Lecture 1: Many Campers Sort Piles

1 What Is A Sorting Algorithm?

Definition. A **algorithm** is a list of instructions that explains how to do something. For example, the following algorithm shows how to take two numbers a, b and find their greatest common divisor:

0. Input: two numbers a, b.
1. If $b = 0$, print a.
2. If $a > b$, set $a = a - b$, and return to (1).
3. Otherwise, we have $b \geq a$. Set $b = b - a$, and go to (1).

For our purposes, a **sorting algorithm** is an algorithm that takes in a list of n distinct integers, and returns that same list **in order**. Given a sorting algorithm, there are two quantities we like to measure:

- Expected run time: given a list of numbers, any sorting algorithm will have to go through some number of steps, usually depending on n, the size of the list.
- Worst-case run time: most sorting algorithms have some sort of "worst-case" input list, that can make them take ridiculous amounts of time to run.

Often, the closed-form expressions for these run times are fairly ugly. Because of this, we typically will describe polynomials of the form $c_0 + c_1 n + \ldots c_k n^k$ as just being $O(n^k)$, because the n^k-terms are the "biggest."

So: we've defined what sorting algorithms are. In computer science classes you'll take later, you'll see lots of examples of "practical" sorting algorithms: i.e. quicksort, mergesort, heapsort, and several others.

Here, we're going to talk about some completely impractical sorting algorithms! And really some fairly interesting ones, which are also not entirely practical but in some cases (i.e. given specific special hardware pieces) kind of fascinating and beautiful.

2 Bogosort

Bogosort is the *best* sorting algorithm.

0. Input: a list of integers.

1. Given this list of integers, we first check if the list is sorted. If it is, then we stop! Our list is clearly sorted.

2. Otherwise, **randomly permute** the elements in this list, and go back to (1).

Basically, bogosort is 52-card-pickup turned into a sorting algorithm.

Properties of bogosort:

- Bogosort has an expected run time of $O(n!)$; each time it randomly sorts our lists, i.e. picks one of the $n!$-many various permutations, and sees if this permutation happens to order our list (and there is exactly one of these permutations when our list is made of distinct integers.)

- Bogosort has a worst-case run time of $O(\infty)$, as if we were quite terrifically unlucky it might never randomly pick the right permutation.

3 Spaghettisort

Spaghetti sort is a sorting algorithm that uses **spaghetti**! We define it here:

0. Input: a list of integers; also, a box of dried spaghetti, a hand, and a table.

1. Given this list of integers, find the largest integer m in our list.

2. Define a full piece of spaghetti as m spaghetti-units. Using these units and your box of spaghetti, create a piece of spaghetti of length k for every number k in our list.

3. Take all of your newly-formed spaghetti-integers, put them in your hand, take them loosely in your hand and lower them to the table, so that they all stand upright, resting on the table surface. They are now sorted by height!

4. One by one, pick out the shortest rod of spaghetti and write it down on our list. This process orders our list.

Properties of spaghettisort:

- Spaghettisort has an expected and worst-case run time of $O(n)$: given n numbers, we need n steps to cut down our spaghetti strands, one move to sort the spaghetti strands, and n steps to pick out the spaghetti strands one by one and write down their corresponding lengths.

4 k-stack-sort

k-stack sort is probably the trickiest sort we're going to talk about today. To describe it, we first define a 1-stack-sort:

0. Input: a permutation $a_1, \ldots a_n$ of the integers $\{1, \ldots, n\}$, placed in a stack Perm; also, a currently empty stack Temp.

1. Take a_1 and place it on the top of Temp.

2. Take the top element p of Perm, and compare it with the top element t of Temp. If $p < t$, place p on top of Temp. Otherwise, remove t from Temp, put it into the output, and put p on top of Temp.

3. If our Perm stack is empty, simply output the elements of Temp in order, starting from the top. Otherwise, go to 2 and repeat this process.

We illustrate a sample run of this algorithm here:

Step	Perm	Temp	Output
0	$2, 4, 1, 3$		
1	$4, 1, 3$	2	
2	$1, 3$	4	2
3	3	$1, 4$	2
4		$3, 4$	$1, 2$
5		4	$3, 1, 2$
6			$4, 3, 1, 2$

which ends by having output the sequence $2, 1, 3, 4$ (because it put 2 in the output string first, then 1, then 3, then 4.)

This process (clearly?) doesn't mess with the identity permutation: i.e. it sends the list $1, \ldots n$ to the list $1, \ldots, n$. Therefore, we can define a k-stack-sort as the sorting algorithm that takes a permutation and runs a 1-stack-sort on it k times in a row.

k-stack-sorting has a bunch of interesting properties/open questions around it! We list some of those here:

- A permutation is 1-stack-sortable iff there is not a sequence of three elements a_i, a_j, a_k in our permutation such that $i < j < k$ and $a_k < a_i < a_j$ (i.e. a 231-sequence.)

- You can use this to prove that the number of 1-stack-sortable permutations of n elements is a Catalan number! Bother Kevin when he gets back to see why this is true.

- Categorizing which permutations are k-stack-sortable is still an open question, on which active research is currently being done.

