

**Problem 1.** (10 points) In this question we're going to analyse a response generated by an AI, in this case ChatGPT. It was asked to design an algorithm for the following problem that runs in  $O(n^2 \log n)$  (expected) time, argue its correctness, and analyse its running time.

**Problem Description:** 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  $O(n^2 \log n)$ .

Example:

$Y = [3, 2, 1]$ ,  $F = [4, 5, 6]$ ,  $V = [50, 8, 13]$ .

We need to return true, since  $Y[1] + F[2] = V[1]$  (i.e.,  $2 + 6 + 8$ ).

For the same  $Y$  and  $F$ , but with  $V = [50, 2123, 9123]$ , we'd return false, as there are no  $i, j$ , and  $k$  such that  $Y[i] + F[j] = V[k]$ .

**AI Output:**

Algorithm Description:

1. Hash Map Initialization: Start by creating a hash map ('map\_v') that maps each price in array 'V' to its index. This allows us to quickly lookup whether a specific price exists in 'V'.
2. Iterate Over Pairs:

- Use two nested loops to iterate over all pairs  $(i, j)$  where  $i$  is an index in array  $Y$  and  $j$  is an index in array  $F$ .
  - For each pair  $(i, j)$ , calculate the sum  $\text{sum} = Y[i] + F[j]$ .
3. Binary Search for Matching Price:
    - For each calculated sum  $\text{sum}$ , check if  $\text{sum}$  exists in  $\text{map}_v$ .
    - If  $\text{sum}$  exists, retrieve its index from  $\text{map}_v$ .
  4. Check Validity: Ensure that the retrieved index from  $\text{map}_v$  is different from  $i$  and  $j$  to satisfy the condition that you and your friend are both contributing at least one coin.
  5. Return Result:
    - If a valid index  $k$  is found that satisfies the conditions, return  $\text{true}$ .
    - If no such triplet  $(i, j, k)$  is found after checking all pairs, return  $\text{false}$ .

Correctness Proof:

- Hash Map Usage: By using a hash map for  $V$ , we achieve  $O(1)$  average-time complexity for lookups, ensuring efficiency in checking whether a sum exists.
- Binary Search: The binary search operation (using  $\text{lower\_bound}$  or similar) for finding the index in  $V$  ensures that each lookup operation is efficient within the nested loop structure.

Time Complexity Analysis:

- Hash Map Construction: Building  $\text{map}_v$  takes  $O(n)$  time.
- Nested Loops: There are  $O(n^2)$  pairs  $(i, j)$  to consider.
- Binary Search: Each binary search operation within the nested loop is  $O(\log n)$ .

Therefore, the overall time complexity is  $O(n^2 \log n)$ , dominated by the nested loops and the binary search operations.

**Your task:** To analyse the above response consider each of the following questions and briefly explain your answer to each.

- a) Does the AI response answer the question, i.e., does the algorithm solve the stated problem and does it run in the required time?
- b) Is the algorithm described in sufficient detail, i.e., is all information present or are any parts unclear or missing?
- c) Is the correctness argument complete or are there leaps or errors in the logic?
- d) Is the running time analysis correct for the provided algorithm?

**Solution 1.**

- a) The AI response correctly solves the stated problem by inserting the vending machine values into a hash table in  $O(n)$  time and iterating over each of the  $O(n^2)$  combinations of coins to check whether it contains the combined value in (hopefully)  $O(1)$  expected time per combination. Despite adding a confused mention of binary search, which does not seem to be used, and forgetting to mention that they use a hash table of  $c \cdot n$  size with constant  $c \geq 2$  (to achieve the mentioned  $O(n)$  construction time), the legible part of the algorithm works and runs in the required time. (One may argue Step 4 is incorrect as the index in  $V$  is allowed to be the same as those in  $Y$  and  $F$  because the condition only requires the indices belong to different arrays. The check would furthermore be redundant.)
- b) The algorithm description is mostly detailed enough. However, some details are unclear or missing, such as:
- Details regarding the size of the hashmap and choice of collision handling, which may influence the expected lookup time.
  - In Step 3, it mentions wanting to use a binary search, but it doesn't actually use one. Instead, it just uses the hash table to constructed.

