Algorithms

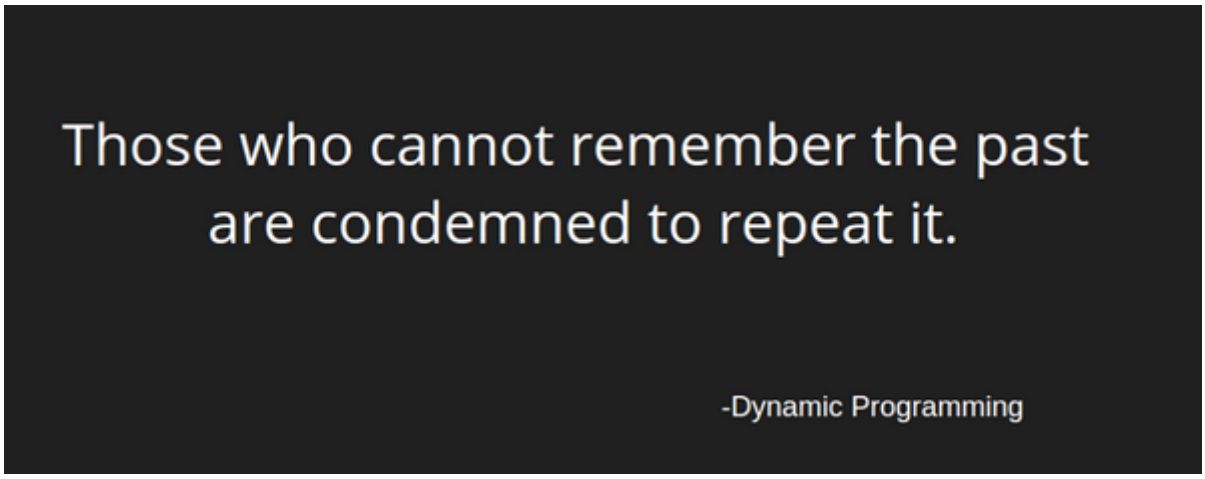
Rank: **23,338**

View Leaderboard

Topics: Introduction to Dynamic Programming 1

# Introduction to Dynamic Programming 1

TUTORIAL PROBLEMS




The image above says a lot about Dynamic Programming. So, is repeating the things for which you already have the answer, a good thing ? A programmer would disagree. That's what Dynamic Programming is about. To *always* remember answers to the sub-problems you've already solved.

Let us say that we have a machine, and to determine its state at time **t**, we have certain quantities called state **variables.** There will be certain times when we have to make a decision which affects the state of the system, which may or may not be known to us in advance. These decisions or changes are equivalent to transformations of state variables. The results of the previous decisions help us in choosing the future ones.

What do we conclude from this? We need to break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems. If you are given a problem, which can be broken down into smaller sub-problems, and these smaller sub-problems can still be broken into smaller ones - and if you manage to find out that there are some over-lappping sub-problems, then you've encountered a DP problem.

?

Some famous Dynamic Programming algorithms are:

- Unix diff for comparing two files
- Bellman-Ford for shortest path routing in networks
- TeX the ancestor of LaTeX
- WASP - Winning and Score Predictor

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds it application in a lot of real life situations.

In programming, Dynamic Programming is a powerful technique that allows one to solve different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time.

Jonathan Paulson explains Dynamic Programming in his amazing Quora answer here.

> Writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper.
> "What's that equal to?"
> Counting "Eight!"
> **Writes down another "1+" on the left.**
> **"What about that?"**
> "Nine!" " How'd you know it was nine so fast?"
> "You just added one more!"
> "So you didn't need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

---

**Dynamic Programming and Recursion:**

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

**The intuition behind dynamic programming is that we trade space for time**, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

> Fibonacci (n) = 1; if n = 0
> Fibonacci (n) = 1; if n = 1
> Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: **1, 1, 2, 3, 5, 8, 13, 21...** and so on!

A code for it using pure recursion:

```
int fib (int n) {
      if (n < 2)
```

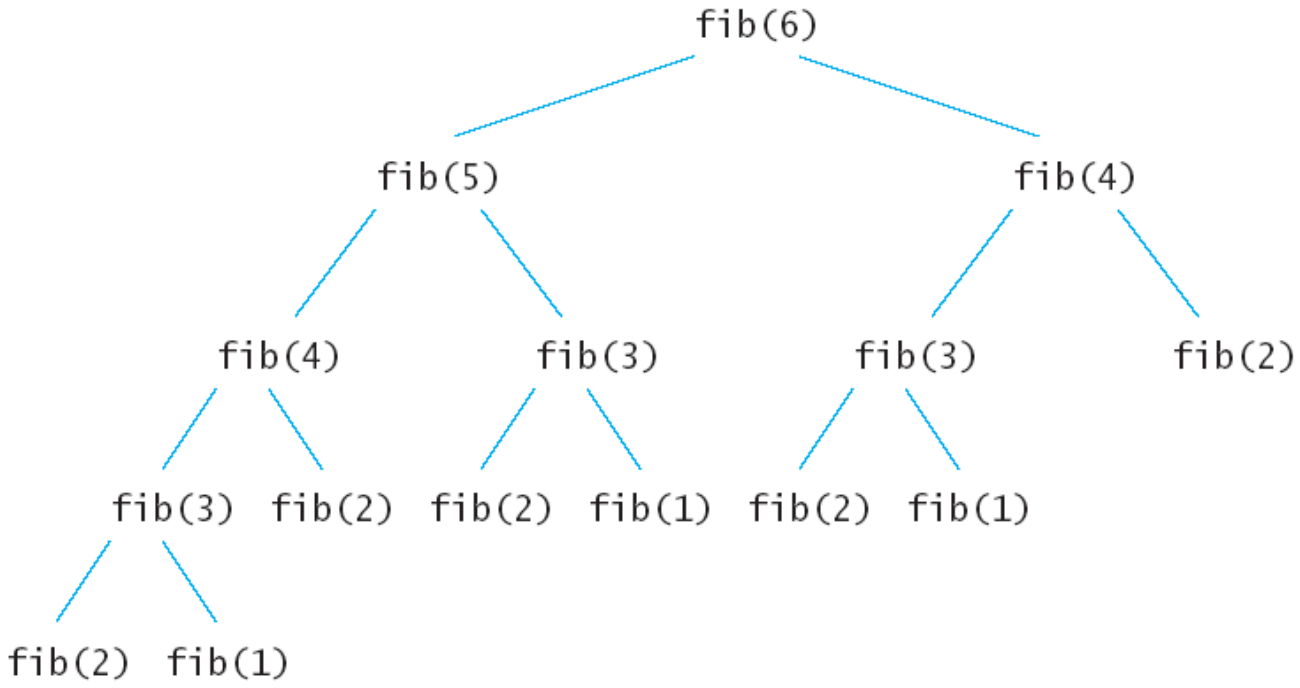
```
            return 1;
        return fib(n-1) + fib(n-2);
    }
```

