Dynamic Programming

Nicolò Felicioni

Email: nicolo . felicioni@ polimi . it

POLITECNICO MILANO 1863

POLITECNICO
MILANO 1863

Intro Presentation

Nicolò Felicioni

- 👉 Ph.D. Student in Computer Science
- 👉 Email: nicolo . felicioni @ polimi . it
- 👉 Research Interests:
  - 🔎 Machine Learning
  - 🔎 Recommendation Systems

Dynamic Programming

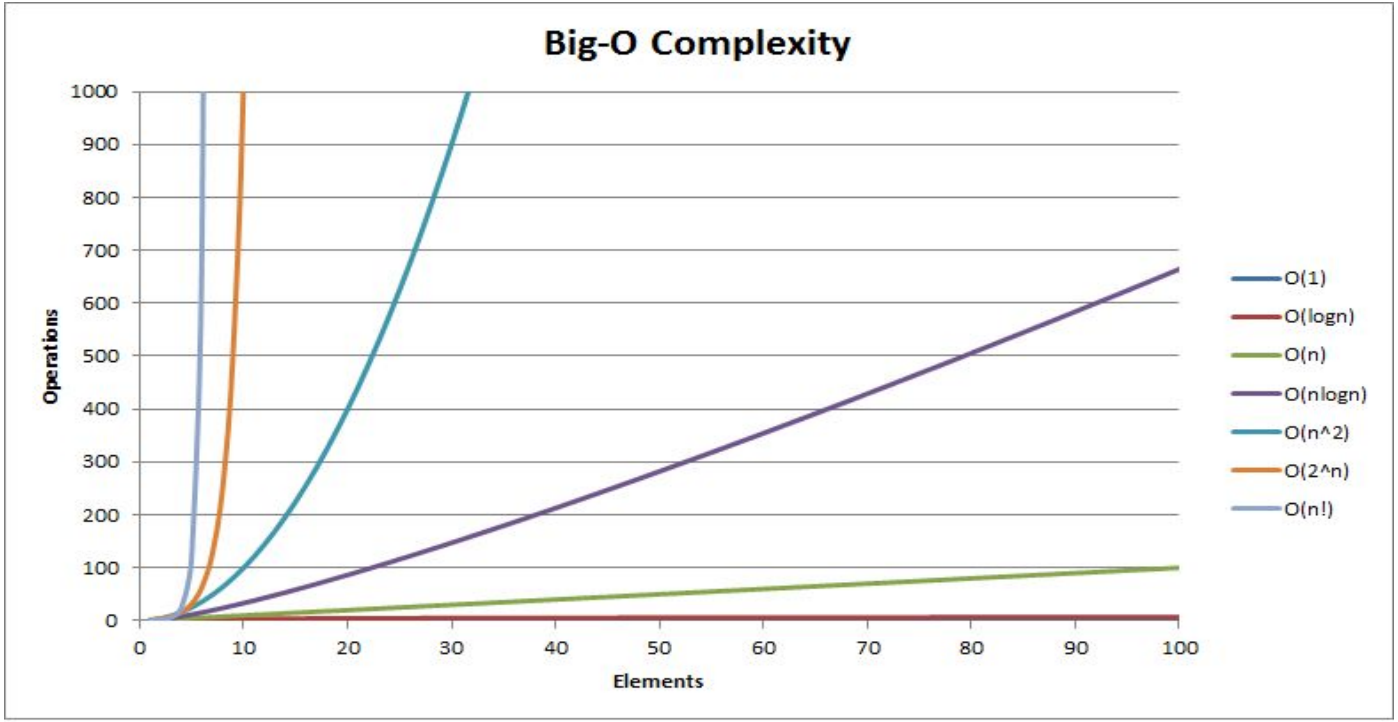
POLITECNICO
MILANO 1863

Dynamic Programming Definition

Dynamic Programming

- **Dynamic programming** is an algorithm design technique
- DP can be viewed also as an **optimization** technique
- *Usually*, brings a naive solution with exponential complexity to polynomial complexity
- $O(c^N) \rightarrow O(N^c)$

