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

```c
// Pre: n >= 0
// Returns the Fibonacci number of order n.
int fib(int n) {
    // Recursive solution
    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,
\[
a^n = a^{n-1} + a^{n-2} \quad \Rightarrow \quad a^2 = a + 1
\]
\[
a = \frac{1 + \sqrt{5}}{2} = \varphi \approx 1.618 \quad \text{(golden ratio)}
\]

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

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

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

Fibonacci numbers: iterative version

```c
// Pre: n >= 0
// Returns the Fibonacci number of order n.
int fib(int n) {
    // iterative solution
    int f_i = 0;
    int f_i1 = 1;
    // Inv: f_i is the Fibonacci number of order i.
    //      f_i1 is the Fibonacci number of order i+1.
    for (int i = 0; i < n; ++i) {
        int f = f_i + f_i1;
        f_i = f_i1;
        f_i1 = f;
    }
    return f_i;
}
```

Runtime: \( n \) iterations

Fibonacci numbers

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

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

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

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

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

\[
A^1 = \begin{bmatrix} 1 & 1 \\
1 & 0
\end{bmatrix} \quad A^2 = \begin{bmatrix} 2 & 1 \\
1 & 1
\end{bmatrix}
\]
\[
A^4 = \begin{bmatrix} 5 & 3 \\
3 & 2
\end{bmatrix} \quad A^8 = \begin{bmatrix} 34 & 21 \\
21 & 13
\end{bmatrix}
\]
\[
A^{16} = \begin{bmatrix} 1597 & 987 \\
987 & 610
\end{bmatrix} \quad \ldots \quad A^n = \begin{bmatrix} F_{n+1} & F_n \\
F_n & F_{n-1}
\end{bmatrix}
\]
Algorithm analysis

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

$$C : D \rightarrow \mathbb{R}^+$$
$$x \mapsto C(x)$$

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

• 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.

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 $n$.

Asymptotic notation

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

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

Examples for Big-O and Big-Ω

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>quasi-linear</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>linear</td>
</tr>
<tr>
<td>$\log n$</td>
<td>root - $n$</td>
</tr>
<tr>
<td>$1$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$\subseteq O(n^3)$</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>$\subseteq O(n)$</td>
</tr>
<tr>
<td>$g(n) \in O(f_1(n))$</td>
<td>$g(n) \in \Omega(f_2(n))$</td>
</tr>
</tbody>
</table>
The limit rule

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

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

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

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

All rules (except the last one) also apply for $\Omega$ and $\Theta$:

- $f \in O(f)$
- $\forall c > 0, O(f) = O(c \cdot f)$
- $f \in O(g) \land g \in O(h) \Rightarrow f \in O(h)$
- $f_1 \in O(g_1) \land f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2) = O(\max\{g_1, g_2\})$
- $f \in O(g) \Rightarrow f + g \in O(g)$
- $f_1 \in O(g_1) \land f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$
Let us consider that every operation can be executed in 1 ns ($10^{-9}$ s).

<table>
<thead>
<tr>
<th>Function</th>
<th>$n = 1,000$</th>
<th>$n = 10,000$</th>
<th>$n = 100,000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_2 n$</td>
<td>10 ns</td>
<td>13.3 ns</td>
<td>16.6 ns</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>31.6 ns</td>
<td>100 ns</td>
<td>316 ns</td>
</tr>
<tr>
<td>$n$</td>
<td>1 μs</td>
<td>10 μs</td>
<td>100 μs</td>
</tr>
<tr>
<td>$n \log_2 n$</td>
<td>10 μs</td>
<td>133 μs</td>
<td>1.7 ms</td>
</tr>
<tr>
<td>$n^2$</td>
<td>1 ms</td>
<td>100 ms</td>
<td>10 s</td>
</tr>
<tr>
<td>$n^3$</td>
<td>1 s</td>
<td>16.7 min</td>
<td>11.6 days</td>
</tr>
<tr>
<td>$2^n$</td>
<td>3.4 · $10^{284}$ yr</td>
<td>6.3 · $10^{2993}$ yr</td>
<td>3.2 · $10^{30086}$ yr</td>
</tr>
</tbody>
</table>

How about “big data”?

This is often the practical limit for big data.

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?

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

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

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 \( d = 2^n \)):
\[ T(d) = d + 2 \sum_{i=0}^{n-1} 2^i = d + 2(2^n - 1) = 3d - 2 = O(d) \]

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 \( O(1) \).
Runtime analysis rules

- 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 $O(\cdot)$, $S_2$ is $O(\cdot)$ and $B$ is $O(\cdot)$, then if ($B$) $S_1$; else $S_2$; is $O(\max\{f+h, g+h\})$, or also $O(\max\{f, g, h\})$.

Algorithm Analysis

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

---

Nested loops: examples

```cpp
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        DoSomething(); // O(1)  \Rightarrow O(n^2)
```

