1854. Maximum Population Year

Easy Array Counting

Leetcode Link

Problem Description

You are provided with a 2D integer array called `logs`. Each element of `logs` is a two-element array that represents the birth and death years of a person, with `birth[i]` being the year they were born and `death[i]` being the year they died. The goal is to find which year had the highest population. A person is considered to be alive in a particular year if that year is between their birth year and the year before they died (inclusive of their birth year and exclusive of their death year).

The problem requires determining the year with the maximum population and, in case of a tie, returning the earliest year. This is akin to finding the year with the peak of a population timeline when plotted on a graph.

Intuition

To solve this problem, the first intuition is that it's not necessary to count the population for every single year in the range. Since the relevant years are defined by the birth or death years of individuals, the population changes only at these points.

We can track the population changes throughout the years by creating an array that represents the deltas in population size. For simplicity, let's base the years on an offset, considering 1950 as year 0, since the problem statement limits years to the 20th century.

For each person, we increment the population for their birth year and decrement it for their death year. This is because a birth adds one to the population, while a death reduces it by one, but not until the following year.

After populating the array with the deltas, we can find the year with the highest population by iterating through the array, maintaining a running sum which represents the current population, and updating the maximum population and year as we go. This process is like creating a cumulative sum or prefix sum array where each element is the total population up to that year.

The solution scans through the array, tracking the highest value seen and the earliest year this maximum occurred. The years we track in the array are offset by 1950, so when we find the result, we must add back the 1950 to obtain the actual year as the final answer.

Solution Approach

The solution uses an array `d` to represent the changes in population. The array has a length of 101 to cover each year from 1950 to 2050 (inclusive of 1950 and exclusive of 2050), which is enough to cover the possible range of years given the problem constraints.

The offset of 1950 is used to normalize the years so that they can easily fit into a fixed-size array and simplify the index calculations.

The solution follows these steps:

1. Initialize an array `d` to track the population change for each year, setting all elements to 0 initially.
2. Iterate through each person's logs. For each log, increment the population at the birth year and decrement the population at the death year. Since the person is not counted in their year of death, the decrement occurs at the index corresponding to their death year.

   For example, for a person born in 1950 (`birth[i] = 1950`) and died in 1951 (`death[i] = 1951`), the population increments by 1 at index 0 and decrements by 1 at index 1 in the `d` array.
3. Next, we need to convert the `d` array into a running sum of changes. This is done by iterating over `d` and adding the current change to a running sum `s`. At each step, the sum `s` represents the population for that year.
4. While generating the running sum, the solution keeps track of the maximum population `mx` seen so far and the corresponding year `j`. If the current running sum is greater than the previously recorded maximum, the maximum value and year are updated to reflect the new maximum.
5. After processing all of the years, the solution returns the year with the maximum population, which is `j + offset` to convert from the normalized index back to the actual year.

Within the code:

- `offset` is set to 1950 to normalize the years within the given range.
- The `for a, b in logs` loop processes each person's birth and death years and reflects them in the `d` array.
- The for loop `for i, x in enumerate(d)` iterates over the `d` array and maintains the running sum `s`, updates the maximum population `mx`, and keeps track of the earliest year `j` with this population.
- Finally, `return j + offset` normalizes the index back to the actual year.

This approach uses the prefix sum pattern to efficiently compute cumulative values over a sequence, and it leverages the fact that person counts only change at birth or death years, thus significantly reducing the number of necessary computations.

Example Walkthrough

Let's illustrate this solution approach with a small example:

Suppose we have the following `logs`:

```
logs = [[1950, 1960], [1955, 1970], [1950, 1955]]
```

Here's how we would walk through the problem:

1. Initialize the population changes array `d` with length 101, to cover years from 1950 to 2050. All values in `d` start at `0`.
2. Process the birth and death years from `logs`:
   - For the first person, born in 1950 and died in 1960, increment `d[0]` (1950 - 1950) by `1` and decrement `d[10]` (1960 - 1950) by `1`.
   - For the second person, born in 1955 and died in 1970, increment `d[5]` (1955 - 1950) by `1` and decrement `d[20]` (1970 - 1950) by `1`.
   - For the third person, born in 1950 and died in 1955, increment `d[0]` by `1` (already `1` from the first person, now becomes `2`) and decrement `d[5]` (already `1` from the second person, now becomes `0`).
3. After processing `logs`, our `d` array now has these changes (only showing the first few indices with changes):

```
d = [2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, ... ]
```

4. Now, we convert `d` to a running sum, `s`, starting at `0`, and we keep track of the maximum population `mx` and the year `j`:
   - Start at index `0`, `s = 2`, `mx = 2`, `j = 0` (which means the year 1950).
   - At index `5`, there's no change, so `s` remains `2`.
   - At index `10`, decrement `s` by `1`. Now `s = 1`, `mx` is still `2`, and `j` remains `0`.
   - Continue scanning until index `20`, where we decrement `s` by `1` again. Now `s = 0`.
5. The running sum does not exceed the maximum population (`mx`) of `2` which occurred first at index `0`.
6. The final step is to add back the offset to `j` to find the actual year. Since `j = 0` initially, and the offset is `1950`, the year with the highest population in our example is `j + offset = 0 + 1950 = 1950`.

Using this method allows us to scale the process for a larger set of input data efficiently, focusing only on years where population changes occur due to births and deaths. We avoid unnecessary calculations for years where no changes happen, thus optimizing the solution.

