CS 161 (Stanford, Winter 2026) Homework 3

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we look for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material. Please do not distribute this material on any public forum.
What we expect: Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Exercises. The following questions are exercises. We suggest you do these on your own. As with any homework question, though, you may ask the course staff for help.

1 Finding Medians

Given two arrays of length n, find the median of the elements in the combined array, i.e., the concatenation of these two arrays.

1.1 (2 pt.)

If the given arrays are unsorted, describe an algorithm that returns the median of the combined array and explain why it has the best possible runtime.

[We are expecting: A brief description of your algorithm, its runtime, and an informal explanation of why it's the best possible runtime.**]**

Solution

1.2 (4 pt.)

If the given arrays are sorted, can you achieve a better runtime? Assume that your algorithm's input is two sorted arrays, each of the same length $n > 0$.

[We are expecting: A brief English description, pseudocode, and an explanation of your algorithm's runtime.**]**

2 Randomized Algorithms (12 pt.)

In this exercise, we'll explore different types of randomized algorithms. We say that a randomized algorithm is a **Las Vegas algorithm** if it is always correct (that is, it returns the right answer with probability 1), but the runtime is a random variable. We say that a randomized

algorithm is a **Monte Carlo algorithm** if there is some probability that it is incorrect. For example, QuickSort (with a random pivot) is a Las Vegas algorithm, since it always returns a sorted array, but it might be slow if we get very unlucky.

We will revisit our population of quaggas and zebras from last week's assignment to gain more insight into randomized algorithms. Assume that there is a **strict majority** of zebras in the population. Additionally, assume you are given a function *isZebra(p)*, which runs in constant time, such that it returns true if animal p is a zebra and returns false otherwise. Also assume that choosing a random animal takes constant time. Here are three algorithms to find a zebra.

Algorithm 1: findZebra1

```
Input: A population P of n animals
while true do
    Choose a random p ∈ P;
    if isZebra(p) then
        return p;
```

Algorithm 2: findZebra2

```
Input: A population P of n animals
for 100 iterations do
    Choose a random p ∈ P;
    if isZebra(p) then
        return p;
return a random p ∈ P;
```

Algorithm 3: findZebra3

```
Input: A population P of n animals
/* Order the animals in P randomly, and assume it takes O(n) time to
   do so.                                                          */
for p ∈ P do
    if isZebra(p) then
        return p;
```

Decide their types (Monte Carlo or Las Vegas), their expected/worst-case runtime, and their correct probabilities by filling in the following table.

Algorithm	Monte Carlo or Las Vegas?	Expected runtime	Worst-case runtime	Probability of returning a zebra
findZebra1				
findZebra2				
findZebra3				

[We are expecting: Your filled in-table, and a short justification for each entry in the table. You may use Big-O notation for the runtimes. For the "probability of returning a zebra" column, provide the tightest bound you can get from the given information.**]**

Problems. The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

3 QuickSort with Duplicate Entries

In the lecture, when studying QuickSort, we assume that the array A consists of distinct numbers. Now, you want to implement QuickSort to sort your own array, which may include duplicates of the same number.

3.1 (3 pt.)

In each iteration of QuickSort, let x be the pivot chosen uniformly at random. Instead of partitioning the rest of the elements into $A_< = \{\text{elements} < x\}$ and $A_> = \{\text{elements} > x\}$, you decide to partition them into $A_\leq = \{\text{elements} \leq x\}$ and $A_> = \{\text{element} > x\}$, and then recursively sort $A_\leq$ and $A_>$ after replacing A by $[A_\leq\ x\ A_>]$.

Construct an array to demonstrate that the **expected runtime** of the QuickSort implemented as described above is $\Theta(n^2)$. Namely, the implemented QuickSort doesn't even retain its expected runtime guarantee!

[We are expecting: An array that may contain duplicate entries, and an analysis of the expected runtime of the implemented QuickSort on this array.**]**

3.2 (7 pt.)

Modify the partition step so that the expected runtime of the resulting QuickSort becomes $O(n \log n)$ even for arrays with duplicate elements. Moreover, what are the expected and worst-case runtimes of the updated QuickSort on the array constructed in the previous sub-problem?

Hint: Try to argue the expected runtime by comparing to QuickSort with distinct elements.

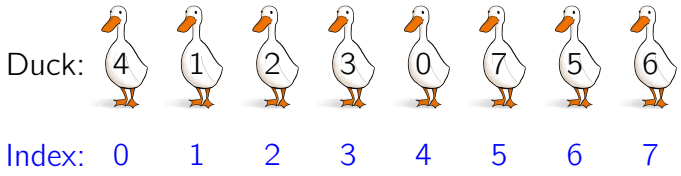
[We are expecting: Pseudocode for the updated QuickSort **AND** a clear English description of the algorithm. Additionally, an informal justification of the $O(n \log n)$ expected runtime,

and an explanation of the expected and worst-case runtimes on the array constructed in the previous subproblem.**]**

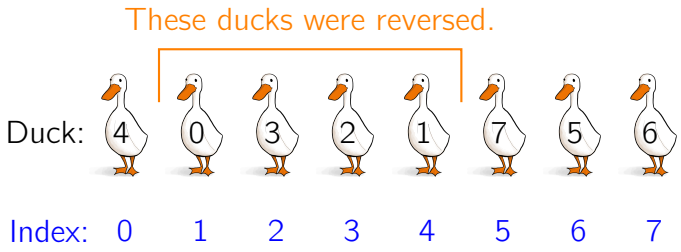
4 Dancing Ducks

You have encountered a troupe of n dancing ducks who are each labeled with a number $0, \ldots, n-1$ in some order, where n is a power of 2. The i^{th} duck is labeled with the number $A[i]$.

