949. Largest Time for Given Digits

Medium String Enumeration Leetcode Link

Problem Description

The problem provides us with an array of 4 digits. Our task is to use these digits to form the latest possible valid 24-hour format time. The 24-hour time format is represented as "HH:MM", where "HH" is the hour part ranging from 00 to 23 and "MM" is the minute part ranging from 00 to 59. We must use each given digit exactly once. If it's possible to form a valid time, we should return it as a string in the "HH:MM" format; otherwise, if no valid combination exists to make a proper time, we should return an empty string.

The key considerations here include ensuring that the hours are within the valid range (00 to 23) and the minutes fall within the right range (00 to 59).

Intuition

To solve this problem, the solution adopts a brute-force strategy; it iterates over all possible permutations of the given four digits and checks which of these permutations can form a valid time. The idea here is that there are only 24 distinct permutations for 4 digits, which is a small and manageable number to examine exhaustively.

Here's the breakdown of the solution approach:

- The solution uses three nested loops to iterate over all possible combinations of the four digits without repetition. Each loop variable (`i`, `j`, `k`) represents an index for a digit in the array `arr`.
- To ensure no digit is repeated, the conditions `i != j`, `i != k`, and `j != k` are enforced.
- Out of these three chosen indices, the hour can be formed by combining the digits at index `i` and `j`. The minute is then formed by combining the digits at index `k` and the remaining digit (which is not used so far). The remaining index is calculated as `6 - i - j - k`, leveraging the fact that the indices are 0, 1, 2, 3 for a 4-element array and their sum is 6.
- The hours (`h`) are valid if they are less than 24, and the minutes (`m`) are valid if they are less than 60.
- If a combination forms a valid time, it's converted to total minutes (`h * 60 + m`) to compare easily with other valid times found so far. The maximum value is maintained as `ans`.
- If no valid time is found, `ans` remains -1, and an empty string is returned.
- If a valid time is found, it is formatted correctly by converting the total minutes back into hours and minutes and then constructing a string with a proper "HH:MM" format using string formatting with padding zeroes where needed.

Thus, the solution explores all permutations to find the maximum possible valid time strictly using the 24-hour time constraints and efficiently formats the final answer.

Solution Approach

The implementation uses a simple backtracking algorithm to generate all permutations of the input array and validates each one as a potential 24-hour time.

To begin, a loop runs four times, from `0` to `3`, corresponding to the first digit of the hour. Inside it, two nested loops run again from `0` to `3` to account for the second digit of the hour and the first digit of the minutes, respectively. The criteria for each of these loops to continue iterating are that their indices must not be equal (`i != j != k`) to avoid using the same digit more than once.

The digits for the hour are created by the expression `arr[i] * 10 + arr[j]`, effectively placing `arr[i]` as the tens digit and `arr[j]` as the ones digit. Similarly, the minutes are created by the expression `arr[k] * 10 + arr[6 - i - j - k]`, where `6 - i - j - k` gives us the one unused index to complete the pair needed for the minutes.

Once `h` (hours) and `m` (minutes) are calculated, they're validated to check if they represent a valid time (i.e., `h < 24` and `m < 60`). If the time is valid, we calculate the total minutes since `00:00` using `h * 60 + m` and update `ans` with the maximum of itself and the computed total minutes. This effectively stores the latest valid time seen so far.

After all permutations are examined, the solution checks if `ans` has been updated from its initial value of `-1`. If it hasn't, this means no valid time was found, and an empty string is returned. If `ans` has been updated, the total minutes are split back into hours and minutes. The formatting for the output string is done using string interpolation where `ans // 60` gives the hours and `ans % 60` gives the minutes. The `:02` formatter is used to pad the output with leading zeroes if needed, ensuring the correct "HH:MM" format is maintained.

Though not explicitly stated, this implementation assumes that mathematical operations and comparisons are constant-time operations, and since there are only `4!` (factorial of 4 i.e., 24) possible permutations, the overall time complexity of the algorithm is O(1) - although it's technically a brute-force solution, its complexity is bound by a small constant because the input size is fixed at 4.

In summary, the implementation utilizes:

- A brute-force approach to generate all possible permutations using nested loops.
- Mathematical operations and checks to construct valid 24-hour times from the permutations.
- A running maximum (`ans`) to determine the latest valid time.
- String formatting and modularity to convert the total minutes back into the required "HH:MM" output format.

Example Walkthrough

Let's say the input array of 4 digits is `[1, 8, 3, 2]`. We are tasked with finding the latest valid time that can be formed using these digits. Using the solution approach described above, we would proceed as follows:

1. Start by initializing the `ans` variable to `-1` to track the latest valid time in minutes.

2. Begin iterating over all possible permutations of these four digits using three nested loops.

3. For the first permutation, we might start with `i=0`, `j=1`, `k=2`. Here, `i`, `j`, and `k` correspond to indices in the digits' array, so the digits would be `arr[0] = 1`, `arr[1] = 8`, `arr[2] = 3`.