Big O Complexities



Difference Explained (from Garey & Johnson, 1979)

Complexity function	n=10
n linear	.00001 second
n^2 polynomial	.0001 second
n^5 polynomial	.1 second
2^n exponential	.001 second

Difference Explained (from Garey & Johnson, 1979)

Complexity function	n=10	n=20
n linear	.00001 second	.00002 second
n^2 polynomial	.0001 second	.0004 second
n^5 polynomial	.1 second	3.2 seconds
2^n exponential	.001 second	1.0 second

Difference Explained (from Garey & Johnson, 1979)

Complexity function	n=10	n=20	n=30
n linear	.00001 second	.00002 second	.00003 second
n^2 polynomial	.0001 second	.0004 second	.0009 second
n^5 polynomial	.1 second	3.2 seconds	24.3 second
2^n exponential	.001 second	1.0 second	17.9 minutes

Difference Explained (from Garey & Johnson, 1979)

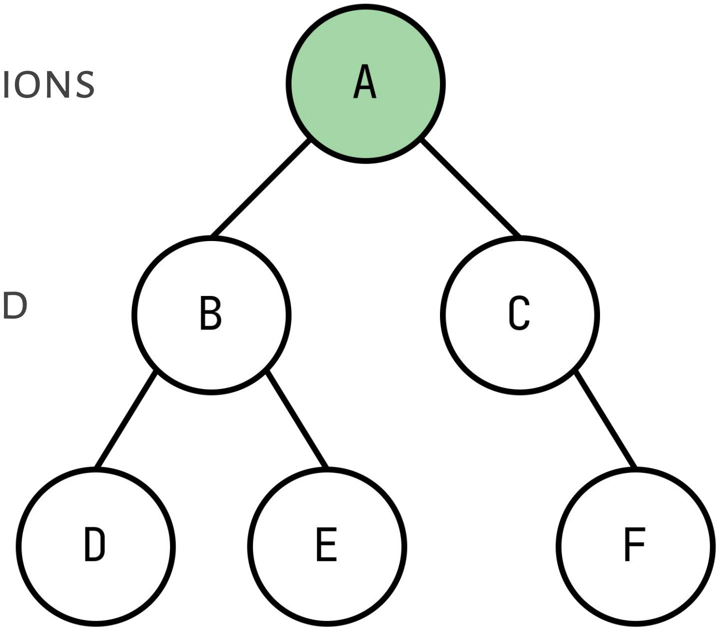
Complexity function	n=10	n=20	n=30	n=40	n=50	n=60
n linear	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2 polynomial	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^5 polynomial	.1 second	3.2 seconds	24.3 second	1.7 minutes	5.2 minutes	13.0 minutes
2^n exponential	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries

WHEN TO USE DP

- 👉 DP CAN BE USED IN A VARIETY OF SITUATIONS

- 👉 THERE MUST BE

  2 FUNDAMENTAL PROPERTIES THAT HOLD



Property 1: Optimal Substructure

The **Optimal Substructure** property holds when:

👉 By solving optimally each subproblem,

you optimally solve the original problem

Property 1: Optimal Substructure

The **Optimal Substructure** property holds when:

👉 By solving optimally each subproblem,
you optimally solve the original problem

Property 2: Overlapping Subproblems

The **Overlapping Subproblems** property holds when:

- 👉 The original problem can be broken down into subproblems that are **reused** several times

Property 2: Overlapping Subproblems

The **Overlapping Subproblems** property holds when:

- 👉 The original problem can be broken down into subproblems that are **reused** several times
- 👉 Any recursive algorithm solving the problem should solve the **same** sub-problems over and over, rather than generating new sub-problems

