<https://grandyang.com/leetcode/751/>

<https://leetcode.ca/all/751.html>

Given a start IP address ip and a number of ips we need to cover n, return a representation of the range as a list (of smallest possible length) of CIDR blocks.

A CIDR block is a string consisting of an IP, followed by a slash, and then the prefix length. For example: “123.45.67.89/20”. That prefix length “20” represents the number of common prefix bits in the specified range.

**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 initial ip address, when converted to binary, looks like this (spaces added for clarity):

255.0.0.7 -> 11111111 00000000 00000000 00000111

The address "255.0.0.7/32" specifies all addresses with a common prefix of 32 bits to the given address,

ie. just this one address.

The address "255.0.0.8/29" specifies all addresses with a common prefix of 29 bits to the given address:

255.0.0.8 -> 11111111 00000000 00000000 00001000

Addresses with common prefix of 29 bits are:

11111111 00000000 00000000 00001000

11111111 00000000 00000000 00001001

11111111 00000000 00000000 00001010

11111111 00000000 00000000 00001011

11111111 00000000 00000000 00001100

11111111 00000000 00000000 00001101

11111111 00000000 00000000 00001110

11111111 00000000 00000000 00001111

The address "255.0.0.16/32" specifies all addresses with a common prefix of 32 bits to the given address,

ie. just 11111111 00000000 00000000 00010000.

In total, the answer specifies the range of 10 ips starting with the address 255.0.0.7 .

There were other representations, such as:

["255.0.0.7/32","255.0.0.8/30", "255.0.0.12/30", "255.0.0.16/32"],

but our answer was the shortest possible.

Also note that a representation beginning with say, "255.0.0.7/30" would be incorrect,

because it includes addresses like 255.0.0.4 = 11111111 00000000 00000000 00000100

that are outside the specified range.

**Note:**

ip will be a valid IPv4 address.

Every implied address ip + x (for x < n) will be a valid IPv4 address.

n will be an integer in the range [1, 1000].

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

**Solution 1: Math + Bit Manipulation (360 min)**

import java.util.ArrayList;

import java.util.List;

class Solution {

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

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

long x = 0;

String[] tokens = ip.split("\\.");

for (String token : tokens) {

x = x \* 256 + Integer.parseInt(token);

}

while (n > 0) {

long blockSize = x & -x;

while (blockSize > n) blockSize /= 2;

res.add(convert(x, (int) blockSize));

x += blockSize;

n -= blockSize;

}

return res;

}

private String convert(long x, int blockSize) {

return String.format("%d.%d.%d.%d/%d",

(x >> 24) & 255,

(x >> 16) & 255,

(x >> 8) & 255,

x & 255,

32 - (int) (Math.log(blockSize) / Math.log(2))

);

}

}

**Why each time after get 'blockSize', we have to add it onto 'x' by 'x += blockSize', what's the logic behind this ?**

while (n > 0) {

long blockSize = x & -x;

while (blockSize > n) blockSize /= 2;

res.add(convert(x, (int) blockSize));

x += blockSize;

n -= blockSize;

}

The logic behind updating x by x += blockSize in the original solution is to correctly advance the IP address range for the next iteration. Let's break it down step by step.

其实有个现象可以和这里的 "correctly advance the IP address range for the next iteration" 遥相呼应:

比如 Input: ip = "255.0.0.7", n = 10
, Output: ["255.0.0.7/32","255.0.0.8/29","255.0.0.16/32"]，结合后面的中文解释里面的一段 "我们的ip地址又不能小于给定的ip地址，所以我们只能将0变为1，而不能将1变为0。所以我们的选择就是要将最低位1后面的0进行变换，比如”255.0.0.8”末尾有3个0，可以变换出8个不同的地址。那么我们只要找出末尾1的位置，就知道能覆盖多少个地址了"

