Introduction to Dynamic Programming

Florian Gatignon

November 4, 2018

Swiss Olympiad in Informatics

Table of Contents

Introduction

What is dynamic programming?

What is dynamic programming?

Dynamic programming is. . .

What is dynamic programming?

Dynamic programming is. . .

- **not** an algorithm

What is dynamic programming?

Dynamic programming is. . .

- **not** an algorithm
- a technique

What is dynamic programming?

Dynamic programming is. . .

- **not** an algorithm
- a technique for solving problems (in particular optimization problems)

What is dynamic programming?

Dynamic programming is. . .

- **not** an algorithm
- a technique for solving problems (in particular optimization problems) more efficiently.

What is dynamic programming: A guideline

Keep it simple, stupid!

What is dynamic programming: A guideline

Keep it simple, stupid!

- Doing the same thing twice is bad.

What is dynamic programming: A guideline

Keep it simple, stupid!

- Doing the same thing twice is bad.
- When you use the same code more than once, you use a function.

What is dynamic programming: A guideline

Keep it simple, stupid!

- Doing the same thing twice is bad.
- When you use the same code more than once, you use a function.
- Your program should also be lazy and avoid to compute what it has already computed.

What is dynamic programming

DP is a technique that you may use when you can divide a problem into subproblems and build the full solution using the partial solutions, but the subproblems overlap and you end up solving the same subproblems over and over again.

What is dynamic programming

DP is a technique that you may use when you can divide a problem into subproblems and build the full solution using the partial solutions, but the subproblems overlap and you end up solving the same subproblems over and over again.
A dynamic program avoids this problem by **remembering** what it has already done and not computing it again.

A simple example: Fibonacci's sequence

Fibonacci's sequence

Fibonacci's sequence can be defined as follows:

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ \forall n \in \mathbb{N} \setminus \{0, 1\}, f_n = f_{n-1} + f_{n-2} \end{cases}$$

Fibonacci's sequence

Fibonacci's sequence can be defined as follows:

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ \forall n \in \mathbb{N} \setminus \{0, 1\}, f_n = f_{n-1} + f_{n-2} \end{cases}$$

Here are a few first values of the sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Fibonacci's sequence: Intuitive algorithm

