1618. Maximum Font to Fit a Sentence in a Screen String ]

Interactive

# **Problem Description**

Medium Array

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:

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

**Leetcode Link** 

 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.

that particular font size. The function checks two conditions:

all characters in text with getWidth(fontSize, c).

**Binary Search** 

- 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.

size can make the text fit on the screen, we are to return -1. Intuition

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

This problem can naturally be approached with a binary search strategy, since the font sizes are sorted and we need to find the

• Define a function check() that determines if a given fontSize can be used to fit the text within the dimensions of the screen (w

maximum fontSize that fits the constraints. The key steps when applying binary search in this context are:

## and h).

• 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.

Run a binary search over the array of font sizes. At each step, the middle font size is tested using the check() function.

- 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.
- 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

• 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

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.

effectively dividing by 2.

Example Walkthrough

Suppose we have the following scenario:

getWidth(fontSize, ch) returns fontSize \* 1.2 units for each ch.

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

 $\circ$  mid = (0 + 5 + 1) >> 1 evaluates to 3, so fontSize = fonts[3] is 16.

len("LeetCode"). If this sum is less than or equal to 100 units, check(16) returns true.

3. We enter a loop that will continue until left and right converge. Initially, left = 0 and right = 5.

specific font size without exceeding the screen dimensions.

getHeight(fontSize) returns fontSize units.

4. Calculate mid. Let's begin with the first iteration:

return -1, indicating no suitable font size is available.

def is\_valid\_size(font\_size):

return False

left, right = 0, len(fonts) - 1

left = mid + 1

right = mid - 1

 $max_valid_font = -1$ 

else:

int left = 0;

} else {

int right = fonts.length - 1;

left = middle;

right = middle - 1;

int middle = (left + right + 1) / 2;

while (left < right) {</pre>

return max\_valid\_font

# Helper function to check if a given font size is valid

if fontInfo.getHeight(font\_size) > height:

# Initialize the pointers for binary search

# for the text with the given width and height constraints

# Variable to store the maximum valid font size found so far

# Perform a binary search to find the largest valid font size

// Perform a binary search to find the largest valid font size.

if (canFitText(text, fonts[middle], width, height, fontInfo)) {

// If the text doesn't fit, decrease the upper bound.

// If the text fits, move the lower bound of the search up.

// Check if the current font size can fit the text.

# Check if the height of the font is within the allowed height

# 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</pre>

# Move the right pointer to search for a smaller valid font size

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

// Calculate the middle index, leaning to the right in case of even number of elements.

description.

larger font size could also fit.

within the screen limits, and -1 is returned.

**Solution Approach** 

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 +

constraints on the screen. Two scenarios are possible: o 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

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

right + 1) >> 1, which is a bit manipulation technique to calculate the average of left and right and shift one bit to the right,

o 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

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.

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

The solution leverages the sorted nature of the fonts array and the binary search algorithm to efficiently zone in on the maximum

 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:

1. We need to define a helper function check(size). This function will determine whether or not the text can be displayed using a

## 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.

screen dimensions.

1 from typing import List

class Solution:

-1.

9

10

11

12

13

14

15

16

17

18

19

26

27

28

29

30

31

32

33

34

9

10

11

12

13

14

15

16

17

18

19

6. Depending on the result, we adjust our left and right:

7. We then repeat this process until left and right converge on the largest font size that can be used without exceeding the

8. Eventually, suppose left converges on index 2, with check(fonts[2]) returning true and check(fonts[3]) returning false. This

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.

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 \*

- 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
- **Python Solution**

def maxFont(self, text: str, width: int, height: int, fonts: List[int], fontInfo: 'FontInfo') -> int:

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

20 while left <= right:</pre> mid = (left + right) // 221 22 if is\_valid\_size(fonts[mid]): # If the current size is valid, store it as the max found 23 24 max\_valid\_font = fonts[mid] 25 # Move the left pointer to search for a larger valid font size

```
1 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) {
```

**Java Solution** 

```
20
21
22
23
           // After exiting the loop, check if the font at the left index fits.
24
           return canFitText(text, fonts[left], width, height, fontInfo) ? fonts[left] : -1;
25
26
27
       // This helper function checks if `text` can fit within given `width` and `height`
28
       // constraints with the specified font size, using the `fontInfo` API.
29
       private boolean canFitText(String text, int fontSize, int width, int height, FontInfo fontInfo) {
30
           // Check the height of the font first; if it's too tall, return false.
           if (fontInfo.getHeight(fontSize) > height) {
31
32
               return false;
33
34
35
           // Sum the width of each character in the text with the current font size.
36
           int totalWidth = 0;
37
           for (char character : text.toCharArray()) {
               totalWidth += fontInfo.getWidth(fontSize, character);
39
               // If the total width exceeds the allowed width, return false.
               if (totalWidth > width) {
40
                    return false;
41
42
43
44
45
           // If the loop completes without returning, the text fits within the width.
46
           return true;
47
48 }
49
C++ Solution
  1 /**
     * // 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)
 10
```

// Function to find the maximum font size that can be used to fit text into given dimensions (w by h).