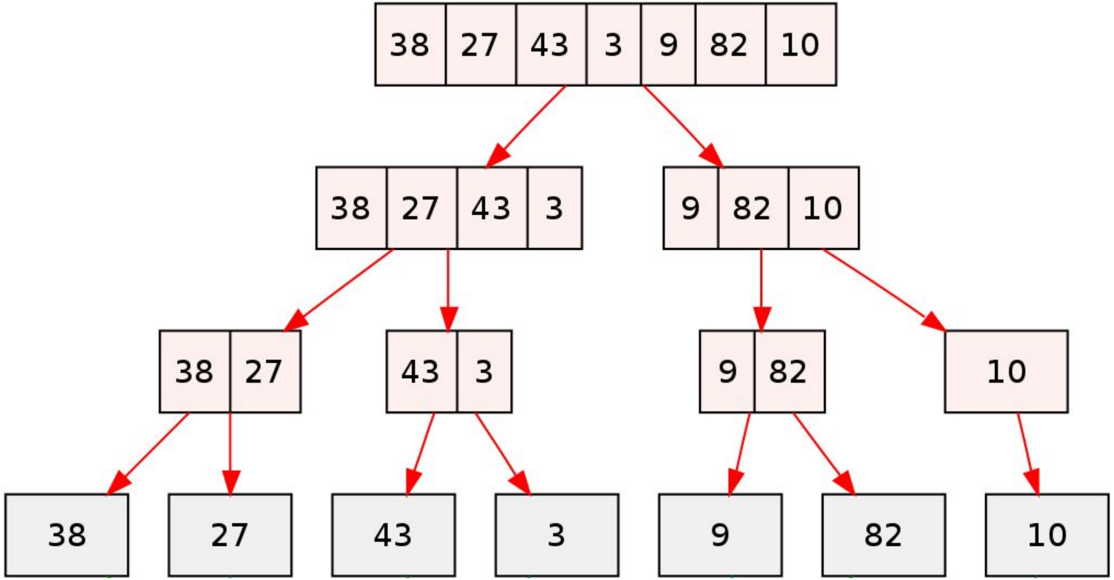
Example: Mergesort

- 👉 MergeSort is **not** a DP algorithm
- 👉 This is because the sorting problem satisfies property 1 (optimal substructure) but not property 2 (overlapping subproblems)

