Computational Complexity: Measuring the Efficiency of Algorithms

- Rosen Ch. 3.2: Growth of Functions
- Rosen Ch. 3.3: Complexity of Algorithms
- Prichard Ch. 10.1: Efficiency of Algorithms

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
  - Adherence to software engineering principles
    - Modularity and Abstraction (separation of concerns principle)
    - Information hiding, good style, readability (design for change principle)

Software cost factors (cont’d)

- Efficiency of algorithms
  - Time to execute algorithms
  - Space required by algorithms
- Focus of this week’s lectures
Measuring the efficiency of algorithms

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

Implement the two algorithms 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.

Objective: analyze algorithms independently 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 an input of given size.

Example: Clicker Q

- Copying an array with \( n \) elements requires ___ invocations of copy operations
  1. 1
  2. \( n \)
  3. 2n
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.

Example: Clicker Q

Number of positions to check when running binary search on an array of size 32 when the element is not there:
1. 1
2. 2
3. 5
4. 32

Growth rates

- Algorithm A requires $n^2/2$ operations to solve a problem of size $n$
- Algorithm B requires $5n+10$ operations to solve a problem of size $n$
- Which one would you choose?

Growth rates

When we increase the size of input $n$, how the execution time grows for these algorithms?

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2/2$</td>
<td>1/2</td>
<td>4/2</td>
<td>9/2</td>
<td>16/2</td>
<td>25/2</td>
<td>36/2</td>
<td>49/2</td>
<td>64/2</td>
</tr>
<tr>
<td>5n+10</td>
<td>15</td>
<td>20</td>
<td>45</td>
<td>70</td>
<td>105</td>
<td>140</td>
<td>185</td>
<td>230</td>
</tr>
<tr>
<td>n</td>
<td>50</td>
<td>100</td>
<td>1,000</td>
<td>10,000</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n^2/2$</td>
<td>1250</td>
<td>5,000</td>
<td>500,000</td>
<td>50,000,000</td>
<td>...</td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Growth Rates

Algorithm A requires \( n^2 / 2 \) operations to solve a problem of size \( n \)
Algorithm B requires \( 5n + 10 \) operations to solve a problem of size \( n \)
For large enough problem size algorithm B is more efficient

Important to know how quickly an algorithm's execution time grows as a function of program 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 \( n \))

Order of magnitude analysis

**Big O notation**: A function \( f(x) \) is \( O(g(x)) \) if there exist two positive constants, \( c \) and \( k \), such that
\[ f(x) \leq c \cdot g(x) \quad \forall x > k \]

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 \( x \)
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 $x > k$.

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

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

whenever $x > k$. 
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 that $f(x)$ is $O(g(x))$ if $f(x)$ is $\Theta(g(x))$ and $f(x)$ is $\Omega(g(x))$.

Order of magnitude analysis

- **Big O notation**: A function $f(x)$ is $O(g(x))$ if there exist two positive constants, $c$ and $k$, such that $f(x) \leq cg(x)$ for all $x > k$.
- $c$ and $k$ are witnesses to the relationship that $f(x)$ is $O(g(x))$.
- If there is one pair of witnesses $(c, k)$ then there are infinitely many.

Common Shapes: Constant

- $O(1)$

examples?
Common Shapes: Linear

- $O(n)$

$$f(n) = an + b$$

Example: copying an array

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

Other Shapes: Sublinear

- $\log_b n$ is the number $x$ such that $b^x = n$
  - $2^3 = 8 \quad \log_2 8 = 3$
  - $2^4 = 16 \quad \log_2 16 = 4$
  - $\log_b n$: (# of digits to represent n in base b) – 1
  - We usually work with base 2

Common Shapes: logarithm
Logarithms (cont.)

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

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

Quadratic

$O(n^2)$:

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

Other Shapes: Superlinear

Big-O for Polynomials

Theorem: Let

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

where $a_n, a_{n-1}, ..., a_1, a_0$ are real numbers.

Then $f(x)$ is $O(x^n)$

Example: $x^2 + 5x$ is $O(x^2)$
Clicker Q

Give as good a Big O estimate as possible for the following growth function.

\[ f(n) = (3n^2 + 8)(n + 1) \]

(a) \( O(n) \)
(b) \( O(n^3) \)
(c) \( O(n^2) \)
(d) \( O(1) \)

Combinations of Functions

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

- **Multiplicative Theorem:**
  
  Suppose that \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \).
  
  Then \( (f_1 f_2)(x) \) is \( O(g_1(x) g_2(x)) \).

Practical Analysis - Combinations

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

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

Worst and Average Case

Time Complexity

- **Worst case**
  - just how bad can it get: the maximal number of steps
  - our focus in this course

- **Average case**
  - amount of time expected “usually”
  - in this course we will hand wave when it comes to average case

- **Best case**
  - The smallest number of steps

- Example: searching for an item in an unsorted array
Practical Analysis - Loops

1. public void insertElementAt(Object obj, int index) {
   ...
2. for (i = elementCount; i > index; i--) {
   3.    elementData[i] = elementData[i-1];
   ...
   }
   ...

   How many times will line 3 repeat?
   On what does the number depend?

Practical Analysis – Dependent loops

   for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
         ...
      }
   }

   i = 0: inner-loop iters = 0
   i = 1: inner-loop iters = 1
   ...
   i = n-1: inner-loop iters = n-1

   Total = 0 + 1 + 2 + ... + (n-1)
   f(n) = n*(n-1)/2
   O(n^2)

Practical Analysis – Recursion

   Number of operations depends on:
   - number of calls
   - work done in each call

   Examples:
   - factorial: how many recursive calls?
   - binary search?

   We will devote more time to analyzing recursive algorithms later in the course.

Final Comments

   - Order-of-magnitude analysis focuses on large problems
   - If the problem size is always small, you can probably ignore an algorithm’s efficiency
   - Weigh the trade-offs between an algorithm’s time requirements and its memory requirements, expense of programming/maintenance…