Previous Next

468. Validate IP Address

Jan. 12, 2020 | 79.2K views

Write a function to check whether an input string is a valid IPv4 address or IPv6 address or neither.

IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots ("."), e.g., `172.16.254.1`;

Besides, leading zeros in the IPv4 is invalid. For example, the address `172.16.254.01` is invalid.

IPv6 addresses are represented as eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons (":"). For example, the address `2001:0db8:85a3:0000:0000:8a2e:0370:7334` is a valid one. Also, we could omit some leading zeros among four hexadecimal digits and some low-case characters in the address to upper-case ones, so `2001:db8:85a3:0:0:8A2E:0370:7334` is also a valid IPv6 address(Omit leading zeros and using upper cases).

However, we don't replace a consecutive group of zero value with a single empty group using two consecutive colons (::) to pursue simplicity. For example, `2001:0db8:85a3::8A2E:0370:7334` is an invalid IPv6 address.

Besides, extra leading zeros in the IPv6 is also invalid. For example, the address `02001:0db8:85a3:0000:0000:8a2e:0370:7334` is invalid.

Example 1:

```
Input: IP = "172.16.254.1"
Output: "IPv4"
Explanation: This is a valid IPv4 address, return "IPv4".
```

Example 2:

```
Input: IP = "2001:0db8:85a3:0:0:8A2E:0370:7334"
Output: "IPv6"
Explanation: This is a valid IPv6 address, return "IPv6".
```

Example 3:

```
Input: IP = "256.256.256.256"
Output: "Neither"
Explanation: This is neither a IPv4 address nor a IPv6 address.
```

Constraints:

- `IP` consists only of English letters, digits and the characters "." and ":".

Solution

Overview

The first idea is to use try/catch construct with built-in facilities: ipaddress lib in Python and InetAddress class in Java.

Note that the code below validates the *real-life* IPv4, and *real-life* IPv6. It will not work for this problem because the problem validates not *real-life* but "*simplified*" versions of IPv4 and IPv6.

Some big companies, for example, Microsoft and Amazon, redefine IPv4 and IPv6 on the interviews for the sake of simplicity. Below one could find an extended discussion about the differences.

Java | Python | Copy

```python
from ipaddress import ip_address, IPv6Address
class Solution:
    def validIPAddress(self, IP: str) -> str:
        try:
            return "IPv6" if type(ip_address(IP)) is IPv6Address else "IPv4"
        except ValueError:
            return "Neither"
```

Note that these facilities both refer to POSIX-compatible `inet-addr()` routine for parsing addresses. That's why they consider chunks with leading zeros not as an error, but as an *octal* representation.

> Components of the dotted address can be specified in decimal, *octal (with a leading 0)*, or hexadecimal, with a leading 0X).

As a result, `01.01.01.012` will be a valid IP address in octal representation, as it should be. To check this behaviour, one can run the command `ping 01.01.01.012` in the console. The address `01.01.01.012` will be considered as the one in octal representation, converted into its decimal representation `1.1.1.10`, therefore the ping command would be executed without errors.

By contrary, problem description directly states that *leading zeros in the IPv4 is invalid*. That's not a real-life case, but probably done for the sake of simplicity. Imho, that makes the problem to be a bit schoolish and less fun. Though let's deal with it anyway, since the problem is very popular recently in Microsoft and Amazon.

