**Program no-1**

**Aim:-To Find the class of the network.**

**Theory:-**

A classful network is a network addressing architecture used in the [Internet](http://en.wikipedia.org/wiki/Internet) from 1981 until the introduction of [Classless Inter-Domain Routing](http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing) in 1993. The method divides the [address](http://en.wikipedia.org/wiki/IP_address) space for Internet Protocol Version 4 ([IPv4](http://en.wikipedia.org/wiki/IPv4)) into five address classes. Each class, coded in the first four bits of the address, defines either a different network size, i.e. number of [hosts](http://en.wikipedia.org/wiki/Host_(network)) for [unicast](http://en.wikipedia.org/wiki/Unicast) addresses (classes A, B, C), or a [multicast](http://en.wikipedia.org/wiki/Multicast) network (class D). The fifth class (E) address range is reserved for future or experimental purposes.

Since its discontinuation, remnants of classful network concepts remain in practice only in limited scope in the default configuration parameters of some network software and hardware components (e.g., default [subnet mask](http://en.wikipedia.org/wiki/Subnet_mask)), but the terms are often still used in general discussions of network structure among network administrators.

**Introduction of address classes:-**

Expansion of the network had to ensure compatibility with the existing address space and the [Internet Protocol](http://en.wikipedia.org/wiki/Internet_Protocol) (IP) packet structure, and avoid the renumbering of the existing networks. The solution was to expand the definition of the network number field to include more bits, allowing more networks to be designated, each potentially having fewer hosts. All existing network numbers at the time were smaller than 64, they only used the 6 least-significant bits of the network number field. Thus it was possible to use the most-significant bits of an address to introduce a set of address classes, while preserving the existing network numbers in the first of these classes.

The new addressing architecture was introduced by [RFC 791](http://tools.ietf.org/html/rfc791) in 1981 as a part of the specification of the Internet Protocol.[[1]](http://en.wikipedia.org/wiki/Classful_network#cite_note-rfc791-0) It divided the address space into primarily three address formats, henceforth called address *classes*, and left a fourth range reserved to be defined later.

The first class, designated as *Class A*, contained all addresses in which the most significant bit is zero. The network number for this class is given by the next 7 bits, therefore accommodating 128 networks in total, including the zero network, and including the existing IP networks already allocated. A *Class B* network was a network in which all addresses had the two most-significant bits set to 1 and 0. For these networks, the network address was given by the next 14 bits of the address, thus leaving 16 bits for numbering host on the network for a total of 65536 addresses per network. *Class C*was defined with the 3 high-order bits set to 1, 1, and 0, and designating the next 21 bits to number the networks, leaving each network with 256 local addresses.

The leading bit sequence *111* designated an "*escape to extended addressing mode*",[[1]](http://en.wikipedia.org/wiki/Classful_network#cite_note-rfc791-0) which was later subdivided in to Class D (*1110*) for multicast addressing, while leaving as reserved for future use the *1111* block designated as Class E.

This addressing scheme is illustrated in the following table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Class | Leading bits | Size of*network number*bit field | Size of*rest* bit field | Number of networks | Addresses per network | Start address | End address |
| Class A | 0 | 8 | 24 | 128 (27) | 16,777,216 (224) | 0.0.0.0 | 127.255.255.255 |
| Class B | 10 | 16 | 16 | 16,384 (214) | 65,536 (216) | 128.0.0.0 | 191.255.255.255 |
| Class C | 110 | 24 | 8 | 2,097,152 (221) | 256 (28) | 192.0.0.0 | 223.255.255.255 |
| Class D ([multicast](http://en.wikipedia.org/wiki/Multicast)) | 1110 | not defined | not defined | not defined | not defined | 224.0.0.0 | 239.255.255.255 |
| Class E (reserved) | 1111 | not defined | not defined | not defined | not defined | 240.0.0.0 | 255.255.255.255 |

The number of addresses usable for addressing specific hosts in each network is always 2N - 2 (where N is the number of rest field bits, and the subtraction of 2 adjusts for the use of the all-bits-zero host portion for network address and the all-bits-one host portion as a broadcast address. Thus, for a Class C address with 8 bits available in the host field, the number of hosts is 254.

Today, IP addresses are associated with a [subnet mask](http://en.wikipedia.org/wiki/Netmask). This was not required in a classful network because the mask was implicitly derived from the IP address itself. Any network device would inspect the first few bits of the IP address to determine the class of the address.

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Classful_network&action=edit&section=3)**]**Bit-wise representation

In the following table:

*n* indicates a binary slot used for network ID.

*H* indicates a binary slot used for host ID.

*X* indicates a binary slot (without specified purpose)

Class A

0. 0. 0. 0 = 00000000.00000000.00000000.00000000

127.255.255.255 = 01111111.11111111.11111111.11111111

0nnnnnnn.HHHHHHHH.HHHHHHHH.HHHHHHHH

Class B

128. 0. 0. 0 = 10000000.00000000.00000000.00000000

191.255.255.255 = 10111111.11111111.11111111.11111111

10nnnnnn.nnnnnnnn.HHHHHHHH.HHHHHHHH

Class C

192. 0. 0. 0 = 11000000.00000000.00000000.00000000

223.255.255.255 = 11011111.11111111.11111111.11111111

110nnnnn.nnnnnnnn.nnnnnnnn.HHHHHHHH

Class D

224. 0. 0. 0 = 11100000.00000000.00000000.00000000

239.255.255.255 = 11101111.11111111.11111111.11111111

1110XXXX.XXXXXXXX.XXXXXXXX.XXXXXXXX

Class E

240. 0. 0. 0 = 11110000.00000000.00000000.00000000

255.255.255.255 = 11111111.11111111.11111111.11111111

1111XXXX.XXXXXXXX.XXXXXXXX.XXXXXXXX

**Algorithm:-**

(A)For Binary IP address:-

1. bin[]🡨IP address

1. If bin[0]=’0'
2. Print This IP address is from class A
3. else if bin[0]='1'&&bin[1]='0'
4. print This IP address is from class B
5. else ifbin[0]='1'&&bin[1]='1'&&bin[2]='0'
6. print This IP address is from class C
7. else if bin[0]='1'&&bin[1]='1'&&bin[2]='1'&&bin[3]='0'
8. print This IP address is from class D
9. else if bin[0]='1'&&bin[1]='1'&&bin[2]='1'&&bin[3]='1'
10. print This IP address is from class E
11. else print Invalid IP address.