Using Dynamic Programming approach with memoization:

```
void fib () {
        fibresult[0] = 1;
        fibresult[1] = 1;
        for (int i = 2; i<n; i++)
           fibresult[i] = fibresult[i-1] + fibresult[i-2];
    }
```

Are we using a different recurrence relation in the two codes? No. Are we doing anything different in the two codes? Yes.

In the recursive code, a lot of values are being recalculated multiple times. We could do good with calculating each unique quantity only once. Take a look at the image to understand that how certain values were being recalculated in the recursive way:




---

Majority of the Dynamic Programming problems can be categorized into two types:

**1. Optimization problems.**
**2. Combinatorial problems.**

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems.

- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.

**Bottom up vs. Top Down:**

- **Bottom Up** - I'm going to learn programming. Then, I will start practicing. Then, I will start taking part in contests. Then, I'll practice even more and try to improve. After working hard like crazy, I'll be an amazing coder.
- **Top Down** - I will be an amazing coder. How? I will work hard like crazy. How? I'll practice more and try to improve. How? I'll start taking part in contests. Then? I'll practicing. How? I'm going to learn programming.

Not a great example, but I hope I got my point across. In Top Down, you start building the big solution right away by explaining how you build it from smaller solutions. In Bottom Up, you start with the small solutions and then build up.

Memoization is very easy to code and might be your first line of approach for a while. Though, with dynamic programming, you don't risk blowing stack space, you end up with lots of liberty of when you can throw calculations away. The downside is that you have to come up with an ordering of a solution which works.

**One can think of dynamic programming as a table-filling algorithm: you know the calculations you have to do, so you pick the best order to do them in and ignore the ones you don't have to fill in.**

---

Let's look at a sample problem:

> Let us say that you are given a number **N,** you've to find the number of different ways to write it as the sum of 1, 3 and 4.

For example, if N = 5, the answer would be 6.

- 1 + 1 + 1 + 1 + 1
- 1 + 4
- 4 + 1
- 1 + 1 + 3
- 1 + 3 + 1
- 3 + 1 + 1

**Sub-problem:** $DP_n$ be the number of ways to write **N** as the sum of 1, 3, and 4.
**Finding recurrence:** Consider one possible solution, n = x1 + x2 + ... $x_n$. If the last number is 1, the sum of the remaining numbers should be n - 1. So, number of sums that end with 1 is equal to $DP_{n-1}$. Take other cases into account where the last number is 3 and 4. The final recurrence would be:

$$DP_n = DP_{n-1} + DP_{n-3} + DP_{n-4}.$$

Take care of the base cases. $DP_0 = DP_1 = DP_2 = 1$, and $DP_3 = 2$.

Implementation:

```
DP[0] = DP[1] = DP[2] = 1; DP[3] = 2;
    for (i = 4; i <= n; i++) {
      DP[i] = DP[i-1] + DP[i-3] + DP[i-4];
    }
```

The technique above, takes a bottom up approach and uses memoization to not compute results that have already been computed.

---

I also want to share Michal's amazing answer on Dynamic Programming from Quora.

> "Imagine you have a collection of **N** wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from **1 to N**, respectively. The price of the $i^{th}$ wine is pi. (prices of different wines can be different).
>
> Because the wines get better every year, supposing today is the year 1, on year y the price of the $i^{th}$ wine will be y*pi, i.e. y-times the value that current year.
>
> You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the shelf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).
>
> You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order?"

So, for example, if the prices of the wines are (in the order as they are placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3. The optimal solution would be to sell the wines in the order **p1, p4, p3, p2** for a total profit 1 * 1 + 3 * 2 + 2 * 3 + 4 * 4 = 29.

**Wrong solution!**

After playing with the problem for a while, you'll probably get the feeling, that in the optimal solution you want to sell the expensive wines as late as possible. You can probably come up with the following greedy strategy:

> Every year, sell the cheaper of the two (leftmost and rightmost) available wines.

Although the strategy doesn't mention what to do when the two wines cost the same, this strategy feels right. But unfortunately, it isn't, as the following example demonstrates.

If the prices of the wines are: p1=2, p2=3, p3=5, p4=1, p5=4. The greedy strategy would sell them in the order p1, p2, p5, p4, p3 for a total profit 2 * 1 + 3 * 2 + 4 * 3 + 1 * 4 + 5 * 5 = 49.

But, we can do better if we sell the wines in the order **p1, p5, p4, p2, p3** for a total profit 2 * 1 + 4 * 2 + 1 * 3 + 3 * 4 + 5 * 5 = 50.

?

This counter-example should convince you, that the problem is not so easy as it can look on a first sight and it can be solved using DP.

