Practical 5 – Computer Networks Lab

Name: Neeraj Belsare

Roll No.: 79

Batch: A4

PRN: 202101040133

Title:

Subnet Calculator

Aim:

Write a program to implement subnet calculator.

Theory:

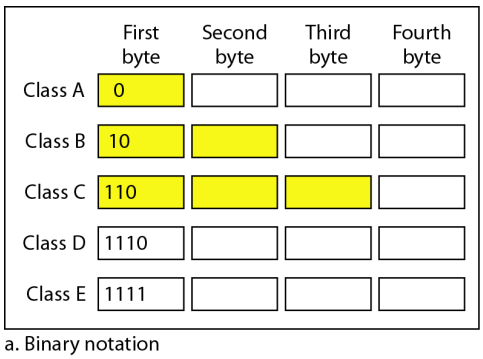
CLASSFUL ADDRESSING:

1. CLASSES:

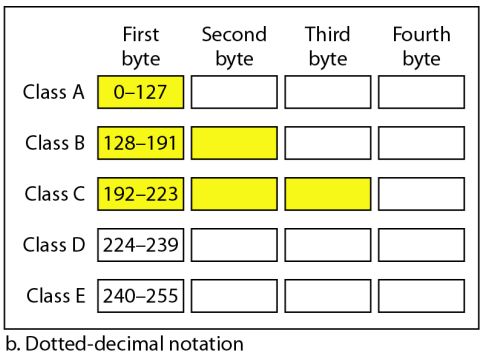
In classful addressing, the address space is divided into five classes: A, B, C, D, and E. The IP address in class A, B, or C is divided into netid and hostid as shown in the figure. We can find the class of an address:

1) If the address is given in binary notation, the first few bits tell the class of the address.

2) If the address is given in decimal-dotted notn. the first byte tells the class.



a. Binary notation



b. Dotted-decimal notation

2. BLOCKS:

Each class is divided into a fixed number of blocks; which have a fixed size.

3. MASK:

Mask is a 32-bit number made up of contiguous n no. of 1s followed by contiguous 32-n no. of 0s. Mask /n means the n leftmost bits in the mask are 1s and the 32 - n rightmost bits are 0s.

In classful addressing, the default mask (n) can be 8, 16, or 24 for class A, B, C respectively. The concept does not apply to classes D and E.

In classless addressing, the mask (n) for a block can take any value from 0 to 32. It is convenient to write the mask as '/n' (CIDR or slash notation). Classless Interdomain Routing (CIDR) notation is used in classless addressing.

Each class has a default mask.

Default masks for classful addressing

Class	*Binary*	*Dotted-Decimal*	*CIDR*
A	**11111111** 00000000 00000000 00000000	**255**.0.0.0	/8
B	**11111111 11111111** 00000000 00000000	**255.255**.0.0	/16
C	**11111111 11111111 11111111** 00000000	**255.255.255**.0	/24

CLASSLESS ADDRESSING:

BLOCKS:

1) There are no classes, but the addresses are still granted in blocks. A block of addresses can be defined as x.y.z.t /*n*
x.y.z.t defines one of the addresses and /*n* defines the mask.

2) Restrictions on classless address blocks:

There are three restrictions on classless address blocks:
1. The addresses in a block must be contiguous, one after another.
2. The number of addresses in a block must be a power of 2 (I,2,4,8,..).
3. The first address in the block must be evenly divisible by the number of addresses.

3)

- First Address: The first address in the block can be found by setting the 32 - n rightmost bits in the binary notation of the address to 0's.

- Last Address: The last address in the block can be found by setting the 32 - n rightmost bits in the binary notation of the address to 1's.

- Number of Addresses: The number of addresses in the block is can be found using the formula 2^{32-n}.

