TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

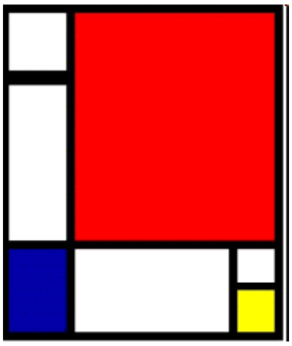
Algorithm Design: Fibonacci Numbers

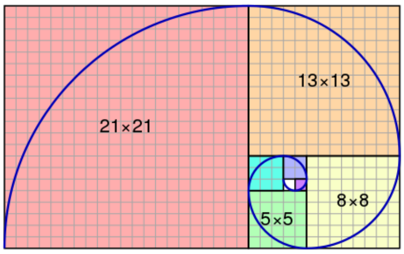
Algorithms, Data and Security
A.Y. 2023/24

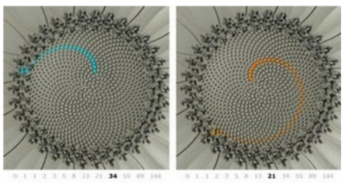
Valeria Cardellini

Global Governance, 3rd year
Science and Technology Major

Fibonacci numbers in art and nature







Fibonacci numbers in nature

- An example of efficiency in nature
- As each row of seeds in a sunflower or pine cone, or petals on a flower grows, it tries to put the maximum number in the smallest space
- Fibonacci numbers are the whole numbers which express the golden ratio
  – Corresponds to the angle which maximises the number of items in the smallest space

Fibonacci

- Why are they called Fibonacci numbers?
- Leonardo of Pisa (aka Fibonacci), 1175-1250
- He wrote Liber Abaci (1202), one of the first books to be published by a European
- One of the first people to introduce the decimal number system into Europe
- On his travels he saw the advantage of Hindu-Arabic numbers compared to Roman numerals
- Interested in many problems, including rabbit problem
  – About how math is related to all kinds of things you'd never have thought of ☺

Rabbit island

- Fibonacci was interested in the following population dynamics problem:

  How fast is a population of rabbits expanding (subject to certain conditions)?

- In particular, if one starts from one rabbit pair (in a deserted island), how many rabbit pairs there will be at year n?

Ōkunoshima island in Japan

Model/Assumptions on rabbits

1. A rabbit pair gives birth to two little rabbits (one male, one female) per year
2. Rabbits start reproducing two years after birth
3. Rabbits do not die

- Last assumption may seem unrealistic
  - But makes the problem simpler...
- We will solve the problem and check it out

