CS106B
Spring 99-00

Handout #15
April 12, 2000

Algorithmic analysis

Evaluating performance by time trial

In the last lecture we covered various sorting algorithms and discussed their performance properties. In particular, once we have established the correctness of an algorithm, we now can be interested in trying to find an optimally performing one. One possible means for measuring efficiency is to code the algorithm up and use a time its execution on various inputs. Evaluating algorithms based on real runtime information is in some ways very comforting – we have an extremely good sense of how the algorithm will perform in practice. But it also causes problems. What happens if my computer is faster than your computer? Or if my Macintosh executes different instructions than your PC? Or if I write my algorithm in Lisp or COBOL and you do yours in assembly? Do the results on small inputs always exactly generalize to the large ones? What about the constant overhead factors? Clearly, trying to accurately evaluate performance is difficult, if not impossible.

Big-O

Rather than producing a chart that reports that sorting 100 integers with this algorithm will take 5 seconds, and 200 will take 20, (such numbers are tied to a particular hardware and input set), we want to be able to say something like:

"Given an input of size *n*, algorithm *X* has a running time on the order of *f*(*n*)."

where *f*(*n*) is some nonnegative function that shows the growth.

Fortunately, we're not worried about *f*(*n*) being the most accurate measure in the world; we simply want to find a mathematical function that roughly characterizes how the running time of the algorithm changes with respect to the size of its input. Thus the function *f*(*n*) is known as the *growth rate* of the algorithm's running time, and we say that the running time of the algorithm is *O*(*f*(*n*)), which is pronounced "oh of *f* of *n*" or "big-oh of *f* of *n*".

In determining *f*(*n*), we will allow ourselves to ignore all low-order terms and all multiplicative constants. For example, suppose that the actual running time of some algorithm *X* is

$T(n) = 4n^2 + 2n - 4.$

We would say that the running time of *X* is $O(n^2)$. Remember, we're interested in the running times of algorithms when their data size is extremely large. When $n = 1000$, for example, 1000^2 is much more significant than (2 * 1000) or (-4).

We are going to rely on BigO so much, that it seems worth giving a bit more formal definition. Given a function Time() representing the actual run time we can say:

$\text{Time}(N) \text{ is } O(f(n)) \quad \text{if} \quad \text{Time}(N) \leq C \times f(n)$

Which is to say that f(n) is an upper bound on the growth of the true running time. At a large enough n and with some constant multiplier C, C*f(n) will be larger than the actual running time.

Note that BigO is not guaranteed to be a tight bound, though. For example, if the true running time of the function is n, it can still be said to be *$O(n^2)$* because that is an upper bound, although loosely so. BigO tells us something about how the algorithm grows, but doesn't guarantee a tight bound. There are other metrics used to compute and refer to tighter bounds, but we'll leave that to the advanced classes and just consider BigO for 106B.

Examples

To compute such a bound for an algorithm, the basic idea is to count the steps in an algorithm, focusing on those steps that vary with the input size, and ignore everything but the largest contribution. Here a few guidelines to keep in mind:

Constant rule: a segment of code takes *O(1)* (constant time) if it doesn't involve any functions calls or loops. So, for example, a sequence of fixed initialization steps that are the same for all inputs will be irrelevant in computing the algorithm's growth and thus can be discarded.

Sequence rule: If a segment that takes *O(f(n))* time is followed by another that takes *O(g(n))* time, the entire sequence takes *O(max(f(n), g(n))* time. This allows us to focus on the bigger contribution, which will dominate the cost in the long run.

Iteration rule: If each iteration of a loop takes *O(f(n))* and there are *O(g(n))* iterations, the entire loop takes *O(f(n)*g(n))* time. This is just simple multiplication at work.

Let's try an example. What is the running time of this function?

```
double Average(int scores[], int n)
{
      int i, sum;

      sum = 0;
      for (i = 0; i < n; i++)
         sum += scores[i];
      return (double)sum/n;
}
```

The initialization of sum happens just once. While it undoubtedly takes some time to execute, that time is not proportional to the number of elements to be manipulated and is therefore runs in constant *O(1)* time. The loop iterates n times. The body of the loop itself takes *O(1)* time to execute each iteration. By the iteration rule, the entire loop is *O(n).* The return statement executes in constant *O(1)* time. Adding them all together and applying the sequence rule, we have an *O(n)* result, this function is said to run in linear time.

How about this one?

```
bool IsPrime(int n)
{
      int i;

      for (i = 2; i < n); i++)
            if (n % i == 0) return FALSE;
      return TRUE;
}
```

What if we changed the upper bound on the for loop to stop at sqrt(n) instead of going all the way to n? How does that change the running time? How do the best and worst cases differ for an algorithm such as this one that doesn't always go through all loop iterations?

