401. Binary Watch

Easy Bit Manipulation Backtracking Leetcode Link

Problem Description

A binary watch displays the time using binary representation with LEDs. A unique feature of this watch is that it has separate sets of LEDs for hours and minutes. The top set of 4 LEDs represent the hours (0 through 11), and the bottom set of 6 LEDs represent the minutes (0 through 59). Each LED signifies a binary digit (bit) with values 0 or 1, with the least significant bit being on the far right. As in any binary number, the bits are weighted based on their position, doubling from right to left (1, 2, 4, 8, etc.). When a certain number of LEDs are turned on, the problem requires us to find all the possible times that configuration can represent on a binary watch.

There are a few rules to keep in mind:

- The hour representation does not have a leading zero. So "01:00" is not an acceptable representation; it should be "1:00".
- The minutes must be two digits, potentially including a leading zero if necessary. For example, "10:2" is not valid; it should be "10:02".

A key task in this problem is to list all the possible times that can be displayed by the watch, given the number of LEDs that are lit, represented by the integer `turnedOn`.

Intuition

To solve this problem, we approach it by simulating the behavior of the binary watch:

1. We generate all possible combinations of the hour and minute by iterating through their maximum possible values, which are 11 for hours (from 0 to 11) and 59 for minutes (from 0 to 59).
2. For each generated hour and minute combination, we convert them to their binary representation and concatenate the strings.
3. The combined string of binary representation for both hour and minute is then checked for the number of '1's it contains. If it matches the `turnedOn` value, it means this time combination has the correct number of LEDs lit.
4. We add the valid time combinations formatted correctly, "{hours}:{minutes}", ensuring minutes are always two digits (padded with leading zero if necessary).
5. The final step is to return all the valid formatted times.

Understanding that the processes of counting lit LEDs and formatting strings are separate, yet equally important parts of the solution enables us to effectively iterate over possible time combinations and filter out the valid ones. This approach efficiently uses Python's list comprehension, formatting, and string operation utilities to come up with a succinct and elegant solution.

Solution Approach

The solution uses a simple, brute force approach to find all valid times that can be represented on a binary watch with a given number of LEDs turned on. Here's a step-by-step breakdown:

1. We initialize an empty list that will be used to store the valid time representations as strings.

2. We utilize two nested `for` loops, with the outer loop iterating through the possible hours (0 to 11) and the inner loop iterating through the possible minutes (0 to 59). This ensures that we cover every potential combination of hours and minutes.

3. For each combination of the hour and minute, we first convert them to binary strings using Python's built-in `bin()` function. This results in strings like `'0b10'` for the decimal number 2.

4. We concatenate the binary strings of both the hour and minute and use the string method `.count('1')` to calculate the total number of '1's, which correspond to the LEDs being on.

5. We compare the count of '1's with the `turnedOn` parameter. If they match, it means the current hour and minute combination is one of the valid representations for the given number of LEDs turned on.

6. For each valid time representation, we format the hour and minute in the string `"{:d}:{:02d}".format(i, j)` to comply with the required time format (no leading zero for hours and two digits with a leading zero for minutes when necessary).

7. We append the formatted time string to our list of results.

8. After iterating through all the possible combinations, the list of valid time representations is complete, and we return it as the final result.

This solution is straightforward since it iterates through all possible representations without the use of complex algorithms or data structures. The use of list comprehensions in Python provides a concise way to pack this entire process into a single line of code while maintaining high readability.

It's worth noting that this brute force solution is feasible because the total number of possible times in a day (12 hours * 60 minutes = 720 possibilities) is small, and therefore, iterating over all of them is not computationally expensive. For a much larger set of possibilities, a more intricate algorithm, perhaps using backtracking or bit manipulation, would have been necessary to keep the computation time practical.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using `turnedOn = 3`.

We will follow the solution steps to find all possible times when exactly 3 LEDs are lit on the binary watch:

1. Start with an empty list to hold our valid times.

2. Iterate over possible hours (0 to 11) and minutes (0 to 59):

   For example, if we select hour 3 (`011` in binary) and minute 5 (`000101` in binary), we count the '1's in the concatenated binary string `011000101`.

3. When we convert these to binary and concatenate we get: `'0b1100b101'`. Cleaning up the binary representation (removing '0b'), we have `'1100101'`.

4. Count the number of '1's in the binary representation: `'1100101'` has four '1's, which does not match our `turnedOn` value of 3. So, this combination is skipped.

5. Continue this process with other combinations:

   - Hour 1 (`1` in binary) and minute 10 (`1010` in binary) total to 3 '1's when concatenated (`11010`), making `1:10` a valid time.
   - Hour 4 (`100` in binary) and minute 3 (`11` in binary) are another example that totals to 3 '1's when concatenated (`1000011`), giving us another valid time: `4:03`.
6. Continue looping through and only add the combinations that have exactly 3 '1's to the list after formatting them correctly.

7. After finishing the loop, we will have a list of valid times.

For a `turnedOn` value of 3, the list might include times like `1:10`, `4:03`, `0:07`, and more, assuming they match the criteria of having exactly 3 LEDs turned on.