(B)For Decimal IP address:-

1. Take IP[3],a;
2. abc:
3. if dec[3]=46

4 .copy dec into ip 3 character

//change the first 3 string value to integer value.

1. A🡨atoi IP
2. ifa>255 or a<0

{

1. print Invalid IP address.
2. return;
3. If a>=240 and a<=255
4. printThis IP address is from class E
5. return;
6. else if a>=22 and a<=239
7. print This IP address is from class D
8. return;
9. else if a>=192 and a<=223
10. print This IP address is from class C
11. return;
12. else if a>=128 and a<=191
13. print This IP address is from class B
14. return;
15. else if a>=0 and a<=127
16. print This IP address is from class A
17. return;

24.else print Sorry! Invalid Ip address.Please! reenter:-

25. goto Step 2;

**Flowchart:-**

Bin[]🡨IP address

Is bin[0]=’0’? bidjkbinbits[i]==55?

No

Yes

Class A IP address

Is bin[0]=’1’and bin[1]=’0’?

No No

Yes

Yes

Class B IP address

Is bin[0]=’1’and bin[1]=’1’andbin[2]=’0’?

No

Yes

Yes

Class C IP address

Is bin[0]=’1’and bin[1]=’1’andbin[2]=’1’and bin[3]=’0’?

No

Yes

Class D IP address

Is bin[0]=’1’and bin[1]=’1’andbin[2]=’1’and bin[3]=’1’?

No

B

A

B

Class E IP address

A

Invalid IP address

**Program:-**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

void BinaryIP(char bin[35]);

void DecimalIP(char dec[19]);

void Choice();

void main()

{

char dec[19];

char bin[35];

int ch;

clrscr();

do

{

Choice();

puts("Enter the choice:=\n");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("Enter the IP address:-\n");

scanf("%s",bin);

BinaryIP(bin);

break;

case 2:

{

printf("Enter the IP address:-\n");

scanf("%s",dec);

DecimalIP(dec);

}

break;

case 3:exit(0);

default:puts("Sorry! Wrong choice.\n");

}

getch();

}while(1);

}

void Choice()

{

printf("1.Enter binary IP address.\n");

printf("2.Enter decimal IP address.\n");

printf("3.Exit.\n");

}

void BinaryIP(char bin[35])

{

if(bin[0]=='0')

printf("This IP address is from class A\n");

else if(bin[0]=='1'&&bin[1]=='0')

printf("This IP address is from class B\n");

else if(bin[0]=='1'&&bin[1]=='1'&&bin[2]=='0')

printf("This IP address is from class C\n");

else if(bin[0]=='1'&&bin[1]=='1'&&bin[2]=='1'&&bin[3]=='0')

printf("This IP address is from class D\n");

else if(bin[0]=='1'&&bin[1]=='1'&&bin[2]=='1'&&bin[3]=='1')

printf("This IP address is from class E\n");

else printf("Invalid IP address.\n");

}

void DecimalIP(char dec[19])

{

char IP[3];

int a;

abc:

if(dec[3]==46)

{

strncpy(IP,dec,3);

//change the first 3 string value to integer value.

a=atoi(IP);

if(a>255||a<0)

{

printf("Invalid IP address.\n");

return;

}

if(a>=240&&a<=255)

{

printf("This IP address is from class E\n");

return;

}

else if(a>=224&&a<=239)

{

printf("This IP address is from class D\n");

return;

}

else if(a>=192&&a<=223)

{

printf("This IP address is from class C\n");

return;

}

else if(a>=128&&a<=191)

{

printf("This IP address is from class B\n");

return;

}

else if(a>=0&&a<=127)

{

printf("This IP address is from class A\n");

return;

}

}

else

{

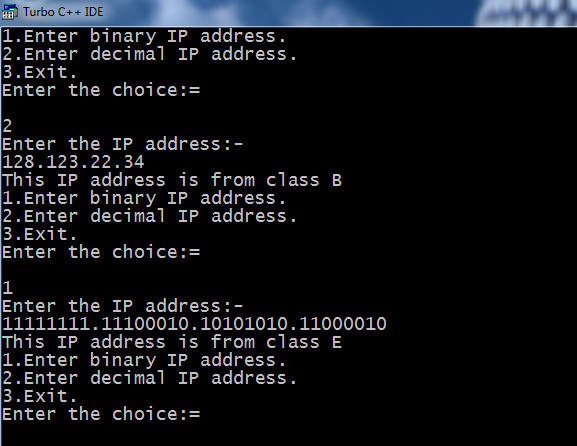
printf("Sorry! Invalid Ip address.Please! reenter:-\n");

goto abc;

}

}

**Output:-**



**Program no-2**

**Aim:-Generate hamming code for a given 7bit binary data.**

Theory:-

In [telecommunication](http://en.wikipedia.org/wiki/Telecommunication), a Hamming code is a [linear](http://en.wikipedia.org/wiki/Linear_code) [error-correcting code](http://en.wikipedia.org/wiki/Error-correcting_code) named after its inventor, [Richard Hamming](http://en.wikipedia.org/wiki/Richard_Hamming). Hamming codes can detect up to two simultaneous bit errors, and correct single-bit errors; thus, reliable communication is possible when the [Hamming distance](http://en.wikipedia.org/wiki/Hamming_distance) between the transmitted and received bit patterns is less than or equal to one. By contrast, the simple [parity](http://en.wikipedia.org/wiki/Parity_bit) code cannot correct errors, and can only detect an odd number of errors.