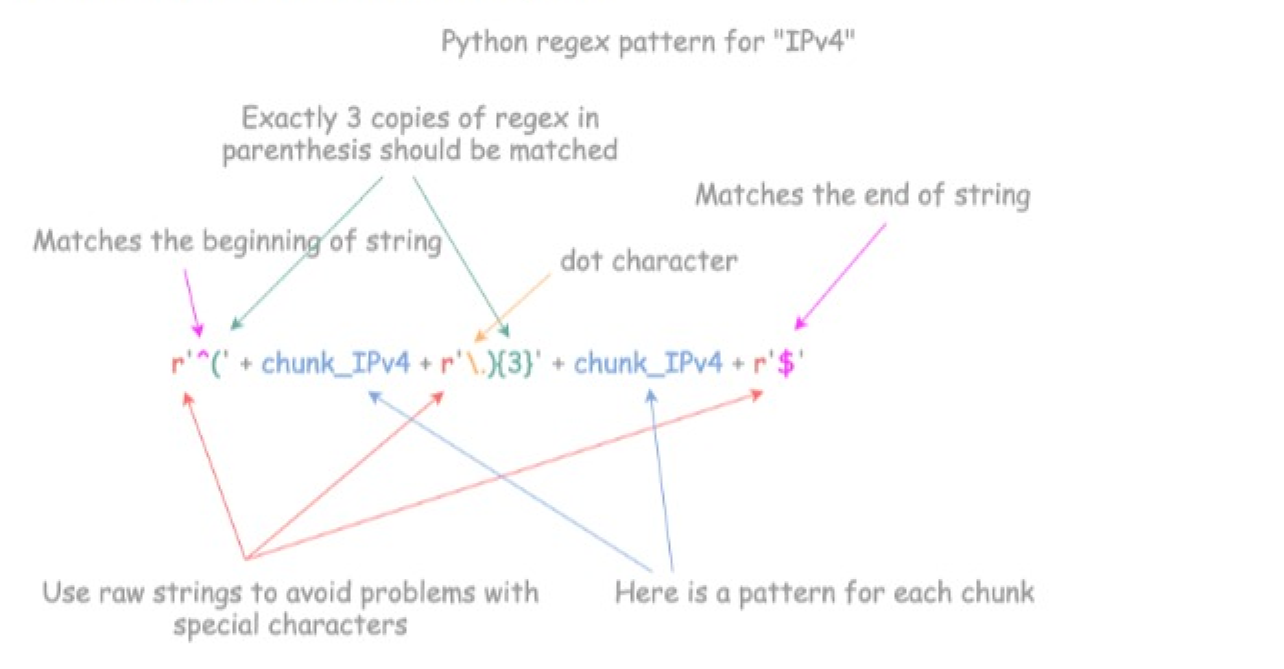
There are three main ways to solve it:

- Regex (*i.e.* regular expression). Less performing one, though it's a good way to demonstrate your knowledge of regex.
- Divide and Conquer, the simplest one.
- Mix of "Divide and Conquer" and "Try/Catch with built-in facilities", this time with ones to convert string to integer. Try/catch in this situation is a sort of "dirty" solution because usually the code inside try blocks is not optimized as it'd otherwise be by the compiler, and it's better not to use it during the interview.

