# Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

# Analysis of Algorithms

- **Correctness**.
  - *Hoare logic*
  - *Termination theorem*
- **Complexity**
  - Time complexity
  - *Space complexity*
  - Worst-Case, Best-Case and Average-Case Complexity
  - Amortized complexity

# Correctness of an Algorithm

**Definition**

On <u>any</u> input, the algorithm returns the desired output.

Correctness $\neq$ No one has found a counter-example!

- Proof of correctness (informal): a set of properties that assert what the content of some variables should be at any step of the algorithm.

<u>Remark</u>: Proving that a property still holds from one step to another ("Invariant") is straightforward if the next instruction is just an elementary operation (assignment, arithmetic,...) or a conditional operator ("`if/else`", ternary operator, etc.).

Proof of correctness in more complex situations:

- **by induction**: suitable to iterative programs (`while/for` loops)
- **by backward induction**: suitable to recursive programs.

# Iterative algorithms

## Is the following program correct for any integer input?

```
int maximum(const vector<int>& a) {
    int m = -1;
    for(int i = 0; i < a.size(); i++) {
        if(a[i] > m)
            m = a[i];
    }
    return m;
}
```

- Execution for $a = \{100, 10, 999, 3\}$

$m = -1 \longrightarrow m = 100 \longrightarrow m = 100 \longrightarrow m = 999 \longrightarrow m = 999$

# Iterative algorithms

## Is the following program correct for any integer input?

```
int maximum(const vector<int>& a) {
    int m = -1;
    for(int i = 0; i < a.size(); i++) {
        if(a[i] > m)
            m = a[i];
    }
    return m;
}
```

- Execution for $a = \{100, 10, 999, 3\}$

$m = -1 \longrightarrow m = 100 \longrightarrow m = 100 \longrightarrow m = 999 \longrightarrow m = 999$

- But for $a = \{-2\}$ ?

# Hoare logic

**Definition**

property $P(i)$ holds <u>before</u> loop $i \longrightarrow^{\text{loop } i}$ property $P(i+1)$ holds <u>after</u> loop $i$.

```
int maximum(const vector<int>& a) {
    int m = a[0];
    /*P(i): before loop i we have m = max_{0≤j<i} a[j]*/
    for(int i = 1; i < a.size(); i++) {
        if(a[i] > m)
            m = a[i];
    }
    return m;
}
```

## Example: sorting

```
void sort(vector<int>& a) {
    /*P(i): the subvector a[i ... n−1] is sorted
    P(n) is true (empty vector) */
    for(int i = a.size() − 1; i ≥ 0; i −−) {
        int j = i;
        /*P(i, k): a[j] = max{a[i]} ∪ {a[0], a[1], ..., a[k − 1]} */
        for(int k = 0; k < i; k + +) {
            if(a[k] > a[j])
                j = k;
        }
        int m = a[j];
        a[j] = a[k];
        a[k] = m;
    }
}
```

Remark: uses maximum computation as a *subroutine*!

Identification and re-use of algorithms for intermediate problems can help in simplifying the code *and* the proof of its correctness.

# Recursive algorithms

**Theorem (Termination theorem)**

*An algorithm is correct if, for some $\mathcal{B} \subseteq \mathbb{N}$,*

- *it is correct on inputs of size n, for every $n \in \mathcal{B}$.*
- *it is correct assuming that all recursive calls are correct, and all recursive calls are for sub-inputs of size $n'$, $d(n', \mathcal{B}) < d(n, \mathcal{B})$.*

Example: $\mathcal{B} = \{0\}$ and $n' = n - 1$.

```
int factorial(int n) {
   if(n == 0) {
      return 1;  B = {0}
   } else return n*factorial(n-1);  n' = n - 1
}
```

## McCarthy's function

### What is the output?

**Theorem**

$$M(n) = \begin{cases} 91 \ \textit{if } n \leq 101 \\ n - 10 \ \textit{otherwise.} \end{cases}$$

Can be proved using the Termination theorem.

```
int M(int n) {
    if(n > 100)
        return n - 10;  //B = {101, 102, ...} = [101; +∞)
    else
        return M(M(n + 11));  n' = n + 11 > n
}
```

Case $n' \geq 91$. We have $M(n') = n' - 10 = n + 1$. Since $n'' = n + 1 > n$, $M(n+1) = 91$.

