NOI.PH Training: DP 1

Introduction to Dynamic Programming

Vernon Gutierrez

Contents

1 Introduction

The easiest way to get started learning dynamic programming is to watch these four DP lecture videos from MIT.

After that, you should learn about a few classic problems that are solvable with dynamic programming. You can look at the problems at `https://progvar.fun/problemsets/dp-classic`. You don't need to solve them from scratch. The solutions to these problems are well-known and many are linked in the tutorial tab on the same page. Convince yourself that the solutions to these problems indeed work and give the correct answer. Then, first learn and practice how to write the code (with memoization) when you are already given the recurrence. Coming up with a brand new solution is much harder than just understanding and implementing an existing solution, so make sure you can do these last two first.

After learning these, you may start asking: "Yes, if you give me the recurrence, I understand why it works. Yes, I can implement and memoize it. But how did you come up with that recurrence in the first place?" Indeed, to be successful at applying DP to contest problems, it is not enough to memorize a bunch of classic problems and memorize their solutions. Problem setters will try hard to give original problems, which require finding new solutions instead of just recalling old ones. Hence, you must learn the thought process behind coming up with DP recurrences.

This tutorial assumes that you've understood the basic concepts from the lecture videos and classic problems above. We'll focus mainly on the question of how to come up with recurrences and practical implementation issues.

If you haven't already gotten the pattern by now, DP can basically be summarized by the following equation:

$$DP = brute\,force + memoization$$

That's all there is to it. It may not seem this way at first, but once you truly understand this equation, your perspective on DP will change: it is one of the easiest algorithmic techniques.

2 Implementing DP Solutions in C++

Surprisingly, the "memoization" part of the equation is easier than the "brute force" part, so we'll cover that first. 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`[1] 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:

```cpp
#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:

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

map<tuple<int, int>, int> memo;
int OPT(int n, int m) {
    if(not memo.count({n, m})) {
        int ans;
        // solve the recurrence here and save the answer to ans
        memo[{n, m}] = ans;
    }
    return memo[{n, m}];
}
```

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, 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.

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

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

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.[2]

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

const int MAX_N = 1000, MAX_M = 1000;
int memo[MAX_N+1][MAX_M+1];
bool in_memo[MAX_N+1][MAX_M+1];
int OPT(int n, int m) {
    if(not 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() {
    // initially, all subproblems are not yet in the memo
    memset(in_memo, false, sizeof in_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>
using namespace std;

const int MAX_N = 1000, MAX_M = 1000;
int memo[MAX_N+1][MAX_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];
}
```

[2] Why not just use `unordered_map`? Well, you would have to define a custom hash function anyway, and the constant factors incurred by `unordered_map` are typically comparable to the log factor incurred by `map`.

```cpp
int main() {
    // use -1 to mean "not yet in the memo"
    memset(memo, -1, sizeof 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>
using namespace std;

const int INF = 1'000'000'000;
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;
    denominations.resize(number_of_denominations);
    for(int i = 0; i < number_of_denominations; i++) {
        cin >> denominations[i];
    }
    cout << C(amount) << endl;
    return 0;
}
```

And here's the memoized version.

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

const int INF = 1'000'000'000, MAX_N = 1000;
int memo[MAX_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;
    denominations.resize(number_of_denominations);
    for(int i = 0; i < number_of_denominations; i++) {
        cin >> denominations[i];
    }
    cout << C(amount) << endl;
    return 0;
}
```

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

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

const int INF = 1'000'000'000, MAX_N = 1000;
int C[MAX_N+1];
vector<int> denominations;
int main() {
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    denominations.resize(number_of_denominations);
    for(int i = 0; i < number_of_denominations; i++) {
        cin >> denominations[i];
    }
    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 very minor differences in 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.[3] 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.

