Asymptotic Analysis of Algorithms

EECS2030: Advanced Object Oriented Programming
Fall 2017

CHEN-WEI WANG
Algorithm and Data Structure

- **A data structure** is:
  - A systematic way to store and organize data in order to facilitate *access* and *modifications*
  - Never suitable for all purposes: it is important to know its *strengths* and *limitations*

- **A well-specified computational problem** precisely describes the desired *input/output relationship*.
  - **Input**: A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output**: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
  - An *instance* of the problem: $\langle 3, 1, 2, 5, 4 \rangle$

- **An algorithm** is:
  - A solution to a well-specified *computational problem*
  - A *sequence of computational steps* that takes value(s) as *input* and produces value(s) as *output*

- Steps in an *algorithm* manipulate well-chosen *data structure(s).*
1. **Correctness**:  
   ○ Does the algorithm produce the expected output?  
   ○ Use JUnit to ensure this.

2. **Efficiency**:  
   ○ *Time Complexity*: processor time required to complete  
   ○ *Space Complexity*: memory space required to store data

*Correctness* is always the priority.

How about efficiency? Is time or space more of a concern?
Measuring Efficiency of an Algorithm

- **Time** is more of a concern than is **storage**.
- Solutions that are meant to be run on a computer should run *as fast as possible*.
- Particularly, we are interested in how running time depends on two **input factors**:
  1. size
e.g., sorting an array of 10 elements vs. 1m elements
  2. structure
e.g., sorting an already-sorted array vs. a hardly-sorted array
- **How do you determine the running time of an algorithm?**
  1. Measure time via *experiments*
  2. Characterize time as a *mathematical function* of the input size
Measure Running Time via Experiments

- Once the algorithm is implemented in Java:
  - Execute the program on *test inputs* of various *sizes* and *structures*.
  - For each test, record the *elapsed time* of the execution.
    ```java
    long startTime = System.currentTimeMillis();
    /* run the algorithm */
    long endTime = System.currentTimeMillis();
    long elapsed = endTime - startTime;
    ```
  - *Visualize* the result of each test.

- To make *sound statistical claims* about the algorithm’s *running time*, the set of input tests must be “reasonably” *complete*. 
Example Experiment

- **Computational Problem:**
  - **Input:** A character \( c \) and an integer \( n \)
  - **Output:** A string consisting of \( n \) repetitions of character \( c \)
    e.g., Given input ‘*’ and 15, output *******************.

- **Algorithm 1 using `String` Concatenations:**
  ```java
  public static String repeat1(char c, int n) {
    String answer = "";
    for (int i = 0; i < n; i++) {
      answer += c;
    }
    return answer;
  }
  ```

- **Algorithm 2 using `StringBuilder` append’s:**
  ```java
  public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
      sb.append(c);
    }
    return sb.toString();
  }
  ```
## Example Experiment: Detailed Statistics

<table>
<thead>
<tr>
<th>n</th>
<th>repeat1 (in ms)</th>
<th>repeat2 (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50,000</td>
<td>2,884</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>7,437</td>
<td>1</td>
</tr>
<tr>
<td>200,000</td>
<td>39,158</td>
<td>2</td>
</tr>
<tr>
<td>400,000</td>
<td>170,173</td>
<td>3</td>
</tr>
<tr>
<td>800,000</td>
<td>690,836</td>
<td>7</td>
</tr>
<tr>
<td>1,600,000</td>
<td>2,847,968</td>
<td>13</td>
</tr>
<tr>
<td>3,200,000</td>
<td>12,809,631</td>
<td>28</td>
</tr>
<tr>
<td>6,400,000</td>
<td>59,594,275</td>
<td>58</td>
</tr>
<tr>
<td>12,800,000</td>
<td>265,696,421 (≈ 3 days)</td>
<td>135</td>
</tr>
</tbody>
</table>

- As *input size* is doubled, **rates of increase** for both algorithms are *linear*:
  - *Running time* of `repeat1` increases by ≈ 5 times.
  - *Running time* of `repeat2` increases by ≈ 2 times.
Example Experiment: Visualization

![Graph showing running time versus n for repeat1 and repeat2]
Experimental Analysis: Challenges

1. An algorithm must be *fully implemented* (i.e., translated into valid Java syntax) in order study its runtime behaviour *experimentally*.
   ○ What if our purpose is to *choose among alternative* data structures or algorithms to implement?
   ○ Can there be a *higher-level analysis* to determine that one algorithm or data structure is *superior* than others?

