Introduction to Dynamic Programming

Florian Gatignon

November 10, 2019

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 (using **recursion**), 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 (using **recursion**), 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

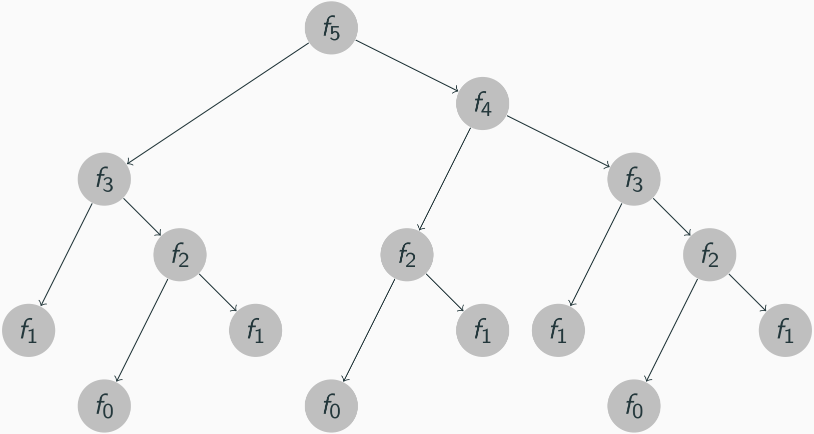
If you try to run this code to compute f_{100}, you'd have to be **very** patient to get an answer.

Why?

Fibonacci's sequence: The problem

Our program computes the same values over and over.

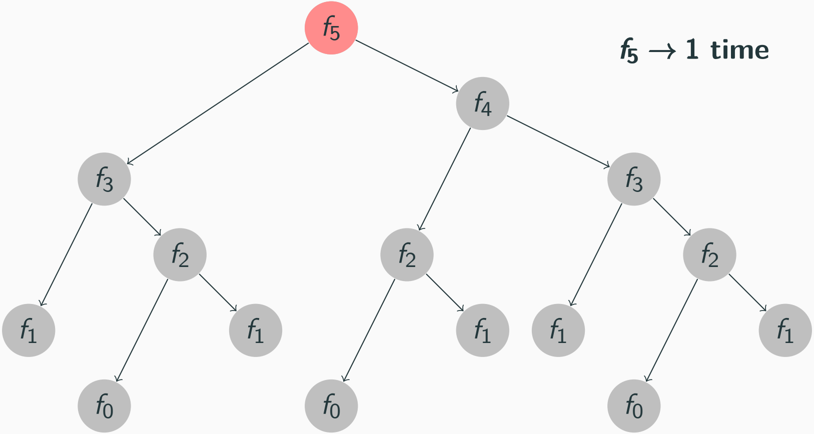
Look at this tree:



Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

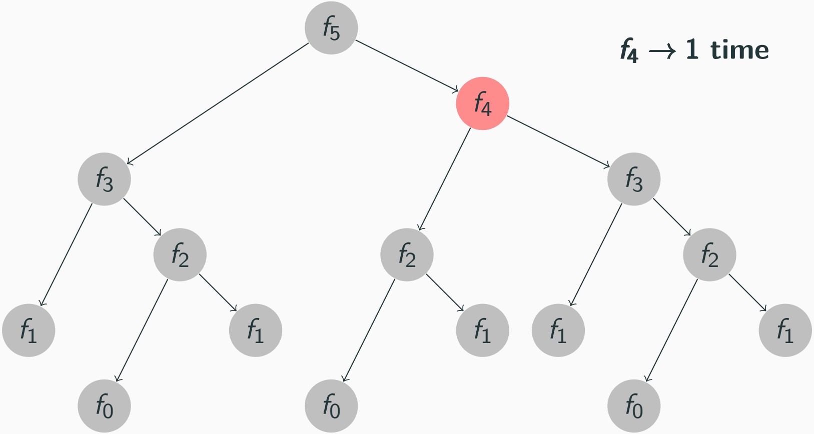


$f_5 \rightarrow$ 1 time

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

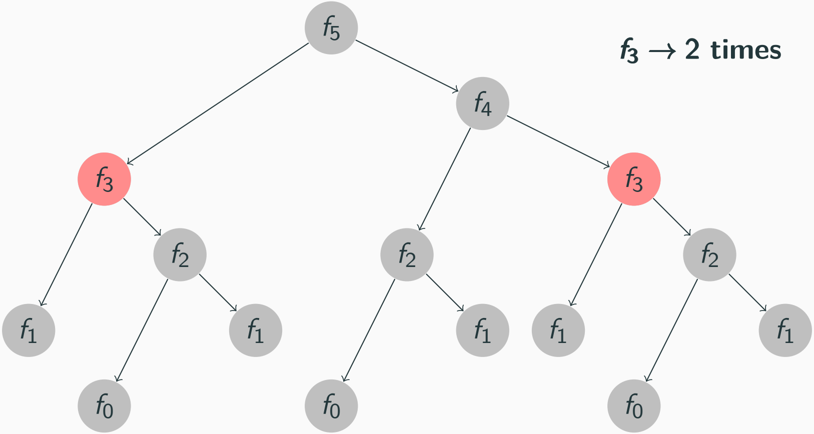


$f_4 \rightarrow$ 1 time

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

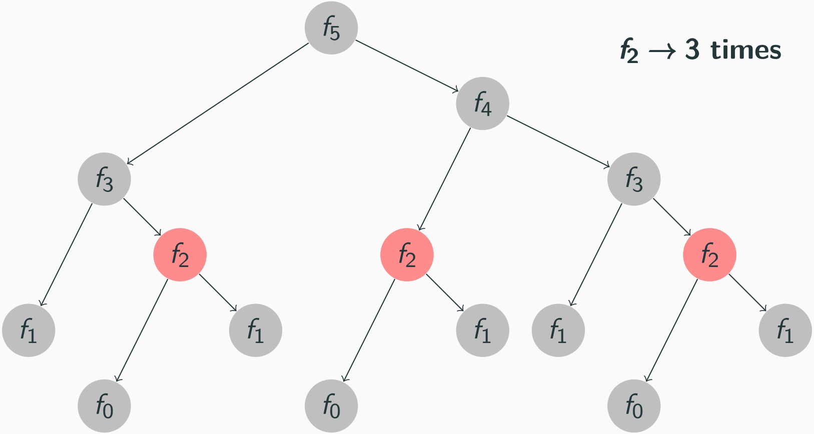


f_3 → 2 times

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

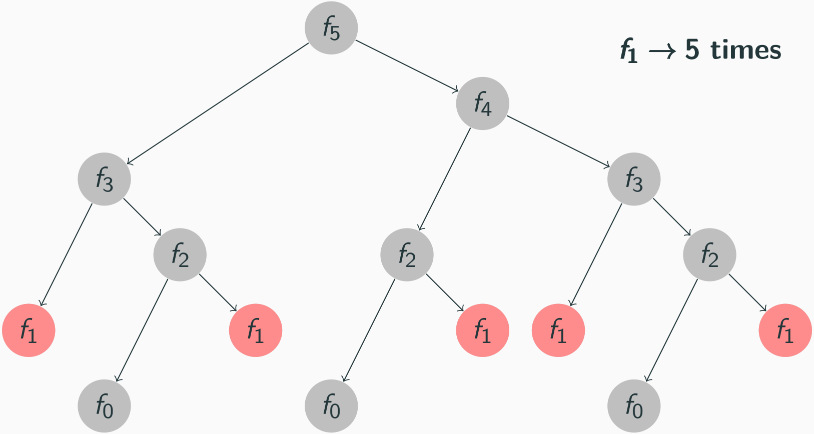


f_2 → 3 times

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:

