IOI Training camp 2020

DYNAMIC PROGRAMMING

Minkyum Kim

What is Dynamic Programming?

Dynamic programming is where you break down the problem into smaller and smaller sub-problems.

Then, these sub-problems are solved in a way, and only once (no repetition) and stored, so their results can be re-used to solve more sub-problems.

The sub-problems can then be used to solve the problem (e.g. summing, max, min, some function, etc.)

2 types of Dynamic Programming:

Bottom up
(Tabulation)

Top Down
(Memoization)

Bottom up
(Tabulation)

Bottom-up or tabulation is basically where you start at a starting point in a graph, vector, table, etc. and you calculate sub-problems in a specific order to store the answers to these sub-problems.

These answers will help solve future sub-problems and so on until they can be used to solve the actual problem.

Top down

(Memoization)

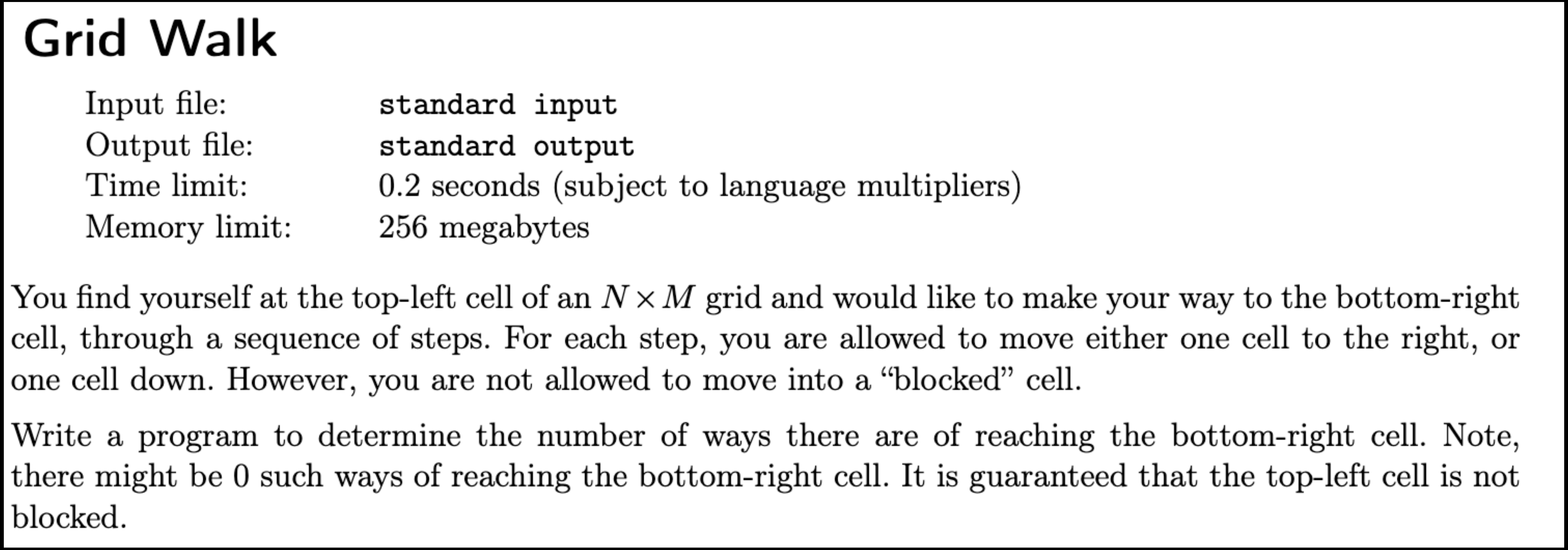
In normal recursion, you will see what you want to calculate/find at the end, and then calculate the things you need, as well as the things you need for the things you need, and so on.

Unfortunately, this creates a lot of repetition, and we calculate many of the things we have already calculated before. This ultimately leads to an exponential time complexity.

To solve this, we use memoization. Instead of recomputing the same values, we store them in memory, so if we have calculated it already, we just retrieve the result. This reduces the time complexity.

Example

SAPO round 2 Q3



Grid Walk

Input file:	`standard input`
Output file:	`standard output`
Time limit:	0.2 seconds (subject to language multipliers)
Memory limit:	256 megabytes

You find yourself at the top-left cell of an $N \times M$ grid and would like to make your way to the bottom-right cell, through a sequence of steps. For each step, you are allowed to move either one cell to the right, or one cell down. However, you are not allowed to move into a "blocked" cell.

Write a program to determine the number of ways there are of reaching the bottom-right cell. Note, there might be 0 such ways of reaching the bottom-right cell. It is guaranteed that the top-left cell is not blocked.