Case $n' < 91$. We have $M(n') = 91$. Furthermore, $M(91) = M(M(102)) = M(92) = M(M(103)) = M(93) = \ldots = M(100) = M(M(111)) = M(101) = 91$.

# Is an algorithm efficient?

**Definition (Complexity – Informal)**

A rigorous way to decide whether an algorithm is "better" than another.

What does "better" mean?

- running-time $\Longrightarrow$ Time Complexity
- space usage $\Longrightarrow$ Space Complexity
- number of processors $\Longrightarrow$ Parallel Complexity
- ...

One often needs to find a trade-off between all these criteria...

# Complexity cont'd

## Running-time Evaluation

First try: in seconds ?

# Complexity cont'd

## Running-time Evaluation

First try: in seconds ? $\Longrightarrow$ Machine-dependent!

# Complexity cont'd

## Running-time Evaluation

First try: in seconds ? $\Longrightarrow$ Machine-dependent!

**Number of elementary operations**

- Arithmetic (Addition, Soustraction,...)
- Comparison, ...

# Complexity cont'd

## Running-time Evaluation

First try: in seconds ? $\Longrightarrow$ Machine-dependent!

**Number of elementary operations**

- Arithmetic (Addition, Soustraction,...)
- Comparison, ...

Can vary with the data types and the machine, but these variations can be neglected.

# Complexity cont'd

## Running-time Evaluation

First try: in seconds ? $\Longrightarrow$ Machine-dependent!

**Order of magnitude for Number of elementary operations**

- Arithmetic (Addition, Soustraction,...)
- Comparison, ...

Can vary with the data types and the machine, but these variations can be neglected.

# The "Big-Oh" notation

- "Big-Oh" (Worst-Case, Upper bound)

$$f(n) = \mathcal{O}(g(n)) \iff \exists c \text{ s.t. } \forall n, f(n) \leq c \cdot g(n).$$

- "Big-Omega" (Best-Case, Lower bound)

$$f(n) = \Omega(g(n)) \iff g(n) = \mathcal{O}(f(n)).$$

- "Big-Theta" (Exact)

$$f(n) = \Theta(g(n)) \iff f(n) = \mathcal{O}(g(n)) \text{ and } g(n) = \mathcal{O}(f(n)).$$

# Basics of Complexity

- Constant: $f(n) = \mathcal{O}(1)$
- Logarithmic: $f(n) = \mathcal{O}(\log n)$ (for any base)
- Linear: $f(n) = \mathcal{O}(n)$
- "Quasi Linear": $f(n) = \mathcal{O}(n \log n)$
- Quadratic: $f(n) = \mathcal{O}(n^2)$
- Cubic: $f(n) = \mathcal{O}(n^3)$
- Polynomial: $f(n) = \mathcal{O}(n^c)$ for some $c > 0$
- Exponential: $f(n) = \mathcal{O}(2^n)$.

# Basics of Complexity cont'd

Worst-case complexity + Bounded time computation $\Longrightarrow$ a maximum size for the inputs

**Table 2.1** The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

| | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $1.5^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 4 sec |
| $n = 30$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 18 min | $10^{25}$ years |
| $n = 50$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 11 min | 36 years | very long |
| $n = 100$ | < 1 sec | < 1 sec | < 1 sec | 1 sec | 12,892 years | $10^{17}$ years | very long |
| $n = 1{,}000$ | < 1 sec | < 1 sec | 1 sec | 18 min | very long | very long | very long |
| $n = 10{,}000$ | < 1 sec | < 1 sec | 2 min | 12 days | very long | very long | very long |
| $n = 100{,}000$ | < 1 sec | 2 sec | 3 hours | 32 years | very long | very long | very long |
| $n = 1{,}000{,}000$ | 1 sec | 20 sec | 12 days | 31,710 years | very long | very long | very long |

# Example #1: complexity of computing the maximum

```
int maximum(const vector<int>& a) {
    int m = a[0];
    for(int i = 1; i < a.size(); i++) {
        if(a[i] > m)
            m = a[i];
    }
    return m;
}
```

- Complexity?