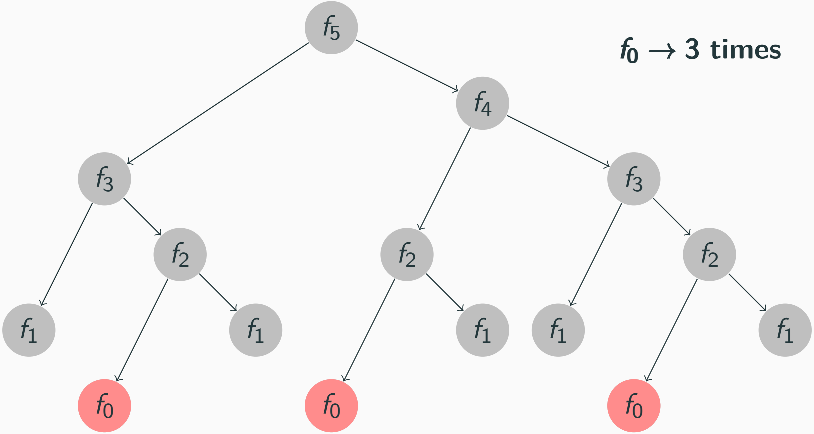


f_1 → 5 times

Fibonacci's sequence: The problem

Our program computes the same values over and over.

Look at this tree:



f_0 → 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.

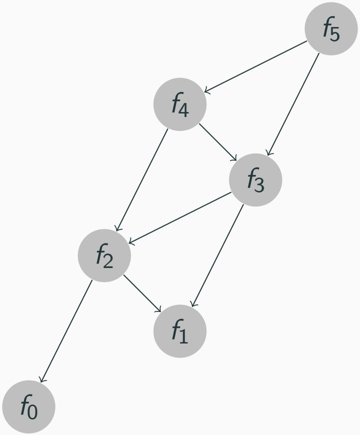
Fibonacci's sequence: The solution

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

We choose to trade memory usage for speed: store what we have computed and check if we have already computed the answer for each call.

Fibonacci's sequence: The solution

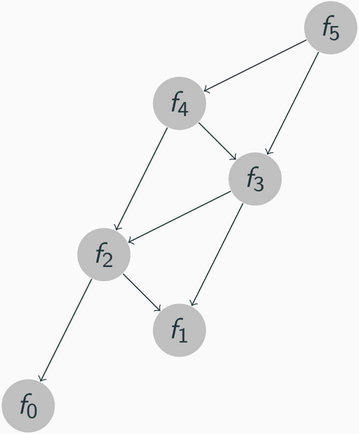
We always check whether we already have the solution for a subproblem:



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

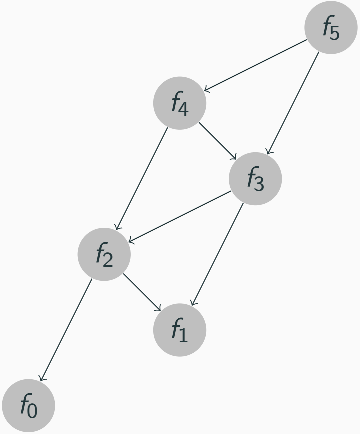
$f_5 = f_4 + f_3$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

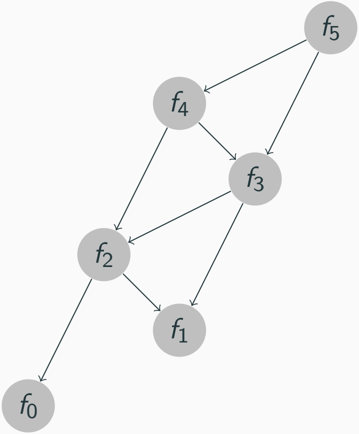
$f_5 = \boldsymbol{f_4} + f_3$
$f_4 = f_3 + f_2$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

