<https://leetcode.com/problems/restore-ip-addresses/description/>

A **valid IP address** consists of exactly four integers separated by single dots. Each integer is between 0 and 255 (inclusive) and cannot have leading zeros.

For example, "0.1.2.201" and "192.168.1.1" are **valid** IP addresses, but "0.011.255.245", "192.168.1.312" and "192.168@1.1" are invalid IP addresses.

Given a string s containing only digits, return all possible valid IP addresses that can be formed by inserting dots into s. You are **not** allowed to reorder or remove any digits in s. You may return the valid IP addresses in **any** order.

**Example 1:**

**Input:** s = "25525511135"

**Output:** ["255.255.11.135","255.255.111.35"]

**Example 2:**

**Input:** s = "0000"

**Output:** ["0.0.0.0"]

**Example 3:**

**Input:** s = "101023"

**Output:** ["1.0.10.23","1.0.102.3","10.1.0.23","10.10.2.3","101.0.2.3"]

**Constraints:**

1 <= s.length <= 20

s consists of digits only.

**Attempt 1: 2024-07-05**

**Solution 1: Brute Force (30 min)**

class Solution {

    public List<String> restoreIpAddresses(String s) {

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

        int len = s.length();

        for(int i = 1; i < 4 && i < len - 2; i++) {

            for(int j = i + 1; j < i + 4 && j < len - 1; j++) {

                for(int k = j + 1; k < j + 4 && k < len; k++) {

                    String s1 = s.substring(0, i);

                    String s2 = s.substring(i, j);

                    String s3 = s.substring(j, k);

                    String s4 = s.substring(k, len);

                        StringBuilder sb = new StringBuilder();

                        sb.append(s1).append(".").append(s2).append(".").append(s3).append(".").append(s4);

                        result.add(sb.toString());

                    }

                }

            }

        }

        return result;

    }

    public boolean isValid(String s) {

            return true;

        }

        return false;

    }

}

**Refer to**

<https://segmentfault.com/a/1190000003704558>

**四分法**

**复杂度**

时间 O(N^2) 空间 O(1)

**思路**

用三个点将字符串分成四段，验证每一段是否是有效的。我们只要控制这三个分割点就行了，注意约束条件有两个，一个是一段字符串不超过3个字母，另一个是控制好每段字符串最远结束的位置，比如第一个字符串最多延伸到倒数第4个字母。

**注意**

使用Integer.valueOf()时要确保所得到数不会超界。

public class Solution {

public List<String> restoreIpAddresses(String s) {

List<String> res = new LinkedList<String>();

int len = s.length();

// 第一个分割点

for(int i = 1; i < 4 && i < len - 2; i++){

// 第二个分割点

for(int j = i + 1; j < i + 4 && j < len - 1; j++){

// 第三个分割点

for(int k = j + 1; k < j + 4 && k < len ; k++){

String s1 = s.substring(0,i), s2 = s.substring(i, j), s3 = s.substring(j, k), s4 = s.substring(k, s.length());

}

}

}

return res;

}

private boolean isValid(String sub){

}

}

**Solution 2: Backtracking (60 min)**

**Style 1: Process as a whole string with StringBuilder (1ms beats 99.69%)**

class Solution {

public List<String> restoreIpAddresses(String s) {

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

if(s == null || s.length() == 0 || s.length() > 12) {

return result;

}

helper(result, new StringBuilder(), s, 0, 0);

return result;

}

/\*\*

Definition of valid IP address:

1. The length of the ip without '.' should be equal to the length of s;

2. The digit order of ip should be same as the digit order of s;

3. Each part separated by the '.' should not start with '0' except only '0';

4. Each part separared by the '.' should not larger than 255;

5. Valid IP address should contain 3 dots (4 tokens)

\*/

private void helper(List<String> result, StringBuilder sb, String s, int index, int dotCount) {

// Early terminate when '.' more than 3

if(dotCount > 3) {

return;

}

if(index == s.length() && dotCount == 3) {

result.add(sb.toString());

return;

}

// Intelligent part as using two points 'index' and 'i' to control the generation

// of token, each recursive level only generate one token, e.g '255' is a token,

// by the control as below:

// start position end position

// index --> i + 1 (range from (index + 1) to s.length())

// only after this token is valid then go ahead to next recurisve level for next

// token generation, and start position update from index to i + 1

for(int i = index; i < s.length(); i++) {

String token = s.substring(index, i + 1);

if(isValid(token)) {

int len = sb.length();

sb.append(token);

