Data Structures and Algorithms

Luciano Bononi
Computer Science Engineering
University of Bologna

bononi@cs.unibo.it
http://www.cs.unibo.it/~bononi/

Slide credits: these slides have been translated from slides created by Moreno Marzolla

Course information

- Luciano Bononi
  `bononi@cs.unibo.it`
  `http://www.cs.unibo.it/~bononi/`
- Lessons
  - Monday 9.00-13.00
  - Friday 9.00-13.00
  - Some variations scheduled (see detailed calendar)
- To talk with me
  - Always drop me an email before to define a date/hour.
  - My office: Mura Anteo Zamboni 7, office T08

General information

Course website

- http://www.cs.unibo.it/~bononi/
  - > Courses > Data Structures and Algorithms A.A. 2011/2012
- Will find:
  - **General information**
  - Lesson slides
  - exercises
  - Links and recommended readings
  - Exam preparation material
- Also check RSS and news on the website:

http://www.unibo.it/SitoWebDocente/default.htm?upn=luciano.bononi%40unibo.it

http://www.unibo.it/SitoWebDocente/default.htm?upn=luciano.bononi%40unibo.it&TabControl1=TabAvvisi

Today I will collect your names for a mailing list

Recommended readings

Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft, Data Structures and Algorithms, Addison Wesley, 1983.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, McGraw-Hill, 2001.

Donald E. Knuth, The Art of Computer Programming, Volumes 1-3, Addison-Wesley Professional, 1998.

S.B. Kishor, Data Structures, Edition 3, Das Ganu Prakashan, Nagpur, 2008.

Further information, books, and material will be provided as a Web reference.

Exam

- Written exam
- Oral exam
-
- Dates will be agreed by using the mailing list.

Algorithms and Data Structures

What is an algorithm?

- A algorithm is a procedure to resolve a problem by means of a finite sequence of basic atomic steps.
- The procedure must be defined in a not ambiguous and accurate way to be executed automatically
- The name comes from a Persian mathematician Abu Ja'far Muhammad ibn Musa Khwarizmi
  - Author of the first reference algebraic text
  - A Moon crater is dedicated to him

