CS 261 – Data Structures

Big-Oh Review
Measuring Performance

What is a Benchmark?

Why is it not a general characterization of your code performance?

- Program Size \((N)\)
- Processor differences
- Load on the machine
- Compiler differences
Complexity Analysis - Intuition

How do you find a name/number in a randomly organized phonebook of N names?

- How much time does each comparison take?
- How many comparisons?
  - Worst and Average case
- What if we double the number of names?

Key: The time for each comparison doesn’t really matter, the time to run is *proportional to N* for *linear search*
Complexity Analysis - Intuition

How would you efficiently find a name/number in an alphabetically ordered phone book?

We can divide an ordered list in half \( \log(N) \) times

Worst Case: \( \log(N) \) comparisons : \( C \times \log(N) \)

What if we double \( N \)?

\[
C \times \log(2N) = C \times \log(2) + C \times \log(N)
\]

\[
= C + C \times \log(N) = C(1+\log(N))
\]

When double list, you expect to perform just 1 extra calculation!!!
Complexity Analysis with Big oh - Intuition

When we say an algorithm is $O(N)$ we are saying that it’s execution time growth is bounded by $N \times$ some constant C

In the previous two examples we saw an algorithm bounded by the function $N$ and an algorithm bounded by the function $\log(N)$

Abstracted away the details of the machine, context, etc. and focus purely on size of the input data, $N$

Now...which one is faster (or smaller) ???
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)]
Summing Execution Times

Most code consists of several parts that must be done sequentially and thus added together to find the execution time.

One function *dominates* another if, as input grows, one is always larger than the other no matter what constants are involved.

So, when summing execution terms, throw away all but the dominant term.
void foo (N) {
    for(s = 0; s < N; s++)
        sum = sum + 1;
    //comment
    for(i = 0; i < N; i++)
        for(j =0; j < N; j++)
            printf( ...)
}

<table>
<thead>
<tr>
<th>Cost</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>n+1</td>
</tr>
<tr>
<td>c2</td>
<td>n</td>
</tr>
<tr>
<td>c3</td>
<td>0</td>
</tr>
<tr>
<td>c4</td>
<td>n+1</td>
</tr>
<tr>
<td>c5</td>
<td>n * n + 1</td>
</tr>
<tr>
<td>c6</td>
<td>n * n</td>
</tr>
</tbody>
</table>

Total Cost: \( c1(n+1) + c2n + c4(n+1) + c5n(n+1) + c6n^2 \)
Summing Execution Times

What is the Big-Oh execution time of the following:

\[ 3N^2 + 2N + 5 \text{ operations} \]

```c
void foo () {
for(s =0; s< N; s++)
    sum = sum + 1;

for(i = 0; i < N; i++)
    for(j =0; j < N; j++)
        printf( ...)
}
```
Limits of Big-Oh

Applies as values get very large

When \( N \) is small, big-Oh behavior may not be consistent

ie. \( O(N^3) \) algorithm may finish before \( O(\log N) \) algorithm, due to the constants. (But who really cares what happens when small!)
Estimating Wall Clock Time

If we know the clock time for some $N$ and we know the Big-Oh, we can estimate the time for another $N$

\[
\frac{\log(N1)}{T1} = \frac{\log(N2)}{X}
\]
Practice

Worksheet 9 - Summing Execution Times