Unit 7

Recursion, Dynamic Programming, and Abstract vs Concrete Data Types

Alberto Casagrande
Email: acasagrande@units.it

a.a. 2019/2020

The Factorial Number

Definition (The Factorial Number of n)

Is the product of all the positive natural numbers $\leq n$.

$$n! = n * (n-1) * (n-2) * \ldots * 1$$

How to compute it?

The Factorial Number

Use a variable to represent the product, initialize it to 1, and multiply its value to that of all the integer between 1 and n by using a loop.

The Factorial Number

Use a variable to represent the product, initialize it to 1, and multiply its value to that of all the integer between 1 and n by using a loop.

Easy? No!

The Factorial Number

Use a variable to represent the product, initialize it to 1, and multiply its value to that of all the integer between 1 and *n* by using a loop.

Easy? No!

Any simplier idea?

A Derivative

$$\frac{d}{dx}(\ln x + 1)^2 = ?$$

A Derivative

$$\frac{d}{dx}(\ln x + 1)^2 = \frac{2 * (\ln x + 1)}{x}$$

Why?

Eating Pizza

Eating Pizza

How do you eat pizza?

Tower of Hanoi

A game for children

- 3 rods
- n disks having different width stacked in the first rod
- only one disk can be moved at a time
- a disk can be placed only on either the floor or widther disks
- the disk tower should be moved from the first rod to the last one

Tower of Hanoi – 1 Disk

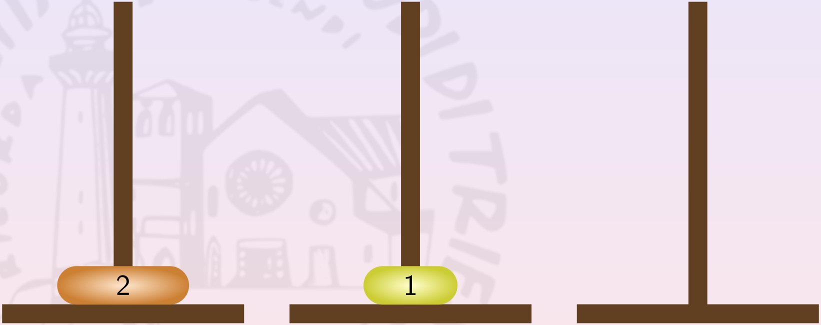
Tower of Hanoi – 1 Disk

Disk moved from rod 1 to rod 3.

Tower of Hanoi – 1 Disk

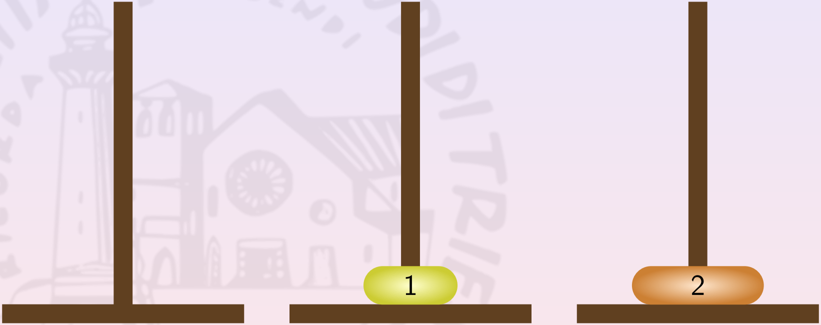
Tower of Hanoi – 2 Disks

Tower of Hanoi – 2 Disks



Disk moved from rod 1 to rod 2.

Tower of Hanoi – 2 Disks



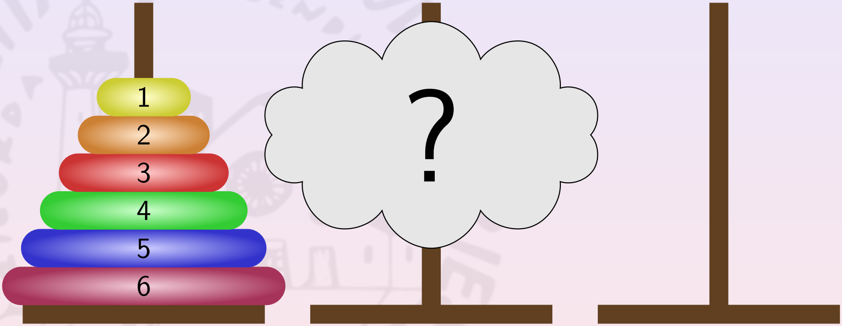
Disk moved from rod 1 to rod 3.

Tower of Hanoi – 2 Disks

Disk moved from rod 2 to rod 3.

Tower of Hanoi – 2 Disks

Tower of Hanoi – 6 Disks



What have

1. the factorial number
2. the computation of $\frac{d}{dx}(\ln x + 1)^2$
3. eating pizza
4. the tower of Hanoi

in common?

What have

1. the factorial number
2. the computation of $\frac{d}{dx}(\ln x + 1)^2$
3. eating pizza
4. the tower of Hanoi

in common?

Nothing!

What have

1. the factorial number
2. the computation of $\frac{d}{dx}(\ln x + 1)^2$
3. eating pizza
4. the tower of Hanoi

in common?

~~Nothing!~~

A solution technique!

