Introduction to Computer Science
Program Analysis

Ryan Stansifer

Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

27 April 2012
Analysis of Algorithms

Two approaches

1. Analytical. Static analysis of the program. Requires program source. (Mathematical guarantees.)
2. Empirical. Time experiments running the program with different inputs. (Scientific method.)
Steps, Worse Case

The work a computer does can be measured in the number of individual instructions it executes. The work a program does can be approximated by the number of operations or steps it calls for—operations like assignment, IO, arithmetic operations and relational comparisons. The size of the steps—10 machine instructions, 100 machine instructions—does not matter. When counting the steps of a program we always assume the worse. We assume that the program will “choose” the path that requires the most steps. This way we get an upper bound on the performance.
Profiling

Measuring the time a program takes is difficult. Many factors influence the time: processor, OS, multitasking, input data, resolution of the clock, etc. It is difficult to compare the performance of a program based on timing experiments. (As we will see, one may be able to predict the performance of the same program in the same environment.) There is a better way using functions.
Useful programs take different steps depending on the input. So, the number of steps a program takes for some particular input does not tell us how good the program is. A bad algorithm may take few steps for some small, simple input; and a good algorithm may take many steps for some large, complex input. Suppose we count the number of steps in terms of the size of the input, call it $N$. The number of steps is a function of $N$. For the program which reads $N$ numbers in order to sum them, the number of steps might be $f(N) = 2N + 1$.

What is the size of the input? Most algorithms have a parameter $N$ that affects the running time most significantly. For example, $N$ might be the size of the file to be sorted or searched, the number of characters in a string, or some other abstract measure of the size on the data set being processed.

In the long run, little differences in the number of steps do not matter, so we group functions together in orders of magnitude.
The number of steps a program takes is a \textit{function} of the size of the input.
(Different results are obtained if the size of input is measured differently. Eg., one integer unit, or $n$ bits representing an integer.)
Asymptotic Notation

We wish to compare functions carefully by their growth. Unimportant information should be ignored, like “rounding” where $1,000,001 \approx 1,000,000$.

And we want the “big picture.” This means that a function $f$ may be smaller than a function $g$ for some particular values, but “in the long run” it may be larger than $g$.

Fortunately, a precise definition that captures our intuition (most of the time) is possible.
Let $f(n)$ and $g(n)$ be functions mapping nonnegative numbers to nonnegative numbers.

*Big-Oh.* $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$. 

*Big-Omega.* $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every number $n \geq n_0$. (One does not get to pick the constant.)

*Big-Theta.* $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $\Omega(f(n))$.

*Little-Oh.* $f(n)$ is $o(g(n))$ if for any $c > 0$ there is a $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$. (One does not get to pick the constant.)

*Little-Omega.* $f(n)$ is $\Omega(g(n))$ if for any $c > 0$ there is a $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every number $n \geq n_0$. (One does not get to pick the constant.)
Big-Oh Notation

Let $f(n)$ and $g(n)$ be functions mapping nonnegative numbers to nonnegative numbers.

**Big-Oh.** $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

**Big-Omega.** $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every integer $n \geq n_0$.

**Big-Theta.** $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $\Omega(f(n))$.

**Little-Oh.** $f(n)$ is $o(g(n))$ if for any $c > 0$ there is $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$. (One does not get to pick the constant.)

**Little-Omega.** $f(n)$ is $\Omega(g(n))$ if for any $c > 0$ there is $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every number $n \geq n_0$. (One does not get to pick the constant.)
Big-Oh Notation

\[ f(n) \text{ is } O(g(n)) \text{ iff } \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty \]
The analogy is rough since some functions are not comparable, while any two real numbers are comparable.
Big-Oh

Graph showing the relationship between running time and input size for functions $c g(n)$ and $f(n)$ with a threshold at $n_0$. The graph illustrates how the running time increases with input size.
Example

The function \( f(n) = 3 \cdot n + 17 \) is \( O(n) \). (Here \( g(n) = n \).)  
Proof. Take \( c = 4 \) and \( n_0 = 17 \). Then \( f(n) = 3 \cdot n + 17 \leq c \cdot g(n) \) for every \( n \geq n_0 \). because \( 3 \cdot n + 17 \leq 4 \cdot n = 3 \cdot n + n \) for every \( n \geq 17 \).
Using the Big-Oh Notation

The notation is very bad. It is difficult to use. [The language of mathematics has (and this is quite amazing) dealt very poorly with functions. Church’s lambda notation is not widely used.] The idea is simple: a function gives rise to a collection of functions containing that function and other functions.