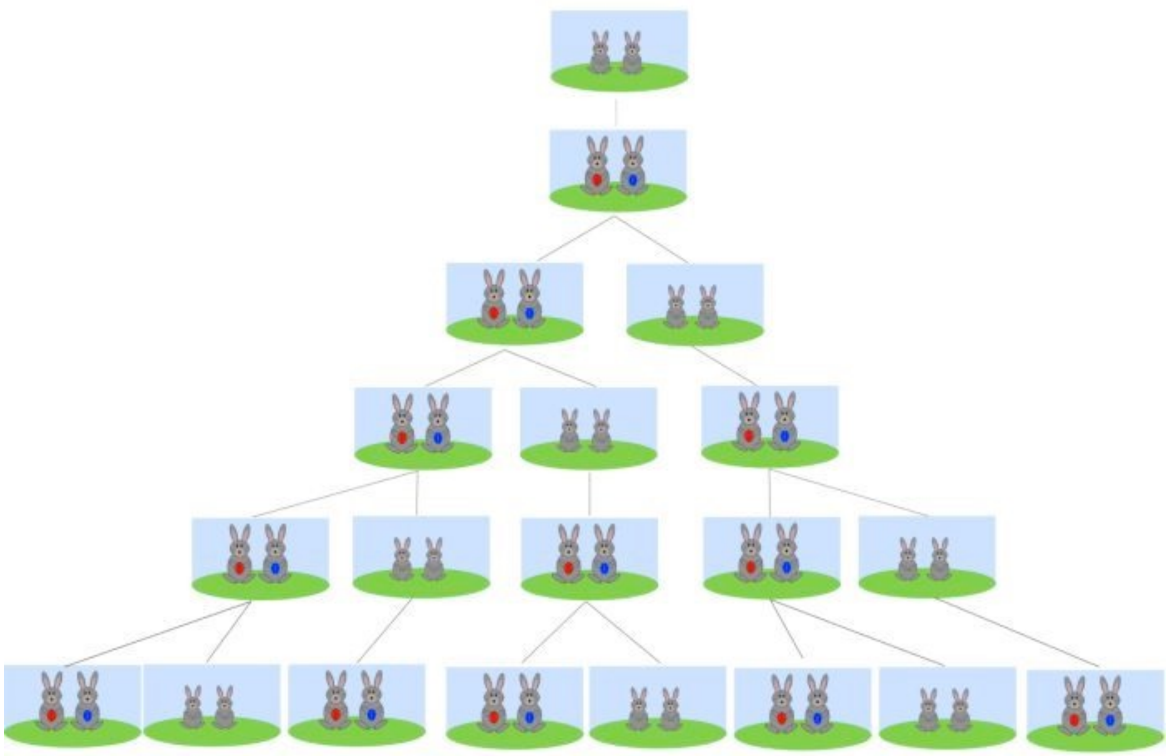
Rabbit island

- F_n: number of rabbit pairs at year n

- $F_1 = 1$ (only one pair)
- $F_2 = 1$ (too young to reproduce)
- $F_3 = 2$ (first pair of little rabbits)
- $F_4 = 3$ (second pair of little rabbits)
- $F_5 = 5$ (first pair of grandchildren)

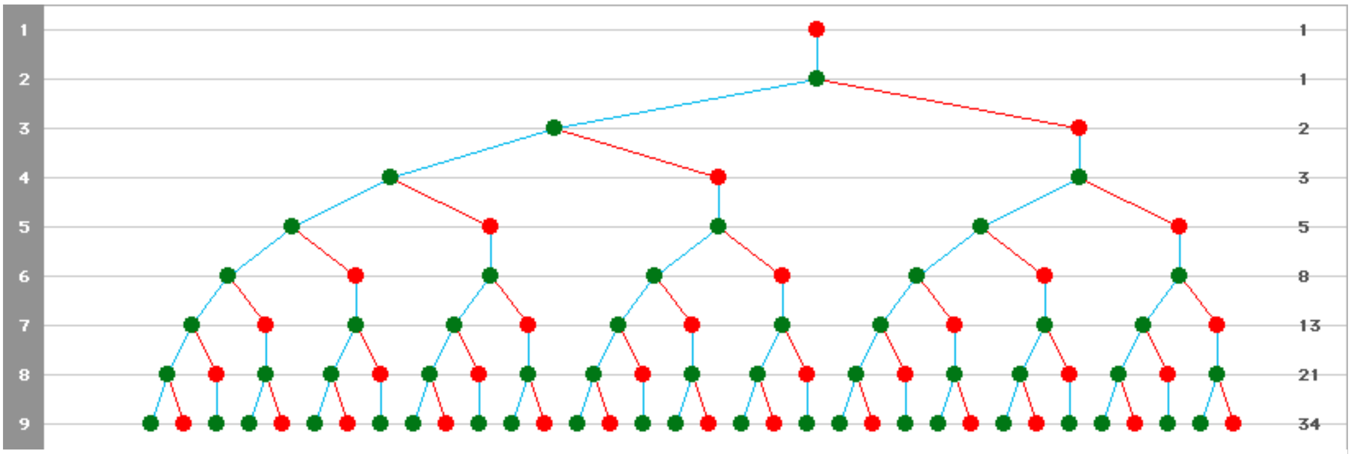
- For sake of completeness, assume $F_0 = 0$

A tree of rabbits



A tree of rabbits

- Rabbits will reproduce according to the following tree:



Rows are years

Red nodes: young pairs

Green nodes: adult pairs

Tree shown till year 9

Red line: birth of a new pair

Blue line: the pair continues next year

A general rule

- At year n there will be all pairs from the year before (year n-1) plus one new pair for each pair from two years before (year n-2)
- This yields the following recurrence:

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & \text{if } n \geq 3 \\ 1, & \text{if } n = 1, 2 \end{cases}$$

  – *Recurrence*: equation that expresses each element of a sequence as a function of the preceding ones

- Our (algorithmic) problem: how to compute F_n?