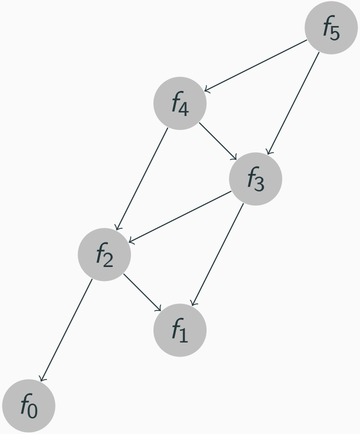
$$f_5 = \boldsymbol{f_4} + f_3$$
$$f_4 = \boldsymbol{f_3} + f_2$$
$$f_3 = f_2 + f_1$$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

$f_5 = \boldsymbol{f_4} + f_3$
$f_4 = \boldsymbol{f_3} + f_2$
$f_3 = \boldsymbol{f_2} + f_1$
$f_2 = f_1 + f_0$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

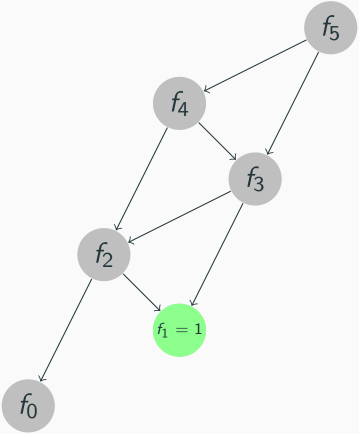
$f_5 = \boldsymbol{f_4} + f_3$

$f_4 = \boldsymbol{f_3} + f_2$

$f_3 = \boldsymbol{f_2} + f_1$

$f_2 = \boldsymbol{f_1} + f_0$

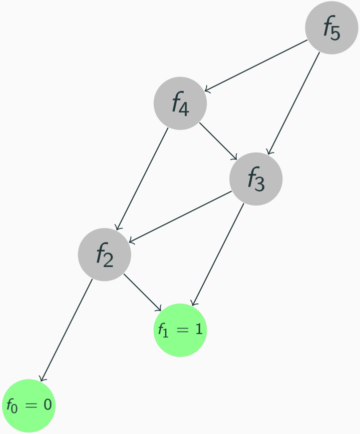
$\boldsymbol{f_1 = 1}$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

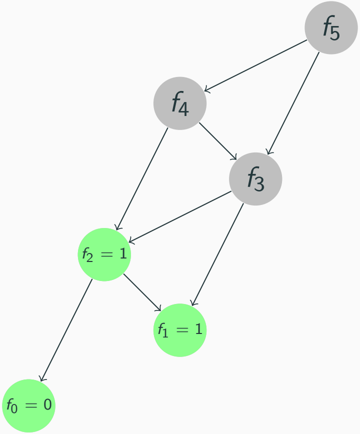
$f_5 = \boldsymbol{f_4} + f_3$
$f_4 = \boldsymbol{f_3} + f_2$
$f_3 = \boldsymbol{f_2} + f_1$
$f_2 = \boldsymbol{f_1} + \boldsymbol{f_0}$
$\boldsymbol{f_1 = 1}$
$\boldsymbol{f_0 = 0}$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

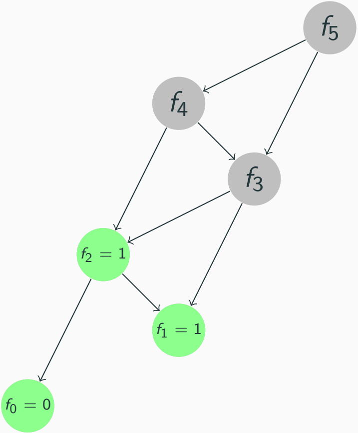
$f_5 = \boldsymbol{f_4} + f_3$
$f_4 = \boldsymbol{f_3} + f_2$
$f_3 = \boldsymbol{f_2} + f_1$
$\boldsymbol{f_2 = f_1 + f_0}$
$\boldsymbol{f_1 = 1}$
$\boldsymbol{f_0 = 0}$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

