Collection Family Tree

ArrayList vs. LinkedList

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(element)</code></td>
<td>IMMEDIATE</td>
<td>IMMEDIATE</td>
</tr>
<tr>
<td><code>remove(object)</code></td>
<td>SLUGGISH</td>
<td>IMMEDIATE</td>
</tr>
<tr>
<td><code>get(index)</code></td>
<td>IMMEDIATE</td>
<td>SLUGGISH</td>
</tr>
<tr>
<td><code>set(index, element)</code></td>
<td>IMMEDIATE</td>
<td>SLUGGISH</td>
</tr>
<tr>
<td><code>add(index, element)</code></td>
<td>SLUGGISH</td>
<td>SLUGGISH</td>
</tr>
<tr>
<td><code>remove(index)</code></td>
<td>SLUGGISH</td>
<td>SLUGGISH</td>
</tr>
<tr>
<td><code>contains(object)</code></td>
<td>SLUGGISH</td>
<td>SLUGGISH</td>
</tr>
<tr>
<td><code>indexOf(object)</code></td>
<td>SLUGGISH</td>
<td>SLUGGISH</td>
</tr>
</tbody>
</table>
Computational complexity

How to measure the difficulty of a problem

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
Sudoku

Problems and Algorithms

- We know the efficiency of the solution
- … but what about the difficulty of the problem?
- Different concepts
  - Algorithm complexity
  - Problem complexity
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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0 &gt; 1, swap</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0 &gt; 2, swap</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0 &gt; 3, swap</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0 &gt; 4, swap</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0 &gt; 5, swap</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1 &lt; 2, ok</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2 &lt; 3, ok</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3 &lt; 4, ok</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4 &lt; 5, ok</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5 &lt; 6, ok</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>sorted</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Analysis

- 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

Analysis

- 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
Complexity Analysis

- $O(\cdot)$
  - big o (big oh)
- $\Omega(\cdot)$
  - big omega
- $\Theta(\cdot)$
  - big theta

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” \( g \)

![Graph showing \( f(n) \) and \( c \cdot g(n) \)]

Big-O (formal)

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

- if there exists positive constants \( c \) and \( n_0 \) such that
  \[
  f(n) \leq c \cdot g(n), \quad \text{for all} \ n > n_0
  \]

- then we can write
  \[
  f(n) = O(g(n))
  \]
Big-O (formal alt)

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

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

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

Example

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

\[ \hat{f}(n) = (\log n)^2 \quad g(n) = n \]

$(\log n)^2 \leq n$ 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” $g$

\[ f(n) \text{ is } \Omega(g(n)) \text{ if } f \text{ grows “at least as fast as” } g \]

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

Big-Omega (formal)

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

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

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

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

\[ \text{time} \]
\[ c_1g(n) \]
\[ f(n) \]
\[ c_2g(n) \]
\[ n_0 \]
\[ n \]

Big-Theta (formal)

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

- if there exists positive constants \( c_1, c_2 \) and \( n_0 \) such that
  \[ c_1g(n) \leq f(n) \leq c_2g(n), \quad \text{for all} \quad n > n_0 \]

- then we can write
  \[ f(n) = \Theta(g(n)) \]
Big-$\Theta$, Big-$O$, and Big-$\Omega$

Big-$\Omega$ and Big-$\Theta$

- Big-$\Omega$: reverse of big-$O$. I.e.
  \[ f(x) = \Omega(g(x)) \]
  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

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Complexity</th>
<th>$n$</th>
<th>$10^1$</th>
<th>$10^2$</th>
<th>$10^3$</th>
<th>$10^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>1</td>
<td>1 µsec</td>
<td>1 µsec</td>
<td>1 µsec</td>
<td>1 µsec</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log n)$</td>
<td>3.32</td>
<td>6.64 µsec</td>
<td>7 µsec</td>
<td>9.97 µsec</td>
<td>10 µsec</td>
</tr>
<tr>
<td>linear</td>
<td>$O(n)$</td>
<td>10</td>
<td>10 µsec</td>
<td>100 µsec</td>
<td>1000 µsec</td>
<td>10 sec</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>33.2</td>
<td>664 µsec</td>
<td>6640 µsec</td>
<td>66400 µsec</td>
<td>10 min</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(n^2)$</td>
<td>$10^2$</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$10^8$</td>
<td>$10^{10}$</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(n^3)$</td>
<td>$10^3$</td>
<td>$10^6$</td>
<td>$10^9$</td>
<td>$10^{12}$</td>
<td>$16.7$ min</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^n)$</td>
<td>$10^{24}$</td>
<td>$10^{48}$</td>
<td>$3.17 \times 10^{72}$ yrs</td>
<td>$10^{96}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Complexity</th>
<th>$n$</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>1</td>
<td>1 µsec</td>
<td>1 µsec</td>
<td>1 µsec</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log n)$</td>
<td>1.33</td>
<td>13.3 µsec</td>
<td>16.6 µsec</td>
<td>19.93 µsec</td>
</tr>
<tr>
<td>linear</td>
<td>$O(n)$</td>
<td>$10^4$</td>
<td>$10^5$</td>
<td>$0.1$ sec</td>
<td>$10^6$</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$133 \times 10^3$</td>
<td>$133$ msec</td>
<td>$166 \times 10^4$</td>
<td>$1.6$ sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(n^2)$</td>
<td>$10^6$</td>
<td>$1.7$ min</td>
<td>$10^{10}$</td>
<td>$16.7$ min</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(n^3)$</td>
<td>$10^{12}$</td>
<td>$31.6$ days</td>
<td>$10^{15}$</td>
<td>$31.7$ yr</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^n)$</td>
<td>$10^{9002}$</td>
<td>$10^{9010}$</td>
<td>$10^{9018}$</td>
<td>$10^{9020}$</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

for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;  \[O(n)\]
    }  \[O(1)\]
}
Nested Loops