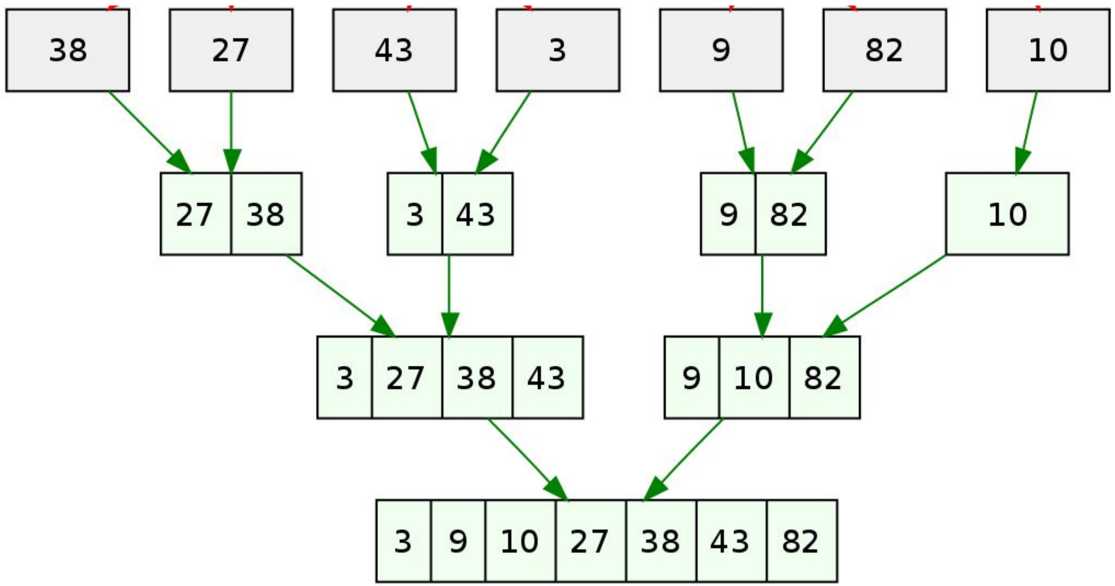
Example: Mergesort

38	27	43	3	9	82	10

EXAMPLE: MERGESORT



Example: Mergesort



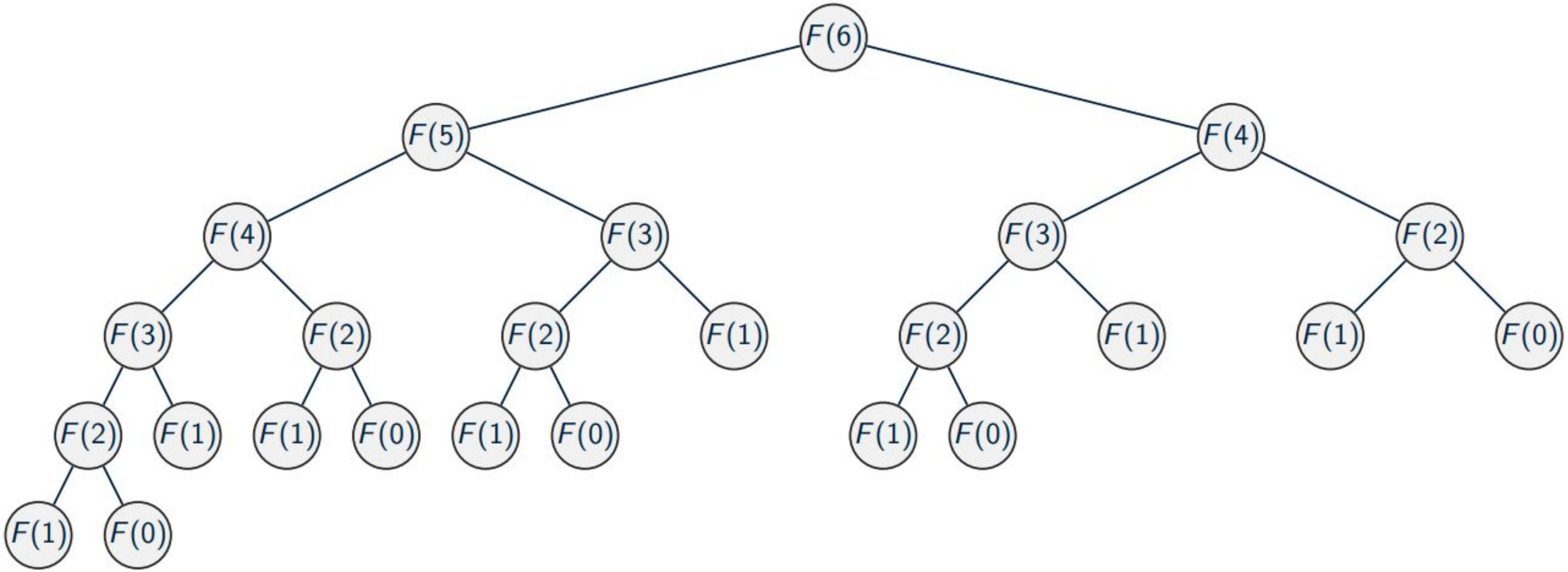
Example: Computing Fibonacci numbers

A **Naive** recursive solution:

```
def fib(i):
    if i < 2: return i
    return fib(i-1) + fib(i-2)
```

