Algorithm Analysis

with

Big Oh

Data Structures and Design with Java and JUnit
Chapter 12
©Rick Mercer
Algorithm Analysis

- Objectives
  - Analyze the efficiency of algorithms
  - Analyze a few classic algorithms
    - Linear Search, Binary Search, Selection Sort
  - Know the differences between $O(1)$, $O(n)$, $O(\log n)$, and $O(n^2)$
  - Visualize runtime differences with experiments
Algorithms continued

- Computer Scientists focus on problems such as
  - How fast do algorithms run
  - How much memory does the process require

- Example Applications
  - Make the Internet run faster
    - Pink-Degemark's routing algorithms
    - Gene Meyers determined the sequences of the Human genome using his whole genome shotgun algorithm
Analysis of Algorithms

- We have ways to compare algorithms
  - Generally, the larger the problem, the longer it takes the algorithm to complete
  - Sorting 100,000 elements can take much more time than sorting 1,000 elements
    - and more than 10 times longer
  - the variable $n$ suggests the "number of things"
  - If an algorithm requires $0.025n^2 + 0.012n + 0.0005$ seconds, just plug in a value for $n$
A Computational Model

- To summarize algorithm runtimes, we can use a computer independent model
  - instructions are executed sequentially
  - count all assignments, comparisons, and increments there is infinite memory
  - every simple instruction takes one unit of time
Simple Instructions

- Count the simple instructions
  - assignments have cost of 1
  - comparisons have a cost of 1
  - let's count all parts of the loop
    for (int j = 0; j < n; j++)
      • j=0 has a cost of 1, j<n executes n+1 times, and j++ executes n times for a total cost of 2n+2
  - each statement in the repeated part of a loop have have a cost equal to number of iterations
Examples

```
sum = 0;                  ->  1
sum = sum + next;         ->  1
Total Cost: 2

for (int i = 1; i <= n; i++)
  sum = sum++;            ->  1 + n+1 + n = 2n+2
Total Cost: 3n + 2

k = 0
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    k++;                     ->  n (2n+2) = 2n² +2n
Total Cost: 3n² + 4n + 3
```
## Total Cost of Sequential Search

<table>
<thead>
<tr>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int index = 0; index &lt; n; index++) -&gt; 2n + 2</td>
</tr>
<tr>
<td>if(searchID.equals(names[index])) -&gt; n</td>
</tr>
<tr>
<td>return index;</td>
</tr>
<tr>
<td>-&gt; 0 or 1</td>
</tr>
<tr>
<td>return -1 // if not found</td>
</tr>
<tr>
<td>-&gt; 0 or 1</td>
</tr>
</tbody>
</table>

Total cost = 3n+3
**Different Cases**

- The total cost of sequential search is $3n + 3$
  - But is it always exactly $3n + 3$ instructions?
  - The last assignment does not always execute
    - But does one assignment really matter?
  - How many times will the loop actually execute?
    - that depends
  - If searchID is found at index 0: _____ iterations
    - best case
  - If searchID is found at index n-1:_____ iterations
    - worst case
Typical Case of sequential (linear)

- The average describes the more typical case
- First, let the entire cost be simplified to n
  - Assume the target has a 50/50 chance of being in the array
    - n comparisons are made: worst-case occurs 1/2 the time
  - Assume if it's in a, it's as likely to be in one index as another

\[
\frac{1}{2}n + \frac{1}{2} \times \frac{n}{2} = \frac{n}{2} + \frac{n}{4} = \frac{3}{4}n
\]

Half the time it is n comparisons, the other half it is n/2 comparisons
- So the typical case is 3/4 n comparisons
The Essence of Linear Search

Plot the function *this is why sequential search is also called linear search*. As \( n \) increases, runtime forms a line.
Linear Search Continued

- This equation is a polynomial: $3n + 3$
- The fastest growing term is the high-order term
- The high order term (which could be $n^2$ or $n^3$), represents the most important part of the analysis function
- We refer to the rate of growth as the order of magnitude, which measure the rate of growth
Rate of Growth