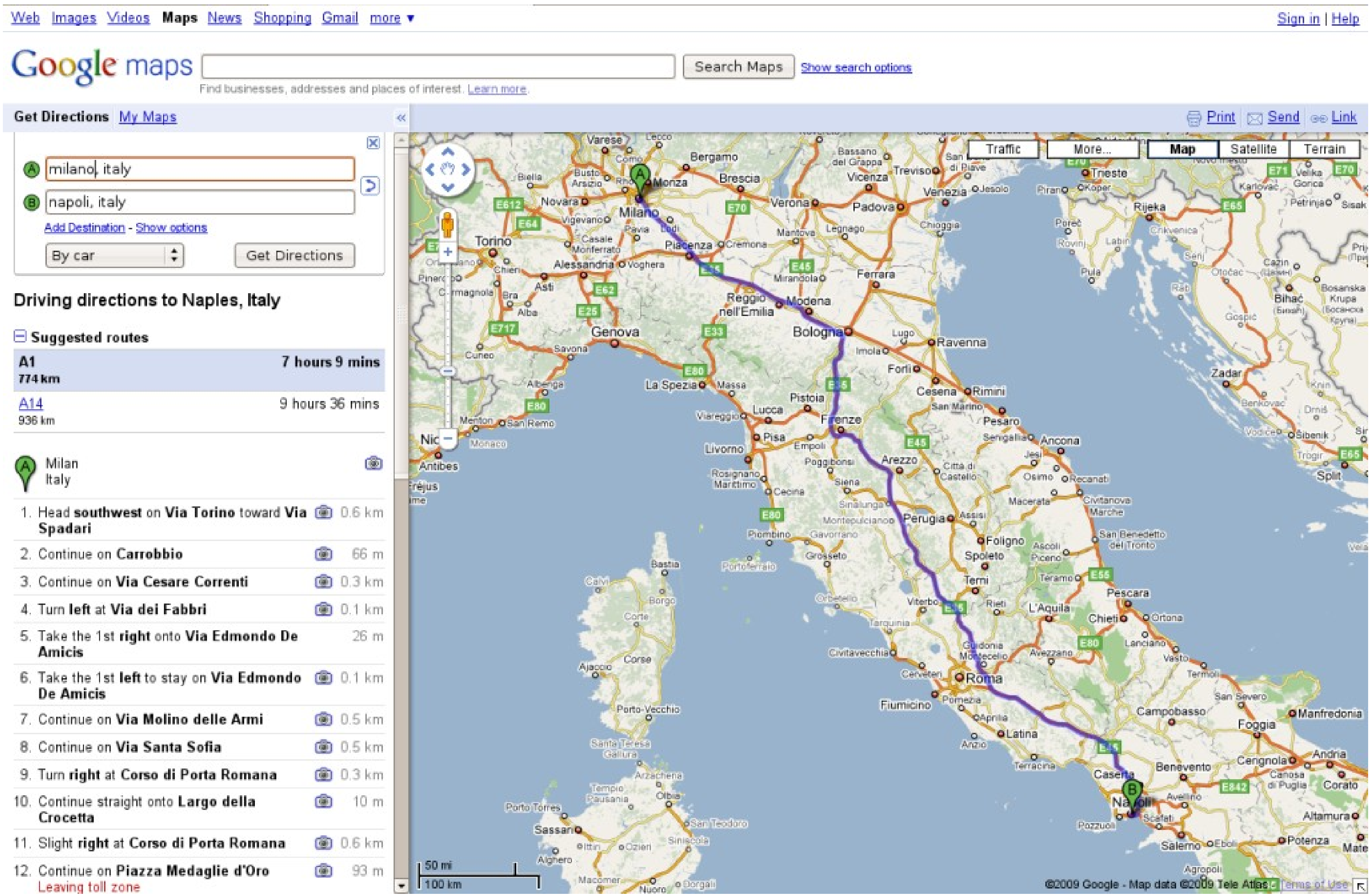
Algorithm vs Program

- A algorithm describes (at high level) a computation procedure which when executed produces a result.
- A program is the implementation of a algorithm by means of a programming language
  - A program can be executed on a computer (creating a process under execution); an algorithm cannot be executed as is in a natural form.

Algorithms are everywhere!

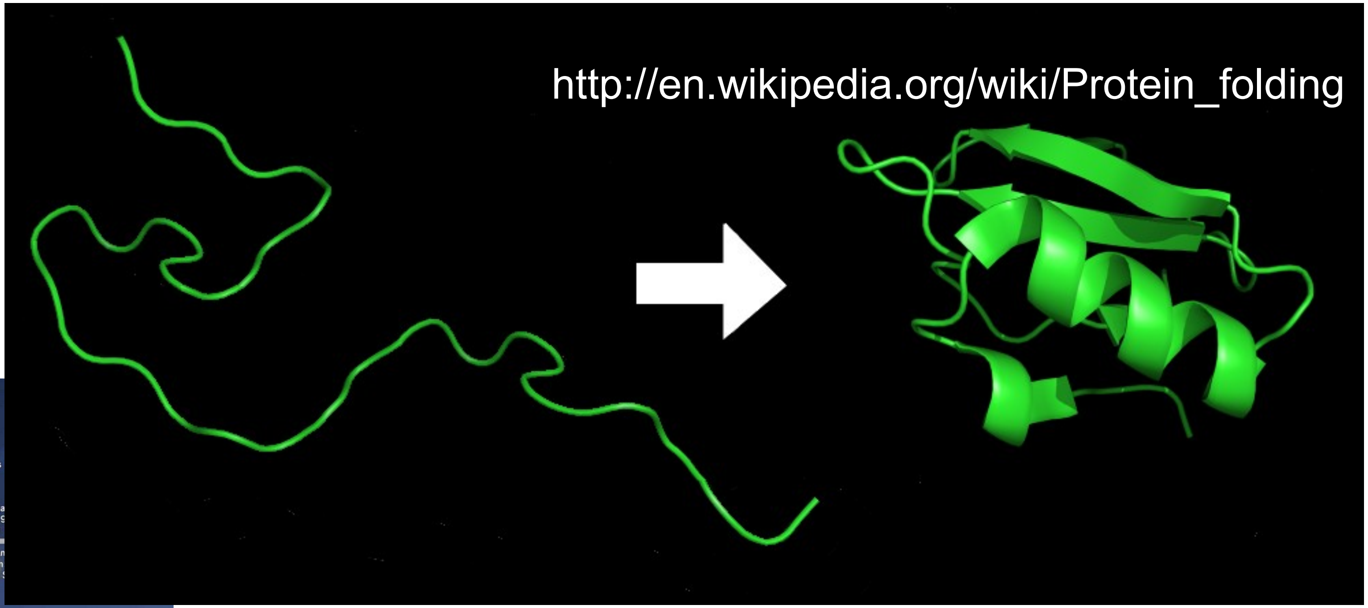
- Internet. Web search, packet routing, distributed file sharing.
- Biology. Human genome project, protein folding.
- Computers. Circuit layout, file system, compilers.
- Computer graphics. Movies, video games, virtual reality.
- Security. Cell phones, e-commerce, voting machines.
- Multimedia. CD player, DVD, MP3, JPG, DivX, HDTV.
- Transportation. Airline crew scheduling, map routing.
- Physics. N-body simulation, particle collision simulation.
- ...