Example

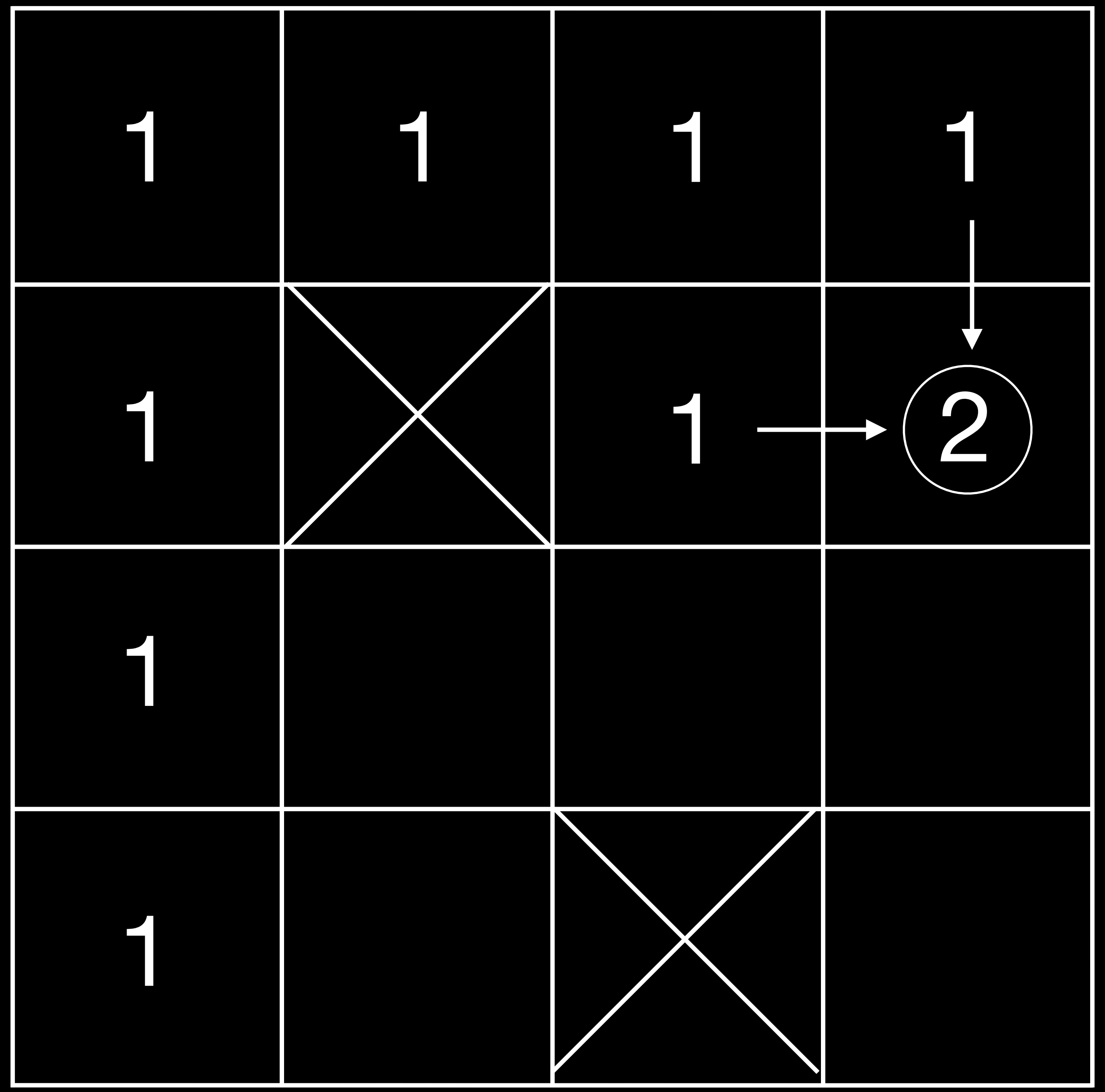
SAPO round 2 Q3

Basically, we have to count the number of ways to go from the top left block to the bottom right, only moving down or right. We cannot pass through the “blocked” cells, marked with an “X” in the diagram.

1	1	1	1
1	X	1	2
1	1	2	4
1	2	X	4

Example

SAPO round 2 Q3

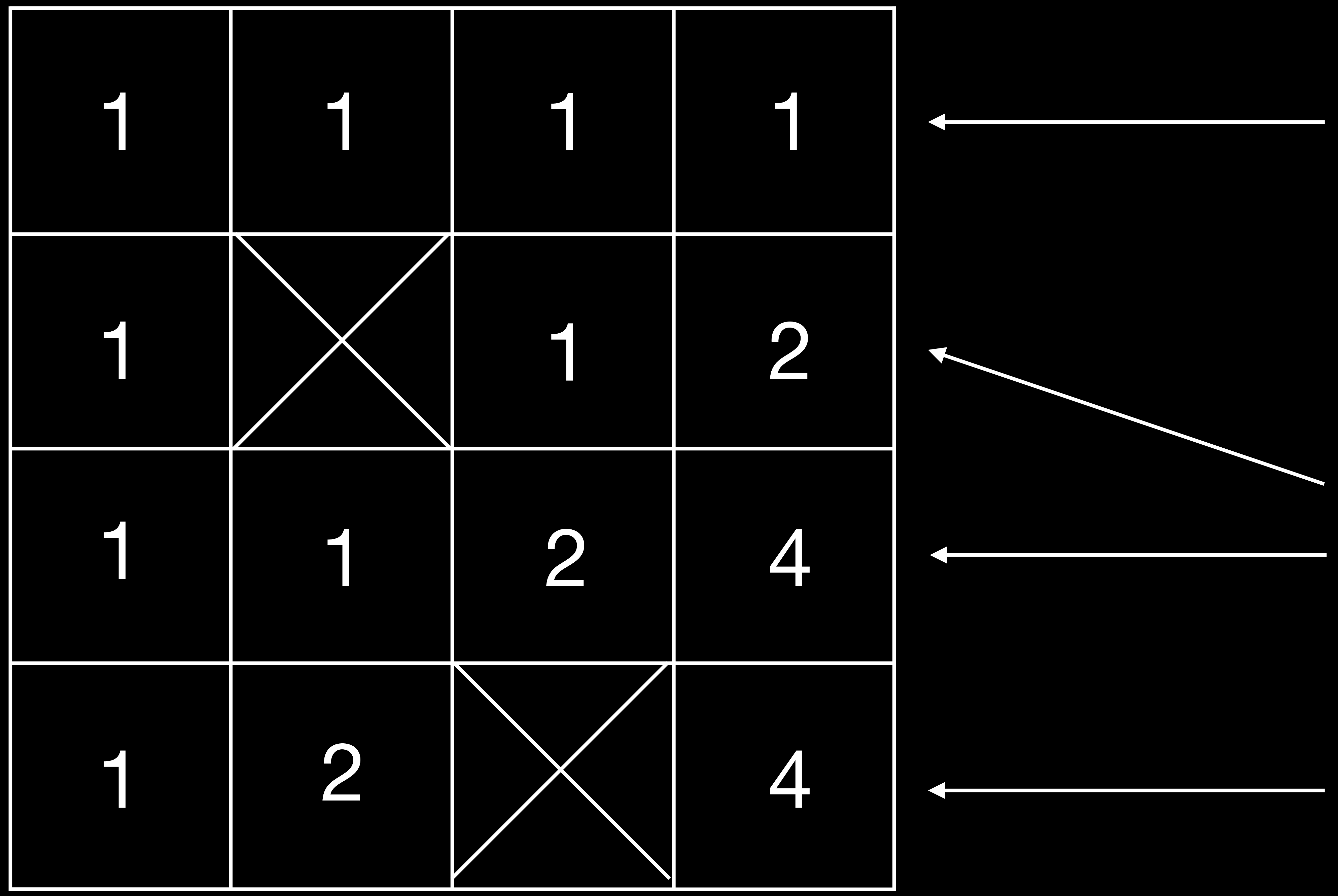


1	1	1	1
1	✕	1	2
1			
1		✕	

Notice how every cell is just the sum of the numbers of the cells above and to the left.