In [mathematical](http://en.wikipedia.org/wiki/Mathematics) terms, Hamming codes are a class of binary linear codes. For each integer m \ge 2 there is a code with *m* parity bits and 2*m* − *m* − 1 data bits. The [parity-check matrix](http://en.wikipedia.org/wiki/Parity-check_matrix) of a Hamming code is constructed by listing all columns of length *m* that are pairwise [independent](http://en.wikipedia.org/wiki/Linear_Independence). Hamming codes are an example of [perfect codes](http://en.wikipedia.org/wiki/Perfect_code), codes that exactly match the theoretical upper bound on the number of distinct code words for a given number of bits and ability to correct errors.

General algorithm

The following general algorithm generates a single-error correcting (SEC) code for any number of bits.

Number the bits starting from 1: bit 1, 2, 3, 4, 5, etc.

Write the bit numbers in binary. 1, 10, 11, 100, 101, etc.

All bit positions that are powers of two (have only one 1 bit in the binary form of their position) are parity bits.

All other bit positions, with two or more 1 bits in the binary form of their position, are data bits.

Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position.

Parity bit 1 covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself), 3, 5, 7, 9, etc.

Parity bit 2 covers all bit positions which have the second least significant bit set: bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.

Parity bit 4 covers all bit positions which have the third least significant bit set: bits 4–7, 12–15, 20–23, etc.

Parity bit 8 covers all bit positions which have the fourth least significant bit set: bits 8–15, 24–31, 40–47, etc.

In general each parity bit covers all bits where the binary AND of the parity position and the bit position is non-zero.

The form of the parity is irrelevant. Even parity is simpler from the perspective of theoretical mathematics, but there is no difference in practice.

This general rule can be shown visually:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | ... |
| Encoded data bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 | d9 | d10 | d11 | p16 | d12 | d13 | d14 | d15 |
| Parity bit coverage | p1 | X |  | X |  | X |  | X |  | X |  | X |  | X |  | X |  | X |  | X |  |
| p2 |  | X | X |  |  | X | X |  |  | X | X |  |  | X | X |  |  | X | X |  |
| p4 |  |  |  | X | X | X | X |  |  |  |  | X | X | X | X |  |  |  |  | X |
| p8 |  |  |  |  |  |  |  | X | X | X | X | X | X | X | X |  |  |  |  |  |
| p16 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | X | X | X | X | X |

Shown are only 20 encoded bits (5 parity, 15 data) but the pattern continues indefinitely. The key thing about Hamming Codes that can be seen from visual inspection is that any given bit is included in a unique set of parity bits. To check for errors, check all of the parity bits. The pattern of errors, called the [error syndrome](http://en.wikipedia.org/wiki/Syndrome_decoding), identifies the bit in error. If all parity bits are correct, there is no error. Otherwise, the sum of the positions of the erroneous parity bits identifies the erroneous bit. For example, if the parity bits in positions 1, 2 and 8 indicate an error, then bit 1+2+8=11 is in error. If only one parity bit indicates an error, the parity bit itself is in error.

If, in addition, an overall parity bit (bit 0) is included, the code can detect (but not correct) any two-bit error, making a SECDED code. The overall parity indicates whether the total number of errors is even or odd. If the basic Hamming code detects an error, but the overall parity says that there are an even number of errors, an uncorrectable 2-bit error has occurred.

**Algorithm:-**

1 .Take input in the array num[]🡨bits

2. Calculate first parity bit.

3. Calculate second parity bit.

4.Calculate fourth parity bit

5.Put the above parity bits in the position of data bits.

6.Print hamming code.

**Flowchart:-**

Bits[]🡨Databits

Bits[1]🡨55(Constant),Bits[2]🡨55,Bits[4]🡨55

Count🡨0,J🡨1,i🡨0

I🡨i+1

Is bits[i]==55?

Yes

No

Is J<=7

No

yes

Is bits[J]==1?

No

Yes

Count🡨count+1

J🡨J+2

Is i<=7?

Yes

No

Is count%2==0?

No

Yes

Bits[1]🡨0

Bits[1]🡨1

**Program:-**

#include<stdio.h>

#include<conio.h>

void First(int bits[8]);

void Second(int bits[8]);

void Fourth(int bits[8]);

int i,bits[8];

void main()

{

clrscr();

printf("Welcome to the Computer Network Program:-\n");

puts("Enter the data bits:-\n");

for(i=1;i<=7;i++)

{

scanf("%d",&bits[i]);

}

printf("Input is following:-\n");

for(i=1;i<=7;i++)

{

printf("%d\t",bits[i]);

}

bits[1]=55;bits[2]=55;bits[4]=55;

First(bits);

Second(bits);

Fourth(bits);

printf("\nWe have use the even prity system.\n");

printf("\nThe hamming code is following:-\n");

for(i=1;i<=7;i++)

{

printf("%d\t",bits[i]);

}

getch();

}

void First(int bits[8])

{

int j,count=0;

j=1;

for(i=1;i<=7;i++)

{

if(bits[i]==55)

continue;

if(j<=7)

{

if(bits[j]==1)

count++;

}

j=j+2;

}

if(count%2==0)

bits[1]=0;

else bits[1]=1;

}

void Second(int bits[8])

{

int j,count=0,check=0;

count=0;

j=2;

for(i=1;i<=7;i++)

{

if(bits[i]==55)

continue;

if(j<=7)

{

if(bits[j]==1)

count++;

}

if(check==2)

{

j=j+2; check=0;

}

else {j=j+1;}

check++;