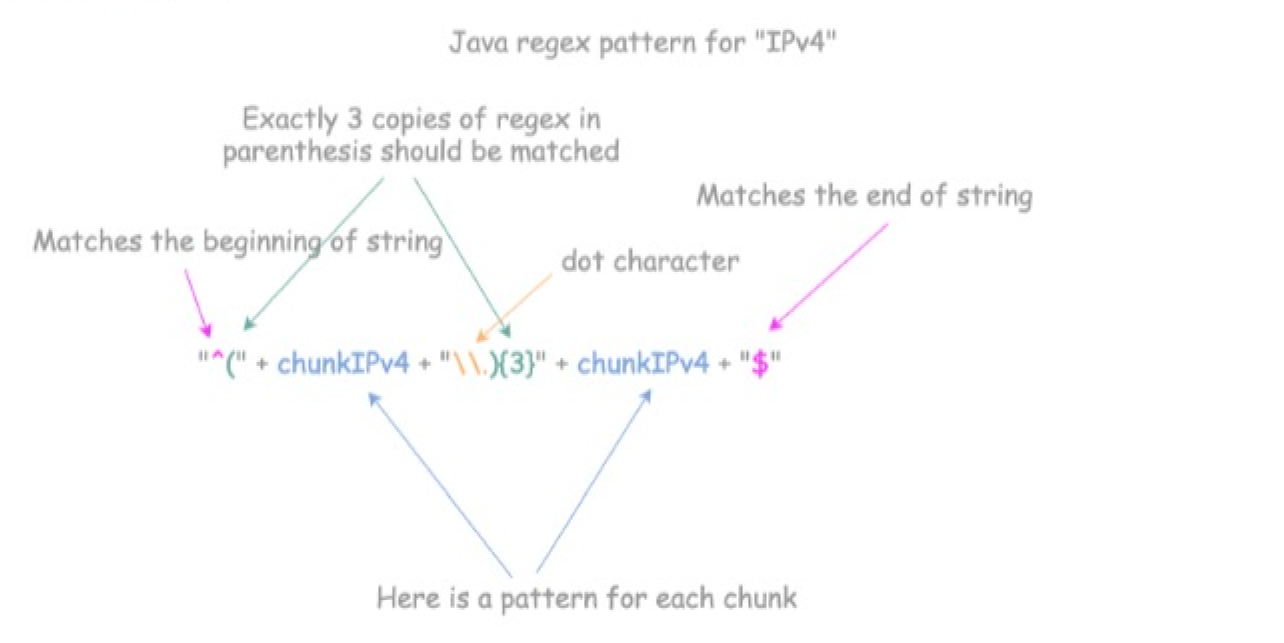
Approach 1: Regex

Let's construct step by step regex for "IPv4" as it's described in the problem description. Note, that it's not a real-life IPv4 because of leading zeros problem as we've discussed above.

Anyway, we start to construct regex pattern by using raw string in Python `r''` and standard string `""` in Java. Here is how its skeleton looks like for Python



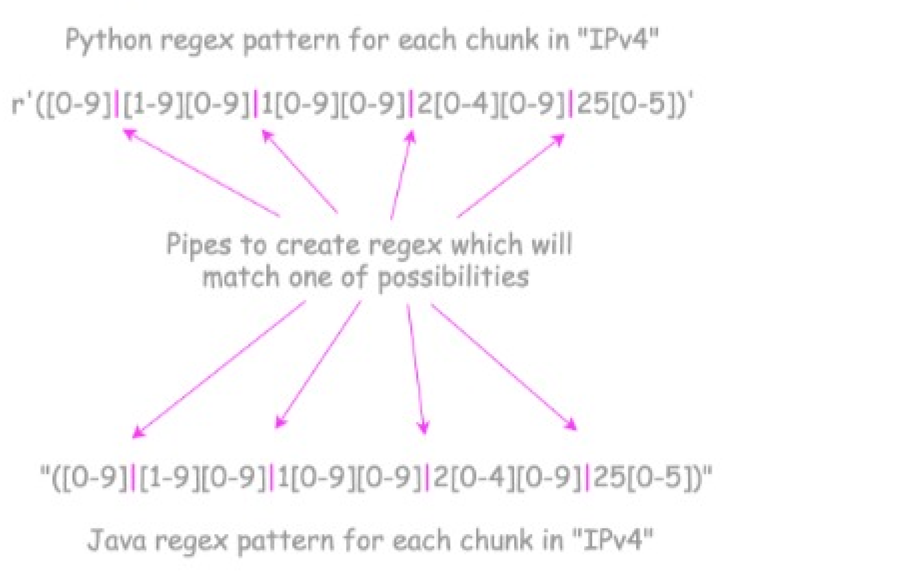
and here is for Java



Now the problem is reduced to the construction of pattern to match each chunk. It's an integer in range (0, 255), and the leading zeros are not allowed. That results in five possible situations:

1. Chunk contains only one digit, from 0 to 9.
2. Chunk contains two digits. The first one could be from 1 to 9, and the second one from 0 to 9.
3. Chunk contains three digits, and the first one is `1`. The second and the third ones could be from 0 to 9.
4. Chunk contains three digits, the first one is `2` and the second one is from 0 to 4. Then the third one could be from 0 to 9.
5. Chunk contains three digits, the first one is `2`, and the second one is `5`. Then the third one could be from 0 to 5.