Computing the Factorial

Whenever $n > 1$:

$$\begin{aligned} n! &= n * (n-1) * (n-2) * \ldots * 1 \\ &= n * ((n-1) * (n-2) * \ldots * 1) \\ &= n * (n-1)! \end{aligned}$$

Thus, we can define $n!$ as:

Computing the Factorial

Whenever $n > 1$:

$$\begin{aligned} n! &= n * (n-1) * (n-2) * \ldots * 1 \\ &= n * ((n-1) * (n-2) * \ldots * 1) \\ &= n * (n-1)! \end{aligned}$$

Thus, we can define $n!$ as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{othewise} \end{cases}$$

Deriving Composited Functions

$$\frac{d}{dx}(f \circ g)(x) = \left(\frac{d}{dx}(f)(g(x))\right) * \left(\frac{d}{dx}(g)(x)\right)$$

$(\ln x + 1)^2$ is the composited function $(f \circ g \circ h)(x)$ where:

- $h(x) = \ln x$
- $g(x) = x + 1$
- $f(x) = x^2$

Deriving Composited Functions

Thus:

$$\begin{aligned}\frac{d}{dx}(\ln x + 1)^2 &= \frac{d}{dx}(f \circ g \circ h)(x)\\ &= \frac{d}{dx}(f)((g \circ h)(x)) * \frac{d}{dx}(g \circ h)(x)\end{aligned}$$

Deriving Composited Functions

Thus:

$$\begin{aligned}
\frac{d}{dx}(\ln x + 1)^2 &= \frac{d}{dx}(f \circ g \circ h)(x) \\
&= \frac{d}{dx}(f)((g \circ h)(x)) * \frac{d}{dx}(g \circ h)(x) \\
&= 2 * (\ln x + 1) * \frac{d}{dx}(g \circ h)(x)
\end{aligned}$$

Deriving Composited Functions

Thus:

$$\begin{aligned}
\frac{d}{dx}(\ln x + 1)^2 &= \frac{d}{dx}(f \circ g \circ h)(x) \\
&= \frac{d}{dx}(f)((g \circ h)(x)) * \frac{d}{dx}(g \circ h)(x) \\
&= 2 * (\ln x + 1) * \frac{d}{dx}(g \circ h)(x) \\
&= 2 * (\ln x + 1) * \frac{d}{dx}(g)(h(x)) * \frac{d}{dx}(h)(x)
\end{aligned}$$

Deriving Composited Functions

Thus:

$$
\begin{aligned}
\frac{d}{dx}(\ln x + 1)^2 &= \frac{d}{dx}(f \circ g \circ h)(x) \\
&= \frac{d}{dx}(f)((g \circ h)(x)) * \frac{d}{dx}(g \circ h)(x) \\
&= 2 * (\ln x + 1) * \frac{d}{dx}(g \circ h)(x) \\
&= 2 * (\ln x + 1) * \frac{d}{dx}(g)(h(x)) * \frac{d}{dx}(h)(x) \\
&= 2 * (\ln x + 1) * 1 * \frac{1}{x}
\end{aligned}
$$

Deriving Composited Functions

Thus:

$$\begin{aligned}
\frac{d}{dx}(\ln x + 1)^2 &= \frac{d}{dx}(f \circ g \circ h)(x) \\
&= \frac{d}{dx}(f)((g \circ h)(x)) * \frac{d}{dx}(g \circ h)(x) \\
&= 2 * (\ln x + 1) * \frac{d}{dx}(g \circ h)(x) \\
&= 2 * (\ln x + 1) * \frac{d}{dx}(g)(h(x)) * \frac{d}{dx}(h)(x) \\
&= 2 * (\ln x + 1) * 1 * \frac{1}{x} \\
&= \frac{2 * (\ln x + 1)}{x}
\end{aligned}$$

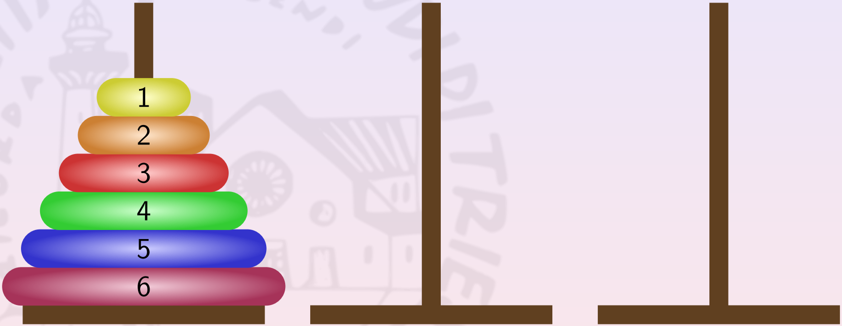
Eating Pizza

If I can eat what's left in a single bite, I do it. Otherwise, I cut a small piece of it, I eat the piece and reduce the original problem to eating a smaller quantity of pizza.

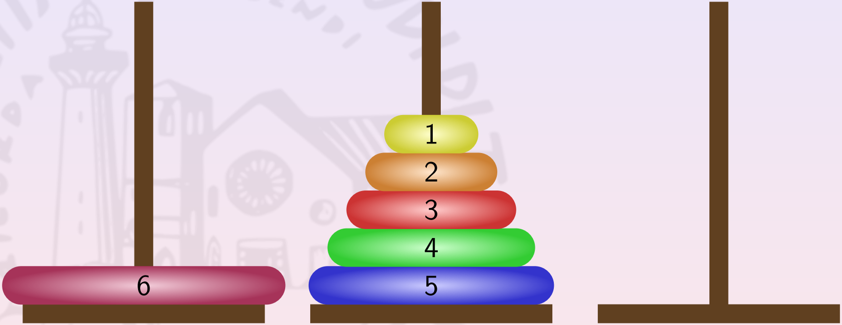
Eating Pizza

If I can eat what's left in a single bite, I do it. Otherwise, I cut a small piece of it, I eat the piece and reduce the original problem to eating a smaller quantity of pizza.

