165. Compare Version Numbers

String

of a revision to be either the end of the string or the character '.'.

respectively. Also, determine the lengths of the two versions, m and n.

non-specified revisions were implicitly treated as 0. Hence, return 0.

is no reliance on additional significant space, making the space complexity O(1).

We set i = 0, j = 0, m = length of "1.02" = 4, and <math>n = length of "1.2.1" = 5.

We start a while loop where i < m or j < n; in this case, 0 < 4 or 0 < 5 is true.

• For version1, there is no dot until the end, so parse the next number, getting a = 2.

 \circ For version2, we get b = 2 after parsing until the next dot at j = 4.

 \circ The comparison shows a (2) == b (2), so we move forward.

Two Pointers

Medium

Problem Description

left (the first revision is revision 0, the next is revision 1, etc.). Revision comparison is done based on the integer value of each revision, without considering any leading zeros. If a revision is missing in one of the version numbers, it should be treated as 0. Based on the comparison, if version1 is less than version2, we

by dots, with each number called a revision. The task is to compare the version numbers revision by revision, starting from the

The challenge is to compare two version numbers, version1 and version2. A version number is a sequence of numbers separated

return -1; if version1 is greater than version2, we return 1; and if both version numbers are the same, we return 0. This problem requires careful parsing of the string that represents each version number and a clear understanding of how version

numbers are structured and compared. Intuition

comparing the first revision of each version. If they are equal, we proceed to the next one; if not, we determine the result based on which one is greater.

The intuition behind the solution is to simulate the way we compare version numbers in a real-world scenario. We start by

Translating this into code, we iterate through both string representations of the version numbers version1 and version2 simultaneously. Using two pointers, i for version1 and j for version2, we process each revision separately. We consider the end

values. If we find a difference between a and b, we return -1 if a is smaller; otherwise, we return 1. If a and b are equal, we move on to the next revision. If we reach the end of both strings without finding any differences, we return 0.

For each revision, we parse the number, skipping any leading zeroes, by multiplying the current value by 10 and adding the next

digit. Once we have the integer values a and b for the current revisions of version1 and version2, respectively, we compare these

number of revisions by treating missing revisions as 0. **Solution Approach**

The solution ensures that we only compare integer values of revisions and handles cases where the versions have a different

The implementation of the solution employs a straightforward parsing technique to compare version numbers. Here's a step-bystep walk-through: Initialize pointers and lengths: Start with defining two pointers, i and j, for iterating over version1 and version2

Iterate over the version strings: Use a while loop to continue the iteration as long as either i < m or j < n. This is done to

return 1 if it's the other way around.

a different number of revisions.

handle scenarios where one version string is longer than the other. Parse revisions: Parse the current revision for both versions. This is done in two nested while loops, one for each version. A

current value of a or b by 10 and add the integer value of the current character. This effectively strips leading zeros and

temporary variable (say a for version1 and b for version2) is set to 0. For each digit encountered that is not a dot, multiply the

- converts the string to an integer. Increments the appropriate pointer, i or j, when a digit is read. Compare revisions: Once both revisions are extracted, compare these integer values. If they are not equal, decide the return
- **Move to the next revision**: Increment the pointers i and j to skip the dot and proceed to the next revision. Return 0 if no differences are found: If the loop concludes without returning -1 or 1, this means all revisions were equal or

value based on which one is less. Return -1 if the integer from version1 (a) is less than the integer from version2 (b), and

The algorithm avoids the use of additional storage or complex data structures, opting for a simple linear parsing approach. It leverages the properties of integer arithmetic to process revisions, and pointer arithmetic to move through the version strings. Additionally, by treating non-specified revisions as 0, the algorithm cleverly simplifies the case handling for version numbers with

The use of while-loops and conditional logic is quite efficient, ensuring that each character in the version strings is processed

exactly once, giving the algorithm a time complexity of O(max(N, M)), where N and M are the lengths of the version strings. There

hallmark of many string parsing problems. By focusing on one revision at a time, the solution achieves a balance between clarity

This approach to breaking down the problem, iterating through each character, and avoiding unnecessary complexity is a

version2: "1.2.1" As per the given solution approach:

Let's walk through a small example to illustrate how the solution approach works. We will compare two version numbers:

We begin by parsing the first revision of each:

Parse revisions:

and efficiency.

