Why is my code slow?

Sebastiano Tronto

2021-05-21

UNIVERSITÉ DU LUXEMBOURG

- **Goal:** estimate the running time of a program
- **How:** count the basic steps that an algorithm takes to complete
- **Why**: find the *bottleneck* of your program, make it faster

Our analysis should not depend on the hardware

UNIVERSITÉ DU LUXEMBOURG

Algorithm

Definition

An algorithm is a sequence of steps needed to solve a class of problems.

Definition (alternative)

An algorithm is a sequence of steps that takes an input satisfying certain conditions and produces an output satisfying other conditions.

Sorting a list

Class of problems

Sort a list L of numbers in increasing order.

Algorithm

1. Let S be an empty list.
2. Take an element from L an insert it in S in its correct position.
3. Repeat step 2 until L is empty.
4. Return S.

UNIVERSITÉ DU LUXEMBOURG

Sorting a list

- It solves a *class* of problems: works for any list
- The specific steps to sort the list $[3, 7, 1]$ are not an algorithm
- Input conditions: must be a list of numbers
- Output conditions: same numbers in increasing order

UNIVERSITÉ DU LUXEMBOURG

How to write an algorithm

- **Human language**:
  - Easy to understand
  - Not precise
- **Computer code**:
  - Can be executed by computers
  - Precise
  - From very low level (machine code) to high level (Python, ...)

UNIVERSITÉ DU LUXEMBOURG

Basic steps

- Arithmetic operations $+, -, *, //, \%$
- Relational operations $==, ! =, >, <, \dots$
- Memory access (read/write variable)

Warning: Depends on data type (integer, floating point, string,...)

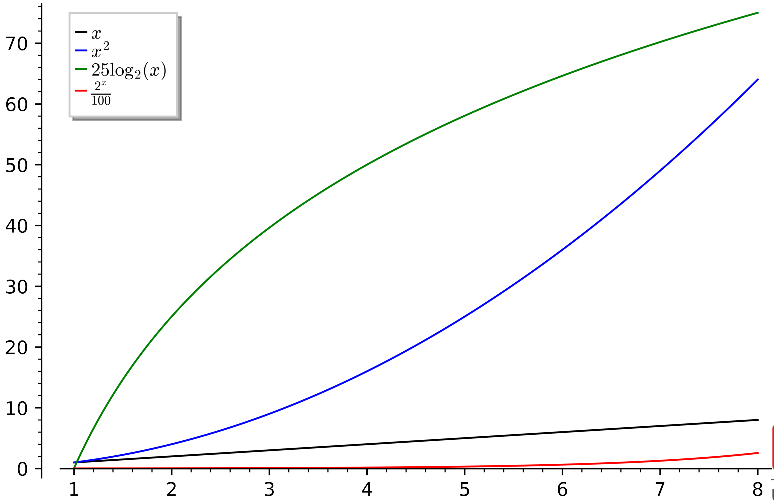
UNIVERSITÉ DU LUXEMBOURG

Running time

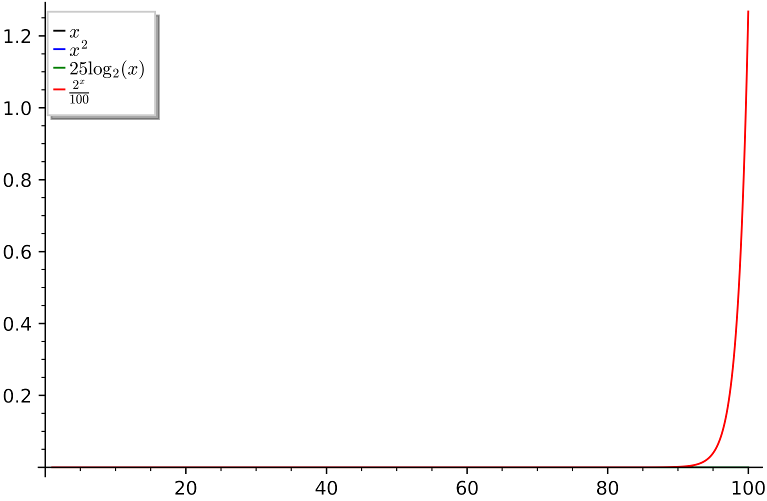
- Depends on computer power, programming language, compiler...
- "Big O" notation: an algorithm runs in time $O(f(n))$ if, when run with input of size n, it takes about $c \cdot f(n)$ steps
- Algorithm A is *asymptotically faster* than algorithm B if it is faster **for n large enough**
- Rule of thumb: $10^7 \sim 10^9$ basic steps per second

