Addis Ababa University, Amist Kilo July 8, 2011

Algorithms and Programming for High Schoolers

Lecture 5

This class is mostly concerned with *algorithms*. So, what is an algorithm? An algorithm is a well-defined computational procedure that takes some data as input and computes new data for output. For example, an algorithm for multiplying integers takes two integers as input and outputs their product. An algorithm for sorting a list of numbers takes a list of numbers as input then outputs a list with the same numbers, but in sorted order.

Analysis of Algorithms:

It is often the case that there are many different algorithms which accomplish the same task, and we must choose which one to use. For example, yesterday we saw four different algorithms to accomplish the task of sorting (`selectionSort`, `insertionSort`, `bubbleSort`, and `mergeSort`).

Order of growth: The most popular way of expressing the running time of an algorithm is by expressing how well it scales as the input size gets larger and larger. Also, to have absolute guarantees, we measure the running time of an algorithm in the *worst case* over all inputs of a given size. Running any one of `selectionSort`, `bubbleSort`, and `insertionSort`, on the list [n,n-1,n-2,...,1] takes roughly $C_1 n^2$ steps for some constant C_1. Meanwhile, it is possible to show that no input can cause `mergeSort` to take more than roughly $C_2 n \log_2 n$ steps for some constant C_2. No matter how C_1, C_2 are related, when n gets large enough, `mergeSort` is the superior choice.

The notation that has become the convention for measuring algorithm running times is the *big-Oh* notation. big-Oh is a way of relating two functions $f(x), g(x)$ (we would either write "$f(x)$ is big-Oh of $g(x)$", or "$f(x) = O(g(x))$"; often we don't write the x and it is understood). The precise mathematical definition of big-Oh is as follows, though roughly you should think of $f = O(g)$ as meaning f does not grow faster than g. You can think of big-Oh as being a kind of "less than or equal to" for functions.

Definition 1. *Let $\mathbb{R}$ denote the real numbers, and let $f, g : \mathbb{R} \to \mathbb{R}$ be two nonnegative functions. We say f is* big-Oh *of g, or $f = O(g)$, if for some positive numbers C and N, as long as $x > N$,*

$$f(x) \leq C \cdot g(x).$$

In the analysis of the running time of some algorithm A, we often care about $f(n)$ being a function that tells us the worst-case running time of A over all inputs of size n.

Just as big-Oh can be seen as a "less than or equal to", big-Omega can be seen as a "greater than or equal to". We say $f = \Omega(g)$ if f grows at least as fast as g.

Definition 2. *For two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$ we say f is* big-Omega *of g, or $f = \Omega(g)$, if for some positive numbers C and N, as long as $x > N$,*

$$|f(x)| \geq C \cdot |g(x)|.$$

In other words, $f = \Omega(g)$ if $g = O(f)$. We say f is Theta *of g, or $f = \Theta(g)$, if $f = O(g)$ and $f = \Omega(g)$ simultaneously.*

For those of you familiar with limits, the following is helpful:

- $f = O(g)$ usually corresponds to $\lim_{x\to\infty} f(x)/g(x) < \infty$
- $f = \Omega(g)$ usually corresponds to $\lim_{x\to\infty} g(x)/f(x) < \infty$
- $f = \Theta(g)$ usually corresponds to $\lim_{x\to\infty} f(x)/g(x) < C$ for some positive constant C.

The above bullet list is not entirely accurate in the cases that the f or g can be zero, or if f/g fluctuates in a constant-sized interval (in which case you would have to use mathematical terms known as the limit superior and limit inferior), but they are good enough approximations to the truth for anything you'd ever encounter when measuring running times in practice.

Here are some examples of big-Oh:

- $n^2 = O(n^2)$
- $n^2 = O(n^3)$
- $3n^2 = O(n^2)$
- $.5n^2 = O(n^2)$
- $5n^8 + 2n = O(n^8)$
- $3n \log_{10} n = O(n \log_2 n)$, since switching bases of log just changes the value by a constant ($\log_{10} n = (1/\log_2 10) \cdot \log_2 n$).

Now, when measuring running times of the sorting algorithms, the example [n,n-1,n-2,...,1] tells us that the running times of `selectionSort`, `bubbleSort`, and `insertionSort` are all $\Omega(n^2)$. It's also not too hard to see their worst case running times are all $O(n^2)$ (in other words, their running times are $O(n^2)$ on *all* input lists of size n). For example, in `insertionSort`, we have a `for` loop that takes n iterations, and in each iteration we have a `while` loop that goes on for at most n iterations. Thus, these three algorithms all have running times which are $\Theta(n^2)$.

