Analysis of Algorithms
Chapter 11

Instructor: Scott Kristjanson
CMPT 125/125
SFU Burnaby, Fall 2013
Scope

Analysis of Algorithms:

- Efficiency goals
- The concept of algorithm analysis
- Big-Oh notation
- The concept of asymptotic complexity
- Comparing various growth functions
Algorithm Efficiency

The efficiency of an algorithm is usually expressed in terms of its use of CPU time.

The analysis of algorithms involves categorizing an algorithm in terms of efficiency.

An everyday example: washing dishes

- Suppose washing a dish takes 30 seconds and drying a dish takes an additional 30 seconds.
- Therefore, n dishes require n minutes to wash and dry.
Algorithm Efficiency

Now consider a less efficient approach that requires us to dry all previously washed dishes each time we wash another one.

Each dish takes 30 seconds to wash.

But because we get the dishes wet while washing,

- must dry the last dish once, the second last twice, etc.
- Dry time = \(30 + 2 \times 30 + 3 \times 30 + \ldots + (n-1) \times 30 + n \times 30\)
- \(= 30 \times (1 + 2 + 3 + \ldots + (n-1) + n)\)
- \(= n \times (30 \text{ seconds wash time}) + \sum_{i=1}^{n} (i \times 30)\)

\[\text{time (n dishes)} = 30n + \frac{30n(n+1)}{2}\]
\[= 15n^2 + 45n \text{ seconds}\]
Problem Size

For every algorithm we want to analyze, we need to define the size of the problem

The dishwashing problem has a size $n$

$n = \text{number of dishes to be washed/dried}$

For a search algorithm, the size of the problem is the size of the search pool

For a sorting algorithm, the size of the program is the number of elements to be sorted
Growth Functions

We must also decide *what* we are trying to efficiently optimize

- *time complexity* – CPU time
- *space complexity* – memory space

CPU time is generally the focus

A growth function shows the relationship between the size of the problem (n) and the value optimized (time)
Asymptotic Complexity

The growth function of the second dishwashing algorithm is

\[ t(n) = 15n^2 + 45n \]

It is not typically necessary to know the exact growth function for an algorithm.

We are mainly interested in the asymptotic complexity of an algorithm – the general nature of the algorithm as \( n \) increases.
Asymptotic Complexity

Asymptotic complexity is based on the **dominant term** of the growth function – the term that increases most quickly as \( n \) increases.

The dominant term for the second dishwashing algorithm is \( n^2 \):

<table>
<thead>
<tr>
<th>Number of dishes ( (n) )</th>
<th>( 15n^2 )</th>
<th>( 45n )</th>
<th>( 15n^2 + 45n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>90</td>
<td>150</td>
</tr>
<tr>
<td>5</td>
<td>375</td>
<td>225</td>
<td>600</td>
</tr>
<tr>
<td>10</td>
<td>1,500</td>
<td>450</td>
<td>1,950</td>
</tr>
<tr>
<td>100</td>
<td>150,000</td>
<td>4,500</td>
<td>154,500</td>
</tr>
<tr>
<td>1,000</td>
<td>15,000,000</td>
<td>45,000</td>
<td>15,045,000</td>
</tr>
<tr>
<td>10,000</td>
<td>1,500,000,000</td>
<td>450,000</td>
<td>1,500,450,000</td>
</tr>
<tr>
<td>100,000</td>
<td>150,000,000,000</td>
<td>4,500,000</td>
<td>150,004,500,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>15,000,000,000,000</td>
<td>45,000,000</td>
<td>15,000,045,000,000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>1,500,000,000,000,000</td>
<td>450,000,000</td>
<td>1,500,000,045,000,000</td>
</tr>
</tbody>
</table>
Big-Oh Notation

The coefficients and the lower order terms become increasingly less relevant as $n$ increases.

So we say that the algorithm is order $n^2$, which is written $O(n^2)$.

This is called *Big-Oh notation*.

There are various Big-Oh categories.

Two algorithms in the same category are generally considered to have the same efficiency, but that doesn't mean they have equal growth functions or behave exactly the same for all values of $n$. 
### Big-Oh Categories

Some sample growth functions and their Big-Oh categories:

<table>
<thead>
<tr>
<th>Growth Function</th>
<th>Order</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t(n) = 17 )</td>
<td>( O(1) )</td>
<td>constant</td>
</tr>
<tr>
<td>( t(n) = 3\log n )</td>
<td>( O(\log n) )</td>
<td>logarithmic</td>
</tr>
<tr>
<td>( t(n) = 20n - 4 )</td>
<td>( O(n) )</td>
<td>linear</td>
</tr>
<tr>
<td>( t(n) = 12n \log n + 100n )</td>
<td>( O(n \log n) )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>( t(n) = 3n^2 + 5n - 2 )</td>
<td>( O(n^2) )</td>
<td>quadratic</td>
</tr>
<tr>
<td>( t(n) = 8n^3 + 3n^2 )</td>
<td>( O(n^3) )</td>
<td>cubic</td>
</tr>
<tr>
<td>( t(n) = 2^n + 18n^2 + 3n )</td>
<td>( O(2^n) )</td>
<td>exponential</td>
</tr>
</tbody>
</table>
Comparing Growth Functions

You might think that faster processors would make efficient algorithms less important.

