Luiss
Libera Università Internazionale degli Studi Sociali Guido Carli

Algorithms A.Y. 2022/2023

Lab – Fibonacci and Ternary Search complexity

Irene Finocchi, Flavio Giorgi, Bardh Prenkaj
finocchi@luiss.it, fgiorgi@luiss.it, bprenkaj@luiss.it©

24 February 2023

courtesy of: *Andrea Coletta*

LUISS

Dipartimento di Impresa e Management

Fibonacci – A first recursive approach

algorithm `fibonacci2`$(integer\ n) \rightarrow integer$
1. **if** $(n \leq 2)$ **then return** 1
2. **else return** `fibonacci2`$(n-1)$ + `fibonacci2`$(n-2)$

Figure 1.4 Algorithm `fibonacci2` to compute the n-th Fibonacci number.

Question: How many recursive call the algorithm does approximately?

Fibonacci – A first recursive approach

algorithm `fibonacci2`($integer\ n$) $\rightarrow$ $integer$
1. **if** ($n \le 2$) **then return** 1
2. **else return** `fibonacci2`($n-1$) + `fibonacci2`($n-2$)

Figure 1.4 Algorithm `fibonacci2` to compute the n-th Fibonacci number.

Question: How many recursive call the algorithm does approximately?
Answer: $O(2^n)$

Fibonacci – A first recursive approach

algorithm `fibonacci2`$(integer\ n) \rightarrow integer$
1. **if** $(n \leq 2)$ **then return** 1
2. **else return** `fibonacci2`$(n-1)$ + `fibonacci2`$(n-2)$

Figure 1.4 Algorithm `fibonacci2` to compute the n-th Fibonacci number.

Question: How many recursive call the algorithm does approximately?

Answer: $O(2^n)$

Question: Can we prove it?

Fibonacci – A first recursive approach

algorithm `fibonacci2`$(integer\ n) \to integer$
1. **if** $(n \leq 2)$ **then return** 1
2. **else return** `fibonacci2`$(n-1)$ + `fibonacci2`$(n-2)$

Figure 1.4 Algorithm `fibonacci2` to compute the n-th Fibonacci number.

Question: How many recursive call the algorithm does approximately?

Answer: $O(2^n)$

Question: Can we prove it?

Answer: ***YES!***

Fibonacci – A first recursive approach

Theorem**:** the computational complexity for $Fibonacci2$ is $O(2^n)$

Big O Notation: A Brief Recap

What is the Big O notation?

Big $\boldsymbol{O}$ notation is a mathematical notation that describes the behavior of a function when the argument tends to infinity

Big O Notation: A Brief Recap

What is the Big O notation?

Big $\boldsymbol{O}$ notation is a mathematical notation that describes the behavior of a function when the argument tends to infinity

Why do we need that?

We use it to classify algorithms according to how their run time or space requirements

Big O Notation: A Brief Recap

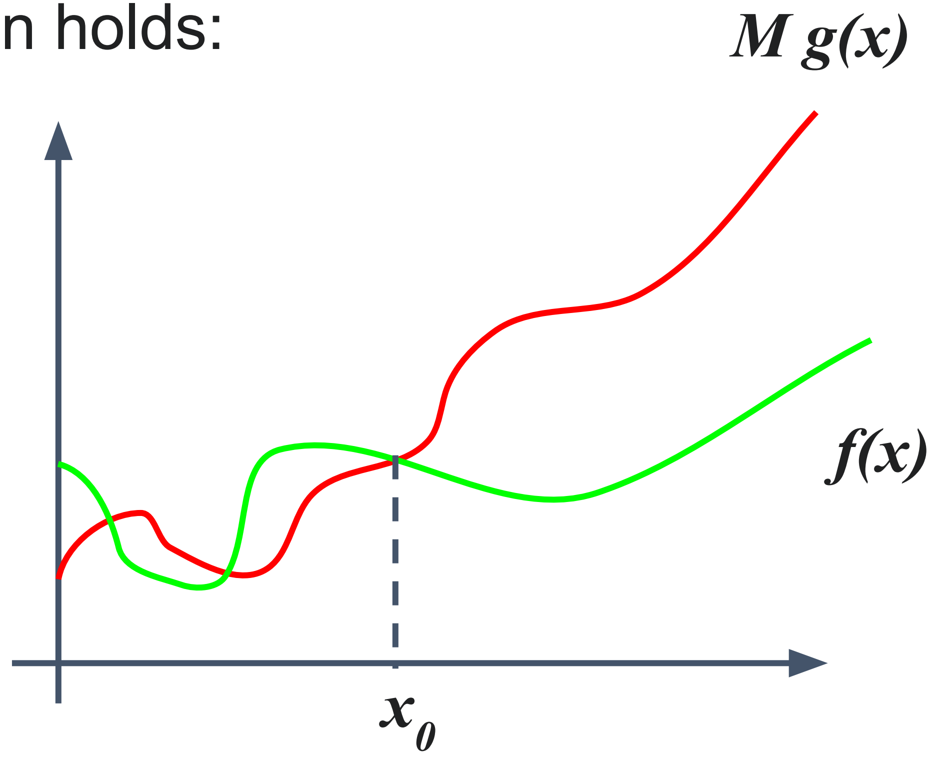
Formal Definition:
let f and g be two functions.
We can say that $f(x) = O(g(x))$ when $x \to \infty$ if given two real numbers M and x_0 the following relation holds:

$|\boldsymbol{f(x)}| \leq \boldsymbol{M\ g(x)}$ for all $\boldsymbol{x > x_0}$

Big O Notation: A Brief Recap

Formal Definition:
let f and g be two functions.
We can say that $f(x) = O(g(x))$ when $x \to \infty$ if given two real numbers M and x_0 the following relation holds:

$|f(x)| \leq M\, g(x)$ for all $x > x_0$



Big O Notation: Rules

To analyze algorithms we want to explore cases with **very large input values**.