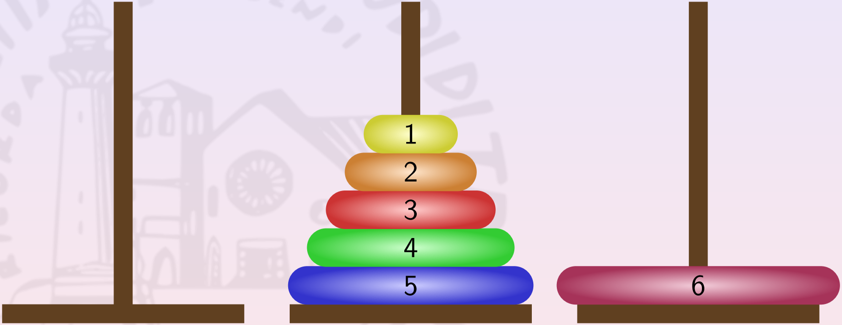
A Solution for the Tower of Hanoi with 6 Disks



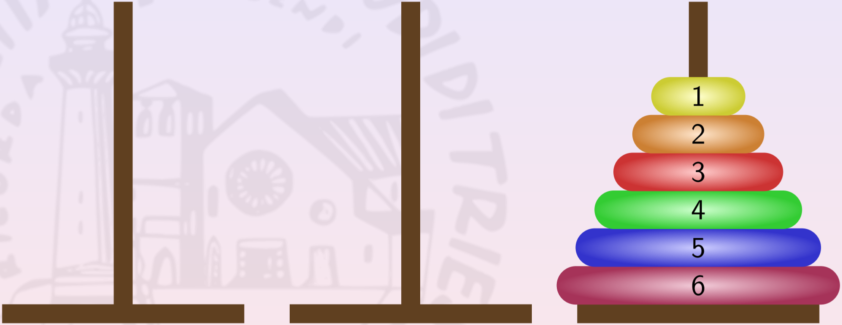
A Solution for the Tower of Hanoi with 6 Disks



A Solution for the Tower of Hanoi with 6 Disks



A Solution for the Tower of Hanoi with 6 Disks



A Solution for the Tower of Hanoi with 6 Disks

Recursion

All the previous problem solutions share the same technique:

1. identify some “easy” cases
2. solve “tough” cases by reducing them to easier instances of the same problem

Recursion

All the previous problem solutions share the same technique:

1. identify some "easy" cases
2. solve "tough" cases by reducing them to easier instances of the same problem

This technique is called recursion and is based on:

1. one or more base cases
2. one or more recursive steps

Recursively Computing the Factorial

Extremely simple and elegant

```
unsigned int fact(unsigned int n) {
  // Base case
  if (n<=1) return 1;

  // Recursive step
  return n*fact(n-1);
}
```

The Recursive Solution to Hanoi

Have you tried to write an iterative (no recursion, only loops) code to solve Hanoi tower problem?

How much difficult is it?

The Recursive Solution to Hanoi (Cont'd)

```
void Hanoi(char from_rod, char tmp_rod,
           char to_rod, unsigned int disks) {
  // Base case
  if (disks == 0) return;

  // Recursive step
  Hanoi(from_rod, to_rod, tmp_rod, disks-1);

  printf("Move disk %u from rod %d to rod %d\n",
         disks, from_rod, to_rod);

  Hanoi(tmp_rod, from_rod, to_rod, disks-1);
}
```

Computing Fibonacci Number By Recursion

Let us have a look to our iterative code

```
unsigned long int Fib(unsigned int n) {
   unsigned long int F0=1, F1=1, F2=1;

   for (unsigned int i=1; i<n; i++) {
      F2 = F1 + F0;

      F0 = F1;    /* this part is not
      F1 = F2;       really easy to
   }                 understand */

   return F2;
}
```

Computing Fibonacci Number By Recursion

The recursive version is shorter, more readable, and elegant (?!?!?!)

```
unsigned long int Fib(unsigned int n) {
   // Base cases
   if (n<2) return 1;

   //Recursive step
   return Fib(n-1) + Fib(n-2);
}
```

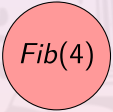
What About Their "Execution Time"?

Roughly extimable by counting instructions to be executed

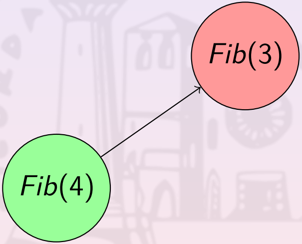
- iterative solution: 3 initializations + 3 instructions per iteration + 1 return. In total, $3 + 3 * n + 1$ instructions.
- recursive solution: 2 instructions per call and the number of calls depends on the input parameter

Follow the Function Calls. . .

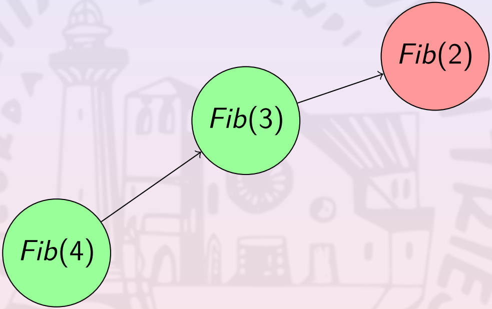
Follow the Function Calls. . .



