# Dynamic Programming

Ulises Tirado Zatarain [1]
(ulises.tirado@cimat.mx)

[1] Algorists Group

November, 2020

# Outline

1. Introduction
   - Definitions
   - A particular problem
   - Naive solutions and limitations
2. Algorithm design
   - Design
   - Analysis
3. More problems
   - Unidimensional problems
   - Multidimensional problems
   - More classic problems

# Outline

# What is dynamic programming?

**Definition (Dynamic Programming)**

It's a technique for mathematical programming (optimization), or in other words, a paradigm to problem solving. So, the word "programming" is not directly meaning a computer program or an algorithm. The actual meaning is more similar to linear programming or planning or taking decisions.

- It consists in:
  - Split the problem in smaller sub-problems:
    - recurrence relationship
    - optimal sub-structure
  - We stop when have a trivial problems (base cases)
  - Store smaller solutions (memoization, states)
  - Merge the solutions when is need it
  - Difference with D&C: **overlapping and compute a value**

# Outline

1. Introduction
   - Definitions
   - A particular problem
   - Naive solutions and limitations
2. Algorithm design
   - Design
   - Analysis
3. More problems
   - Unidimensional problems
   - Multidimensional problems
   - More classic problems

# Recurrence relationship

Fibonacci sequence $(0, 1, 1, 2, 3, 5, \ldots)$ is the simplest examples

$$F_N = \begin{cases} N & N \in \{0, 1\} \\ F_{N-1} + F_{N-2} & N > 1 \end{cases}$$

## Fibonacci numbers

Given Q queries compute the N-th Fibonacci number for each of them. Each query is a single line with a single integer N. Your task is write a function which receives the integer N as input and should return the N-th Fibonacci number.

# Outline

# Fibonacci classic iterative implementation

```
const int MAX = 90;
function long int fibonacci(int N):
    if n < 0 or N > MAX then throw "Out of range.";
    long current = 0, next = 1;
    for k = 1 to N do:
        long aux = current + next;
        current = next;
        next = aux;
    end
    return current;
end
```

# Fibonacci classic recursive implementation

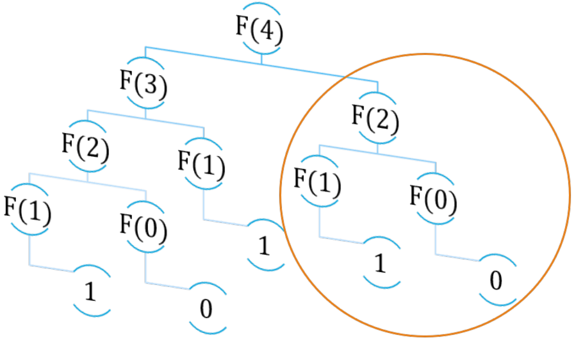
**const int** $MAX = 90$;
**function long int** `fibonacci`(**int** N)**:**
  **if** $N < 0$ **or** $N > MAX$ **then throw** "Out of range.";
  **if** $N < 2$ **then return** N;
  **return** `fibonacci`($N - 1$) + `fibonacci`($N - 2$);
**end**

# Problems with the classic implementations

- In the iterative implementation we compute all the values for every query.

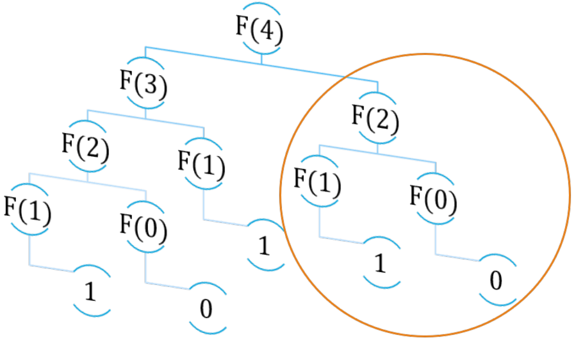
# Problems with the classic implementations

- In the iterative implementation we compute all the values for every query.
- In the recursive one, some values are computed twice (or more times), in particular for $F(4)$:



# Problems with the classic implementations

- In the iterative implementation we compute all the values for every query.
- In the recursive one, some values are computed twice (or more times), in particular for $\mathrm{F}(4)$:




- This is called **overlapping**

# Complexity?

- What is run time complexity for the iterative function?

# Complexity?

- What is run time complexity for the iterative function? $O(N)$

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one?

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$
- What is the run time complexity for the whole program?

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$
- What is the run time complexity for the whole program?
  - Iterative: $O(QN)$

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$
- What is the run time complexity for the whole program?
    - Iterative: $O(QN)$
    - Recursive: $O(2^N Q)$

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$
- What is the run time complexity for the whole program?
  - Iterative: $O(QN)$
  - Recursive: $O(2^N Q)$
- What about the memory?

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$
- What is the run time complexity for the whole program?
  - Iterative: $O(QN)$
  - Recursive: $O(2^N Q)$
- What about the memory?
  - Iterative: $O(1)$

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$
- What is the run time complexity for the whole program?
  - Iterative: $O(QN)$
  - Recursive: $O(2^N Q)$
- What about the memory?
  - Iterative: $O(1)$
  - Recursive: $O(N)$–Why?

