TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

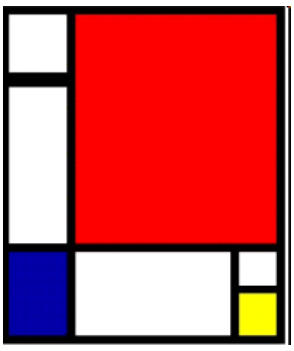
Algorithm Design: Fibonacci Numbers

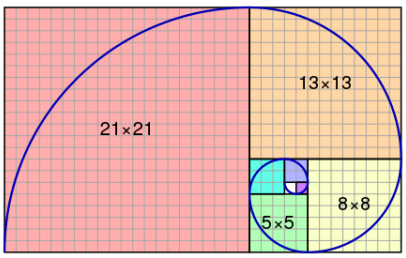
Algorithms, Data and Security
A.Y. 2025/26

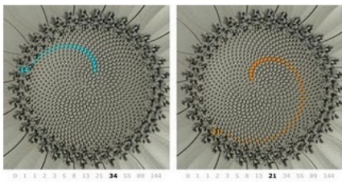
Valeria Cardellini

Global Governance, 3rd year
Science and Technology Major

Fibonacci numbers in art and nature



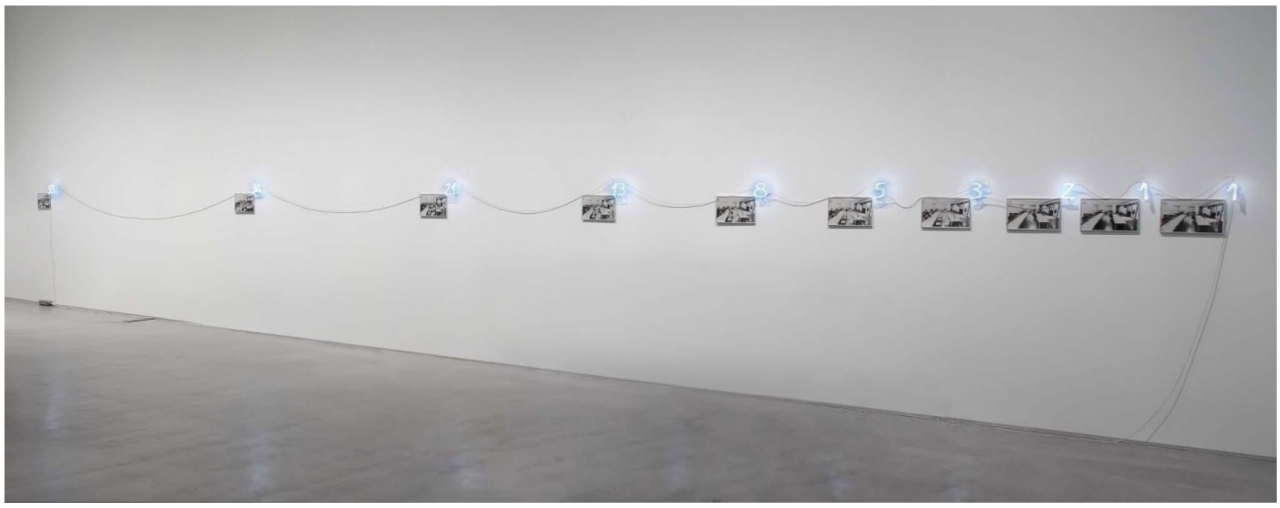




Fibonacci numbers in nature

- An example of efficiency in nature
- Sunflower seeds, pine cones, and flower petals arrange themselves in spirals to fit 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 numbers at the museum



Fibonacci Napoli [Factory in San Giovanni a Teduccio]
Mario Merz, 1971
Museo Reina Sofia, Madrid

