Last Time

• What is an algorithm?
• How do we compare algorithms?
• StupidSort vs QuickSort vs SleepSort
• Function growth rates and precision
Big O

- \( f(x) \in O(g(x)) \) iff there exists some constant \( c > 0 \) and some input size \( x_0 \) such that \( f(x) < c \times g(x) \) for all \( x > x_0 \).
Asymptotic Performance

• **Big O** – Upper bound on the asymptotic performance
• **Big Omega** – Lower bound on the asymptotic performance
• **Theta** – Exact bound (both upper and lower)

• We use Big O almost exclusively. *(why?)*
  – Theta is harder to show.
  – Lower bounds are typically harder to show than upper bounds
Tilde Approximation

• Practically speaking, Tilde is like Theta but without dropping the multiplicative constants
  – Ex: \( f(n) = 6n^2 + 18n - 50 \)
    • Theta\( (n^2) \) (order of growth)
    • Also O\( (n^2) \)
    • \(~6n^2\)

• Tilde approximation can be useful, but the coefficients are implementation-dependent.
Average Case vs. Worst Case

• Consider QuickSort again:
  – In the average case, the key we select will split the list in half for the next level. \( n \times \log(n) \).
  – In the worst case, the key we select will be the largest or smallest number, which won’t reduce the number of levels. \( n \times n = n^2 \)

• Which is more important?
A Sample Program

• **Input:** A file containing a list of integers
• **Output:** The number of triples that sum to 0

• Example:

  324110 -442472 626686 -157678 508681 123414
  -77867 155091 129801 287381 604242 686904

  -442472 + 155091 + 287381 = 0
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>$t_1 \times n$</td>
</tr>
<tr>
<td>C</td>
<td>$t_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$t_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$t_4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summations

• Triangular sum
  \[ 1 + 2 + 3 + \cdots + n = \frac{n(n-1)}{2} \]

• Adding a third level
  \[ (1 \times 2) + (2 \times 3) + \cdots + ((n-1) \times n) = \frac{n(n-1)(n-2)}{6} \]
  – “Tetrahedral sum” or “Triangular pyramidal sum”
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 \cdot n)</td>
</tr>
<tr>
<td>C</td>
<td>(t_2)</td>
<td>(n^2/2 - n/2)</td>
<td>(t_2 \cdot (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 \cdot (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 \cdot 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 illustrates the running time $T(N)$ of an algorithm as a function of the problem size $N$. The left graph shows $T(N)$ for problem sizes $1K$, $2K$, $4K$, and $8K$. The right graph plots the logarithm of the running time $\lg(T(N))$ against the logarithm of the problem size $\lg(N)$. The line on the right graph is a straight line of slope 3, indicating that the running time grows exponentially with 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 runtime comparison between ThreeSum and ThreeSumFast](image-url)
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?