$f_5 = \boldsymbol{f_4} + f_3$
$f_4 = \boldsymbol{f_3} + f_2$
$f_3 = \boldsymbol{f_2} + \boldsymbol{f_1}$
$\boldsymbol{f_2} = \boldsymbol{f_1} + \boldsymbol{f_0}$
$\boldsymbol{f_1 = 1}$
$\boldsymbol{f_0 = 0}$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

$f_5 = \boldsymbol{f_4} + f_3$

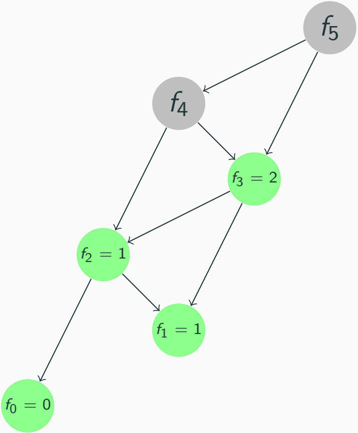
$f_4 = \boldsymbol{f_3} + f_2$

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

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

$\boldsymbol{f_1 = 1}$

$\boldsymbol{f_0 = 0}$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

$f_5 = \boldsymbol{f_4} + f_3$

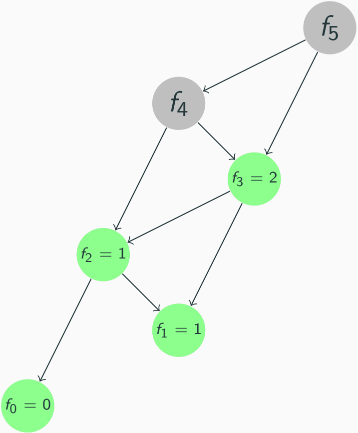
$f_4 = \boldsymbol{f_3} + \boldsymbol{f_2}$

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

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

$\boldsymbol{f_1 = 1}$

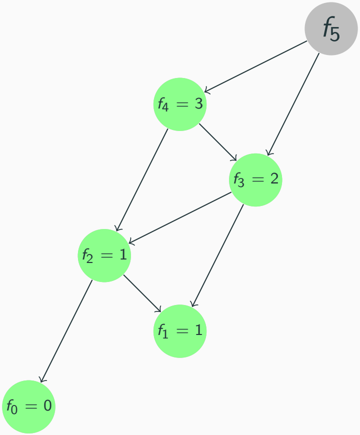
$\boldsymbol{f_0 = 0}$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

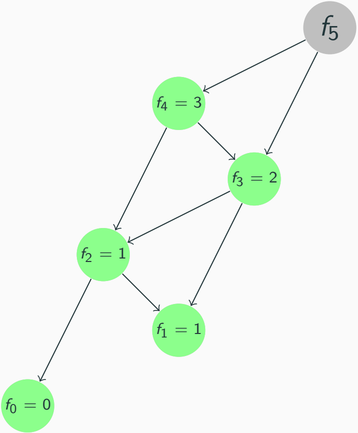
$$
\begin{aligned}
f_5 &= \boldsymbol{f_4} + f_3 \\
\boldsymbol{f_4} &= \boldsymbol{f_3} + \boldsymbol{f_2} \\
\boldsymbol{f_3} &= \boldsymbol{f_2} + \boldsymbol{f_1} \\
\boldsymbol{f_2} &= \boldsymbol{f_1} + \boldsymbol{f_0} \\
\boldsymbol{f_1} &\boldsymbol{= 1} \\
\boldsymbol{f_0} &\boldsymbol{= 0}
\end{aligned}
$$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

