Solution 1.

a

Let $M = \{2, 2, 3, 7\}$. The **wrongAlgorithm** begins by removing the pair {2, 2} from M and adding one 2 to each of M_1 and M_2. After this initial step:

$$M = \{3, 7\},$$
$$M_1 = \{2\},$$
$$M_2 = \{2\}.$$

Next, the **wrongAlgorithm** selects the largest integer x in M, which is 7 in this scenario. It then attempts to find a multiset M' from $M \setminus \{x\}$, currently $\{3\}$, such that the sum of the integers in M' equals x. The algorithm returns false because no such M' exists. However, if M_1 were {2, 2, 3} and M_2 were {7}, the total sum of all integers in M_1 would equal the total sum of all integers in M_2.

b

To prove the correctness of the permutationAlgorithm approach, we must determine whether the method guarantees that if it is possible to partition a multiset M into two subsets M1 and M2 such that the sum of integers in M1 equals the sum of integers in M2, then the algorithm will indeed find such a partition. Conversely, if no such partition exists, the algorithm will not incorrectly claim one does.

- If there exists a subset that sums to exactly half of M, then there exists at least one permutation of elements where, when considered in sequence up to a certain point, their cumulative sum will equal half of M. Thus, the algorithm will eventually encounter this permutation and return true.
- If no such subset exists, no permutation will achieve a cumulative sum of exactly half, so the algorithm will exhaust all permutations and conclude that it is not possible to split M into M1 and M2 with equal sums.

c

Incorret. Let $M = \{2, 3, -1\}$, **BalancingAlgorithm** starts by sorting M.M becomes $\{3, 2, -1\}$ now.It first add 3 to M_1 and then add 2 to M_2.In ther third iteration,the sum of integers in M_1 is greater than M_2, so that -1 is added to the M_2.After above steps:

$$M_1 = \{3\}\ M_2 = \{2, -1\}$$

It returns false. But in fact if $M_1 = \{2\}$, $M_2 = \{3, -1\}$, the total sum of all integers in M1 would equal the total sum of all integers in M2.

Solution 2.

a

```
1: function InitializeDataStructure(L_1, ..., L_k)
2:     Create empty Min-Heap H_min and Max-Heap H_max
3:     for i ← 1 to k do
4:         node_min ← reference to the first node of L_i
5:         node_max ← reference to the last node of L_i
6:         Insert node_min into H_min
7:         Insert node_max into H_max
8:     BuildHeap(H_min, true)
9:     BuildHeap(H_max, false)

10: function RemoveMin
11:     if n = 0 then
12:         return null                                  ▷ Heap is empty
13:     else
14:         minNode ← H_min.removeMin()                  ▷ Remove the node directly
15:         nextNode ← minNode.next
16:         if nextNode ≠ null then                      ▷ Check if there is a next node
17:             Insert nextNode into H_min               ▷ Directly store next node
18:         n ← n − 1
19:         return value of minNode

20: function RemoveMax
21:     if n = 0 then
22:         return null
23:     else
24:         maxNode ← H_max.removeMax()                  ▷ Remove the node directly
25:         prevNode ← maxNode.prev
26:         if prevNode ≠ null then                      ▷ Check if there is a previous node
27:             Insert prevNode into H_max               ▷ Directly store previous node
28:         n ← n − 1
29:         return value of maxNode
```

We use a min-heap and a max-heap. Assuming that all the lists are sorted in non-decreasing order, we start by inserting the first node from each list into the min-heap and adding the last node from each list to the max-heap. We duplicate the nodes without affecting the original lists.

The initial step of the structure involves setting up the heaps:

- **Min-Heap Initialization:** We begin by creating an empty min-heap, initializing it with the first node from each list.To construct the min-heap, We use the method of "building a heap in one go" explained in the lecture to ensure the property of a min-heap.

- **Max-Heap Initialization:** Similarly, we create an empty max-heap by collecting the last node from each list. To ensure the max-heap property, we will use the same method as in the lecture, but this time we will move the larger one to the position of the parent.

During insertion, nodes are duplicated rather than moved, ensuring that operations within the heaps do not modify the original lists.

