93. Restore IP Addresses

Jan. 20, 2019 | 23.7K views
Average Rating: 3.97 (30 votes)

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

A valid IP address consists of exactly four integers (each integer is between 0 and 255) separated by single points.

Example:

```
Input: "25525511135"
Output: ["255.255.11.135", "255.255.111.35"]
```

Solution

Intuition

The naive solution would be to brute-force, *i.e.* to check all possible positions for the dots and keep only the valid ones. In the worst case that means `11` possible positions and hence $11 \times 10 \times 9 = 990$ validations.

That could be optimized with the help of two conceptions.

> The first one is called *constrained programming*.

That basically means to put restrictions after each dot placement.

If one already put a dot that leaves only `3` possibilities for the next dot to be placed : after one digit, after two digits, or after three digits. The first dot has only `3` available slots as well.

That propagates *constraints* and helps to reduce a number of combinations to consider. Instead of 990 combinations it's enough to check just $3 \times 3 \times 3 = 27$.

> The second one called *backtracking*.

Let's imagine that one put one or two dots already and that left no way to place the others to create a valid IP address. What to do? *To backtrack*. That means to come back, to change the position of the previously placed dot and try to proceed again. If that would not work either, *backtrack* again.

Approach 1: Backtracking (DFS)

Here is an algorithm for the backtrack function `backtrack(prev_pos = -1, dots = 3)` which takes position of the previously placed dot `prev_pos` and number of dots to place `dots` as arguments :

- Iterate over three available slots `curr_pos` to place a dot.
- Check if the segment from the previous dot to the current one is valid :
  - Yes :
    - Place the dot.
    - Check if all `3` dots are placed :
      - Yes :
        - Add the solution into the output list.
      - No :
        - Proceed to place next dots `backtrack(curr_pos, dots - 1)`.
    - Remove the last dot to backtrack.

1 / 22

Java | Python | Copy

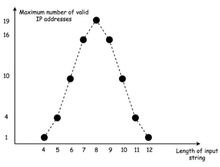
```python
class Solution:
    def restoreIpAddresses(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        def valid(segment):
            """
            Check if the current segment is valid :
            1. less or equal to 255
            2. the first character could be '0'
               only if the segment is equal to '0'
            """
            return int(segment) <= 255 if segment[0] != '0' else len(segment) == 1

        def update_output(curr_pos):
            """
            Append the current list of segments
            to the list of solutions
            """
            segment = s[curr_pos + 1:n]
            if valid(segment):
                segments.append(segment)
                output.append('.'.join(segments))
                segments.pop()

        def backtrack(prev_pos = -1, dots = 3):
```

Complexity Analysis

- Time complexity : as discussed above, there is not more than `27` combinations to check.
- Space complexity : constant space to keep the solutions, not more than `19` valid IP addresses.



Rate this article:

Previous Next

Comments: 9 Sort By

Type comment here... (Markdown is supported)

Preview Post

jianchao-li ★ 14335 March 8, 2019 8:03 AM
The final plot is really interesting.
17 Share Reply

calvinchankf ★ 2917 May 9, 2019 10:33 AM
Home come you made it so hard to understand LOL

btw here is my approach, the basic idea is to generate combinations, quite similar to #91

- slice the candidates with 1 digit, 2 digits, and 3 digits begining from the left

Read More

8 Share Reply

jackson-cmd ★ 21 March 10, 2020 8:39 AM
For python solution, you need a "return output" in the bottom.
5 Share Reply

dyckia ★ 199 December 18, 2019 10:39 PM
Similar idea, but storing the index of dots to be inserted in an ArrayList.

```
class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> res = new ArrayList<>();
```

Read More

2 Share Reply

dnmange ★ 17 March 9, 2019 11:10 AM
can't we use backtracking with memoization to optimize the code further?
2 Share Reply

SHOW 2 REPLIES

kkk20080142 ★ 35 February 11, 2019 2:16 PM
Excellent analysis and code!
1 Share Reply

mars2024 ★ 53 December 28, 2019 3:16 AM
my code is same like in solution but gives time limit exceeded for one test case :

```
class Solution {

    List<String> res = new ArrayList<>();
```

Read More

0 Share Reply

SHOW 1 REPLY

maverick1990 ★ 16 May 2, 2020 11:11 AM
wow what a way to turn a simple solution into complex one. This problem is no different than Decode Ways or Remove Invalid parentheses. You can use backtracking to get to the solution.
-1 Share Reply

rp514 ★ 3 March 2, 2020 5:12 AM
You can do this without backtracking if you generate all the permutations for different string sizes and then just try each one. i.e for 8, it would be all permutations of 2222, 2213, and 1133.
-2 Share Reply