// Tricky point:

// The last position to append dot should before last character

// if reach the last character, we only append last token and no dot

if(i + 1 != s.length()) {

sb.append(".");

helper(result, sb, s, i + 1, dotCount + 1);

} else {

helper(result, sb, s, i + 1, dotCount);

}

sb.setLength(len);

}

}

}

private boolean isValid(String token) {

if(token.length() > 3 || Integer.valueOf(token) > 255

|| (token.length() > 1 && token.charAt(0) == '0')) {

return false;

}

return true;

}

}

Time Complexity: O(1)

Space Complexity: O(1)

**OR we can just move the condition "token.length() > 3" into main for loop which helps on early termination, then no need to wait till isValid() method to check**

class Solution {

    public List<String> restoreIpAddresses(String s) {

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

        if(s == null || s.length() == 0 || s.length() > 12) {

            return result;

        }

        helper(result, new StringBuilder(), s, 0, 0);

        return result;

    }

    /\*\*

     Definition of valid IP address:

     1. The length of the ip without '.' should be equal to the length of s;

     2. The digit order of ip should be same as the digit order of s;

     3. Each part separated by the '.' should not start with '0' except only '0';

     4. Each part separared by the '.' should not larger than 255;

     5. Valid IP address should contain 3 dots (4 tokens)

    \*/

    private void helper(List<String> result, StringBuilder sb, String s, int index, int dotCount) {

        // Early terminate when '.' more than 3

        if(dotCount > 3) {

            return;

        }

        if(index == s.length() && dotCount == 3) {

            result.add(sb.toString());

            return;

        }

        // Intelligent part as using two points 'index' and 'i' to control the generation

        // of token, each recursive level only generate one token, e.g '255' is a token,

        // by the control as below:

        // start position         end position

        //      index       -->     i + 1 (range from (index + 1) to s.length())

        // only after this token is valid then go ahead to next recurisve level for next

        // token generation, and start position update from index to i + 1

        for(int i = index; i < index + 3 && i < s.length(); i++) {

            String token = s.substring(index, i + 1);

            if(isValid(token)) {

                int len = sb.length();

                sb.append(token);

                // Tricky point:

                // The last position to append dot should before last character

                // if reach the last character, we only append last token and no dot

                if(i + 1 != s.length()) {

                    sb.append(".");

                    helper(result, sb, s, i + 1, dotCount + 1);

                } else {

                    helper(result, sb, s, i + 1, dotCount);

                }

                sb.setLength(len);

            }

        }

    }

    private boolean isValid(String token) {

        if(Integer.valueOf(token) > 255 || (token.length() > 1 && token.charAt(0) == '0')) {

            return false;

        }

        return true;

    }

}

Time Complexity: O(1)

Space Complexity: O(1)

**Refer to**

Yet the part of using StringBuilder can be improved a bit. If you keep using the original StringBuilder object and trim its length after each recursion, it would save you some time. Some early termination will also help such as

if(count > 3) {

return;

}

My code as follows:

public List<String> restoreIpAddresses(String s) {

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

if(s == null || s.length() == 0 || s.length() > 12) {

return res;

}

helper(s, res, new StringBuilder(), 0, 0);

return res;

}

private void helper(String s, List<String> res, StringBuilder sb, int pos, int count) {

if(pos == s.length() && count == 3) {

res.add(sb.toString());

return;

}

if(count > 3) {

return;

}

for(int i = pos; i < s.length(); i++) {

String t = s.substring(pos, i+1);

if(t.length() > 3 || t.length() > 1 && t.charAt(0) == '0' || Integer.valueOf(t) > 255) {

break;

}

int len = sb.length();

sb.append(t);

if(i+1 != s.length()) {

sb.append(".");

helper(s, res, sb, i+1, count+1);

} else {

helper(s, res, sb, i+1, count);

}

sb.setLength(len);

}

}

**Style 2: Process as token after token (2ms, slower than Style 1)**

class Solution {

public List<String> restoreIpAddresses(String s) {

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

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

helper(s, result, tokens, 0, 0);

return result;

}

/\*\*

Definition of valid IP address:

1. The length of the ip without '.' should be equal to the length of s;

2. The digit order of ip should be same as the digit order of s;

3. Each part separated by the '.' should not start with '0' except only '0';

4. Each part separared by the '.' should not larger than 255;

5. Valid IP address should contain 3 dots (4 tokens)

\*/

private void helper(String s, List<String> result, List<String> tokens, int index, int tokenCount) {