}

if(count%2==0)

bits[2]=0;

else bits[2]=1;

}

void Fourth(int bits[8])

{

int j,count=0,check=0;

count=0;

j=4;

for(i=1;i<=7;i++)

{

if(bits[i]==55)

continue;

if(j<=7)

{

if(bits[j]==1)

count++;

}

if(check==4)

{

j=j+5; check=0;

}

else {j=j+1;}

check++;

}

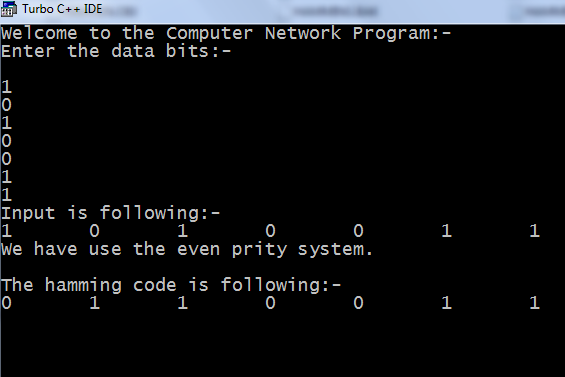
if(count%2==0)

bits[4]=0;

else bits[4]=1;

}

Output:-



**Program no-3**

**Aim:-Program to check the error in the hamming code.**

**Theory:-**To check the error in the hamming code we count the number of 1’s if it is even or odd, there is error in the hamming code.

**Algorithm:-**

1. Take input in the array num[]🡨bits

2. Find length len🡨Length

3. Set count🡨0

4. For i🡨1 to len

5. do if num[i]=’1’ then do

6. count🡨count+1

7.if sum=0

Print No Input found.

8.if sum%2=0

Print There is no error in hamming code.

else

Print Sorry! There is an error in the hamming code.

**Flowchart:-**

Num[]🡨bits

len🡨length

Count🡨0

i🡨0

i🡨i+1

Count🡨count+1

Is num[i]=

‘1’?

no

is i=len?

no

Is count%2=0 ?

yes

Print no error

Print error

**Program:-**

#include<stdio.h>

#include<conio.h>

#define MAX 65535

void Error(char number[]);

void main()

{

charnum[MAX];

int i;

clrscr();

puts("Welcome to the Computer Network Program.\n");

printf("Program for checking the error in hamming code.\n");

printf("Enter the hamming code:-\n");

gets(num);

Error(num);

getch();

}

void Error(char n[])

{

intlen,sum=0,i;

len=strlen(n);

for(i=0;i<len;i++)

{

if(n[i]=='1')

sum++;

}

if(sum==0)

{

printf("Sorry! No Input found.");

getch();

return;

}

if(sum%2==0)

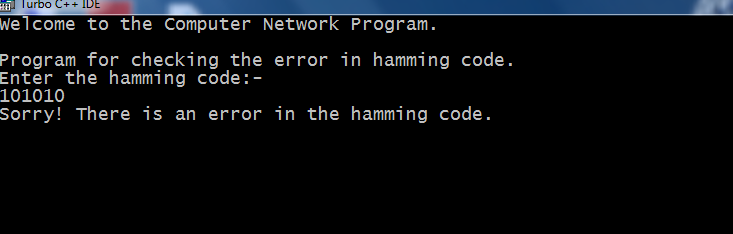
puts("There is no error in hamming code.");

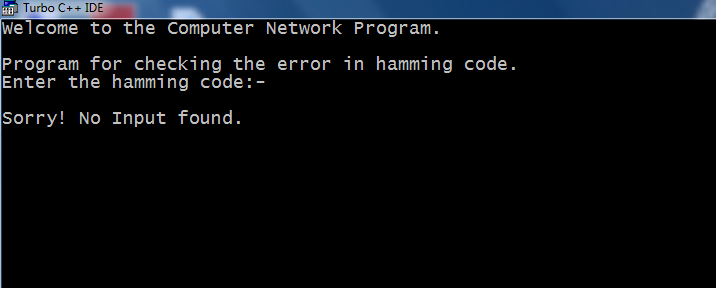
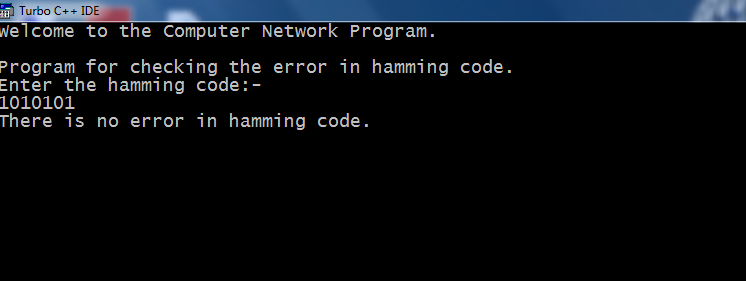
else

puts("Sorry! There is an error in the hamming code.");

}

**Output:-**





**Program no-4**

**Aim:-Program to check the error in the hamming code using even parity.**

**Theory:-**To check the error in the hamming code we count the number of 1’s if it is even , there is error in the hamming code.

**Algorithm:-**

1. Take input in the array num[]🡨bits

2. Find length len🡨Length

3. Set count🡨0

4. For i🡨1 to len

5. do if num[i]=’1’ then do

6. count🡨count+1

7.if sum=0

Print No Input found.

8. if sum%2=0

Print There is no error in hamming code.

else

Print Sorry! There is an error in the hamming code.