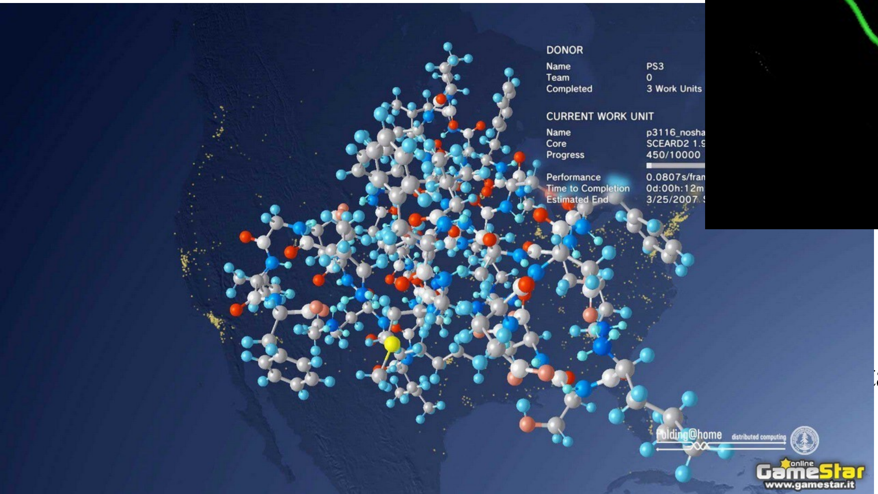
Why we're studying algorithms?



Why WE're studying algorithms?

- e.g. A protein 3D structure is determined by interactions of aminoacids.
- Some health issues generated by wrong folding, to be studied.
- Folding@Home





Algorithms again?

- Hide rendering surfaces, gaming, physical simulation, etc.

Why to care about algorithms?

- Algorithms provide advantages
  - An efficient algorithm is often the difference between being or being not able to solve a problem with the given resources
- Many algorithms we will see were invented by students!
- Algorithms are fun. :-)... yes they are. No, seriously.

Where do we start from?

- There are some classical algs to resolve common problems
  - Ordering, searching, visit of graphs...
- How could we evaluate the efficiency of an algorithm?
- How to derive or invent new algorithms that better exploit the resources tradeoffs (and the opportune data structures)?

Warmup: Fibonacci numbers

- The Fibonacci sequence $F_1, F_2, \ldots F_n, \ldots$ is defined as:

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}, n > 2$$

Leonardo Fibonacci
(Pisa, 1170—Pisa, 1250)
http://it.wikipedia.org/wiki/Leonardo_Fibonacci

Closed form

- Good news: a close form exists for F_n

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

  where

- $$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618 \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

- Bad news: to evaluate this formula errors are introduced due to need to compute floating point aritmetics