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

$O(n)$

Nested Loops

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

$O(n^2)$
Nested Loops

- Note: Running time grows with nesting rather than the length of the code

```java
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
\[O(n^2)\]
```

More Nested Loops

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

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

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

Running time: \(\max(O(n), O(n^2)) = O(n^2)\)

Conditionals

```java
for(int t=0; t<n; ++t) {
    if(t%2) {
        for(int u=t; u<n; ++u) {
            ++zap;
        }
    } else {
        zap = 0;
    }
}
```

Running time: \(O(n)\) and \(O(1)\)
Conditionals

```java
for(int t=0; t<n; ++t) {
    if(t%2) {
        for(int u=t; u<n; ++u) {
            ++zap;
        }
    } else {
        zap = 0;
    }
}
```

$O(n^2)$
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
Caveats

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

- Depends on input problem instance type

Input Space

- Worse configuration
- Neither worse or best
- Best configuration

Behaviour

- Worst-Case Behaviour
- Average-Case Behaviour
- Best-Case Behaviour

Work

N
Computational Complexity Theory

- In computer science, computational complexity theory is the branch of the theory of computation that studies the resources, or cost, of the computation required to solve a given computational problem.
- Complexity theory analyzes the difficulty of computational problems in terms of many different computational resources.

Note

**Solve a problem**

*vs.*

**Verify a solution**

- E.g.,
  - Sort
  - Shortest path
Complexity Classes

- A complexity class is the set of all of the computational problems which can be solved using a certain amount of a certain computational resource

Deterministic Turing Machine

- Deterministic or Turing machines are extremely basic symbol-manipulating devices which — despite their simplicity — can be adapted to simulate the logic of any computer that could possibly be constructed
- Described in 1936 by Alan Turing.
  - Not meant to be a practical computing technology
  - Technically feasible
  - A thought experiment about the limits of mechanical computation
Deterministic Turing Machine

Non-Deterministic Turing Machine

- Turing machine whose control mechanism works like a non-deterministic finite automaton
### Class Resource Model Constraint

<table>
<thead>
<tr>
<th>Class</th>
<th>Resource</th>
<th>Model</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTIME(f(n))</td>
<td>Time</td>
<td>DTM</td>
<td>f(n)</td>
</tr>
<tr>
<td>P</td>
<td>Time</td>
<td>DTM</td>
<td>O(n^k)</td>
</tr>
<tr>
<td>EXPTIME</td>
<td>Time</td>
<td>DTM</td>
<td>O(2^n^k)</td>
</tr>
<tr>
<td>NTIME</td>
<td>Time</td>
<td>NDTM</td>
<td>f(n)</td>
</tr>
<tr>
<td>NP</td>
<td>Time</td>
<td>NDTM</td>
<td>O(n^k)</td>
</tr>
<tr>
<td>NEXPTIME</td>
<td>Time</td>
<td>NDTM</td>
<td>O(2^n^k)</td>
</tr>
<tr>
<td>DSPACE(f(n))</td>
<td>Space</td>
<td>DTM</td>
<td>f(n)</td>
</tr>
<tr>
<td>L</td>
<td>Space</td>
<td>DTM</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>PSPACE</td>
<td>Space</td>
<td>DTM</td>
<td>O(n^k)</td>
</tr>
<tr>
<td>EXPSPACE</td>
<td>Space</td>
<td>DTM</td>
<td>O(2^n^k)</td>
</tr>
<tr>
<td>NSPACE(f(n))</td>
<td>Space</td>
<td>NDTM</td>
<td>f(n)</td>
</tr>
<tr>
<td>NL</td>
<td>Space</td>
<td>NDTM</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>NPSpace</td>
<td>Space</td>
<td>NDTM</td>
<td>O(n^k)</td>
</tr>
<tr>
<td>NEXPSpace</td>
<td>Space</td>
<td>NDTM</td>
<td>O(2^n^k)</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>ln</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>nlgn</td>
<td>“n-log-n”</td>
<td>Some divide and conquer</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 \log 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>
<tr>
<td>( n^3 )</td>
<td>Cubic</td>
<td>Loop inside 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>\text{nlgn}</td>
<td>“n-log-n”</td>
<td>Some divide and conquer</td>
</tr>
<tr>
<td>\text{} \text{n}</td>
<td>Quadratic</td>
<td>Loop inside loop = “nested loop”</td>
</tr>
<tr>
<td>\text{} \text{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></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>$O(n) + O(1)$</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 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/](http://creativecommons.org/licenses/by-nc-sa/3.0/)