825. Friends Of Appropriate Ages

Medium | Array | Two Pointers | Binary Search | Sorting

Problem Description

In this problem, we're modeling a simplified version of friend requests on a social media platform. Every user has an age, and there is an array called `ages` where `ages[i]` represents the age of the `i-th` person.

The task is to count the number of friend requests made on the platform under certain rules. Person `x` can send a friend request to person `y` only if they meet the following three conditions:

1. Person `y`'s age is **not** less than or equal to `0.5` times the age of person `x` plus `7`.
2. Person `y`'s age is **less than or equal** to the age of person `x`.
3. If person `y` is over 100 years old, then person `x` must also be over 100 (no one under 100 can send a friend request to someone over 100).

It's important to note that friend requests aren't reciprocal (if `x` requests `y`, `y` does not automatically request `x`), and no one can send a friend request to themselves. The goal is to figure out the total number of these possible friend requests.

Intuition

To solve the problem, we analyze the conditions under which one person will send a friend request to another and apply these to all possible pairs of ages. We need to consider the age limitations and the fact that there could be multiple people with the same age.

Since the problem limits ages to be between 1 and 120, we can iterate over each possible age for both `x` and `y` (the sender and the receiver of the friend request). For each age pair, we calculate the number of possible friend requests based on the number of people who have those ages.

The solution uses a `Counter` to tally the number of people at each age which allows efficient lookups. For each pair of ages `(i, j)`, if `i` can send a request to `j` according to the rules above, we increase our friend request count by `n1 * n2` where `n1` is the number of people with age `i` and `n2` is the number of people with age `j`. We need to account for the fact that people cannot friend request themselves, so if `i` equals `j`, we subtract the number of people who would have otherwise friend-requested themselves (which is `n2`).

This approach uses two nested loops to check all possible age pairs and applies the given conditions to calculate the number of friend requests.

Solution Approach

The solution provided proceeds through a process that we can break down step by step:

1. **Import the Counter Class**: The Counter class from the `collections` module is used, which provides a way to count the number of occurrences of each unique element in an iterable. In this case, it's used to keep track of how many people are of each age.
2. **Initialize the Answer**: A variable `ans` is initialized to `0`. This variable keeps a running total of all valid friend requests.
3. **Double Loop Through Each Age**: Two nested loops iterate over all possible ages from `1` to `120`. The first loop uses `i` to represent the age of the person sending the request (person `x`), and the second loop uses `j` to represent the age of the potential recipient (person `y`).
4. **Getting Count of Each Age Group**: Using the Counter created, we retrieve `n1`, the count of people with age `i` and `n2`, the count of people with age `j`.
5. **Apply the Conditions**: Within the nested loop, for each `(i, j)` pair, the code checks whether the three conditions mentioned in the problem statement are `not` met. If all three conditions are false, it means that a person with age `i` can send a friend request to the person with age `j`.
6. **Update the Answer**: If the conditions are satisfied (meaning that `i` can send a friend request to `j`), the number of friend requests is updated by adding `n1 * n2` to the total. This accounts for all possible friend requests that can be made by `n1` people of age `i` to `n2` people of age `j`.
7. **Self-request Adjustment**: Since no one can send a friend request to themselves, if `i` equals `j`, the code subtracts the number of self requests, which is `n2`. This is because when the ages are equal, each person has been counted as sending a request to themselves once, which is not allowed.
8. **Return the Answer**: After all pairs of ages have been checked, the total number of friend requests is returned as the final answer.

Here is how the conditions look like in code:

- `j <= 0.5 * i + 7`
- `j > i`
- `j > 100 && i < 100`

The solution iterates over all age combinations only once, resulting in an efficient approach with a time complexity that is constant—not dependent on the number of people but rather on the fixed number range (which is 1 to 120). This is a key insight that greatly simplifies an otherwise potentially complex problem, especially since a straightforward nested loop over the original list of ages would have a time complexity more directly tied to the number of persons.

Example Walkthrough

Suppose we have an array of ages: `ages = [14, 16, 16, 60]`.

Let's illustrate the solution approach:

