468. Validate IP Address (/problems/validate-ip-address/)

Jan. 12, 2020 | 13.7K views

Average Rating: 4.78 (9 votes)

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.

Note: You may assume there is no extra space or special characters in the input string.

Example 1:

```
Input: "172.16.254.1"

Output: "IPv4"

Explanation: This is a valid IPv4 address, return "IPv4".
```

Example 2:

```
Input: "2001:0db8:85a3:0:0:8A2E:0370:7334"

Output: "IPv6"

Explanation: This is a valid IPv6 address, return "IPv6".
```

Example 3:

```
Input: "256.256.256.256"

Output: "Neither"

Explanation: This is neither a IPv4 address nor a IPv6 address.
```

Solution

Overview

The most straightforward way is to use try/catch construct with built-in facilities: ipaddress (https://docs.python.org/3/library/ipaddress.html) lib in Python and InetAddress (https://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html) class in Java.

Java Python Copy

```
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 (https://linux.die.net/man/3/inet_addr) `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.

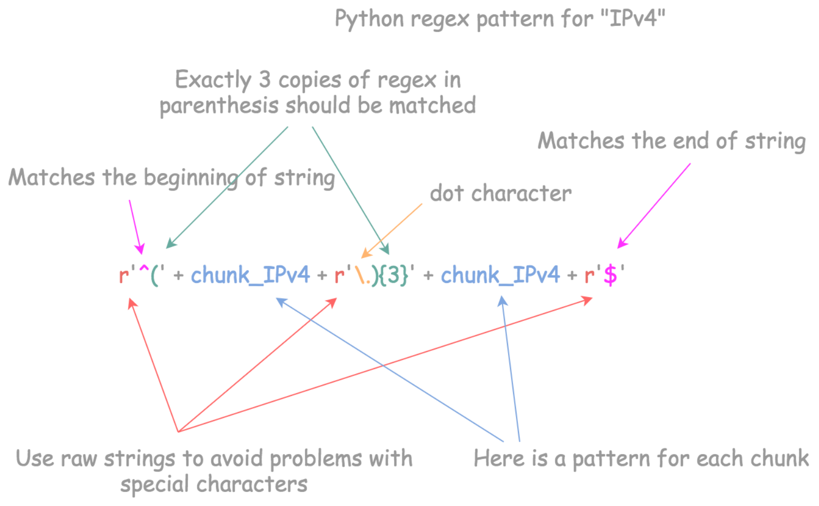
There are three main ways to solve it:

- 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 (https://blogs.msmvps.com/peterritchie/2007/06/22/performance-implications-of-try-catch-finally/), and it's better not to use it during the interview.

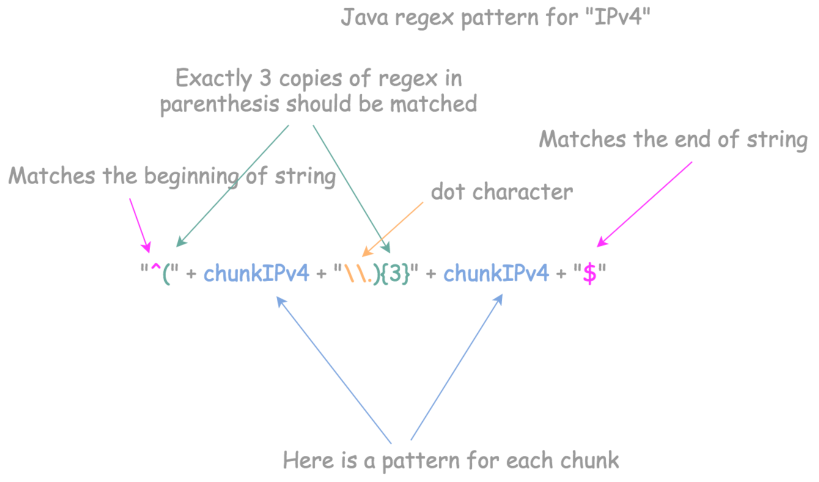
Approach 1: Regex

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


Python regex pattern for "IPv4"
Exactly 3 copies of regex in parenthesis should be matched
Matches the end of string
Matches the beginning of string
dot character
r'^(' + chunk_IPv4 + r'\.){3}' + chunk_IPv4 + r'$'
Use raw strings to avoid problems with special characters
Here is a pattern for each chunk

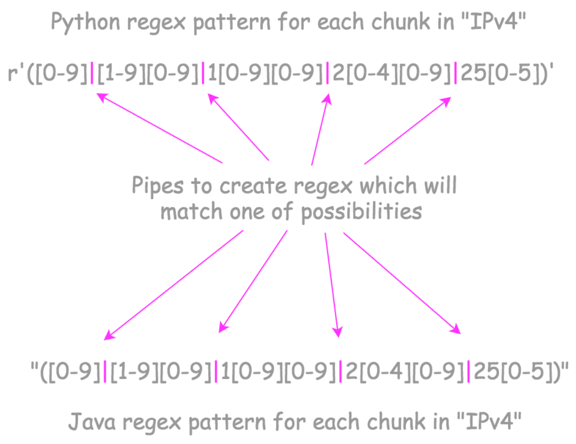
and here is for Java


Java regex pattern for "IPv4"
Exactly 3 copies of regex in parenthesis should be matched
Matches the end of string
Matches the beginning of string
dot character
"^(" + chunkIPv4 + "\\.){3}" + chunkIPv4 + "$"
Here is a pattern for each chunk

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.


Python regex pattern for each chunk in "IPv4"
r'([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])'
Pipes to create regex which will match one of possibilities
"([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])"
Java regex pattern for each chunk in "IPv4"

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"
```

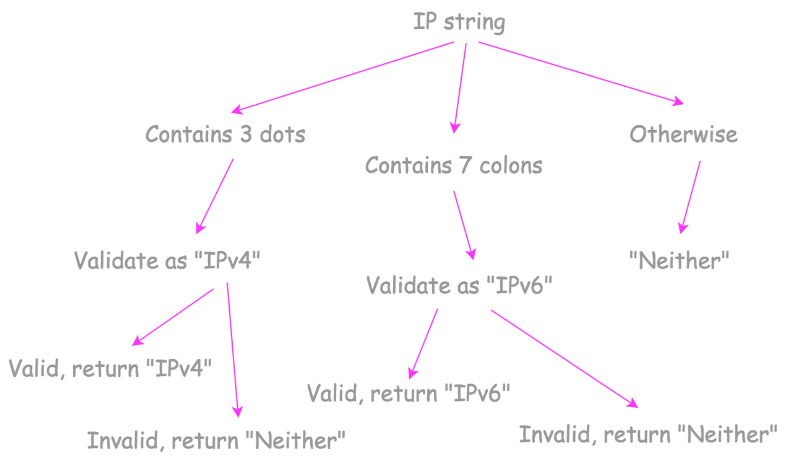
Complexity Analysis

- 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: $\mathcal{O}(1)$.

Analysis written by @liaison (https://leetcode.com/liaison/) and @andvary (https://leetcode.com/andvary/)

Rate this article:

Previous (/articles/design-hashset/)

Next (/articles/serialize-and-deserialize-n-ary-tree/)

Comments: 7

Sort By

Type comment here... (Markdown is supported)

Preview Post

xxxffxxx (/xxxffxxx) 10 January 16, 2020 9:10 AM

Technically you can skip zeroes in IPv4 as well. Try it with 127.1 instead of 127.0.0.1

8 Share Reply

SHOW 1 REPLY

azimbabu (/azimbabu) ★ 102 February 22, 2020 2:04 PM

How can regex approach have time complexity O(1)? Won't it be at least O(n) ?

3 Share Reply

SHOW 4 REPLIES

mehta_vijapur (/mehta_vijapur) ★ 9 January 12, 2020 8:54 PM

Nice! and easy.

3 Share Reply

ganesh_1648 (/ganesh_1648) ★ 1 February 1, 2020 12:12 AM

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/ (https://www.ipv6.com/general/ipv6-addressing/)

1 Share Reply

SHOW 1 REPLY

SJSU153 (/sjsu153) ★ 10 January 13, 2020 10:58 PM

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) ??

1 Share Reply

SHOW 1 REPLY

tolinwei (/tolinwei) ★ 9 February 29, 2020 8:14 PM

This question is sick, the posted answer using Java `InetAddress` cannot pass test case `"01.01.01.01`

0 Share Reply

SHOW 1 REPLY

sam0hack (/sam0hack) ★ 0 February 14, 2020 2:06 PM

The Divide and Conquer method for IPv4 is not always given the correct results.
for E.g if we use validate_IPv4('25.52.55.11.135') It will give us IPv4.
But if we use this method validIPAddress('25.52.55.11.135') with the python ipaddress package it will give us Neither
Which is correct because 25.52.55.11.135 is an invalid Ip address.

0 Share Reply

SHOW 1 REPLY