The n ducks dance in a line, and they only know one type of dance move, called `Flip(A,i,j)`: for any i and j such that $0 \leq i < j \leq n$, `Flip(A,i,j)` flips the order of the ducks standing in positions $i, i+1, \ldots, j-1$ in array A. For example, if the ducks started out like this:



then after executing `Flip(A,1,5)`, the ducks would look like this:



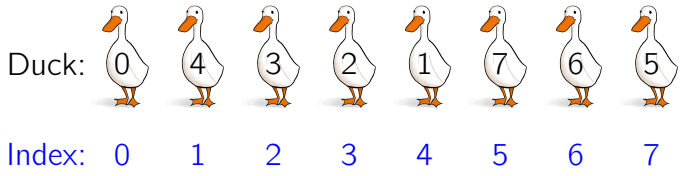
Executing this dance move is pretty complicated: it takes the ducks $O(j-i)$ seconds to perform `Flip(A,i,j)`.

In this problem, you will design and analyze a duck dance (a.k.a. a sorting algorithm) that solely uses the `Flip` move (after all, it's the only dance move ducks know) such that after finishing the dance, all n ducks are sorted by their labels.

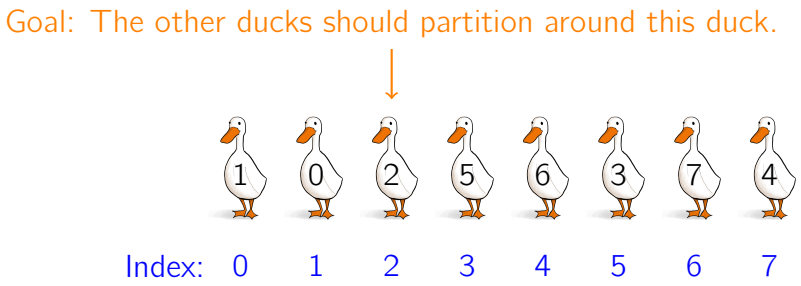
The input to your algorithm will be an array A that contains the ducks' values. As an example, if the ducks were organized as pictured above after the flip, then the input array would be $A = [4, 0, 3, 2, 1, 7, 5, 6]$. Your algorithm can do whatever computations it wants, but the only way you can get the ducks to move is by calling `Flip(A,i,j)`, which will make them do their `Flip(A,i,j)` dance move. The algorithm will wait until the ducks are done dancing to continue its computations. Thus, the total running time of the algorithm includes *both* the time spent computing, and the time that the ducks spend dancing.

4.1 (10 pt.)

First, you will design an algorithm that tells the ducks how to perform a dance called `Partition(A, i)`. For this dance, the ducks will partition themselves around the duck in position i, so that all ducks whose labels are smaller than the duck's label at position i will end up to the left, and all the ducks whose labels are larger will end up to the right. For example, if the ducks were initially ordered as so:



then after executing `Partition(A,3)`, the ducks partition themselves around the duck in position 3, whose label is 2. The resulting ducks formation could look like this:



Notice that it doesn't matter what order the smaller ducks and larger ducks are in.

Give an algorithm that instructs the ducks to implement `Partition` that runs in total time $O(n \log n)$. Remember, you can only move the ducks around with calls to `Flip`.

Hint: Try divide-and-conquer.

[We are expecting: Pseudocode **AND** a clear English explanation of what your algorithm is doing. Additionally, an informal justification of the running time. You may appeal to the master theorem if it is relevant. **]**

4.2 (1 pt.)

You are excited about the `Partition` algorithm from part (a), because it allows you to tell the ducks how to perform a dance called `Sort()`, which puts the ducks in sorted order. The algorithm is as follows:

```
def Sort(A):
    // A is an array of length n, with the positions of the n ducks
    // Assume that n is a power of 2.

    // Base case:
```

```
if n == 1:
    return

// Get the index of the median, using the Select algorithm from lecture 4
i = Select(A,n/2)

// Tell the ducks to partition themselves around the i'th duck:
Partition(A,i)

// Recurse on both the left and right halves of the duck array.
Sort(A[:n/2])
Sort(A[n/2:])
```

That is, this algorithm first partitions the ducks around the median, which puts the smaller-labeled ducks on the left and the larger-labeled ducks on the right. Then it recursively sorts the smaller-labeled ducks and the larger-labeled ducks, resulting in a sorted list.

Let $T(n)$ be the running time of `Sort(A)` on an array A of length n. Write down a recurrence relation that describes $T(n)$.

[We are expecting: A recurrence relation of the form $T(n) = a \cdot T(n/b) + O(\text{something...})$, and a short explanation.**]**

4.3 (3 pt.)

Explain, without appealing to the master theorem, why the running time of `Sort(A)` is $O(n \log^2 n)$ on an array of length n. (Note: "$\log^2 n$" means $(\log n)^2$).

If it helps, you may ignore the big-Oh notation in your answer in part (b). That is, if your answer in part (b) was $T(n) = aT(n/b) + O(something)$, then you may drop the big-Oh and assume that your recurrence relation is of the form $T(n) = aT(n/b) + something$. You may assume that $T(1) \leq 1$, $T(2) \leq 2$. Recall that we are assuming that n is a power of 2.

Hint: Either the tree method or the substitution method is a reasonable approach here.

[We are expecting: An explanation. You do not need to give a formal proof, but your explanation should be convincing to the grader. You should **NOT** appeal to the master theorem, although it's fine to use the "tree method" that we used to prove the master theorem.**]**