4. Since we need two digits for the hour (`HH`) and two for the minutes (`MM`), we choose the first two for the hour and the last two for the minute. Here, `i` and `j` are for the hours, and `k` and the unused index for the minutes.

5. The hours would be `arr[i] * 10 + arr[j]`, which is `1 * 10 + 8 = 18`. The minutes would be `arr[k] * 10 + arr[6 - i - j - k]`, which is `3 * 10 + arr[6 - 0 - 1 - 2] = 30 + arr[3] = 30 + 2 = 32`. The time so far is `18:32`.

6. The time `18:32` is a valid 24-hour format time since `18 < 24` and `32 < 60`. Therefore, we calculate the total minutes as `18 * 60 + 32 = 1152` and update `ans` with this value since it's the first and thus the maximum found so far.

7. We continue this process, generating and checking each permutation. Some permutations would be invalid, for instance, if the chosen indices led to hours >= 24 or minutes >= 60; these permutations are ignored.

8. Assume that after checking all permutations, the largest value for `ans` remains `1152`. This would be the latest valid time in terms of total minutes from `00:00`.

9. We convert `1152` back into hours and minutes. `1152 // 60` gives us `19` hours and `1152 % 60` gives us `12` minutes.

10. We format these back into the string representation `"19:12"`, padding the hours or minutes with a leading zero if necessary to adhere to the "HH:MM" format. However, in this case, no padding is needed.

Finally, the latest valid time we can form with the input array `[1, 8, 3, 2]` is `"19:12"`, so this is the string we would return. If we had found no valid permutations, we would return an empty string.

Python Solution

```python
from typing import List

class Solution:
    def largest_time_from_digits(self, digits: List[int]) -> str:
        # Initialize the variable to store the maximum minutes which
        # represents the time formed by the digits
        max_minutes = -1

        # Generate all possible combinations of the digits
        for i in range(4):
            for j in range(4):
                for k in range(4):
                    # Ensure the digits i, j, and k are distinct
                    if i != j and i != k and j != k:
                        # Calculate the hours by taking digits[i] as the tens place
                        # and digits[j] as the ones place
                        hours = digits[i] * 10 + digits[j]
                        # Calculate the minutes by taking digits[k] as the tens place
                        # and digits[6 - i - j - k] as the ones place (since 0+1+2+3=6)
                        # to ensure using all distinct indices
                        minutes = digits[k] * 10 + digits[6 - i - j - k]

                        # Check if the hours and minutes are valid time values
                        if hours < 24 and minutes < 60:
                            # Update the maximum minutes if the current combination
                            # is greater than what we have so far
                            max_minutes = max(max_minutes, hours * 60 + minutes)

        # If no valid time could be formed, return an empty string
        if max_minutes < 0:
            return ""
        else:
            # Format the time in HH:MM format and return
            return f'{max_minutes // 60:02}:{max_minutes % 60:02}'