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

const int INF = 1'000'000'000, MAX_N = 1000;
int C[MAX_N+1];
vector<int> denominations;
int main() {
    int amount, number_of_denominations;
    cin >> amount >> number_of_denominations;
    denominations.resize(number_of_denominations);
    for(int i = 0; i < number_of_denominations; i++) {
        cin >> denominations[i];
    }
    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.[4] 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

[3] 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.

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

in which to compute the subproblems, 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 Minimum 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 eventually need to learn how to implement DP bottom-up.

2.1 Common Mistakes

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 people prefer to just add a lot of allowance to the memo size to avoid the pain, so they would instead define `MAX_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.[5] 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.[6] There are correct shortcuts for setting 1D arrays, namely `fill` and `fill_n`, but they only really work nicely with 1D arrays.

Exercise 2.1. If you want to understand what can go wrong with `memset`, try to compile and run the code below.

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

const int N = 10;
int memo1[N];
int memo2[N][N];
int main() {
    memset(memo1, 2, sizeof memo1);
    for(int i = 0; i < N; i++)
```

[5] See this for more explanation.

[6] Or, just use a separate Boolean array for marking if a subproblem has been solved or not.

```
        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.

Exercise 2.2. Try changing the types of the arrays above to hold `double` values instead. What happens when you run it?

Exercise 2.3. Before moving on, you should solve a bunch of the problems listed in https://progvar.fun/problemsets/dp-classic. Implementation should be the easiest part in writing a DP solution. You should make sure you have this down first.

3 How to Come Up with Recurrences

3.1 The Right Way to Think About Recursion

Often, the reason why people struggle with DP is not because they don't understand the main idea of DP, which is really simple: store results so that you don't have to compute them again. The trouble seems to be that they have a broken idea of how recursion works.

> **Exercise 3.1.** Try solving a few of the problems listed here: https://progvar.fun/problemsets/recursion.

If you had trouble with these problems, this section might help. Otherwise, you can safely 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*.

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[7], then this won't feel that weird.

This excellent article reiterates these points and gives a nice example. In more complicated scenarios, there may be not just one but many subproblems to your recurrence, but the key thing to remember is that each of these subproblems must be *one step* smaller. "One step" here does not necessarily just mean that the size of the input to the subproblem is literally one smaller than the original problem. See, for example, Minimum Coin Change. The important thing is that you can conceivably transform the answer to the smaller subproblem into an answer for the original problem in *one step*. It may not even be meaningful to talk about the "size" of a subproblem, as in Shortest Paths.

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.

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

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 subproblems 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 exhausted all the possibilities. I can express this as the following recurrence[8]:

$$C(i) = \min(C(i-1)+1, C(i-10)+1, C(i-25)+1)$$

If this seems hard to swallow, you can try to imagine the universe continuing to split off into three[9] 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.[10]

Of course, in reality, my computer doesn't really become super-parallel, and I'm not really splitting universes.[11] 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*, *exhaustive* choices, I just try all those choices. If I know how to count the number of ways to generate smaller versions of the combinatorial object in question, and if I *exhaust* all the possible choices at each stage, then by the Rule of Sum, the total number of ways to generate the original version of the combinatorial object is just the sum of the number of ways to generate the smaller versions. 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.[12] 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.[13]

For example, let's consider the counting version of the coin change problem, where we now

[8] 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.

[9] Or fewer with smaller amounts of money

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

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

[12] 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.

[13] Or, we apply the Principle of Inclusion-Exclusion

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[14]
$$C(i) = C(i-1) + C(i-10) + C(i-25)$$

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 3.2. Come up with the correct recurrence for **Let Me Count The Ways:** UVa 357

Exercise 3.3. Why didn't we have this problem for the optimization version of the problem?

3.2 Pure Brute Force Recurrences vs. DP-Able Recurrences

DP is easy, because basically all that is required is brute force. And brute force means, "Just do it." Right?

$$DP = bruteforce + memoization$$

Actually, this is a slight lie. DP recurrences need to be a little bit smart and carefully designed so that subproblems can overlap.

