<https://leetcode.com/problems/text-justification/description/>

Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left-justified, and no extra space is inserted between words.

**Note:**

A word is defined as a character sequence consisting of non-space characters only.

Each word's length is guaranteed to be greater than 0 and not exceed maxWidth.

The input array words contains at least one word.

**Example 1:**

Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16

Output:

[

  "This    is    an",

  "example  of text",

  "justification.  "

]

**Example 2:**

Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16

Output:

[

  "What  must  be",

  "acknowledgment  ",

  "shall be        "

]

Explanation: Note that the last line is "shall be    " instead of "shall    be", because the last line must be left-justified instead of fully-justified.

Note that the second line is also left-justified because it contains only one word.

**Example 3:**

Input: words = ["Science","is","what","we","understand","well","enough","to","explain","to","a","computer.","Art","is","everything","else","we","do"], maxWidth = 20

Output:

[

  "Science  is  what we",

  "understand      well",

  "enough to explain to",

  "a  computer.  Art is",

  "everything  else  we",

  "do                  "

]

**Constraints:**

1 <= words.length <= 300

1 <= words[i].length <= 20

words[i] consists of only English letters and symbols.

1 <= maxWidth <= 100

words[i].length <= maxWidth

**Attempt 1: 2023-08-14**

**Solution 1: String handling in intuitive way (120 min)**

class Solution {

    public List<String> fullJustify(String[] words, int maxWidth) {

        List<String> result = new ArrayList<>();

        int i = 0;

        while(i < words.length) {

            List<String> curLine = getWords(i, words, maxWidth);

            i += curLine.size();

            result.add(justify(curLine, maxWidth, i, words));

        }

        return result;

    }

    private List<String> getWords(int i, String[] words, int maxWidth) {

        List<String> curLine = new ArrayList<String>();

        int curLen = 0;

        while(i < words.length && curLen + words[i].length() <= maxWidth) {

            curLine.add(words[i]);

            curLen += words[i].length() + 1;

            i++;

        }

        return curLine;

    }

    private String justify(List<String> curLine, int maxWidth, int i, String[] words) {

        int baseLen = 0;

        for(String word : curLine) {

            baseLen += word.length() + 1;

        }

        // Remove the extra padding space since last word no need

        // extra padding space during base length calculation

        baseLen -= 1;

        int extraSpaces = maxWidth - baseLen;

        // If line only has one string, then the delimiter-join will do nothing, and we just fill the string with spaces.

        // String.join(delimiter, elements) -> able to handle if only one element, then no

        // delimiter will add

        if(curLine.size() == 1) {

            return String.join(" ", curLine) + " ".repeat(extraSpaces);

        }

        // If we are dealing with the last line which has multiple words, then the delimiter-join will put the mandatory space between each word, and then we fill the string with spaces to meet the length requirement.

        if(i == words.length) {

            return String.join(" ", curLine) + " ".repeat(extraSpaces);

        }

        // Attempt to evenly distribute extra spaces

        int wordSplitCount = curLine.size() - 1;

        int spacesPerWordSplit = extraSpaces / wordSplitCount;

        for(int j = 0; j < wordSplitCount; j++) {

            curLine.set(j, curLine.get(j) + " ".repeat(spacesPerWordSplit));

        }

        // If still remain some extra spaces until evenly distribute extra spaces

        int needsExtraSpace = extraSpaces % wordSplitCount;

        for(int j = 0; j < needsExtraSpace; j++) {

            curLine.set(j, curLine.get(j) + " ");

        }

        return String.join(" ", curLine);

    }

}

**Refer to**

<https://leetcode.com/problems/text-justification/editorial/>

**Overview**

Unlike most problems on LeetCode, this is one that can be solved by just doing exactly what the problem statement is telling us to do.

You don't need any data structures or algorithmic tricks to solve this problem. The point of this problem is to test your ability to quickly write clean code while navigating edge cases.

The solution we will present in this article is only one of many ways to approach this problem.

**Approach:**

**Intuition**

This problem falls into a class of problems that are rare on LeetCode - not algorithmic, but more representative of a real-life task. Another good example of a problem in this class is [Valid Number](https://leetcode.com/problems/valid-number/).

These problems are more annoying than difficult because there are multiple moving parts that sometimes aren't really related to each other. It can be frustrating managing everything without running into errors.

One of the most important principles in software engineering is [modularity](https://en.wikipedia.org/wiki/Modular_programming). We can increase modularity by splitting the overall problem into subtasks and then allocating sections of code (like with a helper function) to accomplish these subtasks independently.

In this problem, a natural approach would be to create one line of length

maxWidth at a time. We can split creating a line into two subtasks:

Determine which words should be on the line.

Take the words from the first task and create a line.

We will look at these subtasks separately.