Example

SAPO round 2 Q3



1	1	1	1
1	✕	1	2
1	1	2	4
1	2	✕	4

We first fill in the first row with 1’s (unless blocked)

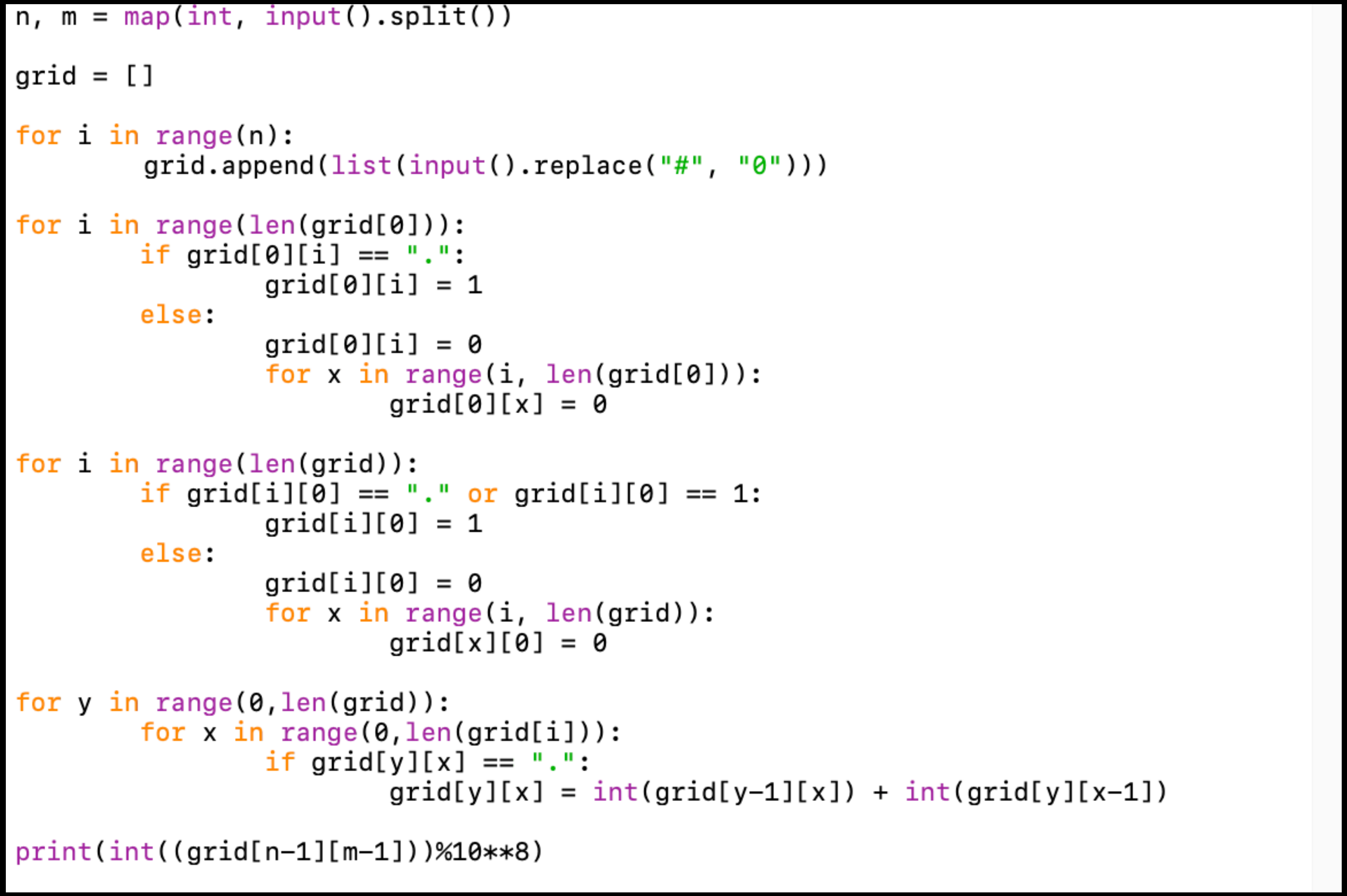
We then work out each cell from left to right by summing the number in the cell above and left to it.

Finally, we get to the bottom right cell we want at the end.

