Prepared by Linan Qiu <lq2137@columbia.edu> and Zeynep Ejder <ze2113@barnard.edu>

Simple Complexity

We need a simple way to describe "how long code takes to run." We can make statements like:

- If we increase the input by 10 times, the runtime will increase by **at most** 1000 times
- If we increase the input by 10 times, the runtime will increase by **at least** 100 times

In tuitively, we only really care about the case for **at most** since it gives us the **upper bound**. However, sometimes we do care about the **at least** part, which gives us the **lower bound**.

These descriptions are very vague, so we codify them with just a little bit of math.

Big-Oh

In the next few sections, we use $T(N)$ to denote the time taken to process N input. For example, if we say that $T(100) = 500ms$, we are saying that the time taken to process an input of size 100 (or value 100 depending on the context) is 500 milliseconds.

Upper Bound

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$

That definition sounds scary, so let's break it down.

We use a function $f(N)$ to describe an upper bound. For example, let's say we are trying to measure the $T(N) = N + 100$. We can pick a function, say $f(N) = N^2$, and say that $f(N)$ grows faster than $T(N)$. Why so?

Some of you may be tempted to use calculus, but that's not really needed. Instead, we can say that at a really large value for N, $f(N) \geq T(N)$. For example, even though $T(1) = 1 + 100 > f(1) = 1$, T(1000) = 1 + 1000 < f(1000) = 1000^2$. In fact, for every integer after 10, $f(N) > T(N)$. So we say that $f(N)$ is an upper bound for $T(N)$ because we can find $n_0 = 10$ such that $f(N)$ is always greater than $T(N)$ for all N greater than n_0. Then, $T(N) = N + 100 = O(N^2)$

We can shrink the bound further. Let's make use of the second part of the definition: the c portion. We can prove that the upper bound of $T(N) = N + 100$ is $f(N) = N$. First, realize that the definition for O doesn't specify that $T(N) \leq f(N)$. Instead, it adds the constant multiplier c in front. In our case, this means that we don't have to prove that $N + 100 \leq N$ which is clearly not possible. Instead, we can just prove that $N + 100 \leq cN$ if we pick a certain c and for all N after a certain n_0. Well we can certainly do that. Let's say we pick a $c = 2$, and set $n_0 = 100$. Then, you can verify for yourself that for every N after $n_0 = 100$, $T(N) = N + 100 < 2N$. Hence, N is an upper bound for $N + 100$.

In other words, $N + 100 = O(N)$

Lower Bound

The same analysis can be applied to lower bound using the definition in Weiss. We leave this out of this set of notes because that is not absolutely essential for this course. We will add this in when we have time later.

Applying Big-Oh to Code

Analyzing code can help us determine the growth rate of algorithms.

For instance, let's look at the following code:

```
public void someFunction(int n) {
  int sum = 0;

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < 2n; j++) {
      sum ++;
    }
  }
}
```

The line `sum++` gets executed $2N^2$ times, but in determining the growth rate of the algorithm, we can show that $2N^2 = O(N^2)$ (usin the method demonstrated earlier) and hence this algorithm is $O(N^2)$.

We also only focus on the most significant term when determining the time complexity.

```
public void someFunction(int n) {
  int sum = 0;
```

```
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < 2n; j++) {
      sum++;
    }
  }

  for (int z = 0; z < n; z++) {
    sum++;
  }
}
```

This block of code will have a total runtime of $2N^2 + N$. This evaluates to $O(N^2)$.

We withold more examples since the deadline for HW1 has not yet passed. We will add the full lo-down once HW1 solutions are up.