CS 261 – Data Structures

Big-Oh and Execution Time: A Review
Big-Oh: **Purpose**

A machine-independent way to describe execution time

Purpose of a big-Oh characterization is: to describe change in execution time relative to change in input size in a way that is independent of issues such as machine times or compilers
We want a method for determining the relative speeds of algorithms 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
Algorithmic Analysis

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

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

• Suppose we have an algorithm that is \( O(n) \) (e.g., summing elements of array)

• Suppose to sum 10,000 elements takes 32 ms.

• How long to sum 20,000 elements?

• If the size doubles, the execution time doubles
Non-Linear Times

• Suppose the algorithm is $O(n^2)$ (e.g., sum elements of a 2-D array)

• Suppose size doubles, what happens to execution time?
• It goes up by 4
• Why 4?
• Need to figure out how to do this …
The Calculation

The ratio of the big-Oh sizes should equal the ratio of the execution times

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

We increased \( n \) by a factor of two:

\[
\frac{n^2}{(2n)^2} = \frac{t}{x}
\]

then solve for \( x \)
A More Complex Problem

- Acme Widgets 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:

\[
\frac{n_1 \log n_1}{n_2 \log n_2} = \frac{t_1}{t_2} \quad \Rightarrow \quad \frac{2^{12} \log 2^{12}}{2^{20} \log 2^{20}} = \frac{66 \text{ ms}}{x}
\]

Solve for \(x\) (remember \(\log 2^y\) is just \(y\))
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;
}
```

$O(n)$
Determining Big Oh: Not-So-Simple Loops

Not always simple iteration and termination criteria

– Iterations dependent on a function of $n$
– Constant operations within loop
– Possibility of early exit:

```c
int isPrime(int n) {
    int i;
    for(i = 2; i * i < n; i++) {
        if (n % i == 0) return 0; // If i is a factor, then not prime.
    }
    return 1; // If loop exits without finding a factor \( \Rightarrow \) n is prime.
}
```

$O(\sqrt{n})$

But what happens if it exits early?

Best case?
Average case?
Worst case?
Determining Big Oh: Nested Loops

Nested loops (dependent or independent) multiply:

```c
void insertionSort(double arr[], unsigned int n) {
    unsigned i, j;
    double elem;
    for(i = 1; i < n; i++) {
        // Move element arr[i] into place.
        elem = arr[i];
        for (j = i - 1; j >= 0 && elem < arr[j]; j--) {
            arr[j+1] = arr[j];  j--;  // Slide old value up.
        }
        arr[j+1] = elem;
    }
}
```

Worst case (reverse order): $1 + 2 + \ldots + (n-1) = \frac{n^2 - n}{2} \rightarrow O(n^2)$
Determining Big Oh: **Recursion**

For recursion, ask yourself:

(a) How many times will function be executed?
(b) How much time does it spend on each call?

Multiply these together

```java
double exp(double a, int n) {
    if (n < 0) return 1 / exp(a, -n);
    if (n = 0) return 1;
    return a * exp(a, n - 1);
}
```

Not always as simple as above example:

Often easier to think about algorithm instead of code
Determining Big Oh: Calling Functions

When one function calls another, big-Oh of calling function also considers big-Oh of called function (and how many times embedded function(s) are called):

```c
void removeElement(double e, 
                    struct Vector *v) {
    int i = vectorSize(v);
    while (i-- > 0)
        if (e == vectorGet(v, i)) {
            vectorRemove(v, i);
            return;
        }
}
```

\( O(1) \)  \( O(n) \)  \( O(n) \)  \( O(1) \)  \( O(n) \)  \( O(n) \)
Determining Big Oh: Logarithmic

• I’m thinking of a number between 0 and 1024

• How many guesses do you need to find my number?

• Answer: approximately $\log 1024 = \log 2^{10} = 10$

• In algorithmic analysis, the log of $n$ is the number of times you can split $n$ in half (binary search, etc)
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 $O(n^2)$.
void insertionSort (double v [], int n) 
{
    for (int i = 1; i < n; i++) {
        double element = v[i];
        int j = i - 1;
        while (j >= 0 && element < v[j])
        {
            { v[j+1] = v[j]; j = j - 1; }
            v[j+1] = element;
        }
    }
} // normally we use worst case time
# Computation Complexities

<table>
<thead>
<tr>
<th>Function</th>
<th>Common Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n!$</td>
<td>Factorial</td>
</tr>
<tr>
<td>$2^n$ (or $c^n$)</td>
<td>Exponential</td>
</tr>
<tr>
<td>$n^d$, $d &gt; 3$</td>
<td>Polynomial</td>
</tr>
<tr>
<td>$n^3$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$n^2$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$n\sqrt{n}$</td>
<td></td>
</tr>
<tr>
<td>$n \log n$</td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>Linear</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>Root-$n$</td>
</tr>
<tr>
<td>$\log n$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>1</td>
<td>Constant</td>
</tr>
</tbody>
</table>

![Graph of function complexities](image)
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( ?? )

```java
int countOccurrences (double [] data, double testValue) {
    int count = 0;
    for (int i = 0; i < data.length; i++) {
        if (data[i] == testValue)
            count++;
    }
    return count;
}
```
O( ?? ) 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 */
}
```
Worst case $O(???)$

```c
void printPrimes (int n) {
    for (int i = 2; i < n; i++) {
        if (isPrime(i))
            printf("Value %d is prime\n", i);
    }
}
```
Nested Loops - $O(??)$

```c
void matMult (int [][] a, int [][] b, int [][] c) {
    int n = n; // assume all same size
    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];
        }
}
```
Less obvious $O(??)$

```c
void selectionSort (double storage [ ], int n) {
    for (int p = n - 1; p > 0; p--) {
        int indexLargest = 0;
        for (int i = 1; i <= p; i++) {
            if (storage[i] > storage[indexLargest])
                indexLargest = i;
        }
        if (indexLargest != p)
            swap(storage, indexLargest, p);
    }
}
```
Reading & Worksheets

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