An intuitive way of computing values of the sequence would be:

```
int fib(int n) {
    if(n==0) return 0;
    if(n==1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Fibonacci's sequence: The problem

If you try to run this code to compute f_{100}, you'd have to be **very** patient to get an answer.

Fibonacci's sequence: The problem

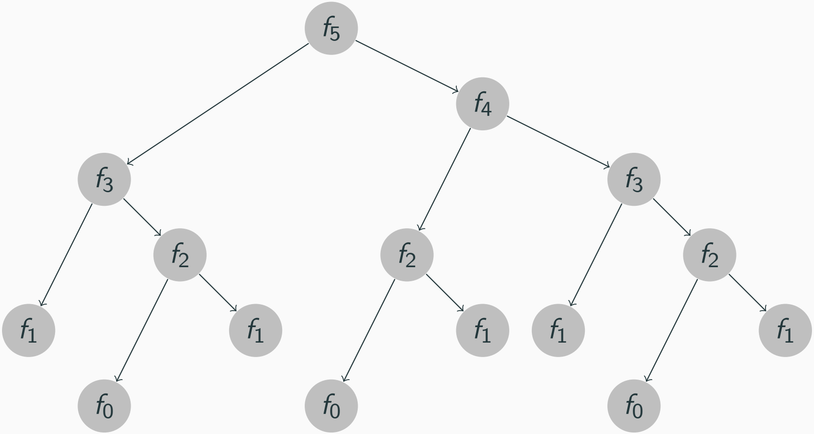
If you try to run this code to compute f_{100}, you'd have to be **very** patient to get an answer.

Why?

Fibonacci's sequence: The problem

Our program computes the same values over and over.

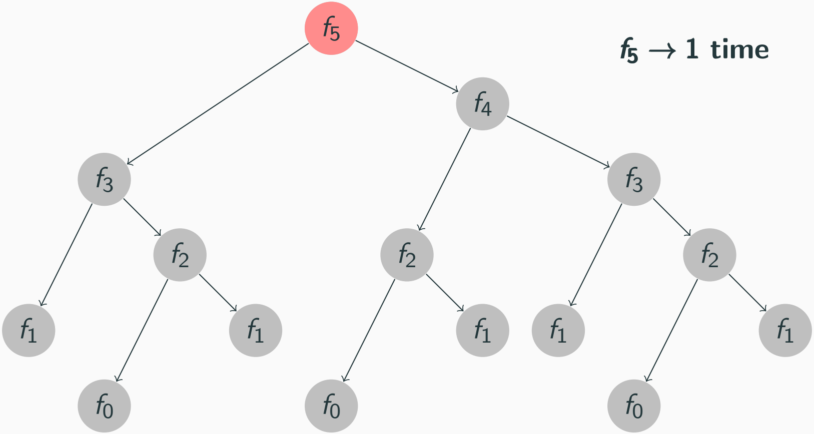
Look at this tree:



Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

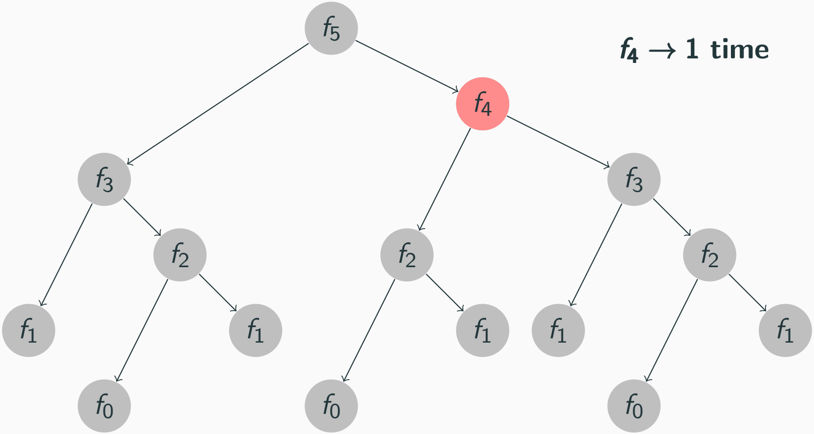


$\boldsymbol{f_5 \to 1}$ **time**

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

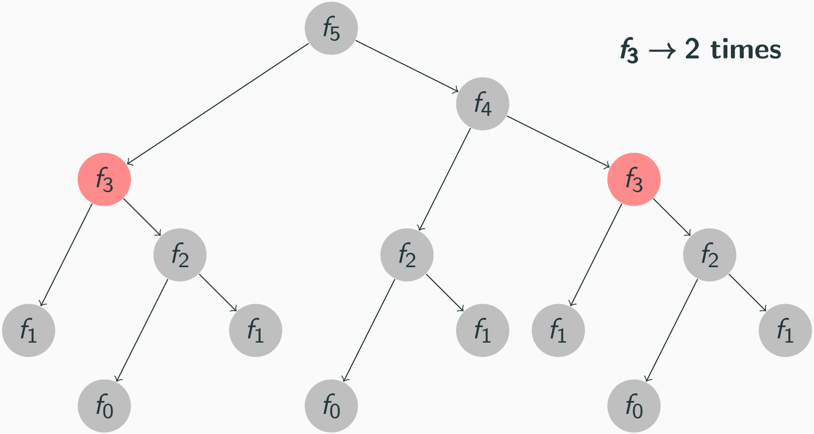


$f_4 \to$ 1 time

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

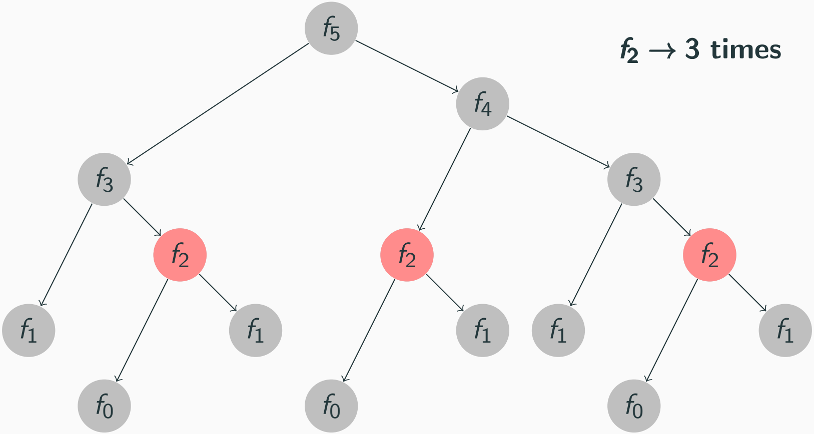


$\boldsymbol{f_3} \rightarrow$ **2 times**

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

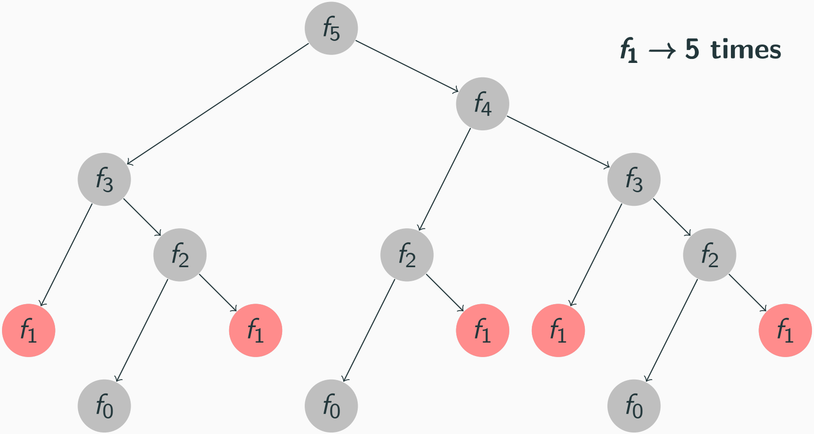


$\boldsymbol{f_2 \rightarrow 3}$ **times**

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

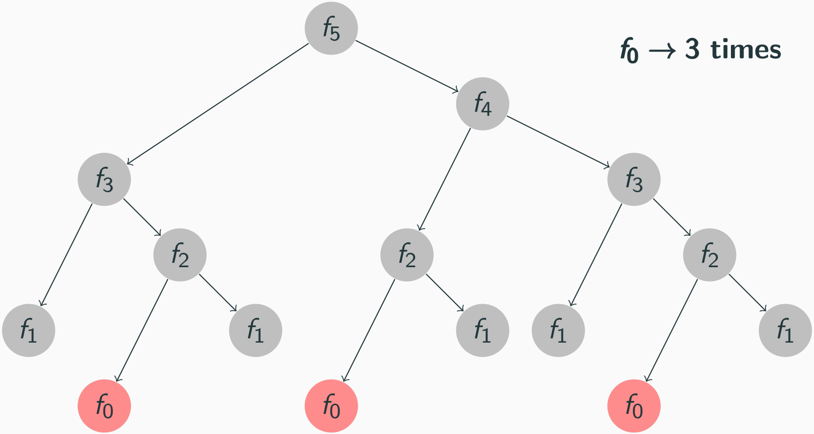


$\boldsymbol{f_1} \rightarrow$ **5 times**

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:



$f_0 \rightarrow$ 3 times

Fibonacci's sequence: The problem

This example shows us that the number of operations increases at the same speed as the results (the number of times we compute each value are all increasing values of Fibonacci's sequence)!

Fibonacci's sequence: The problem

This example shows us that the number of operations increases at the same speed as the results (the number of times we compute each value are all increasing values of Fibonacci's sequence)!