SUBNETTING:

Ex:

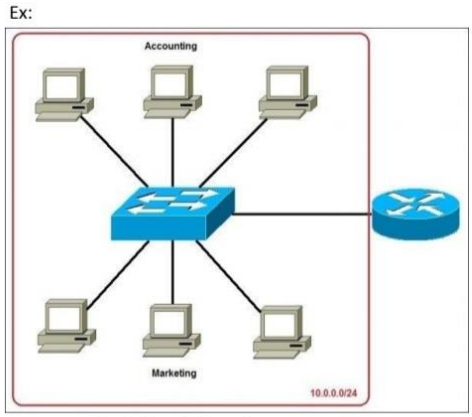


Fig:Before Subnetting

Disadvantages

- A single broadcast domain
- Network Security
- Organizational Problems

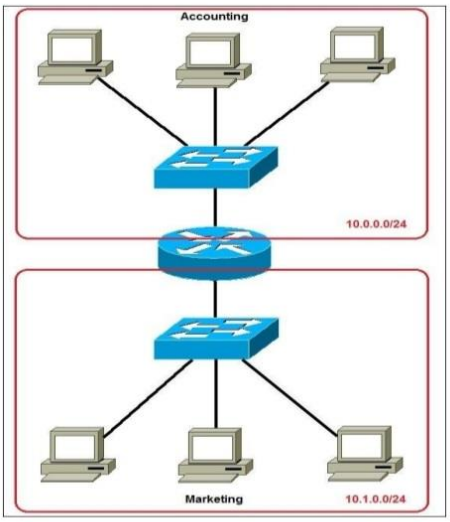


Fig:After Subnetting

1. When we subnet a network, we basically split it into smaller networks (subnets). This helps to reduce traffic and hides the complexity of the network.

2. The network has its own mask; each subnet also has its own mask.

3. Due to subnetting, the IP address has three levels of hierarchy:
- Network prefix,
- Subnet prefix (subnet mask) and
- Host address.

4. A subnet contains 2^h addresses, where h is the number of host bits. (n=mask).

The first IP address in a subnet is the Network address for that subnet. The last IP address in the subnet is the Broadcast address for that subnet. The remaining IP addresses in the subnet can be used for hosts. So we subtract 2 from 2^h while finding number of hosts per subnet.

Number of network bits = n

Number of host bits h = 32 - n.

Number of hosts = 2^h - 2.

Number of subnets bits = n - default mask for that class

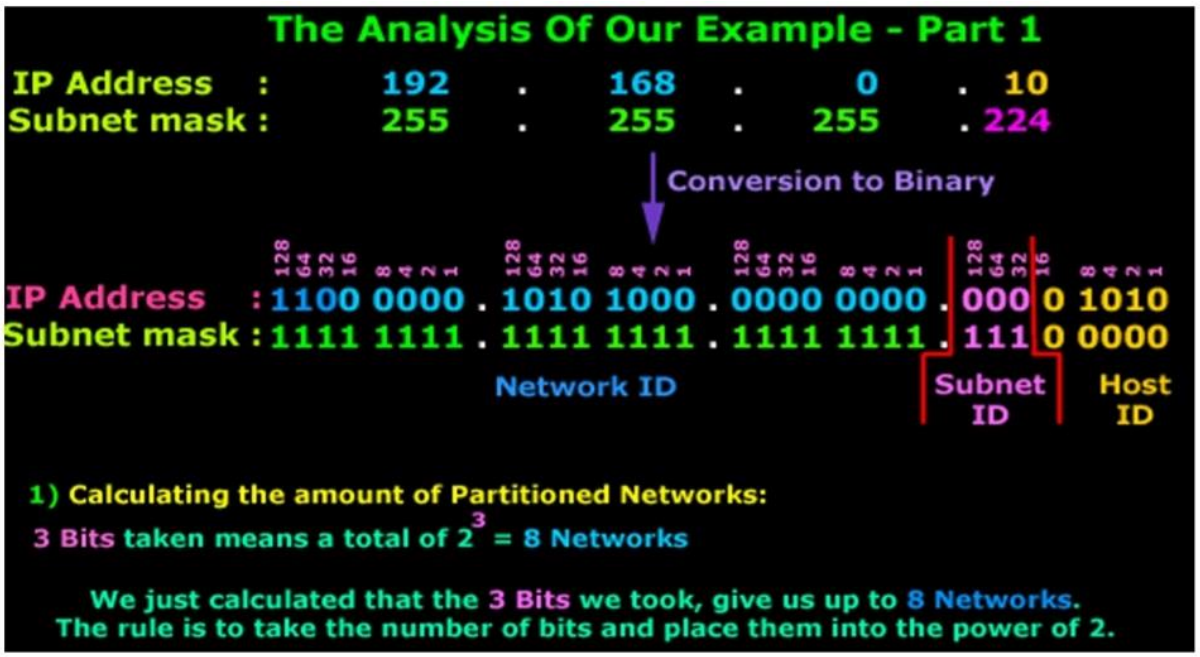
Number of subnets = $2^{\text{no. of subnet bits}}$