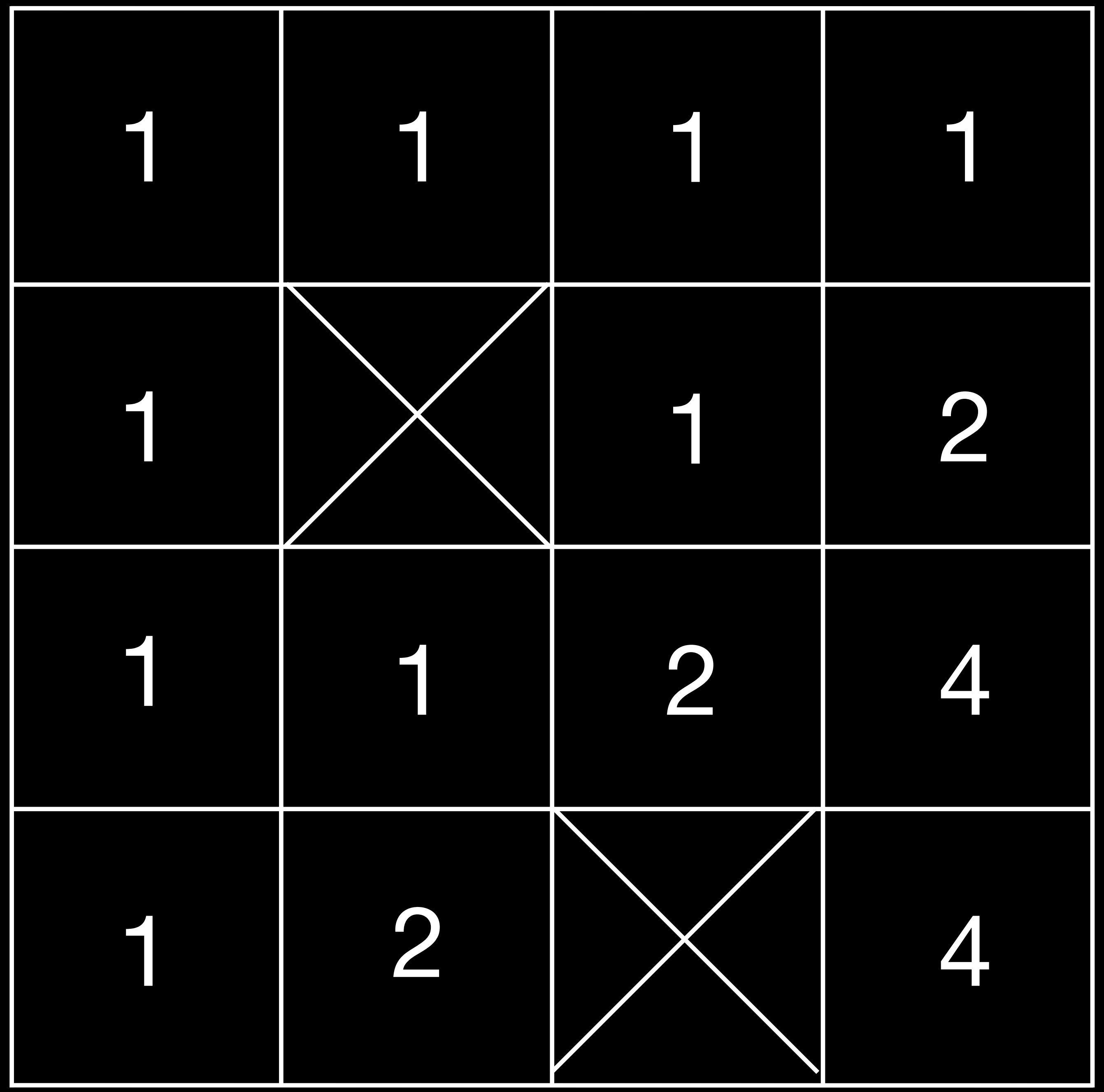
Example

SAPO round 2 Q3

Code:



```python
n, m = map(int, input().split())

grid = []

for i in range(n):
    grid.append(list(input().replace("#", "0")))

for i in range(len(grid[0])):
    if grid[0][i] == ".":
        grid[0][i] = 1
    else:
        grid[0][i] = 0
        for x in range(i, len(grid[0])):
            grid[0][x] = 0

for i in range(len(grid)):
    if grid[i][0] == "." or grid[i][0] == 1:
        grid[i][0] = 1
    else:
        grid[i][0] = 0
        for x in range(i, len(grid)):
            grid[x][0] = 0

for y in range(0,len(grid)):
    for x in range(0,len(grid[i])):
        if grid[y][x] == ".":
            grid[y][x] = int(grid[y-1][x]) + int(grid[y][x-1])

print(int((grid[n-1][m-1]))%10**8)
```



Time complexity: O(nm)

