Data Structure	Lab 6
Spring 2021	Tuesday, May 18

Lab Goals:

- Project Screens – Register and Login
- Pre-lecture Exercises Complete Understanding (Next Week)
- Python notebooks Complete and Thorough Understanding (Next Week)
- Homework Questions

Project Rubrics:

	Implementation
5	Properly formatted and visually pleasant UI/Template integrated + Rubric IV requirements.
4	Integrated database with Registration Screen + Rubric III requirements.
3	Integrated database with Login Screen + Rubric II requirements**.**
2	Registration Screen with user interface + Rubric I requirements
1	Login Screen with user interface created
0	No screen created

Lab Rubrics:

	Implementation
5	Completed Homework Question 3 (a, b) + Rubric IV requirements.
4	Completed Homework Question 2 (a, b) + Rubric III requirements.
3	Completed Homework Question 1 (a, b, c, d) + Rubric II requirements**.**
2	Solved and Understood Python Notebook + Rubric I requirements
1	Solved pre-lecture exercises.
0	Lab missed or solved none of the problems

Project Screens:

Students must create their Registration and Login screens of their project and get themselves evaluated in next week.

Pre-lecture Exercises

Suppose you are sorting in DSA (say n students) based on what month they were born in. So, if Plucky and Lucky were both born in January, Siggi was born in April, and Ollie was born in May, then it would be fine to return either {Plucky, Lucky, Siggi, Ollie} or {Lucky, Plucky, Siggi, Ollie }. In the above example, your input would be a list of the form [(Plucky, 1), (Lucky, 1) , (Siggi, 4), (Ollie, 5)] (Here the 1 is for January, which is the first month, and so on).

1. How would you do this using MergeSort? How long would it take (as a function of n)?

2. Can you think of a way to do it faster? [Hint, there is a way to do it faster.]

Python Notebooks:
Solve and thoroughly understand Python notebook provided in this lab.

Homework Question:

1. In this exercise, we will 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 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.
   We will revisit the Majority Element problem to get more insight on randomized algorithms.

Algorithm	**Monte Carlo or Las Vegas?**	**Expected Running Time**	**Worst-Case Running Time**	**Probability of returning a majority element**
Algorithm 1				
Algorithm 2				
Algorithm 3				

[**We are expecting:** Your filled in-table, and a short justification for each entry of the table. You may use asymptotic notation for the running times, for the probability of returning a majority element, give the tightest bound that you can give the information

provided. Fill in the table below and justify your answers.]

Algorithm 1: findMajorityElement1

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

Algorithm 2: findMajorityElement2

```
Input: A population P of n elements
for 100 iterations do
    Choose a random p ∈ P;
    if isMajority(P, p) then
        return p;
return P[0];
```

Algorithm 3: findMajorityElement3

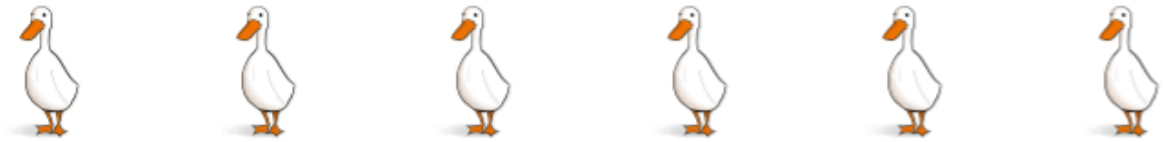
```
Input: A population P of n elements
Put the elements in P in a random order.;
/* Assume it takes time Θ(n) to put the n elements in a random order
   */
for p ∈ P do
    if isMajority(P, p) then
        return p;
```

Algorithm 4: isMajority

```
Input: A population P of n elements and a element p ∈ P
Output: True if p is a member of a majority species
count ← 0;
for q ∈ P do
    if p = q then
        count ++;
if count > n/2 then
    return True;
else
    return False;
```

2. [**Counting Inversions]:** Given an array A of n elements, we call (i, j) an inversion if $0 \leq I < j < n$ and A[i] > A[j].
   (a) Describe an $O(n^2)$ algorithm to count the number of inversions in a given array.

[**We are expecting**: Actual code, and a short English description explaining the main idea of the algorithm. No justification of the correctness or running time is required.]
(b) Describe an O(n*log(n)) algorithm to count the number of inversions in a given array. [Hint: Think about how you can modify MergeSort to solve this problem.]
[**We are expecting**: Actual code, 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.]
3. [**Unified Duck Problem]** Suppose that the n ducks are standing in a line. Each duck has a favorite activity: **honking, eating,** or simply doing nothing (**idling**). And since they are ducks, they do whatever they like to do all day long. You'd like to sort the ducks so that all the honking ducks are on the left, the eating ducks are on the right, and the idling ducks are in the middle*. You can only do two types of operations on the ducks:



*Having honking and eating ducks close would be disastrous. Don't ask why.
** see, e.g., https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two
***You don't need to use all seven storage spots, but you can if you want to. Can you do it with only two?

Operation	Result
`ask(j)`	Ask the duck in position *j* about its favorite activity
`swap(i,j)`	Swap the duck in position *j* with the duck in position *i*

You want to sort the ducks as soon as possible, but each of the above operations takes a constant time to execute. Also, you didn't bring a piece of paper or a pencil, so you can't write anything down and have to rely on your memory. Like many humans, you can remember up to seven integers** between 0 and n − 1 at a time.
(a) Design an algorithm to sort the ducks which takes O(n) seconds and requires you to remember no more than seven integers*** between 0 and n − 1 at a time.
[**We are expecting:** Pseudocode AND a short English description of your algorithm.]
(b) Justify why your algorithm is correct, why it takes only O(n) seconds, and why it requires you to remember no more than seven integers at a time.
[**We are expecting:** Informal justifications of the correctness, runtime, and memory usage of your algorithm that are both clear and convincing to the grader. If it's easier for you to be clear, you can give a formal proof of correctness, but you do not have to. It is okay to appeal to the correctness of an algorithm that we have seen in class, as long as you clearly explain the relationship between the two algorithms.]