For example, if in the knapsack problem[15], 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 $\mathcal{O}(n^2 2^n)$[16] rather than $\mathcal{O}(nW)$, which is bad if we have a lot of items but the weights are small.[17]

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

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

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

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

[17] Theoretically, $\mathcal{O}(nW)$ 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.

```
        }
    } 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.

Hence,

$$DP = \textbf{careful}\ brute\ force + memoization$$

4 Common 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.

4.1 Explicit state

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

Example 4.1. The best example of this is Minimum Path in a Grid, where the goal of the game is to get from the upper-left corner of the grid to the lower-right corner of the grid, the state of the game is your position in the grid, and the transitions are the moves you can make, already stated in the problem statement.

Example 4.2. Cards: https://codeforces.com/problemset/problem/626/B. The state is literally the state of the deck of cards, and the transitions you can make are already given in the problem statement.

4.2 Implicit state, but explicit choices

This is a little bit less easier than *explicit state*, but still quite easy. 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. However, it's still quite obvious what your moves should be.

Example 4.3. A classic example is Minimum Coin Change, 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.

Example 4.4. Vacations: https://codeforces.com/problemset/problem/698/A. The moves are literally the actions that Vasya can do on a given day.

4.3 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 form $OPT(i, ...)$, 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.

Example 4.5. A classic example would be Longest Increasing Subsequence (LIS).

When the input to a problem is an integer, as in the case of Minimum Coin Change or Integer Partition, you can view the integer x as an array of x 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.

4.3.1 Subsets

This is a special type of DP on prefixes, where you want to complete-search 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 of a set systematically, it helps to impose an order on the input set, turning it into a list. Typically, the input is already given as an array anyway so this is trivial. To generate subsets, consider the items from right to left (or left to right) and consider the choice of including or excluding 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.

Example 4.6. The canonical example is 0-1 Knapsack.

Example 4.7. Boredom: https://codeforces.com/problemset/problem/455/A. The recurrence here is less obvious. 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.

4.3.2 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.

Example 4.8. Buns: https://codeforces.com/problemset/problem/106/C

One way to complete-search over all possible multisets is to 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

$$OPT(i, \dots) = \bigoplus_j OPT(i-1, \dots) + c(i, j)$$

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.[18]

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

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

We can avoid this extra factor by converting the *multiway* choice of taking some variable number of 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

$$OPT(i, \dots) = \oplus(OPT(i-1, \dots), OPT(i, \dots) + c(i))$$

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.[19]

Take a moment to convince yourself that this does the same thing as the earlier recurrence, except it is computationally much more efficient.[20]

Notice that i is not decreased in the second branch of the recurrence. To avoid infinite recursion, ensure that the second branch is indeed a smaller subproblem: there must be other parameters whose values differ from the original problem, reflecting the fact that item i was chosen.

Example 4.9. Writing Code: https://codeforces.com/problemset/problem/543/A. See if you can find a solution for this problem that runs in time. Ignore memory usage for now.

4.3.3 Subsequences with constraints on consecutive elements

This is almost like DP on 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 complete-search through all possible decisions of which of the n items to make the last element of the subsequence and "jump" to that position in the array, complete-search through all possible decisions of which of the remaining items to make 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 to complete-search over all possible next elements of our subsequence, the second method due to

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

[20] 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 apply to all items taken together. That is, we can have

$$OPT(i, W, \dots) = \bigoplus_j OPT(i-1, W - w(i, j), \dots) + c(i, j)$$

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

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

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

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[21], and because the for-loop is often much easier to optimize away than the extra parameter.[22]

4.3.4 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 complete-search over all possible ways of cutting out some suffix of our current prefix to make the last cell of our partition. The recurrence typically looks like

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

where $\oplus$ is min, max, or $\sum$ depending on our problem, and $c(x, y)$ represents the cost or reward of putting all elements of the subarray of the input between indices x and y into a single partition.[23]

Example 4.10. A classic example is Text Justification.[a]

[a]Here, the strategy that Erik Demaine follows is DP on *suffixes*, which is basically the same thing as DP on prefixes, except 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: those where $OPT(i, \dots)$ depends on subproblems of the form $OPT(i-k, \dots)$ subproblems rather than of the form $OPT(i+k, \dots)$. Usually, this makes the math nicer. It will also make it easier to implement the recurrence with a bottom-up style DP and to later apply optimizations.