2. Comparison of multiple algorithms is only *meaningful* when experiments are conducted under the same environment of:
   ○ *Hardware*: CPU, running processes
   ○ *Software*: OS, JVM version

3. Experiments can be done only on *a limited set of test inputs*.
   ○ What if “*important*” inputs were not included in the experiments?
Moving Beyond Experimental Analysis

- A better approach to analyzing the **efficiency** (e.g., **running times**) of algorithms should be one that:
  - Allows us to calculate the **relative efficiency** (rather than absolute elapsed time) of algorithms in a ways that is *independent of* the hardware and software environment.
  - Can be applied using a **high-level description** of the algorithm (without fully implementing it).
  - Considers **all** possible inputs.

- We will learn a better approach that contains 3 ingredients:
  1. Counting **primitive operations**
  2. Approximating running time as a **function of input size**
  3. Focusing on the **worst-case** input (requiring the most running time)
Counting Primitive Operations

- A **primitive operation** corresponds to a low-level instruction with a **constant execution time**.
  - Assignment [e.g., \( x = 5; \)]
  - Indexing into an array [e.g., \( a[i] \)]
  - Arithmetic, relational, logical op. [e.g., \( a + b, z > w, b1 && b2 \)]
  - Accessing a field of an object [e.g., \( \text{acc.balance} \)]
  - Returning from a method [e.g., \( \text{return result}; \)]
  - Why is a method call is in general *not* a primitive operation?

- The **number of primitive operations** required by an algorithm should be **proportional** to its **actual running time** on a specific environment: \( RT = \sum_{i=1}^{N} t(i) \) [ \( N = \# \text{ of PO's} \) ]
  - Say \( c \) is the **absolute** time of executing a **primitive operation** on a specific computer platform.
  - \( RT = \sum_{i=1}^{N} t(i) = c \times N \approx N \)

\( \Rightarrow \) approximate \# of primitive operations that its steps contain.
Example: Counting Primitive Operations

```c
findMax (int[] a, int n) {
    currentMax = a[0];
    for (int i = 1; i < n; ) {
        if (a[i] > currentMax) {
            currentMax = a[i];
        }
        i ++
    }
    return currentMax;
}
```

# of times \( i < n \) in **Line 3** is executed? \( [n] \)

# of times the loop body (**Line 4 to Line 6**) is executed? \( [n-1] \)

- **Line 2**: 2 \( [1 \text{ indexing} + 1 \text{ assignment}] \)
- **Line 3**: \( n+1 \) \( [1 \text{ assignment} + n \text{ comparisons}] \)
- **Line 4**: \( (n-1) \cdot 2 \) \( [1 \text{ indexing} + 1 \text{ comparison}] \)
- **Line 5**: \( (n-1) \cdot 2 \) \( [1 \text{ indexing} + 1 \text{ assignment}] \)
- **Line 6**: \( (n-1) \cdot 2 \) \( [1 \text{ addition} + 1 \text{ assignment}] \)
- **Line 7**: 1 \( [1 \text{ return}] \)

**Total # of Primitive Operations**: \( 7n - 2 \)
Example: Approx. # of Primitive Operations

- Given # of primitive operations counted precisely as $7n^1 - 2$, we view it as
  
  $$7 \cdot n - 2 \cdot n^0$$

- We say
  - $n$ is the highest power
  - 7 and 2 are the multiplicative constants
  - 2 is the lower term

- When approximating a function (considering that input size may be very large):
  - Only the highest power matters.
  - multiplicative constants and lower terms can be dropped.

