Math/CCS 103 — Professor: Padraic Bartlett

Minilecture 2: Sorting Algorithms

Week 1 — *UCSB 2014*

On Monday's homework, we looked at an algorithm designed to sort a list! In this class, we will study several such algorithms, and study their runtimes.

1 Bubblesort

The sorting algorithm from the HW, **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 on the next page:

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	–	–	$(1,2,3,4,5)$

On the HW, you were asked to find the complexity of this algorithm, which (spoiler) is $O(n^2)$. Instead of studying this further, we're going to look at several other sorting algorithms!

Here's a sorting algorithm that is slightly worse than bubblesort: **bogosort**!

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 repeating until it's sorted.

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 study here is **sleepsort**, which is famous mostly for being the only sorting algorithm discovered by 4chan[1]. 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!

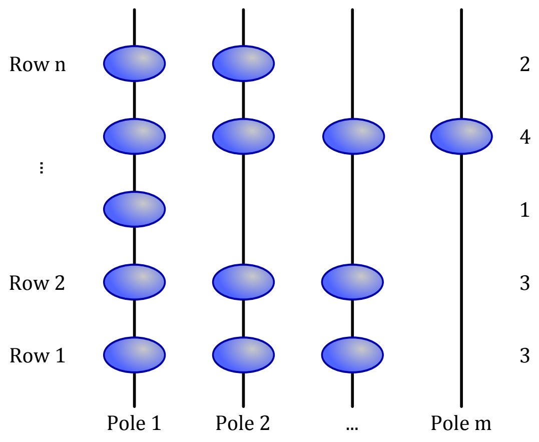
[1]Don't look up 4chan.

Algorithm. Take as input some list $L = (l_1, \dots l_n)$ 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.

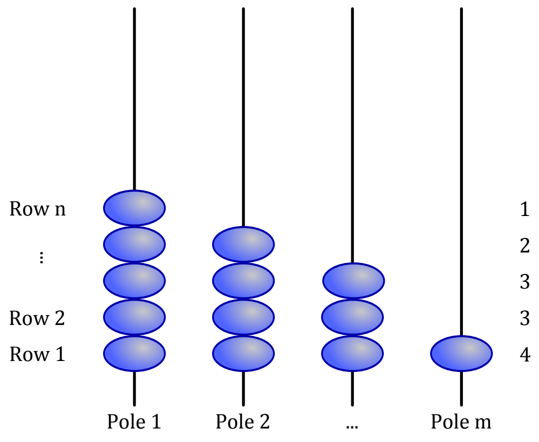
1. Label our poles $1 \dots m$.
2. For each element l_i in our list, place one bead on each of the poles $1, 2 \dots l_i$.
3. Let gravity occur.
4. Starting from height n and working your way down, read off the number of beads at that given height. Write down this number as l_k.

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!

We close here by mentioning a fourth kind of sorting algorithm: quicksort!

Algorithm. Take as input some list $L = (l_1, \dots l_n)$ of positive integers.

1. Pick an element l_k out of our list. Call this a **pivot** element.
2. Take the rest of our list, and split it into two lists: the list $L_<$ of elements less than our pivot, and the list $L_\geq$ of elements that are greater than or equal to our pivot.
3. Quicksort the two lists $L_<$ and $L_\geq$.

Note that this algorithm is recursive in nature: it calls lots of copies of itself to sort a list.

The difficulty with quicksort implementations is in that word "pick" at the first step. Typically, as we will discuss in a bit, quicksort takes something like $O(n \log(n))$ steps to sort our list; however, there are occasional lists where it takes n^2-many steps to sort the list! In particular, if we have any deterministic method that's defined ahead of time for picking out elements, like (say) "always choose the first element of our list for the pivot," it's possible that the list we start from will make us take n runs to sort our list. For example, suppose that we started with the list $(1, 2, \dots n)$:

1. At the start, we choose the element 1 as a pivot. We then split our list into two lists, $L_<$ and $L_\geq$: the first list, by definition, is empty, while the second list is $(2, 3, \dots n)$.
2. We now quicksort these two lists. The empty set is trivially sorted! To quicksort the second list, we choose its first element (2) as a pivot, and split it into two lists $L_< = \emptyset$ and $L_\geq = (3, 4, \dots n)$.
3. ...repeat this process!