A possible approach

- Look up in textbooks and find that Fibonacci recurrence has the following solution:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$$

where

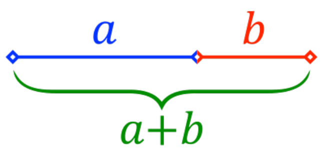
$$\phi = \frac{1+\sqrt{5}}{2} \approx +1.618$$

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

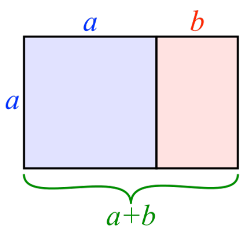
is the golden ratio

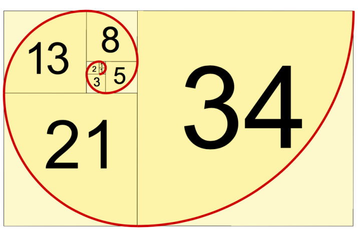
Golden ratio

$$\phi := \frac{a+b}{a} = \frac{a}{b}$$



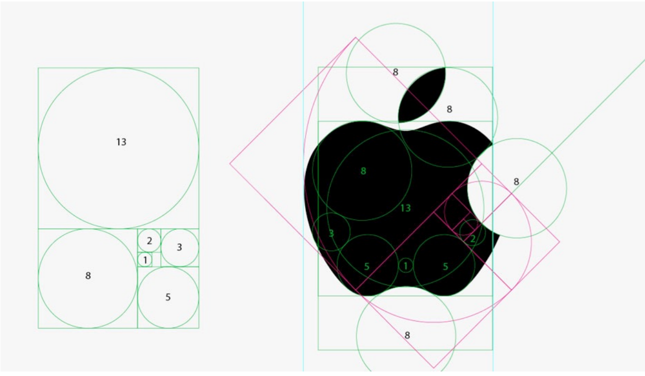
$a+b$ is to a as a is to b

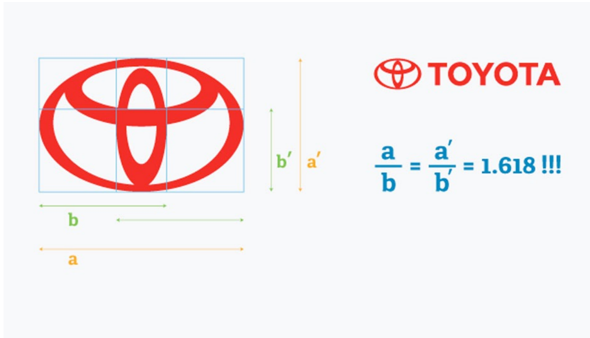


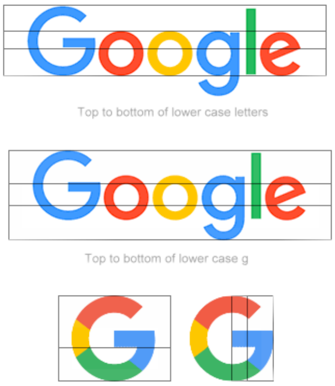


Golden ratio

- Many famous logos: Apple, Google, Toyota, …







Algorithm `fibonacci1`

$$\textbf{algorithm } \texttt{fibonacci1}(\mathit{integer}\ n) \rightarrow \mathit{integer}$$
$$\quad \textbf{return } \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$$

Algorithm `fibonacci1`

input (with data type) output data type

algorithm `fibonacci1`$(\boxed{integer\ n}) \to \boxed{integer}$

return $\boxed{\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)}$

output

- We represent algorithms using pseudocode
- Pseudocode: informal high-level description of the operating principle of an algorithm
- It uses the structural conventions of a programming language (e.g., return), but is intended for human reading rather than machine reading

Is `fibonacci1` correct?

- What's the accuracy needed for ϕ and $\hat{\phi}$ in order to get a correct output?
- E.g., if we only used 3 decimal digits:

$$\phi \approx 1.618 \text{ e } \hat{\phi} \approx -0.618$$

n	`fibonacci1`(n)	Rounding	F_n
3	1.99992	2	2
16	986.698	987	987
18	2583.1	2583	2584

Algorithm `fibonacci2`

- Since `fibonacci1` is not (always) correct, we could implement directly the recursive definition:

algorithm `fibonacci2`(*integer n*) $\rightarrow$ *integer*

if $n \leq 2$ **then return** 1

else return `fibonacci2`(n-1) + `fibonacci2`(n-2)

end if

Conditional statement:
if … then … else … end if

- Works only with natural numbers (non-negative integers)
- Is it a good solution?

Fast or slow algorithm?