It is best to write

\[ f(n) \text{ is } O(g(n)) \]

Some authors write \( f(n) \in O(g(n)) \). or even \( f(n) = O(g(n)) \), but I find that misleading.
Big-Oh Math

If \( d(n) \) is \( O(f(n)) \), then \( a \times d(n) \) is \( O(f(n)) \), for any constant \( a > 0 \). Just take \( c = a \times c_1 \).
Fact: $f(n) = n$ is $O(2^n)$ because, by induction, $n < 2^n$ for all $n$.

Another fact: $2^{n+4} = 2^4 \times 2^n < (2^4 + 1) \times 2^n$, so take $c = 2^4 + 1$ and therefore, $2^{n+4}$ is $O(2^n)$.

Another fact: If $f(n)$ and $h(n)$ are $O(g(n))$, then $f(n) + h(n)$ is $O(g(n))$; just take $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$.

We have $f(n) = an^2 + bn + c$ is $O(n^2)$ by using the previous facts.

If $f(n)$ is a polynomial of degree $d$, then $f(n)$ is $O(n^d)$. 
| $O(1)$   | constant       | $f(n) = 45$ |
| $O(\log n)$ | logarithmic   | $f(n) = 2\log n + 4$ |
| $O(\sqrt{n})$ | “square root of n” | $f(n) = 3\sqrt{n} + \log \log n$ |
| $O(\log^2 n)$ | “log squared” | $f(n) = 2\log^2 n + 4 \log n$ |
| $O(n)$ | linear        | $f(n) = 3n + 87$ |
| $O(n \log n)$ | loglinear    | $f(n) = 2n \log n + 4n$ |
| $O(n^2)$ | quadratic     | $f(n) = 2n^2 + 5n + 62$ |
| $O(n^3)$ | cubic         | $f(n) = n^3 + 9n + 1$ |
| $O(2^n)$ | exponential   | $f(n) = 2^{n+4} + n^{19}$ |
| $O(3^n)$ | exponential   | $f(n) = 3^{5n} + n^{253}$ |
Important Categories of Functions

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$: loglinear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
Observation 1: You cannot make an inefficient algorithm efficient by how you choose to implement it or what machine you choose to run it on.

Observation 2: It is virtually impossible to ruin the efficiency of an efficient algorithm by how you implement it or what machine you run it on.

So, the efficiency is determined by the algorithms and data structures used in your solution. Efficiency is not significantly affected by how well or how poorly you implement the code.
Fast Growing Functions

*The order of an algorithm is generally more important than the speed of the processor.*
Fast growing functions grow really fast. Their growth is stupefying. Don’t be fooled.
Goodrich and Tamassia, Table 3.2, page 120.
Comparing Functions

In finding a name in phonebook, suppose every comparison takes one microsecond (0.001 sec).

<table>
<thead>
<tr>
<th>city</th>
<th>pop</th>
<th>linear</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port St. Lucie</td>
<td>164,603</td>
<td>2.8 min</td>
<td>0.017 sec</td>
</tr>
<tr>
<td>Fort Lauderdale</td>
<td>165,521</td>
<td>2.8 min</td>
<td>0.017 sec</td>
</tr>
<tr>
<td>Tallahassee</td>
<td>181,376</td>
<td>3.0 min</td>
<td>0.017 sec</td>
</tr>
<tr>
<td>Hialeah</td>
<td>224,669</td>
<td>3.7 min</td>
<td>0.018 sec</td>
</tr>
<tr>
<td>Orlando</td>
<td>238,300</td>
<td>4.0 min</td>
<td>0.018 sec</td>
</tr>
<tr>
<td>St. Petersburg</td>
<td>244,769</td>
<td>4.0 min</td>
<td>0.018 sec</td>
</tr>
<tr>
<td>Tampa</td>
<td>335,709</td>
<td>5.6 min</td>
<td>0.018 sec</td>
</tr>
<tr>
<td>Miami</td>
<td>399,457</td>
<td>6.7 min</td>
<td>0.019 sec</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>821,784</td>
<td>13.7 min</td>
<td>0.020 sec</td>
</tr>
</tbody>
</table>
Comparing Functions

In finding a name in phonebook, suppose every comparison takes one microsecond (0.001 sec).

