Complexity

> *In the future, the world has become memory starved. The megacorporations own memory by the megabytes, and charge console cowboys small fortunes for the privledge to run their programs. You can't help but feel sorry for your chummer, River, who was framed for running a bogo sort program on the corp servers. While the deckers are making a steady advance in freeing the masses from the iron grip of these megacorps, we are forced into paying their price... Luckily, a runner like you know how to make use of the Big O analysis to reduce complexities of programs, saving you just enough money for some grub, and imported viruses from China, specially made for the very megacorp. Get them back real good for what they did to River.*

What is complexity

Complexity, usually defined by the **Big O** notation, is the time it takes for an algorithm to complete it's program as a function of input size **in it's worst case**. The fastest a program can run is O(1), while the slowest algorithm can take up to O(2^n). High complexity programs will cause the execution time to grow rapidly as inputs raise, while low complexity programs have a slow execution growth time. Panhandlers like us always opt for the low complexity programs... But then again, who wouldn't? Time IS money, and even megacorps would opt out for the low complexity programs, too.

Determining Complexity

We can determine complexity by the amount of simple operations. We can define simple operations as things such as comparisons, additions, and assignments. We can assume that these operations take the same amount of time.

Constant runtime

For the function below, we see a **constant runtime** program, which means the program will **have the same big O runtime regardless of input:

```
1  public static void countToN(int N) {
2    for (int i=0; i<=3; i++)
3      System.out.println(n);
4  } //End countToN()
```

In the above, we initialize `i` once, and have a comparison, print, and increment for each iteration. N does not have a say on the execution time (whatever execution time difference can be negliable). Since there are a total of 4 iterations and one initialization, the big O notation will be O(n) = 13.

Linear Runtime

In the following function, n is a **linear runtime** program, which would mean the runtime is a linear function of the input:

```
1  public static void countToN(int n) {
2    for(int i=0; i<=n; i++)
3      System.out.print(i);
4  } //End countToN()
```

Since the condition for the loop termination is based on n, we can see this loop as O(n) = 3n + 1. The extra operation is from the variable initialization. This algorithms tend to be safe in terms of memory management.

Quadratic Runtime

```
1  public static void countToNTwice(int n) {
2    for(int i=0; i<=n; i++) {
3      for(int j=0; j<=n; j++) {
4        System.out.println(j);
5      }
6    }
7  } //End countToNTwice()
```

This is a **quadratic runtime** program. This program complexity will be in the form of a quadratic function, like Ax^2 + Bx + C. These algorithms tend to grow rapidly and eat memory quickly. Use carefully.

Less Common Complexities

Less common complexities can be researched on your own time. Luckily, you know how to prevent your programs from using the O(n!) runtime algorithms unless absolutely necessary.

Common Complexities

Common complexies for some algorithms are as follows:

1. Unbalanced Arrays - O(n)
2. Sorted Arrays - O(log(n))
3. Balanced BST - O(log(n))
4. Unbalanced BST - O(n)

Armed with this knowledge, you know which is the best for your algorithms.