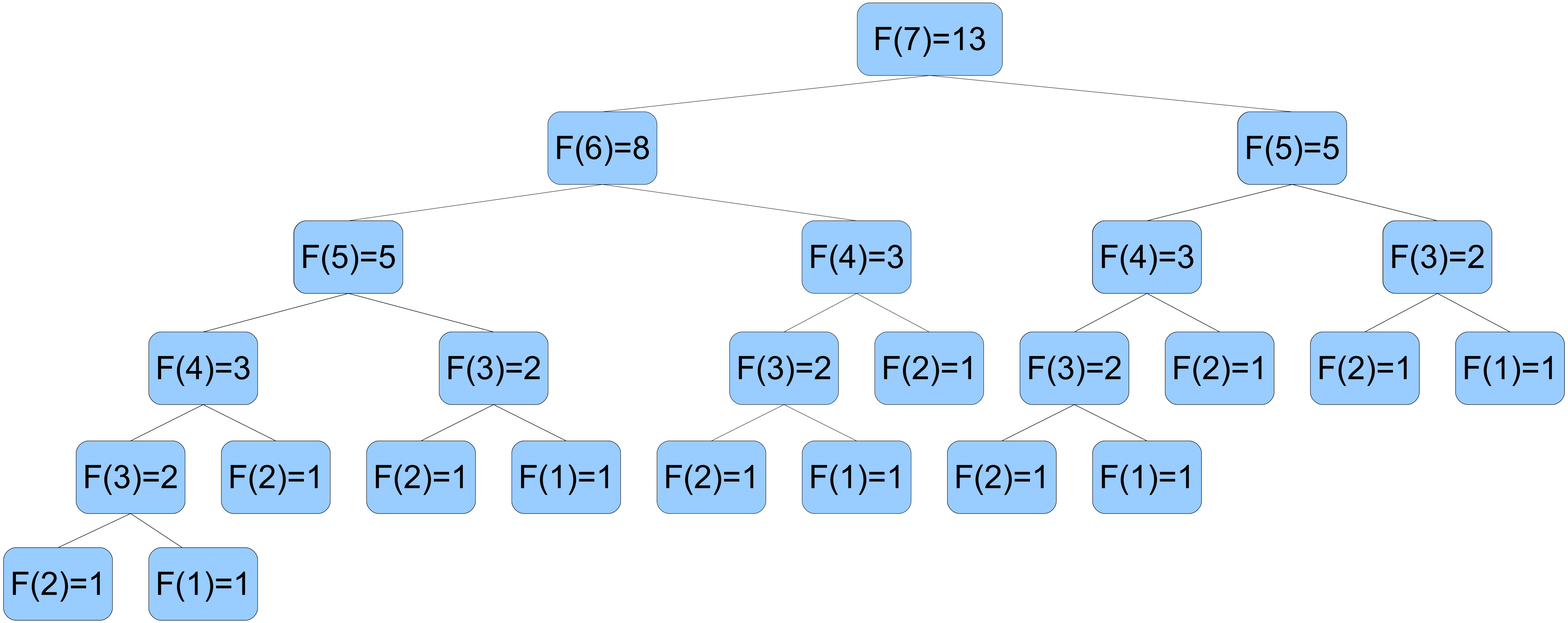
The trivial Fibonacci algorithm

- Let's define an algorithm to compute Fn based on a trivial recursive function:

```
algorithm Fibonacci2(int n) → int
    if ( n==1 || n==2 ) then
        return 1;
    else
        return Fibonacci2(n-1)+Fibonacci2(n-2);
    endif
```