// Tricky point, the token count should be exactly 4,

// since we construct the first token when token index = 0

// the relation between token index and token count is

// token count = token index + 1

// e.g when we finish the construct of first token and move

// to next recurisve level, the token count will increase

// from 0 to 1, and after construct all 4 tokens and move

// to next recursive level, the expected terminate condition

// is token count = 4

if(index == s.length() && tokenCount == 4) {

result.add(String.join(".", tokens));

return;

}

if(index == s.length() || tokenCount >= 4) {

return;

}

// Build one token each recursive level, and token length range

// is 1 to 3

// Don't miss '=' in 'index + i <= s.length()' since substring

// method exclude the last character, to include it have to '='

for(int i = 1; i <= 3 && index + i <= s.length(); i++) {

String token = s.substring(index, index + i);

if(isValid(token)) {

tokens.add(token);

helper(s, result, tokens, index + i, tokenCount + 1);

tokens.remove(tokens.size() - 1);

}

}

}

private boolean isValid(String token) {

if(Integer.valueOf(token) > 255 || (token.length() > 1 && token.charAt(0) == '0')) {

return false;

}

return true;

}

}

Time Complexity: O(1)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/93>

**Problem Description**

The task is to generate all possible valid IP addresses from a given string containing only digits. A valid IP address has four integer sections separated by dots, and each integer must lie within the range of 0 to 255. Importantly, the integers must not have leading zeros, except for the digit "0" itself. To solve this problem, it is necessary to insert dots in the correct positions within the given string without changing the order of digits to create valid IP addresses. Each section of the IP address is essentially a substring of the original string. The challenge is to determine the partitions that yield a valid sequence of four integers, respecting the constraints stated above.

**Intuition**

To arrive at the solution, we adopt a depth-first search (DFS) strategy, exploring each potential segment step by step and backtracking if a segment turns out to be invalid. The DFS algorithm tries to place the dots in all allowable positions, and upon placing each dot, it checks if the resulting segment is a valid integer between 0 and 255. It is important to note that a leading zero is permissible if and only if the segment is a single zero.

The steps of the algorithm are as follows:

Proceed with the DFS traversal from the first digit of the string.

In each function call, iterate over the possible lengths for the next segment, which could be 1, 2, or 3 digits long.

At each iteration, check if the segment is a valid integer within the IP address criteria (0 to 255 and no leading zeros, unless the integer is 0).

If the segment is valid, add it to a temporary list and recursively call the function with the updated index.

If we reach a point where all four segments are valid and we have used all digits of the string, we add the combination to the result list.

If at any point the segment is invalid or we've got 4 segments but the string is not fully used, we backtrack and try another segment length or configuration.

By exploring each possible dot placement and segment length, and ensuring that each constitutes a valid part of the IP address, we can guarantee that all valid IP addresses formed from the input string are found.

**Solution Approach**

The provided solution uses a depth-first search (DFS) recursive algorithm to explore all possible combinations of the digits in the string that could result in a valid IP address. The following is a detailed explanation of the implementation:

**Recursive Function (DFS):** The dfs(i: int) function is a recursive function that takes one argument, i, which represents the current index in the string s. This function is called recursively to explore all potential valid IP segments starting at index i.

**Base Case and Result Collection:** In the recursive function, there are two base cases. One checks if we are at the end of the string (i >= n) and have exactly four segments in the temporary list (len(t) == 4), which is a condition for a valid IP address. If these conditions are met, we join the segments with dots and append the result to the answer list ans. The other base case handles the situation where reaching the end of the string without having exactly four segments or having four segments but not using all characters, which leads to termination of the current path.

**Checking for Valid Segments:** A helper function check(i: int, j: int) is used to determine if the substring from index i to j (inclusive) constitutes a valid IP segment. The check function accounts for leading zeros and the valid integer range. A leading zero is only valid if it is a singular zero (e.g., "0"). Any segment leading with a "0" and having additional digits (e.g., "01") is invalid.

**Exploration and Backtracking:** The core of the DFS is a for-loop that iterates from the current index i to the minimum between i + 3 and n (the length of the string). This accounts for the fact that IP address segments have at most three digits. If a valid segment (substring) is found using the check function, it's added to the current temporary list t, and the DFS continues from the next index j + 1. After the recursive call returns, the added segment is removed (backtracking), and the loop continues to explore the next possible segment.

