Computational complexity 1

How to measure the efficiency of an algorithm

Background

- What is an Algorithm?
- A problem can be solved by many algorithms
  - E.g., sorting data
- An algorithm is a method or a process followed to solve a problem
  - I.e., a recipe
- An algorithm takes the input to a problem (function) and transforms it to the output
  - I.e., a mapping
Why consider Efficiency?

- There are often many algorithms to solve a problem
- How do we choose between them? (Usually) conflicting goals:
  - To design an algorithm that is easy to understand, code, debug
    - software engineering
  - To design an algorithm that makes efficient use of the computer’s resources
    - data structures and algorithm analysis

(Un programma che funziona) in fretta vs. Un programma che (funziona in fretta)
Why consider Efficiency?

- There are often many algorithms to solve a problem.
- How do we choose between them? (Usually conflicting goals):
  - To design an algorithm that is easy to understand, code, debug.
    - software engineering
  - To design an algorithm that makes efficient use of the computer’s resources.
    - data structures and algorithm analysis

from the LinkedList javadoc page:
“All of the operations perform as could be expected for a doubly-linked list.”

How to Measure Efficiency?

- Critical resources
  - programmer’s effort
  - time, space (disk, RAM)
- Analysis
  - empirical (run programs)
  - analytical (asymptotic algorithm analysis)
- Worst case vs. Average case
Empirical Approach

- Implement each candidate
  - That could be lots of work – also error-prone
- Run it
  - Which inputs?
  - Worst case, average case, or best case?
- Time it
  - What machines
  - Which OS?

Analytical Approach

- How to solve “which algorithm” problems without machines nor test data?
Analytical Approach

- An algorithm is a mapping
- For most algorithms, running time depends on "size" of the input
- Running time is expressed as $T(n)$
  - some function $T$
  - input size $n$

Bubble sort

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>unsorted</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6 &gt; 1, swap</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6 &gt; 2, swap</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6 &gt; 3, swap</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6 &gt; 4, swap</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>6 &gt; 5, swap</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1 &lt; 2, ok</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2 &lt; 3, ok</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3 &lt; 4, ok</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4 &lt; 5, ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sorted</td>
</tr>
</tbody>
</table>
The bubble sort takes $(n^2-n)/2$ “steps”

Different implementations/assembly languages

- Program A on an Intel Pentium IV: $T(n) = 58(n^2-n)/2$
- Program B on a Motorola: $T(n) = 84(n^2-2n)/2$
- Program C on an Intel Pentium V: $T(n) = 44(n^2-n)/2$

Note that each has an $n^2$ term

as $n$ increases, the other terms will drop out

As a result:

- Program A on Intel Pentium IV: $T(n) \approx 29n^2$
- Program B on Motorola: $T(n) \approx 42n^2$
- Program C on Intel Pentium V: $T(n) \approx 22n^2$
Analysis

- As processors change, the constants will always change
  - The exponent on \( n \) will not
  - We should not care about the constants

As a result:
- Program A: \( T(n) \approx n^2 \)
- Program B: \( T(n) \approx n^2 \)
- Program C: \( T(n) \approx n^2 \)

- Bubble sort: \( T(n) \approx n^2 \)

Intuitive motivations

- Asymptotic notation captures behavior of functions for large values of \( x \).
- Dominant term of \( 3x^3 + 5x^2 - 9 \) is \( 3x^3 \)
- As \( x \) becomes larger and larger, other terms become insignificant and only \( 3x^3 \) remains in the picture
\[ y = 3x^3 + 5x^2 - 9 \]

\[ y = x^3 \]

\[ y = x^2 \]

\[ y = x \]
\[ y = 3x^3 + 5x^2 - 9 \]

\[ y = 5x^3 \]
**Complexity Analysis**

- **O(·)**
  - big o (big oh)
- **Ω(·)**
  - big omega
- **Θ(·)**
  - big theta

**O(·)**

- Upper Bounding Running Time
- Why?
  - Little-oH
  - "Order of"
  - D’Oh
Upper Bounding Running Time