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

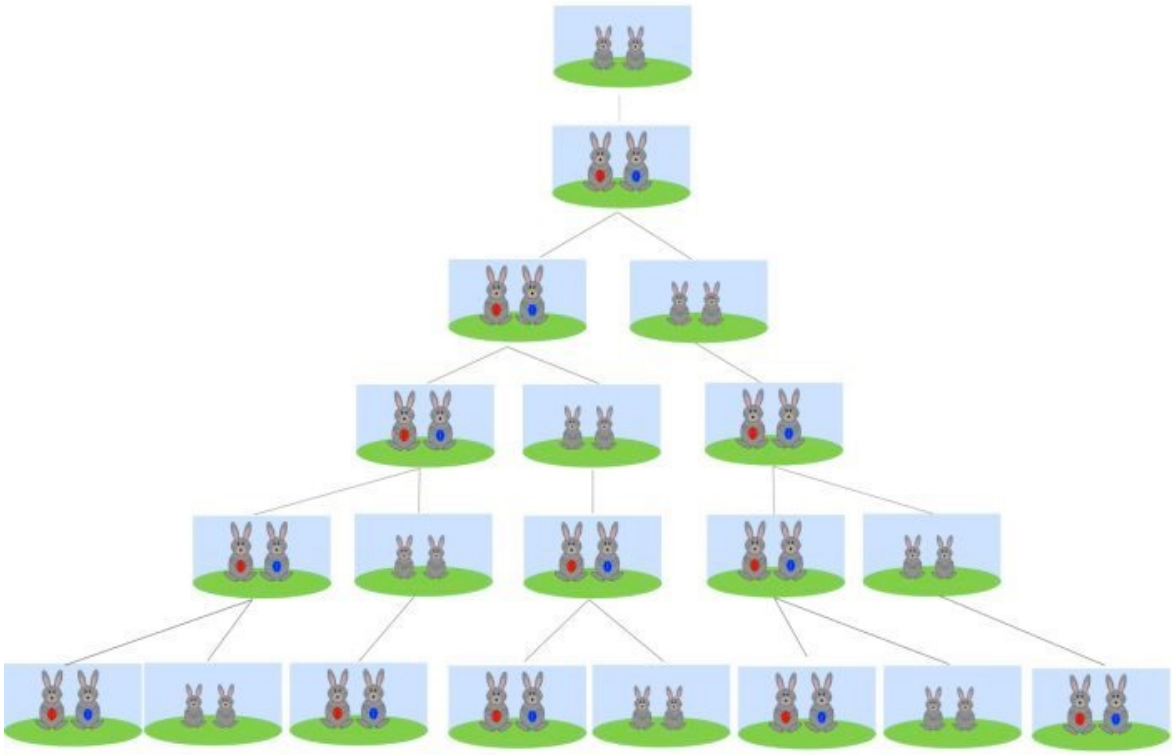
Rabbit island

- 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)

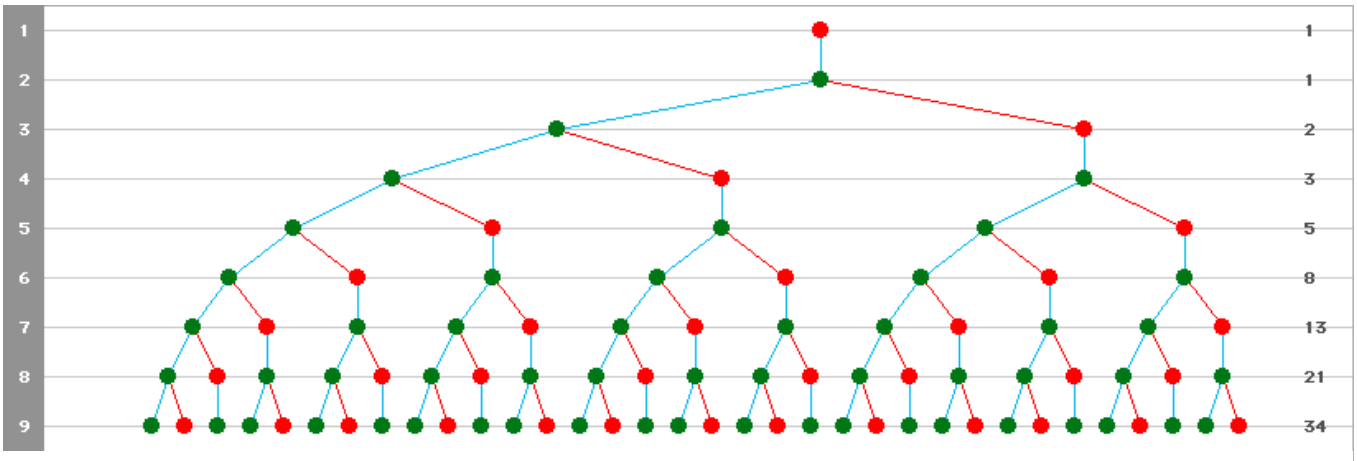
- For sake of completeness, assume $F_0 = 0$

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

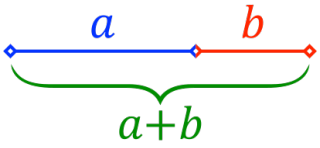
$$\phi = \frac{1+\sqrt{5}}{2} \approx +1.618$$

