1618. Maximum Font to Fit a Sentence in a Screen

Medium Array String Binary Search Interactive Leetcode Link

Problem Description

The problem requires us to display a given string `text` on a screen with specific width `w` and height `h`. We are given a list of possible font sizes `fonts` sorted in ascending order, and we need to find out the largest font size we can use to display the entire text on the screen.

We are provided with a `FontInfo` interface that contains two methods:

- `getWidth(fontSize, ch)`: which returns the width of character `ch` when `fontSize` is used.
- `getHeight(fontSize)`: which provides the height of any character when `fontSize` is used.

We must ensure that the total width of `text`, calculated by summing up the width of each character in `text` using a given `fontSize`, does not exceed the screen width `w`. Additionally, the height of the text for the chosen `fontSize`, given by `getHeight(fontSize)`, cannot be greater than the screen height `h`. The `text` is displayed in a single line.

Constraints are applied to both font width and height to ensure that they are non-decreasing with the increase in font size. If no font size can make the text fit on the screen, we are to return `-1`.

Intuition

This problem can naturally be approached with a binary search strategy, since the font sizes are sorted and we need to find the maximum `fontSize` that fits the constraints. The key steps when applying binary search in this context are:

- Define a function `check()` that determines if a given `fontSize` can be used to fit the text within the dimensions of the screen (`w` and `h`).
- Run a binary search over the array of font sizes. At each step, the middle font size is tested using the `check()` function.
- If `check()` returns `true`, indicating the current font size fits the text within the screen, we search in the higher (larger font size) half of the remaining fonts, since our goal is to find the maximum size.
- If `check()` returns `false`, we search in the lower half, discarding the current and larger font sizes as they will also not fit.
- We keep narrowing our search range until we pinpoint the maximum size that fits, or we determine that no font size is suitable, in which case we return `-1`.

The binary search concludes either when we have successfully found the largest `fontSize` that allows the text to fit within the given `w` and `h`, or when we have checked all possible font sizes, and none can fit the text.

Solution Approach

The provided solution in Python implements the binary search strategy to identify the maximum usable font size as follows:

1. A helper function `check(size)` is defined that accepts a font size and determines if the text can be displayed on the screen using that particular font size. The function checks two conditions:
   - Whether the height of the text using the specified `fontSize` exceeds the available height `h` of the screen.
   - Whether the total width of `text` at the specified `fontSize` can fit within the width `w` of the screen by summing the widths of all characters in `text` with `getWidth(fontSize, c)`.
2. The binary search algorithm is used by initializing two pointers: `left` and `right`. `left` starts at 0, and `right` starts at the last index of the `fonts` array.
3. The main loop of the binary search continues until `left` and `right` converge. In each iteration of the loop, a variable `mid` is calculated to find the middle index of the current search range (`left` and `right`). We calculate `mid` using the expression `(left + right + 1) >> 1`, which is a bit manipulation technique to calculate the average of `left` and `right` and shift one bit to the right, effectively dividing by 2.
4. The `check(fonts[mid])` function call tests if the font size indicated by the `mid` index along with `text` will fit both width and height constraints on the screen. Two scenarios are possible:
   - If `true`, it means the text fits the screen with `fonts[mid]`, so the search continues in the right half (including `mid`) to see if a larger font size could also fit.
   - If `false`, the text does not fit with `fonts[mid]`, and hence all larger font sizes will not fit either, so the search is confined to the left half, excluding `mid`.
5. Upon completion of the loop, if the `check` function returns `true` for `fonts[left]`, that means `fonts[left]` is the largest font size that the text can be displayed with on the screen. If `check` fails for `fonts[left]`, then no available font sizes can display the text within the screen limits, and `-1` is returned.

The solution leverages the sorted nature of the `fonts` array and the binary search algorithm to efficiently zone in on the maximum font size that satisfies the display constraints. The time complexity of the binary search algorithm is O(log n), which significantly reduces the number of checks needed compared to a linear search, especially for larger font arrays.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using a binary search strategy mentioned in the problem description.

Suppose we have the following scenario:

- The given `text` is "LeetCode".
- The screen dimension, width (`w`), is 100 units, and the height (`h`), is 20 units.
- We have a sorted array of possible font sizes `fonts = [10, 12, 14, 16, 18, 20]`.
- A `FontInfo` interface with two hypothetical results:
  - `getWidth(fontSize, ch)` returns `fontSize * 1.2` units for each `ch`.
  - `getHeight(fontSize)` returns `fontSize` units.

