Runtimes usually change when we loop a statement or a group of statements a certain number of times. There are some simple "laws" that I noticed when I took this course that I believe are useful.

Listing 1: Basic Rules

```
1  public class Law {
2      public static void main(String[] args) {
3          for (int i = 0; i < n; i ++) {
4              // Any O(1) statements
5          }
6
7          for (int i = 0; i < n; i++) {
8              for (int j = 0; j < n; j ++) {
9                  // Any O(1) statements
10             }
11         }
12
13         for (int i = 0; i < n; i++) {
14             for (int j = 0; j < i; j++) {
15                 // Any O(1) statements
16             }
17         }
18     }
19 }
```

The first for loop from lines 3-5 is $O(n)$. This is because we loop an $O(1)$ statement $n$ times. The second for loop from lines 7-11 is $O(n^2)$. The third loop from lines 13-17 is $O(n^2)$ as well. This one is less obvious however we will prove it later on.

## Examples

What is the Big-Oh! of each of the following functions?

1. $f(n) = 5n^2 + n + 1$ $O(n^2)$
2. $f(n) = \log n + n! + n \log n$ $O(n!)$
3. $f(n) = n^4 \log n + n \log n + 3n^7$ $O(n^7)$
4. $f(n) = (n^2+1)(n+1)^3$ $= (n^2+1)(n^3 + \dots) = n^6 + \dots$ $O(n^6)$
5. $f(n) = 600000n + 309n^4$ $O(n^4)$

## Practice 1

Consider a sorting algorithm BOGO Sort. This sorting algorithm takes an array of size $n$ and sorts the array by randomly permuting the elements in the array and checking if it is sorted. If the random permutation is sorted, the algorithm is complete, if it is not sorted it randomly shuffles the array again until it is sorted. What is the runtime of this algorithm? $n!$ ways to arrange an array of $n$ elements so $O(n!)$

## Practice 2

The following code is the getIndexOf(T anEntry) method from an ArrayBag implementation.

Listing 5: getIndexOf for array bag

```
1  private int getIndexOf(T anEntry) {
2      int where = -1;
3      boolean stillLooking = true;
4      int index = 0;
5      while ( stillLooking && (index < numberOfEntries)) {
6          if (anEntry.equals(bag[index])) {
7              stillLooking = false;
8              where = index;
9          } // end if
10         index++;
11     } // end while
12     return where;
13 }
```

n — Assume equals is O(1)

What is the runtime of the method? Hint you must also know the runtime of any methods inside getIndexOf.

Worst case $O(n)$ - meaning anEntry is the last element

Avg case $O(n)$

Best case $O(1)$ anEntry is first element.

## Practice 3

The following code is the add(T newEntry) for a resizable array bag implementation. Assume that the bag always has an initial capacity of 10.

Listing 6: add for array bag

```
1  public boolean add(T newEntry) {
2    checkInitialization();
3    boolean result = true;
4
5    bag[numberOfEntries] = newEntry;
6    numberOfEntries++;
7
8    // If there is no room left in the array, resize it to twice the length.
9    if (numberOfEntries == bag.length) {
10     T[] newBag = (T[]) new Object[bag.length*2];
11     for (int i = 0; i < numberOfEntries; i++) {
12       newbag[i] = bag[i];
13     }
14     bag = newBag;
15   }
16
17   return result;
18
19 } // end add
```

What is the runtime of this code? Is it always the same? When is it different?

O(1) if no resize O(n) if resize

## Bonus

Show that $\log_{10} n$ and $\log_4 n$ are both $O(\log n)$. In other words, show that the base of the logarithm does not change the runtime. Hint use the fact that ~~$\log_a n = \frac{\log n}{\log_e n}$~~.

$$\log_a n = \frac{\log n}{\log_e a}$$

$$\log_{10} n = \frac{\log n}{\log_e 10} = \frac{1}{\log_e 10} \cdot \log n = O(\log n) \checkmark$$

we drop multiplicative constants

$$\log_4 n = \frac{\log n}{\log_e 4} = \frac{1}{\log_e 4} \cdot \log n = O(\log n) \checkmark$$