UNIVERSITÉ DU LUXEMBOURG

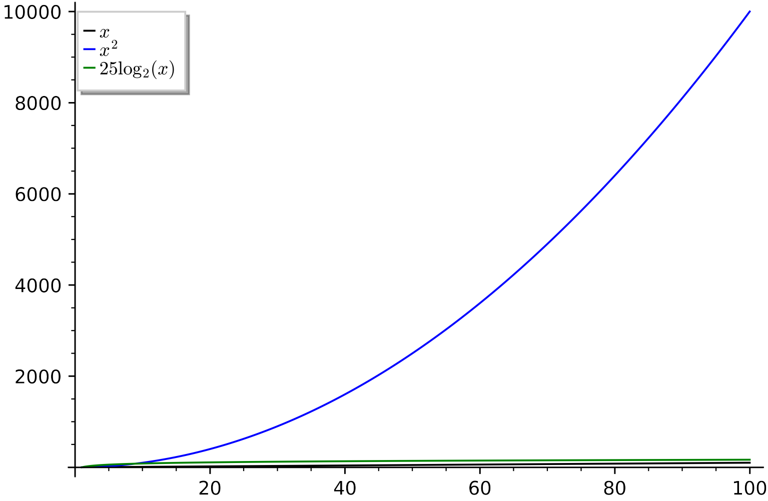
Asymptotical analysis vs constant factors