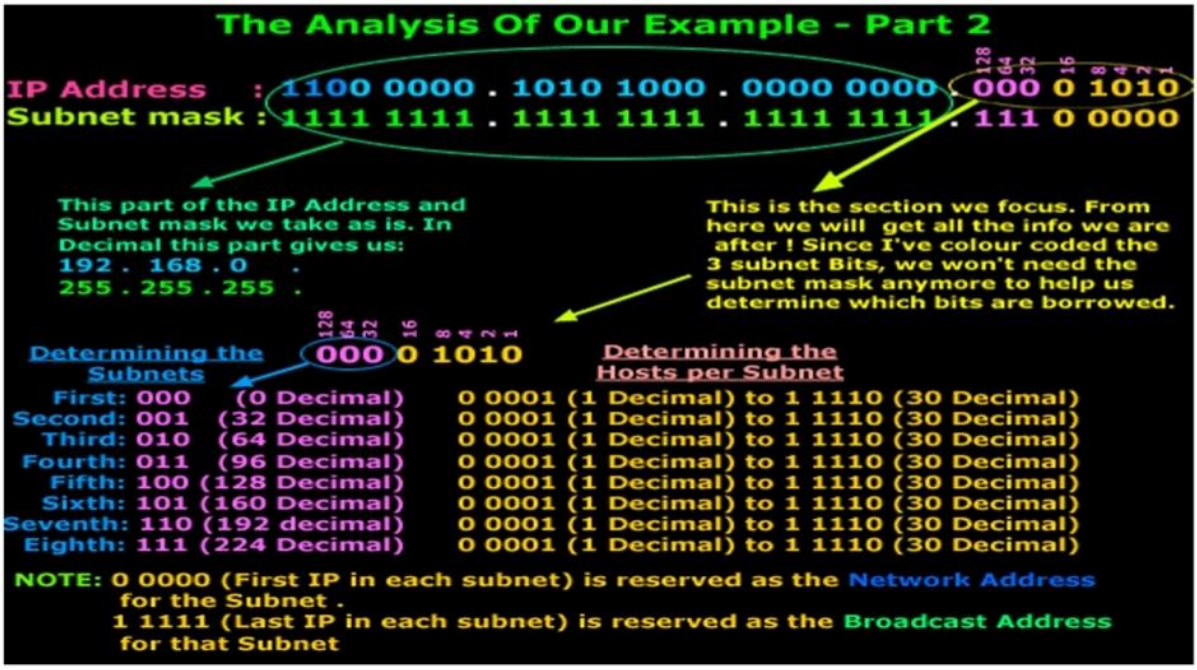
Number of hosts per subnet = 2^h - 2.

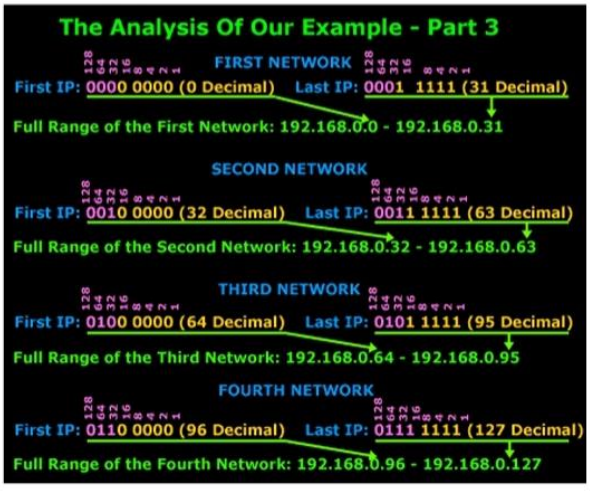
5. Example: Determine the number of Subnets and Hosts per subnets from the given IP address and Subnet mask.

