CMPSC 250
Analysis of Algorithms
Fall 2012
Bob Roos

Notes for Weds., 12 September 2012

Reading Assignment

Finish section 1.5 (pages 216–240).

The World Is Messy

We've been looking at very simple algorithms that have quite simple, clear running times—a single loop executing n times (cn for some constant c); doubly-nested loops (cn^2 for some constant c); triply-nested loops (cn^3 for some constant c); repeated halving as in binary search ($c \log n$ for some constant c). However, real algorithms for real problems are usually much more complicated than simple nested loops or repeated halving or other "easy-to-see" factors that determine time.

A real algorithm might operate in several phases, where, for instance, the first phase might take time $22.7n \log n$, the second phase might take time $5.2n^2$, the third phase might take time $300.8n + 1000$, the fourth phase might take time $.5n^2 \log n + 2\sqrt{n}$.

What can we do with an expression like the following?

$$22.7n \log n + 5.2n^2 + 300.8n + 1000 + .5n^2 \log n + 2\sqrt{n}$$

We examine each term and focus on the one that grows most rapidly. The leading constants can be ignored for this. In the example, we need to choose between constant, $\sqrt{n}$, n, $n \log n$, n^2, and $n^2 \log n$. Of all of these, $n^2 \log n$ is the most rapidly-increasing function, so we would say that the expression grows proportionally to $n^2 \log n$ and ignore the lower order terms.

> **When $T(n)$ is expressed as a sum of terms, ignore the leading constants and choose the highest-order term**

Why can we ignore the constants? After all, if we create a table of the first few values of $.001n^2$ and $100 \log_2 n$, we clearly see that the second is bigger:

n	$.001n^2$	$100 \log_2 n$
2	.004	100
4	.016	200
8	.064	300
16	.256	400
32	1.024	500
...	...	...

However, as n gets larger, the trend reverses. For instance, when $n = 1024$, $.001(1024)^2 = 1048.576$, while $100 \log_2 1024 = 1000$. From this point on, the value of $.001n^2$ will always exceed the value of $100 \log_2 n$.

When analyzing algorithms, we are usually interested in the behavior of $T(n)$ FOR SUFFICIENTLY LARGE n

Whenever we say that an algorithm's running time $T(n)$ is "on the order of n^2" we mean that $T(n)$ is approximately cn^2 *for sufficiently large values of* n.

Notation

The authors of your book use the symbol "$\sim$" to denote that a function is "on the order of" some other other function. Thus, if we write that $T(n) \sim n \log n$ we mean that $T(n)$ is approximately equal to $cn \log n$ for some constant c and for all sufficiently large values of n.

In CMPSC 112 you may have learned "big-oh" notation, for instance, $T(n) = O(n \log n)$. This means that $T(n) \leq cn \log n$ for some constant c and all sufficiently large n. The only difference here is in the inequality symbol "$\leq$". "Big-oh" indicates an upper bound, but is not necessarily the most accurate way to identify the running time. *Any* function that is less than or equal to cn^2 is $O(n^2)$. For instance, a constant running time is $O(n^2)$. Linear time is $O(n^2)$. The "$\sim$" notation is more precise.

Some books use the notation "$\Theta(n^2)$" instead of "$\sim n^2$". They mean the same thing.

Remarks on Yesterday's Lab

One of the goals of the lab was to provide convincing experimental evidence that the running time of some algorithms *really can* be approximated as a constant times some function. You saw (I hope!) that with an appropriate constant c (which you computed), the estimate of n^3 for the running time of the "ThreeSum" algorithm was nearly equal to a constant times the running time.

What about the "`ThreeSumFast`" algorithm? I told you it should be proportional to $n^2 \log n$, but how was that formula obtained? Let's think about the problem. Initially, the integer array was not sorted. Could we solve the problem more quickly if the array were sorted?

Yes! Imagine the elements are now sorted. We can begin as we did before, with two nested loops to select elements `a[i]` and `a[j]`. We know that the third element, if it exists, must be equal to `-a[i]-a[j]` since the three of them have to sum to zero. Here's an example:

`a:`

-9	-7	-1	0	1	2	8	10
0	1	2	3	4	5	6	7

And here's the algorithm (not yet completely filled in):

```
for (int i = 0; i < n; i++) {
   for (int j = i+1; j < n; j++) {
```

```
      // Find k > j such that a[k] = -(a[i]+a[j])
   }
}
```

Suppose $i = 1$, $j = 2$. Then we need to see whether or not there is a value $k > 2$ such that `a[k]` = -(`a[1]` + `a[2]`) $= -(-7-1) = 8$. But instead of just looking at array elements starting at position 3, we can use the fact that the array `a` is sorted—we can do a binary search to see if an "8" appears between positions $j + 1$ and $n - 1$.

Binary search takes time proportional to $\log n$. In this case we are searching just part of the array, namely, the part from position $j + 1$ through $n - 1$, so it will take time $\leq c \log n$ for some constant c. (We've discussed binary search—it was our first example of an algorithm exhibiting logarithmic time.)

Java's `Arrays` class has a method named "`binarySearch`" that takes, as parameters, an array `a`, a starting position, an ending position, and a goal value to be searched for. It returns the location of the goal value if it finds it, otherwise returns -1. That means our algorithm now looks like:

```
for (int i = 0; i < n; i++) {
   for (int j = i+1; j < n; j++) {
      int k = Arrays.binarySearch(a,j+1,n-1,-a[i]-a[j]);
      if (k >= 0)
        count++;
   }
}
```

Since the $\log n$ binary search is contained in two nested loops that have running time no worse than cn^2, the algorithm takes time proportional to $n^2 \log n$, an apparent improvement over n^3.

There is still one problem, however—we assumed that we had a sorted array, and that changes the problem. The solution is to add a sorting step at the very beginning of the algorithm:

```
Arrays.sort(a); // built-in Java sorting method
for (int i = 0; i < n; i++) {
   for (int j = i+1; j < n; j++) {
      int k = Arrays.binarySearch(a,j+1,n-1,-a[i]-a[j]);
      if (k >= 0)
        count++;
   }
}
```

As we will see in chapter 2, sorting can be done efficiently in time proportional to $n \log n$. Therefore, the total running time of our algorithm is:

$$c_1 n \log n + c_2 n^2 \log n$$

where c_1 and c_2 are constants. Using our rules of thumb above regarding taking the highest-order term and considering only large n, we see that the running time of algorithm `ThreeSumFast` is $\sim n^2 \log n$.

Improving the Union-Find Algorithm from Last Time

We saw an algorithm (`QuickFindUF`) last time that can perform a "find" operation in constant time but requires linear time to perform a union. Today we began looking at an improvement on this. To make the notes more coherent I'll just place this discussion in Friday's notes.