The Fibonacci sequence grows exponentially, so we have an exponential running time...

Fibonacci's sequence: The solution

We can do (much) better.

Fibonacci's sequence: The solution

We can do (much) better. We don't need to compute anything twice:

Fibonacci's sequence: The solution

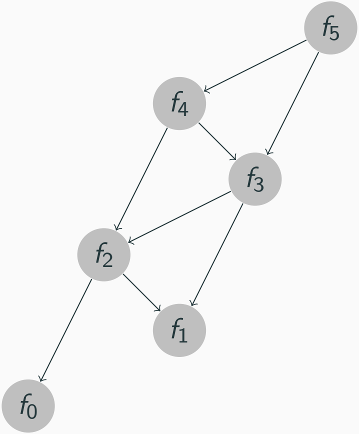
We can do (much) better. We don't need to compute anything twice:
Just remember the previous values!

Fibonacci's sequence: The solution

We can do (much) better. We don't need to compute anything twice:
Just remember the previous values! We can first compute lower values and then combine them to get the next one.

Fibonacci's sequence: The solution

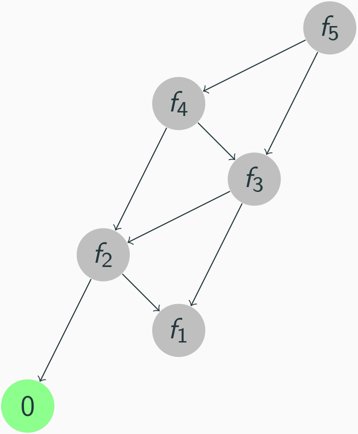
We can use already computed values to build the higher ones in order:



Fibonacci's sequence: The solution

We can use already computed values to build the higher ones in order:

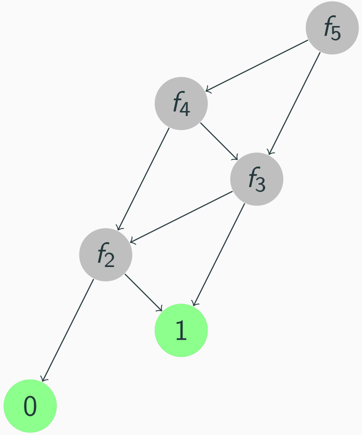
$$\boldsymbol{f_0 = 0}$$



Fibonacci's sequence: The solution

We can use already computed values to build the higher ones in order:

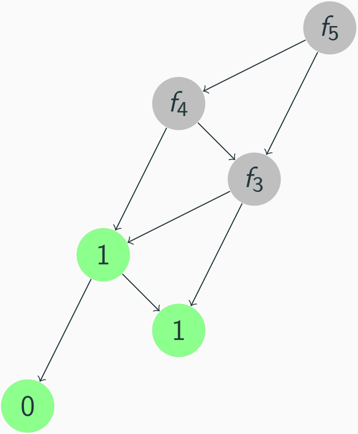
$$f_1 = 1$$



Fibonacci's sequence: The solution

We can use already computed values to build the higher ones in order:

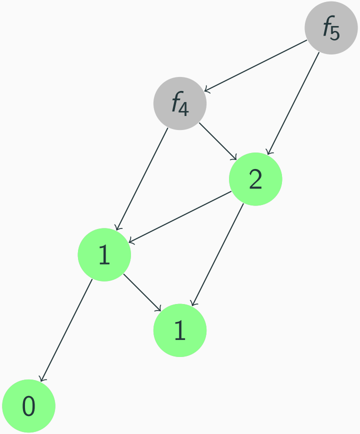
$$\boldsymbol{f_2 = f_1 + f_0}$$



Fibonacci's sequence: The solution

We can use already computed values to build the higher ones in order:

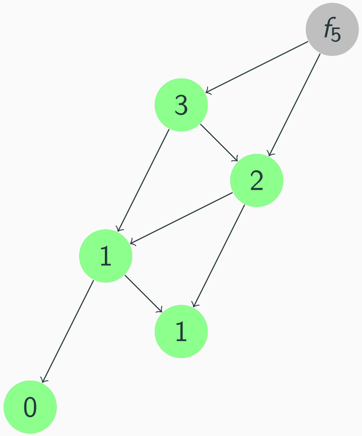
$$\boldsymbol{f_3 = f_2 + f_1}$$



Fibonacci's sequence: The solution

We can use already computed values to build the higher ones in order:

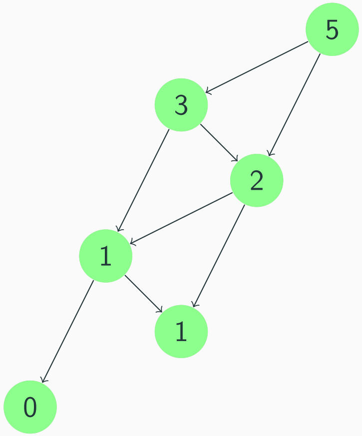
$$f_4 = f_3 + f_2$$



Fibonacci's sequence: The solution

We can use already computed values to build the higher ones in order:

$$f_5 = f_4 + f_3$$



Fibonacci's sequence: The solution

We compute all values (once) from f_0 up to f_n:

```
int fib(int n) {
    vector<int> v;
    v.push_back(0);
    v.push_back(1);
    for(int i = 2; i <= n; i++) {
        v.push_back(v[v.size()-1] + v[v.size()-2]);
    }
    return v[n];
}
```

Fibonacci's sequence: The solution

We compute all values (once) from f_0 up to f_n:

```
int fib(int n) {
    vector<int> v;
    v.push_back(0);
    v.push_back(1);
    for(int i = 2; i <= n; i++) {
        v.push_back(v[v.size()-1] + v[v.size()-2]);
    }
    return v[n];
}
```

Our running time is now down to...

Fibonacci's sequence: The solution

We compute all values (once) from f_0 up to f_n:

```
int fib(int n) {
    vector<int> v;
    v.push_back(0);
    v.push_back(1);
    for(int i = 2; i <= n; i++) {
        v.push_back(v[v.size()-1] + v[v.size()-2]);
    }
    return v[n];
}
```

Our running time is now down to $\mathcal{O}(n)$.

Fibonacci's sequence: bonus solution

Bonus solution: you don't need $\mathcal{O}(n)$ space.

```
int fib(int n) {
    int a = 0, b = 1;
    for(int i = 0; i < n; i++) {
        swap(a,b);
        b += a;
    }
    return a;
}
```

Fibonacci's sequence: bonus solution

Bonus solution: you don't need $\mathcal{O}(n)$ space.