In this setting, the contribution of the terms that grow **"most quickly"** will eventually make the other ones irrelevant.

So we can apply the following rules:

- If $f(x)$ is a sum of several terms **we can keep just the one with largest growth rate**
- If $f(x)$ is a product of several factors, **any constants that do not depend on** x **(the input) can be omitted**

Big O Notation: Example

Given $f(x) = 6x^4 + 2x^3 + 5$

Applying the rules we get: $f(x) = O(x^4)$.

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

<u>**Before we start we are going to use a bit of syntactic sugar:**</u>

- we define $\boldsymbol{T(n) = Fibonacci2(n)}$ as the number of operations needed to compute the $n\text{-}th$ Fibonacci number

- we define $\boldsymbol{c}$ as a constant value for each operation that can be executed in a constant amount of time (for example sum two numbers)

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

Let's start

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$

Proof:

First of all we can say that the time needed to compute *Fibonacci2(n)* is equal to:

$$Fibonacci2(n) = Fibonacci2(n\text{-}1) + Fibonacci2(n\text{-}2) + c$$

Thus using our notation, just to be concise, it will become:

$$T(n) = T(n-1) + T(n-2) + c$$

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$

Now we can assume that the time needed to compute $T(n\text{-}1)$ is approximately equal to the time to compute $T(n\text{-}2)$.
Mathematically we can write this approximation as $T(n\text{-}1) \geq T(n\text{-}2)$

Is it ok to do that? Yes, but we know that the result won't be exactly the right one

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
2) $T(n\text{-}1) \geq T(n\text{-}2)$

Because of it we can rewrite $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$ (Equation 1) **substituting $T(n\text{-}2)$ with $T(n\text{-}1)$**.

But now **the equivalence does not hold anymore** so we have to change = with ≤ getting as result $T(n) \leq T(n\text{-}1) + T(n\text{-}1) + c$.

That we can rewrite as $\boldsymbol{T(n) \leq 2T(n\text{-}1) + c}$

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
2) $T(n\text{-}1) \geq T(n\text{-}2)$

Because of it we can rewrite $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$ (Equation 1) as $T(n) \leq 2T(n\text{-}1) + c$

Why?

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
2) $T(n\text{-}1) \geq T(n\text{-}2)$

Because of it we can rewrite $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$ (Equation 1) as $T(n) \leq 2T(n\text{-}1) + c$

Why?
Intuitively $\boldsymbol{T(n\text{-}1) > T(n\text{-}2)}$.
To understand this you can make an example.
It requires more time to compute $T(5)$ than $T(4)$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$

Proof:

1) $T(n) = T(n-1) + T(n-2) + c$
2) $T(n-1) \geq T(n-2)$

Because of it we can rewrite $T(n) = T(n-1) + T(n-2) + c$ (Equation 1) as $T(n) \leq 2T(n-1) + c$

Why?

If you follow the line of reasoning what we are saying is that:

$\boldsymbol{T(6) = T(5) + T(4) + c \leq T(5) + T(5) + c}$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$

Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
2) $T(n\text{-}1) \geq T(n\text{-}2)$

Because of it we can rewrite $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$ (Equation 1) as $T(n) \leq 2T(n\text{-}1) + c$

Why do we need this approximation?

Simple: to make things easier!

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
2) $T(n\text{-}1) \geq T(n\text{-}2)$
3) $T(n) \leq T(n\text{-}1) + T(n\text{-}1) + c$ or $T(n) \leq 2T(n\text{-}1) + c$

Now, we need to get the time needed to compute $T(n\text{-}1)$
How can we do that? Easy! we can say that:
$\boldsymbol{T(n\text{-}1) = T(n\text{-}2) + T(n\text{-}3) + c}$
But using the same line of reasoning used before
(but now with $T(n\text{-}2) \geq T(n\text{-}3)$) we get: $T(n\text{-}1) \leq T(n\text{-}2) + T(n\text{-}2) + c$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

1) $T(n) = T(n-1) + T(n-2) + c$
2) $T(n-1) \geq T(n-2)$
3) $T(n) \leq T(n-1) + T(n-1) + c$ or $T(n) \leq 2T(n-1) + c$

So $\boldsymbol{T(n-1) \leq T(n-2) + T(n-2) + c}$ can be written as $\boldsymbol{T(n-1) \leq 2T(n-2) + c}$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$

Proof:

1) $T(n) = T(n-1) + T(n-2) + c$
2) $T(n-1) \geq T(n-2)$
3) $T(n) \leq T(n-1) + T(n-1) + c$ or $T(n) \leq 2T(n-1) + c$
4) $T(n-1) \leq 2T(n-2) + c$

You can easily see that we can substitute in equation 3 the equation 4. Even if we make the substitution the **inequality will hold in any case**. So we can write equation 3 as $T(n) \leq 2 * (2T(n-2) + c) + c$