IP address: 192.168.0.10

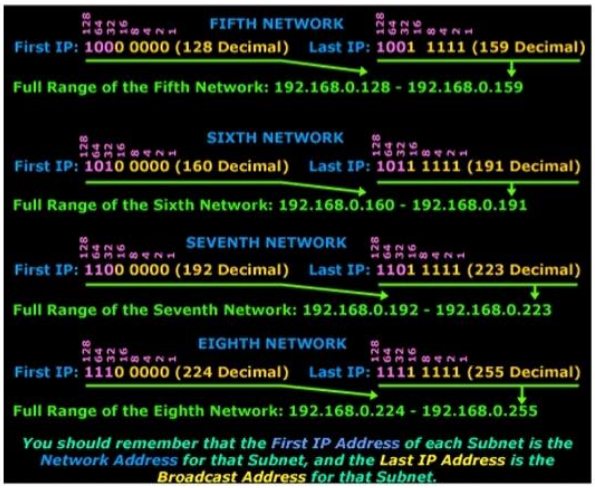
Subnet mask: 255.255.255.224







continued from above



Procedure/code:

```
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <stdio.h>
#include <math.h>
using namespace std;

int getOctetsIP(string ip, vector<int> &octetsIP) {
	stringstream sip(ip);
	string temp;
	octetsIP.clear();                                                    // Clears the octetsMask vector, in case main function re-runs this function
	vector<bool> ipInRange;
	while (getline(sip,temp,'.'))                                        // Every time getline recieves new stream element from ss, save to temp
		octetsIP.push_back(atoi(temp.c_str()));                          //... until reaches '.' delimiter, then push_back octet with new element.
	if (octetsIP.size() == 4) {
		for(int i = 0; i < octetsIP.size(); i++){
			if (octetsIP[i] >= 0 && octetsIP[i] <= 255)
				ipInRange.push_back(true);
			else
				ipInRange.push_back(false);
		}
		if (ipInRange[0]==true&&ipInRange[1]==true&&ipInRange[2]==true&&ipInRange[3]==true){
```

```
                return 0;
        }else{
                cout << endl << "There are only 255 bits per octet. Please re-
enter IP." << endl << endl;
                return 1;
        }
    }else{
        return 1;
    }
}

int getOctetsMask(string mask,  vector<int> &octetsMask) {
    stringstream smask(mask);
    string temp;
    octetsMask.clear();          // Clears the octetsMask vector, in case main
function re-runs this function
    vector<bool> maskInRange;
    while (getline(smask,temp,'.'))
        octetsMask.push_back(atoi(temp.c_str()));
    if (octetsMask.size() == 4){
        for(int i = 0; i < octetsMask.size(); i++){
            if (octetsMask[i] == 0 || octetsMask[i] == 128 ||
octetsMask[i] == 192 || octetsMask[i] == 224 || octetsMask[i] == 240 ||
octetsMask[i] == 248 || octetsMask[i] == 252 || octetsMask[i] == 254 ||
octetsMask[i] == 255)
                maskInRange.push_back(true);
            else
                maskInRange.push_back(false);
        }
```

```
	if(maskInRange[0]==true&&maskInRange[1]==true&&maskInRange[2]
==true&&maskInRange[3]==true){
			return 0;
		}else{
			cout << endl << "Subnet masks only use 2^[0-7]. Please re-
enter mask." << endl << endl;
			return 1;
		}
	}else{
		return 1;
	}
}

int calcClass(vector<int> &octetsIP) {
	if (octetsIP[0] == 10) {
		return 1;
	}else if (octetsIP[0] == 172 && octetsIP[1] >= 16 && octetsIP[1] <= 31)
{
		return 2;
	}else if (octetsIP[0] == 192 && octetsIP[1] == 168) {
		return 3;
	}else if (octetsIP[0] == 127) {
		return 4;
	}else if (octetsIP[0] >= 0 && octetsIP[0] < 127) {
		return 5;
	}else if (octetsIP[0] > 127 && octetsIP[0] < 192) {
		return 6;
	}else if (octetsIP[0] > 191 && octetsIP[0] < 224) {
```

```
                return 7;
        }else if (octetsIP[0] > 223 && octetsIP[0] < 240) {
                return 8;
        }else if (octetsIP[0] > 239 && octetsIP[0] <= 255) {
                return 9;
        }else{
                return 0;
        }
}

// Perform ANDing of IP and Subnet Mask to generate Network ID range
vector<int> getNetID(vector<int> &octetsIPBits, vector<int>
&octetsMaskBits){
        vector<int> netID;
   for (int j=0; j < octetsIPBits.size(); j++)
   {
      if ((j > 0) && (j%8 == 0))
         cout << ".";

                netID.push_back(octetsIPBits[j] & octetsMaskBits[j]);