# Example #1: complexity of computing the maximum

```
int maximum(const vector<int>& a) {
    int m = a[0];
    for(int i = 1; i < a.size(); i++) {
        if(a[i] > m)
            m = a[i];
    }
    return m;
}
```

- Complexity? Linear in $n = a.size()$

Short justification: there are $n$ loop iterations and for each iteration we only perform $\mathcal{O}(1)$ elementary operations.

# Example #2: complexity of sorting

```
void sort(vector<int>& a) {
    for(int i = a.size() − 1; i ≥ 0; i − −) {
        int j = i;
        for(int k = 0; k < i; k + +) {
            if(a[k] > a[j])
                j = k;
        }
        int m = a[j];
        a[j] = a[k];
        a[k] = m;
    }
}
```

- Complexity?

# Example #2: complexity of sorting

```
void sort(vector<int>& a) {
    for(int i = a.size() - 1; i >= 0; i--) {
        int j = i;
        for(int k = 0; k < i; k++) {
            if(a[k] > a[j])
                j = k;
        }
        int m = a[j];
        a[j] = a[k];
        a[k] = m;
    }
}
```

- Complexity? Quadratic in $n = a.size()$.

Short justification: $n$ calls to `maximum()`.

$i^{\text{th}}$ call on size-$(n-i)$ subvector... but $\sum(n-i) = \sum i = \Theta(n^2)$

# Example #2: complexity of sorting

```
void sort(vector<int>& a) {
    for(int i = a.size() - 1; i >= 0; i--) {
        int j = i;
        for(int k = 0; k < i; k++) {
            if(a[k] > a[j])
                j = k;
        }
        int m = a[j];
        a[j] = a[k];
        a[k] = m;
    }
}
```

- Complexity? Quadratic in $n = a.size()$.

Short justification: $n$ calls to `maximum()`.

$i^{\text{th}}$ call on size-$(n-i)$ subvector... but $\sum(n-i) = \sum i = \Theta(n^2)$

*Is it optimal? (Spoiler: NO)*

# Algorithms vs. Data Structures

- Time complexity for an algorithm: a function that associates, to each possible input size $n$, the lonfest possible runtime $T(n)$.

- For a data structure we have:
  - **Pre-processing time.** Initialization of the data structure.

    Can be non-constant, *e.g.*, if the set/number of data inputs is fixed in advance (like we are doing for an array)

  - **Query time.** Complexity of the algorithm for answering a query.
    Different types of queries may have different query times.

$\longrightarrow$ Trade-off between pre-processing time and query time(s).

# Alternative Complexity measures: **Space Complexity**

**Definition**

Memory usage for executing the code, leaving asides the storage of the input (= Work Space Complexity)

Examples:

- Use of an auxiliary array: $\mathcal{O}(n)$
- Use of an auxiliary counter: $\mathcal{O}(1)$

$\longrightarrow$ For items of limited memory (*e.g.*, cell phones), it is preferable to have constant memory usage: **In-place algorithms**.

$\longrightarrow$ For Data Structures, Space complexity indicates how much more space we need than just the space needed for storing the data (which is $\mathcal{O}(n)$ for $n$ elements)

# Alternative Complexity measures: **Parallel Complexity**

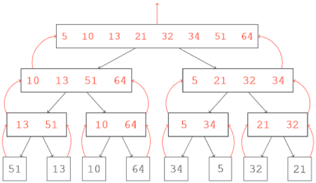
**Definition (Informal)**

Whether the operations can be "splitted" to be executed by different (independent) processors.

