CMPSC112
Lecture 17: Evaluating Algorithms

Prof. John Wenskovitch
03/02/2016
Last Time

• Rather than timing an algorithm, we can evaluate its runtime through counting primitive operations.
• We can use the bounds of loops to determine which complexity class / order of growth an algorithm belongs to.
• Big-O notation allows us to express this order of growth while limiting the details – gives us a ballpark figure.
Big-O Notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there exists a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$. 

[Diagram showing the relationship between running time and input size, with $f(n)$ and $cg(n)$ curves, and $n_0$ marked on the x-axis.]
Big-O Rules

• Ignore constant factors.
  – $3n$, $6n$, $0.5n$, and $10000000000000n$ are all $O(n)$.

• Ignore lower-order terms.
  – $3n + 6$ is still $O(n)$, $4n^3 + 8n - 6$ is still $O(n^3)$,
    $2n + 1000000000000000$ is still $O(n)$.

• Use the most specific order of growth.
  – $3n + 6$ is $O(n)$, but it’s also $O(n^2)$ and $O(2^n)$ for different $c$ and $n_0$ values.
Better Hardware is not a Solution

• Let’s say that we have three algorithms, which run at different rates, in different complexity classes.

<table>
<thead>
<tr>
<th>Running Time (n = 1μs)</th>
<th>Problem Size Solved in 1s</th>
<th>Problem Size Solved in 1m</th>
<th>Problem Size Solved in 1h</th>
</tr>
</thead>
<tbody>
<tr>
<td>400n</td>
<td>2,500</td>
<td>150,000</td>
<td>9,000,000</td>
</tr>
<tr>
<td>2n^2</td>
<td>707</td>
<td>5,477</td>
<td>42,426</td>
</tr>
<tr>
<td>2^n</td>
<td>20</td>
<td>26</td>
<td>32</td>
</tr>
</tbody>
</table>
Better Hardware is not a Solution

• Now, let’s say that we improve our hardware, so that our new machine is 256 times faster than our old machine.

<table>
<thead>
<tr>
<th>Running Time (n = ( \frac{1}{256} ) μs)</th>
<th>Problem Size Solved in 1s</th>
<th>Problem Size Solved in 1m</th>
<th>Problem Size Solved in 1h</th>
</tr>
</thead>
<tbody>
<tr>
<td>400n</td>
<td>640,000</td>
<td>38,400,000</td>
<td>2,304,000,000</td>
</tr>
<tr>
<td>( 2n^2 )</td>
<td>11,314</td>
<td>87,636</td>
<td>678,822</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>28</td>
<td>34</td>
<td>40</td>
</tr>
</tbody>
</table>
Better Hardware is not a Solution

• If we set our old program running speed to m, then our speedup with 256x better hardware is:

<table>
<thead>
<tr>
<th>Algorithm Running Time</th>
<th>New Maximum Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>400n</td>
<td>256m</td>
</tr>
<tr>
<td>2n^2</td>
<td>16m</td>
</tr>
<tr>
<td>2^n</td>
<td>m + 8</td>
</tr>
</tbody>
</table>
Let’s Analyze the arrayMax Function

double arrayMax(double[] data) {
    int n = data.length;
    double currentMax = data[0];
    for (int j = 1; j < n; j++) {
        if (data[j] > currentMax) {
            currentMax = data[j];
        } //if
    } //for
    return currentMax;
} //arrayMax
Let’s Analyze the `arrayMax` Function

• The n initialization, currentMax initialization, and return are all constant-time operations.
• The runtime of `arrayMax` is dominate by the `for` loop, which executes \( n - 1 \) times.
  – We conclude that the running time of `arrayMax` is \( O(n) \), because the if is computed on every iteration.
• What about the currentMax update inside of the if?
  – If the input sequence is in random order, the probability that the \( j^{th} \) element is the largest of the first \( j \) elements is \( \frac{1}{j} \).
  – Thus, the expected number of times that we will update the biggest is \( H_n = \sum_{j=1}^{n} \left( \frac{1}{j} \right) \), which is called the \( n^{th} \) Harmonic number.
  – \( H_n \) converges to \( O(\log(n)) \), so the number of times we expect to update currentMax is \( O(\log(n)) \).
Let’s Analyze the disjoint Function

```java
boolean disjoint(int[] A, int[] B, int[] C) {
    for (int a: A) {
        for (int b: B) {
            for (int c: C) {
                if ((a==b) && (b==c)) {
                    return false;
                } //if
            } //for
        } //for
    } //for
    return true;
} //disjoint
```
Let’s Analyze the disjoint Function

• If sets A, B, and C all have size $n$, then we iterate through the a loop $n$ times, the b loop $n$ times, and the c loop $n$ times.
  – Each of these loops are nested, so our running time is $O(n^3)$.

• How can we improve this performance?
Let’s Analyze the disjoint2 Function

```java
boolean disjoint2(int[] A, int[] B, int[] C) {
    for (int a: A) {
        for (int b: B) {
            if (a == b) {
                for (int c: C) {
                    if (a == c) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```
Let’s Analyze the disjoint2 Function

• The outer two `for` loops are quadratic, $O(n^2)$.
  – Assuming that each item in A and B is unique, there are at most $n$ times when items a and b are equal.
  – Thus, the innermost `for` loop over C only runs $n$ times. Our overall running time is $O(n^2) + O(n^2)$. 
Any Questions?