Exercises

Please do the exercises on your own.

1. **(2 pt.)** In your pre-lecture Exercise for Lecture 3, you saw two different ways to solve the recurrence relation $T(n) = 2 \cdot T(n/2) + n$ with $T(1) = 1$. We saw that $T(n)$ is exactly $n(1 + \log n)$, when n is a power of two. In this exercise, you'll do the same for a few variants.

   (a) What is the exact solution to $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$ with $T(1) = 3$, when n is a power of 2?

   (b) What is the exact solution to $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 3n$ with $T(1) = 1$, when n is a power of 2?

   [**We are expecting:** *Your answer; no justification required. Notice that we want the exact answer, so don't give a $O()$ statement.*]

2. **(2 pt.)** Consider the recurrence relation $T(n) = T(n - 10) + n$ for $n > 10$, with $T(n) = 1$ for all $n \leq 10$. Your friend claims that $T(n) = O(n)$, and offers the following justification:

   > Let's use the Master Theorem with $a = 1$, $b = \frac{n}{n-10}$, and $d = 1$. This applies since, for any $n > 10$, we have
   > $$\frac{n}{b} = n \cdot \left(\frac{n - 10}{n}\right) = n - 10.$$
   > Then for any $n > 10$, we have $a < b^d$, so the Master Theorem says that $T(n) = O(n^d) = O(n)$.

   What's wrong with your friend's argument, and what is the correct answer? If it helps you may assume that n is a multiple of 10.

   [**HINT:** *It is totally fine to apply the Master Theorem when b is a fraction; that's not the problem.*]

   [**We are expecting:** *A clear identification of the faulty logic above; your solution to this recurrence (you may use asymptotic notation[1]), and a short but convincing justification.*]

3. **(8 pt.)** Use any of the methods we've seen in class so far to solve the following recurrence relations. You may either treat fractions like $n/2$ as $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$, or just as real numbers (not integers), whichever you prefer.

   (a) **(2 pt.)** $T(n) = T(n/3) + n^2$, for $n > 3$, and $T(n) = 1$ for $n \leq 3$.

   (b) **(2 pt.)** $T(n) = 2T(n/2) + 10 \cdot n + 4$, for $n > 2$, and $T(n) = 1$ for $n \leq 2$.

   (c) **(4 pt.)** $T(n) = T(n/2) + T(n/4) + n$ for $n > 4$, and $T(n) = 1$ for $n \leq 4$.

   [**We are expecting:** *The answer using asymptotic notation, and a justification. You do not need to give a formal proof, but your justification should be convincing to the grader. You may use the master theorem if it applies.*]

[1] Unless specified otherwise, in every problem set, when we ask for an answer in asymptotic notation, we are asking either for a $\Theta(\cdot)$ result, or else the tightest $O(\cdot)$ result you can come up with.

Problems

You may talk with your fellow CS161-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.

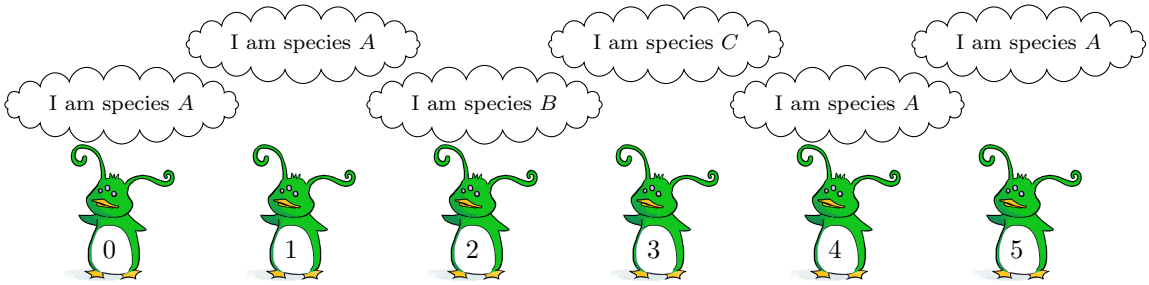
---

4. **(8 pt.) [Application to interstellar ecology.]** On the same alien planet you landed on in HW1, there are n aliens of many different species. The differences between the species are very subtle, so without help you can't tell the aliens apart at all. Fortunately, you have an expert with you, and she can tell you whether or not two aliens belong to the same species. More precisely, she can answer queries of the form:

$$\texttt{isTheSame(alien1, alien2)} = \begin{cases} \textbf{True} & \text{if } \texttt{alien1} \text{ and } \texttt{alien2} \text{ belong to the same species} \\ \textbf{False} & \text{if } \texttt{alien1} \text{ and } \texttt{alien2} \text{ belong to different species} \end{cases}$$

The only way you can get any information about the aliens is by running `isTheSame`. You cannot ask them what species they are, or compare them in any other way.

