Problem 751: IP to CIDR

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

An

IP address

is a formatted 32-bit unsigned integer where each group of 8 bits is printed as a decimal number and the dot character

'.'

splits the groups.

For example, the binary number

00001111 10001000 11111111 01101011

(spaces added for clarity) formatted as an IP address would be

"15.136.255.107"

.

A

CIDR block

is a format used to denote a specific set of IP addresses. It is a string consisting of a base IP address, followed by a slash, followed by a prefix length

k

. The addresses it covers are all the IPs whose

first

k

bits

are the same as the base IP address.

For example,

"123.45.67.89/20"

is a CIDR block with a prefix length of

20

. Any IP address whose binary representation matches

01111011 00101101 0100xxxx xxxxxxxx

, where

x

can be either

0

or

1

, is in the set covered by the CIDR block.

You are given a start IP address

ip

and the number of IP addresses we need to cover

n

. Your goal is to use

as few CIDR blocks as possible

to cover all the IP addresses in the

inclusive

range

[ip, ip + n - 1]

exactly

. No other IP addresses outside of the range should be covered.

Return

the

shortest

list of

CIDR blocks

that covers the range of IP addresses. If there are multiple answers, return

any

of them

.

Example 1:

Input:

ip = "255.0.0.7", n = 10

Output:

["255.0.0.7/32","255.0.0.8/29","255.0.0.16/32"]

Explanation:

The IP addresses that need to be covered are: - 255.0.0.7 -> 11111111 00000000 00000000 00000111 - 255.0.0.8 -> 11111111 00000000 00000000 00001000 - 255.0.0.9 -> 11111111 00000000 00000000 00001001 - 255.0.0.10 -> 11111111 00000000 00000000 00001010 - 255.0.0.11 -> 11111111 00000000 00000000 00001011 - 255.0.0.12 -> 11111111 00000000 00000000 00001100 - 255.0.0.13 -> 11111111 00000000 00000000 00001101 - 255.0.0.14 -> 11111111 00000000 00000000 00001110 - 255.0.0.15 -> 11111111 00000000 00000000 00001111 - 255.0.0.16 -> 11111111 00000000 00000000 00010000 The CIDR block "255.0.0.7/32" covers the first address. The CIDR block "255.0.0.8/29" covers the middle 8 addresses (binary format of 11111111 00000000 00000000 00001xxx). The CIDR block "255.0.0.16/32" covers the last address. Note that while the CIDR block "255.0.0.0/28" does cover all the addresses, it also includes addresses outside of the range, so we cannot use it.

Example 2:

Input:

ip = "117.145.102.62", n = 8

Output:

["117.145.102.62/31","117.145.102.64/30","117.145.102.68/31"]

Constraints:

7 <= ip.length <= 15

ip

is a valid

IPv4

on the form

"a.b.c.d"

where

a

,

b

,

c

, and

d

are integers in the range

[0, 255]

.

1 <= n <= 1000

Every implied address

ip + x

(for

x < n

) will be a valid IPv4 address.

Code Snippets

C++:

```
class Solution {
public:
vector<string> ipToCIDR(string ip, int n) {

}
};
```

Java:

```
class Solution {
public List<String> ipToCIDR(String ip, int n) {

}
}
```

Python3:

```
class Solution:
def ipToCIDR(self, ip: str, n: int) -> List[str]:
```

Python:

```
class Solution(object):
def ipToCIDR(self, ip, n):
"""
:type ip: str
:type n: int
:rtype: List[str]
"""
```

JavaScript:

```
/**
* @param {string} ip
* @param {number} n
* @return {string[]}
*/
var ipToCIDR = function(ip, n) {

};
```

TypeScript:

```
function ipToCIDR(ip: string, n: number): string[] {

};
```

C#:

```
public class Solution {
public IList<string> IpToCIDR(string ip, int n) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** ipToCIDR(char* ip, int n, int* returnSize) {

}
```

Go:

```
func ipToCIDR(ip string, n int) []string {

}
```

Kotlin:

```
class Solution {
fun ipToCIDR(ip: String, n: Int): List<String> {

}
}
```

