Big-O
Analyzing Algorithms Asymptotically

CS 226
5 February 2002
Noah Smith (nasmith@cs)

Comparing Algorithms
Should we use Program 1 or Program 2?
Is Program 1 “fast”? “Fast enough”?

You and Igor: the empirical approach
Implement each candidate → Run it → Time it
That could be lots of work — also error prone.
Which inputs?
What machine/OS?

Toward an analytic approach ...
Today is just math: How to solve “which algorithm” problems without running tests, data, or Igor?

The Big Picture
Input (n = 3)
Input (n = 4)
Input (n = 8)
Algorithm
How long does it take for the algorithm to finish?

Primitives
- Primitive operations
  - x = 4 assignment
  - ... x + 5 ...
  - if (x < y) ...
  - x[4] index an array
  - *x dereference (C)
  - x.foo() calling a method
- Others
  - new/malloc memory usage
How many foos?

for (j = 1; j <= N; ++j) {
    foo();
}

\[ \sum_{j=1}^{N} 1 = N \]

How many foos?

for (j = 1; j <= N; ++j) {
    for (k = 1; k <= j; ++k) {
        foo();
    }
}

\[ \sum_{j=1}^{N} \sum_{k=1}^{j} 1 = \frac{N(N+1)}{2} \]

How many foos?

void foo(int N) {
    if(N <= 2)
        return;
    foo(N / 2);
}

The trick

\[ a^k + b^{k+1} + \ldots + \frac{y_n + z}{r^k} \]
\[ \approx \log_2 n \]
**Big O**

**Definition:** Let \( f \) and \( g \) be functions mapping \( \mathbb{N} \) to \( \mathbb{R} \). We say that \( f(n) \) is \( O(g(n)) \) if there exist \( c \in \mathbb{R} \), \( c > 0 \) and \( n_0 \in \mathbb{N} \), \( n_0 \geq 1 \) such that \( f(n) \leq cg(n) \) for all \( n \in \mathbb{N} \), \( n \geq n_0 \).
**Example 4**

Some complexity classes ...

<table>
<thead>
<tr>
<th>Class</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Cubic</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$O(n^r)$</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>

**Don’t be confused ...**

- We typically say, $f(n) = O(g(n))$ or $f(n) \leq c g(n)$.
- But $O(g(n))$ is really a set of functions.
- It might be more clear to say, $f(n) \in O(g(n))$.
- But I don’t make the rules.
- Crystal clear: “$f(n)$ is $\Theta(g(n))$”

**Intuitively ...**

- To say $f(n) = O(g(n))$ is to say that $f(n)$ is “less than or equal to” $g(n)$.
- We also have (G&T pp. 118-120):

<table>
<thead>
<tr>
<th>Sets</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(g(n))$</td>
<td>$f(n) \leq c g(n)$ for some $c &gt; 0$</td>
</tr>
<tr>
<td>$\Omega(g(n))$</td>
<td>$f(n) \geq c g(n)$ for all $n$ and some $c &gt; 0$</td>
</tr>
<tr>
<td>$\Theta(g(n))$</td>
<td>$f(n) \in O(g(n))$ and $g(n) \in O(f(n))$</td>
</tr>
</tbody>
</table>

**Big-Omega and Big-Theta**

$\Omega$ is just like $O$ except that $f(n) \geq c g(n)$.

- $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$.

$\Theta$ is both $O$ and $\Omega$ (and the constants need not match):

- $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n)) \iff f(n) \in \Theta(g(n))$.

**little o**

**Definition:** Let $f$ and $g$ be functions mapping $\mathbb{N}$ to $\mathbb{R}$. We say that $f(n) = o(g(n))$ if for any $c > 0$, there exists $n_0 \in \mathbb{N}$ such that $f(n) \leq c g(n)$ for all $n \geq n_0$.

G&T pp. 118-120.
Multiple variables

```
for(j = 1; j <= N; ++j)
    for(k = 1; k <= N; ++k)
        for(l = 1; l <= M; ++l)
            foo();
```

\(Q(N^2M + NM^2)\)

Multiple primitives

```
for(j = 1; j <= N; ++j) {
    sum += A[j];
    for(k = 1; k <= M; ++k) {
        sum2 += B[j][k];
        for(l = 1; l <= k; ++l)
            B[j][k] -= B[j][l];
    }
}
```

Tradeoffs: an example

![Diagram showing tradeoffs between different values of N and M.]

Another example

- I have a set of integers between 0 and 1,000,000.
- I need to store them, and I want O(1) lookup, insertion, and deletion.
- Constant time and constant space, right?

Big-O and Deceit

- Beware huge coefficients
- Beware key lower order terms
- Beware when \(n\) is “small”

Does it matter?

Let \(n = 1,000\), and 1 ms / operation.

<table>
<thead>
<tr>
<th>(n)</th>
<th>(n\log n)</th>
<th>(n^2)</th>
<th>(n^3)</th>
<th>(n^n)</th>
<th>(2^n)</th>
<th>(n = 1000, 1) ms/op</th>
<th>max time one day</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 sec</td>
<td>10 sec</td>
<td>17 min</td>
<td>12 days</td>
<td>32 yrs</td>
<td>1 sec</td>
<td>86,400,000</td>
</tr>
<tr>
<td>10</td>
<td>10 secs</td>
<td>3,943,234</td>
<td>9,295</td>
<td>442</td>
<td>96</td>
<td>10 secs</td>
<td>3,943,234</td>
</tr>
<tr>
<td>100</td>
<td>3.17 (\times) 10^9 years</td>
<td>3.17 (\times) 10^9 years</td>
<td>3.17 (\times) 10^9 years</td>
<td>3.17 (\times) 10^9 years</td>
<td>3.17 (\times) 10^9 years</td>
<td>3.17 (\times) 10^9 years</td>
<td>3.17 (\times) 10^9 years</td>
</tr>
<tr>
<td>1,000</td>
<td>1.07 (\times) 10^{30} years</td>
<td>1.07 (\times) 10^{30} years</td>
<td>1.07 (\times) 10^{30} years</td>
<td>1.07 (\times) 10^{30} years</td>
<td>1.07 (\times) 10^{30} years</td>
<td>1.07 (\times) 10^{30} years</td>
<td>1.07 (\times) 10^{30} years</td>
</tr>
</tbody>
</table>
Worst, best, and average

Gideon is a fast runner
- ... up hills.
- ... down hills.
- ... on flat ground.

Gideon is the fastest swimmer
- ... on the JHU team.
- ... in molasses.
- ... in our research lab.
- ... in 5-yard race.
- ... on Tuesdays.
- ... in an average race.

What’s average?

- Strictly speaking, average (mean) is relative to some probability distribution.

\[
\text{mean}(X) = \sum_x \Pr(X) \times x
\]

- Unless you have some notion of a probability distribution over test cases, it’s hard to talk about average requirements.

Now you know ...

- How to analyze the run-time (or space requirements) of a piece of pseudo-code.
- Some new uses for Greek letters.
- Why the order of an algorithm matters.
- How to avoid some pitfalls.