Example Walkthrough

version1: "1.02"

Initialize pointers and lengths:

Iterate over the version strings:

 \circ For version1, we parse until we encounter a dot. We skip the leading zero, and a = 1. For version2, we do the same and get b = 1.

• We increment i and j to skip the dot, so i = 3 and j = 3. Parse and compare the next revisions:

Move to the next revision:

Move to the next revision:

Compare revisions:

 \circ Increment i and j to skip the dots. i is now m (end of version1), but j = 5 is still within version2. Parse remaining revisions:

Since i has reached the end, a remains 0.

• For version2, b is parsed as b = 1.

Solution Implementation

Python

class Solution:

Since a (1) == b (1), we move forward.

The pointers now point to the dots, so i = 2 and j = 2.

Final comparison and result: ∘ The next comparison is between a (0) and b (1). Since a is less, according to our rule, we return −1.

be -1, indicating that version1 is less than version2.

Length of the version strings

pointer1 += 1

pointer2 += 1

if num1 != num2:

Compare the parsed numbers

return -1 if num1 < num2 else 1

Move past the dot for the next iteration

pointer1, pointer2 = pointer1 + 1, pointer2 + 1

pointer1 = pointer2 = 0

def compareVersion(self, version1: str, version2: str) -> int:

Initialize pointers for each version string

len_version1, len_version2 = len(version1), len(version2)

num1 = num1 * 10 + int(version1[pointer1])

num2 = num2 * 10 + int(version2[pointer2])

Loop until the end of the longest version string is reached while pointer1 < len_version1 or pointer2 < len_version2:</pre> # Initialize numeric values of the current version parts num1 = num2 = 0

Parse the version number from version1 until a dot is found or end is reached

Parse the version number from version2 until a dot is found or end is reached

If they are not equal, determine which one is larger and return -1 or 1 accordingly

while pointer1 < len_version1 and version1[pointer1] != '.':</pre>

while pointer2 < len_version2 and version2[pointer2] != '.':</pre>

Therefore, for the example given version1: "1.02" and version2: "1.2.1", the result of our version number comparison would

```
# If no differences were found, the versions are equal
```

public:

return 0

```
Java
class Solution {
    public int compareVersion(String version1, String version2) {
        int length1 = version1.length(), length2 = version2.length(); // Store the lengths of the version strings
        // Initialize two pointers for traversing the strings
        for (int i = 0, j = 0; (i < length1) || (j < length2); ++i, ++j) {</pre>
            int chunkVersion1 = 0, chunkVersion2 = 0; // Initialize version number chunks
            // Compute the whole chunk for version1 until a dot is encountered or the end of the string
            while (i < length1 && version1.charAt(i) != '.') {</pre>
                // Update the chunk by multiplying by 10 (moving one decimal place)
                // and adding the integer value of the current character
                chunkVersion1 = chunkVersion1 * 10 + (version1.charAt(i) - '0');
                i++; // Move to the next character
            // Compute the whole chunk for version2 until a dot is encountered or the end of the string
            while (j < length2 && version2.charAt(j) != '.') {</pre>
                chunkVersion2 = chunkVersion2 * 10 + (version2.charAt(j) - '0');
                j++; // Move to the next character
            // Compare the extracted chunks from version1 and version2
            if (chunkVersion1 != chunkVersion2) {
                // Return -1 if chunkVersion1 is smaller, 1 if larger
                return chunkVersion1 < chunkVersion2 ? -1 : 1;</pre>
           // If chunks are equal, proceed to the next set of chunks
       // If all chunks have been successfully compared and are equal, return 0
        return 0;
C++
#include <string> // Include necessary header
class Solution {
```

int v1Length = version1.size(), v2Length = version2.size(); // Store the sizes of both version strings

```
return 0;
};
TypeScript
```

if (num1 != num2) {

// Compares two version numbers 'version1' and 'version2'