<table>
<thead>
<tr>
<th>city</th>
<th>pop</th>
<th>linear</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dallas, TX</td>
<td>1,299,543</td>
<td>21.7 min</td>
<td>0.020 sec</td>
</tr>
<tr>
<td>San Diego, CA</td>
<td>1,306,301</td>
<td>21.8 min</td>
<td>0.020 sec</td>
</tr>
<tr>
<td>San Antonio, TX</td>
<td>1,373,668</td>
<td>22.9 min</td>
<td>0.020 sec</td>
</tr>
<tr>
<td>Philadelphia, PA</td>
<td>1,547,297</td>
<td>25.8 min</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>Phoenix, AZ</td>
<td>1,601,587</td>
<td>26.7 min</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>Houston, TX</td>
<td>2,257,926</td>
<td>37.6 min</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>Chicago, IL</td>
<td>2,851,268</td>
<td>47.5 min</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>Los Angeles, CA</td>
<td>3,831,868</td>
<td>63.9 min</td>
<td>0.022 sec</td>
</tr>
<tr>
<td>New York, NY</td>
<td>8,391,881</td>
<td>139.9 min</td>
<td>0.023 sec</td>
</tr>
</tbody>
</table>
Comparing Functions

In finding a name in phonebook, suppose every comparison takes one microsecond (0.001 sec).

<table>
<thead>
<tr>
<th>city</th>
<th>pop</th>
<th>linear</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seoul</td>
<td>10,575,447</td>
<td>2.9 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>São Paulo</td>
<td>11,244,369</td>
<td>3.1 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>Moscow</td>
<td>11,551,930</td>
<td>3.2 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>Beijing</td>
<td>11,716,000</td>
<td>3.3 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>Mumbai</td>
<td>12,478,447</td>
<td>3.5 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Delhi</td>
<td>12,565,901</td>
<td>3.5 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Istanbul</td>
<td>12,946,730</td>
<td>3.6 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Karachi</td>
<td>12,991,000</td>
<td>3.6 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Shanghai</td>
<td>17,836,133</td>
<td>5.0 hr</td>
<td>0.024 sec</td>
</tr>
</tbody>
</table>
## Fast Growing Functions

<table>
<thead>
<tr>
<th>log ( n )</th>
<th>( n )</th>
<th>( n \log n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>30</td>
<td>100</td>
<td>1,000</td>
<td>1,024</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>80</td>
<td>400</td>
<td>8,000</td>
<td>1,048,576</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>120</td>
<td>900</td>
<td>27,000</td>
<td>1,073,741,824</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>200</td>
<td>1,600</td>
<td>64,000</td>
<td>1,099,511,627,776</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>250</td>
<td>2,500</td>
<td>125,000</td>
<td>1,125,899,906,842,624</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>300</td>
<td>3,600</td>
<td>216,000</td>
<td>1.15 \times 10^{18}</td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>420</td>
<td>4,900</td>
<td>343,000</td>
<td>1.18 \times 10^{21}</td>
</tr>
<tr>
<td>6</td>
<td>80</td>
<td>480</td>
<td>6,400</td>
<td>512,000</td>
<td>1.21 \times 10^{24}</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>540</td>
<td>8,100</td>
<td>729,000</td>
<td>1.24 \times 10^{27}</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>600</td>
<td>10,000</td>
<td>1,000,000</td>
<td>1.27 \times 10^{30}</td>
</tr>
</tbody>
</table>
A problem is said to be intractable if the algorithm takes an impractical amount of time to find the solution. Roughly speaking, we consider polynomial algorithms to be tractable and exponential algorithms to be impractical.
Categorizing Programs

Compute $\sum_{i=1}^{n} i$

Algorithm 1 – $O(n)$

```java
final int n = Integer.parseInt(args[0]);
int sum = 0;
for (int count = 1; count <= n; i++) {
    sum += count;
}
```

Algorithm 2 – $O(1)$

```java
final int n = Integer.parseInt(args[0]);
int sum = (n*(n+1))/2;
```
Categorizing Programs

Compute \(\lceil \log n \rceil\)

Algorithm 1 – \(O(\log n)\)

```java
for (lgN=0; Math.pow(2, lgN) < n; lgN++);
```

Algorithm 2 – \(O(\log n)\)

```java
for (lgN=0; n > 0; lgN++, n /= 2);
```

Algorithm 3 – \(O(\log n)\)

```java
for (lgN=0, t=1; t < n; lgN++, t += t);
```
Give a Big-Oh analysis in terms of $N$ of the running time for each of the following program fragments:

```c
for (int i=1; i<N; i++) {
    sum ++;
}
```

$O(N)$

```c
for (int i=1; i<N; i+=2) {
    sum ++;
}
```

$O(N/2) = O(N)$
Give a Big-Oh analysis in terms of $N$ of the running time for each of the following program fragments:

```c
for (int i=1; i<N; i++) {
    sum ++;
}
```

$O(N)$

```c
for (int i=1; i<N; i+=2) {
    sum ++;
}
```

$O(N/2) = O(N)$
Give a Big-Oh analysis in terms of $N$ of the running time for each of the following program fragments:

```java
for (int i=1; i<N; i++) {
    sum++;
}

$O(N)$