  $\Rightarrow 7n - 2$ is approximately $n$

Exercise: Consider $7n + 2n \cdot \log n + 3n^2$:

- highest power?
- multiplicative constants?
- lower terms?
Approximating Running Time as a Function of Input Size

Given the *high-level description* of an algorithm, we associate it with a function $f$, such that $f(n)$ returns the *number of primitive operations* that are performed on an *input of size $n$*. 

- $f(n) = 5$ [constant]
- $f(n) = \log_2 n$ [logarithmic]
- $f(n) = 4 \cdot n$ [linear]
- $f(n) = n^2$ [quadratic]
- $f(n) = n^3$ [cubic]
- $f(n) = 2^n$ [exponential]
Focusing on the Worst-Case Input

- **Average-case** analysis calculates the *expected running times* based on the probability distribution of input values.
- **Worst-case** analysis or **best-case** analysis?
What is Asymptotic Analysis?

**Asymptotic analysis**

- Is a method of describing *behaviour in the limit*:
  - How the *running time* of the algorithm under analysis changes as the *input size* changes without bound
  - e.g., contrast $RT_1(n) = n$ with $RT_2(n) = n^2$

- Allows us to compare the *relative* performance of alternative algorithms:
  - For large enough inputs, the *multiplicative constants* and *lower-order* terms of an exact running time can be disregarded.
  - e.g., $RT_1(n) = 3n^2 + 7n + 18$ and $RT_1(n) = 100n^2 + 3n − 100$ are considered equally efficient, asymptotically.
  - e.g., $RT_1(n) = n^3 + 7n + 18$ is considered less efficient than $RT_1(n) = 100n^2 + 100n + 2000$, asymptotically.
Three Notions of Asymptotic Bounds

We may consider three kinds of *asymptotic bounds* for the *running time* of an algorithm:

- Asymptotic *upper* bound $[O]\text{ [O]}$
- Asymptotic lower bound $[\Omega]\text{ [Ω]}$
- Asymptotic tight bound $[\Theta]\text{ [Θ]}$
Asymptotic Upper Bound: Definition

- Let $f(n)$ and $g(n)$ be functions mapping positive integers (input size) to positive real numbers (running time).
  - $f(n)$ characterizes the running time of some algorithm.
  - $O(g(n))$ denotes a collection of functions.
- $O(g(n))$ consists of all functions that can be upper bounded by $g(n)$, starting at some point, using some constant factor.
- $f(n) \in O(g(n))$ if there are:
  - A real constant $c > 0$
  - An integer constant $n_0 \geq 1$
  such that:
  \[ f(n) \leq c \cdot g(n) \quad \text{for} \quad n \geq n_0 \]

- For each member function $f(n)$ in $O(g(n))$, we say that:
  - $f(n) \in O(g(n))$ [f(n) is a member of “big-Oh of g(n)”]
  - $f(n)$ is $O(g(n))$ [f(n) is “big-Oh of g(n)”]
  - $f(n)$ is order of $g(n)$
Asymptotic Upper Bound: Visualization

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_0$</td>
<td></td>
</tr>
</tbody>
</table>

From $n_0$, $f(n)$ is upper bounded by $c \cdot g(n)$, so $f(n)$ is $O(g(n))$. 

![Graph showing asymptotic upper bound]
Asymptotic Upper Bound: Example (1)

**Prove**: The function $8n + 5$ is $O(n)$.

**Strategy**: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$8n + 5 \leq c \cdot n$$

Can we choose $c = 9$? What should the corresponding $n_0$ be?

<table>
<thead>
<tr>
<th>n</th>
<th>8n + 5</th>
<th>9n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>29</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>53</td>
<td>54</td>
</tr>
</tbody>
</table>

Therefore, we prove it by choosing $c = 9$ and $n_0 = 5$.

We may also prove it by choosing $c = 13$ and $n_0 = 1$. Why?
**Asymptotic Upper Bound: Example (2)**

**Prove:** The function \( f(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1 \) is \( O(n^4) \).

**Strategy:** Choose a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \), such that for every integer \( n \geq n_0 \):

\[
5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq c \cdot n^4
\]

\[
f(1) = 5 + 3 + 2 + 4 + 1 = 15
\]

Choose \( c = 15 \) and \( n_0 = 1 \)!
Asymptotic Upper Bound: Proposition (1)

If \( f(n) \) is a polynomial of degree \( d \), i.e.,

\[
f(n) = a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d
\]

and \( a_0, a_1, \ldots, a_d \) are integers (i.e., negative, zero, or positive), then \( f(n) \) is \( O(n^d) \).

