Computer Science 1MD3

Lab 3 – Complexity and Algorithm Efficiency

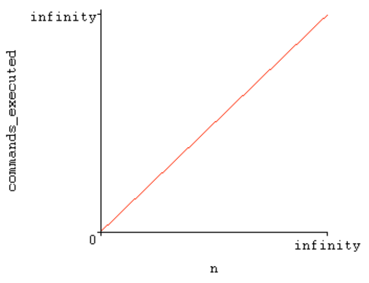
Regardless of how fast computers will get, computer scientists will always be concerned with the efficiency of the algorithms they design. In order to determine what algorithms are better, we will need to know how to compare them. This lab will demonstrate how to determine and compare an algorithms complexity.

COMPLEXITY

Since computers have different processing capabilities, it is more meaningful to represent the speed of an algorithm by the number of times a command is executed rather then the time it takes to complete the algorithm. This representation is called complexity. The complexity of an algorithm is a function which relates the number of executions in a procedure to things like loops or file size which govern these executions.

Consider the code:

```
void proc1(int n) {
      int i;
      for (i=0; i<n; i++){
            command();
      }
      return;
}
```



The number of times `command` is executed is directly related to the size of `n`. A function modeling this relation would be $f(n) = n$, where `f(n)` represents the number of times `command` is evoked. If a machine took two minutes to execute `command` it would take `(2 minutes)*f(n)` to run the procedure.

In complexity we say that `proc1` is $O(n)$, (big-oh of n), or that the running time is governed by a linear relation.

AN EXACT DEFINITION OF BIG-OH

When we say that the running time $T(n)$ of a program is $O(f(n))$, we mean that there is a positive integer c such that $T(n) \leq cf(n)$.

There is an important consequence that follows from this definition. If an algorithm has $O(6n^2 + n + 1)$ it suffices to say that the function has $O(n^2)$ since $7n^2 \geq 6n^2 + n + 1$ for large n. More generally we have that $O(a_0 + a_1n^1 + a_2n^2 + ... + a_xn^x) = O(n^x)$.

For example:

$$\frac{1}{4}n^5 + 3n^2 + 2 = O(n^5) \qquad n^2 + 1000000n + 10 = O(n^2)$$

DETERMING COMPLEXITY OF MORE COMPLICATED PROGRAMS

The following examples will further demonstrate an algorithms complexity.

Example: 1

```
void proc2(int n) {
      int i,j;

      for(i=0; i<n; i++) {
            for(j=0; i<n; i++) {            n^2
                  command();
            }
      }

      for(i=0; i<10000000; i++) {         10000000
            command();
            command();
      }

}
```

$f(n^2 + 10000000) = O(n^2)$

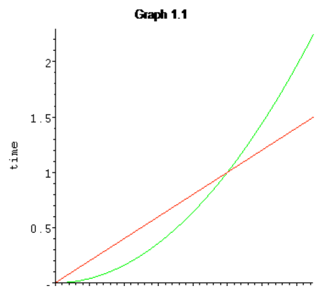
Example 2:

```
void proc3(int n) {
      int i,j;

      for(i=0; i<n; i++) {
            command();                     2n
            command();                                   2n + n^2
            for(j=0; i<n; i++) {           n^2
                  command();
            }
      }

      proc2(n);   n^2

}
```

$f(2n + 2n^2) = O(n^2)$

COMPARING ALGORITHM COMPLEXITY

Now that we can determine an algorithms complexity it would be nice to be able to compare them. Consider the two procedure P1 and P2 whose corresponding complexities are $O(n)$, and $O(n^2)$. Plotting these complexities on a graph we get Graph 1.1.



Investigating this graph we find that P2 is faster than P1 up till a given point, a_0, where P1 becomes faster.

So when deciding to use algorithm P1 or P2 we must first determine their complexities and then decide which one is better for the problem being applied to it. This is very typical in sorting algorithms where algorithms that have O(n^2) complexity such as bubble sort, would be better than O(nlogn) complexity sorts such as quicksort, in the case that the amount of items to be sorted is very small.

PROVING COMPLEXITIES

Our definition of big-oh says that in order for `f1(n)` to be big-oh of `f2(n)`, `f1(n)` must exceed `f2(n)` from a point till infinity. This implies that if we were to divide values of `f2(n)` by `f1(n)` for large `n` we should get a finite number.

For instance, if we would like to show that $\frac{7000 \cdot n \cdot 2^n}{n^2 + 3} = O(2^n)$ it is satisfactory to show that $\lim_{n \to \infty} = \frac{\frac{7000 \cdot n \cdot 2^n}{n^2 + 3}}{2^n} \neq \infty$. In this case $\lim_{n \to \infty} = \frac{\frac{7000 \cdot n \cdot 2^n}{n^2 + 3}}{2^n} = \frac{7000 \cdot n}{n^2 + 3} = 0$ which proves that 2^n is ultimately always larger than $\frac{7000 \cdot n \cdot 2^n}{n^2 + 3}$, which proves our initial statement true.

In general if we would like to prove that $f_1(n) = O(f_2(n))$ it suffices to show that $\lim_{n \to \infty} \frac{f_1}{f_2} \neq \infty$.

SELF TEST PROBLEMS

1. Are the following statements true or false:
   - $100 \cdot n^3 \cdot 2^n + 6 \cdot n^2 \cdot 3^n = O(4^n)$
   - $3^n = O(2^n)$
   - $\log n + \sqrt{n} = O(\log(n))$
   - $n! = O(n^n)$

2. In your own words what does it mean for $f_1(n) = O(f_2(n))$?

3. Determine the complexity of the following programs:

```
void proc1 (int n,int *A) {

        int i;

        for (i=0; i<n/2; n++) {
                A[i]=A[i]*A[i];
        }

        for (i=0; i<n*n; n++) {
                A[i]=A[i}*A[i]/4;
        }

        return;

}
```

```
void proc2(int n) {
   int i,j;

   for(i=0; i<100000; i++) {
         for(j=0; j<log(n); j++) {
               command();
         }

         for(j=0; j<n*n; j++) {
               command();
         }
   }

   for(i=0; i<n*n*n; i++) {
         for(j=0; j<n/2; j++) {
               command();
         }
   }
}
```

```
*hard*

int factorial(int n) {

   if n==0 return 1;
   return n*factorial(n-1)

}
```