Example

SAPO round 2 Q3

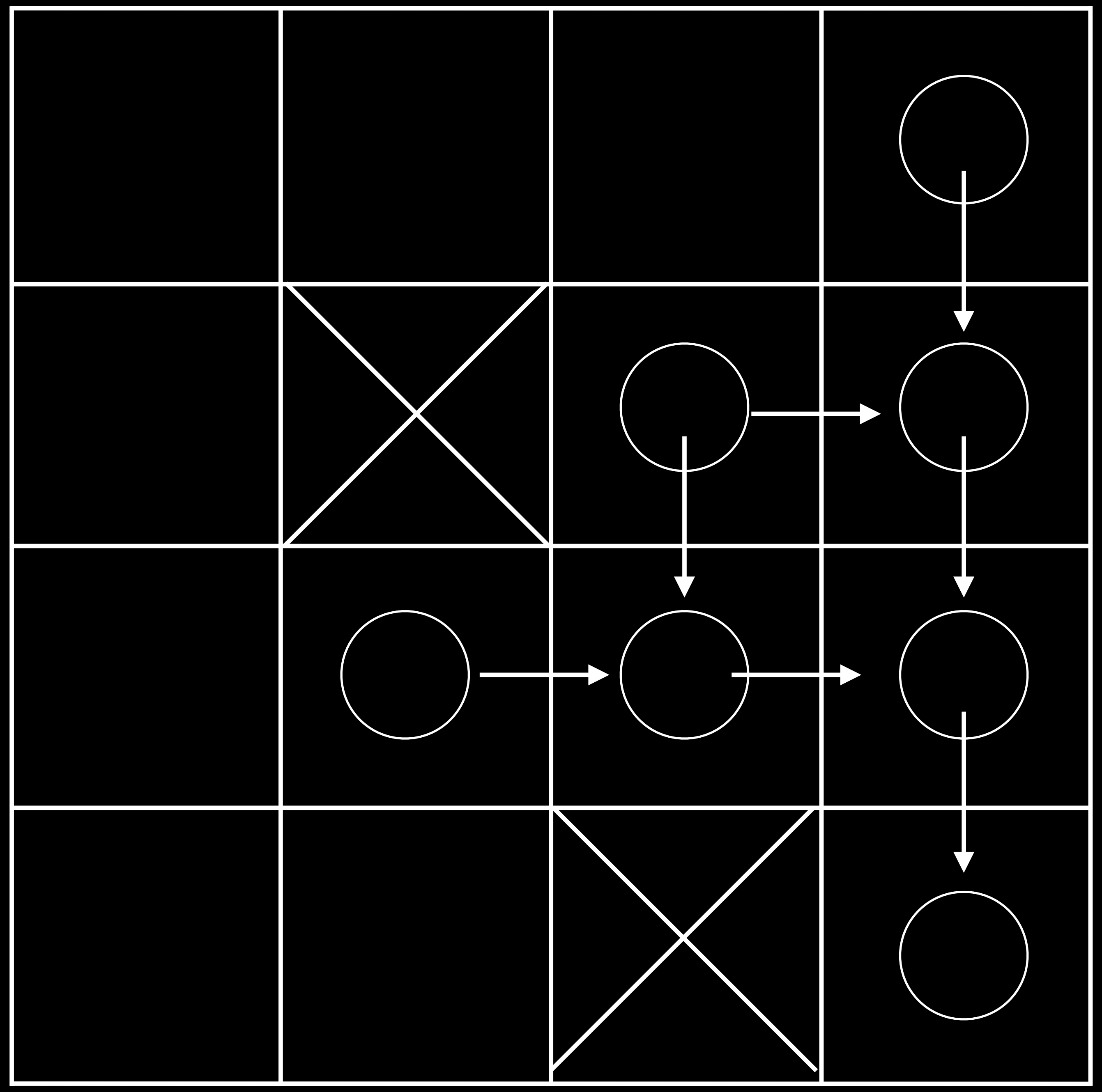
This was an example of **bottom up** or **tabulation**, as we started at a point (the upper left cell) and calculated values in an order (row by row) until we got to the end (the bottom right cell

1	1	1	1
1	╳	1	2
1	1	2	4
1	2	╳	4

Example

SAPO round 2 Q3

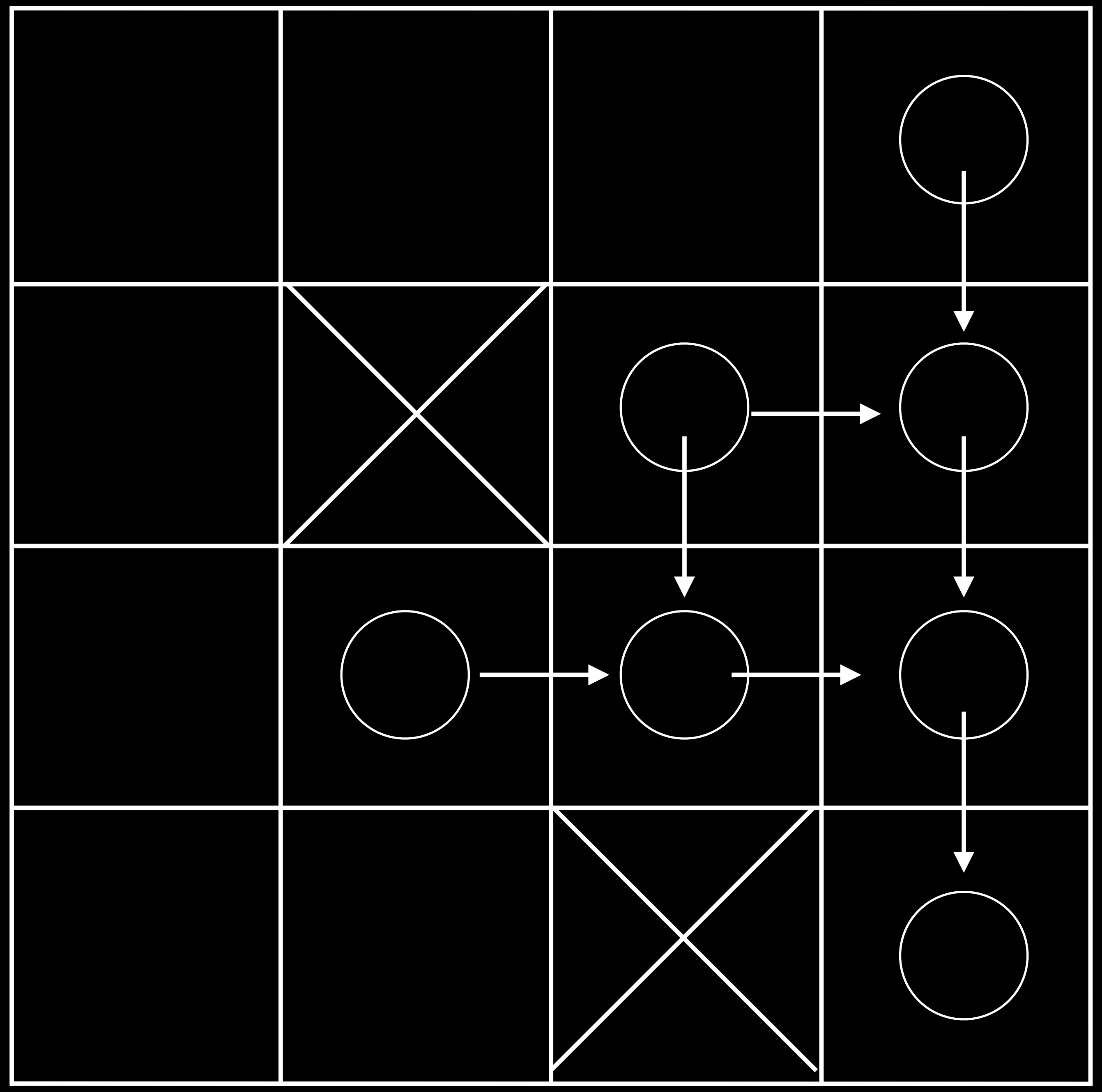
If we wanted to solve this question using **top down** or **memoization**, we need to first look at how we would solve this problem using **recursion**. We need to start at the end (the right bottom cell), and see which cells we have to add to get this cell. Then, we see which cells we have to add to get those cells, and so on.