我们发现最后一个CIDR块是7, 那么它的构成就只能在 >= 7的CIDR块里找，比如答案中的7, 8, 16，但是如何正确的获得7, 8, 16呢 ？这就是 x += blockSize 的目的所在，即在保证区段覆盖面不重复的情况下找到下一个区段的开始，比如255.0.0.7 (4278190087) 在找到最低位1的时候即知晓了如果保持255.0.0.7 (4278190087) 不变的话，因为二进制表达的 4278190087 最低位的1本身就是最后一位，那么我们没法修改任何位 (遵循前述的只能将0变为1，而不能将1变为0)，所以255.0.0.7 (4278190087) 所代表的CIDR块自己只能算一个解，即255.0.0.7/32，这只覆盖了1个ip，距离要求的10个ip还差9个，如何找到余下9个呢？必须要在255.0.0.8 (4278190088) 开始的区段寻找，因为在保证区段覆盖面不重复的情况下，算上已经覆盖掉的1个ip，下一个区段的开始就是 4278190087 + 1 (blockSize) = 4278190088，也即255.0.0.8，255.0.0.8 (4278190088) 的最低位1在倒数第4位，即...1000，这样一来就提供了从000到111一共8个可供覆盖的ip，距离9个ip还差1个，如何找到余下1个呢？必须要在255.0.0.16 (4278190096) 开始的区段寻找，因为在保证区段覆盖面不重复的情况下，算上已经覆盖掉的8个ip，下一个区段的开始就是 4278190088 + 8 (blockSize) = 4278190096，也即255.0.0.16，255.0.0.16 (4278190096) 的最低位在倒数第5位，即...10000，这样一来就提供了0000到1111一共16个可供覆盖的ip，但是我们只需要1个了，所以此时我们要人为的将覆盖区块从16个缩小到1个，即通过斜线后面的数字加以限制，这样就255.0.0.16/32后面的32的由来，不同于之前255.0.0.8/29不用人为缩小区块而直接覆盖了 32 - 29 = 3个数位所代表的所有8个ip (即将这3个数位逐个从0变为1共有8个组合)，255.0.0.16/32的32相当于没有数位可以从0变为1，就只有这个数本身。

**Understanding the Process**

Initialization:

x represents the starting IP address in numeric format.

Finding **blockSize**:

blockSize = x & -x calculates the largest power of two that divides x. This helps in determining the largest block size (in terms of CIDR notation) that starts at x.

If step is larger than n, it is adjusted by repeatedly dividing by 2 until it is less than or equal to n.

Converting and Adding to Result:

res.add(convert(x, (int) blockSize)) converts the current x and blockSize to CIDR notation and adds it to the result list.

Updating **x and n**:

x += blockSize moves x forward by the size of the current block. This means the next iteration will start from the next available IP address.

n -= blockSize decreases n by the size of the current block, keeping track of how many IP addresses are left to cover.

**Detailed Explanation with ip = "192.168.0.0" and n = 4**

To illustrate, let's use an example.

Assume the initial IP address ip = "192.168.0.0" and n = 4. The numeric equivalent of 192.168.0.0 is 3232235520.

**Iteration 1:**

Current **x**: 3232235520 (192.168.0.0)

Calculate **blockSize**: x & -x = 3232235520 & -3232235520 = 256 (2^8)

Check **blockSize vs n**: step = 256 is greater than n = 4, so step is repeatedly halved until **blockSize** <= n, resulting in **blockSize** = 4.

Add CIDR block: The block 192.168.0.0/30 (covering IPs 192.168.0.0 - 192.168.0.3) is added to res.

Update **x**: x = x + **blockSize** = 3232235520 + 4 = 3232235524 (next available IP address: 192.168.0.4)

Update **n**: n = n - **blockSize** = 4 - 4 = 0

Since n is now 0, the loop terminates.

**Detailed Explanation with ip = "255.0.0.7" and n = 10**

Initial Setup:

IP: "255.0.0.7" converts to long value 4278190087.