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

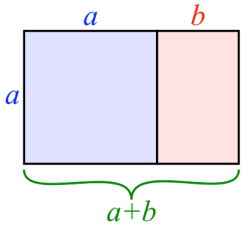
is the golden ratio

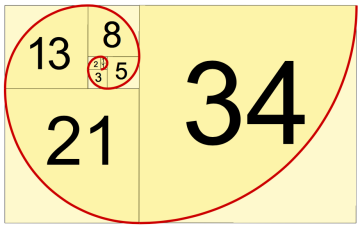
Golden ratio

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



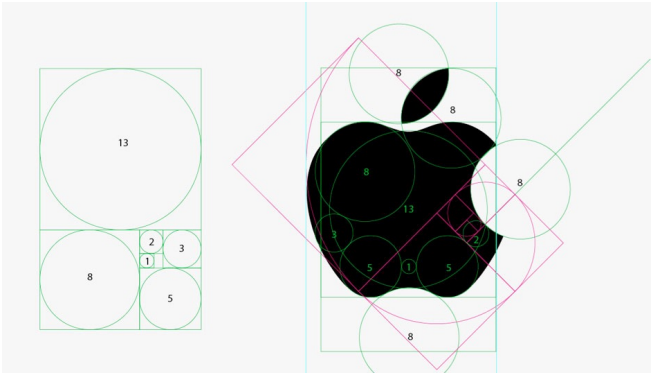
$a+b$ is to a as a is to b

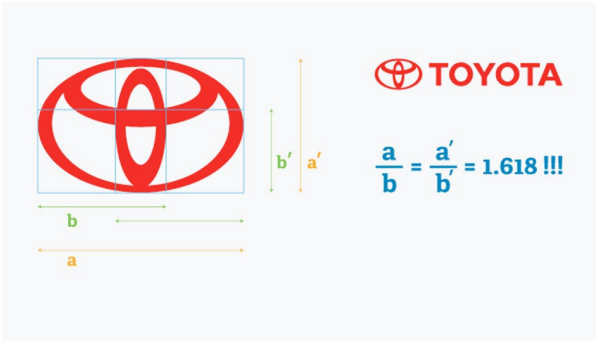




Golden ratio

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





Algorithm `fibonacci1`

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

Algorithm `fibonacci1`

input (with data type) output data type

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

output

- We represent algorithms using pseudocode
- Pseudocode: informal, high-level description of an algorithm's operation
  - It uses the structural conventions of a programming language (e.g., return), but is designed for human understanding rather than machine execution
  - Goal: focus on algorithm's logic without worrying about syntax of specific programming language

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

- For large n, the rounding errors will lead to incorrect results

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?

Programming: conditional statement

- A conditional statement executes different code based on whether a certain condition is true or false

if condition **then** *statement*
else *elsestatement*
end if

- Meaning: if condition is true then execute *statement*, otherwise execute *elsestatement*

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 for `fibonacci2`

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

Running time for `fibonacci2`

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

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

Again a recurrence

- For n ≥ 3 `fibonacci2`(n) executes 2 lines of code, plus the lines of code for the recursive calls to `fibonacci2`(n-1) and `fibonacci2`(n-2): the running time is

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

- Except for the additional factor of 2, this matches the Fibonacci recurrence (see slide 10)
- Indeed, its solution is

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

Can we do better?

- `fibonacci2` is very slow

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

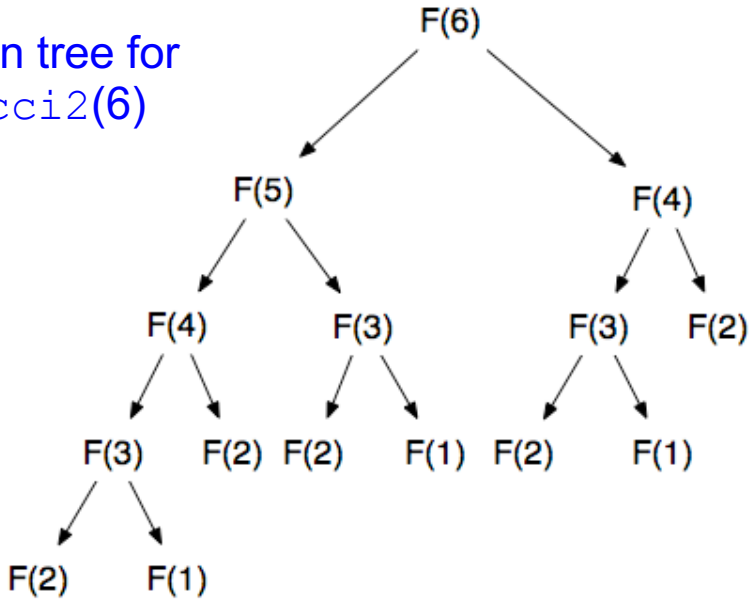
- Why is `fibonacci2` very slow?

