CS 161 (Stanford, Winter 2023) Homework 3

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. You should cite all sources you used outside of the course material.
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 Median of two arrays

Given two arrays of length n, find the median of all elements of the two arrays.

1.1 Unsorted arrays (2 pt.)

If the 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 runtime and a brief description of your algorithm.**]**

1.2 Sorted arrays (8 pt.)

If the arrays are sorted, can you achieve a better runtime?

[We are expecting: A short English description, Pseudocode, runtime analysis. You may assume that the length of the arrays is a power of 2.**]**

2 Dinosaur demographics (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, if it returns, then it returns the right answer), but the running time 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.

In this problem, we will consider a population of n dinosaurs. We would like to discover a dinosaur species that appears at least k times in our population. (We will call such a species

a *common species*). We are guaranteed that exactly one such species exists. You can't tell the dinosaurs apart, but there's an expert paleontologist who can, and she can answer queries to `isTheSame(d1,d2)` which returns True if `d1` and `d2` are dinosaurs of the same species, and False otherwise. Consider the three algorithms below, all of which call the subroutine `countSpecies` (also below) and all of which attempt to find a common species.

Algorithm 1: findCommonDinoTriassic

Input: A population D of n dinos, a commonness goal k
while *true* **do**
    Choose a random $d \in D$;
    if countSpecies*(d)* $\geq k$ **then**
        return d;

Algorithm 2: findCommonDinoJurassic

Input: A population D of n Dinos, a commonness goal k
for *100 iterations* **do**
    Choose a random $d \in D$;
    if countSpecies*(d)* $\geq k$ **then**
        return d;
return $D[0]$;

Algorithm 3: findCommonDinoCretaceous

Input: A population D of n dinos, a commonness goal k
Put the dinos in D in a random order.;
`/* Assume it takes time` $\Theta(n)$ `to put the` n `dinos in a random order */`
for $d \in D$ **do**
    if countSpecies*(d)* $\geq k$ **then**
        return d;

Algorithm 4: countSpecies

Input: A population D of n dinos and an dino $d \in D$
Output: The number of dinos in d's dinosaur species (including d itself.
count $\leftarrow 0$;
for $q \in D$ **do**
    if isTheSame*(d,q)* **then**
        count ++;
return *count ;*

Figure 1: Three randomized algorithms for finding a common dino

Fill in the table below.

Algorithm	Monte Carlo or Las Vegas?	Expected running time	Worst-case running time	Probability of returning a common dino
Algorithm 1				
Algorithm 2				
Algorithm 3				

[We are expecting: Your filled in-table. Your answers should be in terms of n and k. You must use asymptotic notation for the running times, as usual ignoring constants and lower order terms. For the probability of returning a common dino, give the **tightest** bound that you can given the information provided (i.e., your bound should still hold with the **worst case** inputs that still meet the conditions outlined above). No justification is required. **]**

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.

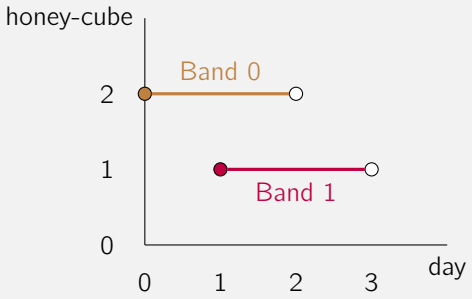
3 Lyric the Bee

Lyric the music-loving bee sometimes goes to musical concerts. Lyric loves listening to the performances of n musical bands. However, Lyric can attend at most one concert during a given day. Each musical band has a range of days when they put up concerts: between a_i and b_i (including a_i and not including b_i), and they charge h_i honey-cubes per ticket. Here, a_i,b_i,h_i are all positive integers and $a_i \leq b_i$. Each day, Lyric chooses to attend the least expensive concert. If there is no concert organized by one of Lyric's favourite musical bands on a day, Lyric does not attend any concert and thus spends 0 honey-cubes.