- What is fast or slow for an algorithm?
- How much time does `fibonacci2` require?
- How do we measure time?
  – In seconds? Depends on hardware
  – In number of instructions? Depends on programming language and compiler
- First approximation:
  – Count number of lines in pseudocode containing instructions
  – Assume each code line requires 1 microsec (hardware and software independent)

Running time

```
algorithm fibonacci2(integer n) → integer
    if n ≤ 2 then return 1
    else return fibonacci2(n-1) + fibonacci2(n-2)
    end if
```

- If n ≤ 2, only 1 line of code
- If n = 3, 4 lines of code:
  - 2 lines for `fibonacci2`(3)
  - 1 line for `fibonacci2`(2)
  - 1 line for `fibonacci2`(1)

Running time

```
algorithm fibonacci2(integer n) → integer
    if n ≤ 2 then return 1
    else return fibonacci2(n-1) + fibonacci2(n-2)
    end if
```

- If n ≤ 2: 1 line of code (constant time)
- If n ≥ 3: 2 lines of code plus
  - lines of code for `fibonacci2`(n-1)
  - lines of code for `fibonacci2`(n-2)

Recurrence

- For n ≥ 3 `fibonacci2`(n) executes 2 lines of code, plus the lines of code executed in the recursive calls `fibonacci2`(n-1) and `fibonacci2`(n-2)

$$T_n = \begin{cases} 2 + T_{n-1} + T_{n-2}, & \text{if } n \geq 3 \\ 1, & \text{if } n = 1, 2 \end{cases}$$

- Except for the additional factor of 2, it is like Fibonacci sequence
- Indeed the solution to the recurrence is

$$T(n) = 3F_n - 2 \approx 3\phi^n$$

Can we do better?

- `fibonacci2` is very slow:

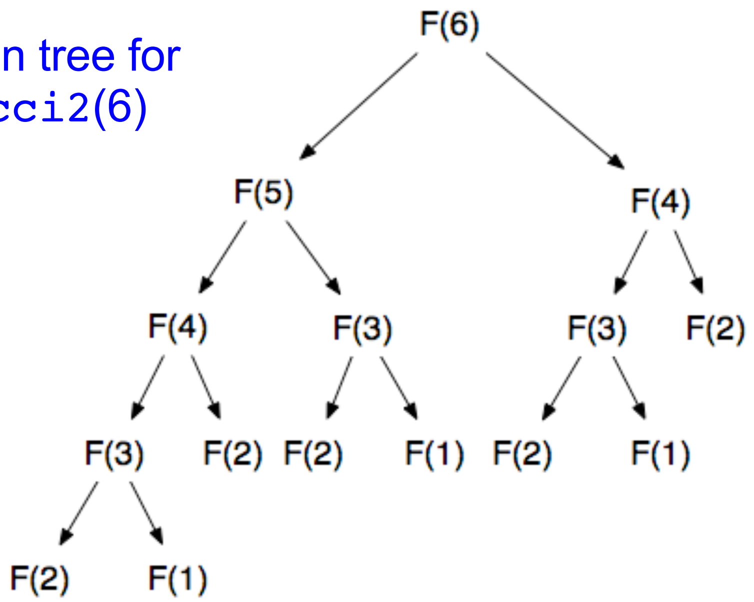
$$T(n) \approx F_n \approx \phi^n$$

- Can we do better?
- Why is `fibonacci2` very slow?

Recursion tree

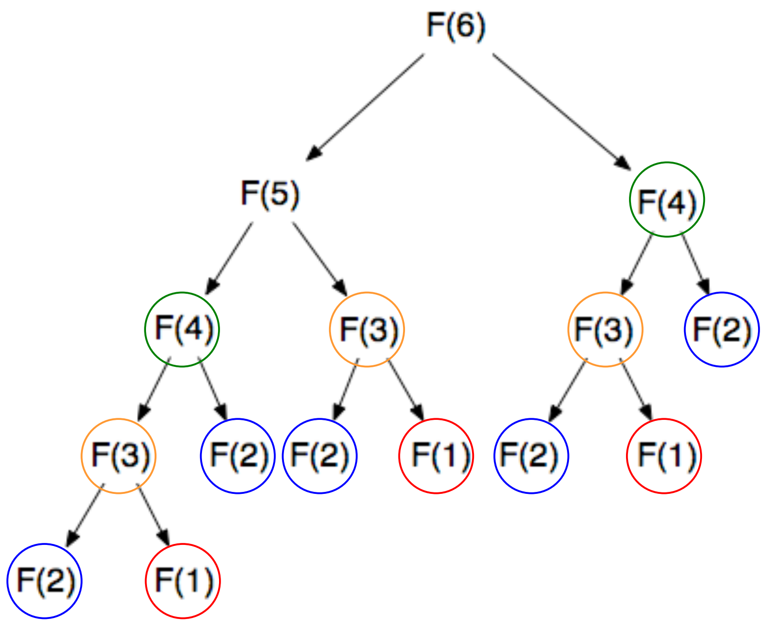
- Tree nodes correspond to recursive calls
- Children of a node correspond to recursive subcalls

Recursion tree for `fibonacci2(6)`



Recursion tree

- F(1) gets calculated 3 times, F(2) 5 times, F(3) 3 times, F(4) twice



Can we do better?