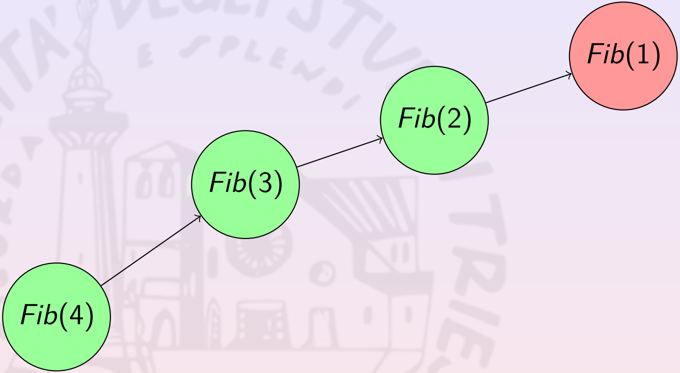
Follow the Function Calls...



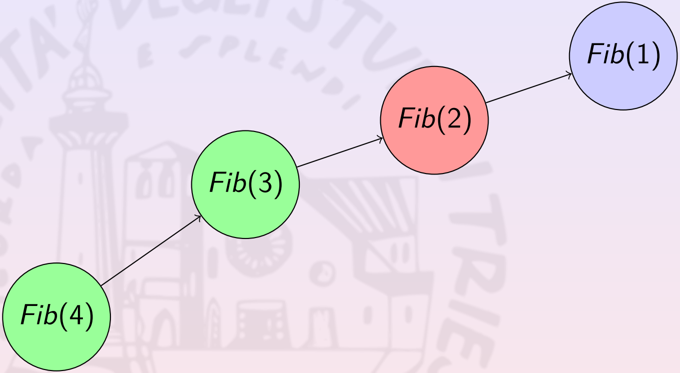
Follow the Function Calls...



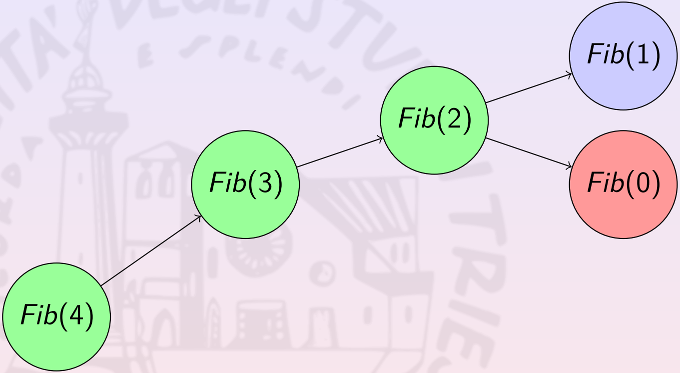
Follow the Function Calls...



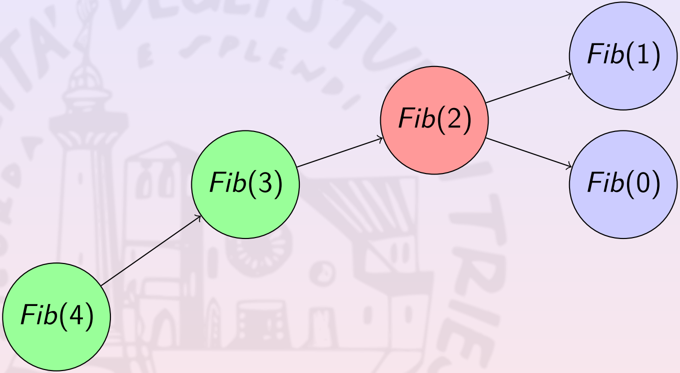
Follow the Function Calls. . .



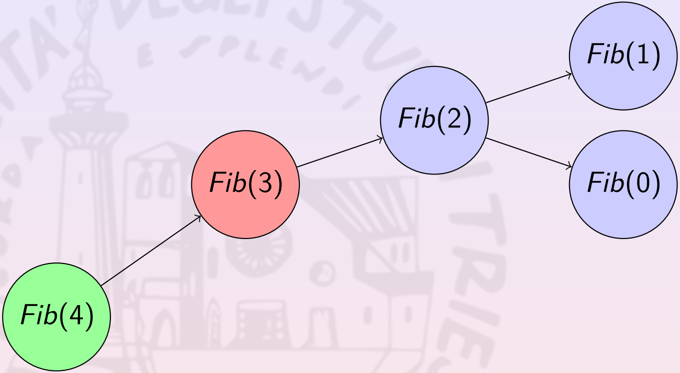
Follow the Function Calls...



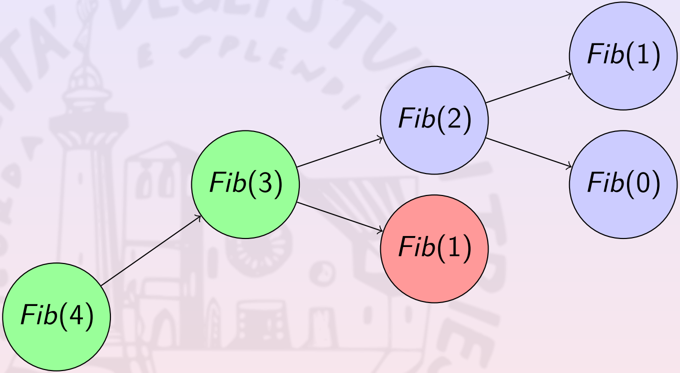
Follow the Function Calls...



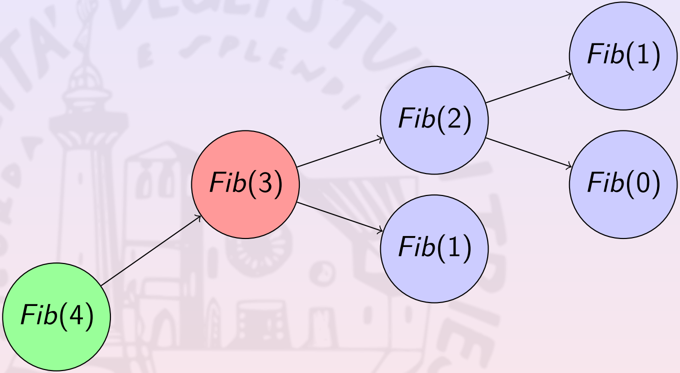
Follow the Function Calls...