Let's use pipe to create a regular expression that will match either case 1, or case 2, ..., or case 5.



The job is done. The same logic could be used to construct "IPv6" regex pattern.

Implementation

Java | Python | Copy

```python
import re
class Solution:
    chunk_IPv4 = r'([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])'
    patten_IPv4 = re.compile(r'^(' + chunk_IPv4 + r'\.){3}' + chunk_IPv4 + r'$')

    chunk_IPv6 = r'([0-9a-fA-F]{1,4})'
    patten_IPv6 = re.compile(r'^(' + chunk_IPv6 + r'\:){7}' + chunk_IPv6 + r'$')

    def validIPAddress(self, IP: str) -> str:
        if self.patten_IPv4.match(IP):
            return "IPv4"
        return "IPv6" if self.patten_IPv6.match(IP) else "Neither"
```

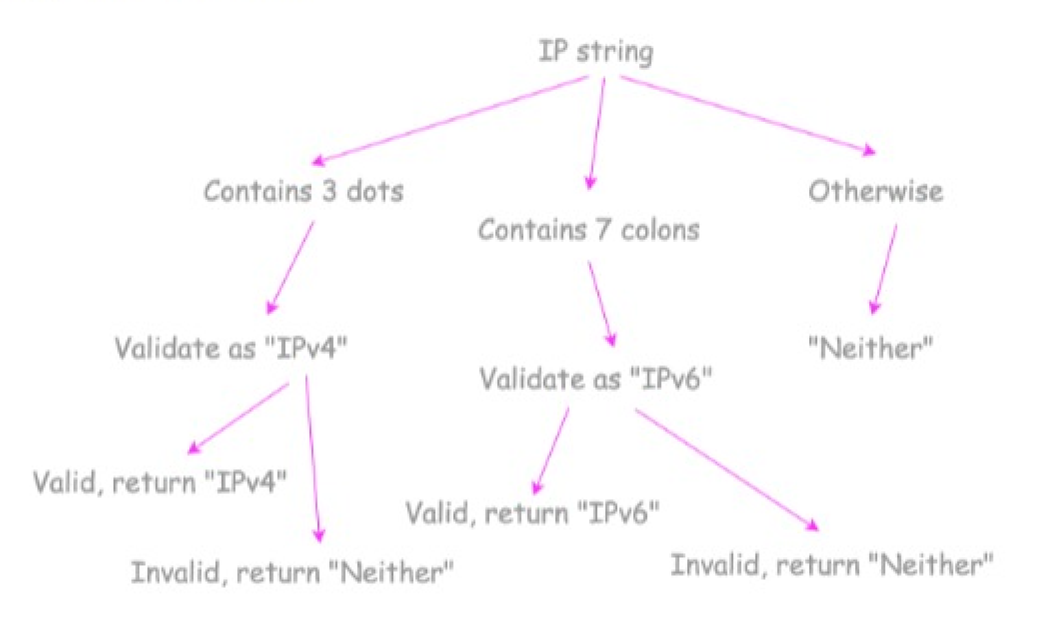
Complexity Analysis

- Time complexity: $\mathcal{O}(1)$ because the patterns to match have constant length.
- Space complexity: $\mathcal{O}(1)$.

Approach 2: Divide and Conquer

Intuition

Both IPv4 and IPv6 addresses are composed of several substrings separated by certain delimiter, and each of the substrings is of the same format.



Therefore, intuitively, we could break down the address into chunks, and then verify them one by one.

The address is valid *if and only if* each of the chunks is valid. We can call this methodology *divide and conquer*.