- $f(n)$ is $O(g(n))$ if $f$ grows “at most as fast as” $g$

Big-O (formal)

- Let $f$ and $g$ be two functions such that $f: \mathbb{N} \to \mathbb{R}^+$ and $g: \mathbb{N} \to \mathbb{R}^+$

- if there exists positive constants $c$ and $n_0$ such that $f(x) \leq cg(x)$ for all $n > n_0$

- then we can write $f \sim O(g)$
Big-O (formal alt)

- Let $f$ and $g$ be two functions such that
  \[ f, g : \mathbb{N} \to \mathbb{R}^+ \]

- if there exists positive constants $c$ and $n_0$ such that
  \[ 0 \leq \lim_{n \to \infty} \frac{f}{g} = c < \infty \]

- then we can write
  \[ f \sim g = O(n) \]

Example

- \((\log n)^2 = O(n)\)

- \(f(n) = (\log n)^2\)
  \(g(n) = n\)

- \((\log n)^2 \leq n\) for all \(n \geq 16\), so \((\log n)^2\) is \(O(n)\)
Notational Issues

- Big-O notation is a way of comparing functions.
- Notation quite unconventional.
  - e.g., $3x^3 + 5x^2 - 9 = O(x^3)$
- Doesn’t mean
  - “$3x^3 + 5x^2 - 9$ equals the function $O(x^3)$”
  - “$3x^3 + 5x^2 - 9$ is big oh of $x^3$”
- But
  - “$3x^3 + 5x^2 - 9$ is dominated by $x^3$”

Common Misunderstanding

- $3x^3 + 5x^2 - 9 = O(x^3)$
- However, also true are:
  - $3x^3 + 5x^2 - 9 = O(x^4)$
  - $x^3 = O(3x^3 + 5x^2 - 9)$
  - $\sin(x) = O(x^4)$
- Note:
  - Usage of big-O typically involves mentioning only the most dominant term
  - “The running time is $O(x^{2.5})$”
Lower Bounding Running Time

- $f(n)$ is $\Omega(g(n))$ if $f$ grows “at least as fast as” $g$

- $cg(n)$ is an approximation to $f(n)$ bounding from below

Big-Omega (formal)

- Let $f$ and $g$ be two functions such that
  
  \[ f \colon N \rightarrow R^+ \text{ and } g \colon N \rightarrow R^+ \]

- if there exists positive constants $c$ and $n_0$ such that
  
  \[ f \geq cg \text{ for all } n > n_0 \]

- then we can write
  
  \[ f \sim \Omega(g) \]
Tightly Bounding Running Time

- \( f(n) \) is \( \Theta(g(n)) \) if \( f \) is essentially the same as \( g \), to within a constant multiple.

![Graph showing \( f(n) \), \( c_1 g(n) \), and \( c_2 g(n) \) with \( n_0 \) as a point where the functions are close to each other.]

---

Big-Theta (formal)

- Let \( f \) and \( g \) be two functions such that:
  \[
  f: \mathbb{N} \rightarrow \mathbb{R}^+ \quad \text{and} \quad g: \mathbb{N} \rightarrow \mathbb{R}^+
  \]

- If there exists positive constants \( c_1, c_2 \) and \( n_0 \) such that:
  \[
  c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all} \quad n > n_0
  \]

- Then we can write \( f \sim \Theta(g) \).
Big-Θ, Big-O, and Big-Ω

- Big-Θ: reverse of big-O. I.e.
  \[ f(x) = \Theta(g(x)) \]
  iff
  \[ g(x) = O(f(x)) \]
- so \( f(x) \) asymptotically dominates \( g(x) \)
**Big-Ω and Big-Θ**

- Big-Θ: domination in both directions. I.e.
  \[
  f(x) = \Theta(g(x))
  \]
  iff
  \[
  f(x) = O(g(x)) \land f(x) = \Omega(g(x))
  \]

