751. IP to CIDR

[Leetcode Link]

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:

1. Convert the given IPv4 to binary format.
2. 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".

Python Solution

Note To write a python solution, 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.

```
python
class Solution:
    def ipToCIDR(self, ip: str, n: int) -> List[str]:
        def ip_to_int(ip):
            ans = 0
            for x in ip.split('.'):
                ans = 256 * ans + int(x)
            return ans

        def int_to_ip(x):
            return ".".join(str((x >> i) % 256) for i in (24, 16, 8, 0))


        def get_max(x):
            if x == 0: return 32
            p = 0
            while x < 2 ** 32:
                x <<= 1
                p += 1
            return p

        ans = []
        x = ip_to_int(ip)
        while n:
            max = get_max(x)
            while 2 ** (max - 1) > n:
                max -= 1
            bx = bin(x)[2:].zfill(32)
            nx = bx[:32 - max + 1] + '0' * (max - 1)
            ans.append(int_to_ip(int(nx, 2)) + '/' + str(max))
            x += 2 ** (max - 1)
            n -= 2 ** (max - 1)
        return ans
```

Java Solution

```
Java
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;
  }
}
```

Javascript Solution

```
javascript
var ipToCIDR = function(ip, n) {
    let ips = ip.split('.').map(Number);
    let x = ips[0] * 256 ** 3 + ips[1] * 256 ** 2 + ips[2] * 256 + ips[3];
    let res = [];
    while(n > 0){
        let mask = Math.max(33 - (n).toString(2).length, 33 - (x & -x).toString(2).length);
        res.push(intToIp(x, mask));
        x += 2**(32-mask);
        n -= 2**(32-mask);
    }
    return res;

    function intToIp(x, mask){
        return [(x >>> 24) & 255,(x >>> 16) & 255,(x >>> 8) & 255,x & 255].join('.') + `/${mask}`;
    }
};
```

C++ Solution

```
cpp
class Solution {
public:
    vector<string> ipToCIDR(string ip, int range) {
        vector<string> result;
        unsigned int cur = ipToInt(ip);

        while(range) {
            int leadingZeros = std::min(__builtin_ctz(cur), __builtin_ctz(range)); // find the leading zeros
            result.push_back(ipStr(cur, leadingZeros));
            range -= (1 << leadingZeros);
            cur += (1 << leadingZeros);
        }

        return result;
    }

    unsigned int ipToInt(string ip) {
        unsigned int result = 0;

        for(int i = 0, start = 0; i < 4; i++) {
            int pos = ip.find('.', start);
            result = (result << 8) + stoi(ip.substr(start, pos-start));
            start = pos+1;
        }

        return result;
    }

    string ipStr(unsigned int ip, int lz) {
        return to_string(ip>>24)      + "." +
                    to_string((ip>>16&255) + "." +
                    to_string((ip>>8&255)  + "." +
                    to_string((ip&255)     + "/" +
                    to_string(32-lz);
    }
};
```

C# Solution

```
csharp
public class Solution {
    public IList<string> IpToCIDR(string ip, int range)
    {
        long x = 0;
        foreach (var part in ip.Split('.'))
        {
            x = x * 256 + int.Parse(part);
        }

        var ans = new List<string>();
        while (range > 0)
        {
            long maxsize = x & -x,
            maxval = range > 32
                ? 1 << (31 - (int)Math.Log(maxsize) / (int)Math.Log(2))
                : Math.Min(maxsize, range);

            ans.Add(longToStringIP(x, (int)Math.Log(32)/ (int)Math.Log(2)- (int)Math.Log(maxval)/ (int)Math.Log(2) + 1));
            x += maxval;
            range -= (int)maxval;
        }

        return ans;
    }

    string longToStringIP(long x, int step)
    {
        int[] ans = new int[4];
        ans[0] = (int) (x & 255);
        x >>= 8;
        ans[1] = (int) (x & 255);
        x >>= 8;
        ans[2] = (int) (x & 255);
        x >>= 8;
        ans[3] = (int) x;
        var ans_str = string.Join(".", ans.Reverse());
        return ans_str + "/" + step;
    }
}
```

In both of these solutions, 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'.## 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.