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).

Solution 1.

a) One small counterexample would be $H = \{4, 3, 1\}$ and $D = \{5, 3\}$. The algorithm would assign 4 and 3 to persuade dragon 5, which leaves 1 to fail at persuading dragon 3, hence returning false. However, the heroes can persuade all dragons if 4 and 1 work together to persuade dragon 5 and hero 3 takes care of the remaining dragon by herself. Thus, the algorithm reports that the heroes get eaten, while they could have succeeded.

b) This algorithm also doesn't work, for example when we have $H = \{4, 3, 2\}$ and $D = \{5, 4\}$. The algorithm assigns hero 4 to dragon 5 and reinserts $5 - 4 = 1$. Next it assigns hero 3 to dragon 4, reinserting $4 - 3 = 1$. Hero 2 is now assigned to one of the two 1s, but one unpersuaded dragon remains, so the algorithm returns false. However, if we had sent hero 4 to persuade dragon 4 and heroes 3 and 2 to work together to persuade dragon 5, they'd all be living happily ever after.

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.

Solution 2.

a) All of the $O(\log n)$ running times should probably hint at AVL trees. However, we need to do something slightly smarter than just use the ones we saw during the lecture, since without changing them we can't efficiently distinguish between calling SMALLESTMISSINGINRANGE$(1, n)$ on an AVL tree storing the integers 1 through n (we need to return null) and an AVL tree storing the integers 1 through $n/2 - 1$ and $n/2 + 1$ through n (we need to return $n/2$). We could try to maintain the size of subtrees and see if we can use that to conclude something about whether things in the subtree are contiguous, but this is likely to get a bit messy, so we'll do something else instead.
We're going to maintain an AVL tree that stores all inserted integers, with a little twist: we store consecutive integers in a single node instead of creating separate nodes for each of them. In other words, each node will store a range $[r_1, r_2]$ of integers (ranges of different nodes are disjoint), effectively making it an AVL tree on the left boundary of all ranges it stores.
Since we're going to need to be able to search in the tree in order to insert and remove things, we'll start with describing that: when searching for i, we start from the root. When at a node storing the range $[r_1, r_2]$, we first check if i falls in this range. If so, we're done as we've found the range we're searching for. Otherwise, if $i < r_1$ we recurse on the left subtree and if $r_2 < i$ we recurse on the right subtree. If at any point we reach a leaf, we stop since i doesn't occur in the tree.
When we insert an integer i, we start by searching for $i - 1$ and $i + 1$ in the tree. If both searches end up at leaf nodes, we insert a new node storing $[i, i]$. If exactly one of the two searches returns an internal node, we simply extend the range of that node, i.e., we change $[r_1, i - 1]$ to $[r_1, i]$ or $[i + 1, r_2]$ to $[i, r_2]$. If both searches return an internal node, we need to merge the two nodes: let their respective ranges be $[r_1, i - 1]$ and $[i + 1, r_2]$. We change the range of $[r_1, i - 1]$ to $[r_1, r_2]$, and delete the node storing $[i + 1, r_2]$ from the tree using a standard AVL tree deletion.
To remove an integer i from the data structure, we start by searching for it. If this ends at a leaf, it didn't exist and thus there's nothing to do. Otherwise, let $[r_1, r_2]$ be the range stored in the internal node we end up in. We update its range to $[r_1, i - 1]$ and insert a new node $[i + 1, r_2]$ (inserting is done by searching for $i + 1$, which will end at a leaf, where we put the new node). If either range is empty, we delete (or just not insert) the corresponding node. While inserting/removing nodes, the AVL tree may of course perform its standard rebalancing operations.
Finally to find the smallest integer in the range $[k_1, k_2]$ that isn't stored in our data structure, we search our tree for the range that includes k_1. If this search ends at a leaf, k_1 isn't part of our tree and we return k_1. If this search ends at an internal node with range $[r_1, r_2]$, we compare r_2 to k_2. If $k_2 \leq r_2$, the range $[k_1, k_2]$ is included in $[r_1, r_2]$ and thus we conclude that there are no missing integers in the search range, returning null. If $k_2 > r_2$, we return $r_2 + 1$.

b) We observe that as long as the ranges of the nodes are disjoint and all ranges in a node's left subtree are on integers smaller than its left boundary and all ranges in a node's right subtree are on integers larger than its right boundary, our search is correct as it works identically to the regular search on an AVL tree.
This gives us a good start for the insertion: we'll find the nodes that we're looking for. Our case distinction guarantees that the nodes stay disjoint and the standard AVL insertion and removal (when considering the left boundary as the key) guarantees the correctness of those parts of the operation. Additionally, we observe that we merge the ranges when needed (something that we'll need for the correctness of the search for the missing integer), ensuring that there's at least one integer between different nodes in the tree.
For the removal, we know that the search returns the correct node and thus removing i from its range ensures that we store the correct integers in disjoint ranges. The case distinction guarantees that we remove empty ranges and thus our number of nodes stays bounded by n.
Finally, finding the missing integer in the range uses our search, so we end up at the node that stores k_1, if it exists (if it doesn't exist, we correctly return k_1 as the smallest missing integer in the range $[k_1, k_2]$). Since we maintain that all consecutive integers are stored in the same node during the insertion, it indeed suffices to compare k_2 and r_2, resulting in us returning $r_2 + 1$ if k_2 is large enough, since everything up to r_2 is stored in our data structure and the next range starts from at least $r_2 + 2$, as otherwise it'd have been merged with this range during one of the insertion operations.