What about `mergeSort`?

Induction: Before getting to the analysis of `mergeSort`, we'll discuss a method of mathematical proof known as *induction*.

Induction is a method for showing that some mathematical statement is true for the *natural numbers* $\mathbb{N}$, i.e. the nonnegative integers $\{0, 1, 2, \ldots\}$. To prove a statement by induction, one first shows that the statement holds for some starting value $n = n_0$ (the "base case"), then shows that if it holds for $1, 2, \ldots, n$, it must hold for $n + 1$ (the "inductive step"). Successfully showing the base case and inductive step together imply that the statement holds for all integers $n_0, n_0 + 1, \ldots$.

Consider for example the puzzle given in the flyer for this class, but generalized to n people: n people enter a room, and everyone shakes everyone else's hand — how many handshakes are there? Let's prove that the answer is $n(n-1)/2$, by induction. This is definitely true for the base case $n = 0$: if there are 0 people, then there are $0(-1)/2 = 0$ handshakes. Now, let's do the inductive step. We assume the answer is $n(n-1)/2$ for n people, and we show that it is $(n+1)n/2$ for $n+1$ people. Label the people as $1, 2, \ldots, n+1$. The first n people all have to shake each other's hands, so there are $n(n-1)/2$ such handshakes. Next, the $(n+1)$st guy has to shake all the other n people's hands, which contributions an additional n handshakes. So, the total number of handshakes is $n(n-1)/2 + n = (n+1)n/2$, so we have proven our claim for all $n \geq 0$.

Recurrences: How does induction fit into analyzing `mergeSort`? Well, let's look at the code for `mergeSort` again:

```
def mergeSort(L):
    if len(L)<=1:
        return L
    A = mergeSort(L[:len(L)/2])
    B = mergeSort(L[len(L)/2:])
    return merge(A, B)
```

We can write a *recurrence* that expresses the running time of `mergeSort`. What is a recurrence? A recurrence is a definition of a function that expresses its value on an input as some combination of its values on smaller inputs. For example, our definition of fib(n) as the nth Fibonacci number in Lecture 3 was a recurrence:

$$\mathrm{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise} \end{cases}$$

Turning back to `mergeSort`, let $T(n)$ be the function which denotes the worst-case running time of `mergeSort` on any input list of length n. Then, given the code for `mergeSort`, we see that

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 2 \cdot T(n/2) + C \cdot n & \text{otherwise} \end{cases}.$$

The $C \cdot n$ term comes from the running time of the `merge` operation, where $C > 1$ is some constant value. The above recurrence isn't *exactly* right since if n is odd we don't break into exactly two pieces of size $n/2$, but to make the presentation simpler let's assume for now that n is a power of 2 so that in all recursive steps we always break the input into exact halves.

Now, let's show by induction that $T(n) \le Cn \log_2 n + n$ with the base case $n_0 = 1$.

- **Base case:** When $n = 1$, $T(n) = 1$. We also have $C \cdot 1 \log_2 1 + 1 = 1$, so indeed the claim holds for $n = 1$.

- **Inductive step:** We assume the running time for $n/2$ satisfies $T(n/2) \le C(n/2)\log_2(n/2) + (n/2)$. Now, we have from the recurrence that

$$\begin{aligned} T(n) &\le 2 \cdot (C(n/2)\log_2(n/2) + (n/2)) + Cn \\ &= Cn\log_2(n/2) + n + Cn \\ &= Cn\log_2 n + n, \end{aligned}$$

  since $\log(a/b) = \log(a) - \log(b)$. Thus we have shown the claim. Now note that $Cn\log_2 n + n = O(n \log_2 n)$, so the running time of `mergeSort` is $O(n\log_2 n)$. One can similarly also show that in fact its running time is also $\Omega(n \log_2 n)$, so that it is $\Theta(n\log_2 n)$. This means that comparing `mergeSort` with the other three sorting algorithms from yesterday, `mergeSort` is faster by a factor of $\Theta(n/\log_2 n)$, which is quite large (for example, $n/\log_2 n$ is about 50000 when n is one million).