**Problem**

- Order the following from smallest to largest asymptotically. Group together all functions which are big-Θ of each other:

  \[
  x + \sin x, \ln x, x + \sqrt{x}, \frac{1}{x}, 13 + \frac{1}{x}, 13 + x, e^x, x^e, x^x
  \]
  \[
  (x + \sin x)(x^{20} - 102), x \ln x, x(\ln x)^2, \log_2 x
  \]
Solution

\[ \frac{1}{x} \]
\[ 13 + \frac{1}{x} \]
\[ \ln x \quad \log_2 x \]
\[ x + \sin x, \frac{1}{x} \]
\[ x \ln x \]
\[ x(\ln x)^2 \]
\[ x^e \]
\[ (x + \sin x)(x^{20} - 102) \]
\[ e^x \]
\[ x^x \]
Practical approach

![Graph showing comparisons between different functions: Exponential, Cubic, Quadratic, Log-linear, Linear, Logarithmic.](image)

<table>
<thead>
<tr>
<th>Class</th>
<th>Complexity</th>
<th>Number of Operations and Execution Time (1 instr/µsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$n$ $10^2$ $10^3$</td>
</tr>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>1 1 µsec 1 1 µsec</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log n)$</td>
<td>3.32 3 µsec 6.64 7 µsec 9.97 10 µsec</td>
</tr>
<tr>
<td>linear</td>
<td>$O(n)$</td>
<td>10 10 µsec 10$^2$ 100 µsec 10$^3$ 1 msec</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>33.2 33 µsec 664 664 µsec 9970 10 msec</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(n^2)$</td>
<td>10$^4$ 100 µsec $10^5$ 10 msec $10^6$ 1 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(n^3)$</td>
<td>10$^5$ 1 msec $10^6$ 1 sec $10^9$ 16 min</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^n)$</td>
<td>1024 10 m sec $10^{10}$ 3.5 x $10^{17}$ yrs $10^{36}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>logarithmic</td>
<td>15.3</td>
<td>15.3</td>
<td>15.3</td>
</tr>
<tr>
<td>linear</td>
<td>$10^4$</td>
<td>$10^5$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$10^5$</td>
<td>$10^6$</td>
<td>$10^7$</td>
</tr>
<tr>
<td>quadratic</td>
<td>$10^8$</td>
<td>$10^9$</td>
<td>$10^{10}$</td>
</tr>
<tr>
<td>cubic</td>
<td>$10^{12}$</td>
<td>$10^{15}$</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>exponential</td>
<td>$10^{300}$</td>
<td>$10^{400}$</td>
<td>$10^{500}$</td>
</tr>
</tbody>
</table>
Would it be possible?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Foo</th>
<th>Bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>$O(n^2)$</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>n = 100</td>
<td>10s</td>
<td>4s</td>
</tr>
<tr>
<td>n = 1000</td>
<td>12s</td>
<td>4.5s</td>
</tr>
</tbody>
</table>

Determination of Time Complexity

- Because of the approximations available through Big-Oh, the actual $T(n)$ of an algorithm is not calculated
  - $T(n)$ may be determined empirically
  - Big-Oh is usually determined by application of some simple 5 rules
Rule #1

- Simple program statements are assumed to take a constant amount of time which is $O(1)$

Rule #2

- Differences in execution time of simple statements is ignored
Rule #3

> In conditional statements the worst case is always used

Rule #4 – the “sum” rule

> The running time of a sequence of steps has the order of the running time of the largest

> E.g.,
  > f(n) = O(n^2)
  > g(n) = O(n^3)
  > f(n) + g(n) = O(n^3)
Rule #5 – the “product” rule

- If two processes are constructed such that second process is repeated a number of times for each \( n \) in the first process, then \( O \) is equal to the product of the orders of magnitude for both products.

