CS 261 – Data Structures

Big-Oh Analysis: A Review
• How can we characterize the runtime or space usage of an algorithm?

• We want a method that:
  – doesn’t depend upon hardware used (e.g., PC, Mac, etc.)
  – the clock speed of your processor
  – what compiler you use
  – even what language you write in
Big-Oh: **Purpose**

• “Big-Oh” notation is used to provide time and space characterizations.

• For example, you might see the statement:
  – “Algorithm A runs in time $O(n)$”
  – Read $O(n)$ as “Big-Oh of $n$” or just “Oh of $n$”

• **Purpose of a big-Oh characterization:**
  Describes how execution time (or space usage) of an algorithm changes relative to a change in the input size
Algorithmic Analysis

- Suppose that algorithm $A$ processes $n$ data elements in time $T$.

- Algorithmic analysis attempts to estimate how $T$ is impacted by changes in $n$. In other words, $T$ is a function of $n$ when we use $A$.

- If $T$ linearly increases with $n$, then we say that $A$ runs in $O(n)$ time
A Simple Example

• Consider summing an array of $n$ integers.

```java
sum = 0;
for (i = 0; i < n; i++)
    sum += array[i];
return sum;
```

• Total running time: $c_1 + c_2 n + c_3$
  – But the constants $c_1, c_2, c_3$ depend on hardware, compiler, etc.

• What is the big-Oh runtime? (big-Oh ignores factors)

  $O(n)$ also known as linear time
A Simple Example

• Consider summing an array of $n$ integers.

```java
sum = 0;
for (i = 0; i < n; i++)
    sum += array[i];
return sum;
```

• Suppose to sum 10,000 elements takes 32 ms.
• How long to sum 20,000 elements?
• If the size doubles, the execution time doubles
• Consider the BubbleSort algorithm.
  – Let $n$ be the size of the input list to be sorted
  – Runtime is $O(n^2)$, also known as quadratic time

• Suppose size doubles, what happens to execution time?

• It goes up by a factor of 4. Why?
The Calculation

Remember $O(n^2)$ means that runtime is proportional to $n^2$.

So, the ratio of the big-Oh sizes should equal the ratio of the execution times

\[
\frac{n_1^2}{n_2^2} = \frac{t_1}{t_2}
\]

$t_i$ is time to run size $n_i$ input

So if $n_2 = 2n_1$ (that is, double input size)

\[
\frac{n_1^2}{(2n_1)^2} = \frac{t_1}{t_2}
\]

then solve for $t_2$
A More Complex Problem

• Widgets Inc uses a merge sort algorithm to sort their inventory of widgets

• If it takes 66 milliseconds to sort 4096 widgets, then approx. how long will it take to sort 1,048,576 widgets?

(Note: merge sort is $O(n \log n)$, 4096 is $2^{12}$, and 1,048,576 is $2^{20}$, and)
A More Complex Problem (cont.)

Setting up the formula:

\[
\frac{n_1 \log n_1}{n_2 \log n_2} = \frac{t_1}{t_2}
\]

\[
\frac{2^{12} \log 2^{12}}{2^{20} \log 2^{20}} = \frac{66 \text{ ms}}{t_2}
\]

Solve for \(x\) (remember \(\log 2^y\) is just \(y\))
Growth Functions

- We’ve abstracted run time as a characterization by these functions that describe the rate of growth in time as \( N \) grows.

![Graph showing growth functions]

- \( O(1) \)
- \( O(\log(N)) \)
- \( O(\sqrt{N}) \)
- \( O(N) \)
- \( O(N^2) \)
- \( O(N^3) \)
Determining Big Oh: **Simple Loops**

For simple loops, ask yourself how many times loop executes as a function of input size:

- Iterations dependent on a variable \( n \)
- Constant operations within loop

```c
double minimum(double data[], int n) {
    // Pre: values has at least one element.
    // Post: returns the smallest value in collection.
    int i;
    double min = data[0];
    for(i = 1; i < n; i++)
        if(data[i] < min) min = data[i];
    return min;
}
```

\[ O(n) \]
What is the Big-Oh?

for(s = 0; s < N; s++)
    sum = sum + 1;

for(i = 0; i < N; i++)
    for(j =0; j < N; j++)
        printf( ...)

