BIG DIVE

TECH. CUSTOM EDITION

A project by TOP-IX
designed for Intesa Sanpaolo

#DataScience

aizoon technology consulting

Algorithms

Alan Perotti, PhD

AUSTRALIA
EUROPE
USA
TECHNOLOGY CONSULTING

What makes a good algorithm?

- Correctness - *will I get the right result?*
- Efficiency - *will I get the result by the time I need it?*
- Elegance - *would anyone understand my code?*

Computational complexity

Time: How long is it going to take?

Space: How much memory is it going to take?

These are mathematical functions defined over the dimension of the input: if the input is an array, it's the number of elements; if it's a file, it's the size in bytes. etc.

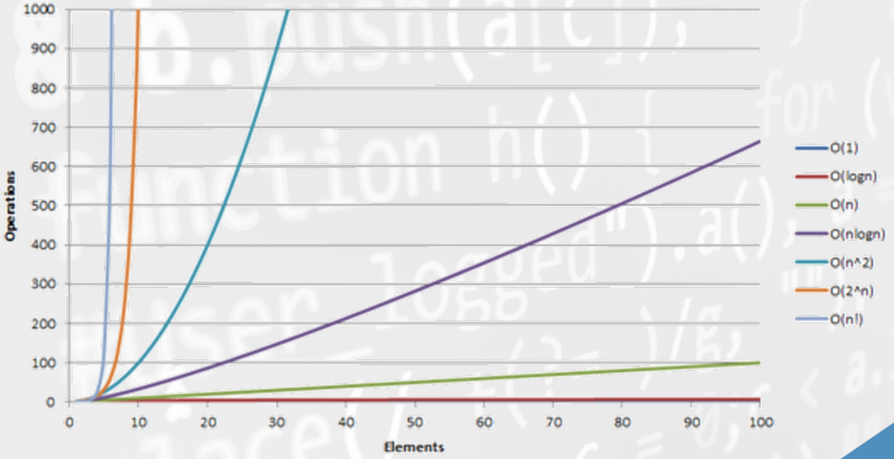
Lil' math: functions!

A computational complexity analysis takes an algorithm and produces a function. So, instead of directly comparing algorithms (infinite), we compare the corresponding functions, and we use them to classify the algorithms.

- Constant
- Logarithmic
- Linear
- Polynomial
- Exponential
- Super-exponential

plus all combinations (e.g. NlogN)

Curve comparison



La méthode

So how to 'extract' a complexity function from an algorithm? There are lots of non-trivial techniques (recurrence relations, telescoping, combinatorics) - but for simple algorithms it's ok to go 'by eye'.
For instance, a single operation is a constant offset, a for-loop is a linear factor, etc.

Informal approach: take a few inputs and roughly estimate the number of operations required to compute the algorithm in each case.
Interpolate to get a function.
Then ask yourself: can I do better than this?

Bounds

Upper Bound ← Practical: your best algorithm so far

Room for improvement!

Lower Bound ← Theoretical: min num of ops

Informal examples | 1

How many ops if the array has 10 elements? 100? 1000?
Can I do better?

```
def get_first_element(array):
    return array[0]
```

The complexity is CONSTANT. It’s the best function, so I can’t do better than that.

Informal examples | 2

How many ops if the array has 10 elements? 100? 1000?
Can I do better?

```
def find_max_element(array):
    temp = array[0]
    for element in array:
        if element > temp:
            temp = element
    return temp
```

The complexity is LINEAR. In order to find out the best element, I need to check all of them, so I need to be at least linear. Therefore, I cannot do better than this.

Informal examples | 3

How many ops if the array has 10 elements? 100? 1000?
Can I do better?

