Recursive algorithms Separate and conquer

Lecture 06.01

by Marina Barsky

Recap: recursion

```
algorithm print_num(count):
    if count < 1:
        return
    print(count)
    print_num(count-1)

print_num(4)
```

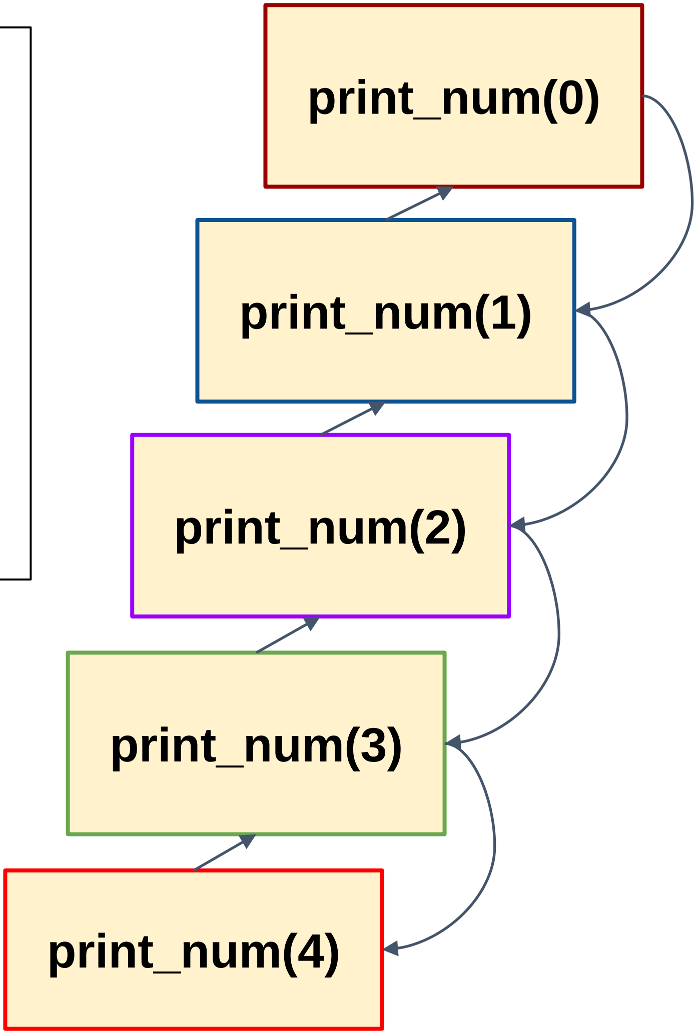
What is printed?

Recap: recursion

```
algorithm print_num(count):
    if count < 1:
        return
    print(count)
    print_num(count-1)

print_num(4)
```

What is printed?



Recap: recursion

```
algorithm print_num(count):
    if count < 1:
        return
    print_num(count-1)
    print(count)


print_num(4)
```

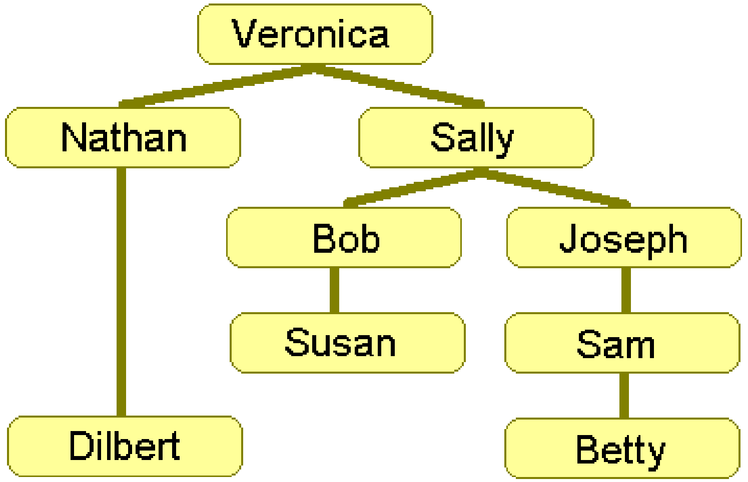
What is printed now?

When to use recursion

1. The structure is defined recursively
2. The problem is defined recursively

1. Recursively defined structure: tree

Name	Manager
Betty	Sam
Bob	Sally
Dilbert	Nathan
Joseph	Sally
Nathan	Veronica
Sally	Veronica
Sam	Joseph
Susan	Bob
Veronica	



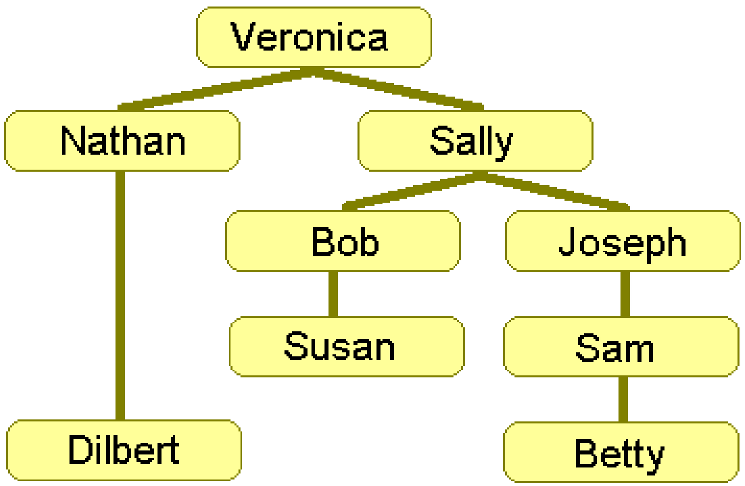
Employee names are stored as pairs of names in list *A*

