these numbers are represented in binary. It does not in general work for any arbitrary value[25]. 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 sentinel value we use. But if -1 happens to be a legitimate answer for one of your subproblems, then you need to use a different sentinel value, and you also need to use a for-loop to initialize your memo[26]. There are safer, more C++-ish alternatives for 1D arrays, namely `fill` and `fill_n`, but they only really work well with 1D arrays.

Exercise 3.1 *If you want to understand what can go wrong with `memset`, try to compile and run the code below.*

```
#include <bits/stdc++.h>
#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;

```

[25] See this for more explanation.
[26] Or, just use a separate Boolean array for marking if a subproblem has been solved or not.

```
    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.

Exercise 3.2 *Try changing the types of the arrays above to hold* `double` *values instead. What happens when you run it?*

The other way to implement DP is to do it bottom-up. Here's a bottom-up implementation for coin change:

```
#include <bits/stdc++.h>
#define N 1000
#define INF 1'000'000'000
using namespace std;

int C[N+1];
vector<int> denominations;

int main() {
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    for(int i = 0; i < number_of_denominations; i++) {
        int denomination; cin >> denomination;
        denominations.push_back(denomination);
    }
    for(int n = 0; n <= N; n++) {
        int ans;
        if(n == 0) {
            ans = 0;
        } else {
            ans = INF;
            for(int d : denominations)
                if(n - d >= 0)
                    ans = min(ans, C[n - d] + 1);
        }
        C[n] = ans;
    }
    cout << C[amount] << endl;
    return 0;
}
```

See how the only real difference between top-down and bottom-up style is the almost-mechanical way of converting a given recurrence into code. The logic of how answers to larger subproblems are obtained from smaller subproblems remains the same.

Sometimes, you may not want to literally mechanically convert the recurrence into code, but move the base cases to the top of the for-loop, and massage some parts a little bit for convenience and to make it look nice[27]. However, for some more complicated recurrences, especially those with multidimensional states, moving the base cases to the top of the for-loop can make the boundary conditions of the for-loop difficult to code bug-free. Exercise discretion on when to use this style and when not to.

[27] Making code look nice should not be a priority in time-pressured contest environments, but you should learn to do it by habit. You can save some time debugging when reading beautiful code over reading ugly code, even if you are reading only your own code.

```cpp
#include <bits/stdc++.h>
#define N 1000
#define INF 1'000'000'000
using namespace std;

int C[N+1];
vector<int> denominations;

int main() {
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    for(int i = 0; i < number_of_denominations; i++) {
        int denomination; cin >> denomination;
        denominations.push_back(denomination);
    }
    C[0] = 0;
    for(int n = 1; n <= N; n++) {
        C[n] = INF;
        for(int d : denominations)
            if(n - d >= 0)
                C[n] = min(C[n], C[n - d] + 1);
    }
    cout << C[amount] << endl;
    return 0;
}
```

The advantage of bottom-up DP 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[28]. We also don't need to initialize the array to hold some dummy value. It's also now much clearer what the running time of our solution is.

The disadvantage of bottom-up DP 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 coin change, but for some problems, like matrix chain multiplication, thinking about the order is wasted contest time. In practice, the running time speedup from writing a DP solution bottom-up instead of top-down does not matter for well-written problems (and should not matter for the IOI).

There are several DP optimization techniques which require that the DP has already been written in a bottom-up style though, so you need to learn how to implement DP bottom-up.

There is a third way of implementing DP solutions, which I will call *Bellman-Ford style*.

```cpp
#include <bits/stdc++.h>
#define N 1000
#define INF 1'000'000'000
using namespace std;

int C[N+1];
vector<int> denominations;

int main() {
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    for(int i = 0; i < number_of_denominations; i++) {
        int denomination; cin >> denomination;
        denominations.push_back(denomination);
    }
```

[28] In modern programming contest environments, this is less and less of an issue.

```
    fill(begin(C), end(C), INF);
    C[0] = 0;
    for(int n = 0; n <= N; n++) {
        for(int d : denominations)
            if(n + d <= N)
                C[n + d] = min(C[n + d], C[n] + 1);
    }
    cout << C[amount] << endl;
    return 0;
}
```

I've personally never found any practical use for it, but it does illuminate how the standard textbook version of the Bellman-Ford shortest paths algorithm can be derived from first principles and dynamic programming.