The expert assures you that one species of alien is in the majority. That is, there are *strictly greater* than $n/2$ aliens of that species. Your goal is to return a single member of that majority species. For example, if the population looked like this:



then species A is in the majority, and your algorithm should return any one of Aliens 0, 1, 4, or 5.

If there is no species with a strict majority, your algorithm may return whatever it wants.

(a) **(4 pt.)** Design a deterministic divide-and-conquer algorithm which uses $O(n \log n)$ calls to `isTheSame` and returns an alien belonging to the majority species. You may assume that n is a power of 2 if it is helpful.

[**We are expecting:** *Pseudocode (which calls* `isTheSame`*)* ***AND*** *a clear English description of what your algorithm is doing.*]

(b) **(1 pt.)** Explain why your algorithm calls `isTheSame` $O(n \log n)$ times.

[**We are expecting:** *A short justification of the number of calls to* `isTheSame`*. You may invoke the Master Theorem if it applies.*]

(c) **(3 pt.)** Prove, using an argument by induction, that your algorithm is correct.

[**We are expecting:** *A rigorous proof by induction.*]

(d) **(1 BONUS pt.)** Is $O(n \log n)$ the best guarantee you can come up with? Either give an asymptotically faster algorithm which finds a majority-species alien, or else prove that no such algorithm exists. (Or, if you think that your algorithm from part (a) already does better than $\Theta(n \log n)$, just write that :)).

[**We are expecting:** *Nothing, this part is not required. To get the bonus point, you should include one of the following:*

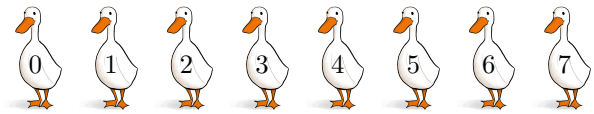
- *Pseudocode, a clear English description, and the best big-Oh running time for your algorithm that you can come up with, or*
- *a proof that any algorithm for finding a member of the majority species must make at least* $\Omega(n \log n)$ *calls to* `isTheSame`*, or*
- *a statement that your algorithm from part (a) is faster than* $O(n \log n)$*, and a correct explanation why if not already included in parts (b) and (c)*

]

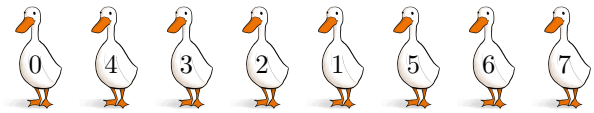
5. **(14 pt.) [Dancing ducks.]**

You have encountered a troupe of n dancing ducks who are each labeled with a number $0, \ldots, n-1$, where n is a power of 2.

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



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



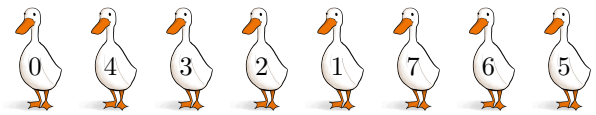
Executing this dance move is pretty complicated: it takes the ducks $O(|i-j|)$ seconds to implement `Flip(i,j)`.

In this problem, you will design and analyze an algorithm to sort the n ducks by their labels, which uses `Flip`, the only move the ducks know. However, you don't know the original order of the ducks until you see them, and so you may also need to spend some extra time in between the `Flip` operations computing which indices i and j you want to call `Flip` on.

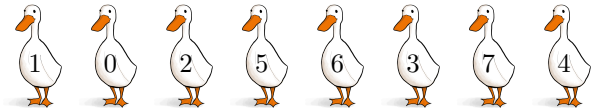
Below, you will consider algorithms which take as input an array that contains the duck's initial positions (for example, if the ducks were originally organized as the picture above (after the flip), then the input array would be $A = [0, 4, 3, 2, 1, 5, 6, 7]$). In addition to normal computation, which can be used to compute indices i and j, your algorithm is allowed to call `Flip(i,j)`, which will cause the

ducks to implement the `Flip(i,j)` dance. The algorithm will wait until the ducks are done executing that dance move 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(i)`. For this dance, the ducks will partition themselves around the duck in position i, so that all of the ducks whose labels are smaller than that duck's label will end up to the left, and all the ducks whose labels are larger will end up to the right. For example, if the situation were this:



then after executing `Partition(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 time $O(n \log n)$.

[**HINT:** *Try divide-and-conquer.*]

[**We are expecting:**

- *Pseudocode* ***AND*** *a clear English explanation of what your algorithm is doing.*
- *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(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 ducks and the larger 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.*]

Feedback

There's no "correct" answer here, and it is completely anonymous.

6. **(1 pt.)** Please fill out the following poll, which asks about the length and difficulty of the homeworks so far:

   `https://forms.gle/ocbbWxrdZKUXB2iu8`

   Did you fill out the poll?

   [**We are expecting:** *The answer "yes."*]