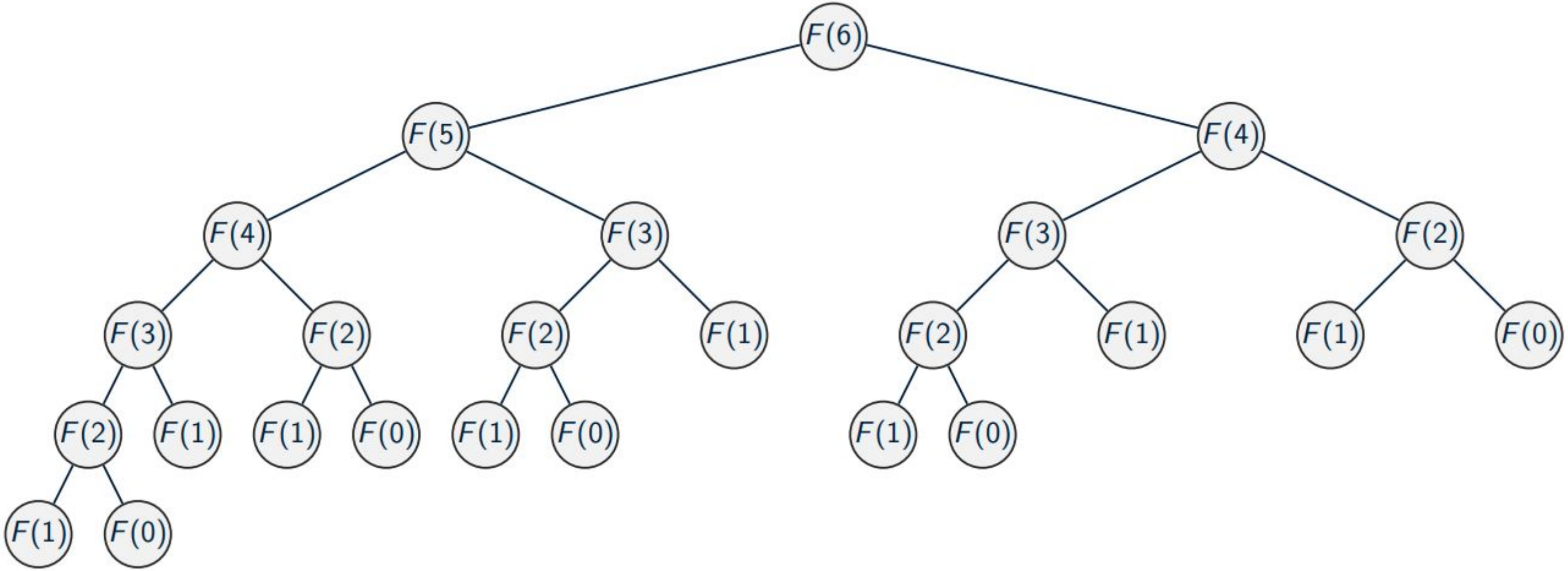
EXAMPLE: COMPUTING FIBONACCI NUMBERS

RECURSION TREE:



Example: Computing Fibonacci numbers

Exponential complexity!



Example: Computing Fibonacci numbers

- 👉 The naive solution does not exploit the **overlapping subproblems** property

Example: Computing Fibonacci numbers

- The naive solution does not exploit the **overlapping subproblems** property
- This problem has a lot of overlaps

Example: Computing Fibonacci numbers

- 👉 The naive solution does not exploit the **overlapping subproblems** property
- 👉 This problem has a lot of overlaps
- 👉 We will use the so-called **Memoization** technique

