Learning Goals

- Review Big-Oh and learn big/small omega/theta notations
- Practice solving recurrences and proving asymptotic bounds
- Practice designing algorithms and justifying runtime and correctness

What are Algorithms?

What is an algorithm? An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. Therefore, an algorithm is a sequence of computational steps that transform the input into output. We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies the desired input/output relationship, and the algorithm describes a specific computational procedure for achieving that relationship.

Best, Average, Worst Case Run Time Analysis

When analyzing algorithms, we are often interested in analyzing the best, average, and worst cases of running time.

Typically, best case performance is not really of concern due to triviality. Algorithms may be modified to make best case performance trivial for small input by hardcoding. In these cases, the best case performance is effectively meaningless!

Often, it is of more concern to perform worst case analysis, i.e. identifying the inputs that cause the algorithm to have the longest running time (or use the most space) and identifying the running time and usage bounds. Why is this useful?

- The worst case running time of an algorithm gives an upper bound on the running time for any input. Knowing this provides a guarantee that the algorithm never takes any longer.
- For some algorithms, the worst case may occur fairly often.
- Often, the “average case” is roughly as bad as the worst case.

Finally, in some cases we may be interested in the average case running time of an algorithm, where we would use probabilistic analysis to examine particular algorithms. This doesn’t surface too much, as what constitutes an “average input” is often not apparent. Often, we would then assume that all inputs of a particular size are equally likely (uniform distribution). This assumption is often violated in practice, so we might modify an algorithm to be randomized, to enable a probabilistic analysis and expected running time. Such algorithms are outside the main scope of this class.

Big-Oh Definitions

Presented below is a brief overview of asymptotic notation that will be fundamental in this course.
Big-Oh Notation

**Definition (1).** \( f(n) \in O(g(n)) \) if there exist positive constants \( n_0 \) and \( c \) such that \( f(i) \leq cg(i) \) for all \( i \geq n_0 \).

**Definition (2).** \( f(n) \in O(g(n)) \) if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \) or a constant.

Simplified: If \( f(n) \) is in \( O(g(n)) \), \( g(n) \) is an asymptotic upper bound for \( f(n) \).

Big-Omega Notation

**Definition (1).** \( f(n) \in \Omega(g(n)) \) if there exist positive constants \( n_0 \) and \( c \) such that \( f(i) \geq cg(i) \) for all \( i \geq n_0 \).

**Definition (2).** \( f(n) \in O(g(n)) \) if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \) or a constant.

Simplified: If \( f(n) \) is \( \Omega(g(n)) \), \( g(n) \) is an asymptotic lower bound for \( f(n) \).

Big-Theta Notation

**Definition (1).** \( f(n) \in \Theta(g(n)) \) if and only if \( f(n) \in O(g(n)) \) and \( f(n) \in \Omega(g(n)) \).

**Definition (2).** \( f(n) \in O(g(n)) \) if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = a \) constant.

Simplified: If \( f(n) \) is \( \Theta(g(n)) \), \( g(n) \) is an asymptotically tight bound for \( f(n) \).

The notations refer to **classes of functions**. When you read \( f(n) = O(g(n)) \), this is equivalent to the statement: \( f(n) \in O(g(n)) \). Specifically, \( f(n) \) is in the class of functions which are asymptotically bounded above by \( g(n) \). Likewise, Big-\( \Omega \), Big-\( \Theta \), etc., all reflect classes of functions. In addition to Big-\( O \), \( \Omega \), and \( \Theta \), we also learned about tilde approximation.

Tilde Notation

**Definition.** \( f(n) \sim g(n) \) if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \).

Simplified: Tilde approximation of a function can be thought of as the highest order term with its coefficient.

Problems (Big-Oh)

Problem 1

**Problem 1 a**

Provide a running time analysis of the following loop. That is, find both Big-Oh and Big-\( \Omega \):

```c
for (int i = 0; i < n; i++)
  for (int j = i; j <= n; j++)
    for (int k = i; k <= j; k++)
      sum++;
```
Problem 1 b

Provide a running time analysis of the following loop:

```java
for (int i = 2; i < n; i = i * i)
    for (int j = 1; j < Math.sqrt(i); j = j+j)
        System.out.println("*");
```

Problem 2

You are given the following algorithm for Bubble-Sort:

Algorithm 1 Bubble Sort

```java
function Bubble-Sort(A, n)
    for i ← 0 to n - 2 do
        for j ← 0 to n - i - 2 do
                swap(A[j], A[j + 1])
            end if
        end for
    end for
end function
```

Given some sequence \(\langle a_1, a_2, \ldots, a_n\rangle\) in \(A\), we say an inversion has occurred if \(a_j < a_i\) for some \(i < j\). At each iteration, Bubble-Sort checks the array \(A\) for an inversion and performs a swap if it finds one. How many swaps does Bubble-Sort perform in the worst-case and in the average-case?

Problem 3

Prove or disprove the following statement:

\[ \lg(n!) \text{ is } \Theta(n \lg n) \]

Problem 4

Prove that a function \(f(n)\) (which is eventually positive for all \(n \geq n_p\)) is polynomially bounded iff \(\log(f(n)) \in O(\log n)\). In other words:

\[ f(n) \in O(n^k) \iff \log(f(n)) \in O(\log n) \]

where \(k\) is some constant.

Problem 5

Prove or disprove the following:

1. \(\lceil \lg n \rceil!\) is polynomially bounded.
2. \(\lceil \lg \lg n \rceil!\) is polynomially bounded.