Math/CCS 103 Professor: Padraic Bartlett

Minilecture 1: Algorithms

Week 1 *UCSB 2014*

Hey! This is a minilecture on the notion of **algorithms**, **runtime** and **complexity**, the foundations for our next two weeks of class (where we're going to talk about P versus NP!) If you're interested in a longer and (I believe) more elegant treatment, Jeff Erickson's online course notes on algorithms are some of the best-written notes I've read on the subject, and are one of the primary sources for this material.

1 Defining Algorithms

Definition. An **algorithm** is a precise and unambiguous set of instructions.

Typically, people think of algorithms as a set of instructions for solving some problem; when they do so, they typically have some restrictions in mind for the kinds of instructions they consider to be valid. For example, consider the following algorithm for proving the Riemann hypothesis:

1. Prove the Riemann hypothesis.
2. Rejoice!

On one hand, this is a fairly precise and unambiguous set of instructions: step 1 has us come up with a proof of the Riemann hypothesis, and step 2 tells us to rejoice. (I guess if you wanted to nitpick, "rejoice" isn't very carefully defined; but I don't think this is really the problem with implementing the above algorithm.)

On the other hand: this is not a terribly useful algorithm. In particular, its steps are in some sense "too big" to be of any use: they reduce the problem of proving the Riemann hypothesis to ... proving the Riemann hypothesis. Typically, we'll want to limit the steps in our algorithms to simple, mechanically-reproducible steps: i.e. operations that a computer could easily perform, or operations that a person could do with relatively minimal training.

In practice, the definition of "simple" depends on the context in which you are creating your algorithm. Consider the following algorithm for making pancakes:

Algorithm.

1. Acquire and measure out the following ingredients:
    - 2 cups of buttermilk.
    - 2 cups of flour.
    - 2 tablespoons of sugar.
    - 2 teaspoons of baking powder.
    - 1/2 teaspoon of baking soda.
    - 1/2 teaspoon of salt.
    - 1 large egg.
    - 3 tablespoons unsalted butter.
    - 1-2 teaspoons of vegetable oil.

2. Whisk the flour, sugar, baking powder, baking soda, and salt in a medium bowl.
3. Melt the butter.
4. Whisk the egg and melted butter into the milk until combined.
5. Pour the milk mixture into the dry ingredients, and whisk until just combined (a few lumps should remain.)
6. Heat a nonstick griddle on medium heat until hot; grease the griddle with a teaspoon or two of oil.
7. Pour 1/4 cup of batter onto the skillet. Repeat in various disjoint places until there is no spare room on the skillet. Leave gaps of 1cm between pancakes.
8. Cook until large bubbles form and the edges set (i.e. turn a slightly darker color and are no longer liquid,) about 2 minutes.
9. Using a spatula, flip pancakes, and cook a little less than 2 minute slonger, until golden brown.
10. If there is still unused batter, go to 5; else, top pancakes with maple syrup and butter, and eat.

This algorithm's notion of "simple" is someone who is (1) able to measure out quantities of various foods, and (2) knows the meaning of various culinary operations like "whisk" and "flip." If we wanted, we could make an algorithm that includes additional steps that define "whisking" and "flipping": i.e. at each step where we told someone to whisk the flour, we could instead have

(a) Grab a whisk. If you do not know what a whisk is, go to this Wikipedia article and grab the closest thing to a whisk that you can find. A fork will work if it is all that you can find.

(b) Insert the whisk into the object you are whisking.

(c) Move the whisk around through the object you are whisking in counterclockwise circles of varying diameter, in such a fashion to mix together the contents of the whisked object.

In this sense, we can extend our earlier algorithm to reflect a different notion of "simple," where we no longer assume that our person knows how to whisk things. It still describes the same sets of steps, and in this sense is still the "same" algorithm – it just has more detail now!

This concept of "adding" or "removing" detail from an algorithm isn't something that will always work; some algorithms will simply demand steps that cannot be implemented on some systems. For example, no matter how many times you type "sudo apt-get 2 cups of flour," your laptop isn't going to be able to implement our above pancake algorithm. As well, there may be times where a step that was previously considered "simple" becomes hideously complex on the system you're trying to implement it on!

For one example of this phenomena, consider the task of multiplying two positive integers together. At first, one would assume that this is a one-step process:

Algorithm. Take as input any two positive integers x, y. Consider the following algorithm:

1. Calculate $x \cdot y$.

But suppose you're trying to teach someone who's never multiplied numbers before! You could attempt the following algorithm:

Algorithm. Take as input any two positive integers x, y. Consider the following algorithm:

1. Define a new number, *prod*, that is initially 0.
2. If $y = 0$, end this process and return prod.
3. Otherwise, if $y \geq 1$, then $x \cdot y = (x - 1) \cdot y + y$. Add y to *prod*, and return to 2.

This is now a much more complicated algorithm! In particular:

- It now takes about $2y$ steps, which can be very large.
- Also, it still assumes the ability to add lots of numbers of very large sizes together many times in a row with no errors, which may be unreasonable.

You might try instead the following somewhat-recursive algorithm, that's more like what you may remember from grade school. This assumes that the audience has the ability to multiply two one-digit numbers together, via times tables or such things.

Algorithm. Take as input any two positive integers x, y. Consider the following algorithm:

1. Let *prod* denote the product of x and y, and assume that prod is initially set to 0. If x and y are single digit numbers, set $prod = x \cdot y$, which we have assumed that we can calculate.
2. Otherwise, write x as a n-digit decimal number $x_n \dots x_1$.
3. For each digit x_i, do the following:
   i. Start a new branch of this process that attempts to calculate $y \cdot x_i$. Denote the result of that process p_i.
   ii. Append $i - 1$ zeroes to the end of p_i.
   iii. Add p_i to *prod*.
4. At the end of the above loop, return *prod*.