Let's walk through the steps of the binary search strategy using this example:

1. We need to define a helper function `check(size)`. This function will determine whether or not the text can be displayed using a specific font size without exceeding the screen dimensions.
2. Our binary search starts with setting `left` to 0 and `right` to the last index of the `fonts` array, which is 5 in this case.
3. We enter a loop that will continue until `left` and `right` converge. Initially, `left = 0` and `right = 5`.
4. Calculate `mid`. Let's begin with the first iteration:
   - `mid = (0 + 5 + 1) >> 1` evaluates to `3`, so `fontSize = fonts[3]` is 16.
5. We call `check(16)` to see if we can use a font size of `16` units. The height at this size is `16` units, which does not exceed our screen height of `20` units. The width would be the sum of each character's width using `getWidth(16, ch)`, which is `16 * 1.2 * len("LeetCode")`. If this sum is less than or equal to `100` units, `check(16)` returns `true`.
6. Depending on the result, we adjust our `left` and `right`:
   - If `check(16)` is `true`, it means we can potentially use an even larger font size, so we set `left = mid`.
   - If `check(16)` is `false`, then we need to try smaller font sizes, and we set `right = mid - 1`.
7. We then repeat this process until `left` and `right` converge on the largest font size that can be used without exceeding the screen dimensions.
8. Eventually, suppose `left` converges on index `2`, with `check(fonts[2])` returning `true` and `check(fonts[3])` returning `false`. This would indicate that `14` is the largest font size we can use.

If during this iterative process no `fontSize` passes the `check` function, then no font size will fit the text on the screen and we return `-1`.

In our example, let's say after iterating we have found that `14` is the maximum size we can use to fit "LeetCode" within the screen dimensions, so that would be our function's return value. If every font size tested returns `false` in our `check` function, then we would return `-1`, indicating no suitable font size is available.

Python Solution

```python
from typing import List

class Solution:
    def maxFont(self, text: str, width: int, height: int, fonts: List[int], fontInfo: 'FontInfo') -> int:
        # Helper function to check if a given font size is valid
        # for the text with the given width and height constraints
        def is_valid_size(font_size):
            # Check if the height of the font is within the allowed height
            if fontInfo.getHeight(font_size) > height:
                return False
            # Check if the total width of all characters is within the allowed width
            return sum(fontInfo.getWidth(font_size, char) for char in text) <= width

        # Initialize the pointers for binary search
        left, right = 0, len(fonts) - 1
        # Variable to store the maximum valid font size found so far
        max_valid_font = -1

        # Perform a binary search to find the largest valid font size
        while left <= right:
            mid = (left + right) // 2
            if is_valid_size(fonts[mid]):
                # If the current size is valid, store it as the max found
                max_valid_font = fonts[mid]
                # Move the left pointer to search for a larger valid font size
                left = mid + 1
            else:
                # Move the right pointer to search for a smaller valid font size
                right = mid - 1

        return max_valid_font

# Note: The FontInfo class mentioned in the comments is assumed to be provided and not shown here.

```

Java Solution

```java
class Solution {

    // This function finds the maximum font size that can be used to fit `text` in a given
    // width `w` and height `h`, utilizing the provided `fonts` array and `fontInfo` API.
    public int maxFont(String text, int width, int height, int[] fonts, FontInfo fontInfo) {
        int left = 0;
        int right = fonts.length - 1;

        // Perform a binary search to find the largest valid font size.
        while (left < right) {
            // Calculate the middle index, leaning to the right in case of even number of elements.
            int middle = (left + right + 1) / 2;
            // Check if the current font size can fit the text.
            if (canFitText(text, fonts[middle], width, height, fontInfo)) {
                // If the text fits, move the lower bound of the search up.
                left = middle;
            } else {
                // If the text doesn't fit, decrease the upper bound.
                right = middle - 1;
            }
        }

        // After exiting the loop, check if the font at the left index fits.
        return canFitText(text, fonts[left], width, height, fontInfo) ? fonts[left] : -1;
    }

    // This helper function checks if `text` can fit within given `width` and `height`
    // constraints with the specified font size, using the `fontInfo` API.
    private boolean canFitText(String text, int fontSize, int width, int height, FontInfo fontInfo) {
        // Check the height of the font first; if it's too tall, return false.
        if (fontInfo.getHeight(fontSize) > height) {
            return false;
        }

        // Sum the width of each character in the text with the current font size.
        int totalWidth = 0;
        for (char character : text.toCharArray()) {
            totalWidth += fontInfo.getWidth(fontSize, character);
            // If the total width exceeds the allowed width, return false.
            if (totalWidth > width) {
                return false;
            }
        }

        // If the loop completes without returning, the text fits within the width.
        return true;
    }
}