```
#include <bits/stdc++.h>
#define N 1000
#define INF 1'000'000'000
using namespace std;

int dist[N+1];
vector<int> denominations;

void relax(int u, int v, int w) {
    dist[v] = min(dist[v], dist[u] + w);
}

int main() {
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    for(int i = 0; i < number_of_denominations; i++) {
        int denomination; cin >> denomination;
        denominations.push_back(denomination);
    }
    fill(begin(dist), end(dist), INF);
    dist[0] = 0;
    for(int n = 0; n <= N; n++) {
        for(int d : denominations)
            if(n + d <= N)
                relax(n, n + d, 1);
    }
    cout << dist[amount] << endl;
    return 0;
}
```

---

4 A Worked Example

In order to keep this document short for return trainees, I encourage the new trainees to check out the worked example of how to solve Codeforces 118D - Caesar's Legions from section 4 of last year's training material.

5 Various Tips

1. Don't forget to analyze the running time of your solution before implementing it, or you might waste time implementing a TLE solution. Just because you used DP it doesn't mean it's automatically fast.

2. You don't necessarily have to come up with a brand-new recurrence every time you solve a new problem. Sometimes, all you need to do is to tweak the recurrence that solves a classic problem. Or, you might use a classic problem as a subroutine to a larger problem. A nice example of where you can apply this strategy is for Codeforces 446A - DZY loves sequences. It's a nice variant of the somewhat well-known longest bitonic sequence problem.

3. When faced with a problem with lots of constraints, one good strategy is to ignore several constraints first, solve a more basic version of the problem, and then progressively tweak your recurrence towards the correct solution by adding in the constraints one at a time. The worked example of Codeforces 118D - Caesar's Legions from section 4 of last year's training material shows this in more detail.

4. Unless the greedy solution is very easy to implement, faced with a problem where both greedy and DP seem applicable, I personally go with DP by default. Compared to greedy it doesn't require too much thinking, and I don't have to spend time and mental energy proving or disproving a greedy solution. It is tempting to do a proof by AC[29] strategy, because in the IOI there are no time penalties for wrong attempts, and you are allowed a lot of wrong submissions. However, you can get stuck trying to get *some* greedy solution to work when in fact the problem cannot be solved by any greedy solution at all. Even if the problem does admit a greedy solution, figuring out the correct greedy solution or just trying out several of them is wasted time if you practiced DP enough that you can quickly implement a DP solution anyway. This advice is all the more important for Codeforces rounds and other contests where the time penalties can be heavy. Take care to ensure that the DP solution actually fits the time limit though.

5. Sometimes, the most obvious recurrence and implementation doesn't work in time or space. It doesn't automatically mean that DP is not a viable solution to your problem. There are techniques for cutting down the time and memory complexity of DP solutions by linear or sometimes even quadratic factors. I've shown one technique for DP on multisets above. A bunch of other techniques are explained in this TopCoder recipe, and you will encounter more advanced techniques in the future.

6. Most video lectures and books leave out how to reconstruct the actual optimal solution after figuring out the optimal value, and I also left it out from this tutorial. The challenging part of an optimization problem is usually just figuring out the recurrence which produces the optimal value, and constructing the actual optimal solution follows easily from there. The solution reconstruction is also somewhat "mechanical" in nature, so most modern programming contest (including Codeforces and OI) problems do away with these and just focus on the real essence of the problem, which is already captured by just asking for the optimal value. This TopCoder recipe explains how to do the solution reconstruction, and it's nice to practice how to do this a few times for those rare instances where you might need it.

6 Problems

