Basic Complexity Theory

Yongkoo Kang

4 October 2013

1 Introduction

Complexity Theory is the way of describing a program in terms of the problem bounds to express general bounds for about how high the problem bounds could be increased with the program performing in reasonable time. The main usage of this in programming contests is to see if a proposed solution would run within the time limits, for which there exist general guidelines that will be discussed later. Beyond programming contests, complexity theory is used to compare various solutions for reasonable applications to datasets of varying sizes.

2 Big-O Notation

Big-O notation is the preferred method of expressing the complexity of a program. It is expressed as a capital O with an expression enclosed within parentheses. For example, O(N) means that the program has a Big-O of N. The meaning of this is that the program is dependent on the argument N linearly. Other examples of commonly occurring functions are $\log N$, which is almost always in base 2, N^2, and 2^N.

2.1 Evaluation of a Program

Evaluation of a program is more or less straightforward. Most low level operations can be considered to be O(1), things such as basic arithmetic or changing values in an array. From these low level operations, loops of various types can amplify the magnitudes of the operations. The basic for-loop iteration over an array is O(N) where N is the number of elements in that array. In applications where sets are split in two, such as in the merge sort or binary searches, factors of $\log N$ appear. Nested loops result in higher order polynomials. Recursion has variable results in Big-O due to the broad nature of recursion, but while some recursive operations are essentially for-loops with O(N), some have exponential complexities.

2.2 Amortization

The purpose of Big-O is to offer an expression for the asymptotic performance of a program and as such, constants and lesser order terms can be removed for a simpler function. The following list shows a few examples.

1. $O(3N) = O(N)$
2. $O(N^3 + N^2 + 3945) = O(N^3)$
3. $O(N! + 2^N + N^9 + N^2 \log N + N \log N + 1) = O(N!)$

2.3 Asymptotic Ordering

The first two examples mentioned previously were more or less trivial, but beyond simple polynomial expressions, some of the other functions can be a bit tricky to order, so the order is as follows:

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

Other results follow easily from this hierarchy through multiplication.

2.4 Contest Guidelines

The number of operations per second runs between 1,000,000, for advanced data structures, and 200,000,000, for basic arithmetic. It is better to be cautious with guidelines, but since USACO has increased the time limit, it is usually safe to think of around 10 to 50,000,000 operations per second with complexities. That being said, the following general guidelines are on the safer side.
$N \leq 10 : O(N!)$
$N \leq 25 : O(2^N)$
$N \leq 500 : O(N^3)$
$N \leq 5,000 : O(N^2)$
$N \leq 100,000 : O(N \log N)$
$N \leq 1,000,000 : O(N)$

3 Problems

1. Haybale Stacking (USACO Bronze, January 2012): Bessie starts with N ($1 \leq N \leq 10,000$, N odd) empty stacks, numbered 1..N. FJ then gives her a sequence of K instructions ($1 \leq K \leq 250$), each of the form "A B", meaning that Bessie should add one new haybale to the top of each stack in the range A..B. For example, if Bessie is told "10 13", then she should add a haybale to each of the stacks 10, 11, 12, and 13. What is the height of the middle stack?

   What is the maximum runtime complexity so that this program would run in time?

   How can this be achieved?

   Suppose the bound on N is increased to 1,000,000 and the bound on K is increased to 25,000.

   What is the new maximum runtime complexity so that this program would run in time?

   How can this be achieved?

2. Cow Photography (USACO Silver, December 2011)Each of FJ's N ($1 \leq N \leq 20,000$) cows has a unique integer ID number. FJ wants to take a picture of the cows in a specific order, but every time he tries, a group of zero or more cows (not necessarily a contiguous group) moves to a set of new positions in the lineup. He reattempts this picture with the original setup multiple times, but each time, a new group of cows moves. This occurs 5 times. What is the actual arrangement?

   What is the maximum runtime complexity?

   How can this be achieved?