- Can we do better?

Recursion tree

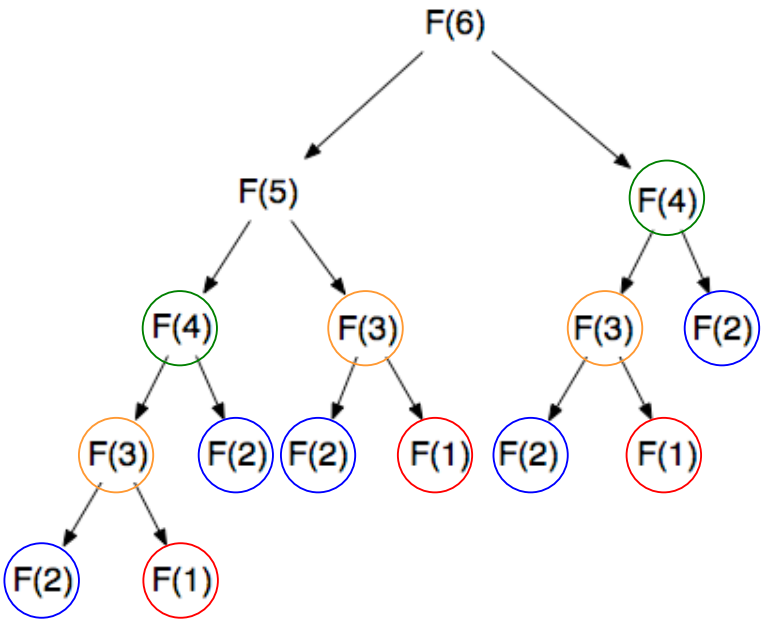
- Tree nodes correspond to recursive calls
- Children of a node correspond to recursive subcalls

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:

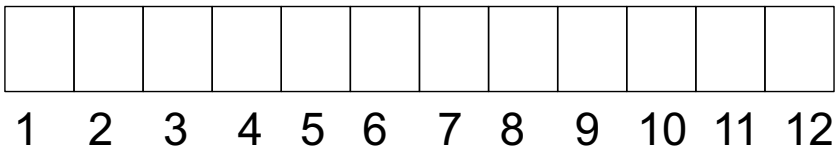
$$T(n) \approx F_n \approx \phi^n$$

- 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)

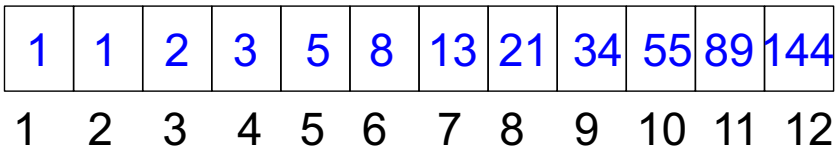
Can we do better?

- 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 compute it!



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

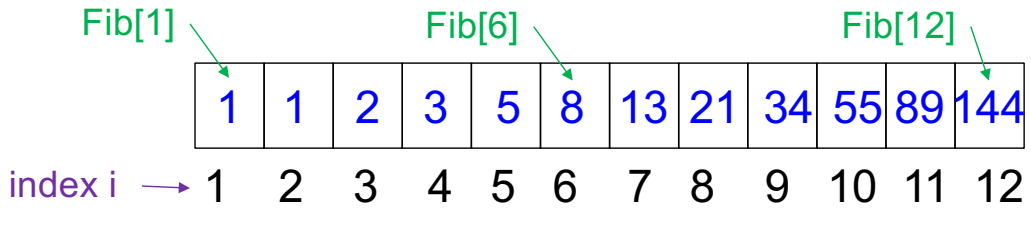
Can we do better?



- 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]

Programming: array

- Array: data structure that stores a collection of elements of the same type
  - In our case: Fib is an array of integer values
  - Fib is the name of the array
  - It stores integers (numbers like 5, 21, 144)
  - Each element in Fib holds one integer value
- Each element has a position number (index)
  - In our case: i is the index



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 $Fib[n]$

What's new?