**How? Write a backtrack.**

When coming up with the memoization solution for a problem, start with a backtrack solution that finds the correct answer. Backtrack solution enumerates all the valid answers for the problem and chooses the best one.

Here are some restrictions on the backtrack solution:

- It should be a function, calculating the answer using recursion.
- It should return the answer with return statement, i.e., not store it somewhere.
- All the non-local variables that the function uses should be used as read-only, i.e. the function can modify only local variables and its arguments.

```
int p[N]; // read-only array of wine prices
// year represents the current year (starts with 1)
// [be, en] represents the interval of the unsold wines on the shelf
    int profit(int year, int be, int en) {
      // there are no more wines on the shelf
      if (be > en)
        return 0;

     // try to sell the leftmost or the rightmost wine, recursively calculate the
      // answer and return the better one
        return max(
        profit(year+1, be+1, en) + year * p[be],
        profit(year+1, be, en-1) + year * p[en]);
}
```

This solution simply tries all the possible valid orders of selling the wines. If there are **N** wines in the beginning, it will try $2^N$ possibilities (each year we have 2 choices). So even though now we get the correct answer, the time complexity of the algorithm grows exponentially.

The correctly written backtrack function should always represent an answer to a well-stated question. In our case profit function represents an answer to a question: "*What is the best profit we can get from selling the wines with prices stored in the array p, when the current year is year and the interval of unsold wines spans through [be, en], inclusive?*"

You should always try to create such a question for your backtrack function to see if you got it right and understand exactly what it does.

We should try to minimize the state space of function arguments. In this step think about, which of the arguments you pass to the function are redundant. Either we can construct them from the other arguments or we don't need them at all. If there are any such arguments, don't pass them to the function. Just calculate them inside the function.

In the above function profit, the argument year is redundant. It is equivalent to the number of wines we have already sold plus one, which is equivalent to the total number of wines from the beginning minus the number of wines we have not sold plus one. If we create a read-only **glob** ?

**variable N,** representing the total number of wines in the beginning, we can rewrite our function as follows:

```
int N; // read-only number of wines in the beginning
    int p[N]; // read-only array of wine prices

        int profit(int be, int en) {
          if (be > en)
            return 0;
          // (en-be+1) is the number of unsold wines
          int year = N - (en-be+1) + 1; // as in the description above
           return max(
            profit(be+1, en) + year * p[be],
            profit(be, en-1) + year * p[en]);
        }
```

We are now 99% done. To transform the backtrack function with time complexity $O(2^N)$ into the memoization solution with time complexity $O(N^2)$, we will use a little trick which doesn't require almost any thinking. As noted above, there are only $O(N^2)$ different arguments our function can be called with. In other words, there are only $O(N^2)$ different things we can actually compute.

So where does $O(2^N)$ time complexity comes from and what does it compute? The answer is - the exponential time complexity comes from the repeated recursion and because of that, it computes the same values again and again. If you run the above code for an arbitrary array of N=20 wines and calculate how many times was the function called for arguments be=10 and en=10 you will get a number 92378.

That's a huge waste of time to compute the same answer that many times. What we can do to improve this is to memoize the values once we have computed them and every time the function asks for an already memoized value, we don't need to run the whole recursion again.

```
int N; // read-only number of wines in the beginning
    int p[N]; // read-only array of wine prices
    int cache[N][N]; // all values initialized to -1 (or anything you
choose)

        int profit(int be, int en) {
          if (be > en)
            return 0;
         // these two lines save the day
          if (cache[be][en] != -1)
            return cache[be][en];
          int year = N - (en-be+1) + 1;
          // when calculating the new answer, don't forget to cache it
          return cache[be][en] = max(
            profit(be+1, en) + year * p[be],
```

```
            profit(be, en-1) + year * p[en]);
        }
```

To sum it up, if you identify that a problem can be solved using DP, try to create a backtrack function that calculates the correct answer. Try to avoid the redundant arguments, minimize the range of possible values of function arguments and also try to optimize the time complexity of one function call (remember, you can treat recursive calls as they would run in O(1) time). Finally, you can memoize the values and don't calculate the same things twice.

---

Read Michal's another cool answer on Dynamic Programming here.

*Contributed by: Prateek Garg*

Algorithms

Rank: **23,338**

View Leaderboard

Topics: 2 Dimensional

# 2 Dimensional

TUTORIAL PROBLEMS

## 1. Introduction

There are many problems in online coding contests which involve finding a minimum-cost path in a grid, finding the number of ways to reach a particular position from a given starting point in a 2-D grid and so on. This post attempts to look at the dynamic programming approach to solve those problems. The problems which will be discussed here are :

1. Finding the Minimum Cost Path in a Grid when a Cost Matrix is given.
2. Finding the number of ways to reach from a starting position to an ending position travelling in specified directions only.
3. Finding the number of ways to reach a particular position in a grid from a starting position (given some cells which are blocked)
4. Edit Distance

---

## 1. Finding Minimum-Cost Path in a 2-D Matrix

> **Problem Statement** : Given a cost matrix Cost[][] where Cost[i][j] denotes the Cost of visiting cell with coordinates (i,j), find a min-cost path to reach a cell (x,y) from cell (0,0) under the condition that you can only travel one step right or one step down. (We assume that all costs are positive integers)

**Solution** : It is very easy to note that if you reach a position (i,j) in the grid, you must have come from one cell higher, i.e. (i-1,j) or from one cell to your left , i.e. (i,j-1). This means that the cost of visiting cell (i,j) will come from the following recurrence relation:

```
MinCost(i,j) = min(MinCost(i-1,j),MinCost(i,j-1)) + Cost[i][j]
```

The above statement means that to reach cell (i,j) wit minimum cost, first reach either cell(i-1,j) or cell (i,j-1) in as minimum cost as possible. From there, jump to cell (i,j). This brings us to the two important conditions which need to be satisfied for a dynamic programming problem:

> **Optimal Sub-structure**:- Optimal solution to a problem involves optimal solutions to sub-problems.
>
> **Overlapping Sub-problems**:- Subproblems once computed can be stored in a table for further use. This saves the time needed to compute the same sub-problems again and again.