Proof:

1. We know that for \( n \geq 1 \):
   \[
n^0 \leq n^1 \leq n^2 \leq \cdots \leq n^d
   \]

2. By choosing \( c = |a_0| + |a_1| + \cdots + |a_d| \):
   \[
a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d \leq |a_0| \cdot n^d + |a_1| \cdot n^d + \cdots + |a_d| \cdot n^d
   \]

3. By choosing \( n_0 = 1 \):
   \[
a_0 \cdot 1^0 + a_1 \cdot 1^1 + \cdots + a_d \cdot 1^d \leq |a_0| \cdot 1^d + |a_1| \cdot 1^d + \cdots + |a_d| \cdot 1^d
   \]

That is, we prove by choosing

\[
\begin{align*}
c &= |a_0| + |a_1| + \cdots + |a_d| \\
n_0 &= 1
\end{align*}
\]
Asymptotic Upper Bound: Proposition (2)

\[ O(n^0) \subset O(n^1) \subset O(n^2) \subset \ldots \]

If a function \( f(n) \) is upper bounded by another function \( g(n) \) of degree \( d, \; d \geq 0 \), then \( f(n) \) is also upper bounded by all other functions of a strictly higher degree (i.e., \( d + 1, \; d + 2, \; \text{etc.} \)).
Asymptotic Upper Bound: More Examples

- $5n^2 + 3n \cdot \log n + 2n + 5$ is $O(n^2)$  
  $[c = 15, n_0 = 1]$
- $20n^3 + 10n \cdot \log n + 5$ is $O(n^3)$  
  $[c = 35, n_0 = 1]$
- $3 \cdot \log n + 2$ is $O(\log n)$  
  $[c = 5, n_0 = 2]$

  ○ Why can’t $n_0$ be 1?
  ○ Choosing $n_0 = 1$ means $\Rightarrow f(1)$ is upper-bounded by $c \cdot \log 1$:
    - We have $f(1) = 3 \cdot \log 1 + 2$, which is 2.
    - We have $c \cdot \log 1$, which is 0.
    $\Rightarrow f(1)$ is not upper-bounded by $c \cdot \log 1$  
  [Contradiction!]
- $2^{n+2}$ is $O(2^n)$  
  $[c = 4, n_0 = 1]$
- $2n + 100 \cdot \log n$ is $O(n)$  
  $[c = 102, n_0 = 1]$
Using Asymptotic Upper Bound Accurately

- Use the big-Oh notation to characterize a function (of an algorithm’s running time) \textit{as closely as possible}. For example, say $f(n) = 4n^3 + 3n^2 + 5$:
  - Recall: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \ldots$
  - It is the \textit{most accurate} to say that $f(n)$ is $O(n^3)$.
  - It is also true, but not very useful, to say that $f(n)$ is $O(n^4)$ and that $f(n)$ is $O(n^5)$.

- Do not include \textit{constant factors} and \textit{lower-order terms} in the big-Oh notation.
  For example, say $f(n) = 2n^2$ is $O(n^2)$, do not say $f(n)$ is $O(4n^2 + 6n + 9)$. 

## Classes of Functions

<table>
<thead>
<tr>
<th>upper bound</th>
<th>class</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
<td><strong>cheapest</strong></td>
</tr>
<tr>
<td>$O(\log(n))$</td>
<td>logarithmic</td>
<td></td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>$O(n \cdot \log(n))$</td>
<td>“n-log-n”</td>
<td></td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
<td></td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>cubic</td>
<td></td>
</tr>
<tr>
<td>$O(n^k), k \geq 1$</td>
<td>polynomial</td>
<td></td>
</tr>
<tr>
<td>$O(a^n), a &gt; 1$</td>
<td>exponential</td>
<td><strong>most expensive</strong></td>
</tr>
</tbody>
</table>
Rates of Growth: Comparison