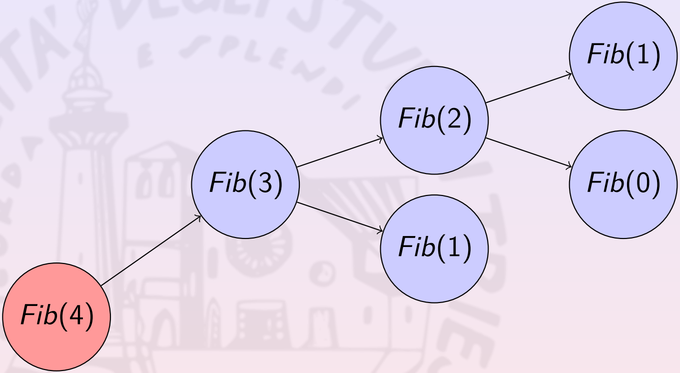
Follow the Function Calls...



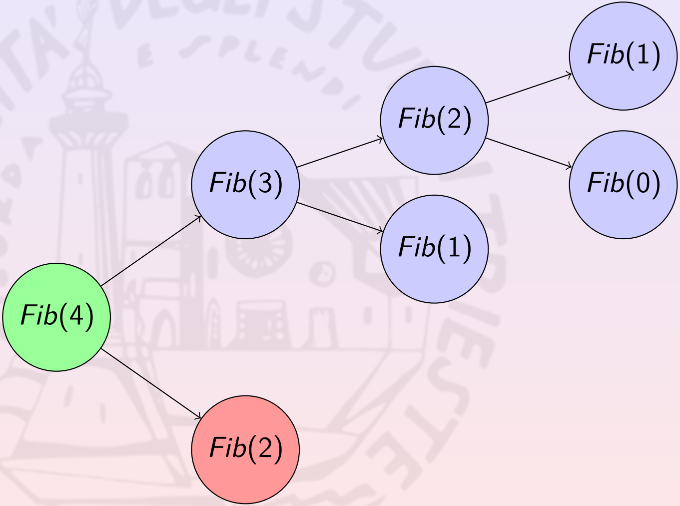
Follow the Function Calls. . .



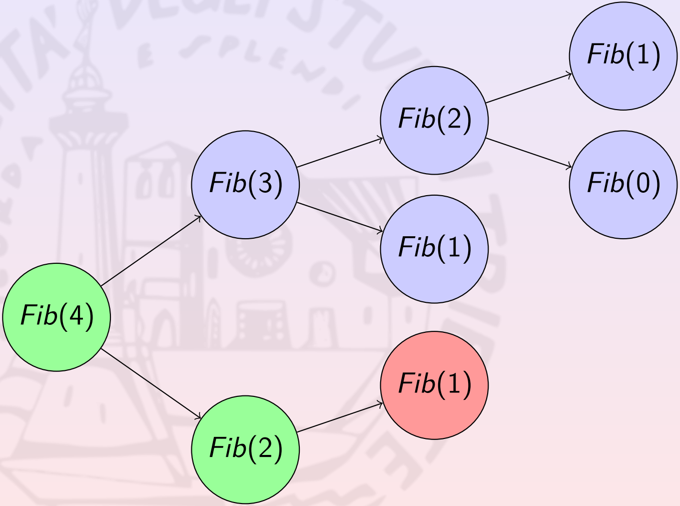
Follow the Function Calls...



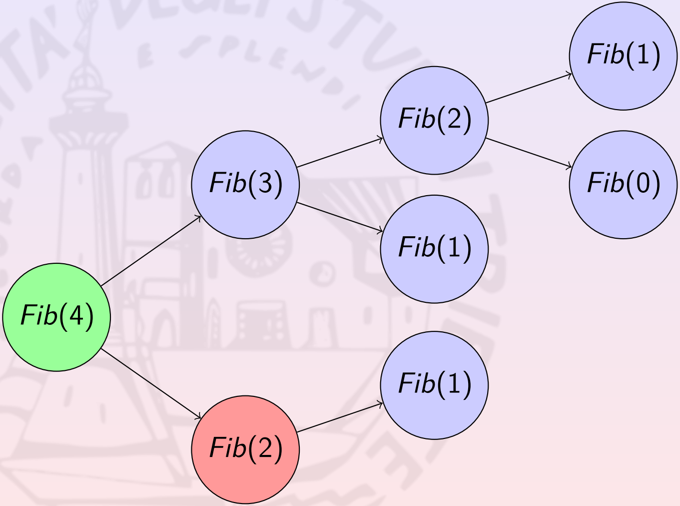
Follow the Function Calls...