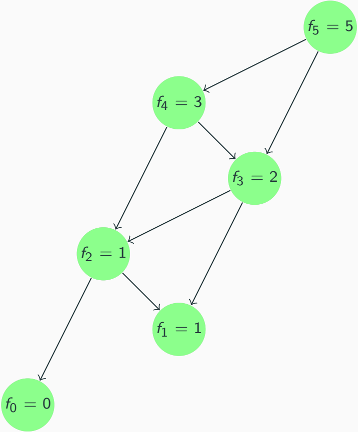
$$\begin{aligned}
f_5 &= \mathbf{f_4} + \mathbf{f_3}\\
\mathbf{f_4} &= \mathbf{f_3} + \mathbf{f_2}\\
\mathbf{f_3} &= \mathbf{f_2} + \mathbf{f_1}\\
\mathbf{f_2} &= \mathbf{f_1} + \mathbf{f_0}\\
\mathbf{f_1} &= \mathbf{1}\\
\mathbf{f_0} &= \mathbf{0}
\end{aligned}$$



Fibonacci's sequence: The solution

We always check whether we already have the solution for a subproblem:

$$\begin{aligned} f_5 &= f_4 + f_3 \\ f_4 &= f_3 + f_2 \\ f_3 &= f_2 + f_1 \\ f_2 &= f_1 + f_0 \\ f_1 &= 1 \\ f_0 &= 0 \end{aligned}$$



Fibonacci's sequence: The solution

A simple modification of our intuitive algorithm suffices:

```
vector<int> v; // initialized with v.resize(n+1,-1)
int fib(int n) {
    if(v[n]!=-1) return v[n];
    if(n==0) return v[0] = 0;
    if(n==1) return v[1] = 1;
    return v[n]=fib(n-1)+fib(n-2);
}
```

Fibonacci's sequence: The solution

A simple modification of our intuitive algorithm suffices:

```
vector<int> v; // initialized with v.resize(n+1,-1)
int fib(int n) {
    if(v[n]!=-1) return v[n];
    if(n==0) return v[0] = 0;
    if(n==1) return v[1] = 1;
    return v[n]=fib(n-1)+fib(n-2);
}
```

Our running time is now down to...

Fibonacci's sequence: The solution

A simple modification of our intuitive algorithm suffices:

```
vector<int> v; // initialized with v.resize(n+1,-1)
int fib(int n) {
    if(v[n]!=-1) return v[n];
    if(n==0) return v[0] = 0;
    if(n==1) return v[1] = 1;
    return v[n]=fib(n-1)+fib(n-2);
}
```

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

Fibonacci's sequence: The solution

A simple modification of our intuitive algorithm suffices:

```
vector<int> v; // initialized with v.resize(n+1,-1)
int fib(int n) {
    if(v[n]!=-1) return v[n];
    if(n==0) return v[0] = 0;
    if(n==1) return v[1] = 1;
    return v[n]=fib(n-1)+fib(n-2);
}
```

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

The four steps of DP

The four steps of DP

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

The four steps of DP

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

Think first, code second!

The four steps of DP

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

Think first, code second!

1. Define subproblems.

The four steps of DP

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.

The four steps of DP

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 the base case(s).

The four steps of DP

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 the base case(s).
4. Which is the relevant subproblem?

The four steps of DP and Fibonacci's sequence

The four steps of DP and Fibonacci's sequence

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

The four steps of DP and Fibonacci's sequence

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

1. Suproblems:

The four steps of DP and Fibonacci's sequence

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

1. Suproblems: f_i.

The four steps of DP and Fibonacci's sequence

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

1. Suproblems: f_i.
2. General formula:

The four steps of DP and Fibonacci's sequence

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

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

The four steps of DP 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:

The four steps of DP 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: $f_0 = 0, f_1 = 1$.

The four steps of DP 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:

The four steps of DP 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

Implementation

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

Implementation

Use a recursive function like you would in an intuitive solution.

Implementation

Use a recursive function like you would in an intuitive solution.

- Call the function for the relevant subproblem.

Implementation

Use a recursive function like you would in an intuitive solution.

- Call the function for the relevant subproblem.
- At the beginning of each call, check if you have already computed the solution for this subproblem.

Implementation

Use a recursive function like you would in an intuitive solution.