1. **Import the Counter Class**: First, we import `Counter` from `collections`.
2. **Initialize the Answer**: We set `ans = 0`.
3. **Double Loop Through Each Age**: Now, we start two nested loops of ages between 1 and 120 (but in practice, we only consider ages present in our array). Here, they are: 14, 16, and 60.
4. **Getting Count of Each Age Group**: We use a `Counter` to count age occurrences in our example.
   - Counter(ages) gives `{14: 1, 16: 2, 60: 1}`.
   - So, `n1` is the count of people per age on the outer loop and `n2` is that on the inner loop.
5. **Apply the Conditions**: We check if person `x` (`i`) can send a friend request to person `y` (`j`) by negating the conditions given.
   - `j > 0.5 * i + 7`
   - `j <= i`
   - Not (`j > 100 && i < 100`), which simplifies to `i >= 100 || j <= 100` (since we can't send to someone over 100 unless `i` is also over 100).
6. **Update the Answer**: We go through combinations, considering our array and other conditions.
   - Person 14 cannot send to anyone (fails condition 1 with everyone else).
   - Persons of age 16 can send to each other but not to age 14 (fails condition 1) or 60 (fails condition 2).
     - 2 people of age 16 sending requests to 2 people of age 16 (including themselves initially) is `2 * 2`.
   - Person 60 can send to everyone 16 and over, but since there are only people of age 16 and 60, they can send to the 2 people of age 16.
7. **Self-request Adjustment**:
   - Since people can't friend request themselves, we subtract each person of age 16 from that count, which gives us `2 * 2 - 2` = 2 valid friend requests from ages 16 to 16.
   - There is no need to adjust for age 14 or 60 as 14 cannot send any requests and there is only one person aged 60.
8. **Return the Answer**:
   - We have 2 requests from persons aged 16 to 16, and each person aged 60 can send to both aged 16, so `2 + 2 = 4`.
   - The final `ans = 4`.

This is the total number of valid friend requests that happen in our small example: `4`. The process uses logical conditions and arithmetic to carefully tally the requests between age groups, accounting for self-requests and avoids unnecessary computations through the use of a count for each age bracket.

Solution Implementation

Python

```python
from collections import Counter  # Import the Counter class from the collections module

class Solution:
    def num_friend_requests(self, ages: List[int]) -> int:
        # Counter object to store the frequency of each age
        age_count = Counter(ages)

        # Initialize the answer to 0
        friend_requests = 0

        # Loop through all possible ages from 1 to 120
        for age_a in range(1, 121):
            count_a = age_count[age_a]  # Number of people with age age_a

            # Inner loop to iterate through all possible ages to find potential friends
            for age_b in range(1, 121):
                count_b = age_count[age_b]  # Number of people with age age_b

                # Check the conditions when a person A can send a friend request to B:
                # Condition 1: B should not be less than or equal to 0.5 * A + 7
                # Condition 2: B should be less than or equal to A (A can send to B of same age or younger)
                # Condition 3: If B is over 100, then A must be over 100 as well
                if not (age_b <= 0.5 * age_a + 7 or age_b > age_a or (age_b > 100 and age_a < 100)):
                    # If the conditions are met, increase the count of friend requests
                    friend_requests += count_a * count_b

                    # If both ages are the same, we have to subtract the instances where A is B
                    # because a person cannot friend request themselves
                    if age_a == age_b:
                        friend_requests -= count_b

        # Return the total friend requests that can be made
        return friend_requests
```

Java

```java
class Solution {
    public int numFriendRequests(int[] ages) {
        // Array to keep count of each age (up to 120)
        int[] ageCount = new int[121];
        // Fill the ageCount array with the frequency of each age
        for (int age : ages) {
            ageCount[age]++;
        }

        // Variable to hold the final result
        int friendRequests = 0;

        // Loop through each age for the sender
        for (int senderAge = 1; senderAge < 121; senderAge++) {
            int senderCount = ageCount[senderAge];

            // Only continue if there are people with this age
            if (senderCount > 0) {
                // Loop through each age for the receiver
                for (int receiverAge = 1; receiverAge < 121; receiverAge++) {
                    int receiverCount = ageCount[receiverAge];
                    // Check if the friend request condition is satisfied
                    if (!(receiverAge <= 0.5 * senderAge + 7 || receiverAge > senderAge || (receiverAge > 100 && senderAge < 100))) {
                        // Add the product of the counts of the respective ages
                        friendRequests += senderCount * receiverCount;
                        // Deduct the count when sender and receiver are of the same age
                        if (senderAge == receiverAge) {
                            friendRequests -= receiverCount;
                        }
                    }
                }
            }
        }
        // Return the total number of friend requests
        return friendRequests;
    }
}
```

C++

```cpp
class Solution {
public:
    int numFriendRequests(vector<int>& ages) {
        // Create a counter array to store the number of people at each age.
        vector<int> ageCounter(121, 0); // Initialized with 121 elements all set to 0

        // Count the number of instances of each age
        for (int age : ages) {
            ageCounter[age]++;
        }

        int totalFriendRequests = 0; // Initialize the total number of friend requests to 0

        // Loop through the ages from 1 to 120 (inclusive)
        for (int ageA = 1; ageA <= 120; ++ageA) {
            int countAgeA = ageCounter[ageA]; // Number of people with age A

            // Loop through all possible ages for potential friends
            for (int ageB = 1; ageB <= 120; ++ageB) {
                int countAgeB = ageCounter[ageB]; // Number of people with age B

                // Check if a friend request can be sent according to the problem's conditions
                if (!(ageB <= 0.5 * ageA + 7 ||   // Age B should not be less than or equal to 0.5 * Age A + 7
                      ageB > ageA ||               // Age B should be less than or equal to Age A
                      (ageB > 100 && ageA < 100)   // If Age B is > 100, then Age A should also be > 100
                     )) {
                    totalFriendRequests += countAgeA * countAgeB;  // Add the count product to total requests

                    // If ages are the same, subtract the self request count
                    if (ageA == ageB) {
                        totalFriendRequests -= countAgeB; // Subtract the diagonal
                    }
                }
            }
        }

        // Return the total number of friend requests
        return totalFriendRequests;
    }
};
```

