CS 161 **Problem Set 2**
Spring 2023 **Due:** Wednesday, April 19 at 1pm on Gradescope.

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

Please list all members of your homework group and all other students in the class you have collaborated with on this homework, in accordance with the Collaboration Policy, at the beginning of each homework submission.

Exercises

We recommend you do the exercises on your own before collaborating with your group.

---

1. **(6 pt.)** Compute the *best* (that is, the smallest) upper-bound that you can on the following recurrence relations. You may use any method we've seen in class (Master Theorem, Substitution Method, algebra, etc.)

   (a) **(2 pt.)** $T(n) = 2T(\lfloor n/2 \rfloor) + O(\sqrt{n})$, where $T(0) = T(1) = 1$.
   (b) **(2 pt.)** $T(n) = T(n-2) + 1$, where $T(0) = T(1) = 1$.
   (c) **(2 pt.)** $T(n) = 6T(\lfloor n/4 \rfloor) + n^2$ for $n \geq 4$, and $T(n) = 1$ for $n < 4$.

   [**We are expecting:** *For each part, a statement of your answer of the form "$T(n) = O(\_\_\_\_)$", as well as a short justification. You don't need to give a rigorous proof, but your justification should be convincing to the grader.*]

2. **(3 pt.)** Consider the following algorithm, which takes as input an array A:

```
def printStuff(A):
    n = len(A)
    if n <= 4:
        return
    for i in range(n):
        print(A[i])

    printStuff(A[:(n//3)]) # recurse on the first floor(n/3) elements of A
    printStuff(A[(n//3):2*(n//3)]) # recurse on the second floor(n/3) elements of A
    return
```

What is the asymptotic running time of `printStuff` on an array of length n?

[**We are expecting:** *The best answer you can give of the from "The running time of* ***printStuff****(n) is $O(\_\_\_\_)$, and a short explanation, including a clear statement of the relevant recurrence relation.*]

3. **(4 pt.)** Let $T(n) = T(\lfloor\sqrt{n}\rfloor) + 1$, where $T(n_{\text{small}}) = 1$ for all $0 \leq n_{\text{small}} \leq 9$. Prove formally that $T(n) = O(\log \log n)$.

   You can use any method you like.

   [**Hint:** *The substitution method is a good approach! If you do that, make sure that you clearly state your inductive hypothesis, base case, inductive step and conclusion. You could also try the Master Theorem...but (hint hint) you might need to manipulate the recurrence relation a bit to get it to apply...*]

   [**We are expecting:** *A formal proof. You may assume that n is 2^{2^m} for some positive integer m, if you like.*]

Problems

4. **(6 pt.)** [**Fishing.**] Plucky the Pedantic Penguin is fishing. Plucky catches n fish, but plans to keep only the k largest fish and to throw the rest back. Plucky has already named all of the fish, and has measured their length and entered it into an array F of length n. For example, F might look like this:

$$F = \begin{bmatrix} \text{(Frederick the Fish, 14.2in)} \\ \text{(Fabiola the Fish, 10in)} \\ \text{(Farid the Fish, 12.35in)} \\ \vdots \\ \text{(Felix the Fish, 6.234523in)} \\ \text{(Finlay the Fish, 6.234524in)} \end{bmatrix}$$

   (a) **(3 pt.)** Give a simple $O(n \log(n))$-time deterministic algorithm that takes F and k as inputs and returns a list of the names of the k largest fish. You may assume that the lengths of the fish are distinct.

   Note: Your algorithm should run in time $O(n \log(n))$ even if k is a function of n. For example, if Plucky wants to keep the largest $k = n/2$ fish, your algorithm should still run in time $O(n \log(n))$.

   [**We are expecting:** *Pseudocode* ***AND*** *a short English description of your algorithm. You may (and, hint, may want to...) invoke algorithms we have seen in class. You do not need to justify why your algorithm is correct or its running time.*]

   (b) **(3 pt.)** Give an $O(n)$-time deterministic algorithm that takes F and k as inputs and returns a list of the names of the k largest fish. You may assume that the lengths of the fish are distinct. Your algorithm should also be fundamentally different than your algorithm for part (a). (That is, don't solve part (b) and then use that as your solution to part (a); part (a) should be easier).

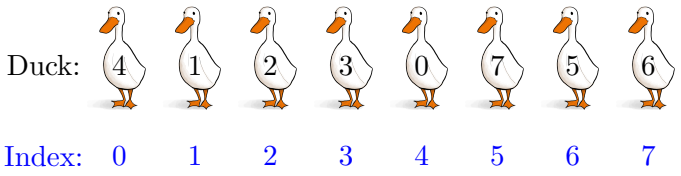
   Note: Your algorithm should run in time $O(n)$ even if k is a function of n. For example, if Plucky wants to keep the largest $k = n/2$ fish, your algorithm should still run in time $O(n)$.

   [**We are expecting:** *Pseudocode* ***AND*** *a short English description of your algorithm. You may (and, hint, may want to...) invoke algorithms we have seen in class. You do not need to justify why your algorithm is correct or its running time.*]

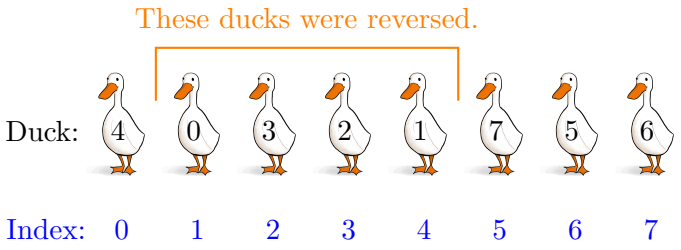
5. **(14 pt.) [Dancing ducks.]**

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:

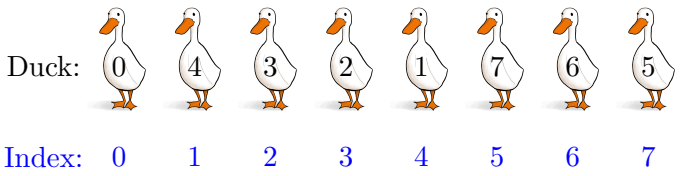


Executing this dance move is pretty complicated: it takes the ducks $O(j-i)$ seconds to perform `Flip(A,i,j)`.

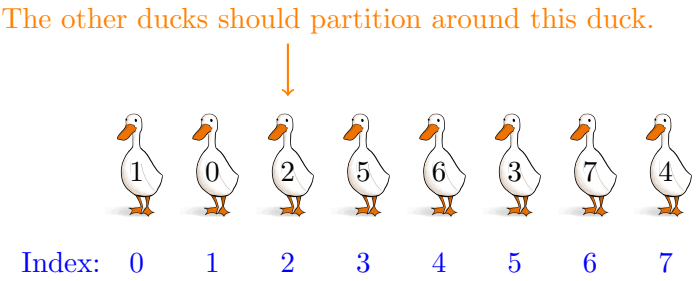
In this problem, you will design and analyze a duck dance (a.k.a. a sorting algorithm) that solely uses the `Flip` move (after all, it's the only dance move ducks know) such that after finishing the dance, all n ducks are sorted by their labels.

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.

(a) **(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.*]

(b) **(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* $T(n) = a \cdot T(n/b) + O(something...)$*, and a short explanation.*]

(c) **(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.*]