Part 5.
Computational Complexity (1)

Outline

- Measuring efficiencies of algorithms
- Growth of functions
- Asymptotic bounds for the growth functions
- Big-O notation

Algorithm and Computational Complexity

- An **algorithm** is a finite sequence of precise instructions for performing a computation for solving a problem.

- **Computational complexity** measures the processing time and computer memory required by the algorithm to solve problems of particular size.

Software cost factors

- Human costs
  - Time of developers, testers, maintainers, support team, users

- Managing human costs
  - Modularity and Abstraction
  - Information hiding, good style, readability

Software cost factors (cont’d)

- Efficiency of algorithms
  - **Time** to execute algorithms
  - **Space** required by algorithms
Measuring the efficiency of algorithms

- We have two algorithms, alg1 and alg2, that solve the same problem.
  - Our application needs a fast running time.
- How do we choose between the algorithms?

Example (1/3)

- An elevator is trying to move 25 packages to the first level.
- From second to sixth floor, on each of the floors, we have 5 packages waiting for the elevator to be transported to the first floor.
- The elevator can move 10 packages at a time.
- Initially elevator is on the first floor.

Example (2/3)

- **Solution 1**: Transport packages on the second floor to the first floor, and transport packages on the third floor to the first floor, etc.
- **Analysis of Solution 1**: What is the total number of levels that the elevator traveled?

Example (3/3)

- **Solution 2**:
  - Pick up packages on the third floor, and stop and pick up packages at the second floor and unload them on the first.
  - Pick up packages on the fifth floor and pick up packages on the fourth floor and unload them on the first floor.
  - Pick up packages on the sixth floor and move them to the first floor.
- **Analysis of Solution 1**: What is the total number of levels travelled?

Comparing Solution 1 and 2 for n levels

- Solution 1: \(2(1+2+3+...+n)\)
- Solution 2:
  - 2(2+4+6+...+n) (if \(n\) is even)
  - 2(2+4+6+...(n-1)+n) (if \(n\) is odd)

<table>
<thead>
<tr>
<th>(n)</th>
<th>Solution 1</th>
<th>Solution 2</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td>42</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>54</td>
<td>12</td>
</tr>
</tbody>
</table>

- In this comparison what did we ignore?

Measuring the efficiency of algorithms

- Implement the two versions in Java and compare their running times
- Issues with this approach:
  - How are the algorithms coded? We want to compare the algorithms, not the implementations.
  - What computer should we use? Choice of operations could favor one implementation over another.
  - What data should we use? Choice of data could favor one algorithm over another
Measuring the efficiency of algorithms

- Objective: **Analyze** algorithms independent of specific implementations, hardware, or data
- Observation: An algorithm’s execution time is related to the number of operations it executes
- Solution: **Count** the number of significant operations the algorithm will perform for the given problem size

Examples

- Copying an array with \( n \) elements requires ___ invocations of operations

Example

- Finding the maximum element in a finite sequence

```java
public int max (in: array of positive integers a[\()]
int max=-1;
for (int i = 0; i < size_of_array; i++){
    if ( max < a[i] ) max = a[i];
}
return max;
}
```

For the input array with size of \( n \) integers, for loop is executed \( n \) times.

Outline

- Measuring efficiencies of algorithms
- **Growth of functions**
- Asymptotic bounds for the growth functions
- Big-O notation

Growth rates

<table>
<thead>
<tr>
<th>( n^2 / 2 )</th>
<th>1/2</th>
<th>4/2</th>
<th>9/2</th>
<th>16/2</th>
<th>25/2</th>
<th>36/2</th>
<th>49/2</th>
<th>64/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5n+10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>50</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^2 / 2 )</td>
<td>1250</td>
<td>5,000</td>
<td>500,000</td>
<td>50,000,000</td>
<td>...</td>
</tr>
<tr>
<td>5n+10</td>
<td>260</td>
<td>510</td>
<td>5,010</td>
<td>50,010</td>
<td>...</td>
</tr>
</tbody>
</table>

Which one would you choose?
Growth Rates

Important to know how quickly an algorithm's execution time grows as a function of its input data size.