**Flowchart:-**

Num[]🡨bits

len🡨length

Count🡨0

i🡨0

i🡨i+1

Count🡨count+1

Is num[i]=

‘1’?

no

is i=len?

no

Is count%2=0 ?

yes

Print no error

Print error

**Program:-**

#include<stdio.h>

#include<conio.h>

#define MAX 65535

void Error(char number[]);

void main()

{

charnum[MAX];

int i;

clrscr();

puts("Welcome to the Computer Network Program.\n");

printf("Program for checking the error in hamming code using odd parity.\n");

printf("Enter the hamming code:-\n");

gets(num);

Error(num);

getch();

}

void Error(char n[])

{

intlen,sum=0,i;

len=strlen(n);

for(i=0;i<len;i++)

{

if(n[i]=='1')

sum++;

}

if(sum==0)

{

printf("Sorry! No Input found.");

getch();

return;

}

if(sum%2==0)

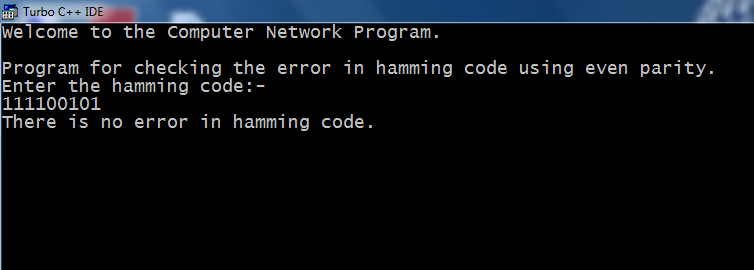
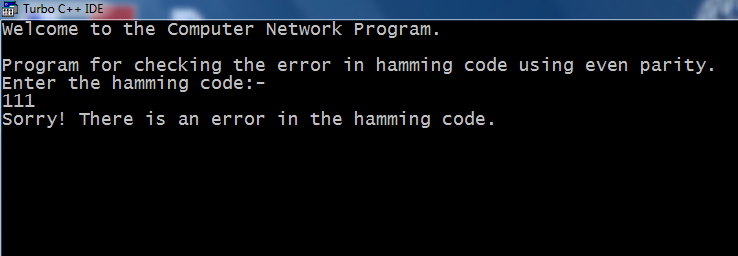
puts("There is no error in hamming code.");

else

puts("Sorry! There is an error in the hamming code.");

}

**Output:-**



**Program no-5**

**Aim:-Program to check the error in the hamming code using odd parity.**

**Theory:-**To check the error in the hamming code we count the number of 1’s if it is odd, there is error in the hamming code.

**Algorithm:-**

1. Take input in the array num[]🡨bits

2. Find length len🡨Length

3. Set count🡨0

4. For i🡨1 to len

5. do if num[i]=’1’ then do

6. count🡨count+1

7.if sum=0

Print No Input found.

8. if sum%2=0

Print There is no error in hamming code.

else

Print Sorry! There is an error in the hamming code.

**Flowchart:-**

Num[]🡨bits

len🡨length

Count🡨0

i🡨0

i🡨i+1

Count🡨count+1

Is num[i]=

‘1’?

no

is i=len?

no

Is count%2!!=0 ?

yes

Print no error

Print error

**Program:-**

#include<stdio.h>

#include<conio.h>

#define MAX 65535

void Error(char number[]);

void main()

{

charnum[MAX];

int i;

clrscr();

puts("Welcome to the Computer Network Program.\n");

printf("Program for checking the error in hamming code using odd parity.\n");

printf("Enter the hamming code:-\n");

gets(num);

Error(num);

getch();

}

void Error(char n[])

{

intlen,sum=0,i;

len=strlen(n);

for(i=0;i<len;i++)

{

if(n[i]=='1')

sum++;

}

if(sum==0)

{

printf("Sorry! No Input found.");

getch();

exit();

}

if(sum%2!=0)

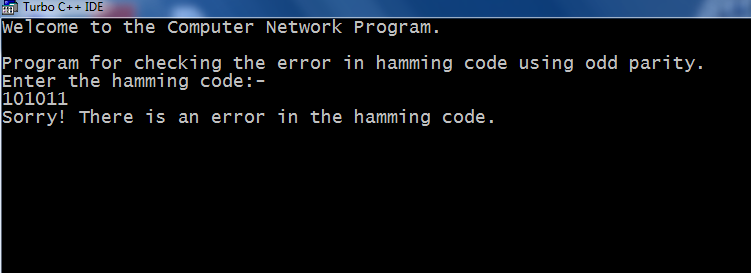
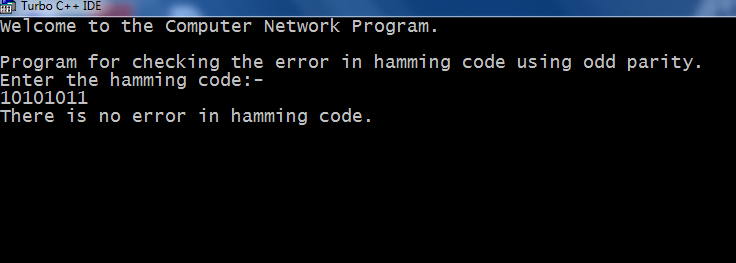
puts("There is no error in hamming code.");

else

puts("Sorry! There is an error in the hamming code.");

}

**Output:-**



**Program no-6**

**Aim:-Program to Implement bit stuffing having 24 bits long input string with 10000001 starting and ending text.**

**Theory:-**

**Bit Stuffing:-**“bit stuffing-The practice of adding bits to a stream of data. Bit stuffing is used by many network and communication protocols for the following reasons: To prevent data being interpreted as control information. For example, many frame-based protocols, such as X.25, signal the beginning and end of a frame with six consecutive 1’s. Therefore, if the actual data has six consecutive 1 bits in a row, a zero is inserted after the first 5 bits… Of course, on the receiving end, the stuffed bits must be discarded…”

data: 01111110111110101010…

This won’t work: transmit: 011111010111110101010…

But this will: transmit: 0111110101111100101010…

**Algorithm:-**

1.bits🡨total bits

2. count🡨0

3. for i🡨0 to len

4. do if bits[i]='1'

5. do count🡨count+1;

6. if count=5

7. dolen++;

8. for k🡨len to k>i+1

9. do bits[k]🡨bits[k-1];

10. bits[i+1]🡨'0';

11. count🡨0;

12. if(bits[i]='0')

13. count=0;

**Flowchart:-**

bits[]🡨total bits

len🡨length

Count🡨0

i🡨0

i🡨i+1

is i=len?

yes

no

Is bits[i]=

‘1’?

no

yes

Count🡨count+1

Is count=5?