**Iteration 1:**

Current **x**: 4278190087 (255.0.0.7)

Calculate **blockSize**: blockSize = x & -x = 1

Add CIDR block: convert(4278190087, 1) -> "255.0.0.7/32"

Update **x**: x += blockSize = 4278190087 + 1 = 4278190088

Update **n**: n -= blockSize = 10 - 1 = 9

**Iteration 2:**

Current **x**: 4278190088 (255.0.0.8)

Calculate **blockSize**: blockSize = x & -x = 8

Add CIDR block: convert(4278190088, 8) -> "255.0.0.8/29"

Update **x**: x += blockSize = 4278190088 + 8 = 4278190096

Update **n**: n -= blockSize = 9 - 8 = 1

**Iteration 3:**

Current **x**: 4278190096 (255.0.0.16)

Calculate **blockSize**: blockSize = x & -x = 16 (adjusted to 1 because 16 > 1)

Add CIDR block: convert(4278190096, 1) -> "255.0.0.16/32"

Update **x**: x += blockSize = 4278190096 + 1 = 4278190097

Update **n**: n -= blockSize = 1 - 1 = 0

**Final Result**

The resulting CIDR blocks are:

"255.0.0.7/32"

"255.0.0.8/29"

"255.0.0.16/32"

**Why is x += blockSize Necessary?**

Prevent Overlap: By moving x forward by blockSize, we ensure that the next CIDR block starts from the next available IP address. This avoids overlapping ranges and ensures all addresses are covered sequentially.

Correct Range Calculation: Each iteration processes a block of IP addresses. If we didn't update x, subsequent iterations would process the same block repeatedly, which is incorrect.

**Conclusion**

Updating x by x += blockSize is essential to correctly progress through the IP address range. It ensures that each CIDR block covers a unique, non-overlapping portion of the range, ultimately covering the required number of IP addresses (n) starting from the initial IP.

**Explain how number convert to IP method ?**

private String convert(long x, int blockSize) {

return String.format("%d.%d.%d.%d/%d",

(x >> 24) & 255,

(x >> 16) & 255,

(x >> 8) & 255,

x & 255,

32 - (int) (Math.log(blockSize) / Math.log(2))

);

}

Let's break down how the convert method works with the input x representing the IP address "255.0.0.7" and blockSize values derived during the iterations.

**convert Method**

The method converts a long integer representing an IP address and its block size into a CIDR notation string.

**Method Parameters**

x: The IP address in long integer form.

blockSize: The size of the IP address block, given in terms of the number of addresses it covers.

**Explanation with Example**

Let's go through the example with ip = "255.0.0.7" and n = 10.

Initial IP Address and Conversion:

The IP "255.0.0.7" converts to the long integer x = 4278190087.

First Iteration:

blockSize = 1

convert(4278190087, 1):

// Break down the IP address from long to its octets

// (x >> 24) & 255 = 255 (first octet)

// (x >> 16) & 255 = 0 (second octet)

// (x >> 8) & 255 = 0 (third octet)

// x & 255 = 7 (fourth octet)

// Calculate the prefix length

// blockSize = 1, so log2(1) = 0

// 32 - 0 = 32

// Result

return "255.0.0.7/32"

Second Iteration:

blockSize = 8

convert(4278190088, 8):

// Break down the IP address from long to its octets

// (x >> 24) & 255 = 255 (first octet)

// (x >> 16) & 255 = 0 (second octet)

// (x >> 8) & 255 = 0 (third octet)

// x & 255 = 8 (fourth octet)

// Calculate the prefix length

// blockSize = 8, so log2(8) = 3

// 32 - 3 = 29

// Result

return "255.0.0.8/29"

Third Iteration:

blockSize = 1

convert(4278190096, 1):

// Break down the IP address from long to its octets

// (x >> 24) & 255 = 255 (first octet)

// (x >> 16) & 255 = 0 (second octet)

// (x >> 8) & 255 = 0 (third octet)

// x & 255 = 16 (fourth octet)

// Calculate the prefix length

// blockSize = 1, so log2(1) = 0

// 32 - 0 = 32

// Result

return "255.0.0.16/32"

**Conclusion**

The convert method takes a long integer representing an IP address and a block size. It then converts the long integer into its four octets using bitwise operations and shifts. The block size is used to determine the subnet mask length by calculating 32 - log2(blockSize), where log2(blockSize) gives the number of bits required to represent the block size. This results in a CIDR notation string.