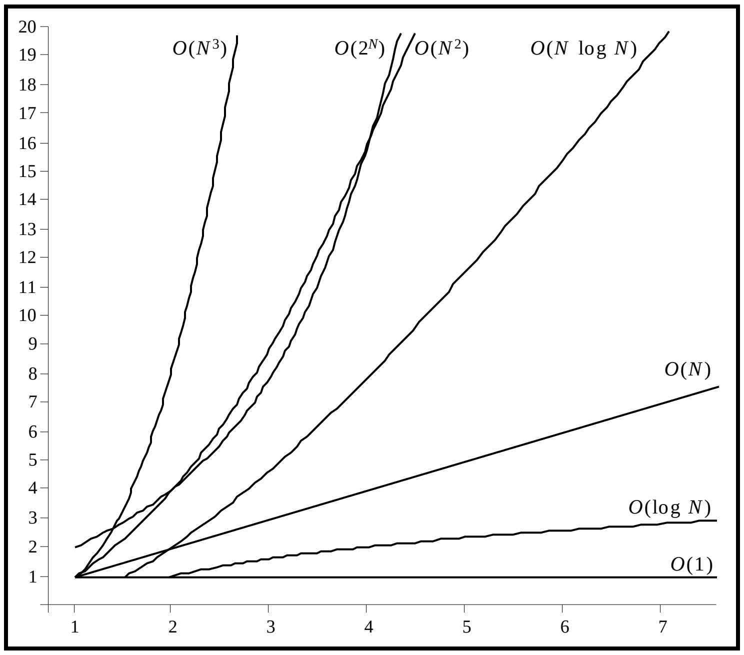
Another one for you to practice on:

```
int Mystery(int arr[], int n)
{
      int i;

      for (i = 1; i < n; i++)
          printf("Average %g\n", Average(arr, i));
      for (i = 0; i < n; i++)
          printf("%d", arr[i]);
      for (i = 2; i < n; i++)
          printf("is prime %d\n", IsPrime(i));
}
```

Complexity classes

The difference between algorithms of different complexity classes has a stunning effect on the performance at runtime. The chart below with some of the more common running times gives you an idea of the growth on small inputs. Of course, as n grows even larger, the differences become even more pronounced.



Recurrence Relations

To calculate the running time of a recursive function, we first associate with the recursive function an "unknown" running time function $T(n)$, where n represents the size of the input. Then generate a *recurrence relation* , that is, an equation for $T(n)$ in terms of $T(k)$ such that $k < n$. Consider the simple recursive factorial function as our first example:

```
int Factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return (n * Factorial(n - 1));
}
```

If n is zero, the running time of the function is merely a constant *O(1).* Otherwise, the running time is the sum of the time required for the comparison to zero, the multiplication, the time required for the recursive call of `Factorial(n-1)`, and the time required to return the function value; these are *O(1), O(1), T(n-1),* and *O(1),* respectively

So, we can express $T(n)$ as follows:

$$T(n) = \{ \begin{array}{ll} 1 & \text{if } n = 0 \\ 1 + T(n-1) & \text{otherwise } \} \end{array}$$

As you might expect, there are several different methods for solving recurrence relations. The method we will employ in this class is to expand the recurrence until any remaining *T* terms can be replaced by a value that does not involve *T*. That is, we expand the recurrence down to the base case, for which the running time is a "known" constant.. Using this approach on the factorial recurrence from above, we iteratively reaplce the recurrence relation with itself to get:

$$T(n) = 1 + T(n-1)$$
$$T(n) = 1 + [1 + T(n-2)]$$
$$T(n) = 1 + [1 + (1 + T(n-3))]$$

Which we can summarize as:

$$T(n) = i*1 + T(n-i)$$

The process continues until we reach T(0) when i = n:

$$\begin{aligned} T(n) &= n * 1 + T(n-n) \\ &= n + T(0) \\ &= n + 1 \end{aligned}$$

which therefore means *O(n)* is the total running time of `Factorial()`.

Now let's consider one of the recursive sorts which we informally analyzed in last lecture using tree diagrams. If $n = 1$, all `Mergesort` does is identify that base case and return, which is a constant-time *O(1)* operation.

If *n* > 1, we perform the following:

• establish that $n \neq 1$	*O(1)*
• call `CopySubArray` to copy left half	$O(n)$
• call `CopySubArray` to copy right half	$O(n)$
• make *two* recursive calls to `Mergesort` each call takes time $T(n/2)$	$2T(n/2)$
• call `Merge` to merge the two sorted arrays	$O(n)$

The sum is O(1) + O(n) + O(n) + 2*T(N/2) + O(n), which boils down to this recurrence relation:

$$T(n) = \{ \begin{array}{ll} 1 & \text{if } n = 1 \\ n + 2 * T(n/2) & \text{otherwise } \} \end{array}$$

Following the same iterative process as before we can expand and substitute:

$$\begin{aligned} T(n) &= n + 2*T(n/2) \\ &= n + 2 * [n/2 + 2 * T(n/4)] \\ &= n + 2 * [n/2 + 2 * (n/4 + 2 * n/8)] \end{aligned}$$

which summarizes to:

$$T(n) = i * n + 2^i * T(n/2^i)$$

Solving for $n/2^i = 1$ results in $i = \log_2 n$. By substitution we get:

$$T(n) = (\log_2 n) * n + n * T(1)$$

which gives us

$$T(n) = O(n \log n)$$

Another recurrence example, how about Towers of Hanoi? If *n* > 0, we perform the following:

• MoveTower n-1	*T(n-1)*
• Move single disk	*O(1)*
• MoveTower n-1	*T(n-1)*

which boils down to this recurrence relation:

$$T(n) = \{ \begin{array}{ll} 1 & \text{if } n = 0 \\ 1 + 2*T(n-1) & \text{otherwise } \} \end{array}$$

Following the same iterative process as before we can write this as:

$$\begin{aligned} T(n) &= 1 + 2*(T(n-1)) \\ &= 1 + 2*[1 + 2*T(n-2)] \\ &= 1 + 2*[1 + 2*[1 + 2*T(n-3)]] \end{aligned}$$

or

$$T(n) = 2^i -1+ 2^i * T(n - i)$$

Solving for i = n, we get:

$$T(n) = 2^n -1 + 2^n*1$$

which gives us

$$T(n) = O(2^n)$$

Try it for yourself on some of the other problems we've seen (permutation, subsets, making change, queens, etc.) to get some more practice!

Because of the uncertainty principle, it is impossible for the student to find out at the same time where precisely this handout is and how fast is moving. Keep your eye on your handout, least it should disappear!