**Parallel Variables:** The solution maintains several variables in parallel: n, which is the length of the input string s; ans, the list holding all the valid IP addresses found; and t, which is a temporary list used to store the current segments of the IP address being explored.

**Initiation of DFS:** The process begins by calling dfs(0), which triggers the recursive exploration from the first character in the string.

The DFS pattern used here is crucial not only to generate all possible segmentations but also to ensure efficiency by halting early in paths that cannot lead to valid IP addresses. The use of recursion with backtracking enables the algorithm to explore different segment lengths and combinations effectively.

**Example Walkthrough**

Let's consider a simple example to illustrate how the solution works with the given string "25525511135".

Begin by calling the recursive DFS function from the first character: dfs(0).

In the first level of recursion, start with an empty list t = []. Iterate over string lengths 1-3 starting from the first character s[0] as long as we remain within the string length (dot placement possibilities), here let's try all three:

Try taking "2" as the first segment: t = ["2"], call dfs(1).

Try taking "25" as the first segment: t = ["25"], call dfs(2).

Try taking "255" as the first segment: t = ["255"], call dfs(3).

Following the depth-first search strategy, let's dig into the case where the first segment is "255": dfs(3).

With t = ["255"], proceed with the next character s[3], trying segment lengths of 1-3.

Try taking "2" as the next segment: t = ["255", "2"], call dfs(4).

Try taking "25" as the next segment: t = ["255", "25"], is invalid as "25525" is not a possible IP address. Backtracking occurs, no recursive call is made.

Try taking "255" as the next segment: t = ["255", "255"], which is valid, so proceed with dfs(6).

Now consider t = ["255", "255"]. At dfs(6), try out different lengths again.

Taking "1" as the next segment, t = ["255", "255", "1"], call dfs(7).

Taking "11" as the next segment, t = ["255", "255", "11"], call dfs(8).

Taking "111" as the next segment is not possible here since we would run out of characters for the last segment. No recursive call happens.

Consider the state t = ["255", "255", "1"]. Now call dfs(7) and follow the same process.

Taking "3" as the next segment: t = ["255", "255", "1", "3"] and calling dfs(8).

Taking "35" as the next segment: t = ["255", "255", "1", "35"] is valid but doesn't use all characters, no recursive call is made.

Taking "135" as the final segment is not possible since the start index would be greater than string length. No recursive call is made.

In dfs(8) with t = ["255", "255", "1", "3"], we are at the end of the string, so join t with dots to get "255.255.1.3" and add this to the answer list ans.

Now backtrack to the state t = ["255", "255", "1"] and attempt the next possible recursion which would have been with the segment "35" or "135". Both these were ruled out in step 6.

Continue backtracking to previous recursion levels and trying different segment lengths and combinations in this manner until all possibilities have been exhausted.

By the end of the described example, we will have all valid IP addresses that can be formed from the given string stored in the ans list.

**Solution Implementation**

class Solution {

private int stringLength; // Length of the input string

private String inputString; // The input string representing the digits of the IP address

private List<String> validIPAddresses = new ArrayList<>(); // List to hold the valid IP addresses

private List<String> currentSegment = new ArrayList<>(); // List to hold the current segments of the IP address being constructed

// Public method to restore IP addresses from the given string.

public List<String> restoreIpAddresses(String s) {

stringLength = s.length();

inputString = s;

backtrack(0); // Begin the depth-first search (DFS) from the first character of the string

return validIPAddresses;

}

// Helper method to perform a DFS to build all valid IP addresses.

private void backtrack(int index) {

// Check if we have processed the entire string and we have exactly 4 segments

if (index >= stringLength && currentSegment.size() == 4) {

// Join the segments and add the resulting IP address to the list

validIPAddresses.add(String.join(".", currentSegment));

return;

}

// If we've processed the entire string or have more than 4 segments, backtrack

if (index >= stringLength || currentSegment.size() >= 4) {

return;

}

// Initialize an integer to store the numeric value of current segment

int segmentValue = 0;

// Consider 1 to 3 digit long segments (as an IP segment ranges from 0 to 255)

for (int j = index; j < Math.min(index + 3, stringLength); ++j) {

segmentValue = segmentValue \* 10 + inputString.charAt(j) - '0'; // Convert current segment to integer

// Check for leading zeroes and if segmentValue is greater than 255

if (segmentValue > 255 || (inputString.charAt(index) == '0' && index != j)) {

break; // If any of those checks fail, stop exploring further and backtrack

}

// Add the current segment to our list and continue the search

currentSegment.add(inputString.substring(index, j + 1));

backtrack(j + 1); // Explore further by calling backtrack recursively

currentSegment.remove(currentSegment.size() - 1); // Remove the last added segment to backtrack

}

}

}

**Time and Space Complexity**

The given Python code defines a method to restore possible IP addresses from a string by implementing a depth-first search (DFS) algorithm.