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$

Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
2) $T(n\text{-}1) \geq T(n\text{-}2)$
3) $T(n) \leq T(n\text{-}1) + T(n\text{-}1) + c$ or $T(n) \leq 2T(n\text{-}1) + c$
4) $T(n\text{-}1) \leq 2T(n\text{-}2) + c$

By simply performing the multiplication we get:

$T(n) \leq 2 * (2T(n\text{-}2) + c) + c = 4T(n\text{-}2) + 3c$

Thus: $\boldsymbol{T(n) \leq 4T(n\text{-}2) + 3c}$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

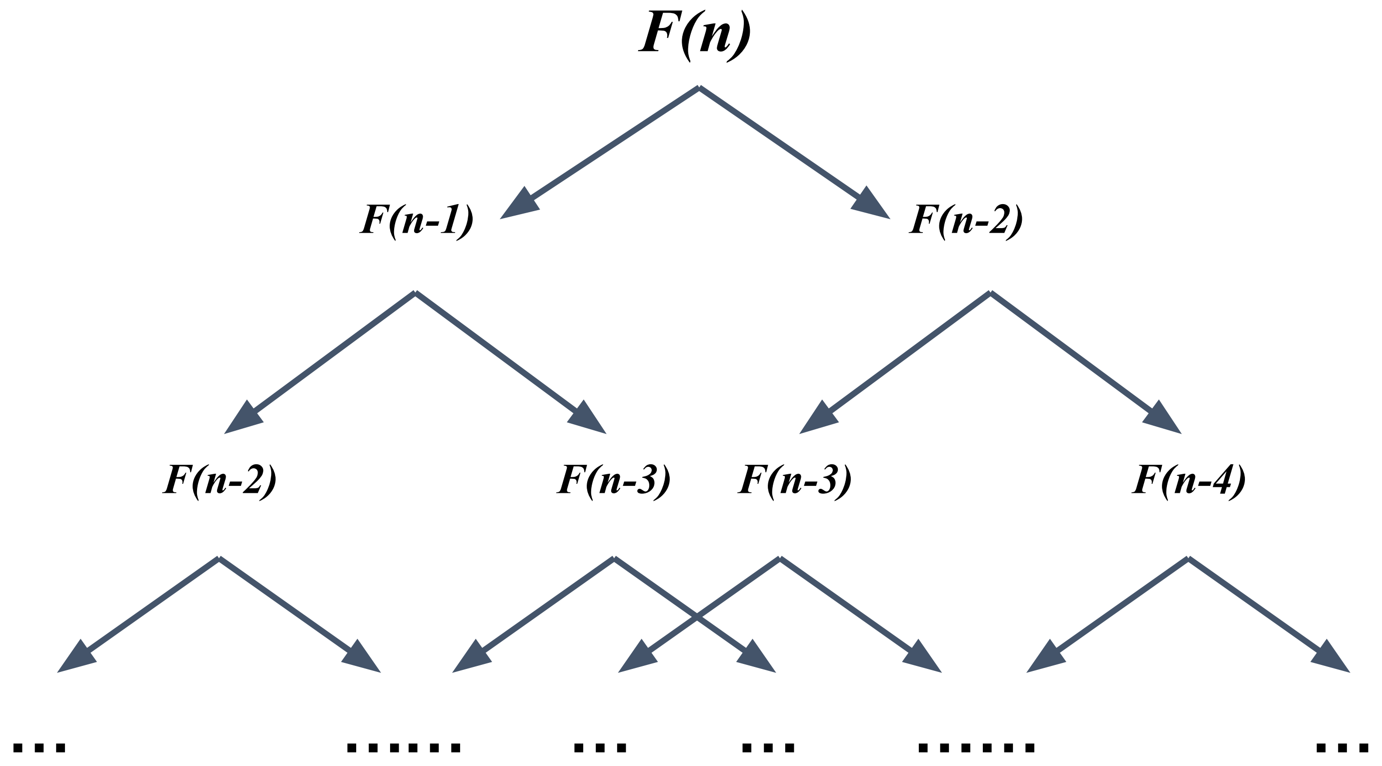
1) $\boldsymbol{T(n) = T(n-1) + T(n-2) + c}$
2) $T(n-1) \geq T(n-2)$
3) $\boldsymbol{T(n) \leq T(n-1) + T(n-1) + c}$ ***or*** $\boldsymbol{T(n) \leq 2T(n-1) + c}$
4) $T(n-1) \leq 2T(n-2) + c$
5) $\boldsymbol{T(n) \leq 4T(n-2) + 3c}$

As you can see the idea is to define $\boldsymbol{T(n)}$ **as the time to compute the sub problems!**

Fibonacci – A first recursive approach

Graphically it means:

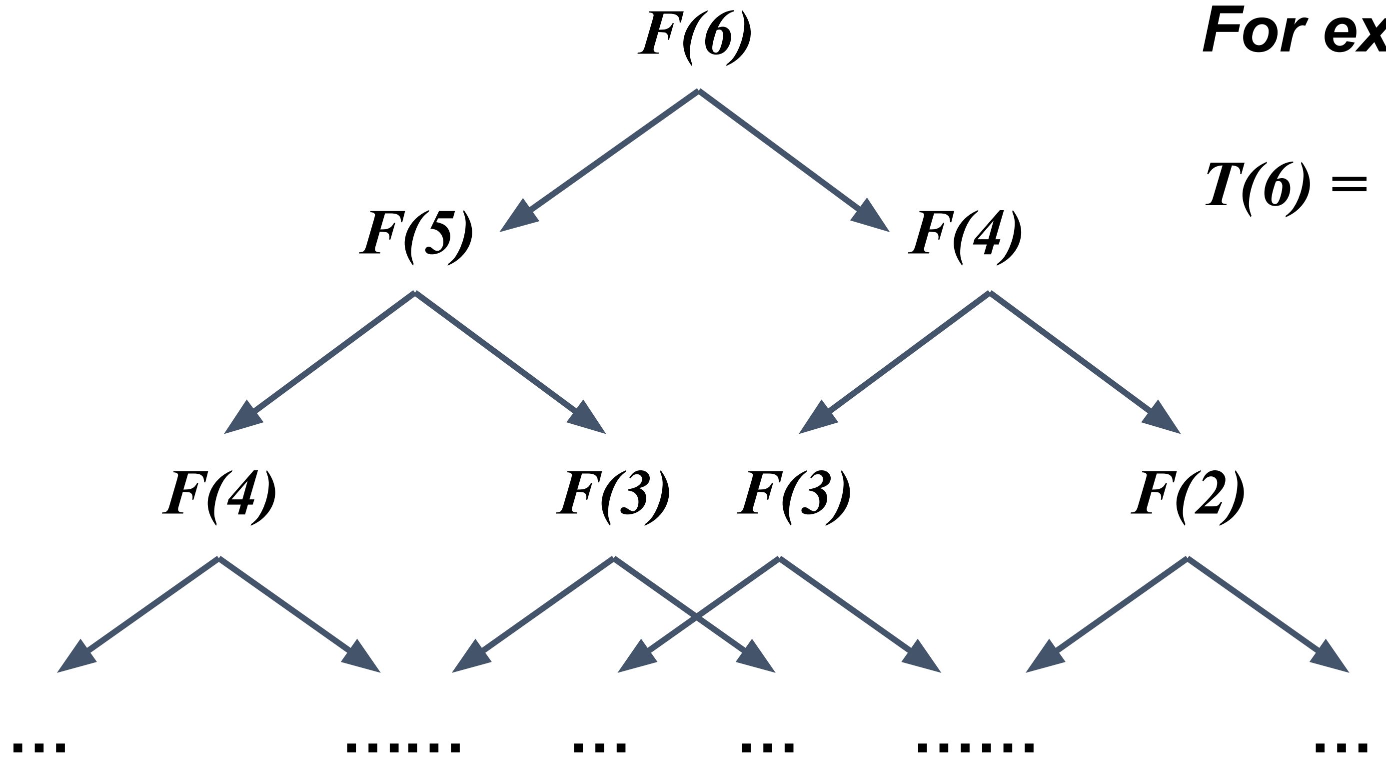
$$T(n) = T(n\text{-}1) + T(n\text{-}2) + c$$



LUISS

Fibonacci – A first recursive approach

Graphically it means:



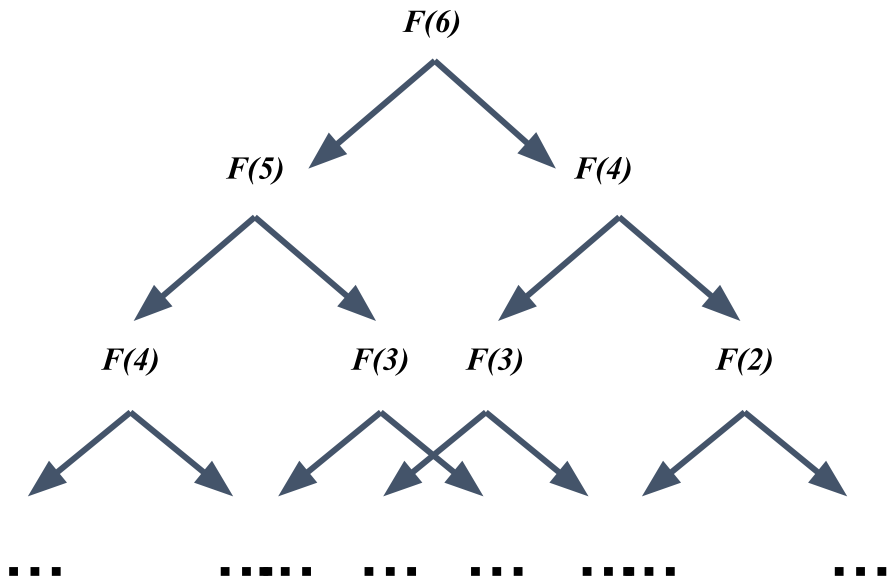
$$T(n) = T(n-1) + T(n-2) + c$$

For example if n=6

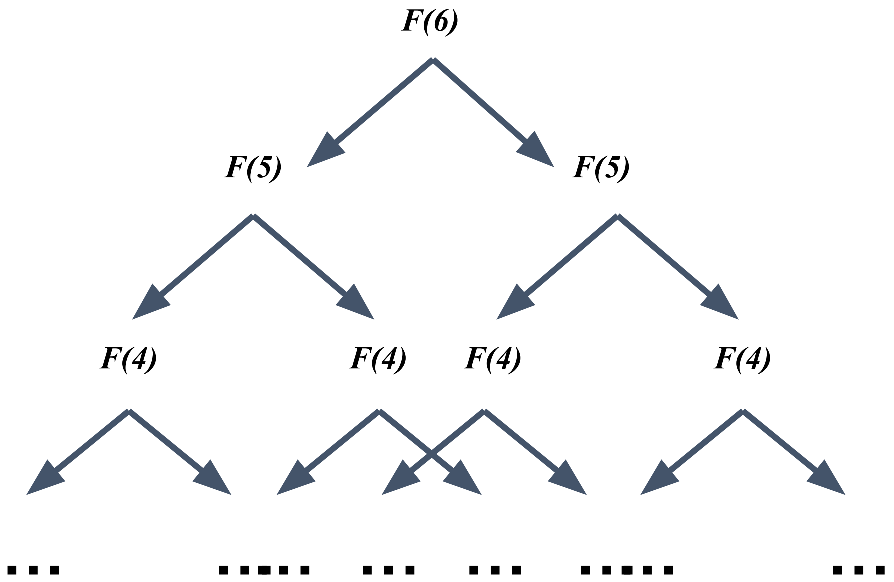
$$T(6) = T(5) + T(4) + c$$

Fibonacci – A first recursive approach

Graphically it means: $T(n) = T(n\text{-}1) + T(n\text{-}2) + c \leq T(n\text{-}1) + T(n\text{-}1) + c$

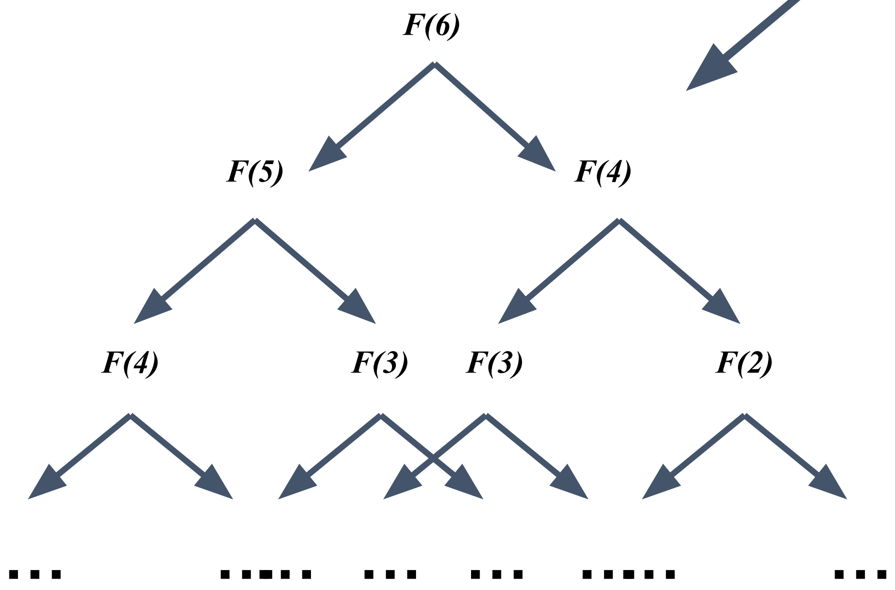


$\leq$

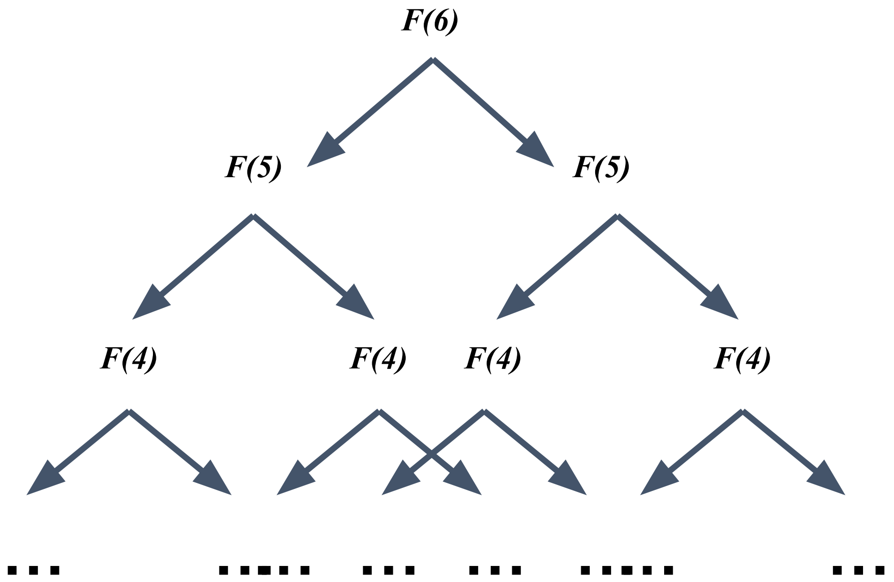


Fibonacci – A first recursive approach

Graphically it means: $\boxed{T(n) = T(n\text{-}1) + T(n\text{-}2) + c} \leq T(n\text{-}1) + T(n\text{-}1) + c$