(You can google the above two terms for more details)

The problem of finding the min-Cost Path is now almost solved. We now compute the values of the base cases: the topmost row and the leftmost column. For the topmost row, a cell can be reached only from the cell on the left of it. Assuming zero-based index,

```
MinCost(0,j) = MinCost(0,j-1) + Cost[0][j]
```

i.e. cost of reaching cell (0,j) = Cost of reaching cell (0,j-1) + Cost of visiting cell (0,j) Similarly,

```
MinCost(i,0) = MinCost(i-1,0) + Cost[i][0]
```

i.e. cost of reaching cell (i,0) = Cost of reaching cell (i-1,0) + Cost of visiting cell (i,0)

Other values can be computed from them. See the code below for more understanding.

```
#include <bits/stdc++.h>
    using namespace std;
    #define F(i,a,b) for(int i = (int)(a); i <= (int)(b); i++)
    #define RF(i,a,b) for(int i = (int)(a); i >= (int)(b); i--)
    int main()
    {
         int X,Y; //X:number of rows, Y: number of columns
         X = Y = 10; //assuming 10X10 matrix
         int Cost[X][Y];

         F(i,0,X-1)
         {
              F(j,0,Y-1)
              {
                   //Take input the cost of visiting cell (i,j)
                   cin>>Cost[i][j];
              }
         }

         int MinCost[X][Y]; //declare the minCost matrix
```

```
        MinCost[0][0] = Cost[0][0];

        // initialize first row of MinCost matrix
        F(j,1,Y-1)
            MinCost[0][j] = MinCost[0][j-1] + Cost[0][j];

        //Initialize first column of MinCost Matrix
        F(i,1,X-1)
            MinCost[i][0] = MinCost[i-1][0] + Cost[i][0];

        //This bottom-up approach ensures that all the sub-problems
needed
        // have already been calculated.
        F(i,1,X-1)
        {
            F(j,1,Y-1)
            {
                //Calculate cost of visiting (i,j) using the
                //recurrence relation discussed above
                MinCost[i][j] = min(MinCost[i-1][j],MinCost[i][j-1]) +
Cost[i][j];
            }
        }

        cout<<"Minimum cost from (0,0) to (X,Y) is "<<MinCost[X-1][Y-1];
        return 0;
    }
```

Another variant of this problem includes another direction of motion, i.e. one is also allowed to move diagonally lower from cell (i,j) to cell (i+1,j+1). This question can also be solved easily using a slight modification in the recurrence relation. To reach (i,j), we must first reach either (i-1,j), (i,j-1) or (i-1,j-1).

```
MinCost(i,j) = min(MinCost(i-1,j),MinCost(i,j-1),MinCost(i-1,j-1)) +
Cost[i][j]
```

---

## 2. Finding the number of ways to reach from a starting position to an ending position travelling in specified directions only.

> **Problem Statement** : Given a 2-D matrix with M rows and N columns, find the number of ways to reach cell with coordinates (i,j) from starting cell (0,0) under the condition that you can only travel one step right or one step down.

?

**Solution** : This problem is very similar to the previous one. To reach a cell (i,j), one must first reach either the cell (i-1,j) or the cell (i,j-1) and then move one step down or to the right respectively to reach cell (i,j). After convincing yourself that this problem indeed satisfies the optimal sub-structure and overlapping subproblems properties, we try to formulate a bottom-up dynamic programming solution.

We first need to identify the states on which the solution will depend. To find the number of ways to reach to a position, what are the variables on which my answer depends? Here, we need the row and column number to uniquely identify a position. For more details on how to decide the state of a dynamic programming solution, see this : How can one start solving Dynamic Programming problems? Therefore, let NumWays(i,j) be the number of ways to reach position (i,j). As stated above, number of ways to reach cell (i,j) will be equal to the sum of number of ways of reaching (i-1,j) and number of ways of reaching (i,j-1). Thus, we have our recurrence relation as :

```
numWays(i,j) = numWays(i-1,j) + numWays(i,j-1)
```

Now, all you need to do is take care of the base cases and the recurrence relation will calculate the rest for you. :)

The base case, as in the previous question, are the topmost row and leftmost column. Here, each cell in topmost row can be visited in only one way, i.e. from the left cell. Similar is the case for the leftmost column. Hence the code is:

```
#include <bits/stdc++.h>
    using namespace std;
    #define F(i,a,b) for(int i = (int)(a); i <= (int)(b); i++)
    #define RF(i,a,b) for(int i = (int)(a); i >= (int)(b); i--)
    int main()
    {
        int X,Y; //X:number of rows, Y: number of columns
        X = Y = 10; //assuming 10X10 matrix

        int NumWays[X][Y]; //declare the NumWays matrix
        NumWays[0][0] = 1;
        // initialize first row of NumWays matrix
        F(j,1,Y-1)
            NumWays[0][j] = 1;
        //Initialize first column of NumWays Matrix
        F(i,1,X-1)
            NumWays[i][0] = 1;
        //This bottom-up approach ensures that all the sub-problems
needed
        // have already been calculated.
        F(i,1,X-1)

        {
            F(j,1,Y-1)
            {
```

```
                //Calculate number of ways visiting (i,j) using the
                //recurrence relation discussed above
                NumWays[i][j] = NumWays[i-1][j] + NumWays[i][j-1];
            }
        }

        cout<<"Number of ways from(0,0) to (X,Y) is "<<NumWays[X-1][Y-1];
        return 0;
    }
```

---

## 3. Finding the number of ways to reach a particular position in a grid from a starting position (given some cells which are blocked)

> **Problem Statement** : A robot is designed to move on a rectangular grid of M rows and N columns. The robot is initially positioned at (1, 1), i.e., the top-left cell. The robot has to reach the (M, N) grid cell. In a single step, robot can move only to the cells to its immediate east and south directions. That means if the robot is currently at (i, j), it can move to either (i + 1, j) or (i, j + 1) cell, provided the robot does not leave the grid. Now somebody has placed several obstacles in random positions on the grid, through which the robot cannot pass. Given the positions of the blocked cells, your task is to count the number of paths that the robot can take to move from (1, 1) to (M, N).