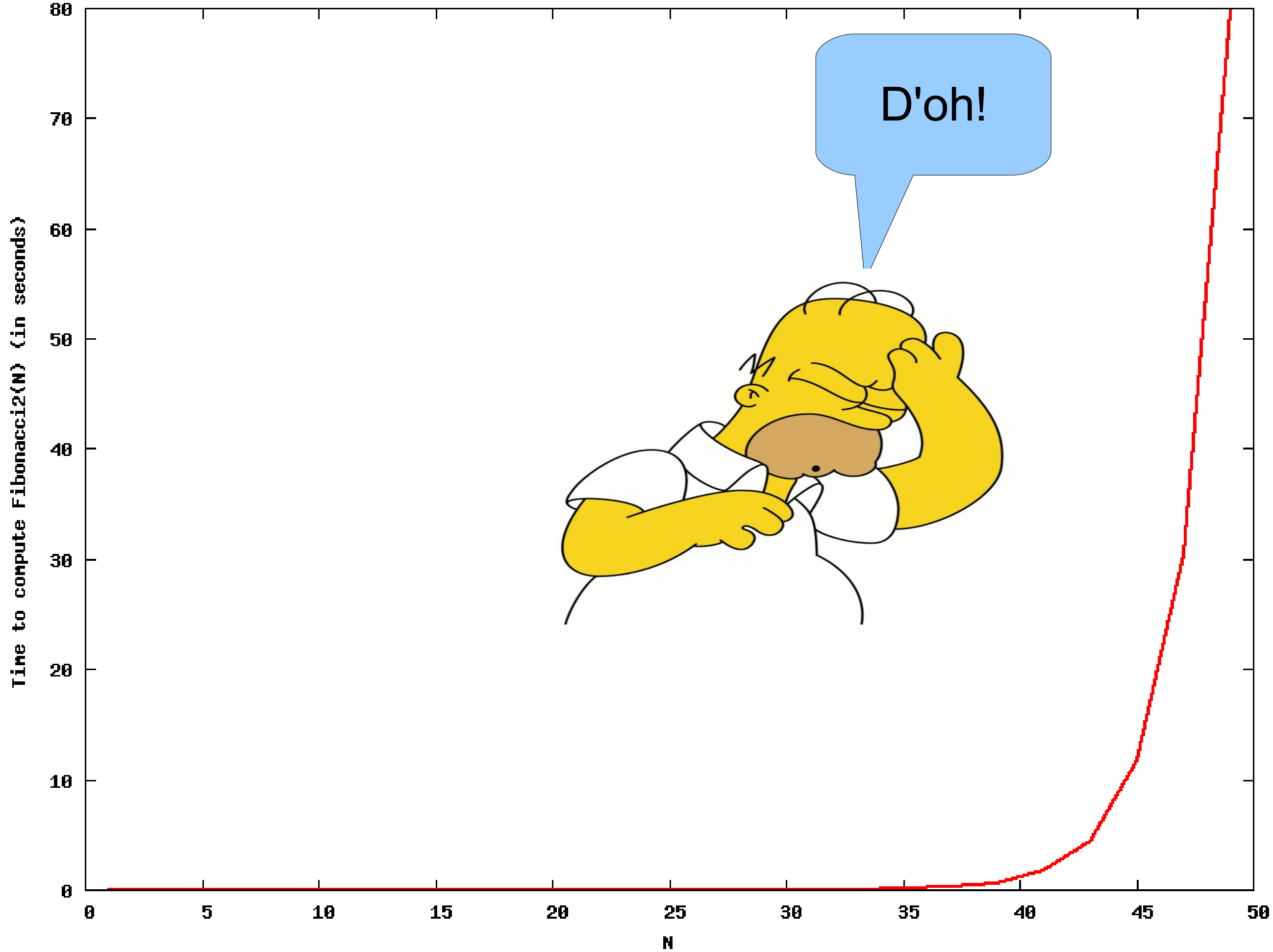
- We will use pseudo-code description of algorithms. The translation in programming languages is quite straight-forward.

Recursion tree



So far so good... but...