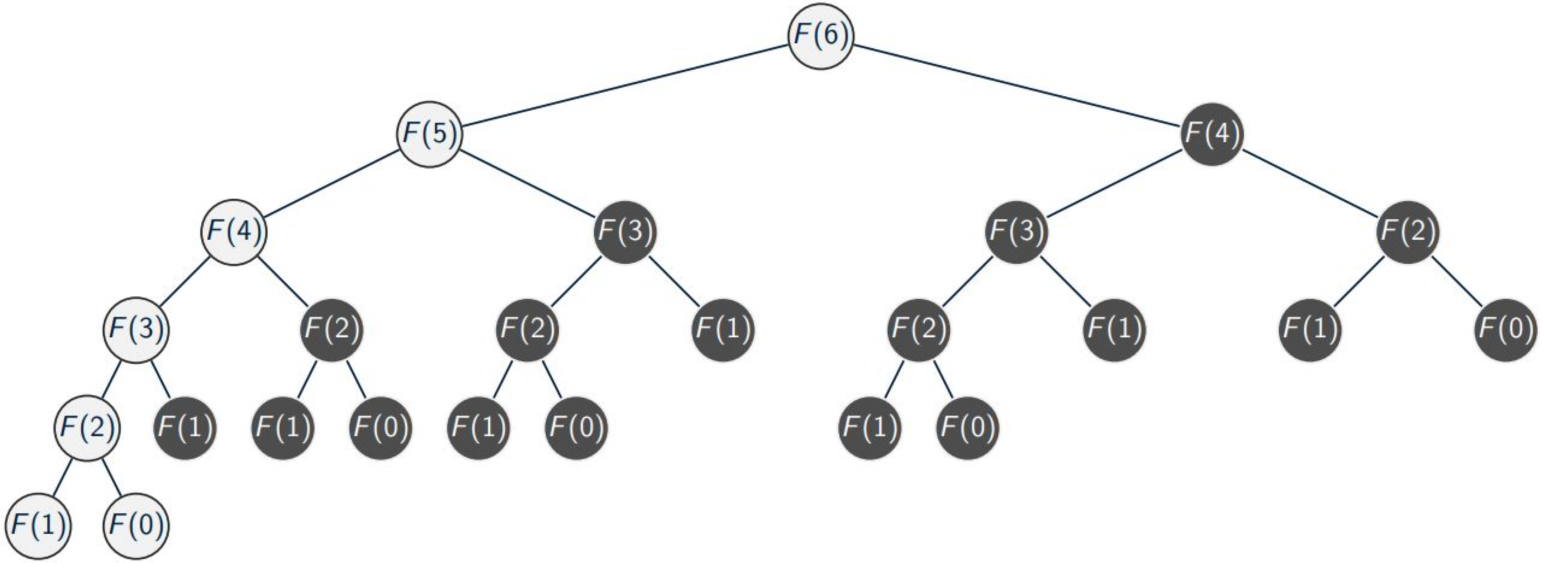
Example: Computing Fibonacci numbers

A solution with **memoization**:

```
def fib_memo(i):
    mem = {} #dict of cached values
    def fib(x):
        if x < 2: return x
        #check if already computed
        if x in mem: return mem[x]
        #only if not already computed
        mem[x] = fib(x-1) + fib(x-2)
        return mem[x]
    fib(i)
    return mem[i]
```