Example 4.11. Coloring Trees: https://codeforces.com/problemset/problem/711/C. First, try to complete-search over all possible ways of cutting out the last cell via looping through the number of elements that must end up in the last cell. This may be too slow, but if implemented correctly, should pass the first few test cases. To optimize, convert the multiway choice of putting some variable number of items into the last cell into a binary choice of either putting the last item and the second-to-the-last item into the same cell, or putting them into different cells.

4.3.5 Subarrays

Here, we want to complete-search over all possible subarrays, but typically the problem input size is too large for even quadratic time solutions. Instead, we try to 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 $i-1$.[24]

Example 4.12. Classic example would be Maximum Subarray.[a]

[a]Arguably, this is not really a dynamic programming algorithm because it has some greedy component. See

[21]Note that it is possible to avoid needing the extra space for the second method, but it is trickier to achieve.
[22]See, for example, the $\mathcal{O}(n \lg n)$ algorithm for LIS.
[23]As usual, there are no associated costs for counting problems.
[24]Or, some constant number of earlier subarrays

this answer on StackOverflow. But ehh, we don't want to be pedantic about it.

Example 4.13. An impassioned circulation of affection: https://codeforces.com/problemset/problem/814/C. 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.

4.3.6 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.

Example 4.14. The classic example is Longest Common Subsequence.

4.4 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.

Example 4.15. This is very common for palindrome-type problems, such as Longest Palindromic Subsequence (2000 IOI – Palindrome).

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. The recurrence is usually of the form

$$OPT(i, j, \dots) = \bigoplus_{i \le k < j} OPT(i, k, \dots) \otimes OPT(k+1, j, \dots) \otimes c(i, j)$$

where $\oplus$ is min, max, or $\sum$ and $\otimes$ is $+$ or $\times$ depending on the problem, and $c(i, j)$ is the cost of considering the interval spanning from i to j.[25]

Example 4.16. An example of this would be Matrix Chain Multiplication.

[25] As usual, this is zero for most counting problems. The cost of making the decision to split at k is paid for at the next level of recursion, where $c(i, k)$ and $c(k, j)$ appears. We could have written this in a different way, so that c has a dependence on k at the current level of recursion, but the way presented above is commonly the cleanest way of writing it.

Example 4.17. Another nice, classic problem where this idea can be applied 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* ways to cut the current interval into two, while in divide-and-conquer, we consider only a single, fixed splitting point.

Example 4.18. Dishonest Driver (2018 ICPC SWERC)

Example 4.19. Zuma: `https://codeforces.com/problemset/problem/607/B`

5 A Worked Example

To get a *true feel* for DP, let's go step-by-step into the thought process of coming up with a DP solution for **Caesar's Legions:** https://codeforces.com/problemset/problem/118/D. Before you read the explanation below, I recommend trying to solve it yourself first.

5.1 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 subproblem, and it's not immediately clear what our 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, $DP = brute force + memoization$. 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, $0 \le i \le n_1 + n_2$.

5.2 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:

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

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.

5.3 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.

5.4 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:

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

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.

5.5 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.

5.6 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:

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

where:

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

and:

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

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.

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

where:

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

and:

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

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 $C(n_1 + n_2, n_1, n_2, k_1)$.

5.7 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:

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

where:

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

and:

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

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.

5.8 Analyzing the running time, attempt 4

The running time of our current solution is $\mathcal{O}((n_1 + n_2)n_1n_2k_1k_2)$. 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?

5.9 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.

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

where:

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

and:

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

The answer we need is $C(n_1, n_2, k_1, k_2)$. The running time reduces to $\mathcal{O}(n_1n_2k_1k_2)$, and our solution is now a sure AC solution.

5.10 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 <bits/stdc++.h>
using namespace std;

const int MOD = 100'000'000;
int add(int x, int y) { return (x + y) % MOD; }

int n1, n2, k1, k2;
int C(int f, int h, int kf, int kh) {
    if(f + h == 0) return 1;
    else {
        int ans = 0;
        if(f > 0 and kf > 0) ans = add(ans, C(f - 1, h, kf - 1, k2));
        if(h > 0 and kh > 0) ans = add(ans, C(f, h - 1, k1, kh - 1));
        return ans;
    }
}
int main() {
    cin >> n1 >> n2 >> k1 >> k2;
    cout << C(n1, n2, k1, k2) << endl;
    return 0;
}
```

