CMPSC250
Lecture 3: Designing Faster Algorithms 2

Prof. John Wenskovitch
01/27/2016
Last Time

• Big O / Big Omega / Theta / Tilde notation
• Average Case vs. Worst Case
• Start of Three Sum Analysis
A Sample Program

A: int count = 0;
B: for (int i = 0; i < a.length; i++) {
C: for (int j = i+1; j < a.length; j++) {
D: for (int k = j+1; k < a.length; k++) {
E: if (a[i] + a[j] + a[k] == 0) {
    count++;

<table>
<thead>
<tr>
<th>Statement Block</th>
<th>Time</th>
<th>Frequency</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$t_0$</td>
<td>1</td>
<td>$t_0$</td>
</tr>
<tr>
<td>B</td>
<td>$t_1$</td>
<td>$n$</td>
<td>$t_1 * n$</td>
</tr>
<tr>
<td>C</td>
<td>$t_2$</td>
<td>$n^2/2 - n/2$</td>
<td>$t_2 * (n^2/2 - n/2)$</td>
</tr>
<tr>
<td>D</td>
<td>$t_3$</td>
<td>$n^3/6 - n^2/2 + n/3$</td>
<td>$t_3 * (n^3/6 - n^2/2 + n/3)$</td>
</tr>
<tr>
<td>E</td>
<td>$t_4$</td>
<td>$x$ (depends on input)</td>
<td>$t_4 * x$</td>
</tr>
</tbody>
</table>
# A Sample Program

<table>
<thead>
<tr>
<th>Statement Block</th>
<th>Time</th>
<th>Frequency</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>( t_0 )</td>
<td>1</td>
<td>( t_0 )</td>
</tr>
<tr>
<td>B</td>
<td>( t_1 )</td>
<td>( n )</td>
<td>( t_1 \times n )</td>
</tr>
<tr>
<td>C</td>
<td>( t_2 )</td>
<td>( n^2/2 - n/2 )</td>
<td>( t_2 \times (n^2/2 - n/2) )</td>
</tr>
<tr>
<td>D</td>
<td>( t_3 )</td>
<td>( n^3/6 - n^2/2 + n/3 )</td>
<td>( t_3 \times (n^3/6 - n^2/2 + n/3) )</td>
</tr>
<tr>
<td>E</td>
<td>( t_4 )</td>
<td>( x ) (depends on input)</td>
<td>( t_4 \times x )</td>
</tr>
</tbody>
</table>

- Grand total: \[ t_3 \times (n^3/6 - n^2/2 + n/3) \] \[ + \] \[ t_2 \times (n^2/2 - n/2) \] \[ + \] \[ t_1 \times n \] \[ + \] \[ t_4 \times x \] \[ + \] \[ t_0 \]

- Tilde approximation: \( \sim \frac{n^3}{6} \)

- Order of growth: \( O(n^3) \)
Quantitative Measurements

• Every time you run a program, you are performing a scientific experiment that relates the program to the natural world.

• Problem size = size of the input, value of an argument.

• Run time is generally insensitive to input values, input size matters more. (why?)

• Discard outliers. (why?)
Plotting Results
Plotting Results

The graph shows the running time $T(N)$ and the logarithm of the running time $\log(T(N))$ as a function of the problem size $N$. The graph includes a straight line of slope 3, indicating a logarithmic growth rate with respect to the problem size.
Improving Runtime

• The ThreeSum problem is $O(n^3)$. Can we do better?
  – Consider the TwoSum problem
  – We need $a[i] + a[j] = 0$
  – Sort the list of numbers ($O(n \times \log(n))$)
  – For each value $a[i]$, binary search for $-a[i]$, increment count if “$j” > i$ ($O(n \times \log(n))$)
Improving Runtime

• Fast algorithm for ThreeSum:
  – We need \(a[i] + a[j] + a[k] = 0\)
  – Sort the list of numbers \((O(n \times \log(n)))\)
  – For each \(a[i]\) \((O(n))\)
    For each \(a[j]\) after \(a[i]\) \((O(n^2))\)
      Binary search for \(-a[i] - a[j]\); increment count if “\(k\)” > \(j\) \((O(n^2 \times \log(n)))\)
Improving Runtime

![Graph showing the comparison between ThreeSum and ThreeSumFast algorithms in terms of array accesses (millions) vs. problem size (N). The graph illustrates the exponential growth of array accesses for ThreeSum as N increases, whereas ThreeSumFast maintains a more linear growth.]
Our Approach This Semester

• Implement and analyze a straightforward solution to a problem.
  – These are commonly called brute-force solutions.
• Examine a variety of algorithmic improvements designed to reduce the order of growth of the running time.
• Run experiments to validate the hypothesis that the new algorithms are faster.
Any Questions?