- **RemoveMin Operation:** This operation extracts the minimum element from the min-heap, which is always located at the root. After removal, the next element from the list (if it exists) is inserted into the min-heap.
- **RemoveMax Operation:** This operation extracts the maximum element from the max-heap, also at the root. After removal, the previous element from the list (if it exists) is inserted into the max-heap.

b

The min-heap is initialized by inserting the first node from each list, ensuring that each element is the smallest from its respective list. We used the "build a heap in one go" method demonstrated in the lecture to construct the min-heap, ensuring its properties are maintained. During the RemoveMin operation, when the minimum element at the root is removed, the algorithm inserts the next node from the same list (if it exists) to maintain its correctness, as it's guaranteed to be greater than or equal to the current element. This process consistently maintains the min-heap's correctness and properties.

Similarly, the max-heap is initialized by inserting the last node from each list, placing the largest available element from each list into the heap. During the RemoveMax operation, when the maximum element is removed, the algorithm inserts the previous node from the same list (if it exists), which is guaranteed to be smaller than or equal to the current element. This process consistently maintains the max-heap's correctness and properties.

Even if elements exist simultaneously in both the min-heap and max-heap, the correctness of these operations remains unaffected until the total number of operations exceeds n (because `RemoveMax` always finds the maximum, and `RemoveMin` always finds the minimum). So that, we utilize n to limit the number of operational executions; when $n = 0$, it indicates that all elements have been either removed through removeMax or removeMin operations, and subsequent operations will return null, even though the original lists remain unaffected. Each node's relative position remains appropriate until it becomes the root, ensuring the heap's correctness.

Overall, the approach maintains separate and correct representations of the minimum and maximum elements at all times, validating the overall correctness and efficiency of the algorithm.

c

The operations in the dual heap structure involve initialization and dynamic updates. Here is the breakdown of the time complexities:

Heap Initialization

- **Min-Heap Initialization:** The min-heap is initialized by inserting the first node from each of the k sorted lists, which takes $O(k)$. Constructing the heap using the method mentioned in the lecture has a time complexity of $O(k)$.
- **Max-Heap Initialization:** Similarly, the max-heap is created by inserting the last node from each list, which takes $O(k)$. Constructing the heap using the method demonstrated in the lecture also has a time complexity of $O(k)$.

Heap Operations

- **RemoveMin Operation:** The time complexity for removing the smallest element from a min-heap is $O(\log(k))$, because the deletion operation moves the heap's last element to the root and then performs a DownHeap operation to rebalance the heap. Inserting the next element from the same list (if it exists) also requires $O(\log(k))$, since the insertion process involves placing the new element at the end of the heap and then performing a UpHeap operation to maintain the heap's min-heap property.Therefore, the total time complexity for the `RemoveMin` operation is $O(\log k)$.
- **RemoveMax Operation:** As with the above, the operation of removing the largest element in this context takes $O(\log(k))$, while inserting the previous element from the same list (if it exists) also takes $O(\log(k))$.Therefore, the total time complexity for the `RemoveMax` operation is $O(\log k)$.

Solution 3.

a

We use two AVL trees to store main dishes and side dishes separately, with the trees constructed based on the lexicographical order of dish names. Each tree node's key is the dish name, and the value is the corresponding dish's price.The trees are named `MainTree` for main dishes and `SideTree` for side dishes.

In addition, we create two fixed-length arrays of length 15 (with indices from 0 to 14), named `MainArray` and `SideArray`. All elements have initial value 0. Each index corresponds to a price (e.g., index 5 corresponds to 5 dollars), and the value at each index represents the number of dishes in the respective tree at that price. For example, if `MainArray[7] = 3`, it indicates there are three main dishes priced at 7 dollars.

When we perform `addNewMainDish` or `addNewSideDish`, the process involves incrementing the count of dishes at a specific price by adding 1 to the corresponding element in either the `MainArray[price]` or `SideArray[price]`. This action represents an increase in the count of main dishes or side dishes at that specific price,if it falls within the range [1,14].Then we insert a node into the corresponding AVL tree, where the key is the dish name, and the key-value is the respective price.

