Math 7h Professor: Padraic Bartlett

Lecture 2: Sorting Algorithms

Week 2 *UCSB 2013*

Hey! This is a minilecture on the notion of **algorithms**; in this talk, we will go over the basics that we'll use in future talks on P vs. NP and other research questions at the intersection of theoretical computer science and mathematics.

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 course.

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!

This is not a terribly useful algorithm. Typically, we'll want to limit the steps in our algorithms to mechanically-reproducible steps: i.e. operations that a computer could easily perform, or operations that a person could do with relatively minimal training.

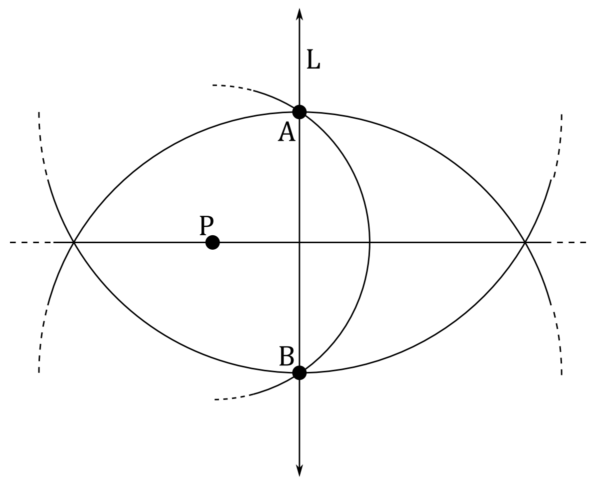
For example, suppose that your space of allowable steps were operations you could perform with a compass and straightedge: specifically, suppose that you can connect points with a straight line using the straightedge, and draw a circle with a given center through any other point with the compass. Then you can solve the following problem:

Problem. Take any line L in $\mathbb{R}^2$ and any point P not on L. Can you construct a line through P that's perpendicular to L?

Algorithm. Take as input any line L and point P not on L, and consider the following process:

1. Pick any point on L. Call it A.
2. Draw a circle centered at P through A.
3. If this circle only intersects L at A and no other point, draw a line through A and P. This line is perpendicular to L.
4. Otherwise, it intersects L at some other point B. Draw a circle through A centered at B, and another circle through B centered at A.
5. These two circles must intersect at two points: find them and call them C, D.
6. Connect C and D. This is a line that is perpendicular to L and that goes through P.

We illustrate a sample run of our algorithm below:



Note that algorithms do not need to contain within themselves a proof that they work; while the above process indeed creates a perpendicular line to L through P, it does not explain the geometry behind why it does so. In general, when we construct algorithms, it's good form to possibly explain after we've done so **why** our algorithm performs the task we claim it does; while we won't always do this in this class, we will attempt to do so whenever we study an algorithm whose functionality isn't immediately obvious. Also, note that 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!

Another example of an algorithm is the process of "peasant multiplication," a process for multiplying large numbers known since the seventeenth century BC that allows relatively innumerate people to perform otherwise-difficult calculations.

Problem. Take any two positive integral numbers x, y, and suppose that the only operations you have are the ability to add a number to itself, divide a number by two (rounding down), and to calculate the parity (even or odd) of any number. Can you efficiently calculate $x \cdot y$?

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 $y = y + y$.

5. 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	–	–

2 Examples of Sorting Algorithms

Throughout the rest of this talk, we're going to primarily focus on **sorting algorithms**: algorithms that take in a list $L = (l_1, \ldots l_n)$ of objects, along with some method $\leq$ of comparing any two elements in our list. Given this, our algorithm will want to output a **sorting** of L: i.e. some permutation $(l'_1, \ldots l'_n)$ of the elements of L, so that $l'_1 \leq l'_2 \leq \ldots l'_n$.

Our first example, **bubblesort**, is a relatively classical and beautiful sorting algorithm:

Algorithm. Take as input some list $L = (l_1, \ldots l_n)$ of objects, along with some operation $\leq$ that can compare any two of these elements. Perform the following algorithm:

1. Create a integer variable *loc* and a boolean variable *didSwap*.

2. Set the variable *loc* to 1, and *didSwap* to *false*.

3. While the variable *loc* is $< n$, perform the following steps:

   (a) If $l_{loc} > l_{loc+1}$, swap the two elements l_{loc}, l_{loc+1} in our list and set $didSwap$ to true.
   (b) Regardless of whether you swapped elements or not, add 1 to the variable loc, and go to 3.

4. If $didSwap$ is $true$, then go to 2.
5. Otherwise, if $didSwap$ is $false$, we went through our entire list and never found any pair of consecutive elements such that the second was larger than the first. Therefore, our list is sorted! Output our list.