Imagine two functions:
\[ f(n) = 100n \quad \quad g(n) = n^2 + n \]

- When \( n \) is small, which is the bigger function?
- When \( n \) is big, which is the bigger function?
- We can say: \( g(n) \) grows faster than \( f(n) \)
Rate of Growth, another view

Function growth and weight of terms as a percentage of all terms as $n$ increases for

$$f(n) = n^2 + 80n + 500$$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$</th>
<th>$n^2$</th>
<th>$80n$</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1,400</td>
<td>100 (7%)</td>
<td>800 (57%)</td>
<td>500 (36%)</td>
</tr>
<tr>
<td>100</td>
<td>18,500</td>
<td>10,000 (54%)</td>
<td>8,000 (43%)</td>
<td>500 (3%)</td>
</tr>
<tr>
<td>1000</td>
<td>1,080,500</td>
<td>1,000,000 (93%)</td>
<td>80,000 (7%)</td>
<td>500 (0%)</td>
</tr>
<tr>
<td>10000</td>
<td>100,800,500</td>
<td>100,000,000 (99%)</td>
<td>800,000 (1%)</td>
<td>500 (0%)</td>
</tr>
</tbody>
</table>

Conclusion: consider highest order term with the coefficient dropped, also drop all lower order terms.
Definition

- The *asymptotic growth* of an algorithm
  - describes the relative growth of an algorithm as $n$ gets very large
  - With speed and memory increases doubling every two years, the asymptotic efficiency *where $n$ is very large* is the thing to consider
  - There are many sorting algorithm that are "on the order of" $n^2$ (there are roughly $n \times n$ instructions executed)
  - Other algorithms are "on the order of" $n \times \log_2 n$
    - and this is a huge difference when $n$ is very large
Constant Function

- Some functions don't grow with n
  - If the sorting program initializes a few variables first, the time required does not change when n increases
  - These statements run in *constant time*
    - e.g. construct an empty List with capacity 20
  - The amount of time can be described as a constant function $f(n) = k$, where $k$ is a constant
  - It takes ~0.0003 seconds no matter the size of n
Big O

- Linear search is "on the order of n", which can be written as $O(n)$ to describe the upper bound on the number of operations.
- This is called big $O$ notation.
- Orders of magnitude:
  
  $O(1)$ constant (the size of $n$ has no effect)
  $O(n)$ linear
  $O(\log n)$ logarithmic
  $O(n \log n)$ no other way to say it, John K’s License plate
  $O(n^2)$ quadratic
  $O(n^3)$ cubic
  $O(2^n)$ exponential
Binary Search

- We'll see that binary search can be a more efficient algorithm for searching

  If the element in the middle is the target
  report target was found and the search is done
  if the key is smaller
  search the array to the left
  Otherwise
  search the array to the right

- This process repeats until the target is found or there is nothing left to search

- Each comparison narrows search by half
<table>
<thead>
<tr>
<th>Data</th>
<th>reference</th>
<th>loop 1</th>
<th>loop 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>a[0] ← left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carl</td>
<td>a[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debbie</td>
<td>a[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evan</td>
<td>a[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Froggie</td>
<td>a[4] ← mid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gene</td>
<td>a[5] ← left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harry</td>
<td>a[6] ← mid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Igor</td>
<td>a[7]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jose</td>
<td>a[8] ← right</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**How fast is Binary Search?**

- **Best case:** 1
- **Worst case:** when target is not in the array
- At each pass, the "live" portion of the array is narrowed to half the previous size.
- The series proceeds like this:
  - $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$
- Each term in the series represents one comparison. How long does it take to get to 1?
  - This will be the number of comparisons
**Binary Search (con.)**

