Computational Complexity

Owen Hoffman

September 28 2012

1 Computational Complexity

Computational complexity, in simple terms, describes the difficulty of a programming task. That is, it assesses the runtime and storage space needed to execute the program. This is important because in many contests, including USACO, your time and space are limited. In USACO you are limited to 1 second runtime (2 in Java) and 16mb of memory. Be careful that your algorithms are efficient!

2 Big-O Notation

Big-O notation is a means of quanitfying a program's efficiency. Efficiency can be expressed as a function in terms of the size of a problem's input, n. Big-O describes the asymptotic performance of a program. That is, as the input n increases, by how much does the runtime performance increase? Big-O answers the question 'what is the growth rate of the function?'.

For example, a problem asks us simply to add a certain number, n, of integers together. This would take n-1 operations to complete. However, for sufficiently large values of n, the -1 becomes irrelevant; we only look at the most rapidly growing term. Therefore we would say that the efficiency of this problem is O(n). Coefficients are also irrelevant because we are looking for the growth rate. A program that runs in $2n^2 + 6$ operations is O(n^2). A program that runs in constant time is said to be O(1).

3 Comparison of Big-Os

Efficiencies can be ordered based on their growth rates. For sufficiently large values of n, the following is true:
$1 < \log n < \sqrt{n} < n < n \log n < n\sqrt{n} < n^2 < n^k (k > 2) < 2^n < n!$ These are the most commonly encountered efficiencies.

4 How Fast Does a Program Need To Be?

In USACO, your programs will have to run in under one second. The problems are usually set up so that you can solve the largest test cases just in time, using the most efficient algorithm (bronze problems are usually more lenient). By looking at the bounds of the problem, you can usually tell what the efficiency of your algorithm should be.
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)

5 Amortized Analysis

Typically we characterize an algorithm based on it's slowest element. If an algorithm runs an O(n^2) function and then a O(n) function, we still call it O(n^2). But what if the algorithm runs the O(n^2) function once and then runs the O(n) function several times? This is called amortization, or performance over time. A very common example is seen in Java's implementation of the ArrayList class. The ArrayList stores a large array, which it can usually add to in O(1) time, but when the array fills up, it has to copy all of the elements to a larger array in O(n) time. Even though it uses an O(n) function rarely, most of the time it adds in O(1), so the overall efficiency is O(1) amortized time.

Another algorithm which uses amortized time is the sliding window minimum. Given a set of n integers, and a window that is k wide, find a set of the minimum values within the window as it slides across the set. For example,

a window of size 3 in the set {1, 2, 3, 4, 5} would store {1, 2, 3}, then {2, 3, 4}, then {3, 4, 5}. It initially takes O(k) to find the minimum in the first window. When the window slides, one element leaves the window, and one enters. If the entering element is smaller than the old min, it is stored as the new min. This takes O(1) time. When the old min slides out, a new min has to be calculated in O(k) time. Because the old min will slide out with 1/k probability, and it takes O(k) time to calculate the new min, it will take on average k * 1/k operations, which is O(1) amortized time. Another important example is the fibonacci heap.

"At the heart of the method is the idea that while certain operations may be extremely costly in resources, they cannot occur at a high-enough frequency to weigh down the entire program because the number of less costly operations will far outnumber the costly ones in the long run, 'paying back' the program over a number of iterations"
(http://en.wikipedia.org/wiki/Amortized_analysis).

6 Problems

1. Given a number N, N $\leq$ 5,000, determine the number of primes below N.
2. Complete the above problem with N $\leq$ 50,000.
3. Can you think of a way to do it with N $\leq$ 100,000?
4. Suppose you are trying to find the Nth fibonacci number. You define a simple recursive function with two base cases: F_0 and F_1. What will the runtime complexity be? (Saketh Are, Albert Gural, 2011)
5. 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? (Saketh Are, Albert Gural, 2011)
6. Given a tree with branching factor F, consisting of N nodes, what is the complexity of a depth-first search to find the node containing value V? What is the space complexity to store this tree?
7. The worst case time for a search in a BST is O(n), when the tree is very unbalanced. Some BSTs are self balancing What is the amortized search time of a self balancing BST?