CS 161	**Exam 2**
Winter 2021	**Length:** 36 hours (designed for 1.5 hours)

This exam is open-book and closed-collaboration. You have at most 36 hours to complete the exam and submit your answers on Gradescope. (If you start the exam late, you might have less than 36 hours.) You may consult inanimate materials or resources, including course materials, but inputting any exam questions or solutions into **any software, apps, or websites (Google, Stack Overflow, etc.) is in direct violation of the honor code**. You may review the Stanford Honor Code online at the following link.

`https://communitystandards.stanford.edu/policies-and-guidance/honor-code`

You may also review the detailed rules and instructions on the Exam portion of the CS 161 webpage, linked below.

`https://stanford-cs161.github.io/winter2021/exams/`

Good luck!

Section 1. A Randomized Algorithm (14 points)

Lucky wants to study for an exam using a deck of m flashcards. Lucky is not good at learning many concepts at once, so he only wants to learn n of these flashcards, one *new concept* each day. He will be happy with any subset of n; after all, the exam will be open-book and he can spend more time and go through the other $m - n$ flashcards during exam if need be. For this problem, assume that $n \leq m$.

Lucky's algorithm for studying is as follows: Every morning he wakes up and randomly shuffles his flashcards. Then he goes over the flashcards one-by-one in the shuffled order. For every card he has studied before, he spends one hour reviewing it again. He stops when he sees a new flashcard, spends one hour studying it, and then he calls it a day. The next morning, he repeats this procedure until he learns a new flashcard, and so on until n days pass.

For this problem, assume that random shuffling of cards takes negligible time. We want to study the runtime T which we define to be the total number of hours that Lucky spends studying.

Example. Below is an example scenario with $m = 4$ and $n = 3$. Assume that flashcards are numbered 1 through m.

- On the first day, Lucky goes over the flashcards in the (randomly chosen) order: $[2, 4, 1, 3]$. He stops after the first flashcard 2, and calls it a day.

- On the second day, Lucky goes over the flashcards in the (randomly chosen) order: $[3, 1, 2, 4]$. He stops after the first flashcard 3, and calls it a day.
- On the third day, Lucky goes over the flashcards in the (randomly chosen) order: $[3, 2, 4, 1]$. He reviews flashcards 3 and 2 (he has seen them before), and then studies flashcard 4. He stops and calls it a day.

In this example, T would be $1 + 1 + 3 = 5$. In general, T depends on the random shuffles each day; in other words, T is a random variable.

(i) **(4 pt.)** What is the *worst case runtime* of this algorithm? In other words, what is the worst case value of T for the worst possible sequence of shuffles? Express your answer in big-O notation in terms of n, AND briefly justify your answer.

(ii) **(5 pt.)** Assume that each day the shuffles are chosen independently and uniformly at random from all permutations. On the i-th day, (where $i \in \{1, \ldots, n\}$) what is the probability that Lucky finds the first flashcard to be new? How about the probability that the first flashcard is old but the second flashcard is new? Express these two probabilities as a function of n, m, i. No need to justify your answer.

(iii) **(5 pt.)** Think of the *expected runtime* of this algorithm, that is $\mathbb{E}[T]$, when the shuffles each day are uniformly and independently chosen random shuffles. If we keep n constant and increase the value of m, does this expected runtime **increase**, **decrease**, or **stay the same**? Circle your answer AND briefly justify your answer.

Section 2. Hashing (20 points)

Lucky studied for the exam using his flashcards, but he got unlucky and did not end up studying hashing much! In particular, he does not remember what our construction of small universal hash families looked like.

Lucky remembers that a hash family H is a collection of functions h, each mapping a large universe U to a set of n buckets. The hash family H is **universal** if and only if for any two distinct elements $x, y \in U$ with $x \neq y$, the collision probability of x and y is at most $1/n$:

$$\mathbb{P}[h(x) = h(y)] \leq \frac{1}{n}.$$

Here the probability is taken over the *random* choice of the function h. Note that h is a *uniformly random* element of H.

(i) **(10 pt.)** Lucky's memory of the construction of small universal hash families from class is unfortunately **incorrect**. Let us assume that U is the collection $\{0, \ldots, p-1\}$ of all numbers between 0 and $p-1$ for some prime number p. For $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ define the function

$$h'_{a,b}(x) = ax + b \pmod{n}.$$

Note that in this definition we are only taking the answer $\bmod n$. Lucky did not remember to put a $\bmod p$ before it. Help Lucky realize that the family

$$H' := \{h'_{a,b} \mid a \in \{1, \ldots, p-1\}, b \in \{0, \ldots, p-1\}\}$$

is not always universal. Provide a specific pair of values for n and p where this hash family is not universal, and briefly justify why it is not universal. [Hint: If you get stuck, plug in small values of n and p and compute the collision probabilities by hand.]