   }
return netID;
}

// Turn Binary back to Decimal
string toString(vector<int> octets){
        ostringstream octStrm;
```

```
	for(int j = 0; j < octets.size(); j++)
	{
		if (j>0)
			octStrm << '.';

		octStrm << octets[j];
	}

	return octStrm.str();
}

// Turn Binary back to Decimal
vector<int> toDecimal(vector<int> octets, vector<int> &decimals){
	stringstream octStrm;
	decimals.clear();
	for(int j = 0; j < octets.size(); j++)
	{
		if (j>0)
			octStrm << '.';

		octStrm << octets[j];
	}

	string temp;
	while (getline(octStrm, temp, '.'))
		decimals.push_back(atoi(temp.c_str()));
```

```
        return decimals;
}

// Get the network increment
int getIncrement(vector<int> decimalMask, vector<int> decimalNetID){
        int increment = 0;
        for (int i=0; i<decimalMask.size(); i++){
                if (decimalMask[i] == 255){
                        increment = 1;
                }else if(decimalMask[i] == 254){
                        increment = 2;
                        break;
                }else if(decimalMask[i] == 252){
                        increment = 4;
                        break;
                }else if(decimalMask[i] == 248){
                        increment = 8;
                        break;
                }else if(decimalMask[i] == 240){
                        increment = 16;
                        break;
                }else if(decimalMask[i] == 224){
                        increment = 32;
                        break;
                }else if(decimalMask[i] == 192){
                        increment = 64;
                        break;
```

```
            }else if(decimalMask[i] == 128){
                increment = 128;
                break;
            }
        }
return increment;
}

// Get network id range
vector<int> getNetIDRange(vector<int> &decimalNetID, int &netInc,
vector<int> &decimalMask) {
        vector<int> netIDEnd;
        for (int i=0; i<decimalNetID.size(); i++){
            if (decimalMask[i] == 255){
                netIDEnd.push_back(decimalNetID[i]);
            }else if (decimalMask[i] < 255 && decimalMask[i] > 0){
                netIDEnd.push_back( (decimalNetID[i] + netInc) - 1 );
            }else{
                netIDEnd.push_back(255);
            }
        }
        return netIDEnd;
}

// Get subnets
int getSubnets(vector<int> &decimalMask, int &ipClass, vector<int>
&subClassMask){
        int netBits = 0;
```

```
subClassMask.clear();
        if (ipClass==1){
                subClassMask.push_back(255);
                subClassMask.push_back(0);
                subClassMask.push_back(0);
                subClassMask.push_back(0);
        }else if(ipClass==2){
                subClassMask.push_back(255);
                subClassMask.push_back(255);
                subClassMask.push_back(0);
                subClassMask.push_back(0);
        }else if(ipClass==3){
                subClassMask.push_back(255);
                subClassMask.push_back(255);
                subClassMask.push_back(255);
                subClassMask.push_back(0);
        }else if(ipClass==4 || ipClass==5){
                subClassMask.push_back(decimalMask[0]);
                subClassMask.push_back(decimalMask[1]);
                subClassMask.push_back(decimalMask[2]);
                subClassMask.push_back(decimalMask[3]);
        }