‘1’?

no

len🡨len+1

K🡨k-1

K🡨len

bit[k]-🡨bit[k-1]

isik>i+1 ?

yes

Bit[i+1]🡨’0’

Count🡨0

Is bit[i]=’0’

?

yes

Count🡨0

yes

Print bit[]

**Program:-**

#include<stdio.h>

#include<conio.h>

#define MAX 65535

voidBitstuff(char bits[]);

inti,len;

void main()

{

char bits[MAX];

clrscr();

puts("Welcome to the Computer Network program.\n");

puts("Enter the bits:-\n");

gets(bits);

Bitstuff(bits);

puts("\nAfter bit Stuffing the code is following:-\n");

for(i=0;i<len;i++)

printf("%c",bits[i]);

getch();

}

voidBitstuff(char bits[])

{

int count=0,k;

len=strlen(bits);

for(i=0;i<len;i++)

{

if(bits[i]=='1')

{

count++;

if(count==5)

{

len++;

for(k=len;k>i+1;k--)

{

bits[k]=bits[k-1];

}

bits[i+1]='0';

count=0;

}

}

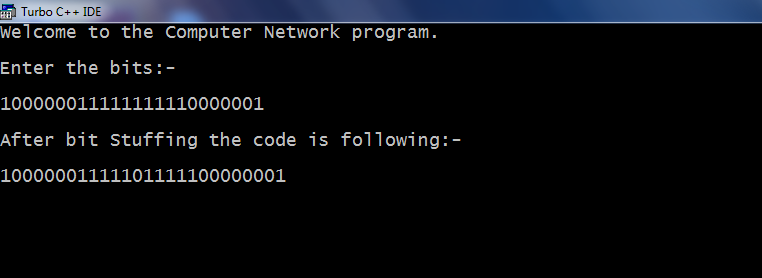
if(bits[i]=='0')

count=0;

}

}

**Output:-**



**Program no-7**

**Aim:-Program illustrating the shortest path algorithm.**

**Theory:-**

**Dijkstra's algorithm**, conceived by Dutch [computer scientist](http://en.wikipedia.org/wiki/Computer_scientist) [Edsger Dijkstra](http://en.wikipedia.org/wiki/Edsger_Dijkstra) in 1956 and published in 1959,[[1]](http://en.wikipedia.org/wiki/Dijkstra's_algorithm#cite_note-Dijkstra_Interview-0)[[2]](http://en.wikipedia.org/wiki/Dijkstra's_algorithm#cite_note-1) is a [graph search algorithm](http://en.wikipedia.org/wiki/Graph_search_algorithm) that solves the single-source [shortest path problem](http://en.wikipedia.org/wiki/Shortest_path_problem) for a [graph](http://en.wikipedia.org/wiki/Graph_(mathematics)) with nonnegative [edge](http://en.wikipedia.org/wiki/Edge_(graph_theory)) path costs, producing a [shortest path tree](http://en.wikipedia.org/wiki/Shortest_path_tree). This algorithm is often used in [routing](http://en.wikipedia.org/wiki/Routing) and as a subroutine in other graph algorithms.

For a given source [vertex](http://en.wikipedia.org/wiki/Vertex_(graph_theory)) (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network [routing protocols](http://en.wikipedia.org/wiki/Routing_protocol), most notably [IS-IS](http://en.wikipedia.org/wiki/IS-IS) and [OSPF](http://en.wikipedia.org/wiki/OSPF)(Open Shortest Path First).

Dijkstra's original algorithm does not use a [min-priority queue](http://en.wikipedia.org/wiki/Min-priority_queue) and runs in *O*(|*V*|2). The idea of this algorithm is also given in ([Leyzorek et al. 1957](http://en.wikipedia.org/wiki/Dijkstra's_algorithm#CITEREFLeyzorekGrayJohnsonLadew1957)). The common implementation based on a min-priority queue implemented by a [Fibonacci heap](http://en.wikipedia.org/wiki/Fibonacci_heap) and running in *O*(|*E*| + |*V*| log |*V*|) is due to ([Fredman & Tarjan 1984](http://en.wikipedia.org/wiki/Dijkstra's_algorithm#CITEREFFredmanTarjan1984)). This is [asymptotically](http://en.wikipedia.org/wiki/Asymptotic_computational_complexity) the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded nonnegative weights. (For an overview of earlier shortest path algorithms and later improvements and adaptations, see: [Single-source](http://en.wikipedia.org/wiki/Single-source_shortest-paths_algorithms_for_directed_graphs_with_nonnegative_weights)

**Algorithm:-**

Let the node at which we are starting be called the **initial node**. Let the **distance of node Y** be the distance from the **initial node** to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes as unvisited. Set initial node as current.
3. For current node, consider all its unvisited neighbors and calculate their *tentative* distance. For example, if current node (A) has distance of 6, and an edge connecting it with another node (B) is 2, the distance to B through A will be 6+2=8. If this distance is less than the previously recorded distance, overwrite the distance.
4. When we are done considering all neighbors of the current node, mark it as visited. A visited node will not be checked ever again; its distance recorded now is final and minimal.
5. If all nodes have been visited, finish. Otherwise, set the unvisited node with the smallest distance (from the initial node, considering all nodes in graph) as the next "current node" and continue from step 3.