**Determine which words should be on a line**

Let's create a helper method getWords(int i). It takes an integer i that indicates the index of the first word in words that we are considering. It returns a list of words that will be included on a line. This list should be a subarray of words starting with words[i]. The problem description says to "pack as many words as you can in each line". Therefore, the length of this subarray should be maximized without the length of the line exceeding maxWidth.

On a line, every word except the final one must be followed by a space. Therefore, each word contributes word.length + 1 to the line's length, except for the final word. We want to add as many words as possible without exceeding maxWidth.

Let's use an integer currLength to indicate the current minimum length of our line. To determine if we should add words[i] to the current line, we can check the condition: currLength + words[i].length <= maxWidth. We don't need the + 1 in this condition because words[i] could be the final word, which doesn't need a space after it.

If the condition passes, then we can add words[i] to the current line. We should increment currLength by words[i].length + 1.

You may be thinking: why can we do the + 1 here? What if words[i] is the final word that can fit? Won't that + 1 be inaccurate since we aren't adding a space?

The reason we can do + 1 here is that we aren't actually building the line yet. We are just trying to determine how many words can fit. Think about the above condition currLength + words[i].length <= maxWidth, but with words[i + 1] instead. If we can add words[i + 1], then there must have been a space after words[i].

The + 1 is necessary when we add words[i] to "prepare" the condition check for words[i + 1].

We can use a while loop to add words to the current line until adding another word would violate the condition.

private List<String> getWords(int i, String[] words, int maxWidth) {

    List<String> currentLine = new ArrayList<>();

    int currLength = 0;

    while (i < words.length && currLength + words[i].length() <= maxWidth) {

        currentLine.add(words[i]);

        currLength += words[i].length() + 1;

        i++;

    }

    return currentLine;

}

**Take the words from the first task and create a line**We have a list of words

line. Now, we need to convert it to a string according to the problem description:

The string must have a length of maxWidth. Use extra spaces to reach this length.

The extra spaces should be distributed between the words as evenly as possible.

If the number of extra spaces does not evenly divide between words, the words on the left should have more spaces than the ones on the right.

The final line should only be left justified with only one space between words.

The first step is to figure out how many extra spaces are needed to force the line to have a length of maxWidth.

As we know from before, each word contributes word[i].length + 1 to the line length except the last one, which only contributes word[i].length. The + 1 was due to the mandatory spaces between the words.

We can start by finding this minimum baseLength. Iterate over the words in line and add word.length + 1 for each word. To account for the final word not having a space after it, we can initialize baseLength = -1.

Now that we know baseLength, we can find how many extra spaces we need as extraSpaces = maxWidth - baseLength.

Next, we need to distribute extraSpaces evenly. Let's consider how many extra spaces should go after each word. Because the final word does not have any spaces after it, we set wordCount = line.length - 1. This is the number of words in the line that need spaces after it.

To evenly distribute extraSpaces, we find the number of spaces between each word as spacesPerWord = extraSpaces / wordCount (floor division). The problem says that if extraSpaces does not evenly divide, then the leftover spaces should go to the leftmost words.

We can find how many words on the left need an additional space by finding the remainder as needsExtraSpaces = extraSpaces % wordCount (modulo/remainder operator).

We finally have all the information we need. Now we can add the spaces between the words. First, let's iterate over the needsExtraSpaces leftmost words and add a space " " to each string. Next, we iterate over all words except the last one and add spacesPerWord spaces to each string.

Finally, don't forget about the mandatory space between each word - we can add that by joining the strings in line with " " as a delimiter.

That's the full process for a normal line. But we have two special cases to handle as well. First, the final line is handled separately. Second, if a line contains only one word, it needs to be handled the same as the final line. This is because the only word is also the final word, and the process we described above makes it so that the final word does not have any spaces after it.

Notice that we set wordCount = line.length - 1 to ignore the final word. If a line has only one word, we would have wordCount = 0, and then run into division-by-zero when trying to calculate spacesPerWord.

Before going through the full process, we first check if the line contains only one word or if we are dealing with the final line. We can check if we are dealing with the final line by checking if we have used all the words from the input.

To check for this, we will define our helper function as: createLine(string[] lines, int i). We will pass i which is an integer that we will use to go through the input. If i == words.length, that means we have iterated over all the words in the getWords helper function we designed above, and we know we are dealing with the last line.

To deal with the special case, we can just join line with " " as a delimiter, and then add extraSpaces spaces at the end of the string. If line only has one string, then the delimiter-join will do nothing, and we just fill the string with spaces. If we are dealing with the last line which has multiple words, then the delimiter-join will put the mandatory space between each word, and then we fill the string with spaces to meet the length requirement.

private String createLine(List<String> line, int i, String[] words, int maxWidth) {

    int baseLength = -1;

    for (String word: line) {

        baseLength += word.length() + 1;

    }

    int extraSpaces = maxWidth - baseLength;

    if (line.size() == 1 || i == words.length) {

        return String.join(" ", line) + " ".repeat(extraSpaces);

    }

    int wordCount = line.size() - 1;

    int spacesPerWord = extraSpaces / wordCount;

    int needsExtraSpace = extraSpaces % wordCount;

    for (int j = 0; j < needsExtraSpace; j++) {

        line.set(j, line.get(j) + " ");

    }

    for (int j = 0; j < wordCount; j++) {

        line.set(j, line.get(j) + " ".repeat(spacesPerWord));

    }

    return String.join(" ",  line);

}

**Algorithm**

This brings us to our final solution.