- `fibonacci2` is very slow:

$$T(n) \approx F_n \approx \phi^n$$

- Can we do better?

- Why is `fibonacci2` very slow?
- Keeps on recomputing the solution to the same subproblem over and over again!

Those who cannot remember the past are doomed to repeat it. (George Santayana)

Can we do better?

- How can we avoid to recompute the solution to the same subproblem over and over again?
- Just store the solution somewhere the first time you find it!

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

- Array named Fib: Fib[i] stores number F_i

Can we do better?

1	1	2	3	5	8	13	21	34	55	89	144
1	2	3	4	5	6	7	8	9	10	11	12

- Fib[i] stores F_i
- In order to compute Fib[i], let's use the F_{i-1} and F_{i-2} values stored in Fib[i-1] and Fib[i-2]

Algorithm fibonacci3

algorithm `fibonacci3`$(integer\ n) \rightarrow integer$

$\quad Let\ Fib\ be\ an\ array\ of\ n\ integers$
$\quad Fib[2] \leftarrow Fib[1] \leftarrow 1$
$\quad$ **for** i=3 **to** n **do**
$\quad\quad Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$
$\quad$ **end for**
$\quad$ **return** $Fib[n]$

What's new?

- $\leftarrow$ is the assignment instruction: copy value 1 into Fib[1] and Fib[2]
- for loop

For loop

$$\textbf{for } \mathrm{i{=}3} \textbf{ to } \mathrm{n} \textbf{ do}$$
$$\quad Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$$

- for loop: control flow statement for specifying iteration, which allows code to be executed repeatedly
- Typically used when the number of iterations to run is known before entering the loop

For loop

H) $\textbf{for } \mathrm{i{=}3} \textbf{ to } \mathrm{n} \textbf{ do}$

B) $\quad Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$

- For loop has two parts:
  - Header H specifies the iteration
  - Body B is executed once per iteration
  - Header defines a loop counter (in our case i), which allows the body to know which iteration is being executed
    - The counter is increased by 1 (in our case) on each loop iteration
    - When the value of the loop counter reaches the last iteration (in our case n), the loop will end and the flow continues to the next instruction after the loop

Algorithm fibonacci3

algorithm `fibonacci3`$(integer\ n) \rightarrow integer$
Let Fib be an array of n integers
$Fib[2] \leftarrow Fib[1] \leftarrow 1$
for i=3 **to** n **do**
  $Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$
end for
return $Fib[n]$

Example: let be n= 6

Fib

1	2	3	4	5	6

Fib

1	2	3	4	5	6
1	1				

Fib i=3

1	2	3	4	5	6
1	1	2			

Fib i=4

1	2	3	4	5	6
1	1	2	3		

Fib i=5

1	2	3	4	5	6
1	1	2	3	5	

Fib i=6

1	2	3	4	5	6
1	1	2	3	5	8

Memoization

- This technique of remembering previously computed values is called memoization

Never recompute a subproblem F(k), k≤n,
if it has been computed before

Is `fibonacci3` faster than `fibonacci2`?

```
algorithm fibonacci3(integer n) → integer
1  Let Fib be an array of n integers
2  Fib[2] ← Fib[1] ← 1
3  for i=3 to n do
4      Fib[i] ← Fib[i − 1] + Fib[i − 2]
   end for
5  return Fib[n]
```

- Lines 1, 2 and 5 are run only once
- Line 3 is run ≤ n times
- Line 4 is run ≤ n times
- How many times is the **for** cycle run? Roughly about n iterations (exactly n-3+1=n-2)

  $T(n) \leq 3 + n + n = 3 + 2n$

  We will see that $T(n) = O(n)$ i.e., $T(n)$ is order of n

`fibonacci3` vs. `fibonacci2`

- `fibonacci3` is much faster than `fibonacci2`, indeed 2n+3 is much better than $3F_n$-2
  - If n=8: 2n+3 = 19 $3F_n$-2 = 61
  - If n=45: 2n+3 = 93 $3F_n$-2 = 3,404,709,508
- We will see that `fibonacci3` runs in linear time, while `fibonacci2` runs in exponential time

Asymptotical notation

- Measuring running time T(n) as the number of lines of codes executed is a rough approximation
  - Because two different lines of code may have very different actual times
  - But it is sufficient for our purpose!