Python Solution

```python
class Solution:
    def maximumPopulation(self, logs: List[List[int]]) -> int:
        # Initialize a list to represent the population changes
        year_deltas = [0] * 101  # There are 101 years from 1950 to 2050

        # Define the offset for the year 1950, as all years are based on it
        year_offset = 1950

        # Accumulate births and deaths in year_deltas
        for birth, death in logs:
            birth -= year_offset
            death -= year_offset
            year_deltas[birth] += 1  # Increment population for birth
            year_deltas[death] -= 1  # Decrement population for death

        # Initialize variables to track the maximum population and the year
        sum_population = max_population = year_with_max_population = 0

        # Loop over each year to find the year with the max population
        for year, delta in enumerate(year_deltas):
            sum_population += delta
            if sum_population > max_population:
                max_population = sum_population
                year_with_max_population = year

        # Return the year with the maximum population by adding back the offset
        return year_with_max_population + year_offset
```

Java Solution

```java
class Solution {
    public int maximumPopulation(int[][] logs) {

        // Create an array to record the changes in population for each year.
        // Since we are interested in years between 1950 and 2050, we use an array of size 101.
        int[] populationDeltas = new int[101];
        // Offset to adjust years to indexes (1950 becomes 0, 1951 becomes 1, etc.).
        final int offset = 1950;

        // Iterate over each log entry.
        for (int[] log : logs) {
            // Calculate the starting index based on the birth year.
            int birthYearIndex = log[0] - offset;
            // Calculate the end index based on the death year.
            int deathYearIndex = log[1] - offset;

            // Increment the population for the birth year index.
            ++populationDeltas[birthYearIndex];
            // Decrement the population for the death year index.
            --populationDeltas[deathYearIndex];
        }

        // Sum and max population to find the maximum population year.
        int currentPopulation = 0;   // Current accumulated population.
        int maxPopulation = 0;       // Maximum population we have seen so far.
        int maxPopulationYearIndex = 0;  // Index (relative to offset) of the year with the maximum population.

        // Iterate through each year to find the year with the maximum population.
        for (int i = 0; i < populationDeltas.length; ++i) {
            // Update the current population by adding the delta for the current year.
            currentPopulation += populationDeltas[i];
            // If the current population is greater than the maximum seen before.
            if (maxPopulation < currentPopulation) {
                // Update the maximum population to the current population.
                maxPopulation = currentPopulation;
                // Update the year index to the current index.
                maxPopulationYearIndex = i;
            }
        }

        // Return the actual year by adding the offset to the index.
        return maxPopulationYearIndex + offset;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int maximumPopulation(vector<vector<int>>& logs) {
        int populationDeltas[101] = {0}; // Initialize all years' population changes to 0
        const int baseYear = 1950; // Use 1950 as the offset since all years are 1950 or later
        for (const auto& log : logs) {
            int birthYearIndex = log[0] - baseYear; // Convert birth year to index based on base year
            int deathYearIndex = log[1] - baseYear; // Convert death year to index based on base year
            ++populationDeltas[birthYearIndex]; // Increment population for birth year
            --populationDeltas[deathYearIndex]; // Decrement population for death year
        }
        int currentPopulation = 0; // Start with zero population
        int maxPopulation = 0; // Initialize max population to zero
        int yearWithMaxPopulation = 0; // This will store the year with the highest population
        for (int i = 0; i < 101; ++i) {
            currentPopulation += populationDeltas[i]; // Update population for the year
            if (maxPopulation < currentPopulation) {
                maxPopulation = currentPopulation; // Update the maximum population if current is greater
                yearWithMaxPopulation = i; // Record the year index that has the new maximum population
            }
        }
        return yearWithMaxPopulation + baseYear; // Convert the index back to an actual year and return it
    }
};
```

Typescript Solution

```typescript
// Define the function maximumPopulation which expects logs as an array of number pairs
function maximumPopulation(logs: number[][]): number {
    // Create an array to store the population changes over the years, initialized to zero.
    // The range covers years from 1950 to 2050 because of the problem constraints.
    const populationDeltas: number[] = new Array(101).fill(0);

    // Set offset to match the starting year 1950 to array index 0
    const baseYear = 1950;

    // Iterate through each log entry
    for (const [birth, death] of logs) {
        // Increment the population count for the birth year
        populationDeltas[birth - baseYear]++;
        // Decrement the population count for the death year
        populationDeltas[death - baseYear]--;
    }

    // Initialize variables for tracking the max population and the year with max population
    let maxPopulationYearIndex: number = 0;
    for (let i = 0, currentPopulation = 0, maxPopulation = 0; i < populationDeltas.length; ++i) {
        // Accumulate population changes to get the current population for each year
        currentPopulation += populationDeltas[i];

        // Check if the current population is greater than the max population observed so far
        if (maxPopulation < currentPopulation) {
            // Update max population and the index of the year with max population
            maxPopulation = currentPopulation;
            maxPopulationYearIndex = i;
        }
    }

    // Return the year corresponding to the index with the max population
    return maxPopulationYearIndex + baseYear;
}
```

Time and Space Complexity

Time Complexity:

The time complexity of the solution involves iterating through the logs and updating the corresponding years, followed by finding the year with the maximum population. The `for a, b in logs:` loop runs once for each set of birth and death years in `logs`, hence it is `O(N)`, where `N` is the number of elements in `logs`. The second `for` loop to find the maximum population runs for a fixed size array `d` of size 101 (since the years are offset by 1950 and only range between 1950 and 2050). This iteration is therefore `O(101)`, which is a constant `O(1)` operation. The overall time complexity is the higher of the two, so the time complexity is `O(N)`.

Space Complexity:

The space complexity is determined by the additional space used by the solution. The array `d` has a fixed size of 101, which is `O(1)` space complexity. The space used for the variables `a`, `b`, `s`, `mx`, and `j` is also constant. Therefore, the space complexity is `O(1)`.