- Time needed to compute F_n grows too much as a function of n

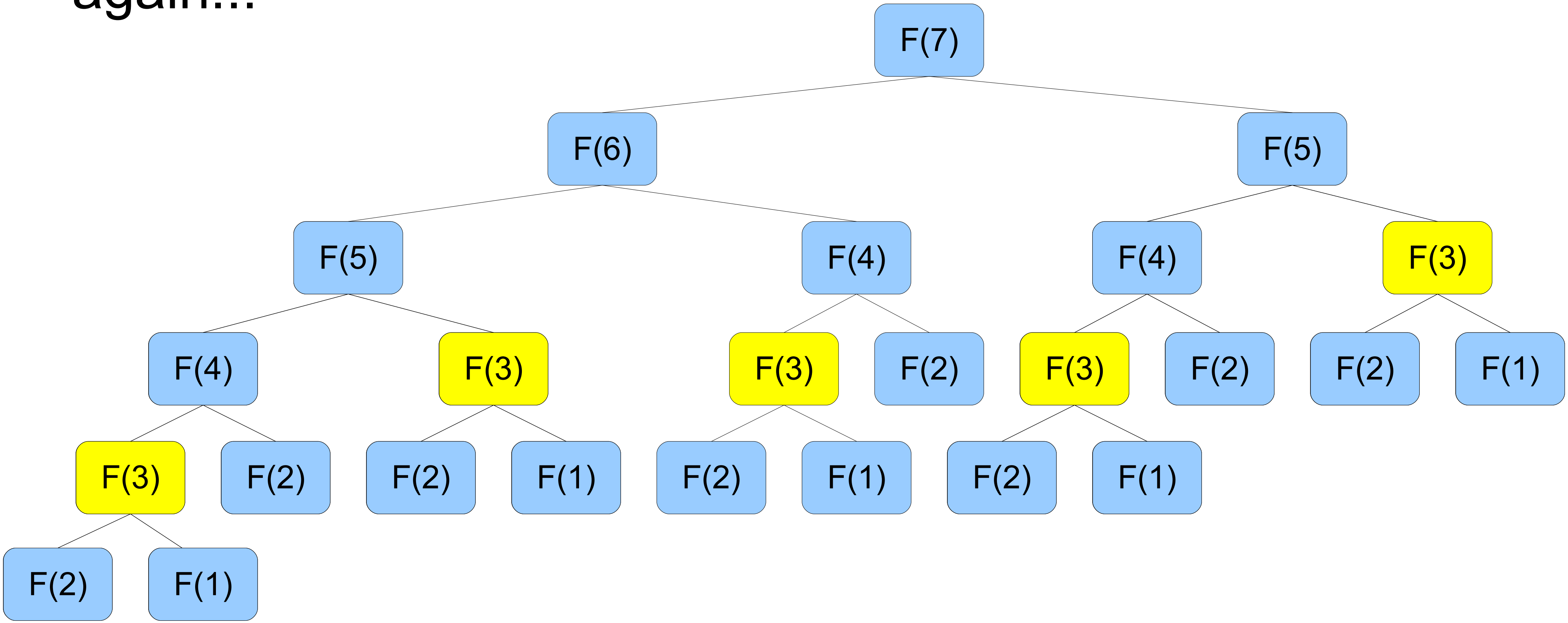


How to estimate the execution time?

- In seconds?
  - … will depend on the computer executing the program
- Number of machine language instructions executed per second?
  - Hard to estimate from pseudo-code, and also still depends on the computer executing the program
- We estimate the execution time by calculating the number of basic operations executed in the pseudo-code.

Where is the efficiency problem?

- Intermediate values are often re-calculated again and again...



Estimation of execution time

- let $T(n)$ be the time needed to compuet the n-th Fibonacci number.
- We estimate $T(n)$ as the number of nodes of the recursion tree of F_n
  - **Question**: how to obtain the recursive expression of T(n) as the number of recursive nodes in the tree for calculating F_n

Estimation of execution time

- We can demonstrate (by induction) that:

$$T(n) = 2F_n - 1$$

  - **Question**: demonstrate that.
- By remembering the close form for F_n we conclude that $T(n)$ grows exponentially
- We can calculate a lower bound for T(n)
  - See next page

Estimation of execution time

```
algorithm Fibonacci2(int n) → int
    if ( n==1 || n==2 ) then
        return 1;
    else
        return Fibonacci2(n-1)+Fibonacci2(n-2);
    endif
```

- let T(n) be the number of nodes of the recursive tree for calculating F_n
  - $T(1) = T(2) = 1$;
  - $T(n) = T(n-1) + T(n-2) + 1$ *(se* $n>2$*)*
  - It is similar to the recurrence that defines F_n

Lower bound of the execution time

$$
\begin{aligned}
T(n) &= T(n-1)+T(n-2)+1 \\
&\geq 2T(n-2)+1 \\
&\geq 4T(n-4)+2+1 \\
&\geq 8T(n-6)+2^2+2+1 \\
&\geq \ldots \\
&\geq 2^k T(n-2k)+\sum_{i=0}^{k-1} 2^i \\
&\geq \ldots \\
&\geq 2^{\lfloor n/2 \rfloor} + \frac{2^{\lfloor n/2 \rfloor}-1}{2-1} \\
&\geq 2^{\lfloor n/2 \rfloor}
\end{aligned}
$$

We exploit the fact that T(n) is monotone increasing

Recursion ends when k=n/2

Can we do it better?