- E.g.,
  - For example, a two-dimensional array has one for loop inside another and each internal loop is executed \( n \) times for each value of the external loop.
  - The function is \( O(n^2) \)

Nested Loops

```c
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

\( O(n) \) \( O(1) \)
Nested Loops

```java
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

$O(n^2)$

Nested Loops

```java
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

$O(n)$
Nested Loops

```c
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

$O(n^2)$

Note: Running time grows with nesting rather than the length of the code
More Nested Loops

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

Sequential statements

\[
\begin{align*}
&\text{for(int } z=0; z<n; ++z) &\text{O(n)} \\
&\quad \text{zap}[z] = 0; \\
&\text{for(int } t=0; t<n; ++t) &\text{O(n^2)} \\
&\quad \text{for(int } u=t; u<n; ++u) \\
&\quad \quad \quad \quad \quad \quad \quad \text{O(n^2)} \quad \text{O(n^2)}
\end{align*}
\]

Running time: \(\max(O(n), O(n^2)) = O(n^2)\)
for(int t=0; t<n; ++t) {
    if(t%2) {
        for(int u=t; u<n; ++u) {
            ++zap;
        }
    } else {
        zap = 0;
    }
}

\[ O(n) \]

\[ O(1) \]

\[ O(n^2) \]
Tips

- Focus only on the dominant (high cost) operations and avoid a line-by-line exact analysis
- Break algorithm down into “known” pieces
- Identify relationships between pieces
  - Sequential is additive
  - Nested (loop / recursion) is multiplicative
- Drop constants
- Keep only dominant factor for each variable
Caveats

- Real time vs. complexity

- CPU time vs. RAM vs. disk
Caveats

- Real time vs. complexity
- CPU time vs. RAM vs. disk
- Worse, Average or Best Case?

Worse, Average or Best Case?

![Graph showing worse, average, and best case scenarios.](image)
Worse, Average or Best Case?

- Depends on input problem instance type

![Diagram showing input space, worse configuration, neither worse or best, and best configuration leading to worse-case, average-case, and best-case behaviors.]

Basic Asymptotic Efficiency Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>Algorithm ignores input (i.e., can’t even scan input)</td>
</tr>
</tbody>
</table>
## Basic Asymptotic Efficiency Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>Algorithm ignores input (i.e., can't even scan input)</td>
</tr>
<tr>
<td>n</td>
<td>Linear</td>
<td>Algorithm scans its input (at least)</td>
</tr>
<tr>
<td>Ign</td>
<td>Logarithmic</td>
<td>Cuts problem size by constant fraction on each iteration</td>
</tr>
</tbody>
</table>
### Basic Asymptotic Efficiency Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>Algorithm ignores input (i.e., can't even scan input)</td>
</tr>
<tr>
<td>lg n</td>
<td>Logarithmic</td>
<td>Cuts problem size by constant fraction on each iteration</td>
</tr>
<tr>
<td>n</td>
<td>Linear</td>
<td>Algorithm scans its input (at least)</td>
</tr>
<tr>
<td>n lg n</td>
<td>“n-log-n”</td>
<td>Some divide and conquer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>Algorithm ignores input (i.e., can't even scan input)</td>
</tr>
<tr>
<td>lg n</td>
<td>Logarithmic</td>
<td>Cuts problem size by constant fraction on each iteration</td>
</tr>
<tr>
<td>n</td>
<td>Linear</td>
<td>Algorithm scans its input (at least)</td>
</tr>
<tr>
<td>n lg n</td>
<td>“n-log-n”</td>
<td>Some divide and conquer</td>
</tr>
<tr>
<td>n^2</td>
<td>Quadratic</td>
<td>Loop inside loop = “nested loop”</td>
</tr>
</tbody>
</table>
## Basic Asymptotic Efficiency Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>Algorithm ignores input (i.e., can’t even scan input)</td>
</tr>
<tr>
<td>lg(n)</td>
<td>Logarithmic</td>
<td>Cuts problem size by constant fraction on each iteration</td>
</tr>
<tr>
<td>(n)</td>
<td>Linear</td>
<td>Algorithm scans its input (at least)</td>
</tr>
<tr>
<td>nl(g)</td>
<td>“n-log-n”</td>
<td>Some divide and conquer</td>
</tr>
<tr>
<td>(n^2)</td>
<td>Quadratic</td>
<td>Loop inside loop = “nested loop”</td>
</tr>
<tr>
<td>(n^3)</td>
<td>Cubic</td>
<td>Loop inside nested loop</td>
</tr>
<tr>
<td>(2^n)</td>
<td>Exponential</td>
<td>Algorithm generates all subsets of (n)-element set</td>
</tr>
</tbody>
</table>
Basic Asymptotic Efficiency Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>Algorithm ignores input (i.e., can't even scan input)</td>
</tr>
<tr>
<td>lg n</td>
<td>Logarithmic</td>
<td>Cuts problem size by constant fraction on each iteration</td>
</tr>
<tr>
<td>n</td>
<td>Linear</td>
<td>Algorithm scans its input (at least)</td>
</tr>
<tr>
<td>n lg n</td>
<td>&quot;n-log-n&quot;</td>
<td>Some divide and conquer</td>
</tr>
<tr>
<td>n^2</td>
<td>Quadratic</td>
<td>Loop inside loop = &quot;nested loop&quot;</td>
</tr>
<tr>
<td>n^3</td>
<td>Cubic</td>
<td>Loop inside nested loop</td>
</tr>
<tr>
<td>2^n</td>
<td>Exponential</td>
<td>Algorithm generates all subsets of n-element set</td>
</tr>
<tr>
<td>n!</td>
<td>Factorial</td>
<td>Algorithm generates all permutations of n-element set</td>
</tr>
</tbody>
</table>