```
int fib(int n) {
    int a = 0, b = 1;
    for(int i = 0; i < n; i++) {
        swap(a,b);
        b += a;
    }
    return a;
}
```

There's an even better solution in $\mathcal{O}(\log(n))$, but it is not in our scope for today.

The recipe for creating a good DP solution

The recipe for creating a good DP solution

This was a simple example, but the same schemata apply to much more complicated problems. We shall now generalize what we've learned from Fibonacci's sequence and apply it to other problems.

DP's four steps

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.
2. Find a general recurrence formula to solve a subproblem using the solution to other subproblems.

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.
2. Find a general recurrence formula to solve a subproblem using the solution to other subproblems.
3. Find base case(s).

DP's four steps

Here is a classic method of thinking about dynamic programming, using four basic steps.

Think first, code second!

1. Define subproblems.
2. Find a general recurrence formula to solve a subproblem using the solution to other subproblems.
3. Find base case(s).
4. Which is the relevant subproblem?

DP's four steps and Fibonacci's sequence

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems:

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems: f_i.

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems: f_i.
2. General formula:

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems: f_i.
2. General formula: $f_i = f_{i-1} + f_{i-2}$.

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems: f_i.
2. General formula: $f_i = f_{i-1} + f_{i-2}$.
3. Base cases:

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems: f_i.
2. General formula: $f_i = f_{i-1} + f_{i-2}$.
3. Base cases: $f_0 = 0, f_1 = 1$.

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems: f_i.
2. General formula: $f_i = f_{i-1} + f_{i-2}$.
3. Base cases: $f_0 = 0, f_1 = 1$.
4. Relevant suproblem:

DP's four steps and Fibonacci's sequence

How does our solution for Fibonacci's sequence match our four steps?

1. Suproblems: f_i.
2. General formula: $f_i = f_{i-1} + f_{i-2}$.
3. Base cases: $f_0 = 0, f_1 = 1$.
4. Relevant suproblem: f_n.

How to implement a DP solution

Subproblem ordering

— Okay, I've followed your four steps. How do I use this to code a solution now?

Subproblem ordering

Most subproblems can be solved only using other subproblems.

Subproblem ordering

Most subproblems can be solved only using other subproblems.

In what order can we compute subproblems?

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.
3. Continue computing further subproblems which are now solvable.

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.
3. Continue computing further subproblems which are now solvable.
4. When the relevant subproblem is found, return the result!

Subproblem ordering

Most subproblems can be solved only using other subproblems.
In what order can we compute subproblems?

1. Start with the base cases. We know the answer for those.
2. Compute other subproblems which only need base cases.
3. Continue computing further subproblems which are now solvable.
4. When the relevant subproblem is found, return the result!

This is the hardest part of most difficult dynamic programming problems. Sometimes, a viable ordering is obvious, sometimes it is not; the best way to get used to it is to solve a lot of this kind of problems.

Another example: Renovate

Renovate: Task statement

The problem is the following:

Renovate: Task statement

The problem is the following:

- You're given a $n \times m$ rectangle "wall" made of zeroes and ones as input.

Renovate: Task statement

The problem is the following:

- You're given a $n \times m$ rectangle "wall" made of zeroes and ones as input.
- Your task is to renovate the wall.

Renovate: Task statement

The problem is the following:

- You're given a $n \times m$ rectangle "wall" made of zeroes and ones as input.
- Your task is to renovate the wall.
- When there's a one at some coordinate, the wall is in good shape at this position, so you leave it as it is.

Renovate: Task statement

The problem is the following:

- You're given a $n \times m$ rectangle "wall" made of zeroes and ones as input.
- Your task is to renovate the wall.
- When there's a one at some coordinate, the wall is in good shape at this position, so you leave it as it is.
- When there's a zero, this section of the wall is holey, and you have two choices: turn it into a window, or rebuild a solid wall at this place.

Renovate: Task statement

The problem is the following:

- You're given a $n \times m$ rectangle "wall" made of zeroes and ones as input.
- Your task is to renovate the wall.
- When there's a one at some coordinate, the wall is in good shape at this position, so you leave it as it is.
- When there's a zero, this section of the wall is holey, and you have two choices: turn it into a window, or rebuild a solid wall at this place.
- Every window needs a full column of solid walls on both its left and its right in order not to affect the stability of the wall.

Renovate: Task statement

The problem is the following:

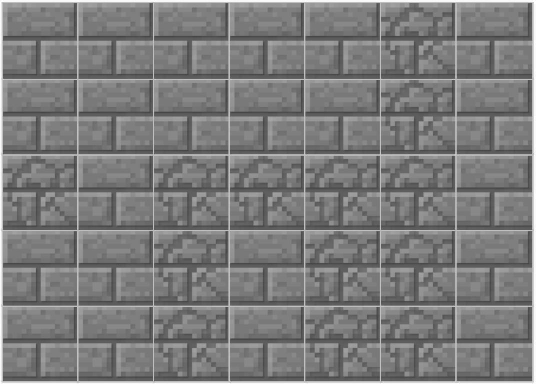
- You're given a $n \times m$ rectangle "wall" made of zeroes and ones as input.
- Your task is to renovate the wall.
- When there's a one at some coordinate, the wall is in good shape at this position, so you leave it as it is.
- When there's a zero, this section of the wall is holey, and you have two choices: turn it into a window, or rebuild a solid wall at this place.
- Every window needs a full column of solid walls on both its left and its right in order not to affect the stability of the wall.
- Maximize the number of windows!

Renovate: Sample case

1	1	1	1	1	0	1
1	1	1	1	1	0	1
0	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	0	1	0	0	1

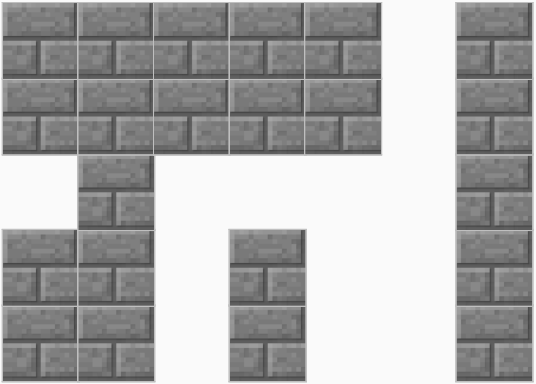
This is our input. Zeroes are broken parts of the wall and ones are solid parts of the wall.

Renovate: Sample case



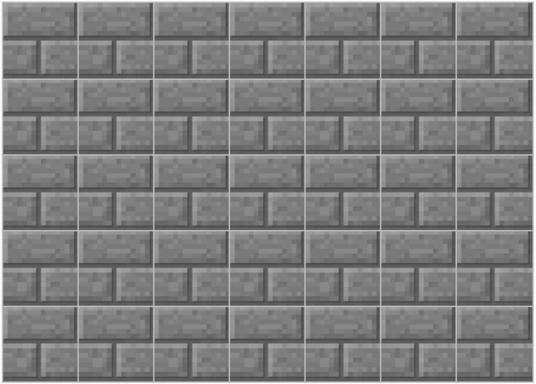
Let's use images as a clearer representation of this example.

Renovate: Sample case



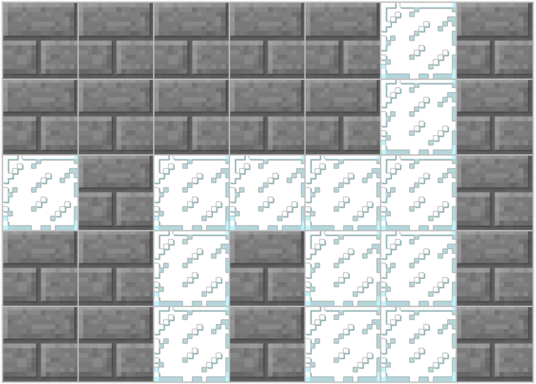
For every hole, we must decide if we fill it with a wall or with a window.

Renovate: Sample case



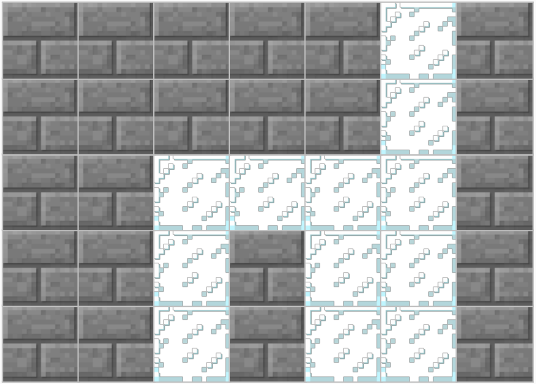
It is always possible to have a complete wall without any windows. Of course, this is very rarely optimal.

Renovate: Sample case



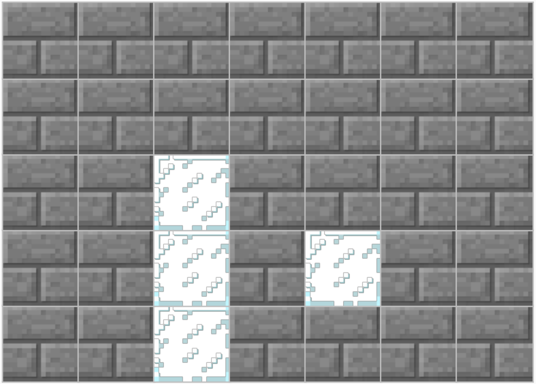
Building windows in every holey part of the wall is not always possible. For example, the wall here is not stable at all.

Renovate: Sample case



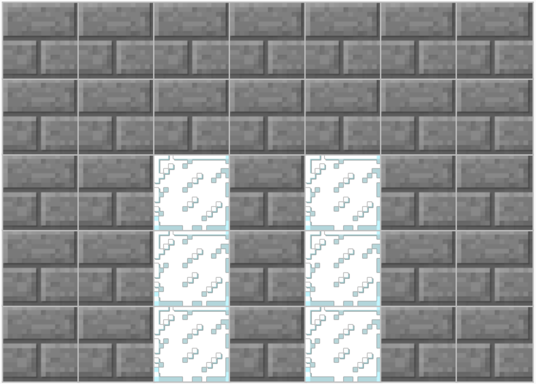
You can't have a window in the leftmost column, because there's no wall on its left. But removing it is still not enough: the wall is still unstable.

Renovate: Sample case



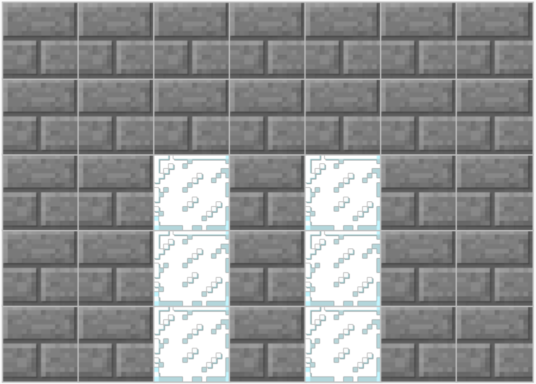
You always have to build two full columns around the windows that you build.

Renovate: Sample case



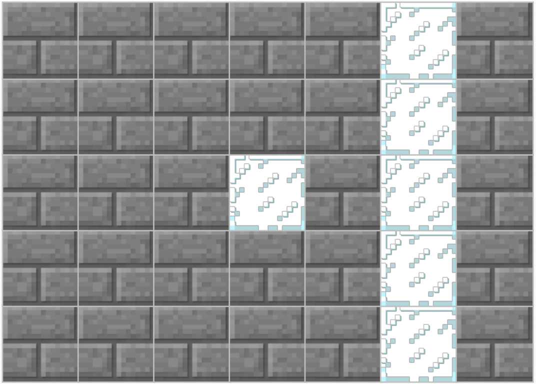
If you build a window in a column, it's always optimal to build all possible windoww in that column, because you don't need to sacrifice any further possible windows in order to build them.

Renovate: Sample case



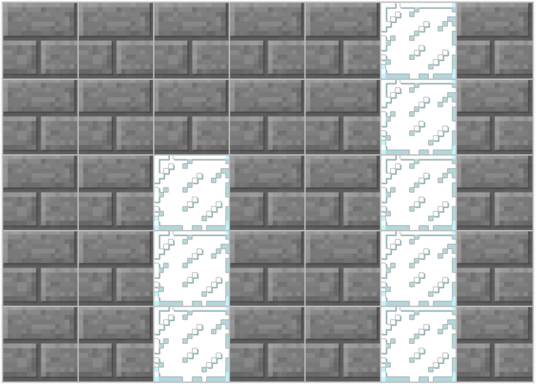
This is not optimal:
6 windows.

Renovate: Sample case



This is not optimal:
6 windows.

Renovate: Sample case



This is optimal: 8 windows.

Renovate: Precomputation

The problem gets easier when we apply these observations before starting to think about dynamic programming.

Renovate: Precomputation

The problem gets easier when we apply these observations before starting to think about dynamic programming.

- We reduce the problem to a one-dimensional problem: compute the number of possible windows in every column (we always either build none or all of them).

Renovate: Precomputation

The problem gets easier when we apply these observations before starting to think about dynamic programming.

- We reduce the problem to a one-dimensional problem: compute the number of possible windows in every column (we always either build none or all of them).
- We set the number of possible windows in the leftmost and rightmost columns to 0.

Renovate: Precomputation

The problem gets easier when we apply these observations before starting to think about dynamic programming.

- We reduce the problem to a one-dimensional problem: compute the number of possible windows in every column (we always either build none or all of them).
- We set the number of possible windows in the leftmost and rightmost columns to 0.

We start by computing an array $w[m]$. For every $0 \leq i < m$, let $w[i]$ be the number of possible windows in the i-th column.

Renovate: Precomputation

