What do we expect from an algorithm?

- Correct
- Easy to understand
- Easy to implement
- Efficient:
  - Every algorithm requires a set of resources
    - Memory
    - CPU time
    - Energy

Fibonacci: recursive version

```python
def fib(n: int) -> int:
    """Returns the Fibonacci number of order n
    Pre: n ≥ 0
    ""
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

How many recursive calls?
Fibonacci: runtime

\[ T(0) = 1 \]
\[ T(1) = 1 \]
\[ T(n) = T(n-1) + T(n-2) \]

Let us assume that \( T(n) = a^n \) for some constant \( a \). Then,
\[ a^n = a^{n-1} + a^{n-2} \Rightarrow a^2 = a + 1 \]
\[ a = \frac{1 + \sqrt{5}}{2} = \varphi \approx 1.618 \text{ (golden ratio)} \]

Therefore, \( T(n) \approx 1.6^n \).

If \( T(0) = 1 \text{ ns} \), then \( T(100) \approx 2.6 \cdot 10^{20} \text{ ns} > 8000 \text{ yrs.} \)

With the age of Universe (14 \cdot 10^9 \text{ yrs}), we could compute up to \( \text{fib}(128) \).

Fibonacci numbers

Algebraic solution: find matrix \( A \) such that

\[
\begin{bmatrix}
F_{n+2} \\
F_{n+1}
\end{bmatrix} = \begin{bmatrix}
? & ? \\
? & ?
\end{bmatrix} \cdot \begin{bmatrix}
F_{n+1} \\
F_n
\end{bmatrix}
\]

\[
\begin{bmatrix}
F_{n+2} \\
F_{n+1}
\end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix}
F_{n+1} \\
F_n
\end{bmatrix}
\]

\[
\begin{bmatrix}
F_{n+1} \\
F_n
\end{bmatrix} = A^n \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}
\]

\[
A^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad A^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}
\]

\[
A^4 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \quad A^8 = \begin{bmatrix} 34 & 21 \\ 21 & 13 \end{bmatrix}
\]

\[
A^{16} = \begin{bmatrix} 1597 & 987 \\ 987 & 610 \end{bmatrix} \quad \cdots A^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}
\]

Runtime \( \approx \log_2 n \) 2x2 matrix multiplications
Algorithm analysis

Given an algorithm that reads inputs from a domain $D$, we want to define a cost function $C$:

$$C : D \to \mathbb{R}^+$$

$x \mapsto C(x)$

where $C(x)$ represents the cost of using some resource (CPU time, memory, energy, ...).

Analyzing $C(x)$ for every possible $x$ is impractical.

Algorithm analysis: simplifications

• Analysis based on the size of the input: $|x| = n$

• Only the best/average/worst cases are analyzed:

$$C_{\text{worst}}(n) = \max \{ C(x) : x \in D, |x| = n \}$$

$$C_{\text{best}}(n) = \min \{ C(x) : x \in D, |x| = n \}$$

$$C_{\text{avg}}(n) = \sum_{x \in D, |x| = n} p(x) \cdot C(x)$$

$p(x)$: probability of selecting input $x$ among all the inputs of size $n$.

Algorithm analysis

• Properties:

  - $\forall n \geq 0 : \quad C_{\text{best}}(n) \leq C_{\text{avg}}(n) \leq C_{\text{worst}}(n)$
  - $\forall x \in D : \quad C_{\text{best}}(|x|) \leq C(x) \leq C_{\text{worst}}(|x|)$

• We want a notation that characterizes the cost of algorithms independently from the technology (CPU speed, programming language, efficiency of the compiler, etc.).

• Runtime is usually the most important resource to analyze.