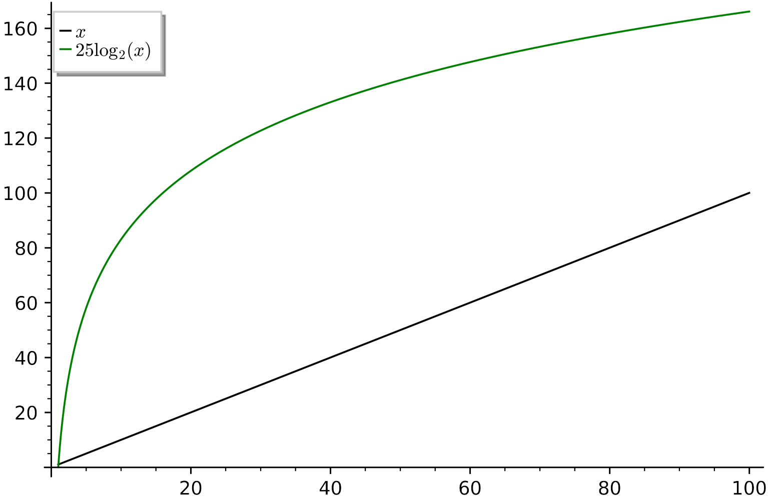


Asymptotical analysis vs constant factors

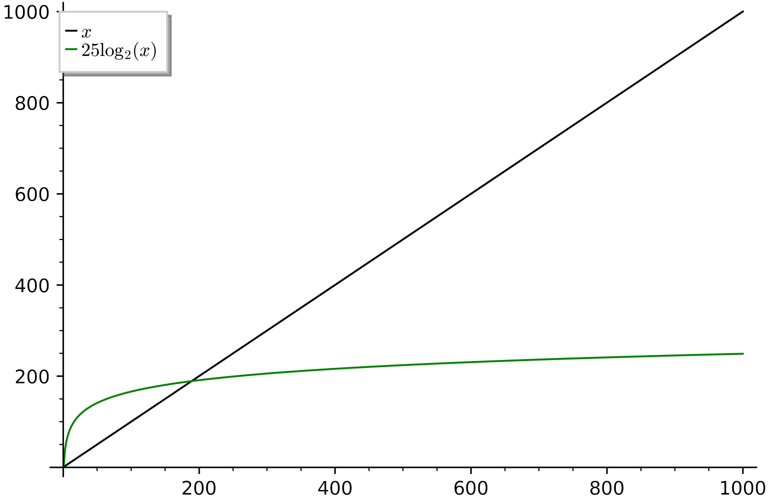


Asymptotical analysis vs constant factors




x
$25\log_2(x)$
160
140
120
100
80
60
40
20
20
40
60
80
100

Asymptotical analysis vs constant factors



Basic complexity analysis

Easy things to do:

- Check documentation for "non-basic steps"
  - Example: check Sage's `is_prime()` (redirects to PARI `isprime()`)
- Count nested loops
  - How many times is a step repeated?

Nested loops - matrix sum and product

```
def add(A, B):
    n = len(A)
    S = [[0] * n for i in range(n)]
    for i in range(0, n):
        for j in range(0, n):
            S[i][j] = A[i][j] + B[i][j]
    return S
```

```
def prod(A, B):
    n = len(A)
    S = [[0] * n for i in range(n)]
    for i in range(0, n):
        for j in range(0, n):
            for k in range(0, n):
                S[i][j] = S[i][j] + A[i][k]*B[k][j]
    return S
```

Nested loops - matrix sum and product

- `add` is $O(n^2)$ (two loops)
- `prod` is $O(n^3)$ (three loops)

Fun fact: there are faster algorithms for matrix multiplication, for example Strassen's algorithm.

UNIVERSITÉ DU LUXEMBOURG

Sorting a list

```python
def correct_position(e, S):
    for i in range(0, len(S)):
        if S[i] > e:
            return i
    return len(S)

def sort_list(L):
    S = []
    for e in L:
        cp = correct_position(e, S)
        S.insert(cp, e)
    return S
```

Sorting a list

- Complexity of `correct_position()`:
  - worst case $O($`len(S)`$)$
  - average $O($`len(S)`$)$

- Complexity of `sort_list` (here $n =$`len(L)`):

$$\sum_{i=0}^{n-1} O(i) = O(n^2)$$

(it calls `correct_position()` n times).

Sorting a list

- For which lists does the "best case" happen?
- For which lists does the "worst case" happen?
- How large can n be for `sort_list()` to run in under a second?

Sorting a list

How to improve our code?

- Improve `correct_position()`
- Take advantage of the fact that S is always sorted

Binary search

Algorithm

Input: a *sorted* list S and a value e.

1. If the list is empty, you have found the position of e
2. Otherwise, compare e to the middle element m of S
   - If $e < m$, repeat from (1) on the first half of S
   - Otherwise, repeat from (1) on the second half of S

UNIVERSITÉ DU LUXEMBOURG

Binary search

```python
# Return position of e in L
def binary_search(e, S, start, end):
    if start == end:
        return start
    midpoint = (end+start)//2
    if e < S[midpoint]:
        return binary_search(e, S, start, midpoint)
    else:
        return binary_search(e, S, midpoint+1, end)
```

UNIVERSITÉ DU LUXEMBOURG

Binary search - example 1

Searching for e= 2:

$$\underbrace{\overset{\overset{\texttt{start}=0}{\downarrow}}{-2}\quad 0\quad 1\quad 3\quad \overset{\overset{\texttt{midpoint}=4}{\downarrow}}{5}\quad 6\quad 7\quad 9\quad 12}\quad \overset{\texttt{end}=9}{\downarrow}$$

$e < 5 \implies$ check left half

Binary search - example 1

Searching for e= 2:

$$\begin{array}{ccccccccc} \scriptstyle\color{blue}{\texttt{start}=0} & & \scriptstyle\texttt{midpoint}=2 & & \scriptstyle\color{red}{\texttt{end}=4} & & & & \\ \color{blue}{\downarrow} & & \downarrow & & \color{red}{\downarrow} & & & & \\ \underbrace{-2 & 0 & 1 & 3} & 5 & 6 & 7 & 9 & 12 \end{array}$$

