# Solution: Restore IP Addresses

Let's solve the Restore IP Addresses problem using the Backtracking pattern.

We'll cover the following

## Statement

Given that a string, `s`, contains digits, return a list of all possible valid IP addresses that can be obtained from the string.

> **Note:** The order in which IP addresses are placed in the list is not important.

A valid IP address is made up of four numbers separated by dots `.`, for example, $255.255.255.123$. Each number falls between $0$ and $255$ (including $0$ and $255$), and none of them can have leading zeros.

**Constraints:**

- The input string `s` consists of digits only.
- $4 \leq$ `s.length` $\leq 12$

## Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

The naive approach would be to check all possible positions of the dots. To place these dots, we initially have $11$ places, then $10$ places for the second dot, $9$ places for the third dot, and so on. So, in the worst case, we would need to perform $11 \times 10 \times 9 = 990$ validations.

After placing the dots, each resulting substring between the dots must be validated as a valid number between $0$ and $255$ (inclusive) and must not have leading zeros.

Overall, this approach has a time complexity of $O(n^3)$ due to the three nested loops required to place the dots. The space complexity is $O(1)$, because we're not storing any additional data structures.

### Optimized approach using backtracking

To optimize the worst case scenario mentioned in the naive approach, we’ll use backtracking along with constraint checking.

**Constraint checking:** The idea behind this concept is that once we’ve placed a dot, we only have three possible positions for the next dot—after one digit, after two digits, or after three digits.

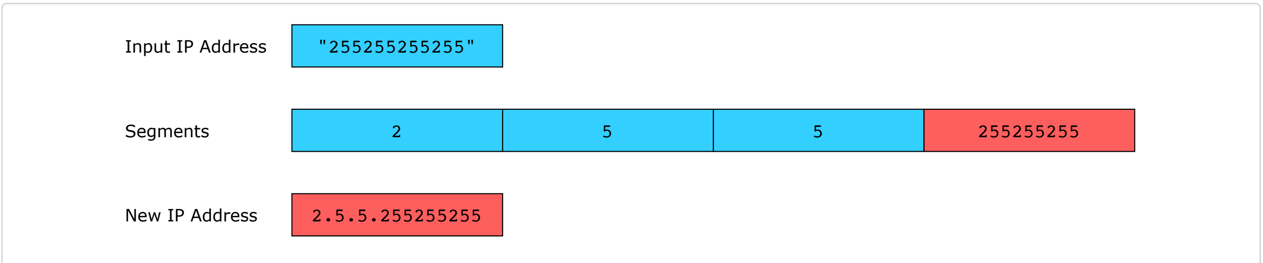
This constraint helps reduce the number of combinations we need to consider. Instead of validating $990$ combinations, we only have to check $3 \times 3 \times 3 = 27$ combinations.

Let’s see how we’ll implement an algorithm that uses the backtracking pattern to find all possible combinations of valid IP addresses.

We will implement the backtrack function that takes the position of the previously placed dot, `prevDot`, and the number of dots, `dots`, to place them as arguments:

- Iterate over the three available slots, `currDot`, to place a dot.
- Check if the current segment from the previous dot to the current one is valid:
  - If yes, place the dot and add the current segment to our `segments` list.
    - Check if all three dots are placed.
      - If yes, concatenate the `segments` list into a string and add the `ip` string to the `result` list.
      - If not, proceed to place the next dots `backtrack(currDot, dots - 1)`.
    - Remove the last dot to backtrack.

The following slides represents the algorithm described above in more detail:



Initially, we've placed all three dots after every single digit. In this way, we get the segments **2**, **5**, **5**, and **255255255**. The value of the last segment is greater than 255. Since it is not a valid IP address segment, we'll backtrack and move the last dot one place forward.

**1** of 23

Let’s look at the code for this solution below.

Java

```java
class RestoreIPAddress {
    static int n;
    static String s;
    static LinkedList <String> segments;
    static ArrayList <String> result;

    public static boolean valid(String segment) {
        int m = segment.length(); // storing the length of each segment
        if (m > 3) // each segment's length should be less than 3
            return false;
        /*
          Check if the current segment is valid
          for either one of following conditions:
          1. Check if the current segment is less or equal to 255.
          2. Check if the length of the segment is 1. The first character of segment
```

```
            can be `0` only if the length of the segment is 1.
        */
        return (segment.charAt(0) != '0') ? (Integer.valueOf(segment) <= 255) : (m == 1);
    }

    // this function will append the current list of segments to the list of results.
    public static void updateSegment(int currDot) {
        String segment = s.substring(currDot + 1, n);
        if (valid(segment)) { // if the segment is acceptable
            segments.add(segment); // add it to the list of segments
            // Utility function to concate the entries of the segments list
            // separated by the dot delimeter.
```

Restore IP Addresses

### Solution summary

Let’s summarize the optimal solution proposed above:

Place each dot after a gap of one digit. Start from the last segment and check whether it is a valid segment. If it is TRUE, then add this segment to the `segments` list and backtrack to place the dot to complete the previous segments. Recursively call the `backtrack` function to make all possible valid IP addresses.

### Time complexity

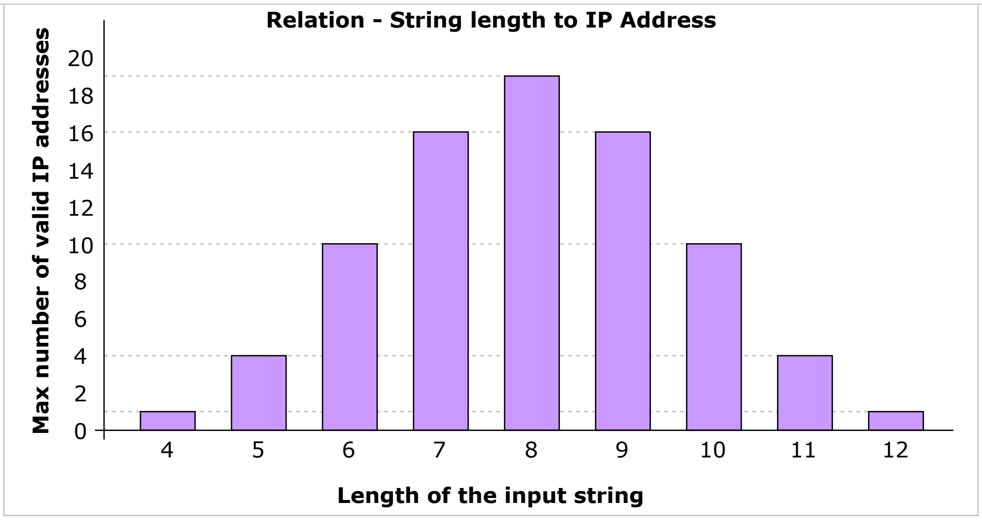
The time complexity is $O(1)$, since we only have to check $3 \times 3 \times 3 = 27$ combinations.

### Space complexity

Space complexity is also constant, since we can have $19$ valid IP addresses at most.

### Additional thoughts

The relationship between the length of the input string and all possible IP addresses that can be generated



The maximum number of IP addresses that can be generated from a string of length $8$ is $19$. Any IP address of a length less than $8$ will have fewer valid IP address combinations.

Back
Restore IP Addresses

Next
Sudoku Solver

✓ Mark as Completed