Lecture 3:
Program Analysis

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Problem of the Day

For each of the following pairs of functions $f(n)$ and $g(n)$, state whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$, or none of the above.

1. $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7$
2. $f(n) = n\sqrt{n}$, $g(n) = n^2 - n$
3. $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2$

Big Oh Multiplication by Constant

Multiplication by a constant does not change the asymptotics:

$$O(c \cdot f(n)) \to O(f(n))$$

$$\Omega(c \cdot f(n)) \to \Omega(f(n))$$

$$\Theta(c \cdot f(n)) \to \Theta(f(n))$$

The "old constant" C from the Big Oh becomes $c \cdot C$.

Big Oh Multiplication by Function

But when both functions in a product are increasing, both are important:

$$O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$$

$$\Omega(f(n)) \cdot \Omega(g(n)) \rightarrow \Omega(f(n) \cdot g(n))$$

$$\Theta(f(n)) \cdot \Theta(g(n)) \rightarrow \Theta(f(n) \cdot g(n))$$

This is why the running time of two nested loops is $O(n^2)$.

Reasoning About Efficiency

Grossly reasoning about the running time of an algorithm is usually easy given a precise-enough written description of the algorithm.

When you *really* understand an algorithm, this analysis can be done in your head. However, recognize there is always implicitly a written algorithm/program we are reasoning about.

Selection Sort

```
selection_sort(int s[], int n)
{
        int i,j;
        int min;

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

Worst Case Analysis

The outer loop goes around n times.
The inner loop goes around at most n times for each iteration of the outer loop
Thus selection sort takes at most $n \times n \rightarrow O(n^2)$ time in the worst case.
In fact, it is $\Theta(n^2)$, because at least $n/2$ times it scans through at least $n/2$ elements, for a total of at least $n^2/4$ steps.

More Careful Analysis

An exact count of the number of times the *if* statement is executed is given by:

$$S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n - i + 1) = \sum_{i=0}^{n-1} i$$

$$S(n) = (n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = n(n+1)/2$$

Thus the worst case running time is $\Theta(n^2)$.

Insertion Sort

```
insertion_sort(item s[], int n)
{
        int i,j; /* counters */

        for (i=1; i < n; i++) {
                j=i;
                while ((j > 0) && (s[j] < s[j-1])) {
                        swap(&s[j],&s[j-1]);
                        j = j-1;
                }
        }
}
```

```
I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T O N S O R T
E I I N O R S T N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T
E I I N N O O R R S S T T
E I I N N O O R R S S T T
```

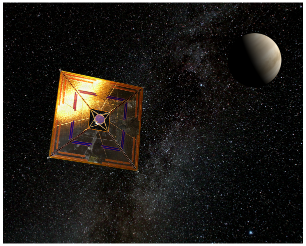
This involves a while loop, instead of just for loops, so the analysis is less mechanical.

But n calls to an inner loop which takes at most n steps on each call is $O(n^2)$.

The reverse-sorted permutation proves that the worst-case complexity for insertion sort is $\Theta(n^2)$.

$$(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$$

Solar Sails vs. Rockets



The bad-ass rocket hits a high speed before it runs out of fuel, then coasts at constant speed v_r.

The solar sail slowly accelerates from the force of radiation/solar wind hitting it, but its speed of $v_s = at$ must eventually exceed the bad-ass rocket.

This is asymptotic dominance in action.

Asymptotic Dominance in Action

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Implications of Dominance

- Exponential algorithms get hopeless fast.
- Quadratic algorithms get hopeless at or before 1,000,000.
- $O(n \log n)$ is possible to about one billion.
- $O(\log n)$ never sweats.

Testing Dominance

$f(n)$ dominates $g(n)$ if $\lim_{n \to \infty} g(n)/f(n) = 0$, which is the same as saying $g(n) = o(f(n))$.

Note the little-oh – it means "grows strictly slower than".

Properties of Dominance

- n^a dominates n^b if $a > b$ since

$$\lim_{n\to\infty} n^b/n^a = n^{b-a} \to 0$$

- $n^a + o(n^a)$ doesn't dominate n^a since

$$\lim_{n\to\infty} n^a/(n^a + o(n^a)) \to 1$$

Dominance Rankings

You must come to accept the dominance ranking of the basic functions:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

Advanced Dominance Rankings

Additional functions arise in more sophisticated analysis than we will do in this course:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n / \log \log n \gg \log \log n \gg \alpha(n) \gg 1$$

Logarithms

It is important to understand deep in your bones what logarithms are and where they come from.

A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.

Logarithms reflect how many times we can double something until we get to n, or halve something until we get to 1.

Binary Search

In binary search we throw away half the possible number of keys after each comparison. Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!

How many time can we halve n before getting to 1?

Answer: $\lceil \lg n \rceil$.

Logarithms and Trees

How tall a binary tree do we need until we have n leaves?
The number of potential leaves doubles with each level.
How many times can we double 1 until we get to n?
Answer: $\lceil \lg n \rceil$.

Logarithms and Bits

How many bits do you need to represent the numbers from 0 to $2^i - 1$?

Each bit you add doubles the possible number of bit patterns, so the number of bits equals $\lg(2^i) = i$.

Logarithms and Multiplication

Recall that

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

This is how people used to multiply before calculators, and remains useful for analysis.
What if $x = a$?

The Base is not Asymptotically Important

Recall the definition, $c^{\log_c x} = x$ and that

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Thus $\log_2 n = (1/\log_{100} 2) \times \log_{100} n$. Since $1/\log_{100} 2 = 6.643$ is just a constant, it does not matter in the Big Oh.

Federal Sentencing Guidelines

2F1.1. Fraud and Deceit; Forgery; Offenses Involving Altered or Counterfeit Instruments other than Counterfeit Bearer Obligations of the United States.
(a) Base offense Level: 6
(b) Specific offense Characteristics

(1) If the loss exceeded $2,000, increase the offense level as follows:

Loss(Apply the Greatest)	Increase in Level
(A) $2,000 or less	no increase
(B) More than $2,000	add 1
(C) More than $5,000	add 2
(D) More than $10,000	add 3
(E) More than $20,000	add 4
(F) More than $40,000	add 5
(G) More than $70,000	add 6
(H) More than $120,000	add 7
(I) More than $200,000	add 8
(J) More than $350,000	add 9
(K) More than $500,000	add 10
(L) More than $800,000	add 11
(M) More than $1,500,000	add 12
(N) More than $2,500,000	add 13
(O) More than $5,000,000	add 14
(P) More than $10,000,000	add 15
(Q) More than $20,000,000	add 16
(R) More than $40,000,000	add 17
(Q) More than $80,000,000	add 18

Make the Crime Worth the Time

The increase in punishment level grows *logarithmically* in the amount of money stolen.

Thus it pays to commit one big crime rather than many small crimes totalling the same amount.