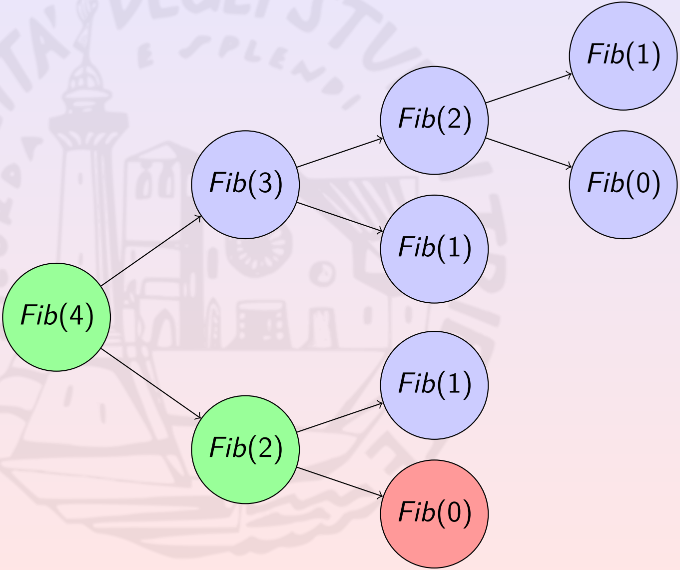
Follow the Function Calls...



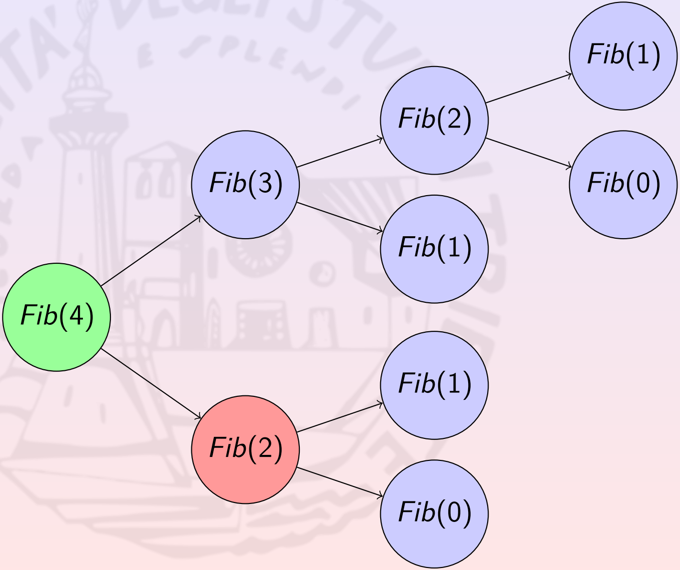
Follow the Function Calls...



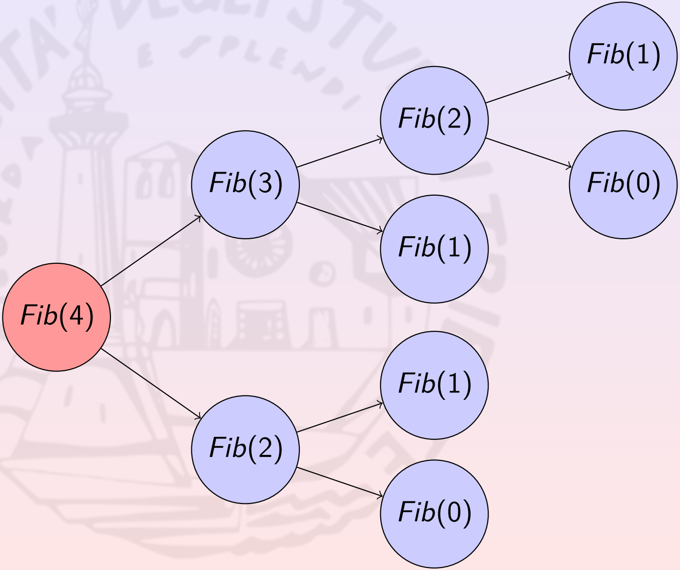
Follow the Function Calls...



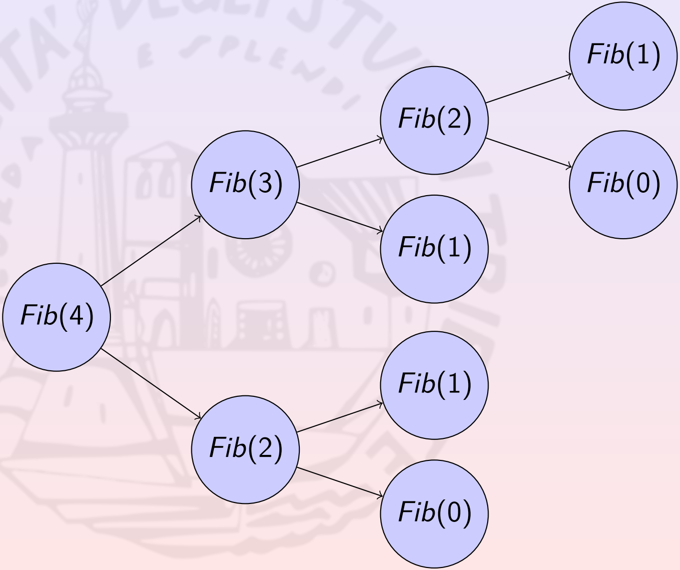
Follow the Function Calls...



Follow the Function Calls...



Follow the Function Calls...