Also, strictly speaking Step 4 isn't needed, as the problem statement mentions that the integers are positive and all integers in all arrays are distinct (also between different arrays). Hence, if you find the sum in the hash table, you already know that each person contributed a value of at least 1 and that the resulting summed value is unique to  $V$ .

- c) The AI response doesn't properly argue correctness. Both paragraphs only analyse time complexity and efficiency. What the AI should be arguing is that there exists a sum if and only if the algorithm returns True. Furthermore, since binary search isn't used by the algorithm, it shouldn't be in the argument.
- d) The running time analysis is partially correct, since the hash table construction (for a hash table of  $O(n)$  size) and the number of nested loops mentioned are correct. However, the AI failed to analyse the time spent during each loop properly. The first issue is that there is no mention of the time complexity of looking up something in the hash table (or which collision handling is used, as that may affect the running time). Furthermore, there are again references to the binary search, which isn't used in the algorithm. This analysis of binary search also implies an incorrect assumption that the hash table would be sorted and of size exactly  $n$ , as otherwise either binary search doesn't work or doesn't run in  $O(\log n)$  time. Overall, the algorithm it describes runs in  $O(n^2)$  expected time, but the analysis seems to be for a different algorithm that runs in  $O(n^2 \log n)$  worst-case time (sort  $V$  and then binary search for every possible sum). So ChatGPT seems to have been quite confused about what it was trying to analyse.

**Problem 2.** (25 points)

The company Versatile Routing Over Outstanding Metal or VROOM produces routers and prides itself in the fact that at no point during the lifetime of any of its routers the average delay is larger than a certain value  $X$ . As one of their Quality Control Engineers, you're tasked with testing a new range of their routers to ensure these too satisfy this requirement. We're testing  $k > 0$  routers, each identified using a distinct integer. Because we're using the ultimate black box testing techniques, we don't know beforehand what these integers are and  $k$  isn't known either (and  $k$  is *not* a constant). Whenever a router exceeds the maximum allowed average delay  $X$ , it's removed from the testing suite and sent to R&D for analysis, never to be seen again. You're tasked with designing a data structure that supports the testing process, including the following operations:

- `INITIALIZE()`: creates the (empty) data structure in  $O(1)$  time.
- `ADD-DATA( $ID$ ,  $delay$ )`: updates the average delay of router  $ID$  using a new data point  $delay$  in  $O(\log k)$  time.
- `CURRENT-DELAY( $ID$ )`: returns the current average delay of router  $ID$  in  $O(\log k)$  time, if it's part of the test suite, and returns null otherwise.
- `MAX-DELAY()`: returns the  $ID$  of the router with the current highest average delay in  $O(1)$  time, or null if no routers remain in the test suite.

You can assume that the delay is always a non-negative integer. You should ensure that after every operation only routers with an average delay of at most  $X$  are stored in the data structure (routers with average delay more than  $X$  are sent to R&D). Your data structure should take  $O(k)$  space, regardless of how much data is added for any specific router. Remember to

- a) Design a data structure that supports the required 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 space of your data structure.

**Solution 2.**

- a) Our data structure consist of two parts. The first is a balanced binary search tree (AVL tree, for example)  $T$  containing all IDs of routers that are still part of the test suite (i.e., haven't exceeded average delay  $X$ ) that have received at least one data packet (as we don't know anything about routers that haven't received any data). This is to keep track of the router IDs currently being tested.

We also maintain a max-heap  $H$  that contains for each router  $r$  in  $T$  a tuple  $(average, sum, count, ID)$  that stores the current average delay of router  $r$ , the total summed delay of  $r$ , the number of data packets we've received for  $r$  and the ID of  $r$ .  $H$  uses *average* as the key to ensure that the router with maximum delay is always at the root of the heap. To ensure that we can easily find the entry in  $H$  of a certain router  $r$ , we add a pointer from the node in  $T$  that stores the ID of  $r$  to the node in  $H$  that stores the information related to this router.

