Previous Next

165. Compare Version Numbers

Feb. 21, 2020 | 12.7K views

Average Rating: 5 (8 votes)

Compare two version numbers *version1* and *version2*.
If `version1 > version2` return `1;` if `version1 < version2` return `-1;` otherwise return `0`.

You may assume that the version strings are non-empty and contain only digits and the `.` character.

The `.` character does not represent a decimal point and is used to separate number sequences.

For instance, `2.5` is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

You may assume the default revision number for each level of a version number to be `0`. For example, version number `3.4` has a revision number of `3` and `4` for its first and second level revision number. Its third and fourth level revision number are both `0`.

Example 1:

```
Input: version1 = "0.1", version2 = "1.1"
Output: -1
```

Example 2:

```
Input: version1 = "1.0.1", version2 = "1"
Output: 1
```

Example 3:

```
Input: version1 = "7.5.2.4", version2 = "7.5.3"
Output: -1
```

Example 4:

```
Input: version1 = "1.01", version2 = "1.001"
Output: 0
Explanation: Ignoring leading zeroes, both "01" and "001" represent the same number "1
```

Example 5:

```
Input: version1 = "1.0", version2 = "1.0.0"
Output: 0
Explanation: The first version number does not have a third level revision number, whi
```

Note:

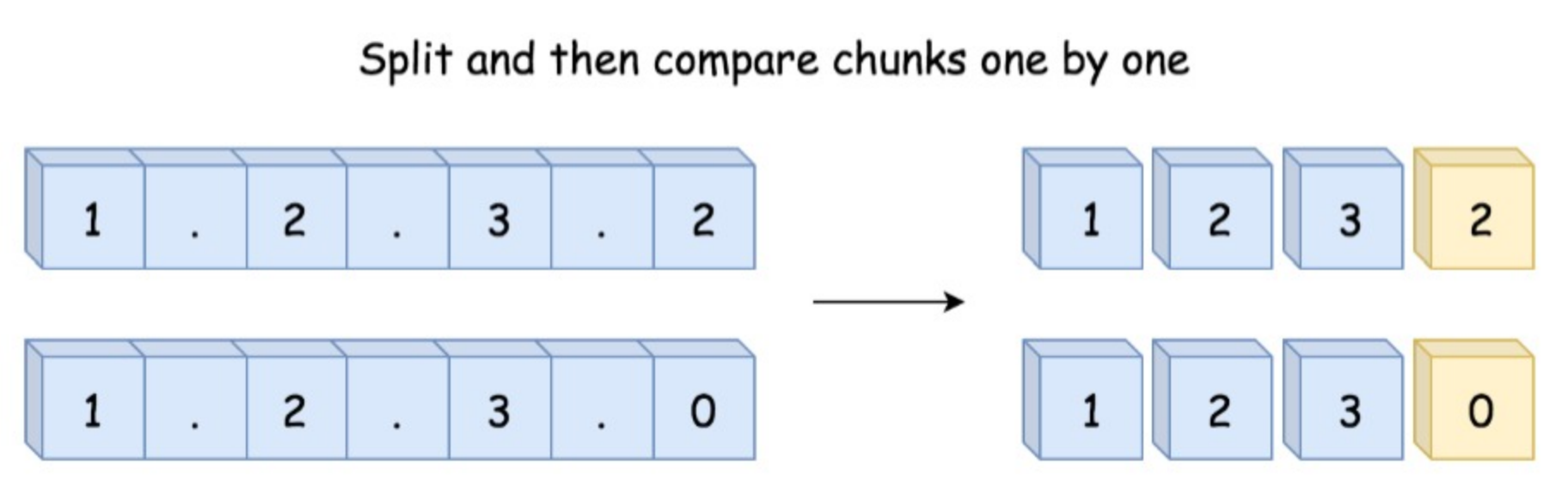
1. Version strings are composed of numeric strings separated by dots `.` and this numeric strings **may** have leading zeroes.
2. Version strings do not start or end with dots, and they will not be two consecutive dots.