// Lambda to check if a given fontSize can be used for the text within the specified dimensions.

// Choose the mid font size. Use right + 1 to prevent infinite loop in case of two elements.

int maxFont(string text, int width, int height, vector<int>& fonts, FontInfo fontInfo) {

// Check if the total width of characters exceeds the allowed width.

// The fontSize is viable since the characters fit within the width.

// Binary search initialization — indices for the range of fonts array.

// Modified binary search to find the biggest viable font size.

// Shift right by 1 is equivalent to integer division by 2.

// If mid font size is viable, move the left boundary to mid.

// Otherwise, eliminate the mid size and all greater sizes.

// After narrowing down, check if the left boundary font size is viable.

return canUseFontSize(fonts[left]) ? fonts[left] : -1;

// If it is, return its size, otherwise return -1 indicating no font size fits.

// If the font height exceeds the allowed height, this font size is not viable.

// The fonts vector contains all possible font sizes in ascending order.

if (fontInfo.getHeight(fontSize) > height) return false;

totalWidth += fontInfo.getWidth(fontSize, c);

// As soon as we exceed width, return false.

if (totalWidth > width) return false;

auto canUseFontSize = [&](int fontSize) {

int left = 0, right = fonts.size() - 1;

int mid = (left + right + 1) / 2;

if (canUseFontSize(fonts[mid])) {

int totalWidth = 0;

return true;

while (left < right) {</pre>

left = mid;

right = mid - 1;

} else {

for (char c : text) {

Typescript Solution

\* };

public:

class Solution {

**}**;

\*/

11

12

14

15

16

17

18

19

20

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

57

56 };

```
1 // Define the interface for FontInfo which outlines the available methods.
  2 interface FontInfo {
         getWidth(fontSize: number, ch: string): number;
        getHeight(fontSize: number): number;
  5 }
  6
    /**
     * 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.
 10
     * @param {string} text The text that needs to be fit into the specified width and height.
 11
    * @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.
    st @return {number} The maximum font size that can be used to fit the text, or -1 if no font size fits.
 17 */
 18 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.
 19
         const canFitTextWithFontSize = (size: number): boolean => {
 20
 21
            if (fontInfo.getHeight(size) > h) {
 22
                 return false; // The height of the font exceeds the max height allowed.
 23
 24
            let widthUsed: number = 0;
 25
            for (const char of text) {
 26
                widthUsed += fontInfo.getWidth(size, char); // Sum up the width of each character in the text.
 27
 28
            return widthUsed <= w; // Return true only if the total width is within the constraint.
        };
 29
 30
 31
        // Perform a binary search to find the maximal feasible font size from the sorted array of font sizes.
 32
        let left: number = 0;
 33
         let right: number = fonts.length - 1;
 34
 35
        // Iterate until the search range is exhausted.
 36
        while (left < right) {</pre>
 37
            // Calculate the middle index. (right + left + 1) ensures that the mid value leans towards the right side.
 38
            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.
 39
            if (canFitTextWithFontSize(fonts[mid])) {
 40
                left = mid; // If it fits, it could be our new minimum feasible font size; search the right side.
 41
 42
            } else {
                right = mid - 1; // If it doesn't fit, search the left side.
 43
 44
 45
 46
 47
        // 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'.
 48
        return canFitTextWithFontSize(fonts[left]) ? fonts[left] : -1;
 49
 50
    };
 51
    // Export the maxFont function if it needs to be used in other modules.
    export { maxFont };
 54
Time and Space Complexity
```

## 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.

operations:

**Time Complexity** 

1. The binary search itself.

fontInfo.getWidth and fontInfo.getHeight.

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 0(1) for each call to

The check function gets called once for every step of the binary search. Inside the check function, there's a sum operation that

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

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).

of space for variables such as left, right, mid, and ans. Hence, the space complexity is 0(1).

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

**Space Complexity** The space complexity of the algorithm is the additional space required outside of the inputs. The algorithm uses a constant amount

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 0(m \* log n).

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.