Example: **Recursive** sorting algorithms

Many different recursive calls, to disjoint subproblems, can be executed independently to each other.

*Merge Sort, Quick Sort, etc...*




Tight relation with Space Complexity.

# Beyond Worst-Case Complexity

We defined Time Complexity of an algorithm as the longest running time $T(n)$ on inputs of size $n$.

- *Best-case Complexity*

$=$ What are the easy instances?

Example: incrementation of a $k$-bit counter.

Worst-Case $\mathcal{O}(k)$: $\texttt{01111...1} +1 \longrightarrow \texttt{10000...0}$

Best-Case $\mathcal{O}(1)$: $\texttt{00000...0} +1 \longrightarrow \texttt{00000...1}$

- *Adaptive algorithms*: Is my algorithm faster when the input is "close" to the easy instances?

Example: Is an algorithm faster on almost sorted instances? On arrays sorted by non-increasing value?

# Average-Case Complexity

Sometimes, only a handful of instances make the worst-case complexity increase, whereas it is much lower for all other inputs (ex.: Quicksort).

1) Consider a probability distribution $\pi$ over all inputs of size $n$ (usually the uniform distribution).

2) The complexity is now a random variable.

$$Pr[\mathcal{A} \text{ runs in } k \text{ steps}] = \sum \{\pi(x) \mid \mathcal{A}(x) \text{ runs in } k \text{ steps}\}$$

3) Average complexity = expectation

$$= \sum_{k \geq 0} k \cdot Pr[\mathcal{A} \text{ runs in } k \text{ steps}]$$

# Example

## Increment of a $k$-bit counter

1) Requires $k' \leq k$ operations iff the lowest-order bits consist of 1 zero followed by $k' - 1$ ones.

Ex: Three operations required for $\dots 011$

2) There are $2^{k-k'}$ possible inputs for which we require $k'$ operations (*i.e.*, just fill in arbitrarily the highest-order bits).

3) Complexity: $\sum_{k'=1}^{k} k' \cdot 2^{k-k'} = 2^k \cdot \sum_{k'=1}^{k} \frac{k'}{2^{k'}} \sim_{+\infty} \frac{k}{\ln 2}$

# Amortized Complexity

- We expect a data structure to answer many queries (not just one).

- Sometimes, the worst-case complexity of answering a query may be large *only* because of past operations.

**Definition (Amortized complexity)**

$\sup_{m \geq 0}\{\frac{1}{m} \cdot (\text{Worst-Case complexity for answering to } m \text{ queries})\}$

Observation: We always have Amortized Complexity $\leq$ Worst-Case Complexity

# The Potential Method

- A **potential** is a function $\Phi$ that associates to a data structure $\mathcal{D}$ a *non-negative number* $\Phi(\mathcal{D})$.
  - Often depends on the size $n$.
  - For an empty data structure, we further impose $\Phi(\mathcal{D}) = 0$.

1) Consider various types of queries $q_1(), q_2(), \ldots, q_r()$.

2) We denote their respective complexities by $T_1, T_2, \ldots, T_r$.

3) Let $\Delta\Phi_1, \Delta\Phi_2, \ldots, \Delta\Phi_r$ be the resulting changes of potential.

**Amortized complexity of operation** $q_j = \max\{T_j + \Delta_j\}$.

Interpretation: Fast operations are overestimated (ton increase the potential), whereas slower operations are compensated (by a decrease in potential).

# Example

- Increment of a $k$-bit counter (initially set to 0).

**Theorem**

*Amortized complexity is in $\mathcal{O}(1)$*

- **Proof**: Potential function $\Phi = \#$ bits set to 1

Observation: Average-Case Complexity $\neq$ Amortized Complexity

# Complexity in practice

- Made difficult by using existing codes/libraries

(Complexity is poorly documented!)

- The "right" complexity measure may depend on the context of the application (*e.g.*, offline vs distributed)

- Good practice consists in counting the number of calls to each subroutine (+ size of inputs).

# Questions