Create two helper methods getWords and createLine that we described above.

Initialize an answer list ans and an integer i to iterate over the input.

Use a while loop to iterate over the input. Each iteration in the while loop will handle one line in the answer.

While i < words.length, do the following steps:

Get the words that should be in the current line as currentLine = getWords(i).

Increment i by currentLine.length.

Create the line by calling createLine(line, i) and add it to ans.

Return ans.

**Implementation**

class Solution {

    public List<String> fullJustify(String[] words, int maxWidth) {

        List<String> ans = new ArrayList<>();

        int i = 0;

        while (i < words.length) {

            List<String> currentLine = getWords(i, words, maxWidth);

            i += currentLine.size();

            ans.add(createLine(currentLine, i, words, maxWidth));

        }

        return ans;

    }

    private List<String> getWords(int i, String[] words, int maxWidth) {

        List<String> currentLine = new ArrayList<>();

        int currLength = 0;

        while (i < words.length && currLength + words[i].length() <= maxWidth) {

            currentLine.add(words[i]);

            currLength += words[i].length() + 1;

            i++;

        }

        return currentLine;

    }

    private String createLine(List<String> line, int i, String[] words, int maxWidth) {

        int baseLength = -1;

        for (String word: line) {

            baseLength += word.length() + 1;

        }

        int extraSpaces = maxWidth - baseLength;

        if (line.size() == 1 || i == words.length) {

            return String.join(" ", line) + " ".repeat(extraSpaces);

        }

        int wordCount = line.size() - 1;

        int spacesPerWord = extraSpaces / wordCount;

        int needsExtraSpace = extraSpaces % wordCount;

        for (int j = 0; j < needsExtraSpace; j++) {

            line.set(j, line.get(j) + " ");

        }

        for (int j = 0; j < wordCount; j++) {

            line.set(j, line.get(j) + " ".repeat(spacesPerWord));

        }

        return String.join(" ",  line);

    }

}

**Complexity Analysis**

Let n be words.length, k be the average length of a word, and m be maxWidth.

Here, we are assuming that you are using immutable strings. A language like C++ has mutable strings and thus the complexity analysis will be slightly different.

Time complexity: O(n⋅k)

**getWords**

The work done in each while loop iteration is O(1). Thus the cost of each call is equal to the number of times the while loop runs in each call. This is amortized throughout the runtime of the algorithm - each index of words can only be iterated over once throughout all calls, so the time complexity of all calls to getWords is O(n).

**createLine**

First, we iterate over the words in line to calculate baseLength. Again, this is amortized over the runtime of the algorithm as each word in the input can only be iterated over once here. Therefore, this loop contributes O(n) over all calls to createLine.

If we are dealing with the special case (one word line or last lane), we create a string of length maxWidth. This costs O(m).

Otherwise, we iterate over the words in line and perform string operations on each. The first for loop which adds the mandatory space costs O(k) per iteration. In the worst-case scenario, we won't have any lines with only one word and the final line has only one word. In this scenario, over the runtime of the algorithm, this for loop will iterate over every word except for the final one, which would cost O(n⋅k).

The second for loop which adds the extra spaces is harder to analyze. At a minimum, each operation will cost O(k). The amount of spaces we add is a function of maxWidth and the number of words in line, as well as the sum of their lengths. One thing is for certain though: on a given call, the strings we create in this for loop cannot exceed maxWidth in length combined. Therefore, we can say that this for loop costs O(m) per call to createLine.

Finally, we join the line with a delimiter, which costs O(m).

Overall, this function contributes O(n⋅k) to the overall runtime, and O(m)O(m)*O*(*m*)per call.

**Main section**

We already determined that all calls to getWords contribute O(n) in total, so we don't have to worry about that.

Each call to createLine costs O(m). We call it in each while loop iteration. The number of while loop iterations is a function of n, k, and m. On average, we can fit m/k words per line. Because we have n words, that implies O(n/m/k)=O(n⋅k/m) iterations. Each iteration costs O(m), so this gives us O(n⋅k).

Summing it all up and canceling constants, we have a time complexity of O(n⋅k)- the sum of the characters in all the words.

Space complexity: O(m)

We don't count the answer as part of the space complexity.

We handle one line at a time and each line has a length of m. The intermediate arrays we use like currentLine hold strings, but the sum of the lengths of these strings cannot exceed m either.