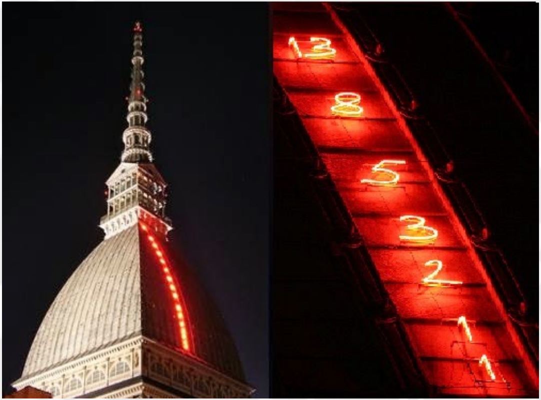
```
def get_last_element(array):
    temp = array[0]
    for element in array:
        temp = element
    return temp
```

The complexity is LINEAR. In order to find the last element, I DON'T need to check all of them: in fact, I only need to check the last one. My lower bound is consequently CONSTANT, so this algorithm can be improved.

From rabbits to golden ratio

Fibonacci(k) = Fibonacci(k-1)+Fibonacci(k-2)

(0),1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584..



Fibonacci: two simple algorithms

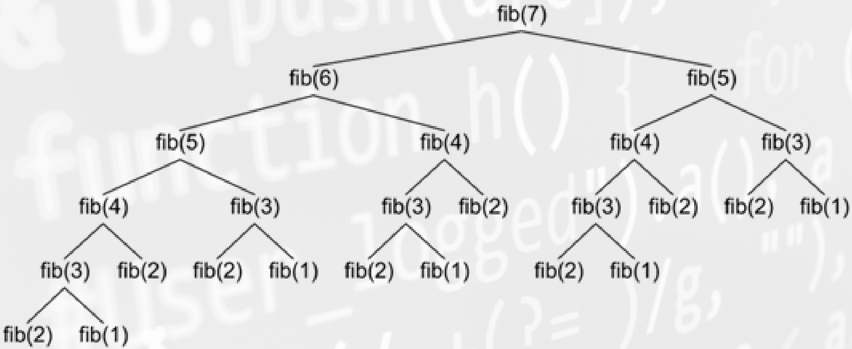
Recursive

```
def fib_rec(n):
    if n<2:
        return n
    else:
        return fib_rec(n-2)+fib_rec(n-1)
```

Iterative

```
def fib_it(n):
    low,high = 0,1
    for i in range(n):
        low,high = high, low+high
    return low
```

How many recursive calls?



Fibonacci: two simple algorithms

Recursive EXPONENTIAL: 2^n

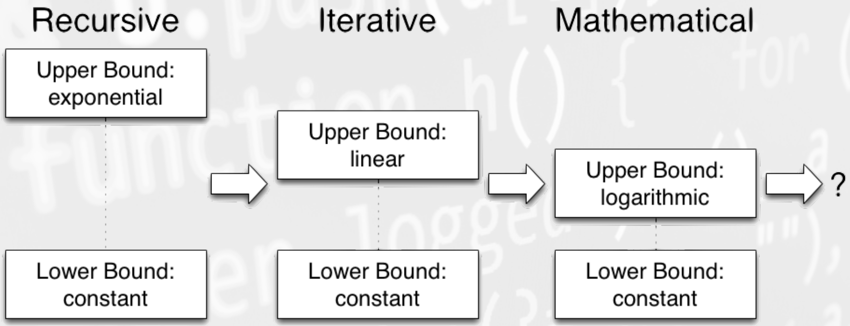
```
def fib_rec(n):
    if n<2:
        return n
    else:
        return fib_rec(n-2)+fib_rec(n-1)
```