5.11 Memoize

Finally! This is the easiest step. I'm sure you can do it on your own, before comparing with mine.

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

const int MOD = 100'000'000, MAX_N = 100, MAX_K = 10;
int add(int x, int y) { return (x + y) % MOD; }

int n1, n2, k1, k2;
int memo[MAX_N+1][MAX_N+1][MAX_K+1][MAX_K+1];
int C(int f, int h, int kf, int kh) {
    if(memo[f][h][kf][kh] == -1) {
        int ans;
        if(f + h == 0) ans = 1;
        else {
            ans = 0;
            if(f > 0 and kf > 0) ans = add(ans, C(f - 1, h, kf - 1, k2));
            if(h > 0 and kh > 0) ans = add(ans, C(f, h - 1, k1, kh - 1));
        }
        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's a good idea to define the sizes of memo as constants. This way, it becomes much easier to spot off-by-one errors, errors due to misread problem constraints, or even just typos.

Because of the introduction of variable `ans` in line 10, we've already minimized the places where we have to repeat typing `memo[f][h][kf][kh]`. Doing this is good practice: it helps avoid bugs and minimizes the amount of code you need to change when you realize your state representation is wrong. The advantage becomes much clearer the more parameters we have in our recurrence.

There is a way to completely avoid having multiple copies of `memo[f][h][kf][kh]` in the code, using C++ references:

```
int C(int f, int h, int kf, int kh) {
    // The variable ans is a reference. Reassigning it also reassigns the
    ↪  right hand side in its definition.
    int& ans = memo[f][h][kf][kh];
    if(ans == -1) {
        if(f + h == 0) ans = 1;
        else {
            ans = 0;
            if(f > 0 and kf > 0) ans = add(ans, C(f - 1, h, kf - 1, k2));
            if(h > 0 and kh > 0) ans = add(ans, C(f, h - 1, k1, kh - 1));
        }
    }
    return ans;
}
```

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

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

const int MOD = 100'000'000, MAX_N = 100, MAX_K = 10;
int add(int x, int y) { return (x + y) % MOD; }

int C[MAX_N+1][MAX_N+1][MAX_K+1][MAX_K+1];
int main() {
    int n1, n2, k1, k2;
    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++) {
```