// Iterate over both version strings

int compareVersion(std::string version1, std::string version2) {

// Parse the next version segment from 'version1'

// Parse the next version segment from 'version2'

while (i < v1Length && version1[i] != '.') {</pre>

while (j < v2Length && version2[j] != '.') {</pre>

++j; // Move to the next character

// If we get to this point, the versions are equal

// Compare the parsed version segments

++i; // Move to the next character

for (int i = 0, j = 0; i < v1Length || j < v2Length; ++i, ++j) {

int num1 = 0, num2 = 0; // Initialize version segment numbers for comparison

num1 = num1 * 10 + (version1[i] - '0'); // Convert char to int and accumulate

num2 = num2 * 10 + (version2[j] - '0'); // Convert char to int and accumulate

return num1 < num2 ? -1 : 1; // Return -1 if 'version1' is smaller, 1 if larger

```
function compareVersion(version1: string, version2: string): number {
      // Split both version strings by the dot (.) to compare them segment by segment.
      let versionArray1: string[] = version1.split('.'),
          versionArray2: string[] = version2.split('.');
      // Iterate through the segments for the maximum length of both version arrays.
      for (let i = 0; i < Math.max(versionArray1.length, versionArray2.length); i++) {</pre>
          // Convert the current segment of each version to a number,
          // using 0 as the default value if the segment is undefined.
          let segment1: number = Number(versionArray1[i] || 0),
              segment2: number = Number(versionArray2[i] || 0);
          // If the current segment of version1 is greater than version2, return 1.
          if (segment1 > segment2) return 1;
          // If the current segment of version1 is less than version2, return -1.
          if (segment1 < segment2) return -1;</pre>
      // If all segments are equal, return 0.
      return 0;
class Solution:
   def compareVersion(self, version1: str, version2: str) -> int:
       # Length of the version strings
        len_version1, len_version2 = len(version1), len(version2)
       # Initialize pointers for each version string
        pointer1 = pointer2 = 0
       # Loop until the end of the longest version string is reached
       while pointer1 < len_version1 or pointer2 < len_version2:</pre>
            # Initialize numeric values of the current version parts
            num1 = num2 = 0
            # Parse the version number from version1 until a dot is found or end is reached
            while pointer1 < len_version1 and version1[pointer1] != '.':</pre>
                num1 = num1 * 10 + int(version1[pointer1])
                pointer1 += 1
            # Parse the version number from version2 until a dot is found or end is reached
```

length of version2. This is because the code uses two while loops that iterate through each character of both version1 and version2 at most once. The inner while loops, which convert the version numbers from string to integer, contribute to the same

pointer2 += 1

if num1 != num2:

Time and Space Complexity

return 0

Compare the parsed numbers

return -1 if num1 < num2 else 1

Move past the dot for the next iteration

pointer1, pointer2 = pointer1 + 1, pointer2 + 1

If no differences were found, the versions are equal

while pointer2 < len_version2 and version2[pointer2] != '.':</pre>

num2 = num2 * 10 + int(version2[pointer2])

overall time complexity because they iterate through each subsection of the versions delimited by the period character '.', still not exceeding the total length of the versions. The space complexity of the code is O(1), since it only uses a fixed number of integer variables and does not allocate any variable-sized data structures dependent on the size of the input.

The time complexity of the given code can be considered to be O(max(M, N)), where M is the length of version1 and N is the

If they are not equal, determine which one is larger and return -1 or 1 accordingly