They represent a hierarchical organizational structure: *X* reports to *Y*

Problem: Count Under

Input: list *A* of pairs (*x*,*y*) meaning *x* reports directly to *y*, and an employee name *S.*

Output: count of all employees who report to *S* (directly or indirectly)



Designing solution

Name	Manager
Betty	Sam
Bob	Sally
Dilbert	Nathan
Joseph	Sally
Nathan	Veronica
Sally	Veronica
Sam	Joseph
Susan	Bob
Veronica	

- We want to find all people who work under S='Sally'
- We can iterate over list A and count the pairs where Sally is a manager
  - But then we also need to iterate over A again and count and collect people working under Bob and Joseph
  - Then all people who work under Sam
- It seems that we need to have several nested loops - but how many levels?

The easiest solution: use ***recursion***!

Recursive solution

Algorithm *count_under*(list *A* of pairs, name *S*)

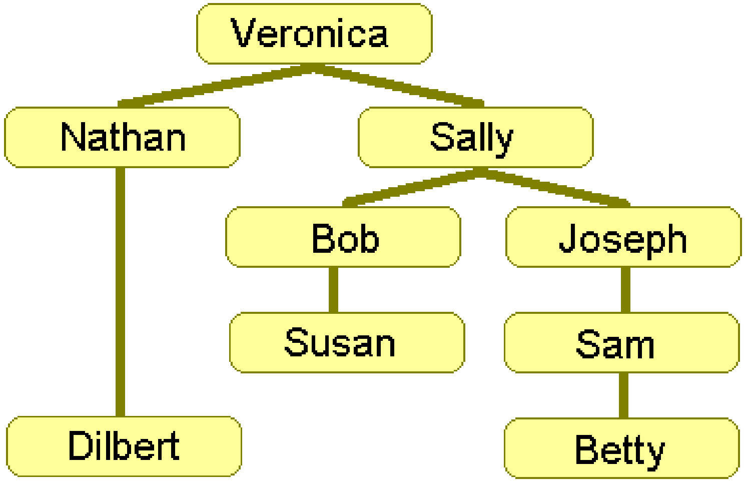
count: = 0

for each pair (name, manager) **in** A:

if manager = S:

count: += 1 + ***count_under***(name)

return count



2. Recursively defined problem: factorial

Definition

factorial(1) = 1
factorial(*n*) = *n***factorial*(*n*-1)

$$F_n = n * F_{n-1}$$

Problem: compute factorial

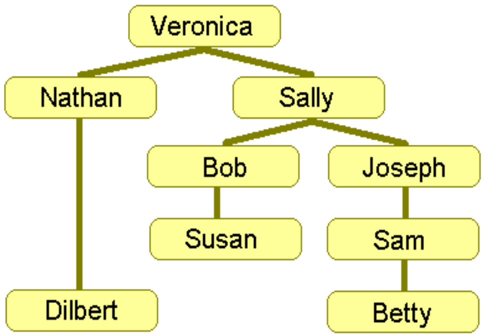
Input: *n*
Output: *factorial*(*n*)

Algorithm *factorial*(*n*)

if *n* = 1 **return** 1
return *n**factorial(*n* -1)

When recursion feels natural

Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms



$$F_n = n * F_{n-1}$$

Definition

A ***recurrence relation*** is an equation recursively defining a sequence of values

$$F_n = \begin{cases} 1, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ n * F_{n-1} \quad \text{if } n > 1 \end{cases}$$

1, 1, 2, 6, 24, 120…

$$F_n = \begin{cases} 0, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ F_{n-1} + F_{n-2} \quad \text{if } n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Note how the function F(n) is defined through F(n-1)

Fibonacci numbers

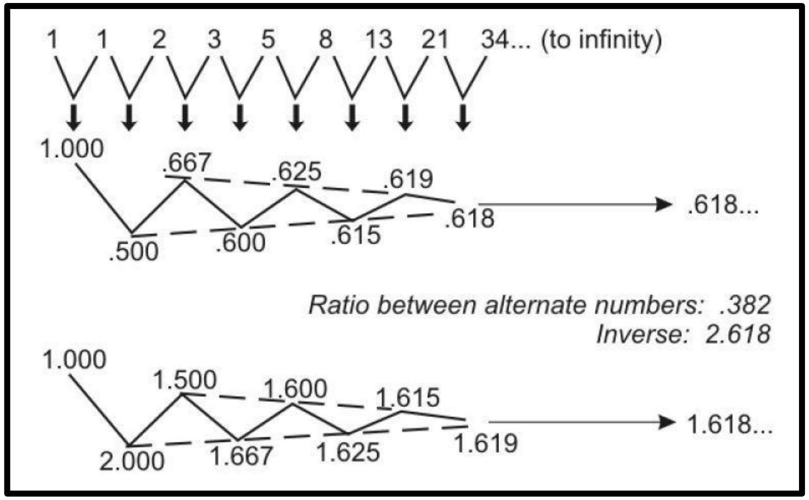
Natural recursive solution?