```cpp
for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
        DoSomething(); // O(1)  \Rightarrow O(n^2)
```

```cpp
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        for (int k = 0; k < p; ++k)
            DoSomething(); // O(1)  \Rightarrow O(n \cdot m \cdot p)
```

---

Linear time: $O(n)$

Running time proportional to input size

```cpp
int m = a[0];
for (int i = 1; i < a.size(); ++i) {
    if (a[i] > m) m = a[i];
}
```
**Linear time: \(O(n)\)**

*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 \(x^y\)**

```c
// Pre: x ≠ 0, y ≥ 0
// Returns \(x^y\)
int power(int x, int y) {
    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 \(O(1)\)

\[
T(x^y) \leq 4 + T((x^2)^{y/2}) \leq 4 + 4 + T((x^4)^{y/4}) \leq \cdots
\]

\[
T(x^y) \leq 4 + 4 + \cdots + 4 \quad \Rightarrow \quad O(\log y)
\]

\[
\underbrace{\log_2 y \ times}_{\text{times}}
\]

**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 \(x_1, \cdots, x_n\) 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.
Selection Sort

- Selection sort uses this invariant:

```
void selection_sort(vector<elem>& v) {
    int last = v.size() - 1; // v.size() = n
    for (int i = 0; i < last; ++i) {
        int k = i;
        for (int j = i + 1; j <= last; ++j) { // i+1..n-1
            if (v[j] < v[k]) k = j;
        }
        swap(v[k], v[i]);
    }
}
```

```
T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} O(1) = O(1) \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = O(1) \sum_{i=0}^{n-2} (n - i - 1)
= O(1) \left((n - 1)^2 - \frac{1}{2}(n - 1) \cdot (n - 2)\right)
= O(1) \cdot O(n^2) = O(n^2)
```

Insertion Sort

- Let us use inductive reasoning:
  - If we know how to sort arrays of size n-1,
  - do we know how to sort arrays of size n?

```
void insertion_sort(vector<elem>& v) {
    for (int i = 1; i < v.size(); ++i) { // n-1 times
        elem x = v[i];
        int j = i;
        while (j > 0 and v[j - 1] > x) { // 0..i times
            v[j] = v[j - 1];
            --j;
        }
        v[j] = x;
    }
}
```

```
T(n) = \Omega(n)
T(n) = O(n^2)
```

```
T_{\text{worst}}(n) = \sum_{i=1}^{n-1} i \cdot O(1) = O(n^2) \quad \Rightarrow \text{sorted in reverse order}
```

```
T_{\text{best}}(n) = \sum_{i=1}^{n-1} O(1) = O(n) \quad \Rightarrow \text{already sorted}
```
The Maximum Subsequence Sum Problem

• Given (possibly negative) integers $A_1, A_2, \ldots, A_n$, find the maximum value of $\sum_{k=i}^{j} A_k$.

  (the max subsequence sum is 0 if all integers are negative).

• Example:
  – Input: -2, 11, -4, 13, -5, -2
  – Answer: 20 (subsequence 11, -4, 13)


\[
T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1
\]

\[
= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)
\]

\[
= \sum_{i=0}^{n-1} \frac{(n - i + 1)(n - i)}{2} = \ldots
\]

\[
= \frac{n^3 + 3n^2 + 2n}{6} = O(n^3)
\]

\[
\text{int maxSubSum(const vector<int>& a) \{}
\text{    int maxSum = 0;}
\text{    // try all possible subsequences}
\text{    for (int i = 0; i < a.size(); ++i) \{}
\text{        int thisSum = 0;}
\text{        for (int j = i; j < a.size(); ++j) \{}
\text{            if (thisSum > maxSum) maxSum = thisSum;
\text{            thisSum += a[j];}
\text{        }\}
\text{    }\}
\text{    return maxSum;}
\text{\}}}
\]

\[
T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = O(n^2)
\]
Max Subsequence Sum: Divide&Conquer

First half  Second half
4 -3 5 -2 -1 2 6 -2

The max sum can be in one of three places:
- 1\textsuperscript{st} half
- 2\textsuperscript{nd} half
- Spanning both halves and crossing the middle

In the 3\textsuperscript{rd} case, two max subsequences must be found starting from the center of the vector (one to the left and the other to the right)

```cpp
int maxSumRec(const vector<int>& a, int left, int right) {
  // base cases
  if (left == right)
    if (a[left] > 0)
      return a[left];
    else
      return 0;
  // Recursive cases: left and right halves
  int center = (left + right)/2;
  int maxLeft = maxSumRec(a, left, center);
  int maxRight = maxSumRec(a, center + 1, right);
  int maxRCenter = 0, rightSum = 0;
  for (int i = center; i >= left; --i) {
    rightSum += a[i];
    if (rightSum > maxRCenter)
      maxRCenter = rightSum;
  }
  int maxLCenter = 0, leftSum = 0;
  for (int i = center + 1; i <= right; ++i) {
    leftSum += a[i];
    if (leftSum > maxLCenter)
      maxLCenter = leftSum;
  }
  int maxCenter = maxRCenter + maxLCenter;
  return max3(maxLeft, maxRight, maxCenter);
}
```

We will see how to solve this equation formally in the next lesson (Master Theorem). Informally:

\[
T(1) = 1 \\
T(n) = 2T(n/2) + O(n)
\]

When \(n = 2^k\) we have that \(k = \log_2 n\)

\[
T(n) = 2^k T(1) + kn = n + n \log_2 n = O(n \log n)
\]

But, can we still do it faster?