```

Java Solution

```java
class Solution {
    public String largestTimeFromDigits(int[] digits) {
        // Initialize the maximum time represented in minutes to an invalid value
        int maxTime = -1;

        // Generate all possible times using the four digits
        // We'll use these digits to create hours and minutes
        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 4; ++j) {
                for (int k = 0; k < 4; ++k) {
                    // Ensure i, j, k are all different to use different digits
                    if (i == j || j == k || i == k) {
                        continue; // Skip the iterations where any two are the same
                    }
                    // Create the hour by combining the ith and jth digits
                    int hour = digits[i] * 10 + digits[j];
                    // Create the minute by combining the kth and remaining digit
                    // The sum of indices (i, j, k, l) for 4 elements is 0+1+2+3=6
                    // l, the index of remaining element, is 6 - (i + j + k)
                    int remainingIndex = 6 - i - j - k;
                    int minute = digits[k] * 10 + digits[remainingIndex];

                    // Validate if the time created is a valid 24-hour format time
                    if (hour < 24 && minute < 60) {
                        // Convert hour and minute into total minutes
                        int totalMinutes = hour * 60 + minute;
                        // Update the maximum time found so far in minutes
                        maxTime = Math.max(maxTime, totalMinutes);
                    }
                }
            }
        }

        // If no valid time found, return an empty string
        if (maxTime == -1) {
            return "";
        }

        // Construct the time in HH:MM format from maxTime
        // maxTime divided by 60 gives the hour and modulo 60 gives the minute
        // "%02d" means an integer is formatted to a string with at least 2 digits, padding with zeros
        return String.format("%02d:%02d", maxTime / 60, maxTime % 60);
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

class Solution {
public:
    // Function to find the largest time that can be made from the given digits
    string largestTimeFromDigits(vector<int>& digits) {
        // Initialize the answer with -1 as an indicator that no valid time is found
        int maxTime = -1;

        // Use nested loops to generate all possible times using the provided digits
        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 4; ++j) {
                for (int k = 0; k < 4; ++k) {
                    // Ensure that i, j, and k are all different to use each digit once
                    if (i != j && j != k && i != k) {
                        // Form the hours component by combining two different digits
                        int hours = digits[i] * 10 + digits[j];
                        // Form the minutes component by combining the remaining two digits
                        int minutes = digits[k] * 10 + digits[6 - i - j - k]; // 6 is the sum of indices (0+1+2+3)
                        // Validate if the formed hours and minutes constitute a valid time
                        if (hours < 24 && minutes < 60) {
                            // Update maxTime with the maximum of current time and maxTime
                            maxTime = std::max(maxTime, hours * 60 + minutes);
                        }
                    }
                }
            }
        }

        // If no valid time found, return an empty string
        if (maxTime < 0) {
            return "";
        }

        // Calculate the hours and minutes from maxTime which is in total minutes
        int maxHours = maxTime / 60;
        int maxMinutes = maxTime % 60;

        // Form the resulting string in the format "HH:MM"
        return FormattedTime(maxHours) + ":" + FormattedTime(maxMinutes);
    }

private:
    // Helper function to format the hours or minutes into a 2-digit string
    string FormattedTime(int value) {
        return std::string(1, '0' + value / 10) + std::string(1, '0' + value % 10);
    }
};

```

Typescript Solution

```typescript
// Define a type for the array of digits.
type DigitsArray = number[];

// Function to format the hours or minutes into a 2-digit string.
const formattedTime = (value: number): string => {
    return String('0' + value).slice(-2);
}

// Function to find the largest time that can be made from the given digits.
const largestTimeFromDigits = (digits: DigitsArray): string => {
    // Initialize the answer with -1 as an indicator that no valid time is found.
    let maxTime: number = -1;

    // Use nested loops to generate all possible times using the provided digits.
    for (let i = 0; i < 4; ++i) {
        for (let j = 0; j < 4; ++j) {
            for (let k = 0; k < 4; ++k) {
                // Ensure that i, j, and k are all different to use each digit once.
                if (i !== j && j !== k && i !== k) {
                    // Form the hours component by combining two different digits.
                    const hours: number = digits[i] * 10 + digits[j];
                    // Form the minutes component by combining the remaining two digits.
                    // 6 is the sum of indices (0+1+2+3) to find the remaining index.
                    const remainingIndex: number = 6 - i - j - k;
                    const minutes: number = digits[k] * 10 + digits[remainingIndex];
                    // Validate if the formed hours and minutes constitute a valid time.
                    if (hours < 24 && minutes < 60) {
                        // Update maxTime with the maximum of current time and maxTime.
                        maxTime = Math.max(maxTime, hours * 60 + minutes);
                    }
                }
            }
        }
    }

    // If no valid time was found, return an empty string.
    if (maxTime === -1) {
        return "";
    }

    // Calculate the hours and minutes from maxTime which is in total minutes.
    const maxHours: number = Math.floor(maxTime / 60);
    const maxMinutes: number = maxTime % 60;

    // Form the resulting string in the format "HH:MM".
    return `${formattedTime(maxHours)}:${formattedTime(maxMinutes)}`;
};

```

Time and Space Complexity

The given code snippet is designed to find the largest possible time that can be displayed on a 24-hour clock, using digits from a given array of four integers.

Time Complexity:

The time complexity of the code is `O(1)` because the number of iterations is fixed and does not depend on the input size, which is always 4 elements in the `arr` list. Specifically, the number of iterations would be a fixed number since we are dealing with three nested loops each running for four possible options but skipping the indexes that have already been used (`i != j` and `i != k` and `j != k`). Therefore, the maximum number of iterations is bounded by `4 * 3 * 2 = 24` for each of the three loops (since one number is used as the first element of the hour, one as the first element of the minute, and one as the second element of the minute, while the remaining is the second element of the hour). Even though some iterations are skipped, 24 is the upper limit, which means the time complexity does not depend on the input size but is rather constant.

Space Complexity:

The space complexity of the algorithm is `O(1)`. The extra space used by the algorithm does not grow with the input size. The variables `i`, `j`, `k`, `h`, `m`, and `ans` use a constant amount of space, and there are no additional data structures that grow with input size.

Thus, regardless of the input, the space used by the program remains constant.