5 Pancakesort

Pancake sort is like spaghetti sort, in that it uses food to sort numbers! Beyond that, not so much. We describe it here:

0. Input: a list of n integers; also, a stack of n pancakes.

1. Take each integer and write it on a pancake.

2. Starting from the bottom of the stack of pancakes and working your way up, find the largest pancake not yet sorted. Take a spatula and put it beneath this pancake; in one move, flip the entire stack of pancakes above this pancake. Then, find the spot where it belongs in our stacks of pancakes, put your spatula in that location, and flip again to put this pancake into its sorted position.

Properties of pancake sort:

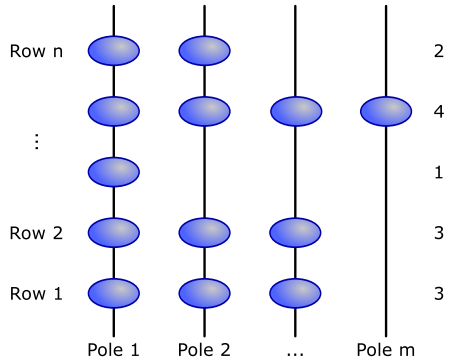
- Pancake sort, as described above, takes at most $2n-3$ flips to sort through a stack of pancakes. (This doesn't count the searching algorithm you need to find the "largest" pancake; counting these steps brings your complexity up to (I think) $O(n^2)$.)
- This sorting algorithm is particularly interesting because it's a way of sorting things with a strongly restricted set of moves: the spatula is the only tool we have, and all it can do is pick some initial segment of our stack of pancakes and reverse this stack.
- There are alternate pancake sorting algorithms (i.e. ways of sorting pancakes using only a spatula) that can bring the number of flips down to $\leq 18n/11$ flips. This proof is fairly notorious because it was the focus of one of the only mathematical papers published by Bill Gates! – we don't have time to discuss it here, but I can point interested readers at the paper if they want to see it.

6 Beadsort

Beadsort is an absolutely beautiful sorting algorithm that actually allows you to sort things in linear time[1], if you have a specific nice way of implementing it in hardware! We describe it here:

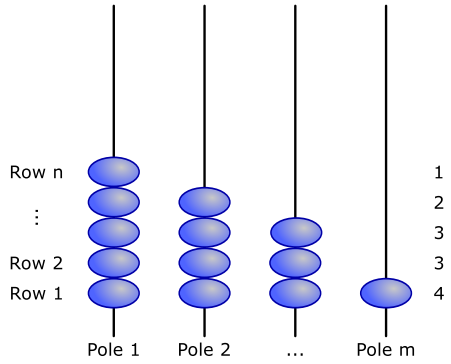
0. Input: a list $\{l_1, \ldots l_n\}$ of n integers; also, $\sum_{i=1}^{n} l_i$ beads, and $l_{max} = \max_{1 \leq i \leq n} l_i$ thin metal rods. Label the metal rods $1, \ldots l_{max}$.

1. For each element l_i in our list, place one bead on each of the rods $1, \ldots l_i$, as depicted below:



[1]This is remarkable because pretty much every algorithm that you can implement by just comparing things takes $O(n \log(n))$ time at best: i.e. this algorithm is markedly faster than quicksort, mergesort, heapsort, and many others!

2. Stand your metal rods up! The beads will all fall, as depicted below:



3. Read the number of beads in each row: these correspond to the numbers in our list, now sorted!

Properties of beadsort:

- Beadsort takes $O(n)$ many steps to run: for each number, placing its beads on the metal strands takes a constant (bounded above by the max size of the elements) amount of time, we have to do this n times, and gravity (the action which sorts our numbers) takes an amount of time proportional to the square root of the maximum height, which is n.

7 Sleepsort

Sleepsort is perhaps most famous 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:

0. Input: a list $\{l_1, \ldots l_n\}$ of n distinct integers. Also, someone with a stopwatch, and another person with a pen.

1. Start your stopwatch.

2. Every time the number of seconds is equal to a number on the list, the person with the stopwatch should shout out the number, and the person with a pen should write it down.

3. 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
```

In this implementation, sleepsort could be argued to always run in a "linear" (i.e. $O(n)$) number of steps, because we just had to run n instances of the sleep command. (HW: why is this a dumb interpretation of the term "linear?")

8 Pigeonsort

Pigeonsort, roughly speaking, is the pigeonhole principle interpreted as a sorting algorithm. It (like the beadsort algorithm) is famous for offering linear sorting times,

0. Input: n pigeons of distinct integer volumes. Failing this, a list of n distinct integers.
1. Let p be the maximum size of your pigeons (or integers, if you're using these.) Create an array of p pigeonholes.
2. Take each pigeon/integer one-by-one from our list, and put it into its pigeonhole.
3. Starting at the top of our pigeonhole array and working our way downwards, write down every value that's present in one of our pigeonholes.

Properties of pigeonsort:

- Pigeonsort takes $O(n + p)$ many steps to run: n steps to find the largest pigeon, n steps (one per pigeon) to put each pigeon into its hole, and p steps to go through the pigeonholes and write down the entries that are present. This, in certain cases, can be far far faster than quicksort or similar things!