TA Lab Week 4 – Sorting

The goal of this lab is to work through some non-conventional sorting algorithms, and to help improve your algorithmic thinking. While many sorting algorithms are implemented and designed for speed and efficiency, others are designed to perform operations in a way which is significant and meaningful. The fastest or most efficient solution is not always the best one for a given problem.

Exercise 1 – Pancake Sorting

Pancake Sort is a colloquial term for the mathematical problem of sorting a disordered stack of pancakes in order of size. A spatula can be inserted at any point `i` in the stack and used to flip all pancakes above it.

Given an unsorted array, design and implement the pancake algorithm to sort it using only the "flip" operation, whose effect is to reverse the elements of the array between elements 0 and `i` (with `i` being the position where you imagine having inserted the spatula for flipping). Whereas a traditional sorting algorithm attempts to sort with the fewest comparisons possible, the goal here is to sort the sequence in as few "flips" as possible. Your algorithm should return the indices at which flips were performed.

Example:
Unsorted Input: `[3,2,4,1]`
Sorted Output: `[1,2,3,4]`
Algorithm Output: `[1, 2, 3]` ← indices where flips where performed (3 flips total).

We have performed 3 flips (bold indicates it has been flipped):

```
Start:              [3, 2, 4, 1]
1st flip (k = 1): [2, 3, 4, 1]
2nd flip (k = 2): [4, 3, 2, 1]
3rd flip (k = 3): [1, 2, 3, 4]
```

Hint

Intuitively, this problem can be solved by:

- Finding the largest out-of-order value
- Flip that largest unsorted value to the bottom (you may need to flip it to the top first)
- Repeat until the pancake stack is ordered

Exercise 2 – Applied, Advanced Pancake Flipping

Perhaps the most common type of genome rearrangement is an inversion, which flips an entire interval of DNA found on the same chromosome. Often, we would like to determine the minimum number of inversions that have occurred on the evolutionary path between two chromosomes.

Occasionally, inversions in DNA are the cause for divergence between species. DNA is read in one direction by the proteins in your body which transcribe DNA; as such, you can probably guess how the reversal or inversion of an entire section of DNA may cause notable changes to an organism.

One other way of computing the difference between two genetic strings is the Hamming distance; the number of positions in the pair of strings which are different. This is a simple, and easy to implement distance metric, however it does not always provide the insight we may be looking for. For example, consider the following two DNA strings:

ACGTTCGAG**C**CCA
CTTGCCGAG**A**CCA

We note a Hamming distance of 6 (bold illustrates differences). However, we can also see that these strings are only 2 inversions away from each other (bold indicates an inversion):

CTTGC**CGAGA**CCA -> **CTTGCA**GAGCCCA -> ACGTTCGAGCCCA

Clearly, the problem here is different, and noticeably more complex. Here, we want to know how to go from the start to the end of the sequence of DNA with the *minimum* number of inversions or reversals.

This motivates our more advanced version of pancake sorting: *sorting by reversals*. The computational problem of sorting by reversals asks that we provide a minimum list of reversals which transform one permutation into another. A reversal of a permutation creates a new permutation by inverting some interval within the permutation. As an example;

`[5,2,3,1,4]`, `[5,3,4,1,2]`, and `[4,1,2,3,5]` are reversals of `[5,3,2,1,4]`.

This is notably more complex than the problem posed in the prior exercise wherein we always flipped at 0 and i. When sorting by reversals, we may flip between any two indices i and j instead. The reversal distance between two permutations A and B, written $d_{reversal}(A, B)$, is the minimum number of reversals required to transform A into B (this assumes that A and B have the same length).

Implement a sort by reversal, and make sure your function returns the reversals performed (with both indices i and j) by outputting the indices at which these reversals were performed. If multiple collections of minimal reversals exist, return any such one.

Example:
Input: `[1, 8, 9, 3, 2, 7, 6, 5, 4, 10]`
Output: `[[2, 5], [4, 9]]`

We have performed 2 flips (bold indicates it has been reversed);

```
Start:            [1, 8, 9, 3, 2, 7, 6, 5, 4, 10]
1st flip (2, 5): [1, 2, 3, 9, 8, 7, 6, 5, 4, 10]
2nd flip (4, 9): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Hint

This type of sort may be achieved, naively, by a greedy algorithm which chooses the “best” reversal at each step – however this often results in a bubble-sort-like behaviour. See if you can improve on this form of the algorithm.

One such way to improve on this is to consider the input as having “breakpoints”, wherein elements of the input are neighbours in the current permutation but are not in the sorted list.

In our example above, the initial list has 4 breakpoints (denoted by vertical bars):

```
[1 | 8 9 | 3 2 | 7 6 5 4 | 10]
```

Therefore, we may consider a “good” reversal as one which eliminates 2 breakpoints. As such, an improved version seeks to minimize the number of breakpoints left after a reversal. However, you may encounter situations in which no flips remove breakpoints (e.g. `[1, 5, 6, 7, 2, 3, 4, 8, 9, 10]`). Make sure that your algorithm does not get stuck there!