```
int main(){
    int n, m; // n = height, m = width
    cin >> n >> m;
    // w[i] = number of holes in the i-th column
    vector<int> w(m, 0);
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            int c;
            cin >> c;
            w[j] += c == 0; // this section is holey
        }
    }
    w[0] = 0; w[m-1] = 0; // no window on the edges
    ... // actual computation
```

Renovate: DP's four steps

How do we modelize this problem using the four steps ?

Renovate: Subproblems

Our general subproblem will be

Renovate: Subproblems

Our general subproblem will be s_i, the maximal number of windows using the first $i + 1$ columns for any $0 \leq i < m$.

Renovate: General formula

When trying to compute s_i, the solution for i, we have two possibilities:

Renovate: General formula

When trying to compute s_i, the solution for i, we have two possibilities:

1. We don't build the windows in the i-th column.

Renovate: General formula

When trying to compute s_i, the solution for i, we have two possibilities:

1. We don't build the windows in the i-th column.
2. We do build the windows in the i-th column. In that case, we can't have any windows in the $(i-1)$-th column.

Renovate: General formula

How does this translate into a formula?

Renovate: General formula

How does this translate into a formula?

1. If we don't build the windows in the i-th column, $s_i =$

Renovate: General formula

How does this translate into a formula?

1. If we don't build the windows in the i-th column, $s_i = s_{i-1}$.

Renovate: General formula

How does this translate into a formula?

1. If we don't build the windows in the i-th column, $s_i = s_{i-1}$.
2. If we do build the windows in the i-th column, we can't use the $(i-1)$-th column's windows, but we can add the windows from the i-th column. Thus, $s_i =$

Renovate: General formula

How does this translate into a formula?

1. If we don't build the windows in the i-th column, $s_i = s_{i-1}$.
2. If we do build the windows in the i-th column, we can't use the $(i-1)$-th column's windows, but we can add the windows from the i-th column. Thus, $s_i = s_{i-2} + w[i]$.

Renovate: General formula

How does this translate into a formula?

1. If we don't build the windows in the i-th column, $s_i = s_{i-1}$.
2. If we do build the windows in the i-th column, we can't use the $(i-1)$-th column's windows, but we can add the windows from the i-th column. Thus, $s_i = s_{i-2} + w[i]$.

Since s_i should be optimal, we take the highest of these values.

Renovate: General formula

How does this translate into a formula?

1. If we don't build the windows in the i-th column, $s_i = s_{i-1}$.
2. If we do build the windows in the i-th column, we can't use the $(i-1)$-th column's windows, but we can add the windows from the i-th column. Thus, $s_i = s_{i-2} + w[i]$.

Since s_i should be optimal, we take the highest of these values.

$$s_i = \max(s_{i-1}, s_{i-2} + w[i])$$

Renovate: Base case

We need two base cases, because our formula goes two steps back.

Renovate: Base case

We need two base cases, because our formula goes two steps back.

- $s_0 = w[0] = 0$ (you can never build windows in the first column).

Renovate: Base case

We need two base cases, because our formula goes two steps back.

- $s_0 = w[0] = 0$ (you can never build windows in the first column).
- $s_1 = w[1]$ (since there are never any windows in the first column, it is always optimal to build the second column's windows if you don't take into account the next columns).

Renovate: Relevant subproblem

The relevant subproblem is the one including all columns, so the subproblem we want to compute is $s_{\ldots}$

Renovate: Relevant subproblem

The relevant subproblem is the one including all columns, so the subproblem we want to compute is s_{m-1}.

Renovate: Relevant subproblem

The relevant subproblem is the one including all columns, so the subproblem we want to compute is $s_{m-1} (= s_{m-2}$, if $m > 1)$.

Renovate: Slow solution

Like for Fibonacci's sequence, we could use this recurrence formula and these base cases and get a slow, but correct, solution.