$\color{red}{e > 1} \implies$ check right half

Binary search - example 1

Searching for e= 2:

$$\begin{array}{ccccccccc} & & & \color{blue}{\texttt{start}}=\texttt{midpoint}=3 & \color{red}{\texttt{end}=4} & & & & \\ & & & \color{blue}{\downarrow} & \color{red}{\downarrow} & & & & \\ -2 & 0 & 1 & \underbrace{3} & 5 & 6 & 7 & 9 & 12 \end{array}$$

$\color{blue}{e < 3} \implies$ check left half

Binary search - example 1

Searching for e$= 2$:

$$\begin{array}{ccccccccc} & & & \overset{\texttt{start}=\texttt{end}=3}{\downarrow} & & & & & \\ -2 & 0 & 1 & 3 & 5 & 6 & 7 & 9 & 12 \end{array}$$

start=end, done

Binary search - example 2

Searching for e$=$ 11:

$$\begin{array}{cccccccccc} \overset{\texttt{start}=0}{\downarrow} & & & & \overset{\texttt{midpoint}=4}{\downarrow} & & & & & \overset{\texttt{end}=9}{\downarrow} \\ \end{array}$$

$$\underbrace{-2 \quad 0 \quad 1 \quad 3 \quad 5 \quad 6 \quad 7 \quad 9 \quad 12}$$

$e > 5 \implies$ check right half

Binary search - example 2

Searching for e= 11:

$$
\begin{array}{cccccccccc}
 & & & & & \overset{\texttt{start}=5}{\downarrow} & & \overset{\texttt{midpoint}=7}{\downarrow} & & \overset{\texttt{end}=9}{\downarrow} \\
-2 & 0 & 1 & 3 & 5 & \underbrace{6 \quad 7 \quad 9 \quad 12} & & & &
\end{array}
$$

$e > 9 \implies$ check right half

Binary search - example 2

Searching for e= 11:

$$\begin{array}{ccccccccccc} & & & & & & & & \overset{\texttt{start}=\texttt{midpoint}=8}{\downarrow} & \overset{\texttt{end}=9}{\downarrow} \\ -2 & 0 & 1 & 3 & 5 & 6 & 7 & 9 & \underbrace{12} & \end{array}$$

$e < 11 \implies$ check left half

Binary search - example 2

Searching for e$=$ 11:

$$
\begin{array}{ccccccccc}
 & & & & & & & & \overset{\texttt{start}=\texttt{end}=8}{\downarrow} \\
-2 & 0 & 1 & 3 & 5 & 6 & 7 & 9 & 12
\end{array}
$$

`start`$=$`end`, done

Binary search

- Works only if the list is sorted
- Complexity $O(\log_2(n))$: at every step we cut the list in half
- Recursive, *divide et impera*

UNIVERSITÉ DU LUXEMBOURG

Sorting a list - binary search version

```python
def sort_list(L):
    S = []
    for e in L:
        cp = binary_search(e, S, 0, len(S)) # This changed
        S.insert(cp, e)
    return S
```

- Complexity:

$$\sum_{i=0}^{n-1} O(\log_2(i)) = O(n \log_2(n))$$

(it calls `binary_search` n times).

Fast exponentiation

Algorithm / formula

$$a^n = \begin{cases} 1 & \text{if } n = 0, \\ (a \cdot a)^{\frac{n}{2}} & \text{if } n \text{ is even}, \\ a \cdot a^{n-1} & \text{if } n \text{ is odd}. \end{cases}$$

UNIVERSITÉ DU LUXEMBOURG

Fast exponentiation

```
# Compute a^n (n>=0 integer)
def power(a, n):
    if n == 0:
        return 1
    if n % 2 == 0:      # n is even
        return power(a*a, n//2)
    else:               # n is odd
        return a*power(a, n-1)
```

Fast exponentiation

- Complexity: $O(\log_2(n))$ (after 2 steps, n is halved)
- Python's operator $**$ does something similar
- Naive algorithm (one loop): $O(n)$