Example

SAPO round 2 Q3

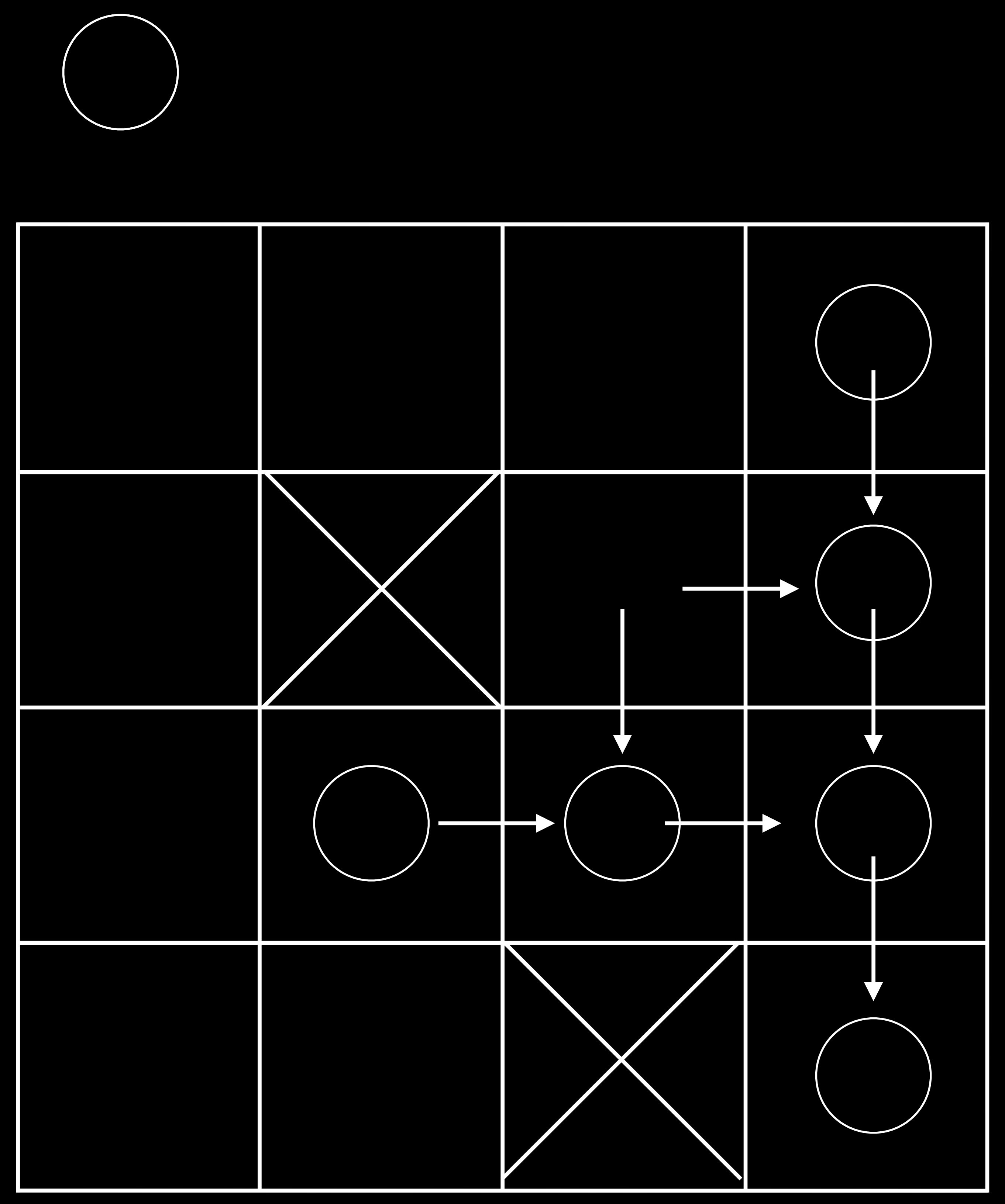
In the previous example, one of the cells had already been calculated **more than once**. As the grid gets larger, the amount of “repetitions” of calculations will grow exponentially. This is bad, as it increases the time complexity of the code.



Example

SAPO round 2 Q3

We use **memoization** to store the things we have already calculated in a list or somewhere in memory so we can just retrieve the result when needed, instead of recalculating the same thing many times.



So, when to use which one?

Bottom up
(Tabulation)

Top Down
(Memoization)

Bottom up

(Tabulation)

We use this method when there is an obvious and nice order in which you can calculate the sub-problems in such a way that you always have everything that you need already calculated for calculating the next/ future sub-problems.

(Like in the SAPO r2 q3, we can just do it by row)

1	1	1	1
1	✗	1	2
1	1	2	4
1	2	✗	4

Bottom up
(Tabulation)

We usually use bottom up or tabulation when there is a nice order because it is faster to code and it runs faster, as recursion and memoization has a lot of recursive calls and return statements. It usually outperforms recursion by a constant factor.

Top down
(Memoization)

Sometimes, there isn’t a nice way to see where to begin and end a problem, or the order in which you do it is not clear. This is when memoization is more useful. Instead of trying to come up with a complicated order in which to evaluate the subproblems, we can just see the end and work backwards.

This works because, first of all, we do not know a order in which we can calculate the subproblem such that the predecessors have already been evaluated to be used to calculate the current subproblem.

Top down
(Memoization)

We use memoization in things like a directed acyclic graph that has not been topologically sorted to give an order for the bottom up or tabulation. Working backwards is easier as we do not have to follow an order, it just finds all the values. We also know this process is finite, as it is acyclic so there are no cycles, hence the process terminates.