**Time Complexity**

The time complexity of the algorithm can be considered as O(1) in terms of the input string's length, since an IP address consists of 4 parts, and each part can have a maximum of 3 digits. The check function is called at each step of the DFS and runs in O(1) time since it operates on a constant size substring (at most 3 characters).

The DFS function will attempt to place a dot after every 1 to 3 digits, but since IP addresses are fixed length (4 parts of at most 3 digits each), the maximum depth of the recursive call stack will be 4, and there will be at most 3^4 possible combinations to check (3 choices at each of the 4 levels of the decision tree). This results in a total of 81 iterations in the worst case, each taking constant time.

Thus, the overall time complexity is O(1) since the size of the input is not a factor beyond a certain length (the length must be between 4 and 12 for a valid IP address).

**Space Complexity**

The space complexity of the solution mainly depends on the size of the recursive call stack and the space used to store the intermediate and final solutions. As previously mentioned, the recursive call stack will have at most 4 levels due to the nature of IP addresses. Plus, a single path t in the recursion tree is a list that can have at most 4 strings, each up to 3 characters long.

The list ans will contain all the valid IP addresses we find. In the worst-case scenario, every partitioning will lead to a valid IP address, but this is highly unlikely. However, if we consider every single character as a digit and each digit forms a valid part of an IP address, the maximum number of valid IP addresses would be 3^4 (though actually it would be less due to the leading zero and value >255 restrictions).

Hence, the space complexity for the output list is O(1), and the overall space complexity of the algorithm including the recursive call stack and the temporary list t is also O(1), since the problem's constraints limit the input size and, consequently, the recursion depth and output size.

**Two Solutions which one is better (or faster) and why ?**

**From ChatGPT**

Both solutions tackle the problem of generating valid IP addresses from a given string s. While the core logic is similar, they differ in implementation details, which may affect readability, maintainability, and potentially performance. Let's analyze both solutions in terms of these factors.

**Solution 1**

**Pros:**

**Uses a StringBuilder to build the IP address dynamically, which can be more efficient than repeatedly creating new strings.**

Keeps track of the current string length, allowing for efficient backtracking by resetting the StringBuilder length.

**Cons:**

The logic involving the placement of dots (.) can be a bit complex and harder to follow.

Requires careful management of the StringBuilder state, which might be error-prone.

**Solution 2**

**Pros:**

Uses a list of tokens to build the IP address, which can be more intuitive and easier to manage.

Simpler logic for backtracking by adding and removing tokens from the list.

Easier to read and understand, as it follows a straightforward approach of adding tokens and recursively generating combinations.

**Cons:**

**May be slightly less efficient due to the overhead of managing the list of tokens and joining them into a final string.**

**Performance Comparison**

**The performance difference between the two solutions is likely minimal for most inputs. However, Solution 1 might have a slight edge in performance due to the use of StringBuilder, which can be more efficient in string manipulations compared to list operations and string joining.**

**Recommendation**

For better readability and maintainability, **Solution 2** is preferable. It is easier to understand and debug, which can be crucial for more complex problems or future modifications. The performance trade-off is negligible for typical input sizes encountered in this problem.

**Summary**

Readability and Maintainability: Solution 2

Slight Performance Edge: Solution 1

Given these points, unless you are facing performance issues with very large inputs, I would recommend using **Solution 2** for its clarity and ease of understanding.

**Refer to**

<https://leetcode.com/problems/restore-ip-addresses/solutions/381545/compare-two-ways-append-1-char-or-dot-each-time-vs-build-1-token-each-time-of-backtracking/>

**Refer to**

[L282.Expression Add Operators (Ref.L93)](note://WEB007eefded627dafec0263aa4ba0de700)

[L751.IP to CIDR (Ref.L93)](note://WEBdb4af37b124750d094dd93f4ad5961d9)