Asymptotical notation

- We aim to describe the order of magnitude of running time T(n) without taking into account low-level details, such as multiplicative or additive constants …
  - We are just interested in the growth rate of the running time
- We will use the notion of asymptotical notation O( ), called Big O notation

  It gives an upper bound: running time grows at most this much (but it could grow more slowly)

Asymptotical notation

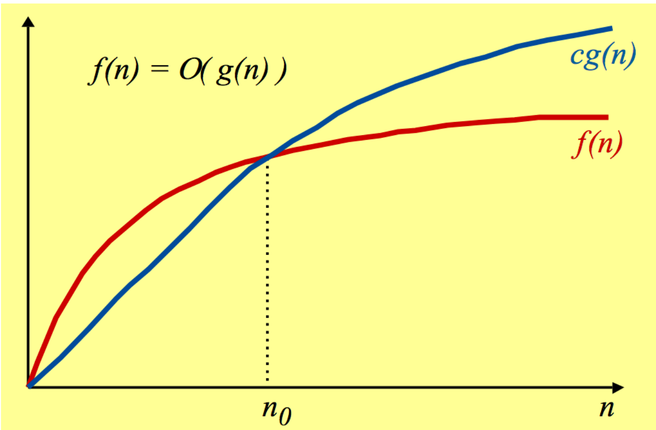
- For example, we can replace:

$T(n) = 2n$ and $T(n) = 4n$	by	$T(n) = O(n)$
$T(n) = 3F_n$	by	$T(n) = O(F_n)$
$T(n) = F_n$	by	$T(n) = O(2^n)$

- Big O notation makes our life easier:
  - Ignore low-level details
  - Compare easily different algorithms
  - `fibonacci3` runs in $O(n)$: much better than `fibonacci2` which runs in $O(2^n)$

Asymptotical notation

- We say that $f(n) = O(g(n))$ if $f(n) \leq c\,g(n)$ for some constant c, and large enough n



Example: $3n+8$ is $O(n)$ being $3n+8 \leq 4n$ for $n \geq 8$

- i.e., f(n) has $c\,g(n)$ as asymptotic upper bound, for some constant c

From pseudocode to code

- Pseudocode of `fibonacci2` can be easily translated into code using a programming language, e.g., Python

```
def fibonacci2(n):
        if n in {1, 2}: # Base case
                return 1
        return fibonacci2(n-1) + fibonacci2(n-2)
```


From pseudocode to code

- `fibonacci3` code in Python

```
def fibonacci3(n):
        # Initialize Fib with the first two values
        Fib = [0, 1, 1]
        while len(Fib) <= n:
        # Calculate the next value by adding the last two values
                next_value = Fib[len(Fib)-1] + Fib[len(Fib)-2]
                # Append the next value to the sequence
                Fib.append(next_value)
        return Fib[n]
```

- Let's run the two programs when n=40 and see the difference in running time!

Exercise

- Midterm-like problem
- "Tribonacci" numbers have been defined by professor Tribonacci as follows:

$$Tr_n = \begin{cases} Tr_{n-1} + Tr_{n-2} + Tr_{n-3}, & \text{if } n \geq 3 \\ 1, & \text{if } n = 1, 2 \\ 0, & \text{if } n = 0 \end{cases}$$

Exercise

- Prof. Tribonacci claims that the following is the fastest algorithm for computing the n-th Tribonacci number:

algorithm `tribonacci`(*integer n*) → *integer*

    if $n = 0$ **then return** 0
    else if $n \leq 2$ **then return** 1
    else return `tribonacci`(n-1) + `tribonacci`(n-2) + `tribonacci`(n-3)
    end if

Exercise

- What do you think is the running time of algorithm `tribonacci`?
  - (You do not need to prove this)
- Can you find a faster algorithm for the same problem?
- What is the running time of your algorithm?
  - (You need to prove this)

References

- Fibonacci Sequence
- Video: A. Benjamin, The magic of Fibonacci numbers
- The Rabbit Hole of Fibonacci Sequences, Recursion and Memoization
- C. Demetrescu, I. Finocchi, G. F. Italiano, "Algoritmi e Strutture Dati", Mc-Graw Hill, 2008 (in Italian)