# Complexity?

- What is run time complexity for the iterative function? $O(N)$
- What is run time complexity for the recursive one? $O(2^N)$
- What is the run time complexity for the whole program?
  - Iterative: $O(QN)$
  - Recursive: $O(2^N Q)$
- What about the memory?
  - Iterative: $O(1)$
  - Recursive: $O(N)$–Why? Stack

# Outline

# Design and implementations

There are two ways to implement dynamic programming:

- Bottom-up (iterative implementation)
  - Solve **all the the possible smaller problems** before the bigger one
- Top-down (recursive implementation)
  - Solve **only the instances which are actually needed** for a given problem
- Both solve the problem in efficient way, the discussion may end up in *religious* arguments.

# Fibonacci bottom-up implementation

Solve all the the possible smaller problems before the bigger one:

```
const int MAX = 90;
const long int UNDEFINED = −1;
long int memo[MAX + 1];
std::memset(memo, UNDEFINED, sizeof(memo));
memo[0] = 0; memo[1] = 1;
function long int fibonacci(int N):
    if n < 0 or N > MAX then throw "Out of range.";
    if memo[N] == UNDEFINED then
        for k = 2 to N do:
            memo[k] = memo[k − 1] + memo[k − 2];
        end
    end
    return memo[N];
end
```

# Fibonacci top-down implementation

Solve only the instances which are actually needed for a given problem:

```
const int MAX = 90;
const long int UNDEFINED = −1;
long int memo[MAX + 1];
std::memset(memo, UNDEFINED, sizeof(memo));
memo[0] = 0; memo[1] = 1;
function long int fibonacci(int N):
    if N < 0 or N > MAX then throw "Out of range.";
    if memo[N] ≠ UNDEFINED then return memo[N];
    memo[n] = fibonacci(N − 1) + fibonacci(N − 2);
    return memo[N];
end
```

# Outline

1. Introduction
   - Definitions
   - A particular problem
   - Naive solutions and limitations
2. Algorithm design
   - Design
   - Analysis
3. More problems
   - Unidimensional problems
   - Multidimensional problems
   - More classic problems

# Complexity analysis

- What is the run time complexity in both cases?

# Complexity analysis

- What is the run time complexity in both cases?
  $O(\max(Q, N))$

# Complexity analysis

- What is the run time complexity in both cases?
  $O(\max(Q, N))$
- And the memory complexity?

# Complexity analysis

- What is the run time complexity in both cases? $\mathrm{O}(\max(\mathrm{Q}, \mathrm{N}))$
- And the memory complexity? $\mathrm{O}(\mathrm{N})$

# Complexity analysis

- What is the run time complexity in both cases? $\mathrm{O}(\max(\mathrm{Q}, \mathrm{N}))$
- And the memory complexity? $\mathrm{O}(\mathrm{N})$
- Generally speaking, DP solutions have a run time complexity $\mathrm{O}(M \times S)$ and in memory $\mathrm{O}(M)$; where $M$ is the number of sub-problems problem and $S$ is the complexity of solving each sub-problem.

# Outline

# Coin change (brainstorming/coding)

## Coin change (source: LeetCode)

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1. You may assume that you have an infinite number of each kind of coin.

## Example

**Input**: coins = [1,2,5], amount = 11
**Output**: 3
**Explanation**: $11 = 5 + 5 + 1$

# David's staircase (brainstorming/coding)

## David's staircase (source: HackerRank)

Davis has a number of staircases in his house and he likes to climb each staircase $1, 2$ or $3$ steps at a time. Being a very precocious child, he wonders how many ways there are to reach the top of the staircase.
Given the respective heights for each of the $s$ staircases in his house, find and print the number of ways he can climb each staircase, module $10^{10} + 7$ on a new line.

## Example

For example, there is $s = 1$ staircase in the house that is $n = 5$ steps high. David can step on up to 13 sequences of steps.

# David's staircase (brainstorming/coding)

## Example ($s = 1, n = 5$ continuation...)

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 2 | |
| 3 | 1 | 1 | 2 | 1 | |
| 4 | 1 | 2 | 1 | 1 | |
| 5 | 2 | 1 | 1 | 1 | |
| 6 | 1 | 2 | 2 | | |
| 7 | 2 | 2 | 1 | | |
| 8 | 2 | 1 | 2 | | |
| 9 | 1 | 1 | 3 | | |
| 10 | 1 | 3 | 1 | | |
| 11 | 3 | 1 | 1 | | |
| 12 | 2 | 3 | | | |
| 13 | 3 | 2 | | | |

# Outline

# Multidimensional (homework challenge)

## Knight on the chess board (source: LeetCode)

On an $N \times N$ chessboard, a knight starts at the $r$-th row and $c$-th column and attempts to make exactly $K$ moves. The rows and columns are $0$ indexed, so the top-left square is $(0, 0)$, and the bottom-right square is $(N-1, N-1)$.
A chess knight has $8$ possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction, then one square in an orthogonal direction.

## Example

**Input**: 3, 2, 0, 0
**Output**: 0.0625

# Outline

1. Introduction
   - Definitions
   - A particular problem
   - Naive solutions and limitations
2. Algorithm design
   - Design
   - Analysis
3. More problems
   - Unidimensional problems
   - Multidimensional problems
   - More classic problems

# More classic problems

We can find many applications of dynamic programming in several knowledge areas (even if they are not necessary related with technology a priori): militia, robotics, image processing, etc.

- Longest Common Subsequence
- Edit distance
- Knapsack
- Floyd-Warshall
- Single-source Shortest Path

# References I


- Introduction to Algorithms, Thomas H. Cormen
- Algorists: Github Repository
- Wikipedia: Dynamic Programming
- HackerRank
- CodeForces
- OmegaUp
- LeetCode