(ii) **(10 pt.)** Lucky thinks he has found a new construction that might some day lead to the construction of new universal hash families, by **composing** hash functions.

Lucky wants to construct a hash function h mapping the universe $\{0, \ldots, p-1\}$ to the buckets $\{0, \ldots, n-1\}$; assume that $p > n+1$. He suggests doing this by composing two hash functions h_1 and h_2. He first picks an intermediate number m such that $n < m < p$. Then he selects a **uniformly random** function h_1 mapping $\{0, \ldots, p-1\}$ to $\{0, \ldots, m-1\}$ and independently selects a **uniformly random** function h_2 mapping $\{0, \ldots, m-1\}$ to $\{0, \ldots, n-1\}$. Note that this means the values $h_1(0), h_1(1), \ldots, h_1(p-1)$ are independently and uniformly at random chosen from $\{0, \ldots, m-1\}$. Similarly the values $h_2(0), h_2(1), \ldots, h_2(m-1)$ are independently and uniformly at random chosen from $\{0, \ldots, n-1\}$.

Lucky proposes that the function $h(x) := h_2(h_1(x))$ has small collision probability. Prove him wrong. Provide a proof that for any pair $x, y \in \{0, \ldots, p-1\}$ with $x \neq y$, we have

$$\mathbb{P}[h_2(h_1(x)) = h_2(h_1(y))] > 1/n.$$

Section 3. Can It be Done? (32 points)

For each of the following tasks, **either** explain briefly and clearly how you would accomplish it, **or** explain why it cannot be done in the worst case.

- To explain how to **accomplish** a task when an algorithm is expected, briefly describe or write pseudocode of an algorithm. You do not need to prove correctness or runtime of this algorithm.
- To explain why a task **cannot be done** in the worst case, write a brief and clear justification or proof.

Below you can find two example tasks, and sample answers we expect for each, to have an idea of the level of detail we are expecting.

Example Task. Find the maximum of an unsorted array of length n in time $O(n \log n)$.

Sample Answer

Use MergeSort to sort the array, and then return the last element of the sorted array.

Example Task. Find the maximum element in an unsorted array of length n in time $O(1)$.

Sample Answer

This cannot be done. Since any of the n entries of the array could be the maximum, we need to at least examine every element in the array, which takes time $\Omega(n)$.

Throughout this section, you may cite any result or algorithm we have seen in class, but don't make any assumptions that are not explicitly stated. Unless stated otherwise, all running times refer to deterministic worst-case running time.

(i) **(8 pt.)** Build a binary search tree rooted at the node with value 0 using **all** of the elements in $\{-1, 0, 1, 2, 3, 4, 5\}$. (Either draw the tree or state why it cannot be done.)

(ii) **(8 pt.)** Design an algorithm that finds the minimum element in any (arbitrary) binary search tree with n elements in time $O(\log n)$.

(iii) **(8 pt.)** Design a deterministic data structure that can store n arbitrary, comparable elements and supports the following operations in $O(\log n)$ time: Insert, Delete, Search, and FindMin. The last operation should return the minimum element amongst those currently in the collection.

(iv) **(8 pt.)** Design an $O(n)$ time algorithm that receives an array A of n distinct arbitrary comparable elements and two numbers $i, j \in \{1, \ldots, n\}$ with $i < j$, and outputs in sorted order all elements of A that are bigger than the i-th smallest and smaller than the j-th smallest element of A.

Section 4. Matrix Sorting (24 pt.)

Suppose you are given n distinct numbers arranged in a matrix A (2-dimensional array) of size $\sqrt{n} \times \sqrt{n}$. For this problem, assume that $\sqrt{n}$ is an integer.

We are guaranteed that each row and each column of A is in sorted (increasing order):

- For $i \in \{0, \ldots, \sqrt{n}-1\}$ and $j \in \{0, \ldots, \sqrt{n}-2\}$, we have $A[i,j] < A[i,j+1]$.
- For $i \in \{0, \ldots, \sqrt{n}-2\}$ and $j \in \{0, \ldots, \sqrt{n}-1\}$, we have $A[i,j] < A[i+1,j]$.

Lucky wants to design an algorithm that produces a (1-dimensional) array B which has exactly the same elements as A, but is in fully-sorted (increasing) order.

Examples. You can see example inputs and expected outputs below:

- For $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, the expected output is $B = [1,2,3,4,5,6,7,8,9]$.
- For $A = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 4 & 7 \\ 6 & 8 & 9 \end{bmatrix}$, the expected output is again $B = [1,2,3,4,5,6,7,8,9]$.
- For $A = \begin{bmatrix} 3 & 8 \\ 7 & 9 \end{bmatrix}$, the expected output is $B = [3,7,8,9]$.