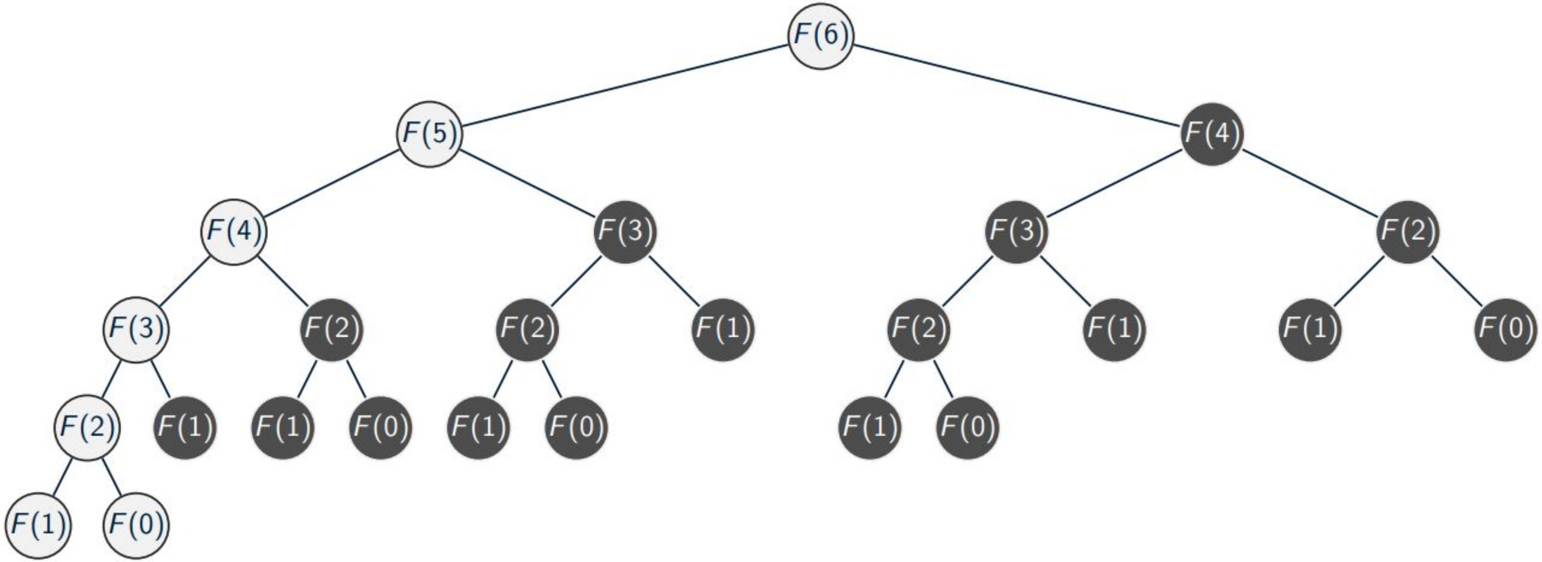
Example: Computing Fibonacci numbers

Recursion tree with memoization:



Example: Computing Fibonacci numbers

The black nodes are not computed anymore: **Linear** complexity!



LeetCode

LeetCode
Problem 1

POLITECNICO
MILANO 1863

70 Climbing Stairs

https://leetcode.com/problems/climbing-stairs/

You are climbing a staircase. It takes **n** steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

POLITECNICO
MILANO 1863

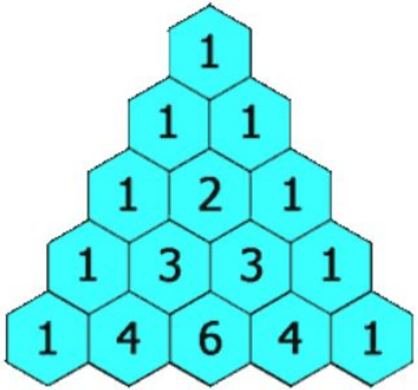
LeetCode Problem 2

118 Pascal's Triangle

HTTPS://LEETCODE.COM/PROBLEMS/PASCALS-TRIANGLE/

Given an integer *numRows*, return the first *numRows* of **Pascal's triangle**.

In Pascal's triangle, each number is the sum of the two numbers directly above it as shown:



LeetCode

Problem 3

115 Distinct Subsequences

https://leetcode.com/problems/distinct-subsequences/

Given two strings **s** and **t**, return the number of distinct subsequences of **s** which equals **t**.

```
Input: s = "babgbag", t = "bag"
Output: 5
Explanation:
As shown below, there are 5 ways you can generate "bag" from s.
babgbag
babgbag
babgbag
babgbag
babgbag
```

POLITECNICO
MILANO 1863

Dynamic Programming
Credits

Credits

- Slides by Carola Wenk
  https://www.cs.tulane.edu/~carola/teaching/cmps6610/fall16/slides/Lecture_DynamicProgramming.pdf
- Slides by Tyler Moore
  https://tylermoore.utulsa.edu/courses/cse3353/s13/slides/l19-handout.pdf
- Wikipedia page on Dynamic Programming
  https://en.wikipedia.org/wiki/Dynamic_programming