Example

Coin problem

Let’s consider a problem where we are given a set of coins and our task is to form a sum of money n using the coins. The values of the coins are `coins=`$\{c_1,c_2, . . . ,c_m\}$, and each coin can be used as many times we want. What is the minimum number of coins needed for n?

Example

Coin problem

We could first try recursion. A recursion code for this would look something like:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Example

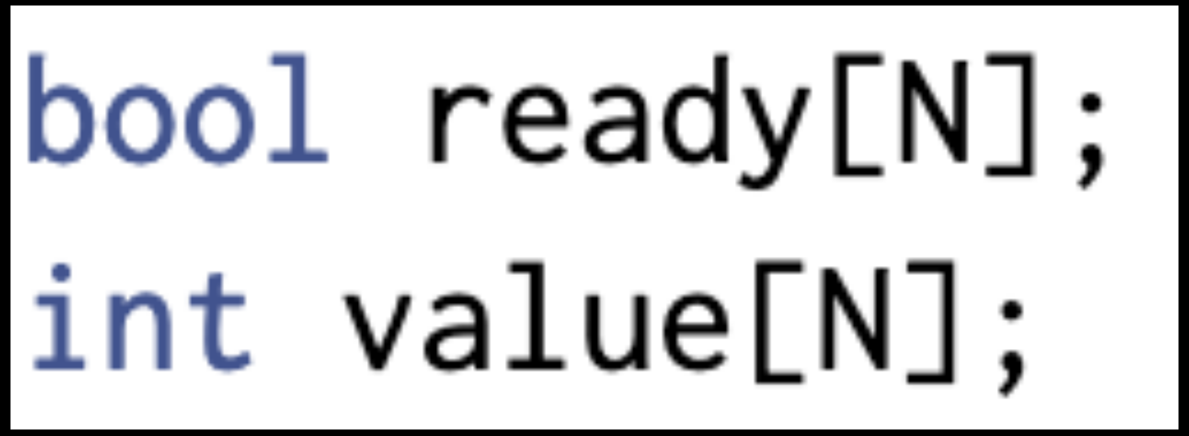
Coin problem

Unfortunately, recursion, just like in the SAPO r2 q3 problem, repeats a lot of calculations for the same thing.

Example

Coin problem

So, we use arrays to store the information. “Ready” tells the program whether the calculation has already been solved, and the “value” tells us what the value of the calculation is.

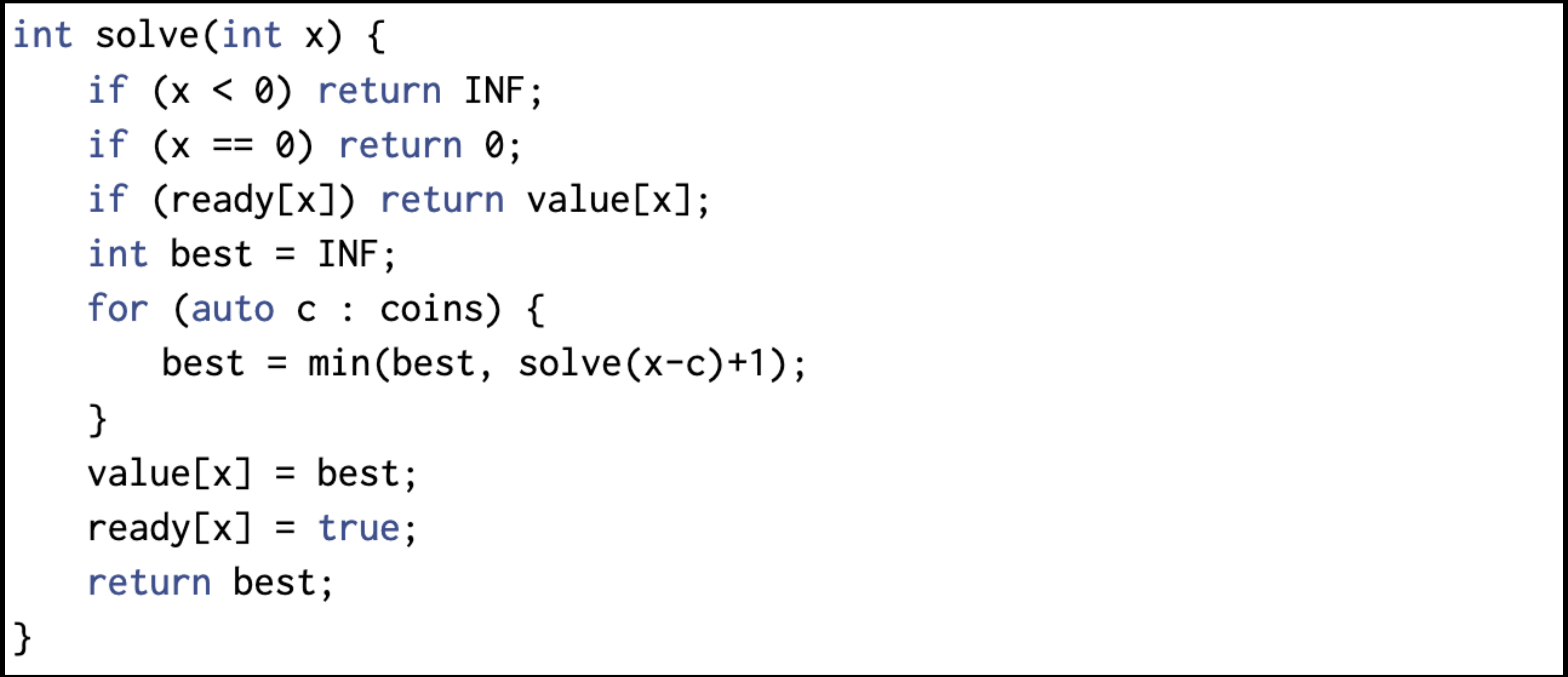


```
bool ready[N];
int value[N];
```

Example

Coin problem

So, a program using memoization would look something like this.



```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}
```

