Big O Practice
CS200 Recitation 6

More on Big O Notation

Eyeballing code

The process of looking at a piece of code and determining it's big O complexity is more intuitive than mechanical. Generally, you should look for where most of the work is being done, and/or for code that runs many times. Next you should think about how many times the code will run for input of a give size (n), and maybe trace out a small example. This should give you an idea of how to express the number of code cycles in terms of the size of it's input.

As an example, lets look at the *traveling salesman problem.* Finding the shortest route for a salesman to visit n cities. Each city is visited only once, and they can be visited in any order. Brute force computation of this problem is to compute all possible routes and then find the lowest distance. Adding up distance between cities is a constant time operation, so we will focus on enumerating all possible routes:

```
//arguments are two lists of cities
method trav_sales(visited, notVisited){
     if(notVisited is empty)
          return visited;
     list results;
     for (x is a city in notVisited){
          list temp = visited + x;
          list notV = remove x from notVisited;
          results.append(trav_sales(temp, notV)); //recursive call
     }
     return results;
}
```

Here we have a loop which contains a recursive call:

- Loop runs n times. After 1 recursion, loop runs n-1 times. After two recursions, loop runs n-2 times, etc...
- First time through the loop, n recursive calls are made. Each one has an n-1 loop, so it has n-1 operations done n times. You can think of it as a nested loop (where the inner loop skips an element) if you want.
- So far it looks like $n * (n-1) * (n-2) * ... * 1$, which is n factorial (n!). This makes a bit of sense, since every time a city is cisited, there are fewer cities which still need to be visited.
- I tried a few small inputs. For $n = 3$, there would be 6 routes ($3! = 6$). For $n = 4$ there were 24 routes ($4! = 24$). So n! does seem to be a good representation of how much looping it does. Since it does not do much else, it's probably O(n!)
- I looked this one up, brute force solutions to the traveling salesman problem really are O(n!)

Proofs

Other problems ask us to prove that $f(x)$ is O($g(x)$), where f and g are mathematical functions. Remember the official definition of big O: $|f(x)| \leq C|g(x)|$, whenever x > k. I actualy like the book's technique for this, so here's an example:

Show that $6x^2 + 2x + 3$ is $O(x^3)$

Start by writing down $f(x) \leq f(x)$ ... no, really, stay with me! We get:

$$6x^2 \leq x^3 \text{ for } x > 6$$

Now, we want to make the right hand side look as like g(x) as we can. To do this, we make some observations. Specifically, one about each term in the original function, relating it to g(x):

$$6x^2 \leq x^3, \text{ for } x > 6$$
$$x \leq x^3 \text{ for } x > 2 \; 3 \leq x^3, \text{ for } x > 2$$

How did I find these? By means of substitutions starting from 0 to 1, we can determine the point where this holds true. Based on these observations, when x is greater than 6 every term in the original function is less that x^3. So we can rewrite the original in equality as:

$$6x^2 + 2x + 3 \leq 6x^3 + 2x^3 + 3x^3$$

I literally just made every term in the right side an x^3 term! In fact, we can remove the coefficients on the right side, the inequality still holds:

$$6x^2 + 2x + 3 \leq x^3 + x^3 + x^3$$

$$6x^2 + 2x + 3 \leq 3x^3$$

This now looks a lot like the definition, $|f(x)| \leq C|g(x)|$, with c = 3 and k = 6 (these are the witness variables. C from the previous equations, and K is the highest bound on x from our observations above. We can now say that:

$6x^2 + 2x + 3$ is O(x^3) with witness: c = 3 and k = 6.

Exercise
Two worksheets on complexity.