**Program:-**

#include<stdio.h>

#include<conio.h>

void printpath(int,int,int[]);

int minimum(int a[],int m[],int k);

void main()

{

int graph[15][15],s[15],pathestimate[15],mark[15];

int num\_of\_vertices,source,i,j,u,predecessor[15];

int count=0;

int minimum(int a[],int m[],int k);

clrscr();

printf("Welcome to the computer network program.\n");

printf("\nEnter the no.of vertices\n");

scanf("%d",&num\_of\_vertices);

if(num\_of\_vertices<=0)

{

printf("\nThis is meaningless\n");

exit(1);

}

printf("\nEnter the adjacent matrix\n");

for(i=1;i<=num\_of\_vertices;i++)

{

printf("\nEnter the elements of row %d\n",i);

for(j=1;j<=num\_of\_vertices;j++)

{

scanf("%d",&graph[i][j]);

}

}

printf("\nEnter the source vertex\n");

scanf("%d",&source);

for(j=1;j<=num\_of\_vertices;j++)

{

mark[j]=0;

pathestimate[j]=999;

predecessor[j]=0;

}

pathestimate[0]=0;

while(count<num\_of\_vertices)

{

u=minimum(pathestimate,mark,num\_of\_vertices);

s[++count]=u;

mark[u]=1;

for(i=1;i<=num\_of\_vertices;i++)

{

if(graph[u][i]>0)

{

if(mark[i]!=1)

{

if(pathestimate[i]>pathestimate[u]+graph[u][i])

{

pathestimate[i]=pathestimate[u]+graph[u][i];

predecessor[i]=u;

}

}

}

}

}

for(i=1;i<=num\_of\_vertices;i++)

{

printpath(source,i,predecessor);

if(pathestimate[i]!=999)

printf("->(%d)\n",pathestimate[i]);

}

getch();

}

int minimum(int a[],int m[],int k)

{

int mi=999;

int i,t;

for(i=1;i<=k;i++)

{

if(m[i]!=1)

{

if(mi>=a[i])

{

mi=a[i];

t=i;

}

}

}

return t;

}

void printpath(int x,int i,int p[])

{

printf("\n");

if(i==x)

{

printf("%d",x);

}

else if(p[i]==0)

printf("No path from %d to %d",x,i);

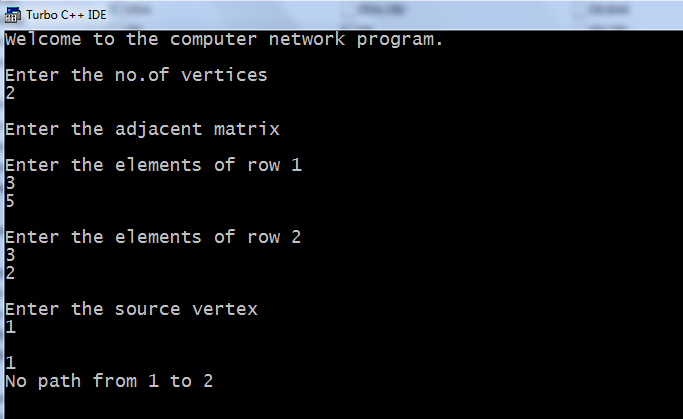
else

{

printpath(x,p[i],p);

printf("..%d",i);}}

**Output:-**

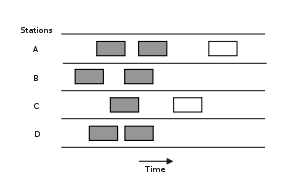
****

**Program no-8**

**Aim:-Study of pure ALOHA and to write a program illustrating the pure ALOHA throughput.**

**Theory:-**

**Pure ALOHA:-**

[](http://en.wikipedia.org/wiki/File:Pure_ALOHA1.svg)

[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Pure_ALOHA1.svg)

Pure ALOHA protocol. Boxes indicate frames. Shaded boxes indicate frames which have collided.

The first version of the protocol (now called "Pure ALOHA", and the one implemented in ALOHAnet) was quite simple:

If you have data to send, send the data

If the message collides with another transmission, try resending "later"

Note that the first step implies that Pure ALOHA does not check whether the channel is busy before transmitting. The critical aspect is the "later" concept: the quality of the backoff scheme chosen significantly influences the efficiency of the protocol, the ultimate channel capacity, and the predictability of its behavior.

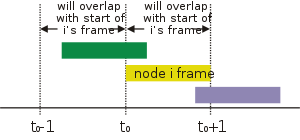
To assess Pure ALOHA, we need to predict its throughput, the rate of (successful) transmission of frames. (This discussion of Pure ALOHA's performance follows Tanenbaum.[[9]](http://en.wikipedia.org/wiki/ALOHAnet#cite_note-tann-8)) First, let's make a few simplifying assumptions:

All frames have the same length.

Stations cannot generate a frame while transmitting or trying to transmit. (That is, if a station keeps trying to send a frame, it cannot be allowed to generate more frames to send.)

The population of stations attempts to transmit (both new frames and old frames that collided) according to a [Poisson distribution](http://en.wikipedia.org/wiki/Poisson_distribution).

Let "*T*" refer to the time needed to transmit one frame on the channel, and let's define "frame-time" as a unit of time equal to *T*. Let "*G*" refer to the mean used in the Poisson distribution over transmission-attempt amounts: that is, on average, there are *G* transmission-attempts per frame-time.

[](http://en.wikipedia.org/wiki/File:Pure_ALOHA.svg)

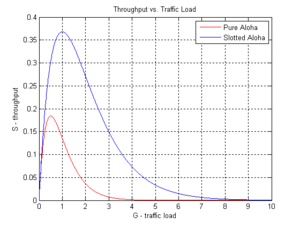
[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Pure_ALOHA.svg)

Overlapping frames in the pure ALOHA protocol. Frame-time is equal to 1 for all frames.