for (int i=0; i<decimalMask.size(); i++){
        if (decimalMask[i] != subClassMask[i]){
                if (decimalMask[i] == 255){
                        netBits += 8;
```

```
            continue;
        }else if (decimalMask[i] == 254){
            netBits += 7;
            continue;
        }else if (decimalMask[i] == 252){
            netBits += 6;
            continue;
        }else if (decimalMask[i] == 248){
            netBits += 5;
            continue;
        }else if (decimalMask[i] == 240){
            netBits += 4;
            continue;
        }else if (decimalMask[i] == 224){
            netBits += 3;
            continue;
        }else if (decimalMask[i] == 192){
            netBits += 2;
            continue;
        }else if (decimalMask[i] == 128){
            netBits += 1;
            continue;
        }else if (decimalMask[i] == 0){
            netBits += 0;
            continue;
        }else{
            netBits += 0;
```

```
                }
            }
        }
        int subnets = pow(2.0,netBits);
        return subnets;
}

// Get hosts per subnet
int getHostsPerSubnet(vector<int> &decimalMask){
        int hostBits = 0;
        for (int i=0; i<decimalMask.size(); i++){
            if (decimalMask[i] == 255){
                hostBits += 0;
                continue;
            }else if (decimalMask[i] == 254){
                hostBits += 1;
                continue;
            }else if (decimalMask[i] == 252){
                hostBits += 2;
                continue;
            }else if (decimalMask[i] == 248){
                hostBits += 3;
                continue;
            }else if (decimalMask[i] == 240){
                hostBits += 4;
                continue;
            }else if (decimalMask[i] == 224){
```

```
                hostBits += 5;
                continue;
            }else if (decimalMask[i] == 192){
                hostBits += 6;
                continue;
            }else if (decimalMask[i] == 128){
                hostBits += 7;
                continue;
            }else if (decimalMask[i] == 0){
                hostBits += 8;
                continue;
            }else{
                hostBits = 0;
                break;
            }
        }
        int hostsPerSubnet = pow(2.0,hostBits)-2;
        return hostsPerSubnet;
}

int main() {
char resp = 'y';
while (resp == 'y') {
            string ip;
            vector<int> octetsIP;
            while (getOctetsIP(ip, octetsIP) == 1) {
            cout << "Enter IPv4 Address: ";
```

```
(getline(cin, ip));
}

string mask;
vector<int> octetsMask;
while (getOctetsMask(mask, octetsMask) == 1) {
cout << endl << "Enter subnet mask in octets for " << ip << ": ";
(getline(cin, mask));
}
cout << endl << endl;

vector<int> decimals;
cout << "IP Address: " << toString(octetsIP) << endl;
vector<int> decimalMask = toDecimal(octetsMask, decimals);
cout << "Subnet Mask: " << toString(octetsMask) << endl;

vector<int> octetsIPBits;
vector<int> octetsMaskBits;
vector<int> netID = getNetID(octetsIP, octetsMask);
vector<int> decimalNetID = toDecimal(netID, decimals);
int netInc = getIncrement(decimalMask, decimalNetID);
cout << endl;

// Print IP Class
        cout << "----------------------------------------" << endl;
        cout << "Class Information" << endl;
        cout << "----------------------------------------" << endl;
```

