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

Linda Shapiro
Winter 2015
Today

• Registration should be done.
• Homework 1 due 11:59 pm next Wednesday, January 14
• Review math essential to algorithm analysis
  – Proof by induction (review example)
  – Exponents and logarithms
  – Floor and ceiling functions

• Begin algorithm analysis
**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.)
Review Example

\( P(n) = \) “the sum of the first \( n \) powers of 2 (starting at \( 2^0 \)) is \( 2^{n-1} \)”

\[
2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1.
\]

in other words: \( 1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1. \)
Review Example

\[ P(n) = \text{“the sum of the first } n \text{ powers of 2 (starting at } 2^0 \text{) is } 2^n-1\text{”} \]

We will show that \( P(n) \) holds for all \( n \geq 1 \)
Proof: By induction on \( n \)
• Base case: \( n=1 \). Sum of first 1 power of 2 is \( 2^0 \), which equals 1.
  And for \( n=1 \), \( 2^n-1 \) equals 1.
**Review Example**

\[ P(n) = \text{“the sum of the first } n \text{ powers of 2 (starting at } 2^0) \text{ is } 2^n - 1 \text{”} \]

- **Inductive case:**
  - Assume \( P(k) \) is true i.e. the sum of the first \( k \) powers of 2 is \( 2^k - 1 \)
  - Show \( P(k+1) \) is true i.e. the sum of the first \( (k+1) \) powers of 2 is \( 2^{k+1} - 1 \)

Using our assumption, we know the first \( k \) powers of 2 is
\[
2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = 2^k - 1
\]

Add the next power of 2 to both sides…
\[
2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} + 2^k = 2^k - 1 + 2^k
\]

We have what we want on the left; massage the right a bit:
\[
2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} + 2^k = 2(2^k) - 1 = 2^{k+1} - 1 \quad \text{Success!}
\]
Mathematical Preliminaries

• The following N slides contain basic mathematics needed for analyzing algorithms.

• You should actually know this stuff.

• Hang in there!
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. \( 2^3 = 2 \times 2 \times 2 = 8 \)
  \((2 \text{ is used } 3 \text{ times in a multiplication to get } 8)\)

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

The graph shows the comparison of different functions as $n$ increases:
- $n$
- $2^n$
- $\log n$
- $n^2$

As $n$ increases, $2^n$ grows much faster than $n$, $\log n$, and $n^2$.
Logarithms and Exponents
Logarithms and Exponents
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 $A$:
    \[ \log_B x = \frac{\log_A x}{\log_A B} \]

• I use this because my calculator doesn’t have $\log_2$. 
Floor and ceiling

\[ \lfloor X \rfloor \quad \text{Floor function: the largest integer } \leq X \]

\[ \lfloor 2.7 \rfloor = 2 \quad \lfloor -2.7 \rfloor = -3 \quad \lfloor 2 \rfloor = 2 \]

\[ \lceil X \rceil \quad \text{Ceiling function: the smallest integer } \geq X \]

\[ \lceil 2.3 \rceil = 3 \quad \lceil -2.3 \rceil = -2 \quad \lceil 2 \rceil = 2 \]
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>
</tbody>
</table>

...  

n 1 to n  ?

Number of times x gets incremented is

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

\[
= \frac{n(n+1)}{2}
\]
Analyzing the loop

• Consider the following program segment:

\[
x := 0;
\]

\[
\text{for } i = 1 \text{ to } n \text{ do}
\]

\[
\quad \text{for } j = 1 \text{ to } i \text{ do}
\]

\[
\quad \quad x := x + 1;
\]

• 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”
    • \(n(n+1)/2 = (n^2 + n)/2 = 1/2n^2 + 1/2n\)
    • 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

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

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 \(1/2\): \((n^2)\) : Result is \(O(n^2)\)

Examples:

- \(4n + 5\) = \(O(n)\)
- \(0.5n \log n + 2n + 7\) = \(O(n \log n)\)
- \(n^3 + 2^n + 3n\) = \(O(2^n)\)
- \(365\) = \(O(1)\)
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;
            }
        }
    }
}