```
int s(int i, vector<int> &w) {
    if(i < 2) return w[i]; // base cases
    return max(s(i-1, w), s(i-2, w) + w[i]);
}
```

Renovate: Computation

We do not, however, want to compute subproblems twice. We just compute them in the order in which they're needed and store them.

Renovate: Computation

We do not, however, want to compute subproblems twice. We just compute them in the order in which they're needed and store them. The order in which they're needed, like for Fibonacci's sequence, is pretty obvious. Every subproblem relies on earlier subproblems only, so we solve them in increasing order.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	Optimal number of windows for the first 1 column
1	0	Optimal number of windows for the first 2 columns
2	3	Optimal number of windows for the first 3 columns
3	1	Optimal number of windows for the first 4 columns
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	w[0] (base case)
1	0	Optimal number of windows for the first 2 columns
2	3	Optimal number of windows for the first 3 columns
3	1	Optimal number of windows for the first 4 columns
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	Optimal number of windows for the first 2 columns
2	3	Optimal number of windows for the first 3 columns
3	1	Optimal number of windows for the first 4 columns
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	w[1] (base case)
2	3	Optimal number of windows for the first 3 columns
3	1	Optimal number of windows for the first 4 columns
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	Optimal number of windows for the first 3 columns
3	1	Optimal number of windows for the first 4 columns
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	$\max(v_1, v_0 + w[2])$
3	1	Optimal number of windows for the first 4 columns
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	Optimal number of windows for the first 4 columns
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	$\max(v_2, v_1 + w[3])$
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	3
4	3	Optimal number of windows for the first 5 columns
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	3
4	3	$\max(v_3, v_2 + w[4])$
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	3
4	3	6
5	5	Optimal number of windows for the first 6 columns
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	3
4	3	6
5	5	$\max(v_4, v_3 + w[4])$
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	3
4	3	6
5	5	8
6	0	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	3
4	3	6
5	5	8
6	0	$\max(v_5, v_4 + w[4])$

This is easy to turn into a `for` loop.

Renovate: Computation

Step by step: sample case

i	$w[i]$	v_i
0	0	0
1	0	0
2	3	3
3	1	3
4	3	6
5	5	8
6	0	8

This is easy to turn into a `for` loop.

Renovate: Computation

Computation, step by step

i	v_i
0	Optimal number of windows for the first 1 column
1	Optimal number of windows for the first 2 columns
2	Optimal number of windows for the first 3 columns
3	Optimal number of windows for the first 4 columns
4	Optimal number of windows for the first 5 columns
...	...
`m-1`	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Computation, step by step

i	v_i
0	$w[0]$ (base case)
1	Optimal number of windows for the first 2 columns
2	Optimal number of windows for the first 3 columns
3	Optimal number of windows for the first 4 columns
4	Optimal number of windows for the first 5 columns
$\dots$	$\dots$
`m-1`	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Computation, step by step

i	v_i
0	$w[0]$ (base case)
1	$w[1]$ (base case)
2	Optimal number of windows for the first 3 columns
3	Optimal number of windows for the first 4 columns
4	Optimal number of windows for the first 5 columns
...	...
`m-1`	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Computation, step by step

i	v_i
0	$w[0]$ (base case)
1	$w[1]$ (base case)
2	$max(v_1, v_0 + w[2])$
3	Optimal number of windows for the first 4 columns
4	Optimal number of windows for the first 5 columns
...	...
`m-1`	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Computation, step by step

i	v_i
0	$w[0]$ (base case)
1	$w[1]$ (base case)
2	$max(v_1, v_0 + w[2])$
3	$max(v_2, v_1 + w[3])$
4	Optimal number of windows for the first 5 columns
$\ldots$	$\ldots$
`m-1`	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Computation, step by step

i	v_i
0	$w[0]$ (base case)
1	$w[1]$ (base case)
2	$max(v_1, v_0 + w[2])$
3	$max(v_2, v_1 + w[3])$
4	$max(v_3, v_2 + w[4])$
$\dots$	$\dots$
`m-1`	Optimal number of windows for all columns

This is easy to turn into a `for` loop.

Renovate: Computation

Computation, step by step

i	v_i
0	$w[0]$ (base case)
1	$w[1]$ (base case)
2	$max(v_1, v_0 + w[2])$
3	$max(v_2, v_1 + w[3])$
4	$max(v_3, v_2 + w[4])$
$\ldots$	$\ldots$
`m-1`	$max(v_m - 2, v_m - 3 + w[m-1])$ (relevant subproblem)

This is easy to turn into a `for` loop.

Renovate: Implementation