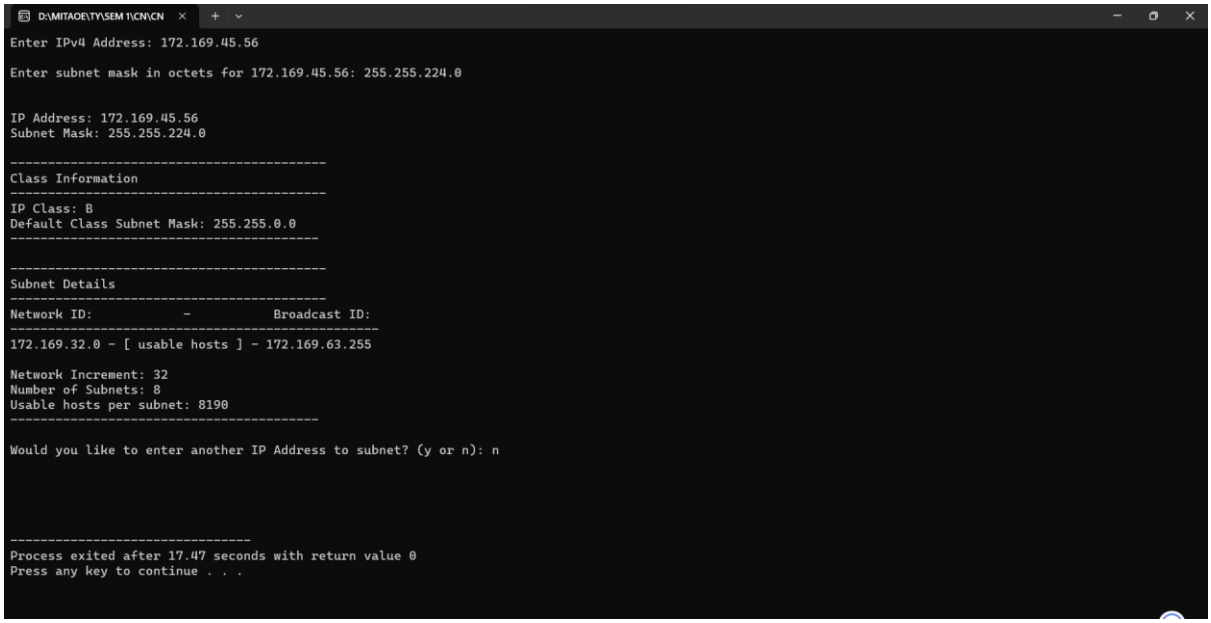
```
                int classResult = calcClass(octetsIP);
                int ipClass = 0;
                switch (classResult){
                        case 1:
                                cout << "IP Class: Private block, Class 'A' " <<
endl;
                                ipClass = 1;
                                break;
                        case 2:
                                cout << "IP Class: Private block, Class 'B'" <<
endl;
                                ipClass = 2;
                                break;
                        case 3:
                                cout << "IP Class: Private block, Class 'C'" <<
endl;
                                ipClass = 3;
                                break;
                        case 4:
                                cout << "IP Class: Reserved block, System
Loopback Address" << endl;
                                ipClass = 1;
                                break;
                        case 5:
                                cout << "IP Class: A" << endl;
                                ipClass = 1;
                                break;
                        case 6:
                                cout << "IP Class: B" << endl;
```

```
                ipClass = 2;
                break;
            case 7:
                cout << "IP Class: C" << endl;
                ipClass = 3;
                break;
            case 8:
                cout << "IP Class: D" << endl;
                ipClass = 4;
                cout << "!! This is a reserved Class D Multicast IP Address Block" << endl;
                break;
            case 9:
                cout << "IP Class: E" << endl;
                ipClass = 5;
                cout << "!! This is a reserved Class E Multicast IP Address Block" << endl;
                break;
            default :
                cout << "Not in Range" << endl;
                break;
        }
    vector<int> subClassMask;
    getSubnets(decimalMask, ipClass, subClassMask);
    cout << "Default Class Subnet Mask: " << toString(subClassMask) << endl;
    cout << "----------------------------------------" << endl << endl;
    cout << "----------------------------------------" << endl;
```

```
            cout << "Subnet Details" << endl;
            cout << "----------------------------------------" << endl;
            vector<int> netIDRange = getNetIDRange(decimalNetID, netInc,
decimalMask);
            cout << "Network ID:          -          Broadcast ID: " << endl;
                cout << "-----------------------------------------------" << endl;
                cout << toString(netID) << " - [ usable hosts ] - ";
            cout << toString(netIDRange) << endl << endl;
            cout << "Network Increment: " << getIncrement(decimalMask,
decimalNetID) << endl;
            cout << "Number of Subnets: " << getSubnets(decimalMask,
ipClass, subClassMask) << endl;
            cout << "Usable hosts per subnet: " <<
getHostsPerSubnet(decimalMask) << endl;
            cout << "----------------------------------------" << endl << endl;


            cout << "Would you like to enter another IP Address to subnet? (y
or n): ";
            cin >> resp;
            cout << endl << endl << endl << endl;
}
        return 0;
}
```

Output:



Conclusion:

In conclusion, the subnet calculator provided allows users to input an IP address and calculate various subnet-related details. It determines the class type (A, B, or C) of the given IP address, calculates the number of usable hosts in the subnet, identifies the subnet and broadcast addresses, and handles invalid inputs gracefully. The program showcases an efficient approach to subnet calculation, enhancing readability and maintainability through concise code. This practical tool empowers network administrators by simplifying the complex task of subnetting, ensuring accurate and reliable network configurations.