Asymptotic notation

Let us consider all functions $f : \mathbb{R}^+ \to \mathbb{R}^+$

Definitions:

$$O(f(n)) = \{ g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \leq k \cdot f(n) \}$$

$$\Omega(f(n)) = \{ g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \geq k \cdot f(n) \}$$

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$
Asymptotic notation

\[ k_1 f_1(n) \]
\[ g(n) \]
\[ k_2 f_2(n) \]

\[ g(n) \in O\left(f_1(n)\right) \]
\[ g(n) \in \Omega\left(f_2(n)\right) \]

Asymptotic notation: example

\[ O(n^2) = \{3n^2 - n + 20, \]
\[ 0.5n^2 - 3, \]
\[ 4n \log_2 n, \]
\[ 2n - 5, \]
\[ n < 20 \ ? \ 2^n : 2n^2 + 1000, \]
\[ \ldots \} \]
\[ \Omega(n^3) = \{3n^2 - n + 20, \]
\[ 0.5n^2 - 3, \]
\[ 3n^3 \log_2 n, \]
\[ 2^n - 4n^3, \]
\[ n < 20 \ ? \ 2^n : 2n^2 + 1000, \]
\[ \ldots \} \]
\[ \Theta(n^2) = O(n^2) \cap \Omega(n^2) = \{3n^2 - n + 20, \]
\[ 0.5n^2 - 3, \]
\[ n < 20 \ ? \ 2^n : 2n^2 + 1000, \]
\[ \ldots \} \]

Examples

\[ 13n^3 - 4n + 8 \in O(n^3) \]
\[ 2n - 5 \in O(n) \]
\[ n^2 \notin O(n) \]
\[ 2^n \in O(n!) \]
\[ 3^n \notin O(2^n) \]
\[ 3 \log_2 n \in O(\log n) \]
\[ n \log_2 n \in O(n^2) \]
\[ O(n^2) \subseteq O(n^3) \]

\[ 13n^3 - 4n + 8 \in \Omega(n^3) \]
\[ n^2 \in \Omega(n) \]
\[ n^2 \notin \Omega(n^3) \]
\[ n! \in \Omega(2^n) \]
\[ 3^n \notin \Omega(2^n) \]
\[ 3 \log_2 n \in \Omega(\log n) \]
\[ n \log_2 n \in \Omega(n) \]
\[ \Omega(n^3) \subseteq \Omega(n^2) \]
# Complexity ranking

<table>
<thead>
<tr>
<th>Function</th>
<th>Common name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n!$</td>
<td>factorial</td>
</tr>
<tr>
<td>$2^n$</td>
<td>exponential</td>
</tr>
<tr>
<td>$n^d$, $d &gt; 3$</td>
<td>polynomial</td>
</tr>
<tr>
<td>$n^3$</td>
<td>cubic</td>
</tr>
<tr>
<td>$n^2$</td>
<td>quadratic</td>
</tr>
<tr>
<td>$n \sqrt{n}$</td>
<td></td>
</tr>
<tr>
<td>$n \log n$</td>
<td>quasi-linear</td>
</tr>
<tr>
<td>$n$</td>
<td>linear</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>root - $n$</td>
</tr>
<tr>
<td>$\log n$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>1</td>
<td>constant</td>
</tr>
</tbody>
</table>

# The limit rule

Let us assume that $L$ exists (may be $\infty$) such that:

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

\[
\begin{cases}
  \text{if } L = 0 & \text{then } f \in O(g) \\
  \text{if } 0 < L < \infty & \text{then } f \in \Theta(g) \\
  \text{if } L = \infty & \text{then } f \in \Omega(g)
\end{cases}
\]

**Note:** If both limits are $\infty$ or 0, use L'Hôpital rule:

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$$

# Properties

- $f \in O(f)$
- $\forall c > 0, \ 0(f) = O(c \cdot f)$
- $f \in O(g) \land g \in O(h) \Rightarrow f \in O(h)$
- $f_1 \in O(g_1) \land f_2 \in O(g_2)$
  \[
  \Rightarrow f_1 + f_2 \in O(g_1 + g_2) = O(\max \{g_1, g_2\})
  \]
- $f \in O(g) \Rightarrow f + g \in O(g)$
- $f_1 \in O(g_1) \land f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$
- $f \in O(g) \iff g \in \Omega(f)$

# Asymptotic complexity (small values)
Asymptotic complexity (larger values)

Algorithm Analysis © Dept. CS, UPC

Let us consider that every operation can be executed in 1 ns (10^{-9} s).

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_{2} n</td>
<td>10 ns</td>
</tr>
<tr>
<td>√n</td>
<td>31.6 ns</td>
</tr>
<tr>
<td>n</td>
<td>1 μs</td>
</tr>
<tr>
<td>n log_{2} n</td>
<td>10 μs</td>
</tr>
<tr>
<td>n^2</td>
<td>1 ms</td>
</tr>
<tr>
<td>n^3</td>
<td>1 s</td>
</tr>
<tr>
<td>n^4</td>
<td>16.7 min</td>
</tr>
<tr>
<td>2^n</td>
<td>3.4 \cdot 10^{284} yr</td>
</tr>
</tbody>
</table>

How about “big data”?

Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

The robot and the door in an infinite wall

A robot stands in front of a wall that is infinitely long to the right and left side. The wall has a door somewhere and the robot has to find it to reach the other side. Unfortunately, the robot can only see the part of the wall in front of it.

The robot does not know neither how far away the door is nor what direction to take to find it. It can only execute moves to the left or right by a certain number of steps.

Let us assume that the door is at a distance $d$. How to find the door in a minimum number of steps?

This is often the practical limit for big data
Algorithm 1:
- Pick one direction and move until the door is found.

Complexity:
- If the direction is correct \( \Rightarrow O(d) \).
- If incorrect \( \Rightarrow \) the algorithm does not terminate.

Algorithm 2:
- 1 step to the left,
- 2 steps to the right,
- 3 steps to the left, ...
- ... increasing by one step in the opposite direction.

Complexity:
\[
T(d) = 3d + \sum_{i=1}^{d-1} 4i = 3d + 4 \frac{d(d - 1)}{2} = 2d^2 + d = O(d^2)
\]

Algorithm 3:
- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 3 steps to the left and return to origin,...
- ... increasing by one step in the opposite direction.

Complexity:
\[
T(d) = d + \sum_{i=1}^{d} 2i = d + 2 \frac{d(d + 1)}{2} = d^2 + 2d = O(d^2)
\]

Algorithm 4:
- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 4 steps to the left and return to origin,...
- ... doubling the number of steps in the opposite direction.

Complexity (assume that \( d = 2^n \)):
\[
T(d) = d + 2 \sum_{i=0}^{n} 2^i = d + 2(2^{n+1} - 1) = 5d - 2 = O(d)
\]
Runtime analysis rules

• Variable declarations cost no time.

• **Elementary operations** are those that can be executed with a small number of basic computer steps (an assignment, a multiplication, a comparison between two numbers, etc.).

• Vector sorting or matrix multiplication are not elementary operations.

• We consider that the cost of elementary operations is $O(1)$.

Consecutive statements:
– If $S_1$ is $O(f)$ and $S_2$ is $O(g)$, then $S_1;S_2$ is $O(\max\{f,g\})$

Conditional statements:
– If $S_1$ is $O(f)$, $S_2$ is $O(g)$ and $B$ is $O(h)$, then $\text{if } (B) \text{ } S_1; \text{ else } S_2;$ is $O(\max\{f+h,g+h\})$, or also $O(\max\{f,g,h\})$.

For/While loops:
– Running time is at most the running time of the statements inside the loop times the number of iterations

Nested loops:
– Analyze inside out: running time of the statements inside the loops multiplied by the product of the sizes of the loops

- For i in range(n):
  - for j in range(n):
    - do_something() # O(1)  $\Rightarrow O(n^2)$

- for i in range(n):
  - for j in range(i, n):
    - do_something() # O(1)  $\Rightarrow O(n^2)$

- for i in range(n):
  - for j in range(m):
    - for k in range(p):
      - do_something() # O(1)  $\Rightarrow O(n \cdot m \cdot p)$
Linear time: $O(n)$

Running time proportional to input size

# Compute the maximum of a vector with n numbers
m = a[0]
for i in range(1, len(a)):
    m = max(m, a[i])

# Equivalent way in Python (same complexity)
m = max(a)

Other examples:

– Reversing a vector
– Merging two sorted vectors
– Finding the largest null segment of a sorted vector: a linear-time algorithm exists (a null segment is a compact sub-vector in which the sum of all the elements is zero)

Logarithmic time: $O(\log n)$

• Logarithmic time is usually related to divide-and-conquer algorithms
• Examples:
  – Binary search
  – Calculating $x^n$
  – Calculating the $n$-th Fibonacci number

Example: recursive $x^y$

def power(x: int, y: int) -> int:
    """Returns $x^y$. Pre: $x \neq 0, y \geq 0$""
    if y == 0: return 1
    if y%2 == 0: return power(x*x, y//2);
    return x*power(x*x, y//2);

    # Assumption: each */% takes $O(1)$

$$T(x^y) \leq 4 + T((x^2)^{y/2}) \leq 4 + 4 + T((x^4)^{y/4}) \leq \cdots$$
$$T(x^y) \leq 4 + 4 + \cdots + 4$$
$$\text{log}_2 y \text{ times}$$

$$\implies O(\log y)$$
Linearithmic time: $O(n \log n)$

- **Sorting**: Merge sort and heap sort can be executed in $O(n \log n)$.

- **Largest empty interval**: Given $n$ time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?
  
  – $O(n \log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.