A faster CPU helps, but not relative to the dominant term. What happens if we increase our CPU speed by 10 times?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Max Problem Size Before Speedup</th>
<th>Max Problem Size After Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$n$</td>
<td>$s_1$</td>
<td>$10s_1$</td>
</tr>
<tr>
<td>B</td>
<td>$n^2$</td>
<td>$s_2$</td>
<td>$3.16s_2$</td>
</tr>
<tr>
<td>C</td>
<td>$n^3$</td>
<td>$s_3$</td>
<td>$2.15s_3$</td>
</tr>
<tr>
<td>D</td>
<td>$2^n$</td>
<td>$s_4$</td>
<td>$s_4 + 3.3$</td>
</tr>
</tbody>
</table>
Comparing Growth Functions

As n increases, the various growth functions diverge dramatically:

![Graph comparing growth functions](image-url)
Comparing Growth Functions

For large values of $n$, the difference is even more pronounced:
Analyzing Loop Execution

First determine the order of the body of the loop, then multiply that by the number of times the loop will execute.

```java
for (int count = 0; count < n; count++)
    // some sequence of O(1) steps
```

N loop executions times O(1) operations results in a O(n) efficiency.

Can write:

- CPU-time Complexity  = n * O(1) = O(n*1) = O(n)
Analyzing Loop Execution

Consider the following loop:

```java
count = 1;
while (count < n)
{
    count *= 2;
    // some sequence of O(1) steps
}
```

How often is the loop executed given the value of n? The loop is executed $\log_2 n$ times, so the loop is $O(\log n)$

CPU-Time Efficiency = $\log n \times O(1) = O(\log n)$
Analyzing Nested Loops

When loops are nested, we multiply the complexity of the outer loop by the complexity of the inner loop

```java
for (int count = 0; count < n; count++)
    for (int count2 = 0; count2 < n; count2++)
    {
        // some sequence of O(1) steps
    }
```

Both the inner and outer loops have complexity of $O(n)$

For Body has complexity of $O(1)$

CPU-Time Complexity $= O(n)*(O(n) * O(1))$

$= O(n) * (O(n * 1))$

$= O(n) * O(n)$

$= O(n^2)$

The overall efficiency is $O(n^2)$
Analyzing Method Calls

The body of a loop may contain a call to a method.
To determine the order of the loop body, the order of the method must be taken into account.
The overhead of the method call itself is generally ignored.
Interesting Problem from Microbiology

Predicting RNA Secondary Structure

• using Minimum Free Energy (MFE) Models

Problem Statement:

Given:

• an ordered sequence of RNA bases $S = (s_1, s_2, \ldots, s_n)$
• where $s_i$ is over the alphabet \{A, C, G, U\}
• and $s_1$ denotes the first base on the 5' end, $s_2$ the second, etc.,

Using Watson-Crick pairings: A-U, C-G, and wobble pair G-U

Find Secondary Structure $R$ such that:

• $R$ described by the set of pairs $i,j$ with $1 \leq i < j \leq n$
• The pair $i,j$ denotes that the base indexed $i$ is paired with base indexed $j$
• For all indexes from 1 to $n$, no index occurs in more than one pair
• Structure $R$ has minimum free energy (MFE) for all such structures
• MFE estimated as sum energies of the various loops and sub-structures
Example RNA Structures and their Complexity

- Left - a pseudoknot-free structure (weakly closed)
- Center - an H-Type pseudoknotted (ABAB) structure
- Right - a kissing hairpin (ABACBC)
Solve the Problem in Parallel

Search the various possible RNA foldings using search trees
Use Branch and Bound to cut off bad choices
Use Parallelism to search multiple branches at the same time on different CPUs
Key Things to take away:

Algorithm Analysis:

• Software must make efficient use of resources such as CPU and memory
• Algorithm Analysis is an important fundamental computer science topic
• The order of an algorithm is found by eliminating constants and all but the dominant term in the algorithm’s growth function
• When an algorithm is inefficient, a faster processor will not help
• Analyzing algorithms often focuses on analyzing loops
• Time complexity of a loop is found by multiplying the complexity of the loop body times the number of times the loop is executed.
• Time complexity for nested loops must multiply the inner loop complexity with the number of times through the outer loop
References: