DetectTerrorist Writeup

I call my recursive method on all the tefillin bags on the flight. When my method is called on a given set, if the set contains a single bag, that must be the terrorist's tefillin bag, so I return its index. Otherwise, if there is an even number of bags, I split the set in half and weigh each half. I recursively call my recursive method on the lighter half. If the number of bags is odd, I ignore the last bag and split the rest of the now even set in half and weigh each half. If one half is lighter than the other, I recursively call my method on that subset. If the two halves are equal, I return the last bag that I had ignored.

Proof by Induction: For any number of passengers n, where weight(charedi) for each of $n - 1$ charedis is the same, but weight(terrorist) < weight(charedi), my algorithm finds which passenger is the terrorist.

Base Case: $n = 1$: Direct proof

1. We know there is a terrorist, so if there is only one passenger, that must be the terrorist.

Inductive Case: Given that cases < n find the terrorist, n also finds the terrorist

2. I will divide this into 2 cases: where n is even and where n is not.

Lemma: The half of the input that contains the terrorist is lighter than the other half

3. The weight of the half of n bags that does not contain the terrorist's bag equals $(n / 2) \times$ weight(charedi), the number of charedi bags times the weight of each.
4. The weight of the half that contains the terrorist bag equals $(n / 2 - 1) \times$ weight(charedi) + weight(terrorist).
5. weight(terrorist) < weight(charedi)
6. We add $(n / 2 - 1) \times$ weight(charedi) to each side to get $(n / 2 - 1) \times$ weight(charedi) + weight(terrorist) < $(n / 2 - 1) \times$ weight(charedi) + weight(charedi)
7. That can be simplified to $(n / 2 - 1) \times$ weight(charedi) + weight(terrorist) < $(n / 2) \times$ weight(charedi)

Case 1: n is even: Direct Proof

8. Since my method calls itself in the lighter subset, it is called on a subset that contains a terrorist, using my Lemma.
9. Since the inductive hypothesis says that my method works for cases < n, and I am calling the method for $n / 2$, my method works for n when n is even.

Case 2: n is odd

10. I treat the subset passengers {1, 2, …, $n-1$} as its own set and effectively apply the recursive method to it.
11. If the subset contains the terrorist, it is functionally equivalent to calling the method on $n-1$, so by induction, if the subset contains the terrorist, my method works for n.
12. The subset does not contain the terrorist if comparing the weights of each half of the subset finds each half to be equal, since they both contain an equal number of equally weighted charedi bags.
13. Since the terrorist was in the set, it must be at index n, since that is the only index that did not enter the subset. Since I have found the terrorist, my method works for all n. BAM! QED

Recurrence: $T_n = \begin{cases} O(1) & if\ n = 1 \\ T_{n/2} + O(1) & if\ n > 1 \end{cases}$ (for the record, this was made in Microsoft Word)

In the base case, all I do is return the current index, an O(1) operation. In the recursive case, I weigh each subset of bags, compare the weights, and call the method on one of the subsets (or just return, if it is odd and they are equal). The latter two operations are O(1). Weighing a subset is not actually O(1), but you told us to pretend it was constant time. Since this is the same recurrence as Binary Search, it shares its closed form: O(lg$n$).

If n is odd, there is a small chance at each level that the method will return early, without reaching the bottom level. In that case, the order of growth would be O(1), functionally equivalent to reaching the base case early. Since, at each level, this most likely does not happen, the overall order of growth is not changed.

I translated my algorithm to your API by calling my method on the entire flight, with parameters 0 and `passengers.length – 1`, in the constructor. When I found the terrorist, I saved its index to a class field, which I returned when `getTerrorist()` was called.

My measured doubling ratio is about 1.9, which implies that my algorithm is linear, not logarithmic. This makes sense when you consider that weighing a subset isn't a constant operation, but an O(n) operation. This means that the recurrence is actually:

$$T_n = \begin{cases} O(1) & if\ n = 1 \\ T_{n/2} + O(n) & if\ n > 1 \end{cases}$$

This recurrence is the same that QuickSelect had when we pretended that it always chose the median as a pivot, and so shares the same closed form order of growth: O(n). For the record, I originally used `IntStream.parallel().sum()`, getting a 1.2 doubling ratio, but I couldn't figure out how to get a closed form for $T_n = T_{n/2} + O(\frac{n}{p} + \lg(n))$, so I made my algorithm worse.