- Could start at 1 and double until we get to n
  \[1, 2, 4, 8, 16, \ldots, k \geq n \quad \text{or} \quad 2^0, 2^1, 2^2, 2^3, 2^4, \ldots, 2^c \geq n\]
- The length of this series is \(c+1\)
- The question is
  2 to what power \(c\) is greater than or equal to \(n\)?
  - if \(n\) is 8, \(c\) is 3
  - if \(n\) is 1024, \(c\) is 10
  - if \(n\) is 16,777,216, \(c\) is 24
- Binary search runs \(O(\log n)\) *logarithmic*
Comparing $O(n)$ to $O(\log n)$

Rates of growth and logarithmic functions

<table>
<thead>
<tr>
<th>Power of 2</th>
<th>n</th>
<th>$\log_2 n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^4$</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>$2^8$</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>4,096</td>
<td>12</td>
</tr>
<tr>
<td>$2^{24}$</td>
<td>16,777,216</td>
<td>24</td>
</tr>
</tbody>
</table>
Graph Illustrating Relative Growth \( n, \log n, n^2 \)
Other logarithm examples

The guessing game:

- Guess a number from 1 to 100
  - try the middle, you could be right
  - if it is too high
    - check near middle of 1..49
  - if it is too low
    - check near middle of 51..100

- Should find the answer in a maximum of 7 tries
  - If 1..250, a maximum of $2^c \geq 250$, $c = 8$
  - If 1..500, a maximum of $2^c \geq 500$, $c = 9$
  - If 1..1000, a maximum of $2^c \geq 1000$, $c = 10$
Logarithmic Explosion

- Assuming an infinitely large piece of paper that can be cut in half, layered, and cut in half again as often as you wish.
  - How many times do you need to cut and layer until paper thickness reaches the moon?
  - Assumptions
    - paper is 0.002 inches thick
    - distance to moon is 240,000 miles
      - 240,000 * 5,280 feet per mile * 12 inches per foot = 152,060,000,000 inches to the moon
Examples of Logarithmic Explosion

- The number of bits required to store a binary number is logarithmic. *Add 1 bit to get much larger ints*
  - 8 bits stored 256 values \( \log_2 256 = 8 \)
  - \( \log 2,147,483,648 = 31 \)

- The inventor of chess asked the Emperor to be paid like this:
  - 1 grain of rice on the first square, 2 on the next, double grains on each successive square \( 2^{63} \)
Compare Sequential and Binary Search

- Output from CompareSearches.java (1995)

Search for 20000 objects

**Binary Search**
- #Comparisons: 267248
- Average: 13
- Run time: 20ms

**Sequential Search**
- #Comparisons: 200010000
- Average: 10000
- Run time: 9930ms

Difference in comparisons: 199742752
Difference in milliseconds: 9910
$O(n^2)$ quadratic

- $O(n^2)$ reads on the order of $n$ squared or quadratic
- When $n$ is small, rates of growth don’t matter
- Quadratic algorithms are greatly affected by increases in $n$
  - Consider the selection sort algorithm
    - Find the largest, $n-1$ times
Actual observed data for $O(n^2)$ sort

Time required to sort an array of size $n$

- Time in seconds
- $n$ in thousands
Two $O(n^2)$ algorithms

- Many known sorting algorithms are $O(n^2)$
- Given $n$ points, find the pair that are closest

  Compare $p_1$ with $p_2, p_3, p_4, p_5$  (4 comparisons)
  Compare $p_2$ with $p_3, p_4, p_5$           (3 comparisons)
  Compare $p_3$ with $p_4, p_5$               (2 comparisons)
  Compare $p_4$ with $p_5$                    (1 comparisons)

  - When $n$ is 5, make 10 comparisons
  - In general, #comparisons is

    \[
    \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}
    \]

  - highest order term is $n^2$, drop $\frac{1}{2}$ and runtime is $O(n^2)$
$O(n^3)$ algorithms