```
15          int& ans = C[f][h][kf][kh];
16          if(f + h == 0) ans = 1;
17          else {
18              ans = 0;
19              if(f > 0 and kf > 0) ans = add(ans, C[f -1][h][kf-1][k2]);
20              if(h > 0 and kh > 0) ans = add(ans, C[f][h -1][k1][kh-1]);
21          }
22      }
23      cout << C[n1][n2][k1][k2] << endl;
24      return 0;
25  }
```

5.12 Attempt 6?

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

6 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.

   > **Example 6.1. DZY loves sequences:** https://codeforces.com/problemset/problem/446/A. It's a nice variant of the somewhat well-known Longest Bitonic Subsequence.

   What's that? You haven't heard of Longest Bitonic Subsequence before? Then you should spend lots of time memorizing all the classic problems listed in GeeksForGeeks.

   Just kidding. It's more important to learn the process of coming up with dynamic programming solutions, and once you've done this, it's not impossible to come up with the solution to new problems without seeing some related classic problem before. However, knowing several classic problems does make it easier.

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. See the worked example of **Caesar's Legions:** https://codeforces.com/problemset/problem/118/D above.

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[26] 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. You will encounter more advanced techniques in the future.

6. Most video lectures and books leave the "how to reconstruct the actual optimal solution after figuring out the optimal value" as a side note, for good reasons. 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. From time to time,

[26] 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.

however, you may encounter a problem that requires it. The probability that you need to do this increases if you are using DP to solve a problem that is intended to be solved greedily. Since you don't want to lose the advantages we talked about in point #4, it's nice to practice how to do this a few times for those instances.

Exercise 6.2. Practice now with **Mysterious Present:** https://codeforces.com/problemset/problem/4/D.

7 Problems

7.1 Coding problems

C1 https://progvar.fun/problemsets/dp-classic

C2 https://progvar.fun/problemsets/dp-classic-with-twists

C3 https://progvar.fun/problemsets/dp-patterns

C4 https://progvar.fun/problemsets/dp-with-construction

7.2 Non-coding problems

N1 Exercise 3.3

N2 The following is an attempt to wrote a bottom-up DP implementation for solving Minimum Path in a Grid. The input is given as a zero-indexed two-dimensional grid A with n rows and m columns. The individual entries of the grid, the number of rows, and the number of columns all do not exceed 100.

```cpp
#include <bits/stdc++.h>
using namespace std;
int opt[100][100];
int solve(int n, int m, vector<vector<int>> A) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            int& ans = opt[i][j];
            if(i == 0 and j == 0) {
                ans = A[i][j];
            } else if(i == 0) {
                ans = opt[i][j - 1] + A[i][j];
            } else if(j == 0) {
                ans = opt[i - 1][j] + A[i][j];
            } else {
                ans = min(opt[i][j - 1], opt[i - 1][j]) + A[i][j];
            }
        }
    }
    return opt[n - 1][m - 1];
}
```

Is this code correct? Explain your answer.

N3 The instructions for this exercise are the same as for the previous one, except with this code instead:

```cpp
#include <bits/stdc++.h>
using namespace std;
int opt[100][100];
int solve(int n, int m, vector<vector<int>> A) {
    for(int i = n - 1; i >= 0; i--) {
        for(int j = 0; j < m; j++) {
            int& ans = opt[i][j];
            if(i == 0 and j == 0) {
                ans = A[i][j];
            } else if(i == 0) {
                ans = opt[i][j - 1] + A[i][j];
            } else if(j == 0) {
                ans = opt[i - 1][j] + A[i][j];
            } else {
                ans = min(opt[i][j - 1], opt[i - 1][j]) + A[i][j];
            }
        }
    }
    return opt[n - 1][m - 1];
}
```

N4 The instructions for this exercise are the same as for the previous one, except with this code instead:

```cpp
#include <bits/stdc++.h>
using namespace std;
int opt[100][100];
int solve(int n, int m, vector<vector<int>> A) {
    vector<pair<int, int>> order;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            order.push_back({i, j});
        }
    }
    auto ordering = [](const pair<int, int>& p) -> pair<int, int> {
        auto [i, j] = p;
        return {i + j, j};
    };
    sort(order.begin(), order.end(), [&](auto& p1, auto& p2) {
        return ordering(p1) < ordering(p2);
    });
    for(auto [i, j] : order) {
        int& ans = opt[i][j];
        if(i == 0 and j == 0) {
            ans = A[i][j];
        } else if(i == 0) {
            ans = opt[i][j - 1] + A[i][j];
        } else if(j == 0) {
            ans = opt[i - 1][j] + A[i][j];
        } else {
            ans = min(opt[i][j - 1], opt[i - 1][j]) + A[i][j];
        }
    }
    return opt[n - 1][m - 1];
```

```
}
```

N5 The instructions for this exercise are the same as for the previous one, except with this code instead:

```cpp
#include <bits/stdc++.h>
using namespace std;
int opt[100][100];
int solve(int n, int m, vector<vector<int>> A) {
    vector<pair<int, int>> order;
    for(int i = 0; i < n; i++) {
        vector<int> Js(m);
        iota(Js.begin(), Js.end(), 0);
        random_shuffle(Js.begin(), Js.end());
        for(int j : Js) {
            order.push_back({i, j});
        }
    }
    for(auto [i, j] : order) {
        int& ans = opt[i][j];
        if(i == 0 and j == 0) {
            ans = A[i][j];
        } else if(i == 0) {
            ans = opt[i][j - 1] + A[i][j];
        } else if(j == 0) {
            ans = opt[i - 1][j] + A[i][j];
        } else {
            ans = min(opt[i][j - 1], opt[i - 1][j]) + A[i][j];
        }
    }
    return opt[n - 1][m - 1];
}
```

N6 The instructions for this exercise are the same as for the previous one, except with this code instead:

```cpp
#include <bits/stdc++.h>
using namespace std;
int opt[100][100];
int solve(int n, int m, vector<vector<int>> A) {
    vector<pair<int, int>> order;
    for(int i = 0; i < n; i++) {
        vector<int> Js(m);
        iota(Js.begin(), Js.end(), 0);
        random_shuffle(Js.begin(), Js.end());
        for(int j : Js) {
            order.push_back({i, j});
        }
    }
    auto ordering = [](const pair<int, int>& p) -> int {
        auto [i, j] = p;
        return i + j;
    };
    sort(order.begin(), order.end(), [&](auto& p1, auto& p2) {
        return ordering(p1) < ordering(p2);
```

```
    });
    for(auto [i, j] : order) {
        cerr << i << " " << j << endl;
        int& ans = opt[i][j];
        if(i == 0 and j == 0) {
            ans = A[i][j];
        } else if(i == 0) {
            ans = opt[i][j - 1] + A[i][j];
        } else if(j == 0) {
            ans = opt[i - 1][j] + A[i][j];
        } else {
            ans = min(opt[i][j - 1], opt[i - 1][j]) + A[i][j];
        }
    }
    return opt[n - 1][m - 1];
}
```

N7 The instructions for this exercise are the same as for the previous one, except with this code instead:

```
#include <bits/stdc++.h>
using namespace std;
int opt[100][100];
int solve(int n, int m, vector<vector<int>> A) {
    vector<pair<int, int>> order;
    for(int i = 0; i < n; i++) {
        vector<int> Js(m);
        iota(Js.begin(), Js.end(), 0);
        random_shuffle(Js.begin(), Js.end());
        for(int j : Js) {
            order.push_back({i, j});
        }
    }
    auto ordering = [](const pair<int, int>& p) -> int {
        auto [i, j] = p;
        return i * i + j * j;
    };
    sort(order.begin(), order.end(), [&](auto& p1, auto& p2) {
        return ordering(p1) < ordering(p2);
    });
    for(auto [i, j] : order) {
        cerr << i << " " << j << endl;
        int& ans = opt[i][j];
        if(i == 0 and j == 0) {
            ans = A[i][j];
        } else if(i == 0) {
            ans = opt[i][j - 1] + A[i][j];
        } else if(j == 0) {
            ans = opt[i - 1][j] + A[i][j];
        } else {
            ans = min(opt[i][j - 1], opt[i - 1][j]) + A[i][j];
        }
    }
    return opt[n - 1][m - 1];
}
```

N8 Write a recurrence that, when implemented as a DP solution, solves **Caesar's Legions:**

https://codeforces.com/problemset/problem/118/D in $\mathcal{O}(n_1 n_2 \max(k_1, k_2))$

N9 Write a recurrence that, when implemented as a DP solution, solves **Writing Code:** https://codeforces.com/problemset/problem/543/A within the time limit. Ignore the memory limit for now.

N10 Problem Statement

Nina is playing a game of darts.

She has 4 darts which she can throw at the dartboard. She doesn't need to throw all 4 darts. She can even choose to not throw any at all.

The dartboard is divided into N areas. Each are has a score written on it. If a dart lands on an area, Nina gains the score written in that area.

Let S be the sum of all the scores Nina gains from all the darts that she throws. For some M decided in advance, if $S \leq M$, then Nina's final score is S. Otherwise, her final score is 0.

Nina can throw darts with perfect accuracy and make them land on the dartboard wherever she wants. What is the maximum possible final score she can have?

Input

The first line of input contains two integers, N and M.

The next N lines of input contain $S_1, S_2, \ldots, S_N$, each line containing one integer. S_i is the score written on the ith area of the dartboard.

Output

Print on a single line the maximum score Nina can have.

Constraints

$1 \leq N \leq 1000$

$1 \leq M \leq 2 \times 10^8$

$1 \leq S_i \leq 10^8$ for all i

Sample Input 1