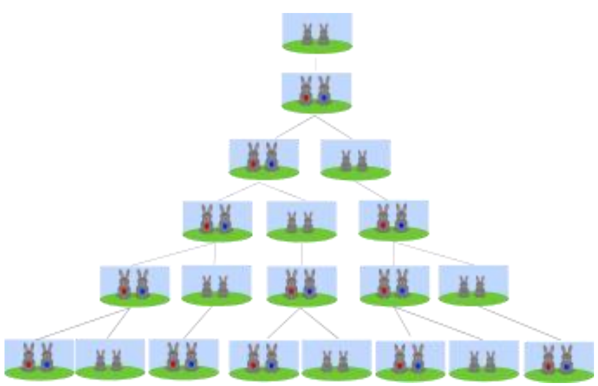
Fibonacci numbers

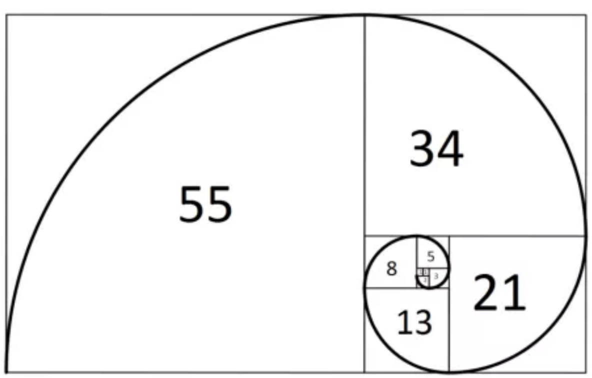
$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

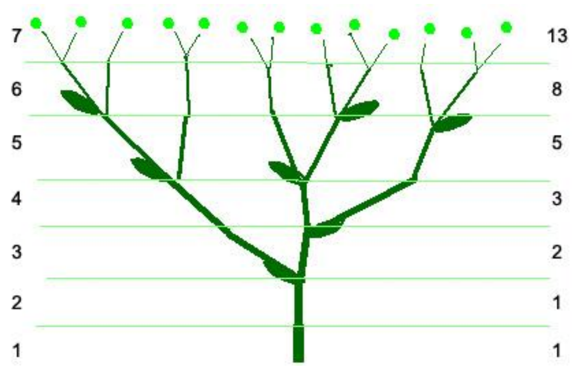
0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Golden ratio



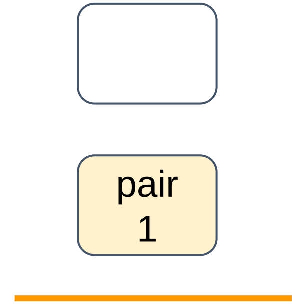






Fibonacci rabbits

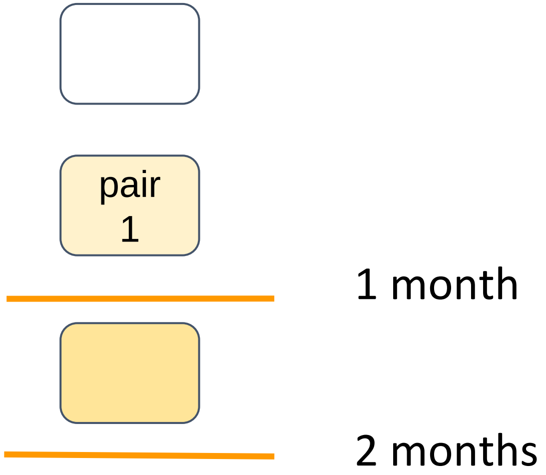
Leonardo
Fibonacci
c1175-1250



1 month

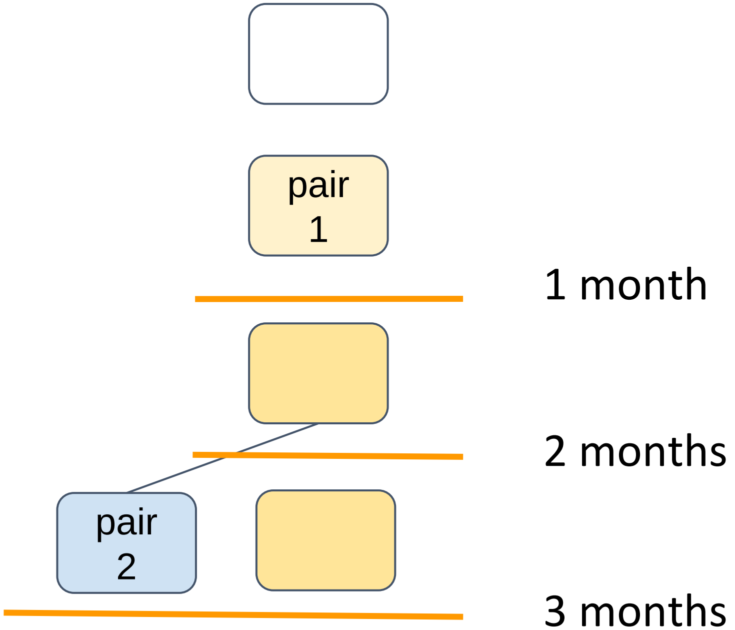
Fibonacci rabbits

Leonardo
Fibonacci
c1175-1250



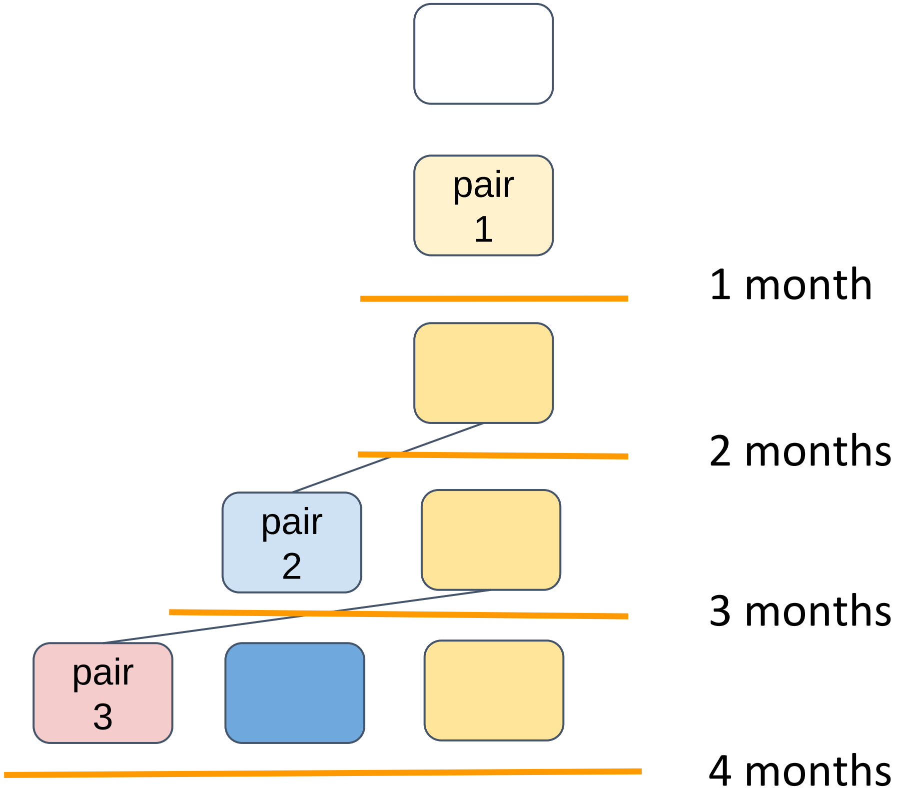
Fibonacci rabbits

Leonardo
Fibonacci
c1175-1250



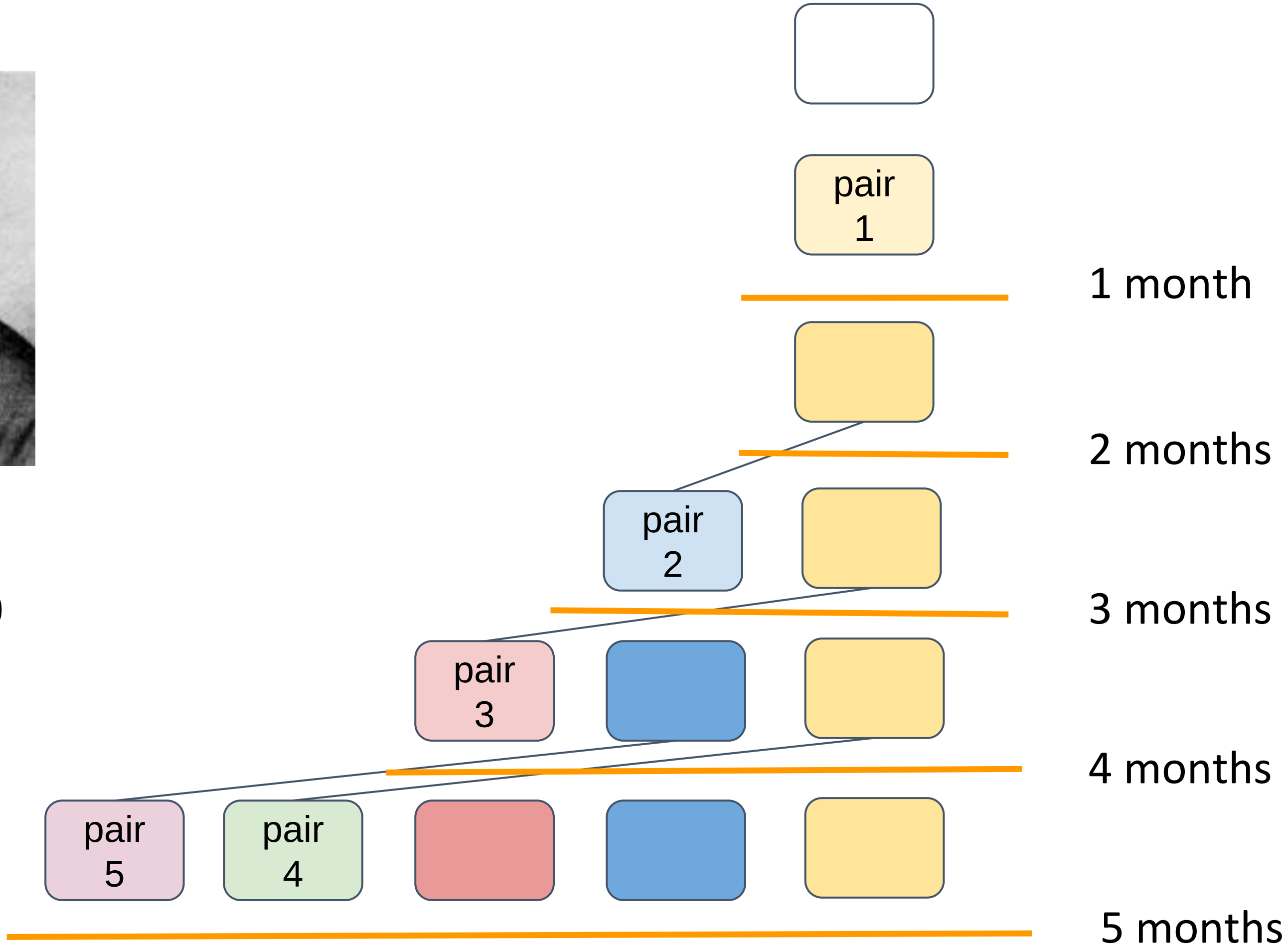
Fibonacci rabbits

Leonardo
Fibonacci
c1175-1250



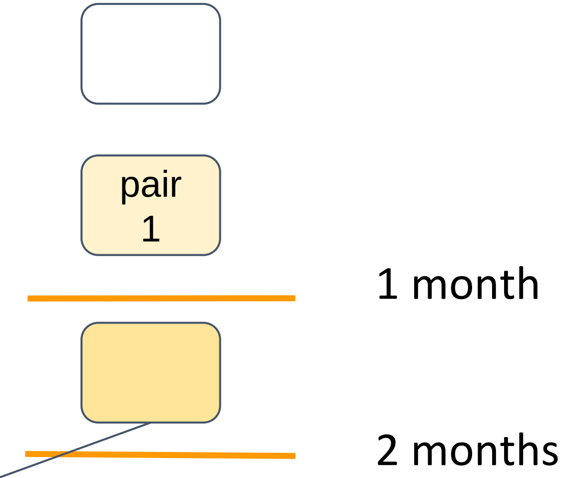
Fibonacci rabbits

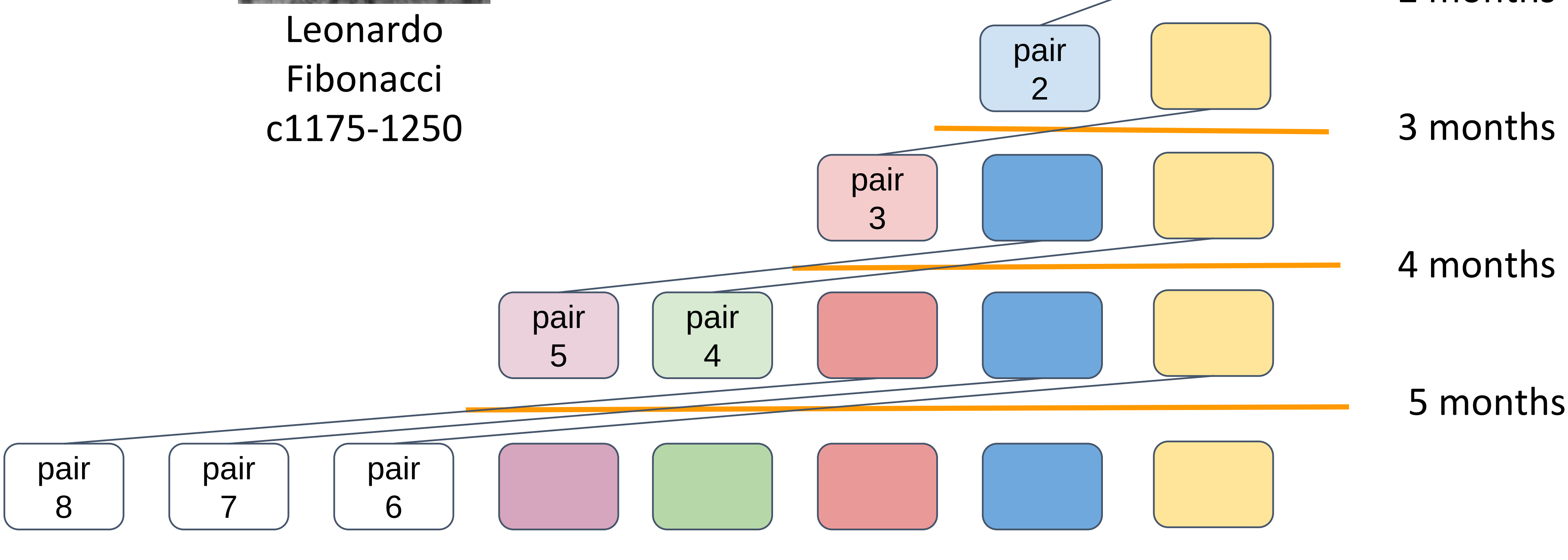
Leonardo
Fibonacci
c1175-1250



Fibonacci rabbits

Leonardo
Fibonacci
c1175-1250





It all started with a single pair…

University of Victoria, BC, Canada, 2010

Fibonacci numbers grow exponentially

Lemma

$F_n \geq 2^{n/2}$ for $n \geq 6$

Proof: By induction

Base case: n = 6, 7 (by direct computation).

