## Big O Reduction Rules

Asymptotic analysis is a useful tool for measuring complexity regardless of the specific hardware or platform being used. While a complex subject, there are a number of theorems that will help reduce the task of determining the reference function g(x) and the witnesses c and k. By making use of these reduction theorems, you can produce a polynomial for any arbitrarily complex code block. In general, we'll divide our code up into methods and examine those. These methods will provide terms that will contribute to our overall polynomial. Once we've produced a formula that corresponds to the code or method in question, we'll use the rules below to (1) reduce big o estimates and determine the most succinct g(x), c, and k.

- **Addition Rule**: If a segment of code a(x) is represented by the sum of two disjoint code sections f(x) and g(x), then the O(a(x)) is equal to the larger of the two : O(f(x)) or O(g(x))
    - Specifically, O(a(x)) = Max(O(f(x), O(g(x))
    - In the example main below, foo() is O(n^3) and bar() is O(n^2), so main is f(x) = x^3 + x^2, and we can pick g(x) to be x^3 correspondingly. (The x^2 term becomes negligible as n moves to infinity, so the Big O is just the larger of the two terms.)

```
public static void main(String[] args)  {
        foo();   //x^3
        bar();  //x^2
}
```

- **Product Rule:** If a segment of code a(x) is represented by the product of two (nested) code sections f(x) and g(x), then the O(a(x)) is equal to the product of the two : O(f(x) * g(x) )
    - Specifically, O(a(x)) = O(f(x) * g(x))
    - In the example main() below, the function foo() is called n times, and inside foo(), we iterate over each of the n items. So foo() is O(n) and main calls foo() n times, resulting in a O(n * n) == O (n^2).

```
public static void main(String[] args)  {
        for(int a = 0; a < n; a++)   //n
                foo();   //n

}
public static foo() {
        for(int a = 0; a < n; a++)   //n
                System.out.println(a);
}
```

- **Log Exponent Rule:** Consider the following logarithm $a(x) = \log_2 X^c$. Note that we can rewrite this using the "log roll" as $c*\log_2 X$. Since constants are factored out during asymptotic analysis, you can simply drop the constant multiplier (which on a log is it's exponent).
    - Example $f(x) = \log_2 X^6$
        - $f(x)$ becomes $6 * \log_2 X$, and $O(\log_2 X)$

- **Log Base Rule:** You may omit the base of the log and assume its base-2.

- **Transitivity:** if $a < b < c$, then $a < c$. Related to big O, if $a = 5x^2+3x$ and I'm trying to prove that a is less than or equal to (i.e, bounded by) $x^2$, we would write $5x^2+3x \leq c * x^2$ and find some pair of c, k such that this relationship holds. Using transitivity,
    - $5x^2+3x \leq c * x^2$
    - $5x^2+3x < 5x^2+3x^2 \leq c*x^2$
    - $5x^2+3x < 8x^2 \leq c*x^2$
        - Now simply choose c ==8 and k==1

## Estimating g(x) Given f(x)

In the following section, indicate what reference function (g(x)) we should use when attempting to prove that f(x) is O( g(x) ). Use the rules and reference functions above to guide you.

(1) $f(x) = n + \log_2 n$

(2) $f(x) = n^2 + \log n^4$

(3) $f(x) = n^2 * n^3$

(4) $f(x) = n^5 / n^2$

(5) $f(x) = n*(\log n) *n$

(6) $f(x) = n + n \log n + \log n$

## Counting Operations to Produce Polynomials

In the following section, I will present you with multiple different bodies of code, and it is your job to analyze each code section and build a polynomial that represents the number of abstract operations the code performs. Once we're done here, we'll have built a polynomial that we will analyze further (in terms of g(x), c,

and k) in the next section. For the following code segments below, count the operations and produce a corresponding polynomial representation.

f(x) =

```
public static boolean isEmpty() {
      return head == null;
}
```

f(x) =

```
public static int num_occurrences(int n) {
      int count = 0;
      for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                  if( i == j ) continue;

                  if(strings[i] == strings[j]) {
                        count++;
                  }
            }
      }
      return count;
}
```

f(x) =

```
public static void c(int n) {  //three loops
      for(int a = 0; a < n; a++) {
            System.out.println( a * a);
      }
      num_occurrences(n);
}
```

f(x) =

```
public static boolean isPrime(int n) {
      if(n == 1) return false;

      for(int i = 2; i <n; i++) {
            if( n % i == 0 ) {
                  return false;
            }
      }
      return true;
}
```

## More Advanced Practice

Come up with the BigO Notation of the following algorithms, and explain its rationale:

1. Binary Search: g(x) =

2. Bubble Sort: g(x) =

**What to turn in:**

**Put your answers for**

1. Estimating g(x) Given f(x)
2. Counting Operations to Produce Polynomials
3. More Advanced Practice

in a text file (PDF, word, txt…), save it to your group code sharing tool (eg. Git repo). Submit the link.

Each group submits only one copy, the score is the grade for each group member.

Research on the BigO Notation of the following algorithms, and explain its rationale:

1. Binary Search
2. Bubble Sort

What to turn in:
A link to a text file (PDF, word, txt…) in your code sharing tool (eg. Git repo)