This assignment is **due on April 14** and should be submitted on Gradescope. All submitted work must be *done individually* without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies.

As a first step go to the last page and read the section: Advice on how to do the assignment.

Problem 1. (10 points)
We have a set of n heroes H and a set of m dragons D (both n and m are at least 1). Each hero has a persuasion value p_i and each dragon has a determination d_j. All p_i and d_j are positive integers. To avoid confusing arguments, each hero is limited to persuading only one dragon, but multiple heroes can work together to persuade the same dragon (if they do so, their combined persuasion is the sum of their individual persuasion values). A dragon is persuaded if the total persuasion value of the heroes persuading them is at least as large as the dragon's determination. We need to find out if the heroes can persuade all dragons to not eat them and play a game of "Houses & Humans" instead.

Example:
When the heroes have persuasion $H = \{4, 7, 1, 2, 2\}$ and the dragons have determination $D = \{4, 1, 10\}$, we can combine the heroes with persuasion 4 and 7 to overcome the dragon with determination 10 (as $4 + 7 \geq 10$), the two 2s can work together to persuade 4 (as $2 + 2 \geq 4$), and 1 can persuade 1. Hence, in this case we should return true.

When the heroes have persuasion $\{1, 1, 1\}$ and the dragons have determination $\{1, 2, 3, 4\}$, there is no way to persuade the dragons and thus the heroes were never heard from again. In this case we should return false.

Your friend says that they've got two algorithms that can solve this problem easily:

WRONGALGORITHM sorts the heroes by their persuasion in non-increasing order and the dragons by their determination also in non-increasing order. It then repeatedly assigns the unassigned hero with highest persuasion to the unpersuaded dragon with highest determination. If at the end all dragons are persuaded (i.e., we reached the last dragon and persuaded it), it returns true, and false otherwise.

```
1:  function wrongAlgorithm(H, D)
2:      Sort H and D in non-increasing order
3:      i, j ← 0, 0
4:      while i < n and j < m do
5:          if p_i ≥ d_j then
6:              d_j ← d_j − p_i
7:              i, j ← i + 1, j + 1          {Proceed to next hero and dragon}
8:          else
9:              d_j ← d_j − p_i
10:             i ← i + 1                    {Proceed to next hero, but same dragon}
11:     return j = m and d_{m−1} ≤ 0
```

HEAPALGORITHM creates two max-heaps $\mathcal{H}_H$ and $\mathcal{H}_D$ and inserts the heroes into $\mathcal{H}_H$ and the dragons into $\mathcal{H}_D$. While $\mathcal{H}_H$ is not empty, it removes the hero and dragon with the highest persuasion p and determination d and inserts $d - p$ into $\mathcal{H}_D$ (reflecting that the hero has partially convinced the dragon). Once it has processed all heroes, it checks whether the highest determination left in $\mathcal{H}_D$ is at most 0 (i.e., whether the heroes together have persuaded all dragons).

```
1: function HEAPALGORITHM(H, D)
2:     H_H, H_D ← new empty heap, new empty heap
3:     Insert H into H_H and D into H_D
4:     while H_H is not empty do
5:         p ← H_H.REMOVEMAX()
6:         d ← H_D.REMOVEMAX()
7:         Insert d − p into H_D
8:     return H_D.MAX() ≤ 0
```

a) Convince your friend that WRONGALGORITHM doesn't always return the correct answer by giving a counterexample, i.e., an instance where the algorithm should return true, but returns false (or vice versa). Remember that describing the allocation of heroes to dragons that the algorithm computes and explaining why the output is incorrect are also part of your counterexample.

b) Argue whether HEAPALGORITHM always returns the correct answer by either arguing its correctness (if you think it's correct) or by providing a counterexample (if you think it's incorrect).

Problem 2. (25 points)
We're attending a trading card game expo and we're looking to complete the collection of our favourite game. To make collecting the cards easier, the distributor decided to number all cards. Every seller sells cards in a specific range, say $[k_1, k_2]$. To make collecting more challenging, we've come up with the rule that when we visit a seller we want to get the card with the smallest integer number that we don't already have. As we want to do this efficiently, we're going to design a data structure that supports the basic operations we need: INSERT(i) which inserts the card numbered i that we don't own already, DELETE(i) which removes the card numbered i (if we own it) as we might want to trade cards, and SMALLESTMISSINGINRANGE(k_1, k_2) which returns the smallest integer in the range $[k_1, k_2]$ that we don't already own.