A sample run of this algorithm on the list $(1, 4, 3, 2, 5)$ is presented below:

run	step	$didSwap$	loc	L
1	2	$false$	1	$(1,4,3,2,5)$
1	3(a)	$false$	1	$(\mathbf{1},\mathbf{4},3,2,5)$
1	3(b)	$false$	2	$(\mathbf{1},\mathbf{4},3,2,5)$
1	3(a)	$true$	2	$(1,\mathbf{3},\mathbf{4},2,5)$
1	3(b)	$true$	3	$(1,\mathbf{3},\mathbf{4},2,5)$
1	3(a)	$true$	3	$(1,3,\mathbf{2},\mathbf{4},5)$
1	3(b)	$true$	4	$(1,3,\mathbf{2},\mathbf{4},5)$
1	3(a)	$true$	4	$(1,3,2,\mathbf{4},\mathbf{5})$
1	3(b)	$true$	5	$(1,3,2,\mathbf{4},\mathbf{5})$
1	4	$true$	5	$(1,3,2,4,5)$
2	2	$false$	1	$(1,3,2,4,5)$
2	3(a)	$false$	1	$(\mathbf{1},\mathbf{3},2,4,5)$
2	3(b)	$false$	2	$(\mathbf{1},\mathbf{3},2,4,5)$
2	3(a)	$true$	2	$(1,\mathbf{2},\mathbf{3},4,5)$
2	3(b)	$true$	3	$(1,\mathbf{2},\mathbf{3},4,5)$
2	3(a)	$true$	3	$(1,2,\mathbf{3},\mathbf{4},5)$
2	3(b)	$true$	4	$(1,2,\mathbf{3},\mathbf{4},5)$
2	3(a)	$true$	4	$(1,2,3,\mathbf{4},\mathbf{5})$
2	3(b)	$true$	5	$(1,2,3,\mathbf{4},\mathbf{5})$
2	4	$true$	5	$(1,2,3,4,5)$
3	2	$false$	1	$(1,2,3,4,5)$
3	3(a)	$false$	1	$(\mathbf{1},\mathbf{2},3,4,5)$
3	3(b)	$false$	2	$(\mathbf{1},\mathbf{2},3,4,5)$
3	3(a)	$false$	2	$(1,\mathbf{2},\mathbf{3},4,5)$
3	3(b)	$false$	3	$(1,\mathbf{2},\mathbf{3},4,5)$
3	3(a)	$false$	3	$(1,2,\mathbf{3},\mathbf{4},5)$
3	3(b)	$false$	4	$(1,2,\mathbf{3},\mathbf{4},5)$
3	3(a)	$false$	4	$(1,2,3,\mathbf{4},\mathbf{5})$
3	3(b)	$false$	5	$(1,2,3,\mathbf{4},\mathbf{5})$
3	4	$false$	5	$(1,2,3,4,5)$
3	5	–	–	$(1,2,3,4,5)$

One property we like to study in algorithms is the idea of **runtime**. For example, if we put the list $(2, 3, 4, 5, 1)$ into the bubblesort algorithm above, it will take five different passes through our list to sort this into the order $(1, 2, 3, 4, 5)$: specifically, each pass will move the symbol 1 one step over to the left and otherwise not change anything. In general, a list of the form $(2, 3, \ldots n, 1)$ will take n passes, and require us to go through roughly n^2 many steps (up to some smaller-order linear and constant bits of extra work we'll have to do.)

This, roughly speaking, is the worst-case scenario for bubblesort: after one run of bubblesort, we can see from examining the algorithm that the largest element will be in the last place in our list (because as soon as we get to it, we'll always swap it with successive elements until it gets to the last spot.) A second run insures that the second-largest element gets to the second-largest spot, and so on/so forth: therefore, we know that n runs will always suffice.

We say that bubblesort has **worst-case** runtimes of $O(n^2)$, based on this analysis: roughly speaking, the worst bubblesort can do is take n passes, and therefore needs at most some constant $\cdot n^2$ many steps to sort a list. However, there are many lists on which bubblesort will perform markedly better than this: for example, if we have a list of the form $(n, 1, 2 \ldots n-1)$, bubblesort will sort it in exactly one pass, i.e. in $O(n)$ many steps. So, while bubblesort may seem slow in general, it may actually be rather fast on certain kinds of lists! (Formally: we say that some quantity $f(n)$ is $O(n^2)$ if there is some constant M such that $\frac{f(n)}{n^2} \leq M$. Roughly speaking, this says that $f(n)$ grows "like" n^2, up to some fixed constant multiplier. This idea extends to $O(n), O(\log(n)$, and really $O(anything)$.)