- Matrix Multiplication (naïve):

```java
for (int i = 0; i < m.length; i++) {
    for (int j = 0; j < m2.length - 1; j++) {
        for (int k = 0; k < m2.length; k++) {
            m[i][j] += m[i][k] * m2[k][j];
        }
    }
}
```
Big O and Style Guidelines

- Big O is similar to saying the runtime is less than or equal to Big O notation.
  - $O(f)$ is an upper bound

- Don't use constants or lower-order terms
  - These are no-nos for now (you will use coefficients in C Sc 345)
    - $O(n^2 + n)$ should be written $O(n^2)$
    - $O(5500n)$ should be written $O(n)$
    - $O(2.5n)$ should be written $O(n)$
Properties of Big-O

Summarizing two main properties

If \( f(n) \) is a sum of several terms, the one with the largest growth rate is kept, and all others omitted

If \( f(n) \) is a product of several factors, any constants (terms in the product that do not depend on \( n \)) are omitted – which means you can drop coefficients
Properties of Big-O

We can drop coefficient

Example:

\[ f(n) = 100*n \]

then \( f(n) \) is \( O(n) \)
**Summation of same Orders**

The property is useful when an algorithm contains several loops of the same order

**Example:**

\[ f(n) \text{ is } O(n) \]
\[ f_2(n) \text{ is } O(n) \]
\[ \text{then } f(n) + f_2(n) \text{ is } O(n) + O(n), \text{ which is } O(n) \]
**Summation of different Orders**

This property works because we are only concerned with the term of highest growth rate

Example:

\[ f_1(n) \text{ is } O(n^2) \]
\[ f_2(n) \text{ is } O(n) \]

so \( f_1(n) + f_2(n) = n^2 + n \text{ is } O(n^2) \)
Product

This property is useful for analyzing segments of an algorithm with nested loops

Example:

$f_1(n)$ is $O(n^2)$

$f_2(n)$ is $O(n)$

then $f_1(n) \times f_2(n)$ is $O(n^2) \times O(n)$, which is $O(n^3)$
Limitations of Big-Oh Analysis

- Constants sometimes make a difference
  - \( n \log n \) may be faster than \( 10000n \)
  - Doesn't differentiate between data cache memory, main memory, and data on a disk--there is a huge time difference to access disk data
    - thousands of times slower
  - Worst case doesn't happen often
    - it's an overestimate
Quick Analysis

- Can be less detailed
- Running time of nested loops is
  - the product of each loop's number of iterations
- Several consecutive loops
  - the longest running loop
  - $3n$ is $O(n)$ after all
Runtimes with for loops

```java
int n = 1000;
int[] x = new int[n];

- O(n)
  for(int j = 0; j < n; j++)
    x[j] = 0;

- O(n²)
  int sum = 0;
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      sum += j * k;
```
Run times with for loops

- $O(n^3)$
  
  ```java
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      for (int l = 0; l < n; l++)
        sum += j * k * l;
  ```

- $O(n)$
  
  ```java
  for (int j = 0; j < n; j++)
    sum++;
  for (int j = 0; j < n; j++)
    sum--;;
  ```

- $O(\log n)$
  
  ```java
  for (int j = 1; j < n; j = 2 * j)
    sum += j;
  ```
Analyze this

```java
public void swap(int[] a, int left, int right) {
    int temp = array[left];
    array[left] = array[right];
    array[right] = temp;
}
```
Analyze that

```c
for (int j = 0; j < n; j++)
    sum += l;

for (int k = 0; k < n; k++)
    sum += l;

for (int l = 0; l < n; l++)
    sum += l;
```
Analyze that

for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
        sum += k + l;

for (int l = 0; l < n; l++)
    sum += l;
Analyze this

for (int top = 0; top < n - 1; top++) {
    int smallestIndex = top;
    for (int index = top; index < n; index++) {
        if (a[index] < a[smallestIndex]) {
            smallestIndex = index;
        }
    }
    // Swap smallest to the top index
    swap(a, top, smallestIndex);
}