Input is three integers M, N and P denoting the number of rows, number of columns and number of blocked cells respectively. In the next P lines, each line has exactly 2 integers i and j denoting that the cell (i, j) is blocked.

**Solution** : The code below explains how to proceed with the solution. The problem is same as the previous one, except for few extra checks(due to blocked cells.)

```
#include <bits/stdc++.h>
 using namespace std;
 typedef long long int ll;

 #define F(i,a,b) for(int i = (int)(a); i <= (int)(b); i++)
 #define RF(i,a,b) for(int i = (int)(a); i >= (int)(b); i--)
 #define MOD 1000000007
 int main()
 {
     int M,N,P,_i,_j;

     //Take input the number of rows, columns and blocked cells
     cin>>M>>N>>P;

     //declaring a Grid array which stores the number of paths
     ll Grid[M+1][N+1];
```

```
        //Note that we'll be using 1-based indexing here.
        //initialize all paths initially as 0
        memset(Grid, 0, sizeof(Grid));

        F(i,0,P-1)
        {
            //Take in the blocked cells and mark them with a special
value(-1 here)
            cin>>_i>>_j;
            Grid[_i][_j] = -1;
        }

        // If the initial cell is blocked, there is no way of moving
anywhere
        if(Grid[1][1] == -1)
        {
            printf("0");
            return 0;
        }


        // Initializing the leftmost column
        //Here, If we encounter a blocked cell, there is no way of
visiting any cell
        //directly below it.(therefore the break)
        F(i,1,M)
        {
            if(Grid[i][1] == 0) Grid[i][1] = 1LL;
            else break;
        }


        //Similarly initialize the topmost row.
        F(i,2,N)

        {
            if(Grid[1][i] == 0) Grid[1][i] = 1LL;
            else break;
        }


        //Now the recurrence part
        //The only difference is that if a cell has been marked as -1,
        //simply ignore it and continue to the next iteration.
        F(i,2,M)
        {
            F(j,2,N)
```

```
            {
                if(Grid[i][j] == -1) continue;

                //While adding the number of ways from the left and top
cells,
                //check that they are reachable,i.e. they aren't blocked

                if(Grid[i-1][j] > 0) Grid[i][j] = (Grid[i][j] + Grid[i-
1][j] + MOD)%MOD;
                if(Grid[i][j-1] > 0) Grid[i][j] = (Grid[i][j] + Grid[i]
[j-1] + MOD)%MOD;
            }
        }


        //If the final cell is blocked, output 0, otherwise the answer
        printf("%lld",(Grid[M][N] >= 0 ? Grid[M][N] : 0));
        return 0;
    }
```

---

## Another variant

Finally, we discuss another variant of problems involving grids.

> **Problem Statement** : You are given a 2-D matrix A of n rows and m columns where A[i][j] denotes the calories burnt. Two persons, a boy and a girl, start from two corners of this matrix. The boy starts from cell (1,1) and needs to reach cell (n,m). On the other hand, the girl starts from cell (n,1) and needs to reach (1,m). The boy can move right and down. The girl can move right and up. As they visit a cell, the amount in the cell A[i][j] is added to their total of calories burnt. You have to maximize the sum of total calories burnt by both of them under the condition that they shall meet only in one cell and the cost of this cell shall not be included in either of their total.

**Solution** : Let us analyse this problem in steps:

> The boy can meet the girl in only one cell.

So, let us assume they meet at cell (i,j).

Boy can come in from left or the top, i.e. (i,j-1) or (i-1,j). Now he can move right or down.That is, the sequence for the boy can be:

```
(i,j-1)-->(i,j)-->(i,j+1)
(i,j-1)-->(i,j)-->(i+1,j)
(i-1,j)-->(i,j)-->(i,j+1)
(i-1,j)-->(i,j)-->(i+1,j)
```

?

Similarly, the girl can come in from the left or bottom, i.e. (i,j-1) or (i+1,j) and she can go up or right. The sequence for girl's movement can be:

```
(i,j-1)-->(i,j)-->(i,j+1)
(i,j-1)-->(i,j)-->(i-1,j)
(i+1,j)-->(i,j)-->(i,j+1)
(i+1,j)-->(i,j)-->(i-1,j)
```

Comparing the 4 sequences of the boy and the girl, the boy and girl meet only at one position (i,j), iff

```
Boy: (i,j-1)-->(i,j)-->(i,j+1) and Girl: (i+1,j)-->(i,j)-->(i-1,j)
```

or

```
Boy: (i-1,j)-->(i,j)-->(i+1,j) and Girl:  (i,j-1)-->(i,j)-->(i,j+1)
```

Convince yourself that in no other case will they meet at only one position.