Solution

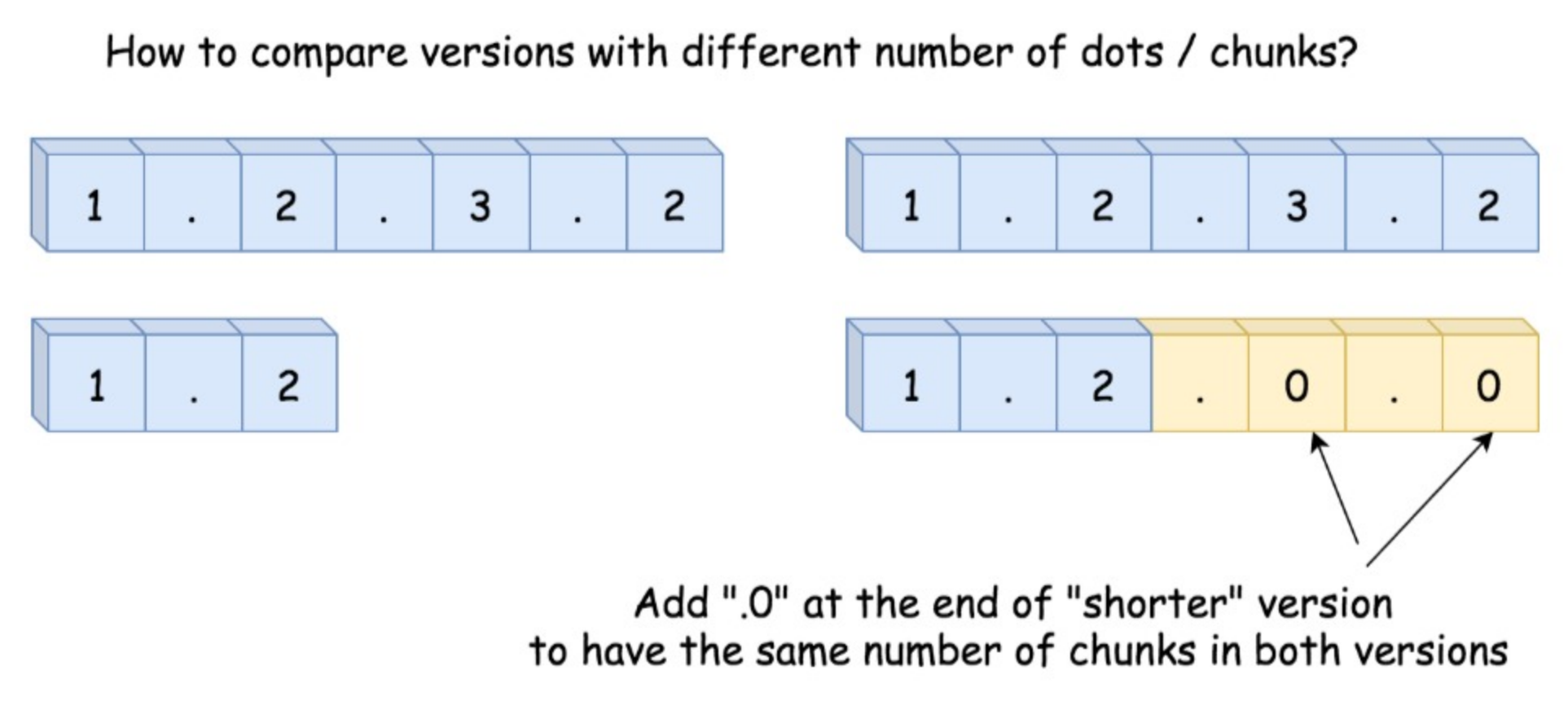
Approach 1: Split + Parse, Two Pass, Linear Space

Intuition

The first idea is to split both strings by dot character into chunks and then compare the chunks one by one.



That works fine if the number of chunks is the same for both versions. If not, we need to pad the shorter string by adding `.0` at the end of the string with less chunks one or several times, so that the number of chunks will be the same.



Algorithm

- Split both strings by dot character into two arrays.
- Iterate over the longest array and compare chunks one by one. If one of the arrays is over, virtually add as many zeros as needed to continue the comparison with the longer array.
  - If two chunks are not equal, return 1 or -1.
- If we're here, the versions are equal. Return 0.