UNIVERSITÉ DU LUXEMBOURG

Fast gcd

Algorithm / formula

$$\gcd(a,b) = \begin{cases} a & \text{if } b = 0, \\ \gcd(b, a \bmod b) & \text{otherwise.} \end{cases}$$

```python
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a%b)
```

- After 2 steps, a is halved $\implies$ complexity $O(\log_2(a))$

UNIVERSITÉ DU LUXEMBOURG

Recursion

- These examples use *recursion* (a function that calls itself)
- If it calls itself more than once, it is slow (*exponential* complexity!)

Fibonacci numbers

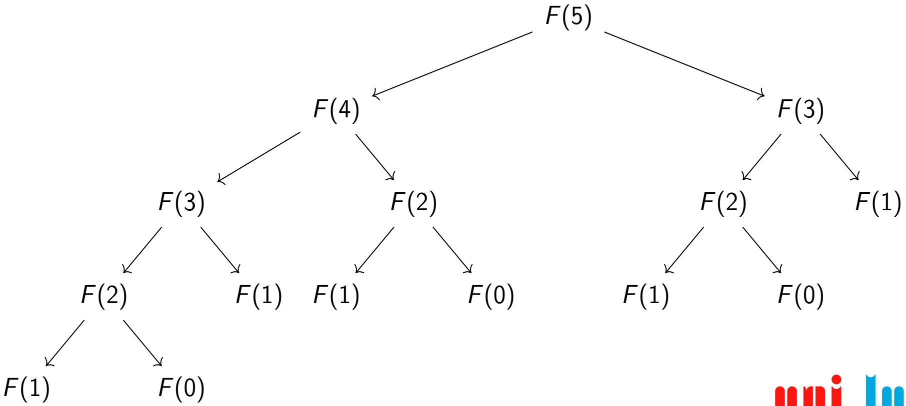
Algorithm / formula

$$F(n) = \begin{cases} n & \text{if } n \leq 1, \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases}$$

```python
def F(n):
    if n <= 1:
        return n
    else:
        return F(n-1) + F(n-2)
```

UNIVERSITÉ DU LUXEMBOURG

Fibonacci



Fibonacci

- Complexity: almost $O(2^n)$ (actually $O(\varphi^n)$ with $\varphi = \frac{1+\sqrt{5}}{2} \sim 1.6$)
- But some values are computed many times!
- Optimization: memorize previously computed values

Fibonacci with memorization

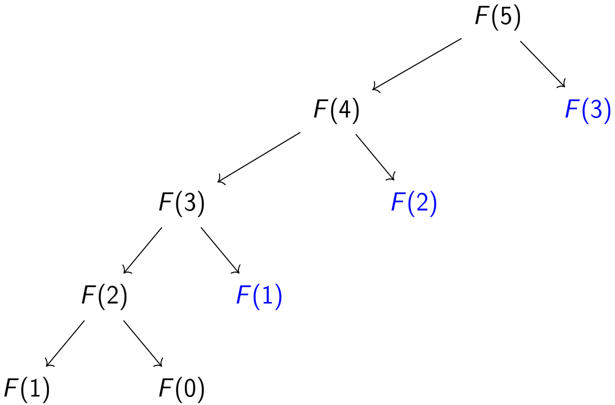
```python
# List with memorized values, N is the largest possible
N = 10**6
F_memorized = [-1] * N

def F(n):
    if F_memorized[n] == -1:
        if n <= 1:
            F_memorized[n] = n
        else:
            F_memorized[n] = F(n-1) + F(n-2)

    return F_memorized[n]
```

UNIVERSITÉ DU LUXEMBOURG

Fibonacci with memorization



Fibonacci with memorization

- Complexity: $O(n)$, huge improvement!
- Further improvement (but still $O(n)$): dynamic programming
- Pay attention to memory usage

UNIVERSITÉ DU LUXEMBOURG

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein - *Introductions to Algorithms*

UNIVERSITÉ DU LUXEMBOURG