We focus on the growth rate:
- Algorithm A requires time proportional to $n^2$
- Algorithm B requires time proportional to $n$
- B's time requirements grows more slowly than A's time requirement (for large $n$)

Outline

- Measuring efficiencies of algorithms
- Growth of functions
- Asymptotic bounds for the growth functions
- Big-O notation

Growth of functions

Measure the speed of algorithm
- Growth of number of operations (relative to the input size)
- $n$: input size, positive integer
- $x$: real number

Asymptotic Bounds

- Big-O notation
- Big-Omega notation
- Big-Theta notation
Let $f$ and $g$ be functions from the set of integers or the set of real numbers to the set of real numbers. We say $f(x)$ is $O(g(x))$ if there are constants $C$ and $k$ such that, 

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$

We say $f(x)$ is $\Omega(g(x))$ if there are positive constants $C$ and $k$ such that, 

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$

We say $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$

### Outline

- Measuring efficiencies of algorithms
- Growth of functions
- Asymptotic bounds for the growth functions
- Big-O notation
Order of magnitude analysis

Big O notation:
• Let \( f \) and \( g \) be functions from the set of integers or the set of real numbers to the set of real numbers. We say \( f(x) \) is \( O(g(x)) \) if there are constants \( C \) and \( k \) such that, \(|f(x)| \leq C|g(x)|\) whenever \( x > k \)
• Focus is on the shape of the function
  – Ignore the multiplicative constant
• Focus is on large \( x \)
  – \( k \) allows us to ignore behavior for small \( x \)

Common Shapes: Constant

• \( O(1) \)

• examples?

Common Shapes: Linear

• \( O(n) \)

\[ f(n) = a*n + b \]

Linear

Example: copying an array

```c
for (int i = 0; i < a.size; i++){
    a[i] = b[i];
}
```

How many times should the line A be executed to finish this task?

Other Shapes: Sublinear

\( \log(x) \), \( \sqrt{x} \)
Common Shapes: logarithm

- \( \log_b n \) is the number \( x \) such that \( b^x = n \)
  
  \[
  \begin{align*}
  2^3 &= 8 & \log_2 8 &= 3 \\
  2^4 &= 16 & \log_2 16 &= 4 \\
  \\
  \cdot \text{ We usually work with base 2}
  \end{align*}
  \]

Logarithms (cont.)

- Properties of logarithms
  - \( \log(xy) = \log x + \log y \)
  - \( \log(x^a) = a \log x \)
  - \( \log_a n = \log_b n / \log_b a \)

  - logarithm is a very slow-growing function
  - Examples of logarithmic complexity?

Quadratic

\( O(n^2) \):

```
for (int i=0; i < n; i++) { 
}\text{n times}
  for (int j=0; j < n; j++) { 
}\text{n times}
  }
```

Other Shapes: Superlinear

- Polynomial \((x^a)\), exponential \((e^x)\)

Proof

- Show that \( f(x) = x^2 + 2x + 1 \) is \( O(x^2) \)

Proof

- Show that \( f(x) = 7x^2 \) is \( O(x^3) \)
**Proof**

- Show that \( f(x) = x^2 \) is NOT \( O(x) \)

---

**Big-O for Polynomials**

Theorem: Let

\[
f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0
\]

where \( a_n, a_{n-1}, \ldots, a_1, a_0 \) are real numbers.

Then \( f(x) \) is \( O(x^n) \).

Example: \( x^2 + 5x \) is \( O(x^2) \)

Proof:

---

**Properties of Growth-rate functions (1/3)**

1. You can ignore low-order terms in an algorithm’s growth-rate function.
   - \( O(n^2 + 4n^2 + 3n) \) it is also \( O(n^3) \)

---

**Properties of Growth-rate functions (2/3)**

2. You can ignore a multiplicative constant in the high-order term of an algorithm’s growth-rate function
   - \( O(5n^3) \), it is also \( O(n^3) \)

---

**Properties of Growth-rate functions (3/3)**

3. You can combine growth-rate functions
   - \( O(n^2) + O(n) \), it is also \( O(n^2 + n) \)
   - Which you write as \( O(n^2) \)

---

**Combinations of Functions**

- **Additive Theorem:**
  Suppose that \( f_1(x) = O(g_1(x)) \) and \( f_2(x) = O(g_2(x)) \).
  Then \( (f_1 + f_2)(x) = O(\max\{ g_1(x), g_2(x) \} ) \).

- **Multiplicative Theorem:**
  Suppose that \( f_1(x) = O(g_1(x)) \) and \( f_2(x) = O(g_2(x)) \).
  Then \( (f_1 f_2)(x) = O(g_1(x)g_2(x)) \).
Practical Analysis - Combinations

- Sequential
  - Big-O bound: Steepest growth dominates
  - Example: copying of array, followed by binary search
    - $n + \log(n) \quad O(?)$

- Embedded code
  - Big-O bound multiplicative
  - Example: a for loop with $n$ iterations and a body taking $O(\log n) \quad O(?)$