Learning Goals

During this lab, you will:

- review Big-Oh notation
- examine certain functions and their relative asymptotic growth rates
- examine the runtime complexity of algorithms

Algorithms and Running Times

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 Analysis

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

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 runtime 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 Notation

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

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

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

**Big-Omega Notation**

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

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

**Big-Theta Notation**

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

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

**Little-Oh Notation**

**Definition.** $f(n) \in o(g(n))$ if for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(i) < cg(i)$ for all $i \geq n_0$.

Alternatively, $f(n) \in o(g(n))$ if,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

**Little-Omega Notation**

**Definition.** $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$

Alternatively, $f(n) = \omega(g(n))$ if and only if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

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.

**Lab Problems**

**Problem 1**

Order the following functions such that if $f$ precedes $g$, then $f(n)$ is $O(g(n))$.

$\sqrt{n}$, $n$, $n^{1.5}$, $n^2$, $n \log n$, $n \log \log n$, $n \log n^2$, $2^{n/2}$, $2^n$, $\log(n!)$, $n^2 \log n$, $n^3$, $2^n$
Problem 2

a) Provide a runtime analysis of the following loop using big-Oh notation:

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

b) Provide a runtime 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(' '*j);
```

Problem 3

Sorting Problem

For example, we consider the numerical sorting problem:

Given a sequence of \( n \) numbers \( \langle a_1, a_2, \ldots, a_n \rangle \), generate a permutation or reordering \( \langle a'_1, a'_2, \ldots, a'_n \rangle \) such that \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \).

You may have seen various algorithms that solve the sorting problem before. Here, we will discuss bubblesort and examine its runtime in detail.

```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 4

In this problem, you are not allowed to use the theorems about Big-Oh given to you in the crash course notes. Your proof should follow exclusively from the definition of Big-Oh.

Prove or disprove the following statement:

\[ f(n) + g(n) \text{ is } \Theta \left( \max \{ f(n), g(n) \} \right), \text{ where } f, g : R \to R^+. \]
Problem 5
Prove or disprove the following statement using induction:

\[ 2^n \text{ is } O(n!). \]

Problem 6
Prove or disprove the following statement:

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