**Result for ip = "255.0.0.7" and n = 10**

After running the algorithm with ip = "255.0.0.7" and n = 10, the convert method produces the following CIDR blocks:

"255.0.0.7/32"

"255.0.0.8/29"

"255.0.0.16/32"

**Refer to**

<https://grandyang.com/leetcode/751/>

这道题博主刚开始做的时候，看了半天，读不懂题目的意思，结果一看是一道Easy的题，直接？？？尼克杨问号脸？？？后来通过研究论坛上大家的解法，才总算明白了这道题让我们做什么。此题给了我们一个用字符串表示的ip地址，还有一个整数n，让我们以给定的ip地址为起点，需要覆盖n个ip地址。而这n个ip地址的写法使用无类别域间路由CIDR块来写，所谓的CIDR块，是由一个正常的ip地址，加上斜杠数字，斜杠后面的数字表示这些ip地址具有相同的前缀的个数，比如”255.0.0.7/32”，如果有32个相同的前缀，说明只有唯一的一个ip地址，因为IPv4总共就只有32位。再比如”255.0.0.8/29”，表示有29个相同的前缀，那么最后3位可以自由发挥，2的3次方为8，所以就共有8个ip地址。同理，”255.0.0.16/32”只表示一个地址，那么这三个CIDR块总共覆盖了10个地址，就是我们要求的结果。

由于题目中要求尽可能少的使用CIDR块，那么在n确定的情况下，CIDR块能覆盖的越多越好。根据我们前面的分析，当CIDR块斜杠后面的数字越小，该块覆盖的ip地址越多 (比如前面的例子中 255.0.0.7/32 和 255.0.0.7/29 对比起来CIDR块斜杠后面 29 就比 32 覆盖的 ip 地址多)。那么就是说相同前缀的个数越少越好，但是我们的ip地址又不能小于给定的ip地址，所以我们只能将0变为1，而不能将1变为0。所以我们的选择就是要将最低位1后面的0进行变换，比如”255.0.0.8”末尾有3个0，可以变换出8个不同的地址。那么我们只要找出末尾1的位置，就知道能覆盖多少个地址了。找末尾1有个trick，就是利用 x & -x 来快速找到，这个trick在之前做的题中也有应用 ([Explaining the Binary Indexed Tree](note://4C5F1793BE5C48938FDF8DB06D200AAA))。知道了最多能覆盖地址的数量，还要考虑到n的大小，不能超过n，因为题目只要求覆盖n个。确定了覆盖的个数，我们就可以进行生成CIDR块的操作了，之前我们为了求 x & -x，将 ip 地址转为了一个十进制的数 x，现在我们要把每一块拆分出来，直接按对应位数量进行右移并与上255即可，斜杠后的数字计算通过覆盖的个数进行log2运算，再被32减去即可，参见代码如下：

class Solution {

public:

vector<string> ipToCIDR(string ip, int n) {

vector<string> res;

// 那么我们只要找出末尾1的位置，就知道能覆盖多少个地址了。

// 找末尾1有个trick，就是利用 x & -x 来快速找到

// 我们为了求 x & -x，将ip地址转为了一个十进制的数 x

long x = 0;

istringstream is(ip);

string t;

while (getline(is, t, '.')) {

x = x \* 256 + stoi(t);

}

while (n > 0) {

long step = x & -x;

while (step > n) step /= 2;

res.push\_back(convert(x, step));

x += step;

n -= step;

}

return res;

}

string convert(long x, int step) {

return to\_string((x >> 24) & 255) + "." + to\_string((x >> 16) & 255) + "." + to\_string((x >> 8) & 255) + "." + to\_string(x & 255) + "/" + to\_string(32 - (int)log2(step));

}

};

**The equivalent Java solution from chatGPT**

import java.util.ArrayList;

import java.util.List;

class Solution {

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

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

long x = 0;

String[] tokens = ip.split("\\.");

for (String token : tokens) {

x = x \* 256 + Integer.parseInt(token);

}

while (n > 0) {

long step = x & -x;

while (step > n) step /= 2;

res.add(convert(x, (int) step));

x += step;

n -= step;

}

return res;

}

private String convert(long x, int step) {

return String.format("%d.%d.%d.%d/%d",

(x >> 24) & 255,

(x >> 16) & 255,

(x >> 8) & 255,

x & 255,

32 - (int) (Math.log(step) / Math.log(2))

);

}

}

**Refer to**

<https://algo.monster/liteproblems/751>

**Problem Explanation**

Given a valid IPv4 address and an integer count 'n', we are supposed to generate 'n' IP addresses in the shortest possible blocks that will cover the desired range. These are represented in CIDR blocks. Classless Inter-Domain Routing (CIDR) is a method for allocating IP addresses and routing IP packets.

