COMP 200: Elements of Computer Science
Fall 2004
Lecture 21: October 18, 2004

More on Algorithmic Complexity

On the Board

Homework 4 — I need to write the solution key
Next homework ready Friday
Will talk about the final project next class

Back to Complexity & Big-O Notation

Last class we talked about the efficiency of algorithms and introduced the big-O notation (pronounced "big Oh"). In essence, big-O notation tells us how the time required by an algorithm grows as a function of the size of its input. In particular, we are interested in the asymptotic behavior — as N grows toward infinity, what function of N describes the algorithm's running time.

Remember, if the running time of an algorithm is bounded from above by

$$c \cdot f(N)$$

then we say that the algorithm is $\mathbf{O}(f(N))$, where N is a measure of the size of the input. Further, we expect that $f(N)$ is the smallest function for which we can prove the bound $\textit{running time} < c \cdot f(N)$.

Consider, for example, our simple program for finding the maximum value in a list.

```
max ← list[1]
for i ← 2 to n
   if (max < list[i])
      then max ← list[i]
```

If the list has n elements, the loop iterates $n-1$ times. Each operation in the loop's body is simple — that is, each operation should take constant (or $\mathbf{O}(1)$ time). It has one operation outside the loop—a simple, $\mathbf{O}(1)$ operation as well. Thus, the running time should be $\mathbf{O}(1) + \mathbf{O}((n-1) \cdot 1)$, which simplifies to $\mathbf{O}(n)$ time. (Remember, big-O notation is intended to provide an upper bound, and $\mathbf{O}(n-1)$ is, effectively $\mathbf{O}(n)$ as n grows large.

If we replace one of the operations in the inner loop with a more expensive operation, such as a mergesort — a $\mathbf{O}(n^2)$ operation — then the complexity will become $O(n) \cdot \mathbf{O}(n^2) = \mathbf{O}(n^3)$.

Notice that our notion of complexity depends quite heavily on the set of operations that we allow as elementary operations in our model of computation. So far in COMP 200, we have used simple arithmetic, comparisons, and evaluating conditionals as $\mathbf{O}(1)$ operations, along with whatever overhead a loop requires to perform its repetitions (& its book-keeping). This issue gets a little subtle: max of two (or three or any small number of arguments) can be done in $\mathbf{O}(1)$ time, but max of an arbitrary list of numbers takes $\mathbf{O}(\textit{length of list})$ time. Some simple operations take longer than is reasonable to count as $\mathbf{O}(1)$. For example, computing n^m using the naïve algorithm takes $\mathbf{O}(m)$ time — $n \cdot n \cdot n \cdot \ldots \cdot n$ (multiply n by itself m consecutive times). Logarithms, sin, cosine, tangent, and many other functions are actually computed using series approximations. It might be unfair to count these as $\mathbf{O}(1)$ operations.

In general, when designing software, we want to use the lowest complexity algorithm possible. Lower complexity means faster running times, as the size of the input grows. Of course, if we know how big the input will be, we can make comparisons based not only on the asymptotic complexity, but also on the size of the constants. In general, an algorithm with a smaller constant will run faster.

Algorithms with low-order polynomial complexity are considered practical for most problems. The algorithm to paint a menu in Windows or Mac OS X takes time that is $\mathbf{O}(\textit{number of menu entries})$. It is linear in the number of menu entries. (Each entry is a fixed height and a bounded width, so the number of pixels is — roughly — a constant for each menu entry.) Many problems that are routinely solved have $\mathbf{O}(n^2)$ or $\mathbf{O}(n^3)$ complexity — for example, matrix-matrix multiply has $\mathbf{O}(n^3)$ cost.

Sometimes, linear cost is too much. When you boot your computer, it usually checks the integrity of the file system. On your machine, this might take as long as a minute or two. Unless major problems are found, the process makes a couple of linear passes over the file systems' structure. (It only examines the structure; it does not look inside every byte on the disk. It must, however, make sure that every sector on the disk belongs to some file — or in the pool of unallocated sectors. Thus, the file-system check has a complexity that is proportional to the number of sectors in the file system. A

sector is typically 1024 or 4096 bytes, so the number of sectors is the proportional to the number of bytes of space in the file system.)

Recently, we bought a file server with two and one-half terabytes of disk space. Running **fsck**, the Linux or Mac OS X file-system checker, on two and one-half terabytes takes a couple of hours. Thus, rebooting the system would take hours. Any computer crash would make the system unavailable for hours — not a good property for a server. Imagine CNN.com or the OwlNet file server going down for three hours to check its disks.

The answer, of course, is to use another method to maintain the file system's integrity. In this case, the system is journalled — it records each write to the disk and marks them off the list when they are known to be completed. After a crash, the system can simply replay the writes, in order, that have not been certified as complete. Replaying the journal takes time proportional to the length of the journal. As long as the time required for writing the file system has a small upper-bound — say a minute — the journaling process takes much less time than checking the entire file system. The cost of checking is paid on each write, rather than on each crash. The cost of checking becomes proportional to the amount of disk-write activity, rather than being proportional to the size of the file system. Over a long period of time, you might pay more for the journaling. However, at any point in time, the amount of time required to replay the journal is small.