```
4 50
3
14
15
9
```

Sample Input 2

```
3 21
16
11
2
```

Sample Output 1

```
48
```

Sample Output 2

```
20
```

Sample Explanation

For the first sample case, Nina will have the maximum possible final score if she lands 3 darts on the area with score 15, and one dart on the area with score 3, for a total score of 48.

For the second sample case, Nina will have the maximum possible final score if she lands 1 dart on the area with score 16, and two darts on the area with score 2, for a total score of 20.

Now, for the actual exercise...

The following is an attempt to solve this problem.

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int MAX_N = 1000, MAX_DARTS = 4;
ll S[MAX_N+1], memo[MAX_N+1][MAX_DARTS+1];
ll opt(int M, int i, int darts) {
    ll& ans = memo[i][darts];
    if(ans == -1) {
        if(i == 0) {
            ans = 0;
        } else {
            ans = 0;
            for(int use = 0; use <= darts; use++) {
                ll total = opt(M, i - 1, darts - use) + use * S[i];
                ans = max(ans, total <= M ? total : 0ll);
            }
        }
    }
    return ans;
}
int main() {
    int N, M;
    cin >> N >> M;
    for(int i = 1; i <= N; i++) {
        cin >> S[i];
    }
    memset(memo, -1, sizeof memo);
    cout << opt(M, N, 4) << endl;
}
```

Explain why this program is wrong, and give a small counterexample.

7.3 More Coding Problems

M1 [1★] Write a solution for **Writing Code:** https://codeforces.com/problemset/problem/543/A that would pass the time limit. Ignore the memory limit for now.

M2 **The Maximum Subarray:** https://www.hackerrank.com/contests/master/challenges/maxsubarray

M3 **SuperSale:** UVa 10130

M4 [1★] **Buns:** https://codeforces.com/problemset/problem/106/C

M5 [1★] **DZY loves sequences:** https://codeforces.com/problemset/problem/446/A

M6 [1★] **Flowers:** https://codeforces.com/problemset/problem/474/D

M7 [1★] **Mashmokh and ACM:** https://codeforces.com/problemset/problem/414/B

M8 [2★] **String factoring:** UVa 11022

M9 [4★] **Red-Green Towers:** https://codeforces.com/problemset/problem/478/D

M10 [4★] **How many trees?:** https://codeforces.com/problemset/problem/9/D

M11 [8★] **Coloring Brackets:** https://codeforces.com/problemset/problem/149/D

M12 [8★] **The Maths Lecture:** https://codeforces.com/problemset/problem/507/D

M13 [16★] **Common ancestor:** https://codeforces.com/problemset/problem/49/E

M14 [32★] **Sum and Xor:** https://www.codechef.com/problems/CHN16H

M15 [32★] **Finding Seats:** https://www.codechef.com/problems/KGP16A