Swift:

```
class Solution {
func ipToCIDR(_ ip: String, _ n: Int) -> [String] {

}
}
```

Rust:

```
impl Solution {
pub fn ip_to_cidr(ip: String, n: i32) -> Vec<String> {

}
}
```

Ruby:

```
# @param {String} ip
# @param {Integer} n
# @return {String[]}
def ip_to_cidr(ip, n)

end
```

PHP:

```
class Solution {

/**
* @param String $ip
* @param Integer $n
* @return String[]
*/
function ipToCIDR($ip, $n) {

}
```

```
}
```

Dart:

```
class Solution {
List<String> ipToCIDR(String ip, int n) {

}
}
```

Scala:

```
object Solution {
def ipToCIDR(ip: String, n: Int): List[String] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec ip_to_cidr(ip :: String.t, n :: integer) :: [String.t]
def ip_to_cidr(ip, n) do

end
end
```

Erlang:

```
-spec ip_to_cidr(Ip :: unicode:unicode_binary(), N :: integer()) ->
[unicode:unicode_binary()].
ip_to_cidr(Ip, N) ->
.
```

Racket:

```
(define/contract (ip-to-cidr ip n)
(-> string? exact-integer? (listof string?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: IP to CIDR
* Difficulty: Medium
* Tags: string
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<string> ipToCIDR(string ip, int n) {

}
};
```

Java Solution:

```
/**
* Problem: IP to CIDR
* Difficulty: Medium
* Tags: string
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public List<String> ipToCIDR(String ip, int n) {

}
}
```

Python3 Solution:

```
"""
Problem: IP to CIDR
```

```
Difficulty: Medium
Tags: string

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def ipToCIDR(self, ip: str, n: int) -> List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def ipToCIDR(self, ip, n):
"""
:type ip: str
:type n: int
:rtype: List[str]
"""
```

JavaScript Solution:

```
/**
* Problem: IP to CIDR
* Difficulty: Medium
* Tags: string
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {string} ip
* @param {number} n
* @return {string[]}
*/
var ipToCIDR = function(ip, n) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: IP to CIDR
* Difficulty: Medium
* Tags: string
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function ipToCIDR(ip: string, n: number): string[] {

};
```

C# Solution:

```
/*
* Problem: IP to CIDR
* Difficulty: Medium
* Tags: string
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public IList<string> IpToCIDR(string ip, int n) {

}
}
```

C Solution:

```
/*
* Problem: IP to CIDR
```

```
* Difficulty: Medium
* Tags: string
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** ipToCIDR(char* ip, int n, int* returnSize) {

}
```

Go Solution:

```
// Problem: IP to CIDR
// Difficulty: Medium
// Tags: string
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func ipToCIDR(ip string, n int) []string {

}
```

Kotlin Solution:

```
class Solution {
fun ipToCIDR(ip: String, n: Int): List<String> {

}
}
```

Swift Solution:

```
class Solution {
func ipToCIDR(_ ip: String, _ n: Int) -> [String] {
```

```
}
}
```

Rust Solution:

```
// Problem: IP to CIDR
// Difficulty: Medium
// Tags: string
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn ip_to_cidr(ip: String, n: i32) -> Vec<String> {

}
}
```

Ruby Solution:

```
# @param {String} ip
# @param {Integer} n
# @return {String[]}
def ip_to_cidr(ip, n)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $ip
* @param Integer $n
* @return String[]
*/
function ipToCIDR($ip, $n) {

}
```

```
}
```

Dart Solution:

```
class Solution {
List<String> ipToCIDR(String ip, int n) {

}
}
```

Scala Solution:

```
object Solution {
def ipToCIDR(ip: String, n: Int): List[String] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec ip_to_cidr(ip :: String.t, n :: integer) :: [String.t]
def ip_to_cidr(ip, n) do

end
end
```

Erlang Solution:

```
-spec ip_to_cidr(Ip :: unicode:unicode_binary(), N :: integer()) ->
[unicode:unicode_binary()].
ip_to_cidr(Ip, N) ->
.
```

Racket Solution:

```
(define/contract (ip-to-cidr ip n)
(-> string? exact-integer? (listof string?))
)
```