Inductive step:

Assume that it is true for F_{n-1}: $F_{n-1} >= 2^{(n-1)/2}$.

Let's show that it is true for F_n

$F_n = F_{n-1} + F_{n-2}$

$\geq 2^{(n-1)/2} + 2^{(n-2)/2} \geq 2 \cdot 2^{(n-2)/2} = 2^{n/2}$ ■

Theorem:

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

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

$\phi = 1.618034...$

$F_{20} = 6765$

$F_{50} = 12586269025$

$F_{100} = 354224848179261915075$

$F_{500} = 139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125$

Recursive algorithm for computing the *n*-th Fibonacci number

$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Problem: Compute F_n

Input: integer n >= 0
Output: F_n

Algorithm Fib_recurs(*n*)

if $n \leq 1$: return n

return Fib_recurs($n - 1$) + Fib_recurs($n - 2$)

What is the running time?

Recursive Fibonacci: running time

Algorithm Fib_recurs(n)

```
if n ≤ 1:
  return n
else:
  return Fib_recurs(n – 1) + Fib_recurs(n – 2)
```

Let $T(n)$ denote the count of **lines of code** executed by Fib_recurs(n).

if $n \leq 1$: $T(n) = 2$

if $n \geq 2$: $T(n) = 3 + T(n-1) + T(n-2)$

Number of operations

$$T(n) = \begin{cases} 2 \text{ if n <= 1} \\ 3 + T(n\text{-}1) + T(n\text{-}2) \end{cases}$$

n-th Fibonacci number

$$F_n = \begin{cases} 0, \text{ if n = 0} \\ 1, \text{ if n = 1} \\ F_{n-1} + F_{n-2} \quad \text{if n > 1} \end{cases}$$

Therefore $T(n) \geq F_n$

Recursive algorithm: running time

Let $T(n)$ denote the count of **lines of code** executed by Fib_recurs(n).

Algorithm Fib_recurs(n)

```
if n ≤ 1:
  return n
else:
   return Fib_recurs(n – 1) + Fib_recurs(n – 2)
```

$$T(n) = \begin{cases} 2 \text{ if } n <= 1 \\ 3 + T(n-1) + T(n-2) \end{cases}$$

$T(n) \geq F_n$

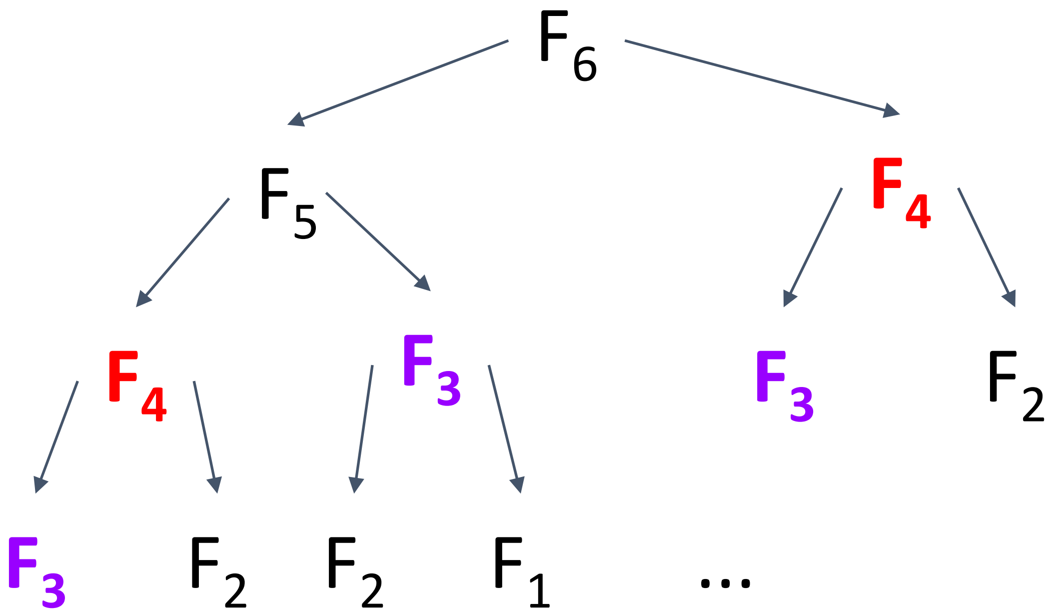
but $F_n \geq 2^{n/2}$ for $n \geq 6$!!!

Running time **$\Omega(2^n)$**

$T(100) \approx 1.77 \cdot 10^{21}$
(1.77 sextillion)

Takes 56,000 years at 1GHz

Why so slow?



Recursion tree

Note the repeating calls with the same arguments

Efficient iterative algorithm

Algorithm Fib_list(n)

create an array $F[0 \ldots n]$

$F[0] \leftarrow 0$

$F[1] \leftarrow 1$

for i from 2 to n:

  $F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

Running time

$T(n) = 2n + 2$

So $T(100) = 202$

Recursion or not recursion?

Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms

- Such recursive definitions do not guarantee that a recursive algorithm is the best way to solve the problem
- The use of recursive algorithms by inappropriate examples created apprehension and antipathy toward the use of recursion in programming:

recursion = inefficiency

Recursion vs. iteration

➔**Recursion**

- Each recursive call **requires extra space** on the stack
- If we get **infinite recursion**, the program will eventually run out of memory, cause stack overflow, and **the program will terminate**
- Solutions to some problems are **easier to formulate** recursively

➔**Iteration**

- Each iteration **does not require extra space**
- An **infinite loop could loop forever** since there is no extra memory being created
- Iterative **solutions** to a problem may **not** always be as **obvious** as a recursive solution

Generally, recursive solutions are less efficient than iterative solutions due to the overhead of function calls

Recursive algorithms: running time

Steps:

- Draw recursion tree
- Estimate the depth of the tree
- Estimate work done at each level of the tree
- Add all level work together

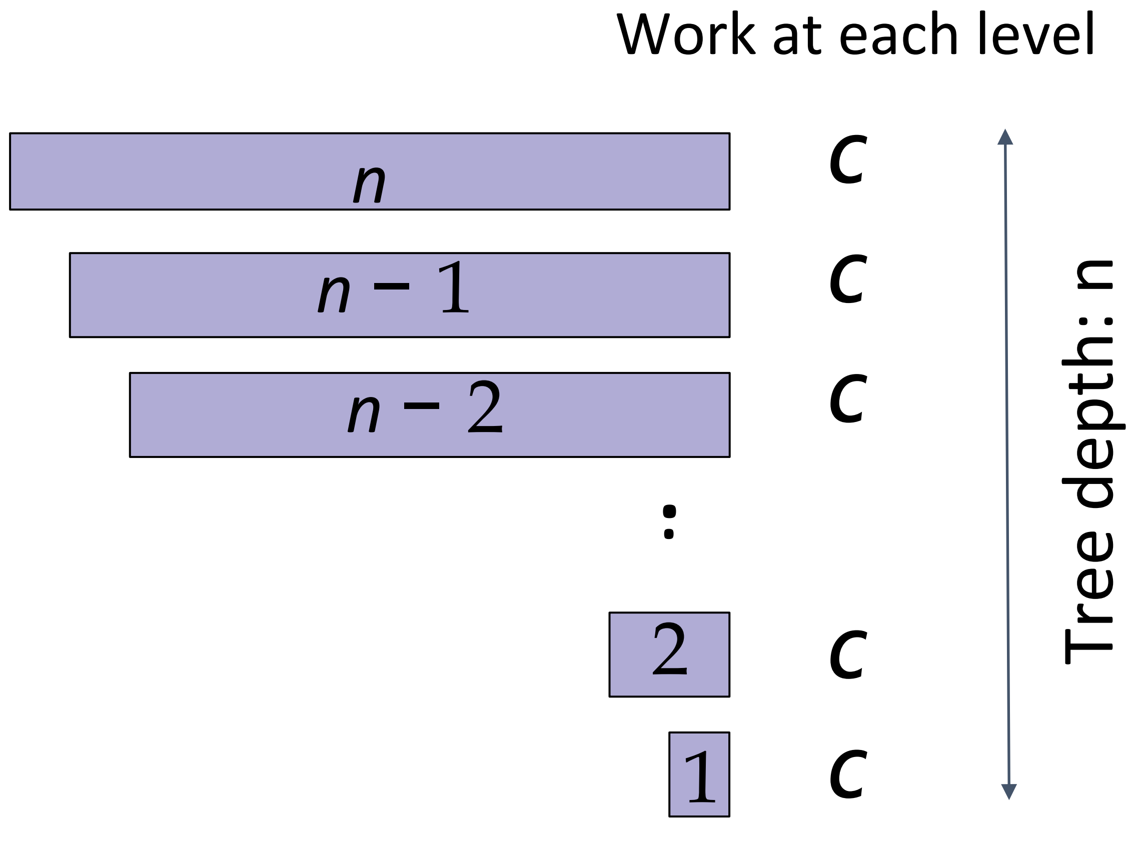
Example: recursive max

Algorithm Recurs_max (non-empty linked list A)

```
if len(A) = 1:
        return A[1]
else:
        if A[1] < A[2]:
                remove A[1] from A
        else:
                remove A[2] from A
        return  Recurs_max (A)
```

What is time complexity?

Recursive Max: time complexity



Total: $cn = \mathrm{O}(n)$

Searching Sorted Data

Separate and conquer

https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

Warm-up: find the fake coin

- There are 8 identical-looking coins
- One of these coins is counterfeit and is known to be lighter than the genuine coins

- What is the minimum number of weighings needed to identify the fake coin with a two-pan balance scale without weights?

Problem: Searching in a sorted array

Input: A sorted array $A[low \; . \; . \; . \; high]$ $(\forall low \leq i < high : A[i] \leq A[i+1])$.

A value ***key*** to search for.

Output: An index, i , $(low \leq i \leq high)$ where $A[i] = key$.