```

C++ Solution

```cpp
/**
 * // This is the FontInfo's API interface.
 * // You should not implement it, or speculate about its implementation
 * class FontInfo {
 *   public:
 *     // Returns the width of character ch when using fontSize.
 *     int getWidth(int fontSize, char ch);
 *
 *     // Returns the height of any character when using fontSize.
 *     int getHeight(int fontSize)
 * };
 */
class Solution {
public:
    // Function to find the maximum font size that can be used to fit text into given dimensions (w by h).
    // The fonts vector contains all possible font sizes in ascending order.
    int maxFont(string text, int width, int height, vector<int>& fonts, FontInfo fontInfo) {
        // Lambda to check if a given fontSize can be used for the text within the specified dimensions.
        auto canUseFontSize = [&](int fontSize) {
            // If the font height exceeds the allowed height, this font size is not viable.
            if (fontInfo.getHeight(fontSize) > height) return false;

            // Check if the total width of characters exceeds the allowed width.
            int totalWidth = 0;
            for (char c : text) {
                totalWidth += fontInfo.getWidth(fontSize, c);
                // As soon as we exceed width, return false.
                if (totalWidth > width) return false;
            }
            // The fontSize is viable since the characters fit within the width.
            return true;
        };

        // Binary search initialization - indices for the range of fonts array.
        int left = 0, right = fonts.size() - 1;

        // Modified binary search to find the biggest viable font size.
        while (left < right) {
            // Choose the mid font size. Use right + 1 to prevent infinite loop in case of two elements.
            // Shift right by 1 is equivalent to integer division by 2.
            int mid = (left + right + 1) / 2;

            // If mid font size is viable, move the left boundary to mid.
            if (canUseFontSize(fonts[mid])) {
                left = mid;
            } else {
                // Otherwise, eliminate the mid size and all greater sizes.
                right = mid - 1;
            }
        }

        // After narrowing down, check if the left boundary font size is viable.
        // If it is, return its size, otherwise return -1 indicating no font size fits.
        return canUseFontSize(fonts[left]) ? fonts[left] : -1;
    }
};

```

Typescript Solution

```typescript
// Define the interface for FontInfo which outlines the available methods.
interface FontInfo {
    getWidth(fontSize: number, ch: string): number;
    getHeight(fontSize: number): number;
}

/**
 * Finds the maximum font size from the list of available font sizes that can be used to fit the given text
 * within the specified width (w) and height (h) constraints.
 *
 * @param {string} text The text that needs to be fit into the specified width and height.
 * @param {number} w The maximum allowed width.
 * @param {number} h The maximum allowed height.
 * @param {number[]} fonts An array of font sizes available.
 * @param {FontInfo} fontInfo An object that provides the width and height of a character for a given font size.
 * @return {number} The maximum font size that can be used to fit the text, or -1 if no font size fits.
 */
const maxFont = (text: string, w: number, h: number, fonts: number[], fontInfo: FontInfo): number => {
    // A helper function to check whether a given font size can be used to fit the text within the constraints.
    const canFitTextWithFontSize = (size: number): boolean => {
        if (fontInfo.getHeight(size) > h) {
            return false; // The height of the font exceeds the max height allowed.
        }
        let widthUsed: number = 0;
        for (const char of text) {
            widthUsed += fontInfo.getWidth(size, char); // Sum up the width of each character in the text.
        }
        return widthUsed <= w; // Return true only if the total width is within the constraint.
    };

    // Perform a binary search to find the maximal feasible font size from the sorted array of font sizes.
    let left: number = 0;
    let right: number = fonts.length - 1;

    // Iterate until the search range is exhausted.
    while (left < right) {
        // Calculate the middle index. (right + left + 1) ensures that the mid value leans towards the right side.
        const mid: number = left + Math.floor((right - left + 1) / 2);
        // Use the helper function to check if the guessed font size can fit the text.
        if (canFitTextWithFontSize(fonts[mid])) {
            left = mid; // If it fits, it could be our new minimum feasible font size; search the right side.
        } else {
            right = mid - 1; // If it doesn't fit, search the left side.
        }
    }

    // After binary search, either we have found a feasible font size or none of them are feasible.
    // We perform a final check for the font size at index 'left'.
    return canFitTextWithFontSize(fonts[left]) ? fonts[left] : -1;
};

// Export the maxFont function if it needs to be used in other modules.
export { maxFont };

```

Time and Space Complexity

Time Complexity

The time complexity of this binary search algorithm within a bounded list of font sizes needs to be analyzed based on two main operations:

1. The binary search itself.
2. The check function which is called at each step of the binary search.

Performing binary search on the sorted list of font sizes has a time complexity of `O(log n)`, where `n` is the number of font sizes in the list `fonts`.

The `check` function gets called once for every step of the binary search. Inside the `check` function, there's a sum operation that iterates through every character `c` in the string `text`.

The time taken to get the width and height for a given fontSize is not specified, so we assume it to be `O(1)` for each call to `fontInfo.getWidth` and `fontInfo.getHeight`.

The total number of characters in `text` is denoted as `m`. Therefore, the complexity for checking all characters in the text at any font size is `O(m)`.

For every binary search step, the `check` function is invoked once which would result in the time complexity of `O(m)` for each step of the binary search. Therefore, the combined time complexity for the binary search and the check function is `O(m * log n)`.

Space Complexity

The space complexity of the algorithm is the additional space required outside of the inputs. The algorithm uses a constant amount of space for variables such as `left`, `right`, `mid`, and `ans`. Hence, the space complexity is `O(1)`.

We are not considering the space taken up by the input text, font list, or the `FontInfo` object, as these are not part of the algorithm's own space requirements.