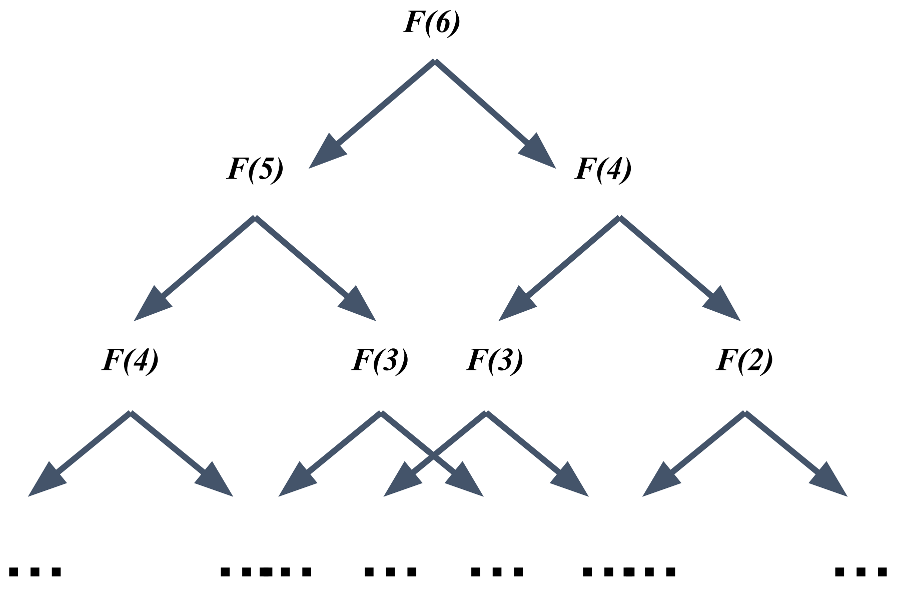
$\leq$



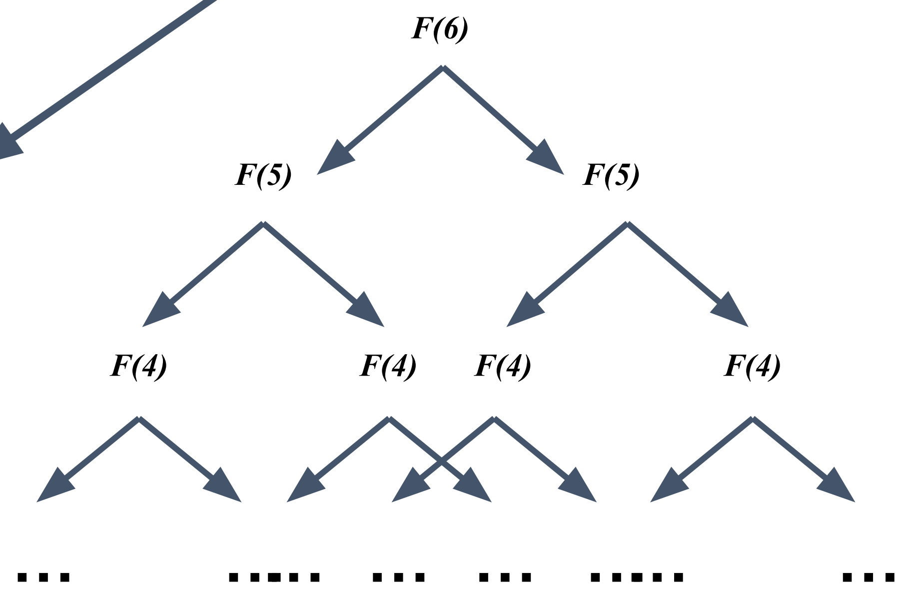
Fibonacci – A first recursive approach

Graphically it means:

$$T(n) = T(n-1) + T(n-2) + c \boxed{\le} T(n-1) + T(n-1) + c$$

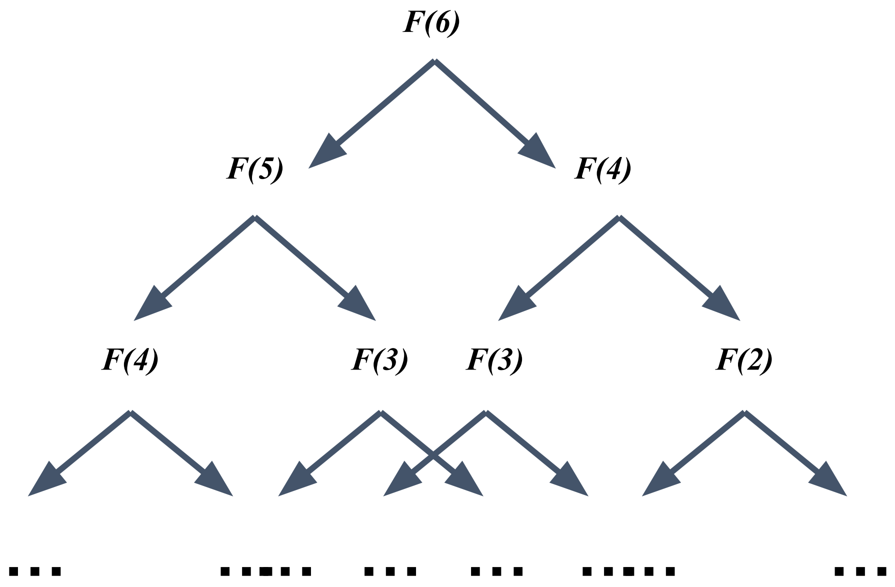


$\le$

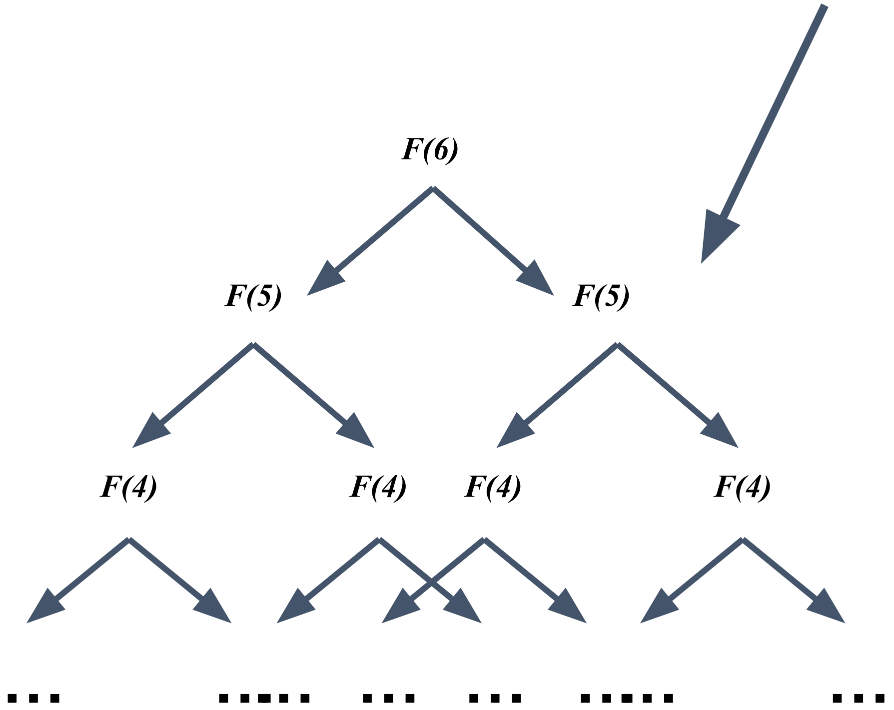


Fibonacci – A first recursive approach

Graphically it means: $T(n) = T(n\text{-}1) + T(n\text{-}2) + c \leq \boxed{T(n\text{-}1) + T(n\text{-}1) + c}$



$\leq$



Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

1) $\boldsymbol{T(n) = T(n-1) + T(n-2) + c}$
2) $T(n-1) \geq T(n-2)$
3) $\boldsymbol{T(n) \leq T(n-1) + T(n-1) + c \;\; or \; T(n) \leq 2T(n-1) + c}$
4) $T(n-1) \leq 2T(n-2) + c$
5) $\boldsymbol{T(n) \leq 4T(n-2) + 3c}$

We can still follow the same line of reasoning and decompose $T(n-2)$ **into sub problems**

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

1) $\boldsymbol{T(n) = T(n\text{-}1) + T(n\text{-}2) + c}$
2) $T(n\text{-}1) \geq T(n\text{-}2)$
3) $\boldsymbol{T(n) \leq T(n\text{-}1) + T(n\text{-}1) + c}$ ***or*** $\boldsymbol{T(n) \leq 2T(n\text{-}1) + c}$
4) $T(n\text{-}1) \leq 2T(n\text{-}2) + c$
5) $\boldsymbol{T(n) \leq 4T(n\text{-}2) + 3c}$

$T(n\text{-}2) = T(n\text{-}3) + T(n\text{-}4) + c \leq T(n\text{-}3) + T(n\text{-}3) + c = 2T(n\text{-}3) + c$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

1) $\boldsymbol{T(n) = T(n-1) + T(n-2) + c}$
2) $T(n-1) \geq T(n-2)$
3) $\boldsymbol{T(n) \leq T(n-1) + T(n-1) + c \;\text{ or }\; T(n) \leq 2T(n-1) + c}$
4) $T(n-1) \leq 2T(n-2) + c$
5) $\boldsymbol{T(n) \leq 4T(n-2) + 3c}$

$T(n-2) = T(n-3) + T(n-4) + c \leq T(n-3) + T(n-3) + c = 2T(n-3) + c$
If we substitute this definition of $T(n-2)$ in equation 5 we get:

$\boldsymbol{T(n) \leq 4(2T(n-3) + c) + 3c = 8T(n-3) + 7c}$

LUISS

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

1) $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
2) $T(n\text{-}1) \geq T(n\text{-}2)$
3) $T(n) \leq T(n\text{-}1) + T(n\text{-}1) + c$ or $T(n) \leq 2T(n\text{-}1) + c$
4) $T(n\text{-}1) \leq 2T(n\text{-}2) + c$
5) $T(n) \leq 4T(n\text{-}2) + 3c$
6) $T(n) \leq 8T(n\text{-}3) + 7c$
7) …

As you can see it seem there is a patter…

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$

Proof:

1) $T(n) \leq 2T(n\text{-}1) + c$
2) $T(n) \leq 4T(n\text{-}2) + 3c$
3) $T(n) \leq 8T(n\text{-}3) + 7c$

…

We can generalize it with:

$$T(n) \leq 2^k\, T(n\text{-}k) + (2^k\text{-}1)c$$

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

$$T(n) \leq 2^k\, T(n-k) + (2^k-1)c$$

Now, which is the value of k such that $n-k = 0$?
Remember: the k here is the value representing the tree depth!

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

$$T(n) \leq 2^k\, T(n-k) + (2^k-1)c$$

Using the following equality: $n-k=0$

Follows that $k = n$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$
Proof:

$$T(n) \leq 2^k\, T(n-k) + (2^k-1)c$$

We can substitute k with n and put $n-k = 0$ and we get

$$T(n) \leq 2^n\; T(0) + (2^n - 1)\, c$$

We can say that $T(0)$ executes in constant time so, we get: ...