- Let's use a vector of size *n* to compute and store the values of $F_1, F_2, \ldots F_n$

```
algorithm Fibonacci3(int n) → int
    let Fib[1..n] be an array of n ints
    Fib[1] := 1;
    Fib[2] := 1;
    for i:=3 to n do
        Fib[i] := Fib[i-1] + Fib[i-2];
    endfor
    return Fib[n];
```

How much does it cost?

- Let's estimate the cost of Fibonacci3 by counting the number of pseudocode operations executed

```
algorithm Fibonacci3(int n) → int
    let Fib[1..n] be an array of n integers
    Fib[1] := 1;  // .......................   1 time
    Fib[2] := 1;  // .......................   1 time
    for i:=3 to n do // ...................    (n-1) times
        Fib[i] := Fib[i-1] + Fib[i-2]; //      (n-2) times
    endfor
    return Fib[n];   // ..................     1 time
                     // Total............      2n
```

- Time is proportional to n
- Space is proportional to n

Can we do it even better?

- Memory usage of Fibonacci3 is proportional to n. Can we use less memory?
- Yes, because to calculate F_n we simply need F_{n-1} e F_{n-2}

```
algorithm Fibonacci4(int n) → int
if ( n==1 || n==2 ) then
    return 1;
else
    F_nm1 := 1;
    F_nm2 := 1;
    for i:=3 to n do
        F_n := F_nm1 + F_nm2;
        F_nm2 := F_nm1;
        F_nm1 := F_n;
    endfor
    return F_n;
endif
```

How much does it cost?

- let's count the number of operations executed

```
algorithm Fibonacci4(int n) → int
if ( n==1 || n==2 ) then
    return 1;
else
    F_nm1 := 1;   // ................ 1 time
    F_nm2 := 1;   // ................ 1 time
    for i:=3 to n do // ........... (n-1) times
        F_n := F_nm1 + F_nm2;   // . (n-2) times
        F_nm2 := F_nm1;  // ........ (n-2) times
        F_nm1 := F_n; // ........... (n-2) times
    endfor
    return F_n;   // ................ 1 time
endif

                  // Total......... 4n-4
```

- Time is proportional to n
- Space (memory) is constant!

That's all folks! Or not?

- Let's consider the matrix A:

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

- Theorem: for any $n \geq 2$, we have:

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

(demonstrable by induction)

Idea! Algorithm Fibonacci6

- We exploit the previous theorem to define algorithm Fibonacci6 as follows

```
algorithm Fibonacci6(int n) : int
```

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

```
    M = MatPow( A, n-1 );
    return M[1][1];
```

M[1][1] is the first item of the row

Yes but...Algorithm MatPow?

- To compute the k-th power of a matrix A, we exploit the fact that, for even K, $A^k = (A^{k/2})^2$

```
algorithm MatPow(Matrix A, int k) → Matrix
if ( k==0 ) then
```

$$M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
else
    if ( k is even ) then
        tmp := MatPow(A,k/2)
        M := tmp '*' tmp;
    else
        tmp := MatPow(A,(k-1)/2);
        M := tmp '*' tmp '*' A;
    endif
endif
return M;
```

operator '*' computes the product of matrices

To sum up

Exponential time

Logaritmic time

Algorithm	*Time*	*Memory*
Fibonacci2	$\Omega(2^{n/2})$	$O(n)$
Fibonacci3	$O(n)$	$O(n)$
Fibonacci4	$O(n)$	$O(1)$
Fibonacc6	$O(\log n)$	$O(\log n)$

Lessons learned?

- For a given problem, we started from a inefficient algorithm (exponential cost) to reach a very efficient algorithm (logaritmic cost).
- The choice of the good algorithm makes the difference between being able to solve a problem or NOT.

Warmup exercise

- Given an array A[1..n-1] containing a permutation of all values 1 - n (extremes included) but one; values in A can be in any order
  - Eg: A = [1, 3, 4, 5] is a permutation of 1..5 without the value 2
  - Eg: A = [7, 1, 3, 5, 4, 2] is a permutation of 1..7 without the value 6
- Let's write an algorithm which takes A[1..n-1], and returns the value in the interval 1..n which is not in A.