Otherwise, return -1 (NOT_FOUND).

binary_search(A, low, $high$, key)

if $high < low$:

  return -1

$mid \leftarrow low + \left\lfloor \frac{high - low}{2} \right\rfloor$

if $key = A[mid]$:

  return mid

else if $key < A[mid]$:

  return binary_search(A, low, $mid - 1$, key)

else:

  return binary_search(A, $mid + 1$, $high$, key)

Example: Searching for key 50

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

Example: Searching for key 50

binary_search(A, 0, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *high*

Example: Searching for key 50

binary_search(A, 0, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

binary_search(A, 0, 10, 50)

binary_search(A, 6, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

binary_search(A, 0, 10, 50)

binary_search(A, 6, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

binary_search(*A*, 0, 10, 50)

binary_search(*A*, 6, 10, 50)

binary_search(*A*, 9, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

binary_search(A, 0, 10, 50)

binary_search(A, 6, 10, 50)

binary_search(A, 9, 10, 50)

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

low *mid* *high*

Example: Searching for key 50

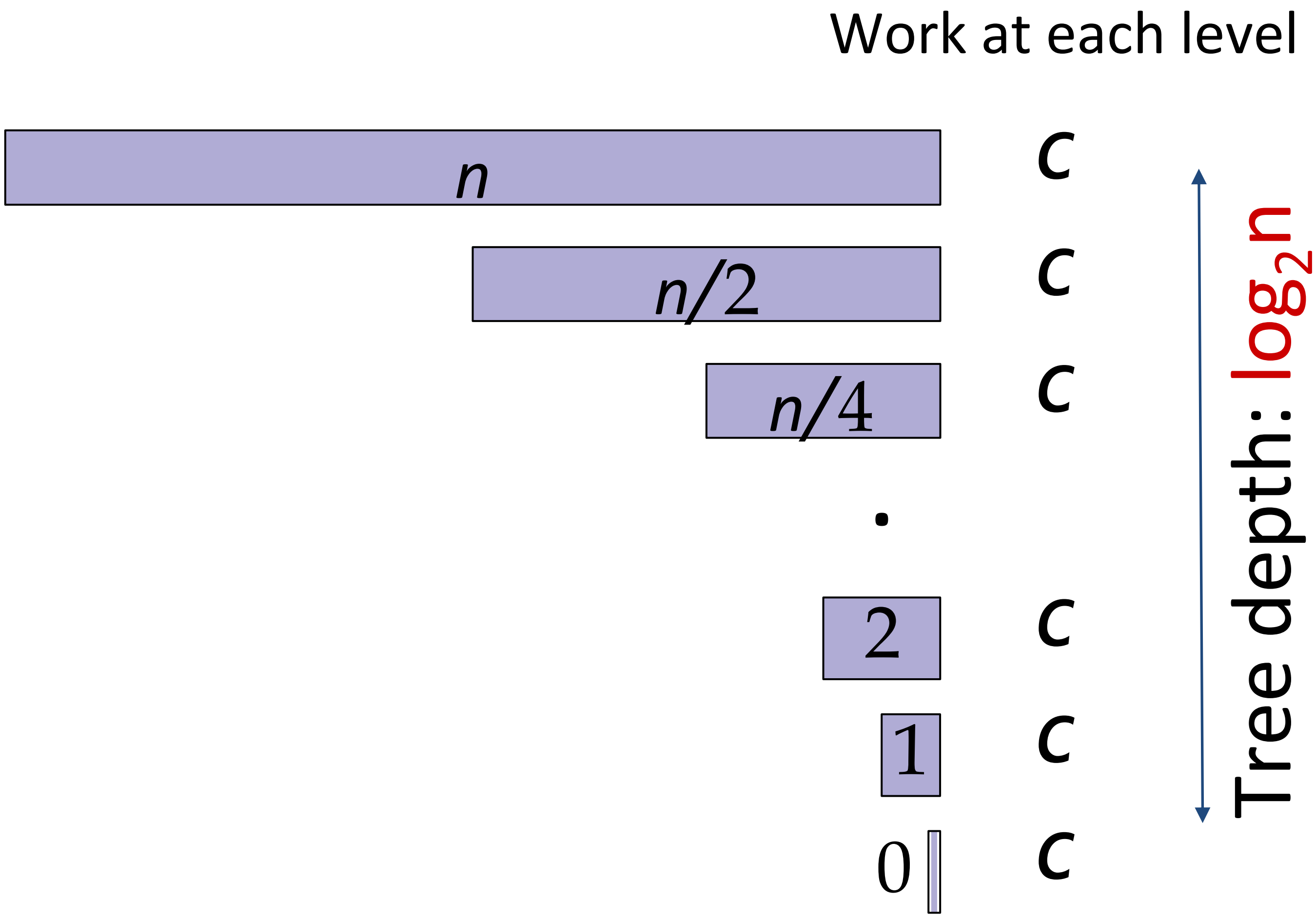
$$\text{binary\_search}(A,\ 0,\ 10,\ 50)$$

$$\text{binary\_search}(A,\ 6,\ 10,\ 50)$$

$$\text{binary\_search}(A,\ 9,\ 10,\ 50) \rightarrow 9$$

0	1	2	3	4	5	6	7	8	9	10
3	5	8	10	12	15	18	20	20	50	60

Running time of Binary Search



$$\text{Total: } \sum_{i=0}^{\log_2 n} c = O(\log n)$$

Iterative Version

binary_search_it(A, low, $high$, key)

while $\boldsymbol{low \leq high}$**:**

$mid \leftarrow low + \left\lfloor \frac{high - low}{2} \right\rfloor$

if $key = A[mid]$:

return mid

else if $key < A[mid]$:

$high = mid - 1$

else:

$low = mid + 1$

return -1

Linear search	Binary search
$O(n)$	$O(\log n)$

Calculating runtime of recursive algorithms
is not always that easy