Lab 6: Asymptotic Analysis & Big O

CSSSKL162: Programming Methodology Skills

Summary[1]

In this lab, we will evaluate code with respect to the **Big O** techniques we've covered in class. This so-called **Landau notation** is interested in expressing the growth of a function relative to its input size (n). Additionally, when using **Big O** we focus on the worst-case growth complexity; note that other notations exist that measure best-case performance (**Big Omega**), and even both best-and-worst case performance (an upper and lower bounds, called **Big Theta**). We'll examine software functions step-by-step to determine the total number of operations a body of code uses when processing n elements. This will produce a polynomial function f(x) whose worst-case runtime (with respect to n) is unknown. It will be up to the reader to select a second function (g(x)) from a list of reference functions and show that for some nonnegative integers c and k:

$$| f(x) | \leq c | g(x) | \text{ when } x > k$$

If we can demonstrate this property by finding g(x) and a pair of *witnesses* c and k, then we have proven that our unknown function f(x) is a member of the set O(g(x)), where g(x) is a known reference function.

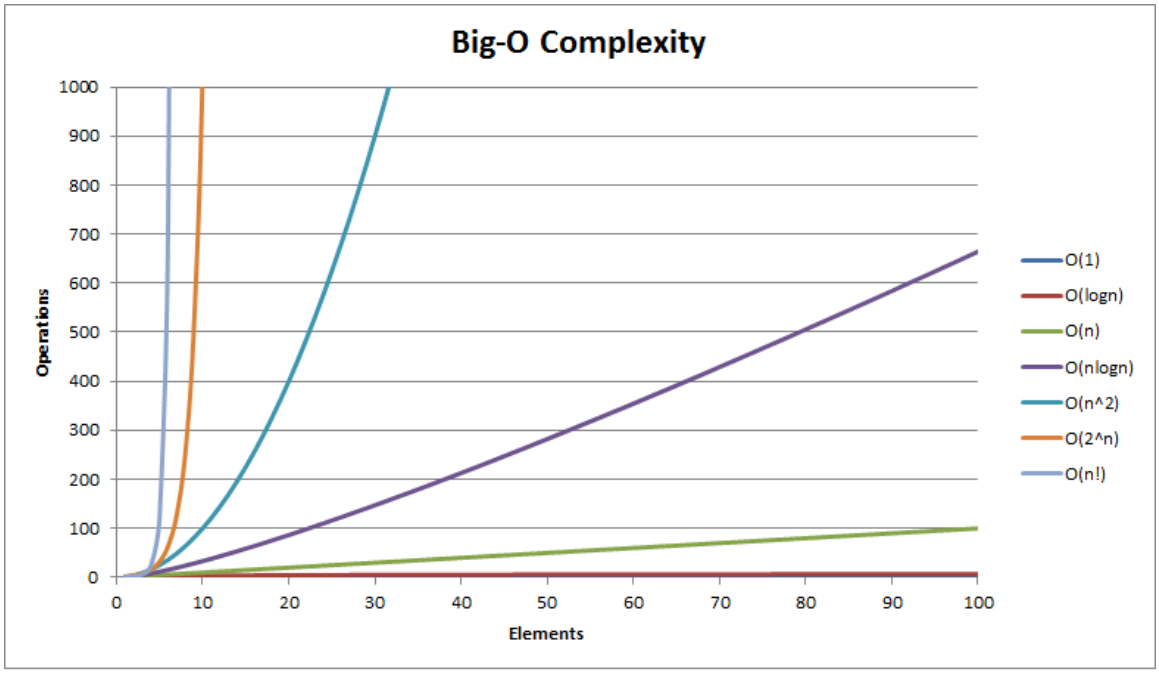
Reference Functions for g(x)

{ 1 , log n, n, n log n, n^2, n^3, …, 2^n,n!, n^n }

Countable Operations (Arithmetic, Relational, Logical, Assignment/Access)

{ + , - , ++, -- ,* , /, % }
{== , != , < , > , <=, >= }
{ & , && , | , || , ! }
{ =, [] }

[1] This lab is originally written by Rob Nash; changes and modifications by A.Retik , 2016



(source: recursive-design.com)

Introduction

We are interested recognizing a set of operations that can generalize well to multiple different hardware platforms. These operations invoke the ALU inside of the CPU and represent a basic computational "step" in a traditional load-store architecture (ie, a Von Neumann architecture). Let's put this into practice and count the number of operations that occur in a basic for loop.

```
//note that "int a = 0;" occurs only once throughout the loop execution
for(int a = 0; a < 10; a++) {   //how many operations occur each loop iteration?
        d = a + a;              //how many operations occur on this line of code?
}
```

Note that we are not concerned with the exact number of operations above, but rather, how does the number of operations change as we change n? If we double the input size, does the operation count double? Does it quadruple? Or does it change in some other way? Perhaps as n increases the operation count doesn't change at all, as we'll see in the first code example below – this is O(1). We'll express the growth of the unknown function f(x) relative to functions we *do* know about, and try to get as close to f(x) as possible while also always being greater than f(x) past some value k.

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;
}
```

Demonstrating | f(x) | ≤ c | g(x) | for all n > k

For each of the polynomials above, pick a reference function (g(x)) and constants c,k such that the g(x) bounds f(x) in the most succinct terms possible for Big O.

(1) For the function isEmpty() above, what is…
g(x) =
c =
k =

(2) For the function num_occurrences() above, what is…
g(x) =
c =
k =

(3) For the function c() above, what is…
g(x) =
c =
k =

(4) For the function isPrime() above, what is…
g(x) =
c =
k =

Future Exam Practice

We have more lecture and homeworks to go before we completely understand this next set of questions, but keep them in mind for future exams.

(5) For your Bubble Sort, what is…
f(x) =
g(x) =
c =
k =

(6) For your Selection Sort, what is…
f(x) =
g(x) =
c =
k =