Another sort, which is undoubtedly much more awful than bubblesort, is **bogosort**, which we define here:

Algorithm. Take as input some list $L = (l_1, \ldots l_n)$ of objects, along with some operation $\leq$ that can compare any two of these elements. Perform the following algorithm:

1. Check this list to see if it is already in order. (Formally: create some variable *loc*, initialize it at 1, and go through the list comparing elements l_{loc}, l_{loc+1}. If there's any pair that are out of order, stop searching and declare the list out of order; otherwise, continue until you've searched the whole list.)

2. If the list was in order, stop! You're done.

3. Otherwise, the list is out of order. Randomly select a permutation of our list, and reorder our list according to this permutation. Go to 1.

This algorithm is akin to sorting a deck of cards by repeatedly (a) checking if it's in order, (b) throwing it up in the air if it's not, and returning to (a).

If L's elements are all distinct, bogosort has an expected runtime of $n!$; there are $n!$ many ways to permute the elements of our list, while exactly one of them is the correct sorting of L. Similarly, bogosort has a worst-case runtime of ∞; given any finite number of steps t, it's easily possible that our first t shuffles don't accidentally sort our list.

The third sorting algorithm we mention here is **sleepsort**, which is famous mostly for being the only sorting algorithm discovered by 4chan. It runs in "linear" time, for very stupidly defined values of linear, and we define it here:

Algorithm. Take as input some list $\{l_1, \ldots l_n\}$ of n distinct integers. Also, your system should have some sort of timekeeping mechanism (if you're doing this analog, you'd want someone with a stopwatch.)

0. For every number l_i in your list, create a process that once we start our timekeeping mechanism will "sleep" for l_i seconds, and then print itself.

1. Start our clock.

2. After t seconds, all of the numbers $\leq t$ have printed themselves, in order of their size. Consequently, after a number of seconds equal to the size of the largest element in your list, you've written down all of the numbers in order! Win.

Properties of sleepsort:

- Sleepsort's name comes from the **sleep** command-line program, which on input n creates a process that waits n seconds before doing anything else. The following script is an implementation of sleepsort:

```
#!/bin/bash
function f() {
    sleep "$1"
    echo "$1"
}
while [ -n "$1" ]
do
    f "$1" &
    shift
done
wait
```

Assuming no issues with parallelization, this takes as long to run as the largest element in our set.

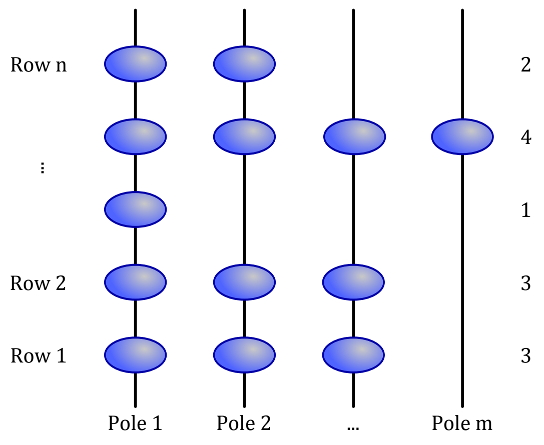
We close with a fourth sort that (while as esoteric in some ways as bogosort) actually has some nice properties and uses: beadsort!

Algorithm. Take as input some list $L = (l_1, \ldots l_n)$ of positive integers; as well, have on hand $m = \sum_{i=1}^n l_i$ many beads, and $\max_{i=1}^n l_i$ many poles on which to place these beads.

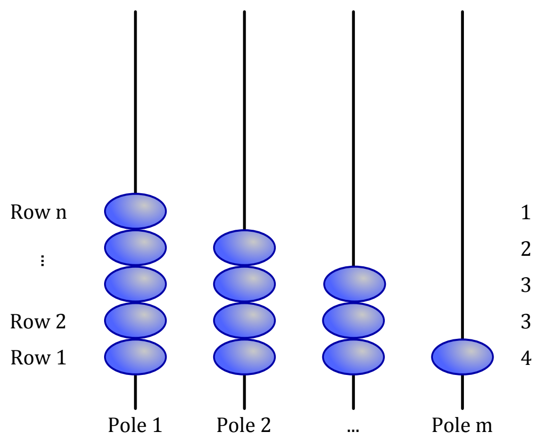
1. Label our poles $1 \ldots m$.
2. For each element l_i in our list, place one bead on each of the poles $1, 2 \ldots l_i$.
3. Let gravity occur.
4. Starting from height n and working your way down, read off the number of beads at that given height. Write down this number as l_k.

This algorithm is perhaps best illustrated with a pair of pictures:

Setup:



Result:



The beautiful thing about this algorithm is that its runtime is (time to place beads on poles) + (time for gravity) + (time to read beads). Gravity takes time roughly on the order of $\sqrt{n}$, while the time to place beads on poles and read beads off of poles takes anywhere between n and m, depending on how efficient you are at it. In many cases, this can be much faster than n^2, when implemented by people or specialized hardware!