```
    ... // precomputation
    vector<int> s(m);
    s[0] = 0; s[1] = w[1]; // base cases
    for(int i = 2; i < m; i++)
        s[i] = max(s[i-1], s[i-2] + w[i]);
    cout << s[m-1] << '\n'; // relevant subproblem
}
```

Runtime analysis

How fast does this solution run?

Runtime analysis

How fast does this solution run? $\mathcal{O}(nm)$ (because of the input; after our precomputations, we solve the problem in $\mathcal{O}(m)$).

Conclusion

When is DP useful?

- When you can divide a problem into subproblems.

When is DP useful?

- When you can divide a problem into subproblems.
- When the subproblems overlap.

When is DP useful?

- When you can divide a problem into subproblems.
- When the subproblems overlap.
- For example, it enables you to compute some recursive functions faster, for example Fibonacci's sequence.

When is DP useful?

- When you can divide a problem into subproblems.
- When the subproblems overlap.
- For example, it enables you to compute some recursive functions faster, for example Fibonacci's sequence.
- A lot of optimization problems require a dynamic programming solution.

When is DP useful?

- When you can divide a problem into subproblems.
- When the subproblems overlap.
- For example, it enables you to compute some recursive functions faster, for example Fibonacci's sequence.
- A lot of optimization problems require a dynamic programming solution.
- DP is often applied in problems with more than one dimension. In that case, finding the order of computation may be more difficult than usual.

Some remarks about recursion

It is also possible to keep the recursive function and store already stored values, for example in a map.

```
vector<int> m(MAX_N);
int fib(int n) {
    if(n<2) return n;
    if(m[n]) return m[n];
    return m[n] = fib(n-1) + fib(n-2);
}
```

Some remarks about recursion

It is also possible to keep the recursive function and store already stored values, for example in a `map`.

```
vector<int> m(MAX_N);
int fib(int n) {
    if(n<2) return n;
    if(m[n]) return m[n];
    return m[n] = fib(n-1) + fib(n-2);
}
```

Just like in the case of Fibonacci's sequence and Renovate, it is not, however, necessary to store all previous values. Recursion can also cause further problems (stack limit exceeded). The approach we used, building up the solutions in order, is called "bottom-up", and it is good to get used to it.

How to be good at DP

- DP is hard

How to be good at DP

- DP is hard for most people.

How to be good at DP

- DP is hard for most people.
- The concept is simple, but...

How to be good at DP

- DP is hard for most people.
- The concept is simple, but applying it to a problem and implementing the solution is difficult.

How to be good at DP

- DP is hard for most people.
- The concept is simple, but applying it to a problem and implementing the solution is difficult.
- Always think before you code!

How to be good at DP

- DP is hard for most people.
- The concept is simple, but applying it to a problem and implementing the solution is difficult.
- Always think before you code!
- Most important of all: solve, solve, solve!

What's next:

Solve DP tasks on the grader.

Next lecture: Subset Sum.