- Call the function for the relevant subproblem.
- At the beginning of each call, check if you have already computed the solution for this subproblem.
- Store and return the answer:

Implementation

Use a recursive function like you would in an intuitive solution.

- Call the function for the relevant subproblem.
- At the beginning of each call, check if you have already computed the solution for this subproblem.
- **Store** and return the answer:

Implementation

Use a recursive function like you would in an intuitive solution.

- Call the function for the relevant subproblem.
- At the beginning of each call, check if you have already computed the solution for this subproblem.
- **Store** and return the answer:
  - If you're at a base case, compute the solution for that one.

Implementation

Use a recursive function like you would in an intuitive solution.

- Call the function for the relevant subproblem.
- At the beginning of each call, check if you have already computed the solution for this subproblem.
- **Store** and return the answer:
    - If you're at a base case, compute the solution for that one.
    - Else, use a recursive call to compute the solution for this subproblem according to the formula.

Another example: Bottles

Bottles: Task statement

The problem is the following:

Bottles: Task statement

The problem is the following:

- On the second day of the SOI workshop, there are *n* bottles of soda.

Bottles: Task statement

The problem is the following:

- On the second day of the SOI workshop, there are n bottles of soda.
- Stofl wants to drink as much soda as possible.

Bottles: Task statement

The problem is the following:

- On the second day of the SOI workshop, there are n bottles of soda.
- Stofl wants to drink as much soda as possible.
- On the first day, he drank all of the soda and the other participants did not get any.

Bottles: Task statement

The problem is the following:

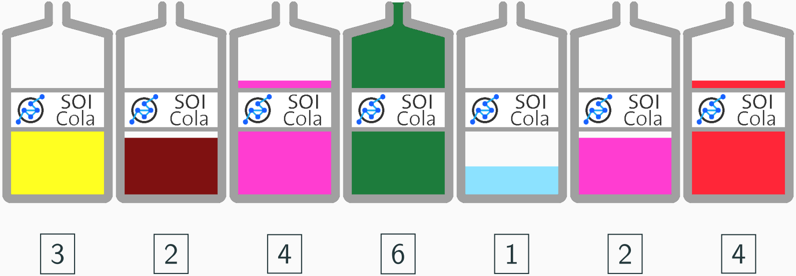
- On the second day of the SOI workshop, there are *n* bottles of soda.
- Stofl wants to drink as much soda as possible.
- On the first day, he drank all of the soda and the other participants did not get any.
- The leaders thus ruled that Stofl should be allowed to drink only from non-adjacent bottles.

Bottles: Task statement

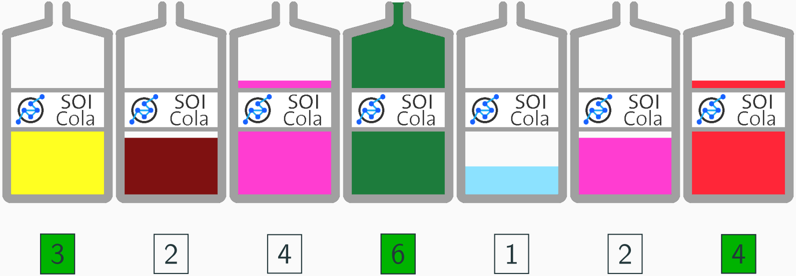
The problem is the following:

- On the second day of the SOI workshop, there are n bottles of soda.
- Stofl wants to drink as much soda as possible.
- On the first day, he drank all of the soda and the other participants did not get any.
- The leaders thus ruled that Stofl should be allowed to drink only from non-adjacent bottles.
- Compute the maximum amount of soda that Stofl can drink, given the volumes v_i of all available bottles.

Bottles: Sample case



Bottles: Sample case



Stofl can drink 13 units of soda at most!

Bottles: The four steps of DP

How do we model this problem using the four steps described earlier?

Bottles: Subproblems

Our general subproblem will be

Bottles: Subproblems

Our general subproblem will be s_i, the maximum amount of soda drinkable if the problem is restricted to the $i+1$ first bottles, for any $0 \leq i < n$.