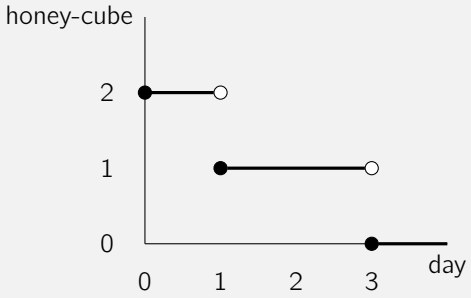
Lyric gets the concert schedules (a_i, b_i, h_i) as inputs, and wants to make a plot of how many honey-cubes will be spent each day. To understand the format Lyric wants the output in, see the example below.

Example: Suppose that $n = 2$. As input, Lyric would get the following data, which can be visualized as the graph below.

Band i	a_i	b_i	h_i
Band 0	0	2	2
Band 1	1	3	1



In this example, Lyric would attend for Band 0 at day 0, and pay 2 honey-cubes. He'd attend Band 1 on days 1, 2, and pay one honey-cube on each of those days. On days 3 and onwards, none of Lyric's favourite bands perform, so Lyric attends no concerts and spends zero honey-cubes. So his output plot would look like this:



To return this plot, Lyric will return a sequence $(t_0, h_0), (t_1, h_1), \ldots$, with $t_i \leq t_{i+1}$, which we interpret as meaning "starting on day t_i and ending on day $t_{i+1} - 1$, Lyric spends h_i honey-cubes per day. In the example above, the return value would be $(t_0 = 0, h_0 = 2), (t_1 = 1, h_1 = 1), (t_2 = 3, h_2 = 0)$.

Notes:

- In the example above, it would also be correct to return $(t_0 = 0, h_0 = 1), (t_1 = 0, h_1 = 2), (t_2 = 1, h_2 = 1), (t_3 = 3, h_3 = 0)$; that is, adding extraneous intervals of length 0 is still correct.
- In the example above, it would also be correct to return $(t_0 = 0, h_0 = 2), (t_1 = 1, h_1 = 1), (t_2 = 2, h_2 = 1), (t_3 = 3, h_3 = 0)$; that is, breaking an interval into two smaller intervals is still correct.
- Concerts will always cost a positive (non-zero) number of honey-cubes.

In this problem you'll design an algorithm for Lyric. Your algorithm should take as input a list of n schedules (a_i, b_i, h_i), one for each musical band $i \in \{0, \ldots, n-1\}$, and return a list `honeyPlot` of (t_i, h_i) pairs as described in the example above. You can assume there are $O(n)$ musical bands in addition to $O(n)$ days.

3.1 Naive algorithm (4 pt.)

Describe a simple $O(n^2)$-time algorithm for Lyric.

[We are expecting: Pseudocode, and a short English description explaining the main idea of the algorithm. No justification of the correctness or running time is required.**]**

3.2 Divide and conquer (16 pt.)

Design a divide-and-conquer algorithm that takes time $O(n \log(n))$.

[We are expecting: Pseudocode, and a short English description explaining the main idea of the algorithm. We are also expecting an informal justification of correctness and of the running time.**]**

4 Sorting with low adaptivity

In practice, when the steps of a given algorithm don't depend on one another (i.e. the inputs to each step are pre-determined and do not depend on the outputs of other steps of the algorithm), we can often execute the steps simultaneously using multiple machines (or multiple cores/processors of the same machine) and thereby gain speed. This approach of using multiple machines to execute different independent parts (steps) of the algorithm is called *parallelization*. Outside this question we won't spend much time discussing about it in this class, but we will still give an example to motivate the following question.

Multiplying a vector by a scalar is an algorithm where the steps don't depend on each other, hence it is *parallelizable*. More precisely, each multiplication of the given scalar with an element of the given vector is naturally independent of the other elements of the vector. So, different computers (or processors) can be used to perform these multiplications in parallel, without talking to one another.