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

2) for loop

Programming: for loop

for i=3 **to** n **do**
    $Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$

- for loop: control flow statement use to repeat a block of code multiple times (iterations)
- Typically used when the number of iterations to run is known *before* the loop starts
  - In our case: we run the loop n-2 times

Programming: for loop

header **for** i=3 **to** n **do**

body $Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$

- A for loop has two parts:
  - Header: defines the loop counter (in our case i) and specifies the number of iterations
  - Body: contains the code executed during each iteration of the loop
- The loop counter allows the body to track which iteration is currently running

Programming: for loop

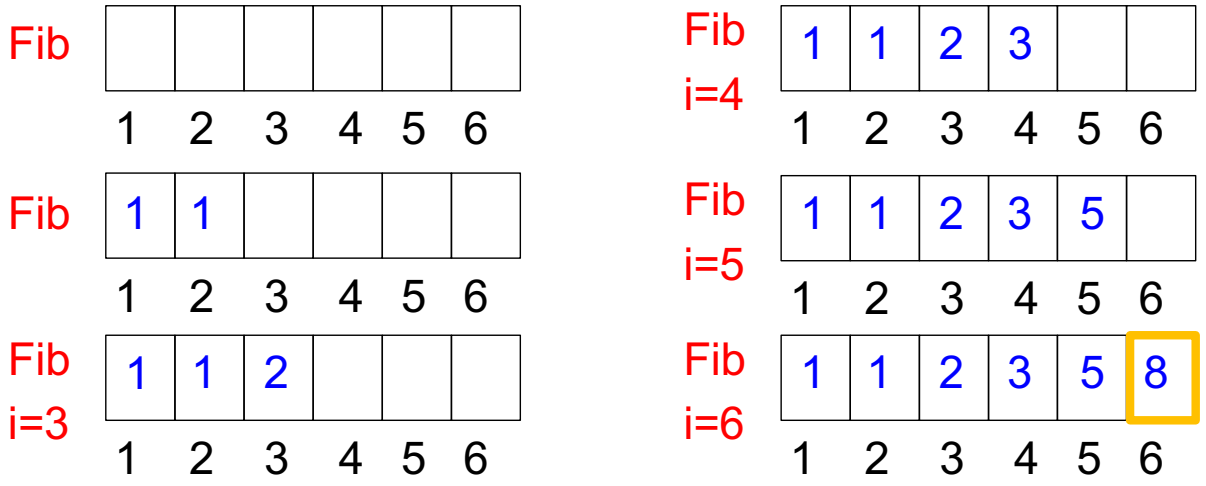
for i=3 **to** n **do**
  $Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$

- How it works:
  - Initialization: the loop counter starts at a specific value (in our case 3)
  - Execution: the body runs using the current counter value (3, 4, …)
  - Increment: after each iteration, the counter increments by 1 (4, 5, …)
  - Condition check: the loop repeats as the counter is less than or equal to the limit (n)
  - Termination: once the limit is exceeded, the program moves to the next instruction after the loop (return Fib[n])

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



Memoization

- This technique of remembering computed values is called memoization

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

Is `fibonacci3` faster than `fibonacci2`?

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

```
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: run only once
- Line 3: runs ≤ n times
- Line 4: runs ≤ n times
- How many times is the **for** cycle run? Roughly n iterations (specifically n-2)

  T(n) ≤ 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 lines of codes executed is a rough approximation
  - Because two different lines may have very different execution times
  - However, this level of detail is sufficient for our purpose!

Asymptotical notation

- Goal: describe the order of magnitude of running time T(n) without taking into account low-level details (e.g., multiplicative or additive constants)
  - We are interested in the growth rate of the running time
- Tool: we use the asymptotical notation O( ) or Big O notation

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

Asymptotical notation

- For example, we can replace:

  $T(n) = 2n + 3$ by $T(n) = O(n)$

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

- Big O notation makes our life easier:
  - Ignore low-level details
  - Compare easily different algorithms: show why `fibonacci3` is superior to `fibonacci2`
    - `fibonacci3` runs in O(n): linear time
    - Much better than `fibonacci2`, which runs in $O(2^n)$: exponential time

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):
    Fib = [0] * (n + 1)
    # Initialize Fib with the first two values
    Fib[1] = 1
    Fib[2] = 1

    for i in range(3, n+1):
        # Calculate the next value by adding the last two values
        Fib[i] = Fib[i-1] + Fib[i-2]
    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 ≤ 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 https://www.mathsisfun.com/numbers/fibonacci-sequence.html
- Video: A. Benjamin, The magic of Fibonacci numbers https://www.youtube.com/watch?v=SjSHVDfXHQ4
- The Rabbit Hole of Fibonacci Sequences, Recursion and Memoization https://medium.com/@sarahsweat/the-rabbit-hole-of-fibonacci-sequences-recursion-and-memoization-f5c34079a488
- C. Demetrescu, I. Finocchi, G. F. Italiano, “Algoritmi e Strutture Dati: terza edizione”, Mc-Graw Hill, 2025 (in Italian)