The Maximum Subsequence Sum Problem

- Observations:
  - If \( a[i] \) is negative, it cannot be the start of the optimal subsequence.
  - Any negative subsequence cannot be the prefix of the optimal subsequence.

- Let us consider the inner loop of the \( O(n^2) \) algorithm and assume that \( a[i..j-1] \) is positive and \( a[i..j] \) is negative:
  - If \( p \) is an index between \( i+1 \) and \( j \), then any subsequence from \( a[p] \) is not larger than any subsequence from \( a[i] \) and including \( a[p-1] \).
  - If \( a[j] \) makes the current subsequence negative, we can advance \( i \) to \( j+1 \).

```cpp
int maxSubSum(const vector<int>& a) {
    int maxSum = 0, thisSum = 0;
    for (int i = 0; i < a.size(); ++i) {
        thisSum += a[i];
        if (thisSum > maxSum) maxSum = thisSum;
        else if (thisSum < 0) thisSum = 0;
    }
    return maxSum;
}
```

Algorithm Analysis © Dept. CS, UPC

Problems on polygons

Given a set of \( n \) points in the plane, connect them in a simple closed path.

Compute the convex hull of \( n \) given points in the plane.
Simple polygon

- **Input:** \( p_1, p_2, \ldots, p_n \) (points in the plane).
- **Output:** \( P \) (a polygon whose vertices are \( p_1, p_2, \ldots, p_n \) in some order).

Select a point \( z \) with the largest \( x \) coordinate (and smallest \( y \) in case of a tie in the \( x \) coordinate). Assume \( z = p_1 \).

- For each \( p_i \in \{ p_2, \ldots, p_n \} \), calculate the angle \( \alpha_i \) between the lines \( z - p_i \) and the \( x \) axis.

Sort the points \( \{ p_2, \ldots, p_n \} \) according to their angles. In case of a tie, use distance to \( z \).

**Complexity:** \( O(n \log n) \).
The runtime is dominated by the sorting algorithm.

Convex hull: gift wrapping algorithm

- **Input:** \( p_1, p_2, \ldots, p_n \) (points in the plane).
- **Output:** \( P \) (the convex hull of \( p_1, p_2, \ldots, p_n \)).

Initial point: \( z \) with the largest \( x \) coordinate (and smallest \( y \) in case of a tie in the \( x \) coordinate).

Iteration: Assume that a partial path with \( k \) points has been built (\( p_k \) is the last point). For each remaining point \( q \) calculate the angle of \( p_k - q \) with the \( x \) axis and pick the smallest one (in counter-clockwise fashion).

- Stop when \( P \) is complete (back to point \( z \)).

Complexity: At each iteration \( k \), we calculate \( n - k \) angles. In the worst case, all points may belong to the convex hull, thus \( T(n) = O(n^2) \).

Convex hull: Graham Scan

- **Input:** \( p_1, p_2, \ldots, p_n \) (points in the plane).
- **Output:** \( P \) (the convex hull of \( p_1, p_2, \ldots, p_n \)).

Intuition: every three consecutive vertices in the convex hull must be in a **counter-clockwise** turn.

```cpp
bool ccw(p1, p2, p3) { 
  return (p2.x - p1.x) \cdot (p3.y - p1.y) > 
  (p2.y - p1.y) \cdot (p3.x - p1.x); 
}
```

https://en.wikipedia.org/wiki/Graham_scan
Convex hull: Graham scan

Input: $p_1, p_2, \ldots, p_n$ (points in the plane).

Output: $q_1, q_2, \ldots, q_m$ (the convex hull).

Initially:
Create a simple polygon $P$ (complexity $O(n \log n)$).
Assume the order of the points is $p_1, p_2, \ldots, p_n$.

Algorithm Analysis

// $Q = (q_1, q_2, \ldots)$ is a vector where the points // of the convex hull will be stored.
q_1 = p_1; q_2 = p_2; q_3 = p_3; m = 3;
for k = 4 to n:
  while not ccw(q_{m-1}, q_m, p_k):
    m = m - 1;
    m = m + 1;
q_m = p_k;

Observation: each point $p_k$ can be included in $Q$ and deleted at most once.
The main loop of Graham scan has linear cost.

Complexity: dominated by the creation of the simple polygon $\Rightarrow O(n \log n)$. 

© Dept. CS, UPC