We now consider the different operations:

- **INITIALIZE()**: creates an empty balanced binary search tree  $T$  and an empty max-heap  $H$ .
- **ADD-DATA( $ID, delay$ )**: we search  $T$  for  $ID$ . If  $ID$  isn't in  $T$ , we insert it in  $T$  and we insert  $(delay, delay, 1, ID)$  into  $H$  (and restore the heap property) and set a pointer from the node in  $T$  to the node in  $H$ . If  $ID$  is found in  $T$ , we use the pointer to access the node in  $H$ . Let  $(average, sum, count, ID)$  be stored at the node in  $H$ . We update this node to  $((sum + delay) / (count + 1), sum + delay, count + 1, ID)$  and perform UPHEAP or DOWNHEAP operations to restore the heap property. If the node we inserted/modified is now the root of the heap (easily checked, since we know the ID of the router we updated), we need to check if  $average > X$ . If so, we remove the node from the heap using a REMOVE-MAX operation, search for its ID in  $T$  and remove it there as well.
- **CURRENT-DELAY( $ID$ )**: we search  $T$  for  $ID$ . If  $ID$  isn't in  $T$ , we return null. If  $ID$  is found in  $T$ , we use the pointer to access the node in  $H$  and return the *average* stored in it.
- **MAX-DELAY()**: returns the  $ID$  stored in the root of  $H$ , or null if  $H$  is empty.

- b) We first note that we don't have to argue that  $T$  stays a balanced binary search tree, as the standard insert operation takes care of that. Similarly, by using the standard UPHEAP and DOWNHEAP operations, the heap is maintained as well.

We maintain the invariant that every router stored in the data structure has average delay at most  $X$  and that for every node in  $H$ , *sum* stores the sum of its delays, *count* stores the number of delays recorded, and *average* stores the average delay at this time. Initializing the empty data structure to be empty satisfies these invariants.

To see that the invariant is maintained when we insert a new data packet, we look at the two cases separately. If the router didn't exist yet, inserting it and setting its average delay to the reported delay maintains that invariant, since the average over 1 integer is that integer itself. Similarly, when a new node is created, the sum is the same as the reported delay and the count is 1, since there's only a single data packet and it provided that data. If afterwards it turns out that  $\text{delay} > X$ , we remove the router again, thus ensuring that every router stored has average delay at most  $X$ . Since we only update a single router, no other router can invalidate the invariants.

If the router already existed, we find it and update its entry in  $H$ . Since we stored the sum so far and the count so far, we observe that both are updated correctly. By setting the average to the updated sum divided by the updated count, we obtain the updated average, thus satisfying that part of the invariant. After restoring the heap property, if the node is at the root, this means that it has the highest average delay. Hence, by comparing this to  $X$ , we can remove it if its average is too high, ensuring both parts of the invariant are maintained. Again, since we only modify one node, no other node can invalidate the invariant.

The invariants imply that by accessing the average delay of a specific router, we correctly return its average delay and by returning the ID of the root of the max-heap, we correctly return the ID of the router with maximum average delay.

- c) We have two parts to our data structure: the tree and the heap. Both take space linear in the number of nodes they have. Since we have a single node in each for every router we know about and there are at most  $k$  routers that haven't exceeded their allowed average delay, this implies  $O(k)$  space in total.

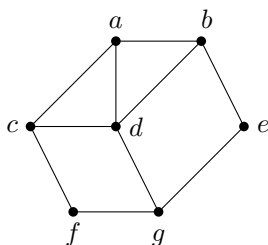
We have the following time complexities:

- **INITIALIZE()**: creating two empty data structures take constant time, as required.
- **ADD-DATA( $ID, delay$ )**: searching  $T$  takes  $O(\log k)$  time. Inserting the node in  $T$  and creating a new node for  $H$ , if it doesn't exist or following a pointer and updating the existing node take constant time. Restoring the heap property takes  $O(\log k)$  time. Finally, checking whether the root should be removed takes constant time and removing it (if needed) from  $H$  and  $T$  takes another  $O(\log k)$  time. This results in a total of  $O(\log k)$  time for this operation.
- **CURRENT-DELAY( $ID$ )**: searching  $T$  takes  $O(\log k)$  time and returning the average delay using the stored pointer takes constant time. Hence, this takes  $O(\log k)$  time in total.
- **MAX-DELAY()**: returning the max in a max-heap takes constant time, as required.

### Problem 3. (25 points)

We're volunteering in the Bushfire Prevention Club and we're tasked with designing an algorithm to predict the spread of bushfires. Our input is given as a simple undirected graph  $G = (V, E)$  ( $|V| = n$  vertices and  $|E| = m$  edges) in adjacency list representation and an array of vertices  $F \subseteq V$  that are on fire. We're told that a vertex  $v \notin F$  catches fire if at least 3 of its neighbours are already on fire. Your task is to compute all vertices of  $V$  that are on fire if fires are started at all vertices in  $F$  (and we make no attempt to prevent the fire from spreading).