How many steps does this algorithm take? At first glance, it's not obvious: so let's look at some sample inputs.

- If x and y are both one-digit numbers: this takes 1 step.
- If x is a n-digit number and y is a one-digit number, it takes $3 + 4n$ steps to calculate $x \cdot y$. This is calculated under the assumption that we have to run the steps i, ii, and iii once each for each digit (i.e. m runs of each step), and that each run of step 3 requires an additional one step (as noted above.)

- Finally, if x is a n-digit number and y is a m-digit number, it takes $3 + m(3 + 4n) + 3m$ steps to calculate $x \cdot y$, under the same assumptions as above.

This kind of estimation is accurate, but not the most useful in many situations: when we're studying an algorithm, we often don't care about the constants around the algorithm (i.e. the $+2$ above.) Instead, we care about the **order of growth** of these runtimes! This leads us to the following definitions:

Definition. Let $f : \mathbb{R} \to \mathbb{R}$ be some function. We say that $f(n)$ is $O(g(x))$, for some other function $g(x)$, if there is some constant C and cutoff point x_0 such that for any $x \geq x_0$,

$$|f(x)| < C|g(x)|.$$

In other words, the function $f(x)$ grows "at most as fast" as $Cg(x)$.

Definition. In these lectures, a **problem** is some sort of general question that we want to find a yes or no answer to, along with some sort of list of associated parameters that (when specified) give a specific **instance** of this problem.

For example, consider the **traveling salesman problem**, which asks (given a list of cities $C = \{c_0, \ldots c_n\}$ and a distance cap B) the following: starting from c_0, is it possible to visit each city exactly once and return to c_0 in such a way that the total distance traveled is less than B? Stated in this way, the parameters of this problem are a list of cities $C = \{c_0, \ldots c_n\}$, a distance function $d : C^2 \to \mathbb{Z}^+$, and a bound B. Similarly, if we consider the task of multiplication, our parameters are just a pair of positive numbers x, y.

Definition. We say that the **input length** of a given problem is the number of parameters needed to specify a given instance of this problem. Note that this process can vary wildly depending on how the inputs to our problem are described.

For example, the input length of the traveling salesman problem as written above is $n + 3$; however, if we were to write its distance function as a collection of m^2 distinct labeled integers instead of as a single function, we would regard it as a different problem with input length $(n + 1)^2 + n + 2$. Similarly, we can think of our multiplication problem as either having input length 2 (i.e. two numbers,) or $\log_{10}(x) + \log_{10}(y)$ (if we think of them as being two strings of digits.)

Definition. Suppose we have a problem P along with some algorithm A that claims to solve that problem. We say that our algorithm A has **complexity** $O(g(n))$ if there is some constant C such that, given an arbitrary input of length n, our algorithm will solve our problem in no more than $Cg(n)$ steps.

For this example, our first algorithm for multiplying numbers has runtime/complexity $O(1)$: it takes a constant number (2) of steps to multiply these two numbers together, regardless of how we interpret the concept of "input length." For our second algorithm, because our algorithm varies in runtime based on the number of digits in the input, we really want to have our notion of "input length" capture this number of digits, as otherwise we will not be able to come up with any good notion of runtime. In this case, if we let the input x, y correspond

to the input length given by the number of digits $\log_{10}(x) + \log_{10}(y)$, we have shown that the runtime is bounded above by $C \cdot \log_{10}(x) \log_{10}(y) \leq C(\log_{10}(x) + \log_{10}(y))^2$. In other words, an input of length n takes Cn^2 steps with which to run our algorithm; therefore, we say that this algorithm has runtime $O(n^2)$ given n digits of input.

Here's a third and arguably better algorithm: "peasant multiplication," a process for multiplying large numbers known since the seventeenth century BC, to allow relatively innumerate people to perform otherwise-difficult calculations. It assumes that its users have the following numerical powers:

- Users can add a number to itself.
- Users can divide a number by two, rounding down.
- Users can calculate the parity (even or odd) of any number.

Algorithm. Take as input any two positive integers x, y. Consider the following algorithm:

1. Define a new number *prod*, and initialize it (i.e. set it equal) to 0.
2. If $x = 0$, stop, and return the number *prod*.
3. Otherwise, if x is odd, set $prod = prod + y$.
4. Regardless of what x was in the step above, set $x = \lfloor x/2 \rfloor$, and $y = y + y$. Then go to step 2.

Roughly speaking, this algorithm succeeds because we can write

$$x \cdot y = \begin{cases} 0, & x = 0, \\ \lfloor x/2 \rfloor \cdot (y + y), & x \text{ even}, \\ \lfloor x/2 \rfloor \cdot (y + y) + y, & x \text{ odd}, \end{cases}$$

and this algorithm is a repeated application of this fact.

We illustrate this algorithm below, on input $x = 123, y = 231$:

run	*prod*	x	y
initialization	0	123	231
1	231	61	462
2	693	30	924
3	693	15	1848
4	2541	7	3696
5	6237	3	7392
6	13629	1	14784
7	28413	0	29568
stop	28413	–	–

Given an input x, y with input size given by the number of digits $\log_{10}(x) + \log_{10}(y)$, the above algorithm needs at most $1 + 3\log_2(x) \leq C \log_{10}(x)$ steps to run: i.e. our algorithm has runtime $O(n)$ given n digits of input.

One relevant thing to note: we've included a sample run of our algorithm after describing it! This is one of the most important things you can do with any algorithm, whenever possible; it can often turn a difficult and confusing writeup into a much clearer solution. Do this!