Introduction to Complexity Theory

Albert Gural & Saketh Are

October 21, 2011

1 Introduction

One important aspect of any programming contest is the performance of the program - how quickly it will compute the answer for a given input. On USACO contests, the time limit is generally given as 1 second (although make sure this is actually the case for each problem).

2 A Program is a Series of Instructions

Every program you write can be represented as a series of instructions. For complexity theory, we assume each instruction takes constant time. If we had a `for` loop going through N iterations, the loop could be represented as a series of N instructions, taking linear time in N.

3 Big-O Notation

The most important way of representing the complexity of some program is its Big-O. Big-O gives a bound on the asymptotic complexity of some program. There is a formal definition of Big-O, but you will only need to get a feel for the complexity of your programs, so I won't go into that here. If you're interested, MIT provides excellent lectures on complexity:

`http://www.youtube.com/watch?v=whjt_N9uYFI`

3.1 O as a Function of $T(n)$

The basic idea is that some program will run in time $T(n)$ for some input (n is some function of the input). Since we're only cnocerned with the asymptotic performance, what we do is take the most significant term from $T(n)$ and remove the constants. Let's say this is $f(n)$. Then our program runs in $O(f(n))$.

Find the Big-O of programs with the following runtimes:

1. $T(n) = 3$
2. $T(n) = 3n^3 + 5n \log n$
3. $T(n, m) = 5n^2 + 6m^3 + 2mn$

3.2 Asymptotic Ordering

So we have a general idea of how to find Big-O for a program whose absolute runtime we know, but how do we find the asymptotically most significant term? Usually, you'll only have to worry about these runtimes:

$$1 < \log n < \sqrt{n} < n < n \log n < n\sqrt{n} < n^2 < n^2 \log n < n^3 < n^k (k > 3) < 2^n < n!$$

(Why is this true?)

By the way, logarithms and exponentials are almost always considered in base 2.

4 Big-O of a Program

So we can find the Big-O given the absolute time complexity of a program $T(n)$, but what if we just know the program/algorithm (as will always be the case when you write a program)? Then you'll need to know how to find the time complexity of your algorithm.

4.1 Inefficient Sort

Consider the following function (written in Java):

```
public static void sort(int[] array) {
    for(int i = 0; i < array.length; i++) {
        int minIndex = i;
        for(int j = i; j < array.length; j++)
            if(array[minIndex] > array[j])
                minIndex = j;
        int temp = array[i];
        array[i] = array[minIndex];
        array[minIndex] = temp;
    }
}
```

We have two nested for-loops and a bunch of constant-time statements. The outer most for-loop goes through `array.length` $= N$ iterations, while the inner for-loop goes through $N + N - 1 + N - 2 + \cdots + 1 = \frac{N(N-1)}{2}$ iterations. So overall, we're going through $C_1 \frac{N(N-1)}{2} + C_2$ programming steps, where C is some constant associated with the number of statements per for-loop. But what is the Big-O?

$$T(N) = C_1 \frac{N(N-1)}{2} + C_2 \quad (1)$$

$$T(N) = \frac{C_1}{2}(N^2 - N) + C_2 \quad (2)$$

N and C_2 are both lower order terms. Additionally, we can remove all the constant factors: C_1 and 2 to get the Big-O: $O(N^2)$.

4.2 More Efficient Sort

Here's a much more efficient sort. You may not understand how it works, but you should be able to figure out its Big-O performance.

```
public static void sort(int[] array) {
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
    for(int i = 0; i < array.length; i++)
        pq.add(array[i]); // Note: O(log n) operation, n is pq.size()
    for(int i = 0; i < array.length; i++)
        array[i] = pq.poll(); // Note: O(log n) operation, n is pq.size()
}
```

We've seen the sort of rigorous way to compute the Big-O - by first computing the time complexity $T(N)$ and finding an upper bound. But that's a bit tedious. What we know in this case is the max value of `pq.size()` is N, where $N =$ `array.length`. If both adding and polling take $O(\log n)$ where n is the size of the structure, the maximum value of n is N, and we do these operations $2N$ times, then clearly we have an $O(2N \log N) = O(N \log N)$ sort.

5 Will My Program Run in Time?

This is an essential question to ask after coming up with an algorithm and before coding it. Since USACO happens in the real world, single instructions could take different amounts of time to run based on how complex they are.

Suppose, for example, that we have a program that does N iterations of adding two numbers: $n_1 + n_2$. Clearly, this runs much faster than another program which takes $n_1/n_2 * (n_2 + n_1) - n_2$ for N iterations. (Note: for large n_1 and n_2, multiplication is not constant time, nor is addition.)

If you have to remember one number, the best limit is $50,000,000$ operations per second. In other words, when you find the Big-O of your program and plugin the largest inputs possible, it should not exceed $50,000,000$ (assuming your program must run in 1 second. Here is a more specific guide for the USACO computers:

1. Basic Arithmetic: $200,000,000$ operations per second
2. Most Operations: $50,000,000$ operations per second
3. Operations with Advanced Data Structures: $1,000,000$ operations per second

With that said, always, always, always test your program with large inputs to make sure it actually *does* run in time.

6 A Note on USACO

You can often use complexity to reason out what sort of algorithm you should use for some programming problem. In general, if you see these sorts of numbers for the range of values of a certain varible, it should give you a clue about the complexity of the intended solution! (Make sure you consider the correct variable(s), of course.)

These are general guidelines, of course, so don't necessarily expect USACO to follow these rules.

1. ≤ 10: $O(n!)$
2. ≤ 25: $O(2^n)$
3. ≤ 50: $O(n^4)$
4. ≤ 500: $O(n^3)$
5. ≤ 5000: $O(n^2)$
6. ≤ 100000: $O(n \log n)$
7. ≤ 1000000: $O(n)$

7 Problems

1. Cow Pinball (Traditional, 2009): You are given a triangle of values. $N(1 \leq N \leq 25)$ values will be on the bottom row, $N-1$ on the next row, etc, and 1 value on the top row. Find a path from the top to bottom moving to either the right of left value below at each point, that maximizes the total. What complexity is your program expected to have? What algorithm can you think of that would accomplish this complexity?
2. Same as above, but $(1 \leq N \leq 1000)$. What complexity is your program expected to have? What algorithm can you think of that would accomplish this complexity?
3. You are given N points in the plane, none of which lie along an axis, and you wish to count the number of triangles containing the origin that can be formed using these points. Suppose you were to consider every possible triangle, one by one. What would the runtime complexity be? Aroung how large can N be if the time limit is 1 second?

4. Same as above, but ($1 \leq N \leq 50000$). What runtime complexity is your program expected to have? Can you think of an approach that will run quickly enough?

5. Suppose you are trying to find the N^{th} Fibonacci number. You define a simply recursive function with two base cases: F_0 and F_1. What will the runtime complexity be?

6. Same as above, but this time you save and reuse values once you've computed them for the first time. What is the new runtime complexity?

7. Can you think of a problem where the more efficient solution would run in $O(\log(\log(N))$? Try to see what other amusing runtimes you can come up with.

8. Bonus: Prove that any comparison sort must have a worst-case runtime of $O(N \log N)$ or worse.