We store the information in the 2 arrays to retrieve information to avoid recalculation. O(nm)

Constructing a solution

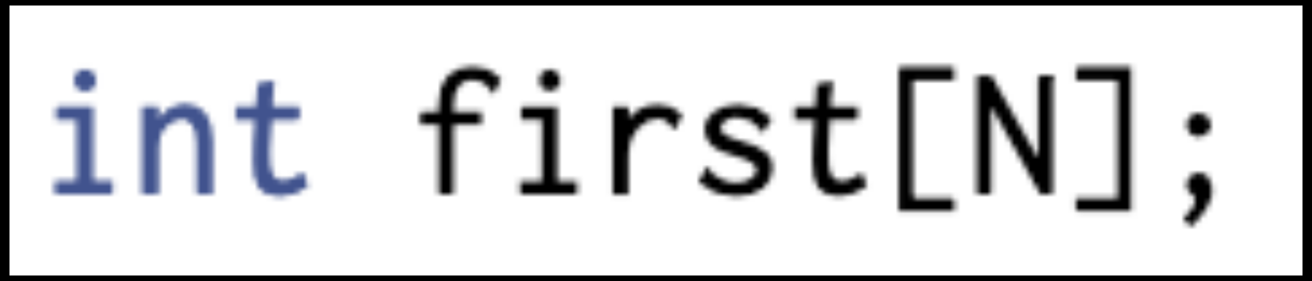
Coin problem

Sometimes we are asked both to find the value of an optimal solution and to give an example how such a solution can be constructed. In our examples in the previous slides, they only tell us what the optimal solution is. Using DP, we can also get the actual coins that make up the optimal solution.

Constructing a solution

Coin problem

We can use an additional array to store the first coin in every solution.

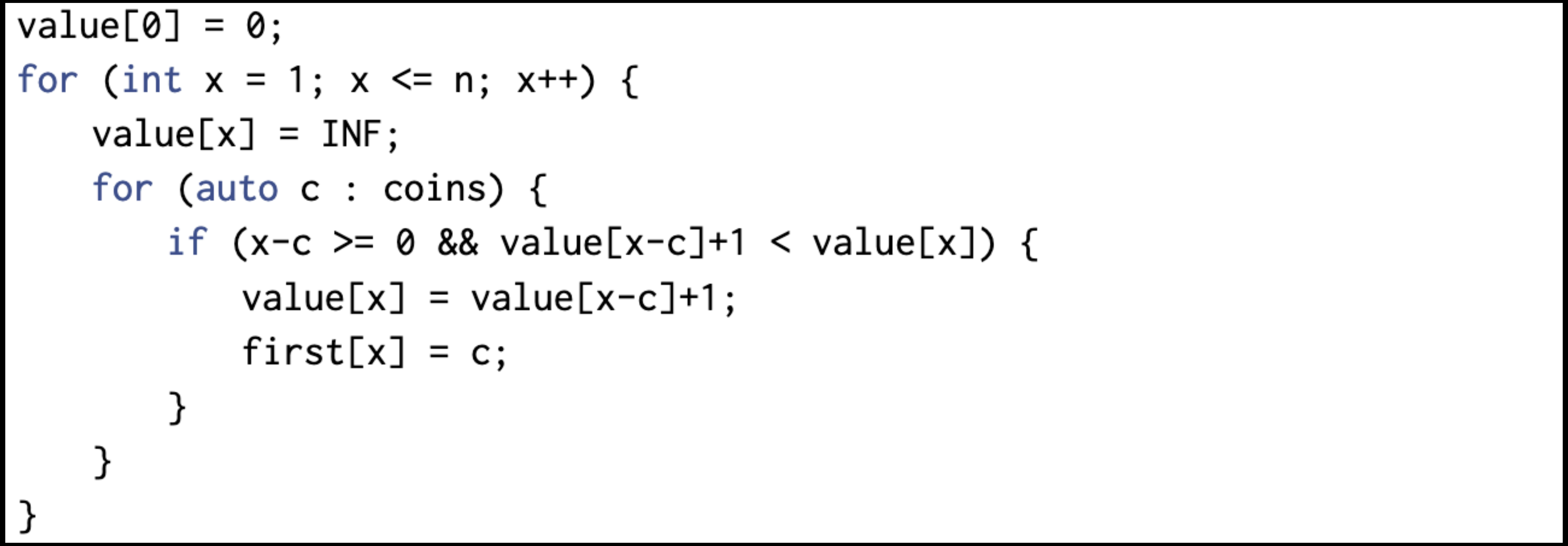


```
int first[N];
```

Constructing a solution

Coin problem

We can then just change the code so it actually stores the first coins in the array.

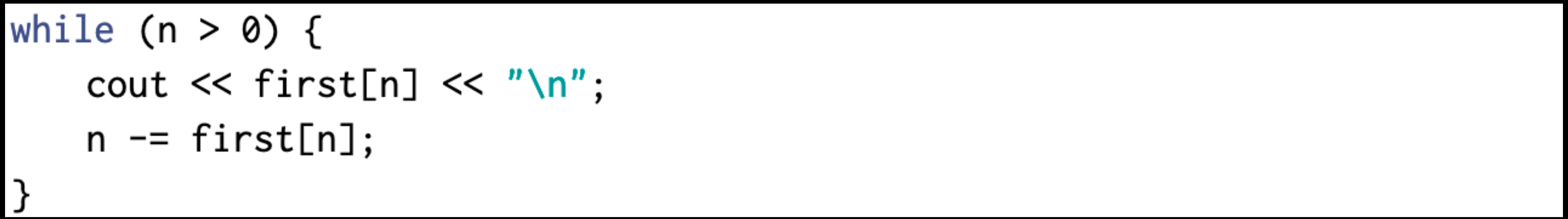


```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}
```

Constructing a solution

Coin problem

Then, we can just calculate the coins in the optimal solution by just adding a few lines of additional code. We keep getting the “first coin”, and just remove the amount of the first coin and repeat.



```
while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}
```