The graph illustrates the rates of growth for different functions:
- Linear
- Exponential
- Constant
- Logarithmic
- N-Log-N
- Quadratic
- Cubic
- Linear-N
- Logarithmic
- Constant

The x-axis represents the input n, while the y-axis represents f(n) on a logarithmic scale.
Upper Bound of Algorithm: Example (1)

```c
maxOf (int x, int y) {
    int max = x;
    if (y > x) {
        max = y;
    }
    return max;
}
```

- # of primitive operations: 4
  2 assignments + 1 comparison + 1 return = 4
- Therefore, the running time is \( O(1) \).
- That is, this is a constant-time algorithm.
Upper Bound of Algorithm: Example (2)

```
findMax (int[] a, int n) {
    currentMax = a[0];
    for (int i = 1; i < n; ) {
        if (a[i] > currentMax) {
            currentMax = a[i];
        }
        i ++
    }
    return currentMax; }
```

- From last lecture, we calculated that the # of primitive operations is \(7n - 2\).
- Therefore, the running time is \(O(n)\).
- That is, this is a linear-time algorithm.
Upper Bound of Algorithm: Example (3)

```java
containsDuplicate (int[] a, int n) {
    for (int i = 0; i < n; ) {
        for (int j = 0; j < n; ) {
            if (i != j && a[i] == a[j]) {
                return true; }
            j ++; }
        i ++; }
    return false; }
```

- Worst case is when we reach Line 8.
- # of primitive operations \(\approx c_1 + n \cdot n \cdot c_2\), where \(c_1\) and \(c_2\) are some constants.
- Therefore, the running time is \(O(n^2)\).
- That is, this is a quadratic algorithm.
Upper Bound of Algorithm: Example (4)

```cpp
int sumMaxAndCrossProducts (int[] a, int n) {
    int max = a[0];
    for (int i = 1; i < n; ) {
        if (a[i] > max) { max = a[i]; }
    }
    int sum = max;
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            sum += a[j] * a[k];
        }
    }
    return sum;
}
```

- # of primitive operations $\approx (c_1 \cdot n + c_2) + (c_3 \cdot n \cdot n + c_4)$, where $c_1$, $c_2$, $c_3$, and $c_4$ are some constants.
- Therefore, the running time is $O(n + n^2) = O(n^2)$.
- That is, this is a *quadratic* algorithm.
Upper Bound of Algorithm: Example (5)

```java
triangularSum (int[] a, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            sum += a[j];
        }
    }
    return sum;
}
```

- # of primitive operations $\approx n + (n - 1) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$
- Therefore, the running time is $O\left(\frac{n^2+n}{2}\right) = O(n^2)$.
- That is, this is a quadratic algorithm.
Index (1)

Algorithm and Data Structure
Measuring "Goodness" of an Algorithm
Measuring Efficiency of an Algorithm
Measure Running Time via Experiments
Example Experiment
Example Experiment: Detailed Statistics
Example Experiment: Visualization
Experimental Analysis: Challenges
Moving Beyond Experimental Analysis
Counting Primitive Operations
Example: Counting Primitive Operations
Example: Approx. # of Primitive Operations
Approximating Running Time as a Function of Input Size
Index (2)

Focusing on the Worst-Case Input
What is Asymptotic Analysis?
Three Notions of Asymptotic Bounds
Asymptotic Upper Bound: Definition
Asymptotic Upper Bound: Visualization
Asymptotic Upper Bound: Example (1)
Asymptotic Upper Bound: Example (2)
Asymptotic Upper Bound: Proposition (1)
Asymptotic Upper Bound: Proposition (2)
Asymptotic Upper Bound: More Examples
Using Asymptotic Upper Bound Accurately
Classes of Functions
Rates of Growth: Comparison
Upper Bound of Algorithm: Example (1)
Index (3)

Upper Bound of Algorithm: Example (2)

Upper Bound of Algorithm: Example (3)

Upper Bound of Algorithm: Example (4)

Upper Bound of Algorithm: Example (5)