c) We observe that our search takes at most $O(\log n)$ time, as our tree has at most n nodes, is balanced, and we perform a constant number of constant time checks per node we encounter.
An insertion performs two searches, which takes $O(\log n)$ time. It may create a single new node (which takes constant time), or remove a single node from the tree (which takes $O(\log n)$ time in an AVL tree), followed by the standard AVL rebalancing operations. Hence, this operation takes $O(\log n)$ in total.
A removal is similar to an insertion, performing an $O(\log n)$ time search and inserting and/or deleting at most one node (also $O(\log n)$ time), followed by the AVL rebalancing operations. In total this thus takes $O(\log n)$ time.
Finding the smallest integer in the range that isn't stored in the tree performs a single search ($O(\log n)$ time), followed by a constant number of comparisons ($O(1)$ time), thus taking $O(\log n)$ time in total.
The space complexity is the same as that of a standard AVL tree: $O(n)$ space, as we maintain that there are at most n nodes. The fact that we now store two integers instead of one per node doesn't affect this and in fact we could have significantly fewer nodes as we're putting consecutive integers in the same node. In the worst case, we'd still have a single integer per node, though, so this doesn't really help us.

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.

Solution 3.

a) The baseline algorithm is to iterate over all $1 \leq i, j, k \leq n$ triples (there are n^3 of them) and, for each of them, check if $Y[i] + F[j] = V[k]$; if so, return true. (Correctness is obvious (if there exists such a triple, we will find it), and time complexity follows from the fact that we do $O(1)$ work for each triple considered.)

b) Consider the following algorithm: we create a hash table T, and insert all n prices from V in T. Once this is done, we loop over all n^2 possible pairs $1 \leq i, j \leq n$, and for each of them do a lookup in T to see if T contains the value $Y[i] + F[j]$: if it does, we know there exists some k such that $Y[i] + F[j] = V[k]$ and return true. If no such pair i, j is found, then we can return false. The pseudocode is given below.

```
1:  function DecideIfCanBuyFoodFast(Y, F, V)
2:      Let n ← size(Y)
3:      Let T be a hash table implementing the Set ADT        ▷ E.g., using
    linear probing, chaining, or cuckoo hashing with a table of size 2n
4:      for all 1 ≤ k ≤ n do
5:          Insert V[k] into T (using add)
6:      for all 1 ≤ i ≤ n do
7:          for all 1 ≤ i ≤ n do
8:              Check if T contains Y[i] + F[j] (using contains)
9:              if T contains Y[i] + F[j] then
10:                 return true
11:     return false
```

c) Suppose there exist i^*, j^*, k^* such that $Y[i^*] + F[j^*] = V[k^*]$. After inserting all prices in T, the hash table contains the value $V[k^*]$; which means that, when looping over all pairs i, j, we will consider i^*, j^* and return true after performing a lookup for $Y[i^*] + F[j^*]$ in T. Conversely, if the algorithm returns true at some iteration i, j, then this means T contains the value $Y[i] + F[j]$; but since we inserted the prices listed in V (and only those values) into T, then there must be some index k such that $V[k] = Y[i] + F[j]$.

d) In total, the algorithm performs n insertions into the hash table T (n iterations of Line 5), and at most n^2 lookups (n^2 iterations of Line 8). Everything else consists of assignments, comparisons and basic mathematical operations and thus takes $O(1)$ (worst-case) time. All three options of collision handling seen in class (linear probing, chaining, and cuckoo hashing) have expected $O(1)$ insertions and lookups for the table size of $2n$, so the total *expected* time complexity is $O(n) + O(n^2) = O(n^2)$.

e) We perform n insertions and at most n^2 lookups in the hash table. Depending on the choice of collision handling, this means the following worst-case time complexity:

- Using linear probing or chaining: all operations take $O(n)$ worst-case time. This means that the worst-case time complexity is

$$\sum_{i=1}^{n} O(i) + \sum_{i=1}^{n} \sum_{j=1}^{n} O(n) = O(n^2) + O(n^3) = O(n^3).$$

- Using cuckoo hashing: insertions still takes $O(n)$ time in the worst case. However, now lookups are only $O(1)$ time even in the worst case, and so the total worst-case time complexity is

$$\sum_{i=1}^{n} O(i) + \sum_{i=1}^{n} \sum_{j=1}^{n} O(1) = O(n^2) + O(n^2) = O(n^2)\,.$$

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.