8. Return this list as the final result.

Python Solution

```python
from typing import List

class Solution:
    def readBinaryWatch(self, num_leds_lit: int) -> List[str]:
        # This list comprehension will collect all possible time formats
        time_formats = [
            '{:d}:{:02d}'.format(hour, minute)
            for hour in range(12)       # Loop through the 12 hours
            for minute in range(60)     # Loop through the 60 minutes
            if (bin(hour) + bin(minute)).count('1') == num_leds_lit
            # Check if the sum of the bits set to '1' in both the hour's and
            # minute's binary representation equals the number of LEDs that are lit
        ]

        return time_formats

```

Java Solution

```java
class Solution {
    public List<String> readBinaryWatch(int numLEDsOn) {
        // This list will hold all the possible times the binary watch can represent with numLEDsOn LEDs lit.
        List<String> possibleTimes = new ArrayList<>();

        // Loop over all possible hours (0-11)
        for (int hour = 0; hour < 12; ++hour) {
            // Loop over all possible minutes (0-59)
            for (int minute = 0; minute < 60; ++minute) {
                // Combine the number of bits (LEDs) set in both hour and minute
                // If the total equals numLEDsOn, this is a correct time and should be added to the list
                if (Integer.bitCount(hour) + Integer.bitCount(minute) == numLEDsOn) {
                    // Format the time as "h:mm" and add it to the list
                    possibleTimes.add(String.format("%d:%02d", hour, minute));
                }
            }
        }
        // Return all the possible times
        return possibleTimes;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include for using vector
#include <string> // Include for using string
#include <bitset> // Include for using bitset

class Solution {
public:
    // Defines a method to read a binary watch.
    // It takes an integer 'turnedOn' to indicate the number of LEDs that are on.
    vector<string> readBinaryWatch(int turnedOn) {
        vector<string> ans; // This will hold all possible times the binary watch could represent.

        // Loop over all possible hours (0 to 11).
        for (int hour = 0; hour < 12; ++hour) {
            // Loop over all possible minutes (0 to 59).
            for (int minute = 0; minute < 60; ++minute) {
                // Check if the total count of bits (LEDs on) for both hour and minute equals 'turnedOn'.
                if (bitset<10>(hour << 6 | minute).count() == turnedOn) {
                    // Use a conditional statement to format minutes (add leading zero if less than 10).
                    string minuteFormatted = minute < 10 ? "0" + std::to_string(minute) : std::to_string(minute);

                    // Construct the time string and add it to the list of answers.
                    ans.push_back(std::to_string(hour) + ":" + minuteFormatted);
                }
            }
        }
        return ans; // Return the list of valid times.
    }
};

```

Typescript Solution

```typescript
function readBinaryWatch(turnedOn: number): string[] {
    // Base case: if no LED is turned on, the time is 0:00
    if (turnedOn === 0) {
        return ['0:00'];
    }

    // Initialize the array representing the state of each LED in the watch
    const ledState = new Array(10).fill(false);
    // Function to create the time string from the LED state array
    const createTimeString = () => {
        // Calculate the hour by interpreting the first 4 bits
        const hours = ledState.slice(0, 4).reduce((acc, led) => (acc << 1) | Number(led), 0);
        // Calculate the minutes by interpreting the last 6 bits
        const minutes = ledState.slice(4).reduce((acc, led) => (acc << 1) | Number(led), 0);
        // Return them as a tuple
        return [hours, minutes];
    };
    // Helper function that generates time strings given an index and remaining count of LEDs that can be turned on
    const backtrack = (index: number, count: number) => {
        // If index plus count exceeds total LEDs, return since we cannot have more turned on LEDs
        // And if count is zero, we know no more LEDs can be turned on.
        if (index + count > ledState.length || count === 0) {
            return;
        }

        // Turn on the LED at the current index and attempt to build time
        ledState[index] = true;
        if (count === 1) {
            // If exactly one LED is left to turn on, create and validate the time
            const [hours, minutes] = createTimeString();
            if (hours < 12 && minutes < 60) {
                // Format the time with leading zero for minutes, if needed, and add to results
                results.push(`${hours}:${minutes < 10 ? '0' + minutes : minutes}`);
            }
        }
        // Recurse for the next LED with one less LED available to turn on
        backtrack(index + 1, count - 1);

        // Backtrack: turn off the LED and try next positions
        ledState[index] = false;
        backtrack(index + 1, count);
    };

    // Initialize the results array
    const results: string[] = [];

    // Call the helper function starting with the first LED
    backtrack(0, turnedOn);

    // Return the resulting time strings
    return results;
}

```

Time and Space Complexity

The given Python code snippet is a function that generates all possible times displayed on a binary watch, specifically when a certain number of LEDs are turned on. The time complexity and space complexity of the code are analyzed as follows:

Time Complexity

The time complexity is determined by the number of iterations in the nested loops and the operation within the loop:

- The outer loop runs 12 times (`i` from 0 to 11) because there are 12 possible hours on a watch.
- The inner loop runs 60 times (`j` from 0 to 59) to represent 60 possible minutes.
- For each combination of `i` and `j`, there are operations that convert these integers to binary strings (`bin()`), concatenate them, and count the number of `'1'` bits.

As such, the time complexity can be expressed as the product of the number of iterations in the loops and the complexity of the operations within them. Assuming that `bin()` and `.count('1')` operations are `O(m)` and `O(n)` respectively, where `m` and `n` are the number of bits in the hour and minute parts. The hour part has at most 4 bits, and the minute part has at most 6 bits. However, since these bit lengths are fixed and independent of input size, their contribution to complexity is constant.

Therefore, the overall time complexity is `O(12 * 60)`, which simplifies to `O(1)` — constant time complexity, as the loop bounds do not depend on the size of the input, but on the fixed size of hours and minutes on a binary watch.

Space Complexity

The space complexity is determined by the space needed to store the output and any intermediary data:

- The list comprehension generates up to a maximum of `12 * 60` time strings (in case `turnedOn` is 0, which would imply all possible times). Each time is a string, and the length of the strings has an upper bound, so they can be considered to take constant space.
- Hence, the space complexity is directly related to the number of valid times that can be outputted, which in the worst case is `12 * 60`.

Therefore, the space complexity is `O(1)` — constant space complexity, since the maximum number of times that can be represented on a binary watch is fixed and does not grow with the size of the input.