Total: $O(n) + O(n^2) = O(n + n^2)$

But for large values of $n$, $n^2$ dominates $n$ so:

$O(n + n^2) = O(n^2)$
Summation and the Dominant Component

- A method’s running time is sum of time needed to execute sequence of statements, loops, etc. within method.
- For algorithmic analysis, the largest component dominates (and constant multipliers are ignored).
  - Function $f(n)$ dominates $g(n)$ if there exists a constant value $n_0$ such that for all values of $n > n_0$, $f(n) > g(n)$

Example: analysis of a given method shows its execution time as $8n + 3n^2 + 23$

Don’t write $O(8n + 3n^2 + 23)$ or even $O(n + n^2 + 1)$, but just $O(n^2)$.
Constant factors and domination

But ….. suppose we have two algorithms with exact runtimes of:

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1,000,000 \cdot n$</td>
<td>$2 \cdot n^2$</td>
</tr>
</tbody>
</table>

Is it reasonable to say that runtime of Algorithm 2 dominates (is worse than) Algorithm 1?

**No** for small values of $n$

**Yes** for very large values of $n$

$1,000,000 \cdot n + 2 \cdot n^2$ behaves like $n^2$ for large $n$
double findValue(double data[], double value, int n) {
    int i = 0;
    while (i < n) {
        if (data[i] == value)
            return 1;
        i++;
    }
    return 0
}

Worst Case: ??? O(n)

Best Case: ??? O(1)

Average Case: ??? Depends on input distribution.
Benchmarking

• Algorithmic analysis is the first and best way, but not the final word

• What if two algorithms are both of the same complexity?

• Example: bubble sort and insertion sort are both $O(n^2)$
  – So, which one is the “faster” algorithm?
  – Benchmarking: run both algorithms on the same machine
  – Often indicates the constant multipliers and other “ignored” components
  – Still, different implementations of the same algorithm often exhibit different execution times – due to changes in the constant multiplier or other factors (such as adding an early exit to bubble sort)
Let's Practice: What is the O(??)

- You are given an array of $n$ numbers that are in sorted order.
- Your program must find whether or not the number $v$ is in the array.
- Can easily do this in $O(n)$ using linear search.
- Can we do better?

- **Binary search**: recursively split array in half and discard half that cannot have the value $v$
- What is big-Oh?

  $O(\log(n))$ since can only split in half $\log(n)$ times.
Let’s Practice: What is the $O(\ ?\ ?\ )$

```c
int firstHalfOccurrences (double data[], double testValue, n) {
    int count = 0;
    for (int i = 0; i < (n / 2); i++) {
        if (data[i] == testValue)
            count++;
    }
    return count;
}
```

<table>
<thead>
<tr>
<th>Worst Case</th>
<th>Best Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
$O(\sqrt{n})$ in terms of $n$ (worst case)

```c
int isPrime (int n) {
    for (int i = 2; i * i <= n; i++) {
        if (0 == n % i) return 0;
    }
    return 1; /* 1 is true */
}
```

$O(\sqrt{n})$
void matrixMult (int a[][], int b[][], int c[][], n) {

    // a and b are square nxn matrices
    // after running function c = a*b

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            c[i][j] = 0;
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
        }

}
void printSums (n) {
    int i, j;
    for (i = 1; i <= n; i++) {
        sum = 0;
        for (j = 1; j <= i; j++) {
            sum += j;
            printf("%d\n", sum);
        }
    }
}

\[
1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)
\]
You are responsible for the following:

- Worksheet 9: Summing Execution Times
- Worksheet 10: Wall Clock Time Estimation