CSC 273 – Data Structures

Lecture 2- Efficiency of Algorithms

Why Efficient Code?

• Computers are faster, have larger memories
  – So why worry about efficient code?
• And … how do we measure efficiency?
Example – Sum of First $n$ Values

• Consider the problem of summing

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + n$$

How would we code this?

Approach A

```java
sum = 0;
for (i = 1;  i <= n;  i++)
    sum = sum + i;
```

Approach B

```java
sum = 0;
for (i = 1;  i <= n;  i++)
    for (j = 1;  j <= i; j++)
        sum = sum + 1;
```

Approach C

```java
Sum = n * (n+1) / 2
```
public static void main(String[] args) {
    long n = 10000;
    long sum = 0;

    // Algorithm A
    for (long i = 1;  i <= n; i++)
        sum = sum + i;
    System.out.println("Sum is " + sum);

    sum = 0;
    // Algorithm B
    for (long i = 1; i <= n; i++)
        for (long j = 1; j <= i; j++)
            sum = sum + 1;
    System.out.println("Sum is " + sum);

    sum = 0;
    // Algorithm C
    sum = n * (n + 1) / 2;
    System.out.println("Sum is " + sum);
}
What is “best”?  

- An algorithm has both time and space constraints – that is complexity  
  - Time complexity  
  - Space complexity  
- This study is called analysis of algorithms  

Counting Basic Operations  

- A basic operation of an algorithm  
  - The most significant contributor to its total time requirement  

Number of required basic operations  

<table>
<thead>
<tr>
<th></th>
<th>Algorithm A</th>
<th>Algorithm B</th>
<th>Algorithm C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additions</td>
<td>$n$</td>
<td>$n(n + 1)/2$</td>
<td>1</td>
</tr>
<tr>
<td>Multiplications</td>
<td>$n$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Divisions</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Total basic operations</td>
<td>$n$</td>
<td>$(n^2 + n)/2$</td>
<td>3</td>
</tr>
</tbody>
</table>
Counting Basic Operations

Number of basic operations required by the algorithm

Algorithm A: $n$ operations

Algorithm B: $(n^2 + n) / 2$ operations

Algorithm C: $3$ operations

Counting Basic Operations

Typical growth-rate functions evaluated at increasing values of $n$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log(n)$</th>
<th>$\log(n)$</th>
<th>$\log^2(n)$</th>
<th>$n$</th>
<th>$\log(n)$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$2^{10}$</td>
<td>$1$</td>
</tr>
<tr>
<td>$10^1$</td>
<td>1.0000</td>
<td>1</td>
<td>1.0000</td>
<td>100</td>
<td>2</td>
<td>$10^4$</td>
<td>$10^5$</td>
<td>$2^{100}$</td>
<td>$10^3$</td>
</tr>
<tr>
<td>$10^2$</td>
<td>2.0000</td>
<td>2.0000</td>
<td>4.0000</td>
<td>1000</td>
<td>3</td>
<td>$10^6$</td>
<td>$10^7$</td>
<td>$2^{1000}$</td>
<td>$10^{30}$</td>
</tr>
<tr>
<td>$10^3$</td>
<td>3.0000</td>
<td>3.0000</td>
<td>9.0000</td>
<td>10,000</td>
<td>4</td>
<td>$10^8$</td>
<td>$10^9$</td>
<td>$2^{10,000}$</td>
<td>$10^{300}$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>4.0000</td>
<td>4.0000</td>
<td>16.0000</td>
<td>100,000</td>
<td>5</td>
<td>$10^{10}$</td>
<td>$10^{11}$</td>
<td>$2^{100,000}$</td>
<td>$10^{3000}$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>5.0000</td>
<td>5.0000</td>
<td>25.0000</td>
<td>1,000,000</td>
<td>6</td>
<td>$10^{12}$</td>
<td>$10^{13}$</td>
<td>$2^{1,000,000}$</td>
<td>$10^{30000}$</td>
</tr>
<tr>
<td>$10^6$</td>
<td>6.0000</td>
<td>6.0000</td>
<td>36.0000</td>
<td>10,000,000</td>
<td>7</td>
<td>$10^{14}$</td>
<td>$10^{15}$</td>
<td>$2^{10,000,000}$</td>
<td>$10^{300000}$</td>
</tr>
</tbody>
</table>
Best, Worst, and Average Cases

• For some algorithms, execution time depends only on size of data set
• Other algorithms depend on the nature of the data itself
  – Here we seek to know best case, worst case, average case