Example:



If  $F = \{b, c, g\}$  then we need to report  $\{a, b, c, d, g\}$ :  $\{b, c, g\}$  are initially on fire and together set  $d$  on fire, and in turn  $\{b, c, d\}$  set  $a$  on fire as well. Note that  $e$  and  $f$  don't catch fire, since fewer than 3 of their neighbours are on fire. If  $F = \{d, e, f\}$ , we report  $\{d, e, f, g\}$  since  $\{d, e, f\}$  make  $g$  catch fire and no other vertex has 3 burning neighbours after that.

Design an algorithm that given an initial set of burning vertices  $F$ , computes all burning vertices of  $V$ . For full marks, your algorithm should run in  $O(n + m)$  time. Remember to:

- a) describe your algorithm in plain English,
- b) argue its correctness, and
- c) analyze its time complexity.

### Solution 3.

- a) Since fires can spread, we can't simply check for each vertex whether it's initially adjacent to 3 vertices in  $F$ . Checking for all previously non-burning vertices whether they now catch fire when a new burning vertex is found would be too costly, as there are  $n$  vertices and since we may find only a single new burning vertex in each iteration, this would result in  $O(n^2)$  time, if not worse. So, we'll have to do something a bit smarter.

We'll use a list *result* to keep track of all burning vertices found so far. Initially, we copy all elements of  $F$  into *result*, since all of those need to be reported regardless. Next, we give each vertex a counter. For vertices not in  $F$ , this counter is initialized at 3, to keep track of the number of adjacent burning vertices it needs until it catches fire itself. For vertices in  $F$ , we set the counter to 0, indicating that it's already on fire. The idea is that we process all burning vertices one at a time and decrement the counters of all adjacent vertices. If any of those vertices now have counter equal to 0, they catch fire.

In order to do this, we maintain a queue of (unprocessed) burning vertices, which initially contains all vertices of  $F$ . Since we have  $n$  vertices in the graph, we initialize the queue to have size  $n$ . When we take a vertex from the queue, we decrement the counter of each of its neighbours. If the counter of a neighbour  $v$  (after being decremented) equals 0,  $v$  catches fire and we add  $v$  to *result* and we add  $v$  to the queue. This process is repeated until the queue is empty, at which point the fire has stopped spreading and we return *result*.



- b) We need to argue that every burning vertex is reported and that no non-burning vertex is reported. We start with the non-burning vertices. Initially, their counters have value 3. Since every burning vertex decrements the counter of its neighbours and we know that a non-burning vertex has at most two burning neighbours, the counter of any non-burning vertex is decremented at most twice. Hence, its counter will have value at least 1 throughout the execution of the algorithm and we thus never report a non-burning vertex.

To see that all burning vertices are reported, we use a proof by contradiction. Consider any ordering in which the burning vertices catch fire (this doesn't have to match the order in which the algorithm processes vertices) and let  $v$  be the first vertex in this ordering that isn't reported by our algorithm. Since  $v$  is burning, it must have at least three burning neighbours that occur before  $v$  in the ordering (or  $v \in F$ , but since all vertices in  $F$  are reported, this isn't possible). However, since  $v$  wasn't reported by our algorithm, at least one of these neighbours wasn't reported either, as otherwise  $v$ 's counter would've reached 0 and it would've been reported. Since the neighbours of  $v$  occur before  $v$  in the ordering, this means that there is a burning vertex that isn't reported preceding  $v$  in the ordering, contradicting that  $v$  is the first one. Hence, we conclude that all burning vertices are indeed reported.

- c) Initializing *result* by copying  $F$ , initializing all counters, and initializing the queue takes  $O(n)$  time. Processing a single element from the queue takes time proportional to the number of neighbours it has (plus  $O(1)$  time to remove it from the queue). Similar to what we saw for BFS and DFS, over the whole execution of the algorithm, the neighbours sum up to at most  $2m = O(m)$ . Adding and removing at most  $n$  vertices from a queue takes  $O(n)$  time. Keeping track of *result* takes at most  $O(n)$  time (we can use either a linked list or an array of size  $n$ ). Hence, in total, this results in  $O(n + m)$  time.