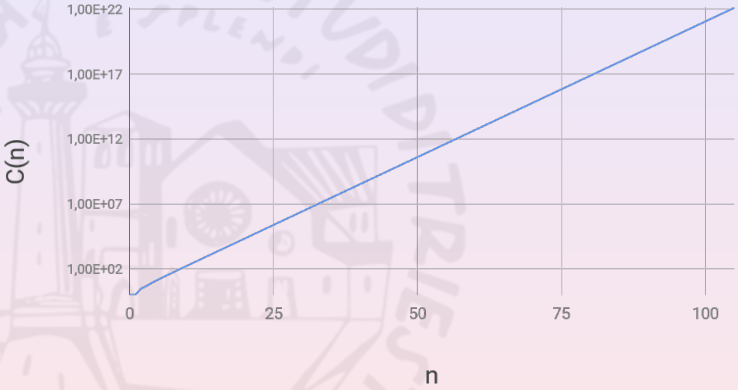
…and Count Them

A generic call to `Fib(n)` produces:

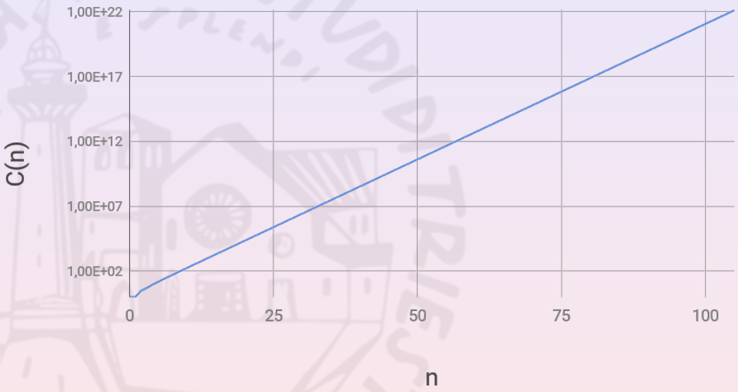
$$C(n) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } n \text{ is either } 0 \text{ or } 1 \\ C(n-1) + C(n-2) + 1 & \text{otherwise} \end{cases}$$

total calls.

. . . and Count Them



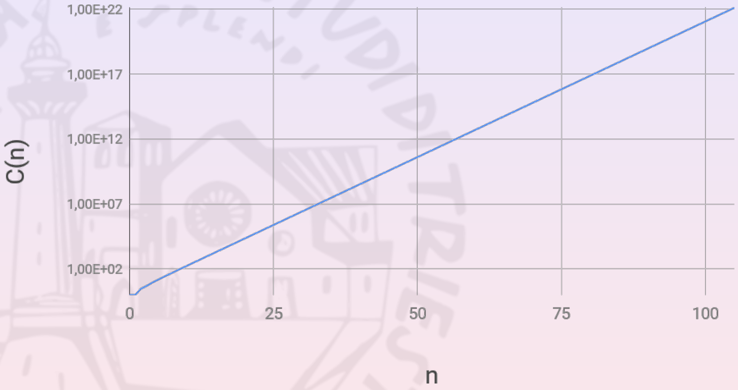
... and Count Them



It is exponential!!!

Any possible solution?

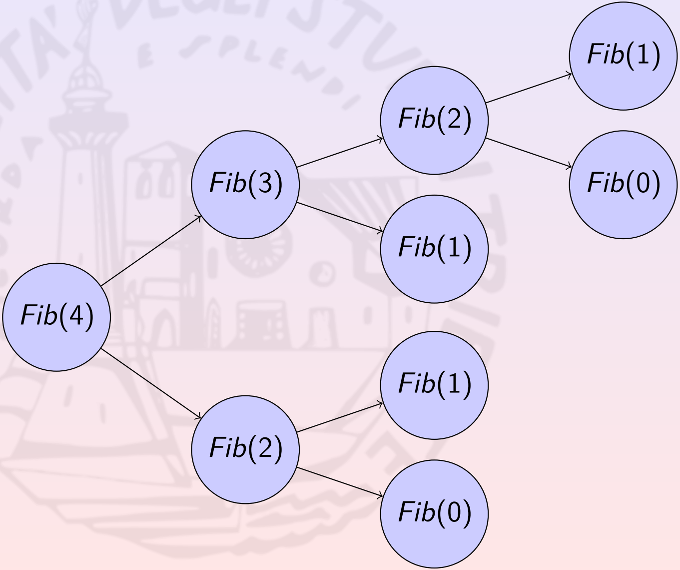
...and Count Them



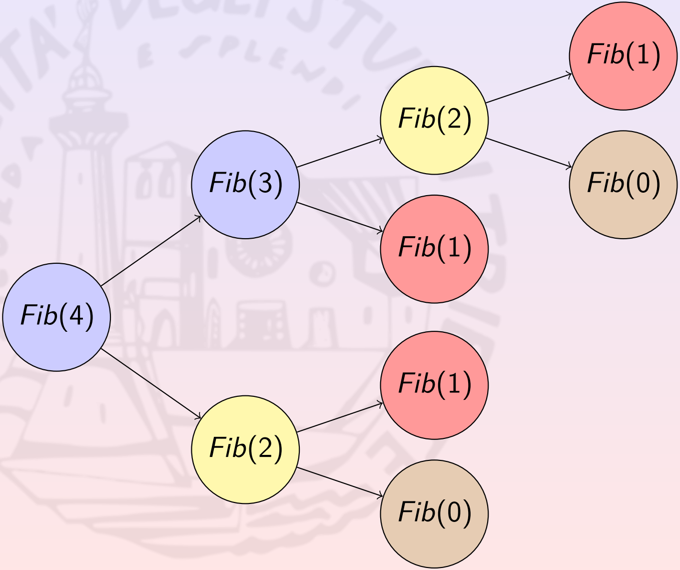
It is exponential!!!

Any possible solution? Back to the calls tree ...

Calls Tree



Calls Tree



Calls Tree

The function performs the very same calls many times.

The computation has two main features:

- evaluates sub-problems i.e., `Fib(n-1)`, `Fib(n-2)`, ...
- its sub-problems are overlapping e.g., `Fib(3)` is evaluated many times

Calls Tree

The function performs the very same calls many times.