Implementation

Java | Python | Copy

```python
class Solution:
    def compareVersion(self, version1: str, version2: str) -> int:
        nums1 = version1.split('.')
        nums2 = version2.split('.')
        n1, n2 = len(nums1), len(nums2)

        # compare versions
        for i in range(max(n1, n2)):
            i1 = int(nums1[i]) if i < n1 else 0
            i2 = int(nums2[i]) if i < n2 else 0
            if i1 != i2:
                return 1 if i1 > i2 else -1

        # the versions are equal
        return 0
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N + M + \max(N, M))$, where N and M are lengths of input strings.
- Space complexity : $\mathcal{O}(N + M)$ to store arrays `nums1` and `nums2`.

Approach 2: Two Pointers, One Pass, Constant Space

Intuition

Approach 1 has two drawbacks:

- It's two-pass solution.
- It has linear space complexity.

Can we implement a solution in a single pass and a constant space complexity?

> The idea is to use two pointers on each string, to track the beginning and the end of each chunk.

In this way, one could move along both strings in parallel, retrieve and compare corresponding chunks. Once both strings are parsed, the comparison is done as well. The process is done in one pass and a constant space complexity.

Algorithm

First, we define a function named `get_next_chunk(version, n, p)`, which is to retrieve the next chunk in the string. This function takes three arguments: the input string `version`, its length `n`, and a pointer `p` set to the first character of chunk to retrieve. It returns an integer chunk in-between the pointer `p` and the next dot. To help with the iteration, it returns as well a pointer set to the first character of the next chunk.

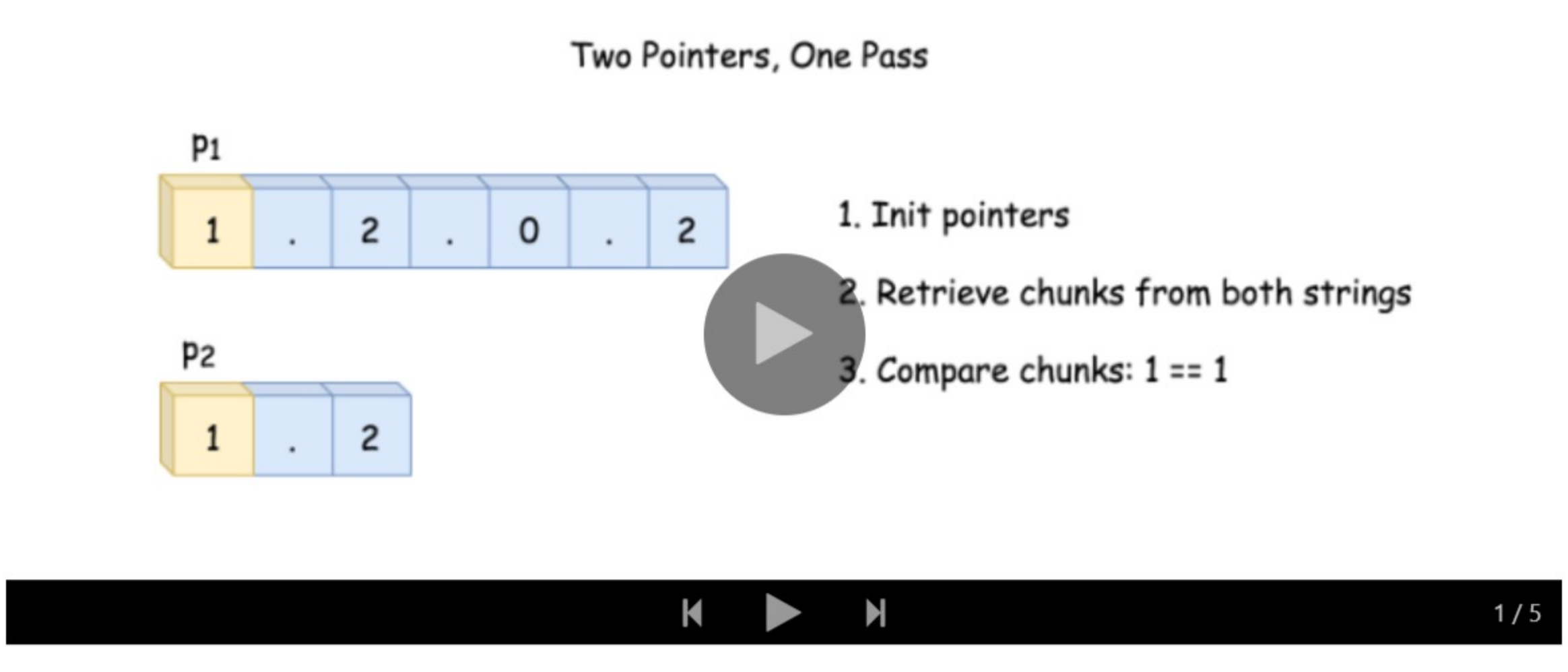
Here is how one could solve the problem using this function:

- Set a pointer `p1` pointed to the beginning of string `version1` and a pointer `p2` to the beginning of string `version2`: `p1 = p2 = 0`.
- Iterate over both strings in parallel. While `p1 < n1 or p2 < n2`:
  - Retrieve the next chunk `i1` from string `version1` and next chunk `i2` from string `version2` using the above-defined `get_next_chunk` function.
  - Compare `i1` and `i2`. If they are not equal, return 1 or -1.
- If we're here, the versions are equal. Return 0.

Now let's implement our `get_next_chunk(version, n, p)` function:

- The beginning of chunk is marked by the pointer `p`. If `p` is set to the end of string, the string is already parsed. To continue the comparison, let's add a virtual `.0` at the end of this string by returning 0.
- If `p` is not at the end of string, move the pointer `p_end` along the string to find the end of chunk.
- Return the chunk `version.substring(p, p_end)`.

Implementation



Java | Python | Copy

