CSE373: Data Structures and Algorithms
Lecture 3: Math Review; Algorithm Analysis

Catie Baker
Spring 2015
Today

• Registration should be done.
• Homework 1 due 11:59pm next Wednesday, April 8th.

• Review math essential to algorithm analysis
  – Proof by induction (another example)
  – Exponents and logarithms
  – Floor and ceiling functions

• Begin algorithm analysis
Homework 1 Clarifications

- You should have `numOracles` Queues for the questions to be added to (if the number of answers changes, so should the number of oracles)
- The Oracles’ answers are stored in the `answers` array and when you dequeue a question from Oracle\(_0\), you can get the answer to the question from `answers[0]`
Mathematical induction

Suppose $P(n)$ is some statement (mentioning integer $n$)

Example: $n \geq n/2 + 1$

We can use induction to prove $P(n)$ for all integers $n \geq n_0$.

We need to

1. Prove the “base case” i.e. $P(n_0)$. For us $n_0$ is usually 1.
2. Assume the statement holds for $P(k)$.
3. Prove the “inductive case” i.e. if $P(k)$ is true, then $P(k+1)$ is true.

Why we care:

To show an algorithm is correct or has a certain running time

no matter how big a data structure or input value is

(Our “$n$” will be the data structure or input size.)
Example

\( P(n) = n \geq n/2 + 1 \)

We will show that \( P(n) \) holds for all \( n \geq 2 \)

Proof: By induction on \( n \)

- Base case: \( n=2 \). \( 2 \geq 2/2 + 1 \)
  
  \[
  2 \geq 1+1
  \]
  
  \[
  2 \geq 2
  \]
Example

\[ P(n) = n \geq n/2 + 1, \ n \geq 2 \]

- Inductive case:
  - Assume \( P(k) \) is true i.e. \( k \geq k/2 + 1 \)
  - Show \( P(k+1) \) is true i.e. \( k+1 \geq (k+1)/2 + 1 \)

Using our assumption, we know \( k \geq k/2 + 1 \) so:

\[
\begin{align*}
  k+1 &\geq (k/2 + 1) + 1 \\
  k+1 &\geq k/2 + 2 \\
  k+1 &\geq k/2 + 2 \geq (k+1)/2 + 1^* \\
  k+1 &\geq (k+1)/2 + 1
\end{align*}
\]

\( *\) \((k+1)/2 + 1 = k/2 + 1.5\)

Success!
Logarithms and Exponents

- Definition: $x = 2^y$ if $\log_2 x = y$
  - $8 = 2^3$, so $\log_2 8 = 3$
  - $65536 = 2^{16}$, so $\log_2 65536 = 16$

- The exponent of a number says how many times to use the number in a multiplication. e.g. $2^3 = 2 \times 2 \times 2 = 8$
  (2 is used 3 times in a multiplication to get 8)

- A logarithm says how many of one number to multiply to get another number. It asks "what exponent produced this?" e.g. $\log_2 8 = 3$ (2 makes 8 when used 3 times in a multiplication)
Logarithms and Exponents

• Definition: \( x = 2^y \) if \( \log_2 x = y \)
  - \( 8 = 2^3 \), so \( \log_2 8 = 3 \)
  - \( 65536 = 2^{16} \), so \( \log_2 65536 = 16 \)

• Since so much is binary in CS, \( \log \) almost always means \( \log_2 \)
• \( \log_2 n \) tells you how many bits needed to represent \( n \) combinations.
• So, \( \log_2 1,000,000 = \) “a little under 20”

• Logarithms and exponents are inverse functions. Just as exponents grow very quickly, logarithms grow very slowly.
Logarithms and Exponents

See Excel file for plot data – play with it!
Logarithms and Exponents

See Excel file for plot data – play with it!
Logarithms and Exponents

See Excel file for plot data – play with it!
Logarithms and Exponents

See Excel file for plot data – play with it!
Properties of logarithms

• \( \log(A \times B) = \log A + \log B \)

• \( \log(N^k) = k \log N \)

• \( \log(A/B) = \log A - \log B \)

• \( \log(\log x) \) is written \( \log \log x \)
  – Grows as slowly as \( 2^y \) grows quickly

• \( (\log x)(\log x) \) is written \( \log^2 x \)
  – It is greater than \( \log x \) for all \( x > 2 \)
  – It is not the same as \( \log \log x \)
Log base doesn’t matter much!

“Any base $B$ log is equivalent to base $2$ log within a constant factor”
– And we are about to stop worrying about constant factors!
– In particular, $\log_2 x = 3.22 \log_{10} x$
– In general we can convert log bases via a constant multiplier
– To convert from base $B$ to base $A$:
  $$\log_B x = \frac{\log_A x}{\log_A B}$$
The floor function is the largest integer less than or equal to $X$. For example, $\lfloor 2.7 \rfloor = 2$, $\lfloor -2.7 \rfloor = -3$, and $\lfloor 2 \rfloor = 2$.

The ceiling function is the smallest integer greater than or equal to $X$. For example, $\lceil 2.3 \rceil = 3$, $\lceil -2.3 \rceil = -2$, and $\lceil 2 \rceil = 2$. 

Floor and ceiling
Facts about floor and ceiling

1. $X - 1 < \lfloor X \rfloor \leq X$
2. $X \leq \lceil X \rceil < X + 1$
3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ if $n$ is an integer
Algorithm Analysis

As the “size” of an algorithm’s input grows (integer, length of array, size of queue, etc.), we want to know
- How much longer does the algorithm take to run? (time)
- How much more memory does the algorithm need? (space)

Because the curves we saw are so different, often care about only “which curve we are like”

Separate issue: Algorithm correctness – does it produce the right answer for all inputs
- Usually more important, naturally
Algorithm Analysis: A first example

- Consider the following program segment:
  
  ```plaintext
  x := 0;
  for i = 1 to n do
    for j = 1 to i do
      x := x + 1;
  ```

- What is the value of x at the end?

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 to 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1 to 2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1 to 3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>1 to 4</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>1 to n</td>
<td>?</td>
</tr>
</tbody>
</table>

Number of times x gets incremented is

\[
= 1 + 2 + 3 + \ldots + (n-1) + n
\]

\[
= n(n+1)/2
\]
Analyzing the loop

• Consider the following program segment:

```plaintext
x := 0;
for i = 1 to n do
    for j = 1 to i do
        x := x + 1;
```

• The total number of loop iterations is $n*(n+1)/2$
  – This is a very common loop structure, worth memorizing
  – This is proportional to $n^2$, and we say $O(n^2)$, “big-Oh of”
    • $n*(n+1)/2 = (n^2+ n)/2$
    • For large enough $n$, the lower order and constant terms are irrelevant, as are the assignment statements
  • See plot… $(n^2+ n)/2$ vs. just $n^2/2$
Lower-order terms don’t matter

\[ n^*(n+1)/2 \quad \text{vs. just } n^2/2 \]

We just say \( O(n^2) \)
Big-O: Common Names

$O(1)$  constant  (same as $O(k)$ for constant $k$)

$O(\log n)$  logarithmic

$O(n)$  linear

$O(n \log n)$  “$n \log n$”

$O(n^2)$  quadratic

$O(n^3)$  cubic

$O(n^k)$  polynomial  (where $k$ is any constant)

$O(k^n)$  exponential  (where $k$ is any constant $> 1$)

$O(n!)$  factorial

Note: “exponential” does not mean “grows really fast”, it means “grows at rate proportional to $k^n$ for some $k>1$”
Big-O running times

- For a processor capable of one million instructions per second

<table>
<thead>
<tr>
<th></th>
<th>( n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 1.5^n )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 10 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>( n = 30 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>( 10^{25} ) years</td>
</tr>
<tr>
<td>( n = 50 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 100 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>( 10^{17} ) years</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 1,000 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 10,000 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 100,000 )</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 1,000,000 )</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
Analyzing code

Basic operations take “some amount of” constant time
  – Arithmetic (fixed-width)
  – Assignment
  – Access one Java field or array index
  – Etc.

(This is an approximation of reality: a very useful “lie”.)

Consecutive statements: Sum of times
Conditionals: Time of test plus slower branch
Loops: Sum of iterations
Calls: Time of call’s body
Recursion: Solve recurrence equation (next lecture)
Analyzing code

1. Add up time for all parts of the algorithm
e.g. number of iterations = (n^2 + n)/2
2. Eliminate low-order terms i.e. eliminate n: (n^2)/2
3. Eliminate coefficients i.e. eliminate 1/2: (n^2)

Examples:
- $4n + 5 = O(n)$
- $0.5n \log n + 2n + 7 = O(n \log n)$
- $n^3 + 2^n + 3n = O(2^n)$
- $n \log (10n^2 )$
  - $n \log(10) + n \log(n^2) = O(n \log n)$
  - $n \log(10) + 2n \log(n) = O(n \log n)$
Try a Java sorting program

private static void bubbleSort(int[] intArray) {
    int n = intArray.length;
    int temp = 0;

    for(int i=0; i < n; i++){
        for(int j=1; j < (n-i); j++){
            if(intArray[j-1] > intArray[j]){  
                //swap the elements!
                temp = intArray[j-1];
                intArray[j-1] = intArray[j];
                intArray[j] = temp;
            }
        }  
    }
}