CIDR notation is a compact representation of an IP address and its associated routing prefix. For instance, "192.0.2.0/24" represents the IPv4 address 192.0.2.0 and its associated routing prefix 192.0.2.0, or equivalently, its subnet mask 255.255.255.0, which has 24 leading 1-bits. This notation describes the operation of a network segment.

The provided solution uses bit manipulation to solve this problem. The steps involved are:

Convert the given IPv4 to binary format.

Loop until count 'n' > 0 :

Initialize the **lowbit** and count.

Get the smallest possible prefix and count that accommodates the required range.

Once the counting rule is figured out, push the CIDR address to the result.

Update the total count and address which is left for iteration.

**NOTE: Here, lowbit is a term for the operation of obtaining the smallest possible prefix of an address, which can be calculated by the operation of (x & -x).**

Let's walk through an example:

For example, if the input is "255.0.0.7" and n=10, the output will be ["255.0.0.7/32","255.0.0.8/29","255.0.0.16/32"]. This is the shortest possible block format in CIDR notation for 10 IP addresses starting from "255.0.0.7".

**Java Solution**

We convert current IP address to 10-based integer, then while 'n' is not '0' left shift for 1 and check if new number is smaller than 'n'. When we have found the biggest possible 2^k that satisfies the requirement, we add it to our result list. The important part is to convert the original IP address into an integer, then increase it by every iteration till we cover all 'n' IP addresses. We compute the smallest possible number of IPs that includes the first IP and ends with 0, then find how much we can give IP addresses with unit '1'.

class Solution {

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

long x = 0;

String[] ips = ip.split("\\.");

for (int i = 0; i < ips.length; i++) {

x = x \* 256 + Integer.parseInt(ips[i]);

}

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

while (n > 0) {

int max = Math.max(33 - Integer.numberOfLeadingZeros(n), 33 - Long.numberOfTrailingZeros(x));

long count = Math.min(1L << (32 - max + 1), n);

result.add(longToIP(x, max));

x += count;

n -= count;

}

return result;

}

//Convert the long format ip address to the standard format

private String longToIP(long x, int m) {

int[] ans = new int[4];

for (int i = 0; i < 4; i++) {

ans[3 - i] = (int) x & 255;

x = x >> 8;

}

return ans[0] + "." + ans[1] + "." + ans[2] + "." + ans[3] + "/" + m;

}

}

**Conclusion**

The problem of generating IP addresses in the shortest possible CIDR blocks is actually a binary manipulation problem in its essence. By converting the IP addresses to integers and performing bitwise operations, we are able to solve this problem efficiently in multiple programing languages including Python, JavaScript, Java, C++ and C#.

The key takeaway here is the understanding of CIDR notation and how IP addresses can be efficiently manipulated by converting them to integers and using bitwise operations. Once grasped, this concept can prove helpful in solving other similar problems as well.

It's important to note that different programming languages have different ways of implementing bit manipulation and hence the syntax and exact way of solving these problems might look different across the languages. However, the underlying concept remains the same in all cases thus make sure to focus on that.

It is important to understand these low-level details as they can greatly improve your problem solving and debugging skills in many types of programming tasks. They can also be helpful in interview situations where knowledge about how things work under the hood can set you apart from other candidates.

All in all, bit manipulation is a powerful tool and CIDR blocks are a unique application of it. Any developer who understands these will find themselves better equipped to handle many networking-related programming tasks.

**Refer to**

[L93.Restore IP Addresses (Ref.L282,L751)](note://WEBa03a392cd2882182805c026c8844ef29)

[Explaining the Binary Indexed Tree](note://4C5F1793BE5C48938FDF8DB06D200AAA)

**The Binary Indexed Tree build process will use i += i & -i for finding least significant bit**