Iterative LINEAR: *n*

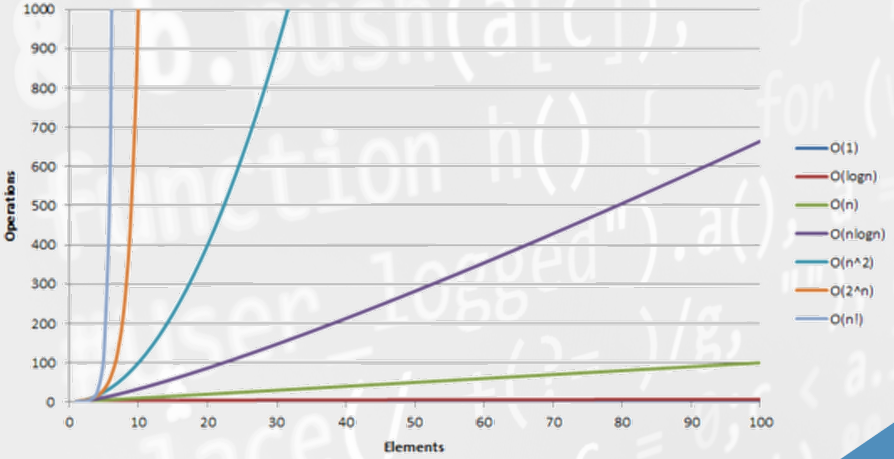
```
def fib_it(n):
    low,high = 0,1
    for i in range(n):
        low,high = high, low+high
    return low
```

(there is also an algorithm with log(n) complexity)

Bounds


Recursive
Iterative
Mathematical
Upper Bound: exponential
Upper Bound: linear
Upper Bound: logarithmic
?
Lower Bound: constant
Lower Bound: constant
Lower Bound: constant

Curve comparison AGAIN



Value comparison

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	$\boxed{10^6}$
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$\boxed{10^{301,030}}$

Correctness vs Efficiency

- We had two seemingly very similar algorithms for computing the n-th Fibonacci number.
- In the same amount of time required for *fib_it* to compute *fibonacci(1M)*, *fib_rec* can compute *fibonacci(38)*
- We found out that, if asked to compute the millionth Fibonacci number, *fib_it* would perform $\sim 10^6$ steps ($\sim$ 15 seconds on my laptop), while *fib_rec* would need to go through $\sim 10^{301030}$ operations (an unthinkable amount of time! Age of the Universe? $\sim 10^{10}$ years..).

Can I just buy a faster PC?

Let's take our *fib_rec* algorithm. Over 1M data points, the algorithm will perform $10^{301.030}$ ops. Let's assume, for the sake of simplification, that every op requires 1 microsecond, so the running time for our algorithm over 1M data points is $10^{301.024}$ seconds : order of magnitude, $10^{301.016}$ years.

What if I buy a PC which is 1000 times faster? Order of magnitude, $10^{301.013}$ years. What if I buy 1000 PCs, each being 1000 times faster? Order of magnitude, $10^{301.010}$ years.

For bad algorithms on big data, the impact of getting more computational power is negligible: **IT HAS NO IMPACT ON THE COMPLEXITY.**

Furthermore, what if the number of data points doubles?
The complexity blows up, from $10^{301.030}$ to $10^{602.060}$ ops!

Is this a completely theoretical problem?

NO

When dealing with big data, the scalablity/complexity of algorithms is paramount. E.g.,

- an algorithm seemingly fast on a data sample might be useless for the whole dataset.
- if the loaded data batch almost fits the RAM, algorithms with worse-than-constant space complexity would break / cause huge swapping.

Any funny story before we're done?

- Many cryptosystems are based on RSA.
- The math core of RSA decryption is the integer factorisation of semiprimes.
- (sub)exponential factorisation algorithms exist - they are just **too slow to be scary!**
- At the same time, there is no guarantee that polynomial algorithms do not exist!
- So RSA relies on a so-called *cryptographic hardness* **assumption**.

Take-home message

1. For big data, efficience is CRUCIAL
2. You can't throw RAM at a bad algorithm
3. Learn to roughly estimate the complexity functions
4. If you get an exponential (or a high-degree polynomial) and you're far from the lower bound, you need to optimise your algorithm