Instructions: Earn points by solving subtask 3, 4, and 5 problems. The maximum number of points is 1000. You may solve as many as you want but only 1000 points are required. Just submit directly to the specific online judges. For the Project Euler problems, in addition to submitting the correct answer to Project Euler, submit to Google Classroom a short explanation of how to solve the problems (including recurrences used), in any format[30]. If you have not completed the prior homework which asked for your online judge handles, make sure to do that immediately, or your submissions will not be counted. Reading editorials, discussions, etc. for the required problems before the end of the week is not allowed. All code you submit must be written by yourself.

6.1 Subtask 1

This section is only for those who are learning DP for the first time. If this does not apply to you, you may safely skip this section. You are not required to submit your solutions to these problems.

All of these are well-known classic problems with well-known DP solutions. It is good to try to come up with the recurrences on your own[31]. After you've tried really hard to do so but still don't seem to have any reasonable idea what

[29] Proof by AC: forget about proving one or more possible greedy solutions formally; just try coding them and submitting all of them; if any of them work, Q.E.D.

[30] Word, Google Docs, LaTeX, handwritten and scanned, etc.

[31] Never mind if you spend so much time that you cannot finish the required subtask 3 and 4 problems below. It is sometimes better to take it in more slowly and to discover some things on your own. But, hey, feel free to surprise me by successfully managing to do that and *still* managing to do as well as the more experienced trainees on the required problems.

the recurrence should look like[32], you should be able to easily find the solutions on the internet (don't read the code, just read the recurrences). After you've discovered the recurrences, whether on your own or by using the internet, test yourself if you can correctly convert them into DP solutions. The implementation should not take you too long to do. If you spend more than 20 minutes writing the code for any of these problems, ask for help on Discord.

- Project Euler #15 - Lattice Paths (on HackerRank) (number of paths in a grid)
- Codeforces 313B - Ilya and Queries (static range sum, prefix sum)
- Codeforces 416B - Art Union (maximum path in a grid)
- UVa 111 - History Grading (longest increasing subsequence)
- UVa 357 - Let Me Count The Ways (coin change)
- UVa 10130 - SuperSale (0-1 knapsack)
- UVa 10304 - Optimal Binary Search Tree
- UVa 10405 - Longest Common Subsequence (longest common subsequence)
- UVa 10684 - The jackpot (maximum subarray)

6.2 Subtask 2

If you are having difficulty solving the Subtask 3 problems, try some of these first. You are not required to submit your solutions to these. These problems are either simple variations of classic problems, or the ideas for solving them have already been discussed in the tutorial above. If you spend more than 10 minutes trying to come up with a reasonable recurrence (implementation aside) for any of these problems, ask for help on Discord.

- Codeforces 2B - The least round way
- Codeforces 4D - Mysterious Present
- Codeforces 106C - Buns
- Codeforces 368B - Sereja and Suffixes
- Codeforces 446A - DZY loves sequences
- Codeforces 455A - Boredom
- Codeforces 474D - Flowers
- Codeforces 626B - Cards
- Codeforces 698A - Vacations
- Codeforces 706C - Hard Problem
- Codeforces 711C - Coloring Trees
- Codeforces 814C - An impassioned circulation of affection
- UVa 108 - Maximum Sum

6.3 Subtask 3

These problems are worth 100 points each.

- Codeforces 82D - Two out of Three
- Codeforces 191A - Dynasty Puzzles
- Codeforces 225C - Barcode

[32] Don't worry if you fail to discover the recurrence on your own. It doesn't mean that you won't be able to understand DP and won't be able to apply it to solve harder problems. Most people learn DP by seeing example recurrences first for a lot of classic problems before being able to write their own for non-classic problems.

- Codeforces 431C - k-Tree
- Codeforces 478D - Red-Green Towers
- Codeforces 566F - Clique in the Divisibility Graph
- Codeforces 607B - Zuma
- UVa 11022 - String factoring
- UVa 10918 - Tri Tiling

6.4 Subtask 4

These problems are worth 300 points each.

- Project Euler #411 - Uphill paths
- Project Euler #467 - Superinteger
- Codeforces 30C - Shooting Gallery
- Codeforces 223B - Two Strings
- Codeforces 372B - Counting Rectangles is Fun
- Codeforces 383D - Antimatter
- Codeforces 835D - Palindromic Characteristics
- Codeforces 946D - Timetable

6.5 Subtask 5

These problems are worth 700 points each.

- ICPC Manila 2017 - Weird Keyboard (on HackerRank)
- Project Euler #480 - The Last Question
- Codechef CHN16H - Sum and Xor
- Codechef KGP16A - Finding Seats