Bottles: General formula

When trying to compute s_i, the solution for the $i + 1$ first bottles, we have two possibilities:

Bottles: General formula

When trying to compute s_i, the solution for the $i + 1$ first bottles, we have two possibilities:

1. We drink the soda in the i-th bottle (0-based).

Bottles: General formula

When trying to compute s_i, the solution for the $i + 1$ first bottles, we have two possibilities:

1. We drink the soda in the i-th bottle (0-based).
2. We do not drink from that bottle.

Bottles: General formula

How does this translate into a formula?

Bottles: General formula

How does this translate into a formula?

1. If we do not drink the *i*-th bottle, Stofl can drink

Bottles: General formula

How does this translate into a formula?

1. If we do not drink the i-th bottle, Stofl can drink s_{i-1}.

Bottles: General formula

How does this translate into a formula?

1. If we do not drink the i-th bottle, Stofl can drink s_{i-1}.
2. If we do drink the i-th bottle, we can't drink the $(i-1)$-th bottle, but we can add the i-th one. Thus, Stofl can drink

Bottles: General formula

How does this translate into a formula?

1. If we do not drink the i-th bottle, Stofl can drink s_{i-1}.
2. If we do drink the i-th bottle, we can't drink the $(i-1)$-th bottle, but we can add the i-th one. Thus, Stofl can drink $s_{i-2} + v_i$.

Bottles: General formula

How does this translate into a formula?

1. If we do not drink the i-th bottle, Stofl can drink s_{i-1}.
2. If we do drink the i-th bottle, we can't drink the $(i-1)$-th bottle, but we can add the i-th one. Thus, Stofl can drink $s_{i-2} + v_i$.

s_i should be as large as possible, so we always take the max.

Bottles: General formula

How does this translate into a formula?

1. If we do not drink the i-th bottle, Stofl can drink s_{i-1}.
2. If we do drink the i-th bottle, we can't drink the $(i-1)$-th bottle, but we can add the i-th one. Thus, Stofl can drink $s_{i-2} + v_i$.

s_i should be as large as possible, so we always take the max.

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

Bottles: Base case

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

Bottles: Base case

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

- $s_0 = v_0$ (if there is only one bottle, just drink it).

Bottles: Base case

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

- $s_0 = v_0$ (if there is only one bottle, just drink it).
- $s_1 = max(v_0, v_1)$ (when there are two bottles, you can always drink exactly one of those, so just pick the largest one).

Bottles: Relevant subproblem

The relevant subproblem is the one including all bottles, so the subproblem we want to compute is $s_{\dots}$.

Bottles: Relevant subproblem

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

Bottles: Slow solution

Applying this recusion intuitively gives this code:

```
vector<int> v   ;
int dp(int i) {

    if(i==0) return        v[0];
    if(i==1) return        max(v[0],v[1]);
    return        max(dp(i-1),dp(i-2)+v[i]);
} int main() {
    int n; cin >> n;
    v.resize(n);

    for(int i = 0; i < n; i++) cin >> v[i];
    cout << dp(n-1) << endl;
}
```

Bottles: DP solution

Improving it with memoization is easy:

```
vector<int> v, m;
int dp(int i) {
    if(m[i]!=-1) return m[i];
    if(i==0) return m[0] = v[0];
    if(i==1) return m[1] = max(v[0],v[1]);
    return m[i] = max(dp(i-1),dp(i-2)+v[i]);
} int main() {
    int n; cin >> n;
    v.resize(n);
    m.resize(n,-1);
    for(int i = 0; i < n; i++) cin >> v[i];
    cout << dp(n-1) << endl;
}
```

Bottles: Runtime analysis

This is just like with Fibonacci: we went from an exponential to a linear runtime!

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. A good way to recognize them is that they are about making choices: choosing whether to build a wall or a window, whether to pack an item in one's bag, etc.

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.