```java
for (int i=1; i<N; i+=2) {
    sum++;
}

$O(N/2) = O(N)$
for (int i=1; i<N; i++) {
    for (int j=1; j<N; j++) {
        sum++;
    }
}

$O(N^2)$
for (int i=1; i<N; i++) {
    for (int j=1; j<N; j++) {
        sum++;
    }
}

$O(N^2)$
for (int i=1; i<10; i++) {
    \( O(N) \) steps in loop
}
for (int i=1; i<10; i++) {
    \(O(N)\) steps in loop
}

\(O(10N) = O(N)\)
for (int i=1; i<=N; i++) {
    sum++;
}
for (int j=1; j<=N; j++) {
    sum++;
}
for (int i=1; i<N; i++) {
    sum ++;
}
for (int j=1; j<N; j++) {
    sum ++;
}
$2O(N) = O(N)$
for (int i=1; i<N; i++) {
    sum ++;
}
for (int j=1; j<N; j++) {
    sum ++;
}

\[ 2O(N) = O(N) \]

for (int i=1; i<N; i++) {
    for (int j=1; j<N*N; j++) {
        for (int k=1; k<j; k++) {
            sum ++;
        }
    }
}
for (int i=1; i<N; i++) {
    sum ++;
}
for (int j=1; j<N; j++) {
    sum ++;
}

2O(N) = O(N)

for (int i=1; i<N; i++) {
    for (int j=1; j<N*N; j++) {
        for (int k=1; k<j; k++) {
            sum ++;
        }
    }
}

N × N² × Σ_{k=1}^{N²} k = O(N³ × (N² × (N² - 1)/2)) = O(N⁵)
for (int i=1; i<N; i*=2) {
    sum++;
}

while (N>1) {
    N = N/2;
    /* O(1) */
}
for (int i=1; i<N; i*=2) {
    sum++;
}

while (N>1) {
    N = N/2;
    /* O(1) */
}

O(log N)
public static void g (int N) {
    if (N==0) return;
    g (N/2);
    g (N/2);
    g (N/2);
    for (int i=0; i<N; i++) {
        /* O(1) */
    }
}
public static void g (int N) {
    if (N==0) return;
    g (N/2);
    g (N/2);
    g (N/2);
    for (int i=0; i<N; i++) {
        /* O(1) */
    }
}

O(NlogN)
public static void f (int N) {
    if (N == 0) return;
    f (N - 1);
    f (N - 1);
    /* O(1) */
}
public static void f (int N) {
    if (N==0) return;
    f (N-1);
    f (N-1);
    /* O(1) */
}
Math Review

$$\log_b a = c \quad \text{if} \quad a = b^c$$

Nearly always we want the base to be 2.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Lots of discrete steps

- $$[x]$$ the largest integer less than or equal to $$x$$.
- $$\lfloor x \rfloor$$ the smallest integer less than or equal to $$x$$.
Problems

Minimum Element in an Array. Given an array of $N$ items, find the smallest item.

Closest Points in the Plane. Given $N$ points in a plane, find the pair of points that are closest together.

Co-linear Points in the Plane. Given $N$ points in a plane, determine if any three form a straight line.
Prefix Averages

oh/PrefixAverages.java Java program
Maximum Contiguous Subsequence Sum Problem. Given (possibly negative) integers $a_1, a_2, \ldots, a_n$, find (and identify the sequence corresponding to) the maximum value of $\sum_{k=i}^{j} a_k$. The maximum contiguous subsequence sum is zero if all the integers are negative.

For example, if the input is $\{-2, 11, -4, 13, -5, 2\}$, then the answer is 20 which corresponds to the contiguous subsequence encompassing elements 2 through 4.

Weiss, Section 5.3, page 153.
Maximum Contiguous Subsequence Sum

The obvious $O(n^3)$ algorithm: for every potential staring element of the subsequence, and for every potential ending element of the subsequence, find the one with the maximum sum.
Maximum Contiguous Subsequence Sum

Since \( \sum_{k=i}^{j+1} a_k = (\sum_{k=i}^{j} a_k) + a_{j+1} \), the sum of the subsequence \( a_i, a_{i+1}, \ldots, a_{j+1} \) can be computed easily (without a loop) from the sum of \( a_i, a_{i+1}, \ldots, a_j \).
Maximum Contiguous Subsequence Sum

Theorem. Let $a_k$ for $i \leq k \leq j$ be any subsequence with $\sum_{k=i}^{j} a_k < 0$. If $q > j$, then $a_k$ for $i \leq k \leq q$ is not a maximum contiguous subsequence.
Proof. The sum of the subsequence $a_k$ for $j + 1 \leq k \leq q$ is larger.
Maximum Contiguous Subsequence Sum

oh/MaxSubsequenceSum.java Java program