This clearly takes n passes, and therefore $O(n^2)$ comparisons, to sort our list. Furthermore, there's no deterministic algorithm that doesn't consider our list that can avoid this problem: given any deterministic algorithm that will pick the $k(L)$-th element out of a list L of n elements, we can put the smallest element of our list at the location $k(L)$, the second-smallest element at the location corresponding to where k would pick an element out of the list $L \setminus$ (smallest element), and so on/so forth[2] This is potentially problematic: if a malicious attacker knew our algorithm and wanted to make our system crash, it could intentionally feed us lots of "worst-case scenario" lists. How do we deal with this?

Again, the answer to this problem is to use randomness! In particular, suppose that we choose our pivot element randomly. What is the expected number of passes we will have to make through any list?

Well, one way to get a rough upper bound is the following: given a list L, call a pivot element p "good" if it's from the middle-half of the elements of L after sorting, and call it "bad" otherwise. The probability that a random pivot is "good" is clearly $1/2$. Furthermore, if we pick such a good pivot, it will cut our list of n elements into two pieces, one of size at most $3n/4$ and the other of size at least $n/4$; otherwise, if we pick a bad pivot, it's at the least the same as not doing anything (i.e. it leaves us with pretty much the same list.)

[2]However, there are deterministic algorithms that can take your list, read through it in $O(n)$ time, and create a set of "good" pivots for us to use! This is a second way around the problem we're describing here, and is pretty efficient, as adding $O(n)$ runtime to a $O(n \log(n))$ algorithm doesn't change the overall nature of its complexity. See http://www.cc.gatech.edu/ mihail/medianCMU.pdf for a discussion of this.

Therefore, if we set $T(n)$ to be the expected amount of steps that it will take to random-quicksort a list with n elements, we have just shown that

$$T(n) \leq n + \frac{1}{2}\left(T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right)\right) + \frac{1}{2}T(n).$$

This is because each choice of pivot takes n comparisons to determine how to split our sets, and we've either chosen a good pivot (in which case we've split our sets into two pieces) or a bad pivot (in which case we've not split our sets at all.) Solving for $T(n)$ gives us

$$T(n) \leq 2n + T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right).$$

Using this, we can solve for an upper bound on $T(n)$: in particular, if we guessed that $T(n)$ was something like $an \cdot \log(n)$, we could prove by induction that $T(n) \leq an \cdot \log(n)$. If $a \geq 2$, this is certainly true for $n = 2$, as it takes us at most 2 steps to quicksort a list of two elements (pick a pivot, the other element is either less than or greater than it.)

So: if we inductively assume that it holds for all values $\leq n$, we get

$$\begin{aligned}
T(n) \leq& 2n + \frac{3an}{4}\log\left(\frac{3n}{4}\right) + \frac{an}{4}\log\left(\frac{n}{4}\right)\\
=& 2n + \frac{an}{4}\log\left(\left(\frac{3n}{4}\right)^3\right) + \frac{an}{4}\log\left(\frac{n}{4}\right)\\
=& 2n + \frac{an}{4}\log\left(\left(\frac{3n}{4}\right)^3 \cdot \frac{n}{4}\right)\\
=& 2n + \frac{an}{4}\log\left(\frac{3^3n^4}{4^4}\right)\\
=& 2n + \frac{an}{4}\log\left(\frac{3^3}{4^4}\right) + \frac{an}{4}\log\left(n^4\right)\\
=& \left(2 + \frac{a}{4}\log\left(\frac{3^3}{4^4}\right)\right)n + an\log(n)\\
\leq& \left(2 + \frac{a}{4}\cdot(-.97)\right)n + an\log(n)
\end{aligned}$$

So, if $a \geq 9$, say, then $1 + \frac{a}{4} \cdot (-.97)$ is less than 0, and therefore we have that this entire quantity is $\leq an\log(n)$, which is what we wanted to prove. So this quicksort has an expected runtime that is $O(n\log(n))$: i.e. up to a constant, it runs like $n\log(n)$! And it's immune to someone intentionally creating "bad" lists, because it's randomized: no matter what list it's fed, it will expect to finish after $\leq 9n\log(n)$ steps.