Consider what needs to happen for a frame to be transmitted successfully. Let "*t*" refer to the time at which we want to send a frame. We want to use the channel for one frame-time beginning at *t*, and so we need all other stations to refrain from transmitting during this time. Moreover, we need the other stations to refrain from transmitting between *t-T* and *t* as well, because a frame sent during this interval would overlap with our frame.

For any frame-time, the probability of there being *k* transmission-attempts during that frame-time is:

\frac{G^k e^{-G}}{k!}

[](http://en.wikipedia.org/wiki/File:Aloha_SvG.PNG)

[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Aloha_SvG.PNG)

Comparison of Pure Aloha and Slotted Aloha shown on Throughput vs. Traffic Load plot.

The average amount of transmission-attempts for 2 consecutive frame-times is 2*G*. Hence, for any pair of consecutive frame-times, the probability of there being *k* transmission-attempts during those two frame-times is:

\frac{(2G)^k e^{-2G}}{k!}

Therefore, the probability (*Probpure*) of there being zero transmission-attempts between *t-T* and *t+T* (and thus of a successful transmission for us) is:

*Probpure* = *e* − 2*G*

The throughput can be calculated as the rate of transmission-attempts multiplied by the probability of success, and so we can conclude that the throughput (*Spure*) is:

*Spure* = *Ge* − 2*G*

The maximum throughput is *0.5/e* frames per frame-time (reached when *G* = 0.5), which is approximately 0.184 frames per frame-time. This means that, in Pure ALOHA, only about 18.4% of the time is used for successful transmissions.

**Algorithm:-**

1. Take G.

2. Input S.

3. Calculate S=G\*(exp(2\*-G));

4. Print S as Throughput

**Flowchart:-**

Input S

S=G\*(exp(2\*-G));

Print S

**Program:-**

#include<stdio.h>

#include<conio.h>

#include<math.h>

void main()

{

float G,S;

clrscr();

printf("Weocome to the computer network program.\n");

printf("Thoughput(S)=G\*e^(2\*-G)\nP0=e^-G\n");

printf("Where G=Channel load or refers to mean used in the Poisson distribution\n");

printf("over transmission attempt amount that is: there are G Transmission attempts\n");

printf("per-frame time.\n");

printf("Enter the Channel load:-\n");

scanf("%f",&G);

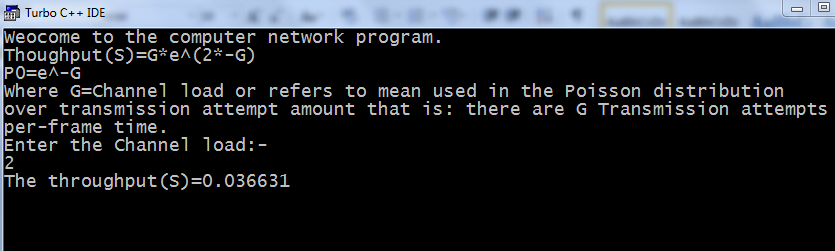
S=G\*(exp(-2\*G));

printf("The throughput(S)=%f",S);

getch();

}

**Output:-**

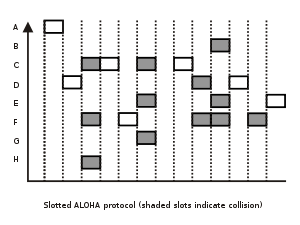


**Program no-9**

**Aim:-Study of slotted ALOHA and write a program illustrating the slotted ALOHA thoughput.**

**Theory:-**

**Slotted ALOHA:-**

[](http://en.wikipedia.org/wiki/File:Slotted_ALOHA.svg)

[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Slotted_ALOHA.svg)

Slotted ALOHA protocol. Boxes indicate frames. Shaded boxes indicate frames which are in the same slots.

An improvement to the original ALOHA protocol was "Slotted ALOHA", which introduced discrete timeslots and increased the maximum throughput.[[10]](http://en.wikipedia.org/wiki/ALOHAnet#cite_note-9) A station can send only at the beginning of a timeslot, and thus collisions are reduced. In this case, we only need to worry about the transmission-attempts within 1 frame-time and not 2 consecutive frame-times, since collisions can only occur during each timeslot. Thus, the probability of there being zero transmission-attempts in a single timeslot is:

*Probslotted* = *e* –*G*the probability of k packets is:

*Probslottedk* = *e* −*G*(1 − *e* −*G*)*k*− 1

The throughput is:*Sslotted* = *Ge* −*G*

The maximum throughput is *1/e* frames per frame-time (reached when *G* = 1), which is approximately 0.368 frames per frame-time, or 36.8%.

Slotted ALOHA is used in low-data-rate tactical [satellite communications](http://en.wikipedia.org/wiki/Communications_satellite) networks by military forces, in subscriber-based satellite communications networks, mobile telephony call setup, and in the contactless [RFID](http://en.wikipedia.org/wiki/RFID) technologies.

**Algorithm:-**

1. Take G.

2. Input S.

3. Calculate S=G\*(exp(-G));

4. Print S as Throughput

**Flowchart:-**

Input S

S=G\*(exp(2\*-G));

Print S

**Program:-**

#include<stdio.h>

#include<conio.h>

#include<math.h>

void main()

{

float G,S;clrscr();

printf("Weocome to the computer network program.\n");

printf("Thoughput(S)=G\*e^(-G)\nP0=e^-G\n");

printf("Where G=Channel load or refers to mean used in the Poisson distribution\n");

printf("over transmission attempt amount that is: there are G Transmission attempts\n");

printf("per-frame time.\n");

printf("Enter the Channel load:-\n");

scanf("%f",&G);

S=G\*(exp(-G));

printf("The throughput(S)=%f",S); getch();

}

**Output:-**