Similarly, when we execute `removeMainDish(name)` or `removeSideDish(name)`, we search for the dish in the corresponding AVL tree. If the dish is not found, no operations are performed on the tree, and NULL is returned. If the dish is located in the AVL tree, the count of dishes at the specified price is decremented by reducing the value in `MainArray[price]` or `SideArray[price]` by 1,if it falls within the range [1,14]. This step represents a decrease in the count of main dishes or side dishes at that specific price.

To perform `countCombinations()` , we can simply iterate through the arrays of main dish prices and side dish prices, calculate their combinations, and then check if the sum satisfies the condition. For the combinations that meet the condition, accumulate their quantities and return the total.

b

The AVL tree automatically balances itself during insertion and deletion, ensuring that the height of the tree does not exceed $\log(n)$. Due to the height limit, the efficiency of search, insertion, and deletion operations is ensured, allowing these operations to complete in logarithmic time.

The arrays `MainArray` and `SideArray` represent the number of main dishes and side dishes at specific prices. Every time a dish is added or removed, the corresponding count in these arrays is updated, ensuring that the data remains consistent for the `countCombinations` operation.

When `addNewMainDish(name, price)` or `addNewSideDish(name, price)` is called, the dish is inserted into the appropriate AVL tree and the count in `MainArray[price]` or `SideArray[price]` is incremented. This method ensures that the AVL trees and the arrays always match each other.

In the case of `removeMainDish(name)` or `removeSideDish(name)`, the corresponding dish is searched within the AVL tree. If found, it is removed from the tree and the count in `MainArray[price]` or `SideArray[price]` is decremented. This step guarantees the consistency of the data between the AVL tree and the arrays.

For the `countCombinations()` operation, since we only need to find combinations that are less than or equal to $15, we don't need to consider dishes costing more than $14 (as a cost of $15 would only pair with 0, which is impossible). This means that an array of length 15 can cover all possible scenarios (from $0 to $14) that meets the requirement. Although the prices are positive integers, I used the range 0-14 to align the price with the index. 0 does not affect the correctness of the result because there are no dishes with prices of 0. We iterates through `MainArray` and `SideArray` to calculate all possible combinations of main and side dishes, checking if their total price does not exceed 15. If a combination meets this condition, the respective counts are added to determine the total number of valid combinations.Through these steps, we can ensure that all scenarios meeting the requirements are identified.

The accuracy of the results is guaranteed by keeping the data up to date during insertion and deletion. This consistency allows `countCombinations()` to operate on accurate data, providing correct results.

c

Insert Operations (`addNewMainDish` , `addNewSideDish`)

Adding a new dish involves incrementing the count at the corresponding position in the array (MainArray or SideArray), updating the count of dishes at that specific price. This is a simple array operation with a time complexity of $O(1)$. Next, a new node is inserted into the corresponding AVL tree. The time complexity of AVL tree insertion is $O(\log n)$,given the number of nodes in this tree is less than or equal to n. Thus, the overall time complexity for adding a new dish is $O(\log n)$.

Delete Operations (`removeMainDish(name)` , `removeSideDish(name)`)

To delete a dish, first search for it in the corresponding AVL tree. The time complexity for this operation is $O(\log n)$, as the height of an AVL tree does not exceed $\log(n)$.

- If the dish is found, decrement the count at the corresponding position in the array (MainArray or SideArray) by 1. This is a constant-time operation, with a time complexity of $O(1)$, since it involves a basic array operation.
- After decrementing the count, delete the corresponding node from the AVL tree. This deletion operation has a time complexity of $O(\log n)$.
- Return the price of the dish that was removed,which takes $O(1)$.
- If the dish is not found, return `null` and take no further action.

Thus, the total time complexity for deleting a dish is primarily determined by the search and deletion operations on the AVL tree, resulting in an overall time complexity of $O(\log n)$.

Combination Operation (`countCombinations()`)

To calculate the combination of main dishes and side dishes, we iterate through the `MainArray` and `SideArray` to count the combinations. Since the arrays have a fixed length 15, iterating through them has a time complexity of $O(225)$,which is equivalent to $O(1)$.