Example execution:

INSERT(23)		[23]
INSERT(4)		[23, 4]
SMALLESTMISSINGINRANGE(4, 15)	returns 5	[23, 4]
INSERT(5)		[23, 4, 5]
SMALLESTMISSINGINRANGE(4, 15)	returns 6	[23, 4, 5]
SMALLESTMISSINGINRANGE(2, 15)	returns 2	[23, 4, 5]
REMOVE(4)		[23, 5]
SMALLESTMISSINGINRANGE(4, 15)	returns 4	[23, 5]

Your task is to design a data structure that supports the above operations, more formally defined as follows:

- INSERT(i): Inserts integer i into the data structure. You can assume the integers we insert are all distinct, i.e., i is not yet present in the data structure upon insertion.
- DELETE(i): Deletes i from the data structure.
- SMALLESTMISSINGINRANGE(k_1, k_2): Returns the smallest integer i such that $k_1 \leq i \leq k_2$ and i isn't present in the data structure. The function returns null if all integers in the specified range are present in the data structure.

For full marks, each operation should run in $O(\log n)$ time and your data structure should use $O(n)$ space, where n is the number of integers stored in the data structure.

a) Design a data structure that supports the above operations in the required time and space.

b) Briefly argue the correctness of your data structure and operations.

c) Analyse the running time of your operations and the total space of the data structure.

Problem 3. (25 points)
You and your best friend just got out of the movies and are very hungry; unfortunately, it is now very late at night and all restaurants are closed. You manage to find, by chance, some vending machine still full of very... nutritious foods (bags of chips, and the occasional cereal bar). Looking into your pockets, you look what change you have, in order to try and buy something (anything!) to eat.

You have n coins of various values. So does your friend! As for the vending machine... it contains n different "food items" each with its own price. Looks like you may eat tonight... except for two things:

- the vending machine is old and somewhat broken: it only accepts at most two coins, and does not return change: you must put **exactly** the right price to get the item you ask for.
- out of fairness, you and your friend refuse to pay for the whole thing alone. So each of you has to contribute (no matter how little): to buy the food, each of you has to contribute at least one coin.

Which means that, if you want to eat tonight, you must figure out if there is an item in the vending machine whose price is exactly equal to the sum of two coins, one from you and one from your friend. And you are very hungry, so you want to figure that out **fast**.

Your task: given three arrays Y, F, and V (You, Friend, Vending machine) each containing n positive integers, decide if there exist $0 \leq i, j, k < n$ such that $Y[i] + F[j] = V[k]$. You can assume that all integers in all arrays are distinct (also

between different arrays). Since you want to eat soon, you want an algorithm for this task which solves your problem fast: running in (expected) time $O(n^2)$.

Example:
$Y = [3, 2, 1]$, $F = [4, 5, 6]$, $V = [50, 8, 13]$.
We need to return true, since $Y[1] + F[2] = V[2]$ (i.e., $2 + 6 + 8$).
For the same Y and F, but with $V = [50, 10, 9823]$, we'd return false, as there are no i, j, and k such that $Y[i] + F[j] = V[k]$.

a) As a warm-up, describe an $O(n^3)$-time algorithm in plain English.

b) Describe your efficient $O(n^2)$ (expected) time algorithm in plain English. Hint: use hashing and assume that you can compute the hash value of a key in constant time.

c) Prove the correctness of your $O(n^2)$ (expected) time algorithm.

d) Analyze the expected time complexity of your $O(n^2)$ (expected) time algorithm.

e) Analyze the worst-case time complexity of your $O(n^2)$ (expected) time algorithm.

Advice on how to do the assignment

- Assignments should be typed and submitted as pdf (no pdf containing text as images, no handwriting).
- Start by typing your student ID at the top of the first page of your submission. Do **not** type your name.
- Submit only your answers to the questions. Do **not** copy the questions.
- When asked to give a plain English description, describe your algorithm as you would to a friend over the phone, such that you completely and unambiguously describe your algorithm, including all the important (i.e., non-trivial) details. It often helps to give a very short (1-2 sentence) description of the overall idea, then to describe each step in detail. At the end you can also include pseudocode, but this is optional.
- In particular, when designing an algorithm or data structure, it might help you (and us) if you briefly describe your general idea, and after that you might want to develop and elaborate on details. If we don't see/understand your general idea, we cannot give you marks for it.
- Be careful with giving multiple or alternative answers. If you give multiple answers, then we will give you marks only for "your worst answer", as this indicates how well you understood the question.
- Some of the questions are very easy (with the help of the slides or book). You can use the material presented in the lecture or book without proving it. You do not need to write more than necessary (see comment above).
- When giving answers to questions, always prove/explain/motivate your answers.
- When giving an algorithm as an answer, the algorithm does not have to be given as (pseudo-)code.
- If you do give (pseudo-)code, then you still have to explain your code and your ideas in plain English.
- Unless otherwise stated, we always ask about worst-case analysis, worst case running times, etc.
- As done in the lecture, and as it is typical for an algorithms course, we are interested in the most efficient algorithms and data structures.
- If you use further resources (books, scientific papers, the internet,...) to formulate your answers, then add references to your sources and explain it in your own words. Only citing a source doesn't show your understanding and will thus get you very few (if any) marks. Copying from any source without reference is considered plagiarism.