In contrast, consider a contrived algorithm that computes $f(f(x))$ for a given input x and an arbitrary function f, by computing $y = f(x)$ and then using y to compute the final output $f(y)$. This algorithm is not parallelizable; instead it is *sequential*.

In this question we will be analysing *comparison-based* sorting algorithms. A *comparison-based* sorting algorithm is one which cannot access the values of the elements, instead it can only compare pairs of elements and find out which element is bigger or smaller. Observe that mergesort and quicksort, both are comparison-based sorting algorithms as the algorithms only perform comparisons without accessing the values of the elements.

4.1 Adaptivity of MergeSort (10 pt.)

We say that a comparison-based sorting algorithm has *adaptivity* t if it proceeds in $t + 1$ stages, where the pairs to be compared in the i-th stage only depend on the outcome of comparisons in stages $1, \ldots, i-1$ (but not on other comparisons in the i-th stage). Observe that a non-adaptive algorithm has 0 adaptivity.

Let's consider an example of an adaptive algorithm with adaptivity 1. Consider the problem of sorting *exactly* three given numbers a, b, c. Let's say that the algorithm commits to comparing a, b and b, c in stage 1. For a given input, if the outputs of the comparisons indicate that $a < b$ and $b < c$, then the algorithm outputs $a < b < c$ without performing any further comparisons. However, if the comparisons indicate that $a < b$ and $c < b$, the

algorithm performs an additional comparison a, c in stage 2. Since the decision to compare the specific pair a, c (and to make no further comparisons in the earlier case) in stage 2 was determined by the comparisons made in stage 1, this algorithm has an adaptivity of 1. Based on this final comparison made in stage 2, the algorithm returns the sorted output.

What is the adaptivity of the MergeSort algorithm?

[We are expecting: Adaptivity in terms of big-Θ notation, and a proof supporting the answer**]**

4.2 Adaptivity of QuickSort (15 pt.)

What is the expected adaptivity of QuickSort?

[We are expecting: Adaptivity in terms of big-Θ notation, and a proof supporting the answer**]**

Hint: There are many ways of solving this problem. One that seems useful is to consider "comparable pairs" i.e. pairs of elements that might still be compared by the algorithm, and analyze how the expected total number of comparable pairs changes over iterations of the algorithm.

5 Flipping frogs

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

The n frogs 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 frogs standing in positions $i, i+1, \ldots, j-1$. For example, if the frogs started out like this:

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

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

In this problem, you will design and analyze an algorithm to sort the n frogs by their labels, which uses `Flip`, the only move the frogs know. However, you don't know the original order of the frogs 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 frog's initial positions (for example, if the frogs 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 frogs to implement the `Flip(i,j)` dance. The algorithm will wait until the frogs 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 frogs spend dancing.

5.1 Partitioning the frogs (15 pt.)

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

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

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

Give an algorithm that instructs the frogs to implement `Partition` that runs in time $O(n \log n)$.

Hint: Try divide-and-conquer. After one call of `Flip(i, k)` *(think of a value for k that works well)*, can you identify two frogs - one for each half ($0, 1, \ldots, k-1$ and $k+1, \ldots, n-1$) - such that recursively partitioning the two halves about the identified frogs, gives a favourable arrangement of frogs. And executing Flip once more completes the dance. Also, be careful with off-by-one errors in your indexing.

[We are expecting: Pseudocode **AND** a clear English expanation of what your algorithm is doing. In addition, we expect an informal justification of the running time. You may appeal to the Master Theorem if it is relevant.**]**

5.2 Sorting the frogs (5 pt.)

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

```
def Sort(A):
    //A is an array of length n, with the positions of the n frogs
    //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 frogs to partition themselves around the i'th frog:
Partition(i)

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

That is, this algorithm first partitions the frogs around the median, which puts the smaller-labeled frogs on the left and the larger-labeled frogs on the right. Then it recursively sorts the smaller frogs and the larger frogs, 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.**]**

5.3 Runtime (10 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.**]**