The computation has two main features:

- evaluates sub-problems i.e., `Fib(n-1)`, `Fib(n-2)`, ...
- its sub-problems are overlapping e.g., `Fib(3)` is evaluated many times

Under such conditions we can use dynamic programming

Dynamic Programming

Is a solution technique that:

- reduces the original problems to sub-problems
- avoid overlapping by memoizing sub-problem solutions

E.g., Use an array to store the results of `Fib` calls and do not recompute them.

Fibonacci and Dynamic Programming

```
unsigned int Fib(unsigned int n) {
  unsigned int *F;

  F=(unsigned int *)calloc(n,
                 sizeof(unsigned int));

  unsigned int result = FDyn(n, F);

  free(F);

  return result;
}
```

Fibonacci and Dynamic Programming (Cont'd)

```
unsigned int FDyn(unsigned int n,
                  unsigned int *F) {
  if (F[n]!=0)
    return F[n];

  if (n<2)
    F[n] = 1;
  else
    F[n] = FDyn(n-1, F) + FDyn(n-2, F);

  return F[n];
}
```

In this case, stay with the iterative solution.

Abstract vs Concrete Data Types

In computer programming being able to distinguish between

- Abstract Data Types: data type models that specify domains and primitives
- Concrete Data Types: implementations for ADT

Abstract vs Concrete Data Types

In computer programming being able to distinguish between

- Abstract Data Types: data type models that specify domains and primitives
- Concrete Data Types: implementations for ADT

Abstract vs Concrete Data Types

In computer programming being able to distinguish between

- Abstract Data Types: data type models that specify domains and primitives
- Concrete Data Types: implementations for ADT

is fundamental.

Abstract vs Concrete Data Types

In computer programming being able to distinguish between

- Abstract Data Types: data type models that specify domains and primitives
- Concrete Data Types: implementations for ADT

is fundamental.

Replacing a CDT by another CDT is almost immediate if they implement the same ADT.

Some Abstract Data Types

Some of the most used abstract data types are:

- arrays
- lists
- queues
- stacks

ADT: Arrays

Can store a set of values and provide the following functions:

- `get(n)` gets the value from position *n*
- `set(n, v)` sets value *v* in position *n*
- `size()` returns the array size

In the C programming language it is implemented by the array.

ADT: Lists

Can store a set of values and provides the following functions:

- `get(n)` gets the element in position *n*
- `insert(n, v)` inserts the element *v* in position *n*
- `replace(n, v)` replaces the element in position *n* by *v*
- `remove(n)` removes the element in position *n* from the list
- `size()` returns the number of elements in the list

Possible Implementations for Lists

```
typedef struct int_list {
    size_t size;
    int *list;
} int_list;

int_list create_empty() {
    int_list L = {0, NULL};

    return L;
}

void destroy(int_list L) {
    free(L.list);
}
```

Possible Implementations for Lists

```
void insert(int_list L, const size_t n,
            const int v) {
  L.list=realloc(L.list,
             (++L.size)*sizeof(int));

  for (int i=L.size-1; i>n; i++)
    L.list[i] = L.list[i-1];

  L.list[n] = v;
}
```

Possible Implementations for Lists (Cont'd)

```
typedef struct list_el {
  struct list_el *next;
  int value;
} list_el;

typedef struct list_el * int_list2;

int_list2 create_empty() {
   return NULL;
}
```

Possible Implementations for Lists (Cont'd)

```
void insert(int_list2 L, const size_t n,
            const int v) {
  if (n==0) {
    list_el next=(list_el *)
          malloc(sizeof(list_el));

    next.value=L->value;
    next.next=L->next;
    L->next = next;
    L->value = v;
  } else if (L->next == NULL)
    {...}
  else
    insert(L->next, n-1, v);
}
```

ADT: Queues

store a set of values and use the **F**irst **I**n **F**irst **O**ut policy

- `enqueue(v)` inserts *v* at the end of the queue
- `dequeue()` removes the first element of the queue and returns it
- `head()` returns the first element of the queue without removing it
- `size()` returns the number of elements in the queue

Possible Implementations for Queues

Circular arrays

```
typedef struct {
  size_t max_size; size_t size;
  size_t front; int *queue;
} int_queue;

int_queue create_empty() {
   int_queue Q={DEFAULT_MAX_SIZE, 0,
                        0, NULL};

   Q.queue=(int *)calloc(DEFAULT_MAX_SIZE,
                          sizeof(int));

   return Q;
}
```

Possible Implementations for Queues

```
void enqueue(int_queue Q, const int v) {
  if (Q.size == Q.max_size) {
     Q.max_size *= 2;
     Q.queue=(int *)realloc(Q.queue,
                          Q.max_size*sizeof(int));
      ....
  }

  size_t idx=((Q.size++)+Q.front)%Q.max_size;
  Q.queue[idx] = v;
}
```

ADT: Stacks

store a set of values and use the First In Last Out policy

- `push(v)` inserts v on the top of the stack
- `pop()` removes the element at the top of the stack and returns it
- `top()` returns the element at the top of the stack without removing it
- `is_empty()` returns `true` if and only if the stack is empty

Possible Implementations for Stacks

```
typedef struct {
  size_t max_size; size_t size;
  int *stack;
} int_stack;

int_stack create_empty() {
   int_stack S={DEFAULT_MAX_SIZE, 0,
                0, NULL};

   S.stack=(int *)calloc(DEFAULT_MAX_SIZE,
                          sizeof(int));

   return S;
}
```

Possible Implementations for Stacks

```
void push(int_stack S, const int v) {
  if (S.size == S.max_size) {
    S.max_size *= 2;
    S.queue=(int *)realloc(S.queue,
                S.max_size*sizeof(int));
  }

  S.queue[S.size++] = v;
}
```

Coming soon. . .

- Exercises!