Big Oh Notation

• A function f(n) is of order at most g(n)

• That is, f(n) is O(g(n))—if
  – A positive real number c and positive integer N exist …
  – Such that f(n) ≤ c × g(n) for all n ≥ N
  – That is, c × g(n) is an upper bound on f(n) when n is sufficiently large
**Big Oh Notation**

The following identities hold for Big Oh notation:

- \( O(k \cdot g(n)) = O(g(n)) \) for a constant \( k \)
- \( O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n)) \)
- \( O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n)) \)
- \( O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max\{g_1(n), g_2(n), \ldots, g_m(n)\}) \)
- \( O(\max\{g_1(n), g_2(n), \ldots, g_m(n)\}) = \max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n))) \)

By using these identities and ignoring smaller terms in a growth-rate function, you can usually find the order of an algorithm's time requirement with little effort. For example, if the growth-rate function is \( 4n^2 + 50n - 10 \),

\[
O(4n^2 + 50n - 10) = O(4n^2) \quad \text{by ignoring the smaller terms}
\]

\[
= O(n^2) \quad \text{by ignoring the constant multiplier}
\]
# Complexities of Program Constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consecutive program segments $S_1$, $S_2$, ..., $S_k$ whose growth-rate functions are $g_1$, ..., $g_k$, respectively</td>
<td>$\max(O(g_1), O(g_2), \ldots, O(g_k))$</td>
</tr>
<tr>
<td>An <code>if</code> statement that chooses between program segments $S_1$ and $S_2$ whose growth-rate functions are $g_1$ and $g_2$, respectively</td>
<td>$O(\text{condition}) + \max(O(g_1), O(g_2))$</td>
</tr>
<tr>
<td>A loop that iterates $m$ times and has a body whose growth-rate function is $g$</td>
<td>$m \times O(g(n))$</td>
</tr>
</tbody>
</table>

# Picturing Efficiency – $O(n)$

```
for i = 1 to n
    sum = sum + i
```
Picturing Efficiency – $O(n^2)$

for $i = 1$ to $n$
    for $j = 1$ to $i$
        sum = sum + 1

$i = 1$
$i = 2$
$i = 3$
$\ldots$
$i = n$

$1$ $2$ $3$ $\ldots$ $n$

$O(1 + 2 + \ldots + n) = O(n^2)$

Picturing Efficiency – $O(n^2)$

for $i = 1$ to $n$
    for $j = 1$ to $n$
        sum = sum + 1

$i = 1$
$i = 2$
$i = 3$
$\ldots$
$i = n$

$1$ $2$ $3$ $\ldots$ $n$

$O(n \times n) = O(n^2)$
Effect of Doubling the Problem Size

<table>
<thead>
<tr>
<th>Growth-Rate Function for Size $n$ Problems</th>
<th>Growth-Rate Function for Size $2n$ Problems</th>
<th>Effect on Time Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>$1 + \log n$</td>
<td>None</td>
</tr>
<tr>
<td>$\log n$</td>
<td>$2n$</td>
<td>Negligible</td>
</tr>
<tr>
<td>$n$</td>
<td>$2n \log n + 2n$</td>
<td>Doubles</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$(2n)^2$</td>
<td>Doubles and then adds $2n$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$(2n)^3$</td>
<td>Quadruples</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$2^{2n}$</td>
<td>Multiplies by 8</td>
</tr>
<tr>
<td>$2^n$</td>
<td></td>
<td>Squares</td>
</tr>
</tbody>
</table>

Time Required To Process One Million Items

<table>
<thead>
<tr>
<th>Growth-Rate Function $g$</th>
<th>$g(10^6)/10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log n$</td>
<td>0.0000199 seconds</td>
</tr>
<tr>
<td>$n$</td>
<td>1 second</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>19.9 seconds</td>
</tr>
<tr>
<td>$n^2$</td>
<td>11.6 days</td>
</tr>
<tr>
<td>$n^3$</td>
<td>31,709.8 years</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^{301,016}$ years</td>
</tr>
</tbody>
</table>

Rate is one million operations per second
Efficiency of Implementations of ADT Bag

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fixed-Size Array</th>
<th>Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(newEntry)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove(anEntry)</td>
<td>O(1), O(n), O(n)</td>
<td>O(1), O(n), O(n)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>getFrequencyOf(anEntry)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains(anEntry)</td>
<td>O(1), O(n), O(n)</td>
<td>O(1), O(n), O(n)</td>
</tr>
<tr>
<td>toArray()</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get Current Size()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>