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:

```
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</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>n(n+1)/2</td>
</tr>
</tbody>
</table>

Number of times x gets incremented is n(n+1)/2

Analyzing the loop

Consider the following program segment:

```
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 vs. just n^2/2

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 is 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>n</th>
<th>1 sec</th>
<th>1 sec</th>
<th>1 sec</th>
<th>1 sec</th>
<th>1 sec</th>
<th>1 sec</th>
<th>1 sec</th>
<th>1 sec</th>
<th>1 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = 20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = 30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = 40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = 50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = 500</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = 1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Analyzing code

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

Examples:
- \( 4n + 5 = O(n) \)
- \( 0.5n \log n + 2n + 7 = O(n \log n) \)
- \( n \log (10n^2) = O(n \log n) \)
- \( 2n \log (10n) = O(n \log n) \)

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
Recursion: Solve recurrence equation (next lecture)