Complexity Theory

Hariank Muthakana

October 10, 2014

1 Introduction

For most programming contest problems, we cannot just find any solution. We would like our solutions to be efficient enough to meet the time limits of the contest. But how can we check how "efficient" our solution is? *Computational complexity* gives us the answer to this question. In short, we can look at the complexity of a proposed solution to see if it runs fast enough for the given test data.

2 Big-O Notation

Big-O is how we quantify this efficiency. It is written as a function $O(g)$, where g itself is a function of N. N is our magic number - it is the number of items we are operating on and is usually given as input for the problem. For example, a simple but important complexity is $O(N)$, or *linear time*. This means that as the input N increases linearly, the *maximum number of operations* taken by the program will also increase linearly. For more than one input, the Big-O function will involve multiple variables.

3 Examples

Suppose we have a simple program to output `"hello world"` N^2 times:

```
s = "hello world"
for i = 1 to N:
    for j = 1 to N:
        print s
print "done"
```

The first line of this program is an assignment of `s`. No matter how much we increase N, this will take the same number of operations, only one. So, this line is $O(1)$, or constant time.

The next three lines are a nested loop. The number of total iterations we have to do is N^2, which is quadratic in N. Therefore, this section is $O(N^2)$, quadratic time.

The final line is a print statement, which is obviously $O(1)$.

So, in total we have $O(1 + 1 + N^2)$. As N gets large, the only *asymptotically significant* term will be (N^2). So, the total complexity of this program is $O(N^2)$. Finding the upper bound in this way is called *amortized analysis*.

3.1 Common Runtimes

- $O(1)$ - constant time (ex. assignment, outputting a single value)
- $O(N)$ - linear time (ex. a single loop)
- $O(\log N)$ - logarithmic time (ex. binary search)
- $O(N^2)$ - quadratic time (ex. a nested loop)
- $O(2^N)$ - exponential time (ex. recursive Fibonacci)

3.2 Asymptotic Ordering

We reduced the complexity earlier by finding the most significant term. This gets slightly more complicated when logs and exponentials are introduced:

$$1 < \log N < \sqrt{N} < N < N \log N < N^2 < 2^N < N!$$

4 USACO Cheat Sheet

In USACO, your code has a fixed amount of time for each test case - 1 second for C++ and 2 seconds for Java. If it takes longer than this for a case, it will "time out" on that case and you won't get points. With that being said, you can generally tell how fast your program needs to be based on the bounds they give you:

- $N \leq 10 : O(N!)$
- $N \leq 25 : O(2^N)$
- $N \leq 50 : O(N^4)$
- $N \leq 500 : O(N^3)$
- $N \leq 5000 : O(N^2)$
- $N \leq 100000 : O(N \log N)$
- $N \leq 1000000 : O(N)$

10^8 operations. That's how many you get per second. If you plug in the maximum number for N into the complexity and you get 10^9, your program is going to take roughly 10 seconds and time out.

5 Problems

1. Suppose we have a recursive solution to find the N^{th} Fibonacci number. What is the runtime complexity? What if we use an iterative approach?
2. Cows in a Row (USACO Bronze, US Open 2012): Farmer John's N cows ($1 \leq N \leq 1000$) are lined up in a row. Each cow is identified by an integer "breed ID"; the breed ID of the i^{th} cow in the lineup is $B(i)$. FJ thinks that his line of cows will look much more impressive if there is a large contiguous block of cows that all have the same breed ID. In order to create such a block, FJ decides remove from his lineup all the cows having a particular breed ID of his choosing. Please help FJ figure out the length of the largest consecutive block of cows with the same breed ID that he can create by removing all the cows having some breed ID of his choosing.

   What is the maximum runtime complexity that our program must meet to run in time?

   How would we approach this problem to achieve this complexity?
3. Cow Photography (USACO Silver, December 2011): The cows are in a particularly mischievous mood today! All Farmer John wants to do is take a photograph of his N ($1 \leq N \leq 20000$) cows standing in a line, but they keep moving right before he has a chance to snap the picture. Just before he can press the button on his camera to snap the picture, a group of zero or more cows (not necessarily a contiguous group) moves to a set of new positions in the lineup. The process above repeats for a total of five photographs. Given the contents of each photograph, reconstruct the original intended ordering. A cow only moves in at most one photograph.

   What is the maximum runtime complexity that our program must meet to run in time?

   How would we approach this problem to achieve this complexity?