Algorithm

- For the IPv4 address, we split IP into four chunks by the delimiter `.`, while for IPv6 address, we split IP into eight chunks by the delimiter `:`.
- For each substring of "IPv4" address, we check if it is an integer between `0 - 255`, and there is no leading zeros.
- For each substring of "IPv6" address, we check if it's a hexadecimal number of length `1 - 4`.

Implementation

Java | Python | Copy

```python
class Solution:
    def validate_IPv4(self, IP: str) -> str:
        nums = IP.split('.')
        for x in nums:
            # Validate integer in range (0, 255):
            # 1. length of chunk is between 1 and 3
            if len(x) == 0 or len(x) > 3:
                return "Neither"
            # 2. no extra leading zeros
            # 3. only digits are allowed
            # 4. less than 255
            if x[0] == '0' and len(x) != 1 or not x.isdigit() or int(x) > 255:
                return "Neither"
        return "IPv4"

    def validate_IPv6(self, IP: str) -> str:
        nums = IP.split(':')
        hexdigits = '0123456789abcdefABCDEF'
        for x in nums:
            # Validate hexadecimal in range (0, 2**16):
            # 1. at least one and not more than 4 hexdigits in one chunk
            # 2. only hexdigits are allowed: 0-9, a-f, A-F
            if len(x) == 0 or len(x) > 4 or not all(c in hexdigits for c in x):
                return "Neither"
        return "IPv6"

    def validIPAddress(self, IP: str) -> str:
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$ because to count number of dots requires to parse the entire input string.
- Space complexity: $\mathcal{O}(1)$.

Rate this article:

Previous Next

Comments: 21 Sort By

Type comment here... (Markdown is supported)

Preview Post

xxxffxxx ★21 January 16, 2020 10:40 PM
Technically you can skip zeroes in IPv4 as well. Try it with 127.1 instead of 127.0.0.1
19 Share Reply
SHOW 2 REPLIES

azimbabu ★137 February 23, 2020 3:34 AM
How can regex approach have time complexity O(1)? Won't it be at least O(n) ?
16 Share Reply
SHOW 8 REPLIES

tolinwei ★31 March 1, 2020 9:44 AM
This question is sick, the posted answer using Java `InetAddress` cannot pass test case `"01.01.01.01`
6 Share Reply
SHOW 1 REPLY

SathOkh ★5 June 16, 2020 1:25 PM
The first answer here not only fails on test cases for this task. It also connects to network, so it recognizes "www.welcome.com" as a valid IP4 address (if you have network on) and returns "Neither" for it if network is off.
4 Share Reply
SHOW 1 REPLY

Oyashiro ★3 May 5, 2020 9:47 PM
The description says " extra leading zeros in the IPv6 is also invalid", yet there are VALID strings containing 0370, which in my opinion does contain a leading zero
2 Share Reply
SHOW 4 REPLIES

mashkin ★3 June 16, 2020 1:46 PM
Why space complexity in Divide and conquer approach is O(1)? I'm not sure about java but at least in python solution split operation `nums = IP.split('.')` takes O(N) time and O(N) space.
3 Share Reply
SHOW 4 REPLIES

SJSU153 ★15 January 14, 2020 12:28 PM Report
I have a doubt.. If we are using `IP.contains(".")`
then the complexity of contains method should be O(n). How come the solution is O(1) ??
2 Share Reply
SHOW 1 REPLY

rocky-andre ★12 January 13, 2020 10:24 AM
Nice! and easy.
2 Share Reply

ganesh_1648 ★2 February 1, 2020 1:42 PM
we can write 2001:cdba:0000:0000:0000:0000:3257:9652 to 2001:cdba::3257:9652

see this https://www.ipv6.com/general/ipv6-addressing/
1 Share Reply
SHOW 1 REPLY

tedb19 ★1 June 9, 2020 5:15 AM Report
Is it fair to say the divide and conquer approach is also constant time complexity, because an ip address is usually of a fixed length? The number of operations is always going to be upper bounded by this fixed length.
1 Share Reply
SHOW 2 REPLIES

‹ 1 2 3 ›