Note that the following two are **invalid** inputs and will never be given to the algorithm:

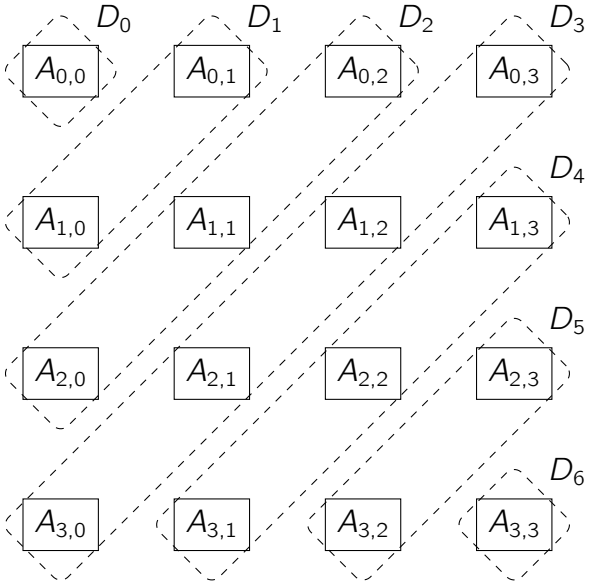
$$\begin{bmatrix} 3 & 4 \\ 1 & 5 \end{bmatrix} \text{ and } \begin{bmatrix} 3 & 1 \\ 4 & 5 \end{bmatrix}.$$

Lucky thinks that because each row and each column of the input A is already sorted, we do not need to do much extra work to sort all of the numbers into the array B. Plucky is skeptical and wants to formally prove that it takes at least $\Omega(n \log n)$ comparisons for any comparison-based sorting algorithm to produce the output B, even if A is guaranteed to have sorted rows and columns.

(i) **(2 pt.)** Briefly describe or write psuedocode for an $O(n \log n)$ time algorithm that receives a matrix A and outputs the correct 1-dimensional array B. You do not need to prove the correctness or runtime of your algorithm.

(ii) **(6 pt.)** Plucky is thinking of designing challenging inputs for this problem, but she finds the condition that rows and columns are sorted a bit annoying. Help Plucky, by proving that the following algorithm always produces **valid** matrices A. A valid matrix A is one with sorted (increasing) rows and columns.

Suppose we want to fill the matrix A with numbers $\{1, \ldots, n\}$. We do this diagonal-by-diagonal. Let the southwest-northeast facing diagonals of A be $D_0, D_1, D_2, \ldots, D_{2\sqrt{n}-2}$. Formally let D_k be all the entries indexed (i, j) with $i + j = k$. See below for examples of these diagonals for $n = 16$:



We go over the diagonals $D_0, D_1, \ldots, D_{2\sqrt{n}-2}$ one-by-one in order. For each diagonal D_k, we take the first $|D_k|$ numbers that we haven't placed in A yet, and put them in an **arbitrary** order on the diagonal D_k. You can find the pseudocode for this algorithm below.

`PopulateMatrix`(n) :
Initialize A as an empty $\sqrt{n} \times \sqrt{n}$ matrix.
$x \leftarrow 1$
for $k = 0, \ldots, 2\sqrt{n} - 2$ **do**
  Place the elements $\{x, x + 1, \ldots, x + |D_k| - 1\}$ in an **arbitrary order** on D_k.
  $x \leftarrow x + |D_k|$
return A

Prove that regardless of how the order on each diagonal is chosen, the output of this algorithm A is a valid matrix. In other words, provide a brief justification for why each row and each column of A is sorted in increasing order.

(Space for Section 4 (ii))

(iii) **(8 pt.)** Note that in each iteration of the for loop in `PopulateMatrix`, we have $|D_k|!$ possible choices for placing the elements on the diagonal. Therefore this algorithm can produce $|D_0|! \times |D_1|! \times \cdots \times |D_{2\sqrt{n}-1}|!$ different matrices A, all filled with numbers $\{1, \ldots, n\}$. Prove that this number is at least $n^{C \cdot n}$ for some constant $C > 0$. We expect a rigorous proof.

(iv) **(8 pt.)** Suppose that `MatrixSort`(A) is a deterministic comparison-based sorting algorithm that receives a valid input A, makes some number of comparisons between elements of A and produces the correct output B. Prove that the number of comparisons that `MatrixSort` needs to make in the worst case is $\Omega(n \log n)$. We expect a rigorous proof.

[END OF EXAM 2]