Evaluate the complexity

- Linear search?
Evaluate the complexity

- Linear search
  - $O(n)$
- Dichotomic search

Evaluate the complexity

- Linear search
  - $O(n)$
- Dichotomic search
  - $O(\log(n))$
Basic Asymptotic Efficiency Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>Algorithm ignores input (i.e., can't even scan input)</td>
</tr>
<tr>
<td>lg n</td>
<td>Logarithmic</td>
<td>Cuts problem size by constant fraction on each iteration</td>
</tr>
<tr>
<td>n</td>
<td>Linear</td>
<td>Algorithm scans its input (at least)</td>
</tr>
<tr>
<td>n log n</td>
<td>&quot;n-log-n&quot;</td>
<td>Some divide and conquer</td>
</tr>
<tr>
<td>n^2</td>
<td>Quadratic</td>
<td>Loop inside loop = &quot;nested loop&quot;</td>
</tr>
<tr>
<td>n^3</td>
<td>Cubic</td>
<td>Loop inside nested loop</td>
</tr>
<tr>
<td>2^n</td>
<td>Exponential</td>
<td>Algorithm generates all subsets of n-element set</td>
</tr>
<tr>
<td>n!</td>
<td>Factorial</td>
<td>Algorithm generates all permutations of n-element set</td>
</tr>
</tbody>
</table>

ArrayList vs. LinkedList

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(element)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove(object)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get(index)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set(index, element)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add(index, element)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove(index)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>contains(object)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>indexOf(object)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### ArrayList vs. LinkedList

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(element)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove(object)</td>
<td>$O(n) + O(n)$</td>
<td>$O(n) + O(1)$</td>
</tr>
<tr>
<td>get(index)</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>set(index, element)</td>
<td>$O(1)$</td>
<td>$O(n) + O(1)$</td>
</tr>
<tr>
<td>add(index, element)</td>
<td>$O(1) + O(n)$</td>
<td>$O(n) + O(1)$</td>
</tr>
<tr>
<td>remove(index)</td>
<td>$O(n)$</td>
<td>$O(n) + O(1)$</td>
</tr>
<tr>
<td>contains(object)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>indexOf(object)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

---

**In theory, there is no difference between theory and practice.**
Licenza d’uso

Queste diapositive sono distribuite con licenza Creative Commons “Attribuzione - Non commerciale - Condividi allo stesso modo (CC BY-NC-SA)”

Sei libero:
- di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
- di modificare quest'opera

Alle seguenti condizioni:
- Attribuzione — Devi attribuire la paternità dell’opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l’opera.
- Non commerciale — Non puoi usare quest'opera per fini commerciali.
- Condividi allo stesso modo — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.

http://creativecommons.org/licenses/by-nc-sa/3.0/

77 Tecniche di programmazione A.A. 2013/14