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$

Proof:

$$T(n) \leq 2^n\, T(0) + (2^n - 1)\, c$$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$

Proof:

$$T(n) \leq 2^n \cancel{T(1)} + (2^n - 1)$$

$O(1)$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$

Proof:

$$T(n) \le 2^n \cancel{T(0)} + (\cancel{2^n} - 1)$$

Fibonacci – A first recursive approach

Theorem: the computational complexity for *Fibonacci2* is $O(2^n)$

Proof:

$$T(n) \le 2^n \cancel{T(0)} + \cancel{(2^n - 1)}$$

$$T(n) = O(2^n)$$

LUISS

Fibonacci – A first recursive approach

Theorem: the computational complexity for $Fibonacci2$ is $O(2^n)$
Proof:

$$T(n) = O(2^n)$$

There are other approximations that are more precise.

Fun fact:
It is possible to prove that the computational complexity of this algorithm is φ^n

Ternary Search

```
def ternarySearch(l, r, key, ar):
    if (r >= l):
        mid1 = l + (r - l) //3
        mid2 = r - (r - l) //3
        if (ar[mid1] == key):
            return mid1
        if (ar[mid2] == key):
            return mid2
        if (key < ar[mid1]):
            return ternarySearch(l, mid1 - 1, key, ar)
        elif (key > ar[mid2]):
            return ternarySearch(mid2 + 1, r, key, ar)
        else:
            return ternarySearch(mid1 + 1, mid2 - 1, key, ar)
    return -1
```

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$

Proof:

Again we define the relation that describes how many operation are needed to compute the search.

We define $\boldsymbol{T(n) = Ternary(n)}$ as the number of operations needed to search a number in a list of length n

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$
Proof:

We can say that $\boldsymbol{T(n) = T(\frac{n}{3}) + c}$

Where n is the length of the list and c is a constant that represents the comparison operations.

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$
Proof:

1) $T(n) = T(\frac{n}{3}) + c$

Now we have to define $T(\frac{n}{3})$

Ideas?

LUISS

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$

Proof:

1) $T(n) = T(\frac{n}{3}) + c$

We can do that by dividing again n by 3 so:

$$T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$$

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$
Proof:

$$1)\quad T(n) = T(\frac{n}{3}) + c$$

$$1)\quad T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$$

substituting equation 2 in equation 1 we get $T(n) = T(\frac{n}{3^2}) + 2\,c$

LUISS

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$
Proof:

1) $T(n) = T(\frac{n}{3}) + c$
1) $T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$
1) $T(n) = T(\frac{n}{3^2}) + 2\,c$

again we have to find the definition of $T(\frac{n}{3^2})$

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$

Proof:

1) $T(n) = T(\frac{n}{3}) + c$
1) $T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$
1) $T(n) = T(\frac{n}{3^2}) + 2\,c$

again we have to find the definition of $T(\frac{n}{3^2}) = T(\frac{n}{3^3}) + c$

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$

Proof:

1) $T(n) = T(\frac{n}{3}) + c$
1) $T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$
1) $T(n) = T(\frac{n}{3^2}) + 2\,c$

We can use the definition to define $T(n)$

LUISS

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$
Proof:

1) $T(n) = T(\frac{n}{3}) + c$
1) $T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$
1) $T(n) = T(\frac{n}{3^2}) + 2\,c$

We can use the definition to define $T(n) = T(\frac{n}{3^3}) + 3\,c$

LUISS

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$

Proof:

1) $T(n) = T(\frac{n}{3}) + c$
1) $T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$
1) $T(n) = T(\frac{n}{3^2}) + 2\ c$
1) $T(n) = T(\frac{n}{3^3}) + 3\ c$

You can start recognizing a pattern…

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(\log_3 n)$
Proof:

1) $T(n) = T(\frac{n}{3}) + c$
1) $T(\frac{n}{3}) = T(\frac{n}{3^2}) + c$
1) $T(n) = T(\frac{n}{3^2}) + 2\,c$
1) $T(n) = T(\frac{n}{3^3}) + 3\,c$

You can start recognizing a pattern…

$$T(n) = T(\frac{n}{3^k}) + k\,c$$

LUISS

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$

Proof:

Now we want to know for which k we have $T(1)$ (the base case)

$$T(n) = T(\frac{n}{3^k}) + k\,c$$

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$

Proof:

Now we want to know for which k we have $T(1)$ (the base case)

So we put $\frac{n}{3^k} = 1$

Thus $n = 3^k$

And so applying log_3 to both sides we get $k = log_3 n$

LUISS

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$
Proof:

Now we want to know for which k we have $T(1)$ (the base case)

We use $k = log_3 n$ and put it into the recurrence relation for $n=1$

$$T(n) = T(1) + \ log_3 n \ \ c$$

LUISS

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$

Proof:

Now we want to know for which k we have $T(1)$ (the base case)

We use $k = log_3 n$ and put it into the recurrence relation for $n=1$

$$T(n) = T(1) + \ log_3 n \ \ c$$

Applying the asymptotic notation we get…

Ternary Search

Theorem: the computational complexity for *Ternary* is $\Theta(log_3 n)$
Proof:

Now we want to know for which k we have $T(1)$ (the base case)

We use $k = log_3 n$ and put it into the recurrence relation for $n=1$

$$T(n) = T(1) + \ log_3 n \ \ c$$

$$\boldsymbol{T(n) = \Theta(log_3 n\ )}$$