```python
class Solution:
    def get_next_chunk(self, version: str, n: int, p: int) -> List[int]:
        # if pointer is set to the end of string
        # return 0
        if p > n - 1:
            return 0, p

        # find the end of chunk
        p_end = p
        while p_end < n and version[p_end] != '.':
            p_end += 1
        # retrieve the chunk
        i = int(version[p:p_end]) if p_end != n - 1 else int(version[p:n])
        # find the beginning of next chunk
        p = p_end + 1

        return i, p

    def compareVersion(self, version1: str, version2: str) -> int:
        p1 = p2 = 0
        n1, n2 = len(version1), len(version2)

        # compare versions
        while p1 < n1 or p2 < n2:
            i1, p1 = self.get_next_chunk(version1, n1, p1)
            i2, p2 = self.get_next_chunk(version2, n2, p2)
            if i1 != i2:
```

Complexity Analysis

- Time complexity : $\mathcal{O}(\max(N, M))$, where N and M are lengths of the input strings respectively. It's a one-pass solution.
- Space complexity : $\mathcal{O}(1)$, since we don't use any additional data structure.

Rate this article:

Previous Next

Comments: 5 Sort By

Type comment here... (Markdown is supported)

Preview Post

pyfjie ★9 March 17, 2020 9:06 AM
Space complexity for approach #2 is wrong (at least for python). The moment you convert a chunk to an int, you are creating a temporary string for that chunk. The size of the chunk can go up to M or N.
4 Share Reply
SHOW 1 REPLY

derreck503 ★104 March 19, 2020 6:59 PM Report
For approach #1, shouldn't the time complexity be O(max(N,M)) and not O(N+M + max(N,M)?

Arn't we only going to do max(N,M) loops of work?
2 Share Reply
SHOW 1 REPLY

parekhmihir ★0 June 24, 2020 8:20 AM
public int compareVersion(String version1, String version2) {
String[] v1 = version1.split("\\.");
String[] v2 = version2.split("\\.");
int maxLen = v1.length >= v2.length?v1.length:v2.length;
int[] v1Int = new int[maxLen];
Read More
0 Share Reply

wildesr ★0 June 12, 2020 8:13 AM Report
C++ solution for approach 2

```
class Solution {
public:
    int compareVersion(string version1, string version2) {
```

Read More
0 Share Reply

littlewang7 ★0 May 3, 2020 6:00 AM
Could anyone explain why use double backslash but not single backslash to escape period (eg. version1.split("\\."))?
0 Share Reply
SHOW 2 REPLIES