Now, we can solve the problem by creating 4 tables:

1. Boy's journey from start (1,1) to meeting cell (i,j)
2. Boy's journey from meeting cell (i,j) to end (n,m)
3. Girl's journey from start (n,1) to meeting cell (i,j)
4. Girl's journey from meeting cell (i,j) to end (1,n)

The meeting cell can range from `2<= i <= n-1 and 2 <= j <= m-1`

See the code below for more details:

```
#include <bits/stdc++.h>
    #define F(i,a,b) for(int i = (int)(a); i <= (int)(b); i++)
    #define RF(i,a,b) for(int i = (int)(a); i >= (int)(b); i--)
    #define MAX 1005
    int Boy1[MAX][MAX];
    int Boy2[MAX][MAX];
    int Girl1[MAX][MAX];
    int Girl2[MAX][MAX];
    using namespace std;
    int main()
    {
        int N,M,ans,op1,op2;
        scanf("%d%d",&N,&M);
        int Workout[MAX][MAX];
        ans = 0;

        //Take input the calories burnt matrix
        F(i,1,N)
```

```
            F(j,1,M)
                scanf("%d",&Workout[i][j]);

        //Table for Boy's journey from start to meeting cell
        F(i,1,N)
            F(j,1,M)
                Boy1[i][j] = max(Boy1[i-1][j],Boy1[i][j-1]) + Workout[i]
[j];

        //Table for boy's journey from end to meet cell
        RF(i,N,1)
            RF(j,M,1)
                Boy2[i][j] = max(Boy2[i+1][j],Boy2[i][j+1]) + Workout[i]
[j];


        //Table for girl's journey from start to meeting cell
        RF(i,N,1)
            F(j,1,M)
                Girl1[i][j] = max(Girl1[i+1][j],Girl1[i][j-1]) +
Workout[i][j];


        //Table for girl's journey from end to meeting cell
        F(i,1,N)
            RF(j,M,1)
                Girl2[i][j] = max(Girl2[i-1][j],Girl2[i][j+1]) +
Workout[i][j];


        //Now iterate over all meeting positions (i,j)
        F(i,2,N-1)
        {
            F(j,2,M-1)
            {
                //For the option 1
                op1 = Boy1[i][j-1] + Boy2[i][j+1] + Girl1[i+1][j] +

Girl2[i-1][j];

                //For the option 2
                op2 = Boy1[i-1][j] + Boy2[i+1][j] + Girl1[i][j-1] +
Girl2[i][j+1];

                //Take the maximum of two options at each position
                ans = max(ans,max(op1,op2));
            }
        }
```

```
        printf("%d",ans);
        return 0;
    }
```

---

## 4. Edit Distance

Edit distance is a way of quantifying how dissimilar two strings are, i.e., how many operations (add, delete or replace character) it would take to transform one string to the other. This is one of the most common variants of edit distance, also called Levenshtein distance, named after Soviet computer scientist, Vladimir Levenshtein. There are 3 operations which can be applied to either string, namely: insertion, deletion and replacement.

```
 int editDistance(string s1, string s2) {
        int m = s1.size();
        int n = s2.size();
        // for all i, j, dp[i][j] will hold the edit distance between the
first
        // i characters of source string and first j characters of target
string
        int dp[m + 1][n + 1];
        memset(dp, 0, sizeof(dp));
        // source can be transformed into target prefix by inserting
        // all of the characters in the prefix
        for (int i = 1; i <= n; i++) {
            dp[0][i] = i;
        }
        // source prefixes can be transformed into empty string by
        // by deleting all of the characters
        for (int i = 1; i <= m; i++) {
            dp[i][0] = i;
        }
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (s1[i - 1] == s2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1]; // no operation
required as characters are the same
                }
                else {
                    dp[i][j] = 1 + min(dp[i - 1][j - 1],     //
substitution
                                       min(dp[i][j - 1],      // insertion
                                           dp[i - 1][j]));     // deletion
                }
            }
        }
```

```
        return dp[m][n];
    }
```

Let's look at the DP table when s1 = "sitting" (source string)
s2 = "kitten" (target string)

| | | k | i | t | t | e | n |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| s | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| i | 2 | 2 | 1 | 2 | 3 | 4 | 5 |
| t | 3 | 3 | 2 | 1 | 2 | 3 | 4 |
| t | 4 | 4 | 3 | 2 | 1 | 2 | 3 |
| i | 5 | 5 | 4 | 3 | 2 | 2 | 3 |
| n | 6 | 6 | 5 | 4 | 3 | 3 | 2 |
| g | 7 | 7 | 6 | 5 | 4 | 4 | 3 |

Dynamic Programming is not an algorithm or data-structure. It is a technique and it is applied to a certain class of problems. The key to figure, if a problem can be solved by DP, comes by practice.

*Contributed by: Prateek Garg*

**TEST YOUR UNDERSTANDING**

## Matrix Sum

You are given a $N * M$ matrix. You need to print the sum of all the numbers in the rectangle which has $(1, 1)$ as the top left corner and $(X, Y)$ as the bottom right corner.

**Input:**
First line contains two integers, *N* and *M*, number of rows and number of columns in the matrix.
Next *N* lines contains *M* space separated integers, elements of the matrix.
Next line contains an integer, *Q*, number of queries.
Each query contains two space separated integers, *X* and *Y*.

**Output:**
For each query, print the sum of all the numbers in the rectangle which has $(1, 1)$ as the top left corner and $(X, Y)$ as the bottom right corner.

**Constraints:**
$1 \leq N, M \leq 10^3$
$1 \leq$ elements of matrix $\leq 10^3$

?

Algorithms

Rank: **23,338**

View Leaderboard

Topics: State space reduction

# State space reduction

TUTORIAL PROBLEMS

In problems for which dynamic programming solutions are considered, there is a concept of a state. A state is, in general, a point in a $d$-dimensional space, where $d$ is called the number of dimensions in the solution. This may sound quite formal, but in fact, each person who solved at least one problem using dynamic programming approach used this concept.

For example, in **Longest Common Subsequence** problem, also known as **LCS**, the goal is for given two strings $S[1..n]$ and $T[1..m]$, to find the length of their longest common subsequence. This is a classical problem which can be solved efficiently using dynamic programming approach. The idea is to compute $f[i][j]$, which is the length of the longest common subsequence of $S[1..i]$ and $T[1..j]$. The value of $f[i][j]$ can be computed in this problem knowing the values of $f[i-1][j]$, $f[i][j-1]$ and $f[i-1][j-1]$. Then the final answer is the value of $f[n][m]$. In this approach, an entry in $f$ table is called a state, and since the table is $2$-dimensional, this dynamic programming solution is $2$-dimensional.

The above problem is very classical, so the above approach is well known and it can be shown that it is optimal. In other words, in a general case the above $O(n \cdot m)$ solution with $O(n \cdot m)$ states is optimal. However, when a new problem has to be solved from the scratch, the common situation is that at first some correct dynamic programming approach is defined, just to check if the problem can be solved correctly with this method. After that, the next step is to try to reduce the time complexity of the solution if it is too slow.

The time complexity of a dynamic programming approach can be improved in many ways. The most common are to either use some kind of data structure like a segment tree to speed up the computation of a single state or trying to reduce the number of states needed to solve the problem. Describing the latter is the main goal of this article.

**Reducing the number of states**

?

The number of states in a dynamic programming solution can be reduced in a few ways. One possible way is to try to reduce the size of one of existing dimensions. When the size is reduced by a factor of $2$, then the overall time complexity of the solution remains the same, but the constant factor is improved. The most significant improvement can be achieved by getting rid of a dimension completely or reducing its size to a small constant.

The actual method of reducing the state space is very problem-specific. There is no general method, which can be applied to all the problems. However, one general idea worth to mention is that the number of dimensions can be reduced if there are some dimensions capturing the same information, or if there is a dimension capturing irrelevant information. Studying more and more war stories is the best approach to learn how to improve existing dynamic programming solutions.

In the following section, an example problem will be given, for which the initial solution will be improved.

**Problem:**

For a given array of distinct $n$ integers $A[1..n]$ and two integers $X$ and $Y$, the goal is to find the number of subsequences of $A$ with exactly $X$ local minimums and $Y$ local maximums. For a subsequence $A[i_1], A[i_2], \ldots, A[i_k]$ of $A$, element $A[i_j]$ for $1 < i_j < k$ is a local minimum if and only if $A[i_{j-1}] > A[i_j] < A[i_{j+1}]$. Similarly, $A[i_j]$ is a local maximum if and only if $A[i_{j-1}] < A[i_j] > A[i_{j+1}]$.

Since subsequences can be build incrementally, dynamic programming approach to this problem sounds promising.

**Initial solution**

Let $dp[i][x][y][0]$ be the number of subsequences of $A[1..i]$ ending in $A[i]$ with exactly $x$ local minimums, exactly $y$ local maximums and the last element smaller than the one before last element taken to a subsequence.

Similarly, let $dp[i][x][y][1]$ be the number of subsequences of $A[1..i]$ ending in $A[i]$ with exactly $x$ local minimums, exactly $y$ local maximums and the last element greater than the one before last element taken to a subsequence.

With these values defined, any entry of $dp$ table with the $i$ as the value of its first dimension can be computed using $dp$ entries with values of first dimension smaller than $i$.

**Details of computing a single state**

Let $i$ be the fixed value of the first dimension of a $dp$ state. In order to update any value for such a state, a possible method is to iterate over all indices $1 \leq j < i$, and for a fixed $j$, consider two possibilities: either $A[j] < A[i]$ or $A[j] > A[i]$. Since all elements of $A$ are distinct, equality is not possible here.

If $A[j] < A[i]$, a subsequence of length $2$ can be created using just $A[j]$ and $A[i]$, so this subsequence can be added to $dp[i][0][0][1]$. Similarly, if $A[j] > A[i]$, such a subsequence can be added to $dp[i][0][0][0]$.

Next step is to extend any previously counted subsequences of length at least $2$ to subsequenc ending with $A[i]$. In order to do this, let $0 \leq x \leq X$ be the fixed number of local minimums and

$0 \leq y \leq Y$ be a fixed number of local maximums. Again, there are two cases to consider:

If $A[j] < A[i]$, a value of $dp[j][x][y][1]$ can be added to $dp[i][x][y][1]$. Moreover, if $x < X$, a value of $dp[j][x][y][0]$ can be added to $dp[i][x+1][y][1]$ creating a new local minimum at index $j$.

Analogously, if $A[j] > A[i]$, a value of $dp[j][x][y][0]$ can be added to $dp[i][x][y][0]$. Moreover, if $y < Y$, a value of $dp[j][x][y][1]$ can be added to $dp[i][x][y+1][0]$ creating a new local maximum at index $j$.

In order to compute the final result, all values $dp[i][X][Y][0]$ and $dp[i][X][Y][0]$ over all $1 \leq i \leq N$ have to be summed. At the end, if $X = 0$ and $Y = 0$, a value of $N$ have to be added to the result in order to count all subsequences of length $1$.

The total time complexity of this method is $O(N^2 \cdot X \cdot Y)$, because $O(N \cdot X \cdot Y)$ $dp$ entries are filled, and a single one is filed in $O(N)$ time.

**Improving the initial solution by reducing the state space**

In the above solution, the state space can be reduced. The basic idea here is based on the fact that input values in array $A$ are distinct.

Let $P$ be any subsequence of $A$ and let $D$ be the difference between local minimums in $P$ and local maximums in it. The crucial observation here is that since all elements of $A$ are distinct, $D$ can be only equal to $-1$, $0$ or $1$.

In order to prove that, let $P$ be a subsequence of $A$ with $1$ local minimum and $0$ local maximums. Let $i$ be the index which is the local minimum. The important fact is that $P$ is an increasing subsequence from index $i$, so any other subsequence $W$ which have $P$ as a prefix will be either increasing from index $i$ till the end, or it will have two adjacent indexes $i_j$ and $i_{j+1}$ such that $A[i_j] > A[i_{j+1}]$, so the value $D$ for $W$ never grows over $1$. The same argument can be used to show why the value of $D$ is never less than $-1$.

This fact can be used to reduce the state space in the original solution, which leads to speed up the overall solution. After the improvement, $dp$ non-constant size dimensions has been reduced from $3$ to $2$. In other words, for each state with $0 \leq x \leq X$ local minimums there are at most $3$ possible values of $y$ - the number of local maximums: the only possible ones are $x-1$, $x$ and $x+1$. This observation leads to a solution with total time complexity $O(N^2 \cdot X)$.

This war story shows how the idea of removing one of two dimensions capturing the same information can be used by a deep analysis of the problem.

It is worth mentioning that even the above faster solution can be improved further using a data structure like a segment tree in order to speed up computing a single state of the solution.

*Contributed by: pkacprzak*

?

Algorithms

Rank: 23,339

View Leaderboard

Topics: Dynamic Programming and Bit Masking

# Dynamic Programming and Bit Masking

TUTORIAL PROBLEMS

First thing to make sure before using bitmasks for solving a problem is that it must be having small constraints, as solutions which use bitmasking generally take up exponential time and memory.

Let's first try to understand what Bitmask means. Mask in Bitmask means hiding something. Bitmask is nothing but a binary number that represents something. Let's take an example. Consider the set $A = \{1, 2, 3, 4, 5\}$. We can represent any subset of $A$ using a bitmask of length $5$, with an assumption that if $i^{th} (0 \leq i \leq 4)$ bit is set then it means $i^{th}$ element is present in subset. So the bitmask $01010$ represents the subset $\{2, 4\}$

Now the benefit of using bitmask. We can set the $i^{th}$ bit, unset the $i^{th}$ bit, check if $i^{th}$ bit is set in just one step each. Let's say the bitmask, $b = 01010$.

**Set the $i^{th}$ bit:** $b|(1 << i)$. Let $i = 0$, so,

$$(1 << i) = 00001$$

$$01010|00001 = 01011$$

So now the subset includes the $0^{th}$ element also, so the subset is $\{1, 2, 4\}$.

**Unset the $i^{th}$ bit:** $b\&!(1 << i)$. Let $i = 1$, so,

$$(1 << i) = 00010$$

$$!(1 << i) = 11101$$

$$01010\&11101 = 01000$$

Now the subset does not include the $1^{st}$ element, so the subset is $\{4\}$.

**Check if $i^{th}$ bit is set:** $b\&(1 << i)$, doing this operation, if $i^{th}$ bit is set, we get a non zero integer otherwise, we get zero. Let $i = 3$

$$(1 << i) = 01000$$

$$01010\&01000 = 01000$$

Clearly the result is non-zero, so that means $3^{rd}$ element is present in the subset.

Let's take a problem, given a set, count how many subsets have sum of elements greater than or equal to a given value.

Algorithm is simple:

```
solve(set, set_size, val)
    count = 0
    for x = 0 to power(2, set_size)
        sum = 0
        for k = 0 to set_size
            if kth bit is set in x
                sum = sum + set[k]
        if sum >= val
             count = count + 1
    return count
```

To iterate over all the subsets we are going to each number from 0 to $2^{set\_size}-1$
The above problem simply uses bitmask and complexity is $O(2^n n)$.

Now, let's take another problem that uses dynamic programming along with bitmasks.

**Assignment Problem:**
There are $N$ persons and $N$ tasks, each task is to be alloted to a single person. We are also given a matrix $cost$ of size $N \times N$, where $cost[i][j]$ denotes, how much person $i$ is going to charge for task $j$. Now we need to assign each task to a person in such a way that the total cost is minimum. Note that each task is to be alloted to a single person, and each person will be alloted only one task.

The brute force approach here is to try every possible assignment. Algorithm is given below:

```
assign(N,  cost)
    for i = 0 to N
        assignment[i] = i              //assigning task i to person i
    res = INFINITY
    for j = 0 to factorial(N)
        total_cost = 0
        for i = 0 to N
            total_cost = total_cost + cost[i][assignment[i]]
        res = min(res, total_cost)
        generate_next_greater_permutation(assignment)
    return res
```

The complexity of above algorithm is $O(N!)$, well that's clearly not good.

Let's try to improve it using dynamic programming. Suppose the state of $dp$ is $(k, mask)$, where $k$ represents that person $0$ to $k-1$ have been assigned a task, and $mask$ is a binary number, whose $i^{th}$ bit represents if the $i^{th}$ task has been assigned or not.
Now, suppose, we have $answer(k, mask)$, we can assign a task $i$ to person $k$, iff $i^{th}$ task is not yet assigned to any peron i.e. $mask \& (1 << i) = 0$ then, $answer(k+1, mask|(1 << i))$ will be given as:

$$answer(k+1, mask|(1 << i)) = min(answer(k+1, mask|(1 << i)), answer(k, mask) + co \quad ?\ i])$$

One thing to note here is $k$ is always equal to the number set bits in $mask$, so we can remove that. So the dp state now is just $(mask)$, and if we have $answer(mask)$, then

$$answer(mask|(1<<i)) = min(answer(mask|(1<<i)), answer(mask) + cost[x][i])$$

here $x$ =number of set bits in $mask$.
Complete algorithm is given below:

```
assign(N, cost)
    for i = 0 to power(2,N)
        dp[i] = INFINITY
    dp[0] = 0
    for mask = 0 to power(2, N)
        x = count_set_bits(mask)
        for j = 0 to N
            if jth bit is not set in i
                dp[mask|(1<<j)] = min(dp[mask|(1<<j)], dp[mask]+cost[x][j])
    return dp[power(2,N)-1]
```

Time complexity of above algorithm is $O(2^n n)$ and space complexity is $O(2^n)$.
This is just one problem that can be solved using DP+bitmasking. There's a whole lot.
Let's go to another problem, suppose we are given a graph and we want to find out if it contains a Hamiltonian Path. This problem can also be solved using DP+bitmasking in $O(2^n n^2)$ time complexity and $O(2^n n)$ space complexity. Here's a link to it's solution.

*Contributed by: Vaibhav Jaimini*