TypeScript

```typescript
// Array to store the number of people at each age, initialized with 121 elements all set to 0
const ageCounter: number[] = new Array(121).fill(0);

// Function that calculates the number of friend requests
function numFriendRequests(ages: number[]): number {
    // Count the number of instances of each age in the input array
    ages.forEach((age) => {
        ageCounter[age]++;
    });

    let totalFriendRequests: number = 0; // Initialize the total number of friend requests to 0

    // Iterate through the ages from 1 to 120 (inclusive) to calculate the number of friend requests
    for (let ageA = 1; ageA <= 120; ++ageA) {
        const countAgeA = ageCounter[ageA]; // Number of people with age A

        // Loop through all possible ages to check for potential friend matches
        for (let ageB = 1; ageB <= 120; ++ageB) {
            const countAgeB = ageCounter[ageB]; // Number of people with age B

            // Check if a friend request can be sent according to the given conditions
            if (!(ageB <= 0.5 * ageA + 7 ||  // Age B should not be less than or equal to 0.5 times Age A + 7
                  ageB > ageA ||             // Age B must be less than or equal to Age A
                  (ageB > 100 && ageA < 100) // If Age B is over 100, then Age A must also be over 100
                 )) {
                totalFriendRequests += countAgeA * countAgeB;  // Accumulate the product of the counts into total requests

                // If Age A is the same as Age B, subtract the count for self requests
                if (ageA === ageB) {
                    totalFriendRequests -= countAgeB; // Adjust for self-request scenarios by subtracting the count
                }
            }
        }
    }

    // Return the total computed friend requests
    return totalFriendRequests;
}
```


```python
from collections import Counter  # Import the Counter class from the collections module

class Solution:
    def num_friend_requests(self, ages: List[int]) -> int:
        # Counter object to store the frequency of each age
        age_count = Counter(ages)

        # Initialize the answer to 0
        friend_requests = 0

        # Loop through all possible ages from 1 to 120
        for age_a in range(1, 121):
            count_a = age_count[age_a]  # Number of people with age age_a

            # Inner loop to iterate through all possible ages to find potential friends
            for age_b in range(1, 121):
                count_b = age_count[age_b]  # Number of people with age age_b

                # Check the conditions when a person A can send a friend request to B:
                # Condition 1: B should not be less than or equal to 0.5 * A + 7
                # Condition 2: B should be less than or equal to A (A can send to B of same age or younger)
                # Condition 3: If B is over 100, then A must be over 100 as well
                if not (age_b <= 0.5 * age_a + 7 or age_b > age_a or (age_b > 100 and age_a < 100)):
                    # If the conditions are met, increase the count of friend requests
                    friend_requests += count_a * count_b

                    # If both ages are the same, we have to subtract the instances where A is B
                    # because a person cannot friend request themselves
                    if age_a == age_b:
                        friend_requests -= count_b

        # Return the total friend requests that can be made
        return friend_requests
```


Time and Space Complexity

Time Complexity

The given code consists of two nested loops each ranging from `1` to `121`, resulting in a fixed number of iterations for the loops. Since the loops do not depend on the size of the input (`ages` list), they have a constant runtime. However, the overall time complexity does still depend on the `Counter` operation performed on the `ages` list earlier, which iterates over the entire list once.

The `Counter` operation has a time complexity of `O(N)` where `N` is the number of elements in `ages`. The nested loops have a constant time complexity of `O(121 * 121)` because they iterate over a fixed range independent of the input size. Therefore, the total time complexity of the code is predominated by the `Counter` operation, resulting in `O(N)`.

Space Complexity

The space complexity of the code is influenced by the additional storage needed for the `counter` variable, which depends on the number of unique elements in `ages`. In the worst case, each age is unique, so the space complexity for `counter` is `O(K)` where `K` is the number of unique ages. Given the constraints of the problem (ages are between 1 and 120), the maximum number of unique ages `K` can be `120`.

Therefore, the space complexity is `O(K)`, yet since `K` is constant and limited to `120`, this could also be considered `O(1)` in the context of this problem where `K` does not scale with the size of the input.