BitStuffing

```
#include<stdio.h>
#include<string.h>

void bitstuffing(char *input,char *output){
int j=0;
int count=0;

for(int i=0;i<strlen(input);i++){
    output[j++]=input[i];


    if(input[i]=='1'){
        count++;
        }else{
        count=0;
        }
    if(count==5){
        output[j++]='0';
        count=0;


    }
}
    output[j]='\0';
}
int main(){
    char input[100],output[200];

    printf("enter the valuers in 0's and 1's: ");
    scanf("%s",&input);
    bitstuffing(input,output);
    printf("stuffed bbits:%s\n",output);
    return 0;

}
```

## CharStuffing

```
#include <stdio.h>
#include <string.h>

int main() {
    char ip[100], op[200];
    int i, j = 0;

    // Take input text
    printf("\nEnter a text: ");
    fgets(ip, sizeof(ip), stdin);

    // Process input to handle FLAG and ESC
    for (i = 0; ip[i]; i++) {
        if (strncmp(ip + i, "FLAG", 4) == 0) {
            strcpy(op + j, "ESC FLAG");
            j += 7;
            i += 3; // Skip the next 3 characters (to move past "FLAG")
        } else if (strncmp(ip + i, "ESC", 3) == 0) {
            strcpy(op + j, "ESC ESC");
            j += 6;
            i += 2; // Skip the next 2 characters (to move past "ESC")
        } else {
            op[j++] = ip[i]; // Copy regular characters as is
        }
    }

    op[j] = '\0'; // Null-terminate the output string

    // Print the output wrapped with FLAG at the beginning and end
    printf("\nOutput is: FLAG %s FLAG\n", op);
    return 0;
}
```

## GoBackN

```
#include<stdio.h>
int main(){
 int windowsize,sent=0,ack;
 printf("enter the window size:");
 scanf("%d",&windowsize);
 while(sent<10){
    for(int i=0;i<windowsize&&sent<10;i++){
        printf("frame %d has been transmitted succesfully.\n",sent++);

    }
    printf("\n enter the last received acknowledgement:");
    scanf("%d",&ack);
 }
 printf("\n all frames transmitted and acknowledged successfully.\n");
 return 0;
}
```

CRC

```
#include <stdio.h>
int main() {
    int messageSize, generatorSize, i, j = 0, r, s;
    int message[50], generator[50], transmitted[50], temp[50], remainder[50], transmittedData[50];

    // Input the message size and the message itself
    printf("Enter the size of message: ");
    scanf("%d", &messageSize);
    printf("Enter the message in (0s and 1s): ");
    for (i = 0; i < messageSize; i++) {
        scanf("%d", &message[i]);
    }

    // Input the generator size and the generator itself
    printf("Enter the size of generator: ");
    scanf("%d", &generatorSize);
    printf("Enter the generator in (0s and 1s): ");
    for (i = 0; i < generatorSize; i++) {
        scanf("%d", &generator[i]);
    }

    r = generatorSize - 1;  // Degree of the generator polynomial
    s = messageSize + r;  // Size of the transmitted data (message + remainder)

    // Append zeros to the message (equivalent to adding the remainder)
    for (i = 0; i < messageSize; i++) {
        transmitted[j++] = message[i];
    }
    for (i = messageSize; i < s; i++) {
        transmitted[j++] = 0;
    }

    // Perform XOR division for CRC calculation
    for (i = 0; i < messageSize; i++) {
        if (transmitted[i] == 1) {  // If the first bit of transmitted is 1, perform XOR with the generator
            for (j = 0; j < generatorSize; j++) {
                transmitted[i + j] ^= generator[j];  // XOR the bits with the generator
            }
        }
    }


    // Store the remainder after division
    for (i = 0; i < r; i++) {
        remainder[i] = transmitted[messageSize + i];
    }

    // Print the remainder (CRC)
    printf("\nRemainder: ");
    for (i = 0; i < r; i++) {
        printf("%d", remainder[i]);
    }

    // Append the remainder to the original message to get the transmitted data
    for (i = 0; i < messageSize; i++) {
        transmittedData[i] = message[i];
    }
    for (i = 0; i < r; i++) {
        transmittedData[messageSize + i] = remainder[i];
    }

    // Print the transmitted data (original message + remainder)
    printf("\nTransmitted data: ");
    for (i = 0; i < s; i++) {
        printf("%d", transmittedData[i]);
    }

    return 0;
}
```

## Token Bucket

```
#include <stdio.h>

int main() {
    int bucketSize, tokenRate, outputRate, simulationTime;
    int tokens = 0, time, incomingPackets, packetsSent = 0, packetsDropped = 0;

    // Input parameters
    printf("Enter the bucket size (maximum tokens): ");
    scanf("%d", &bucketSize);

    printf("Enter the token generation rate (tokens added per time unit): ");
    scanf("%d", &tokenRate);

    printf("Enter the output rate (packets sent to network per time unit): ");
    scanf("%d", &outputRate);

    printf("Enter the total simulation time (in time units): ");
    scanf("%d", &simulationTime);

    // Output header
    printf("\nTime\tPackets Sent by Host\tBucket Tokens\tTotal Packets Sent\tPackets Dropped\n");

    // Simulation loop for each time unit
    for (time = 1; time <= simulationTime; time++) {
        // Add tokens at the token generation rate
        tokens += tokenRate;

        // Ensure tokens do not exceed the bucket size
        if (tokens > bucketSize) {
            tokens = bucketSize;
        }

        // Input for incoming packets
        printf("\nEnter the number of packets sent by the host at time %d: ", time);
        scanf("%d", &incomingPackets);

        // Output time and incoming packets
        printf("%d\t%d\t\t\t", time, incomingPackets);

        // If enough tokens are available, send packets
        if (incomingPackets <= tokens) {
            packetsSent += incomingPackets; // Sent packets
            tokens -= incomingPackets;      // Reduce the tokens by the number of packets sent
        } else {
            packetsSent += tokens;  // Send as many packets as tokens are available
            packetsDropped += (incomingPackets - tokens);  // Drop the rest of the packets
            tokens = 0;  // No tokens left after sending the packets
        }

        // Output the current bucket status, total packets sent, and dropped packets
        printf("%d\t\t\t%d\t\t\t%d\n", tokens, packetsSent, packetsDropped);
    }

    return 0;
}
```

Leaky Bucket

```
#include <stdio.h>

void leakybucket(int bucketsize, int outgoingrate) {
  int time = 0, buffer = 0, packetsize;
  printf("Enter the packet size for each time interval (-1 to stop):\n");
  while (1) {
    printf("For time %d: ", time);
    scanf("%d", &packetsize);
    if (packetsize == -1) {
      break;
    }
    if (packetsize <= (bucketsize - buffer)) {
      buffer += packetsize;
      printf("Packet accepted\n");
    } else {
      printf("Packet rejected\n");
    }
    if (buffer > outgoingrate) {
      buffer -= outgoingrate;
    } else {
      buffer = 0;
    }
    printf("Current buffer size is %d\n", buffer);
    time++;
  }
}

int main() {
  int bucketsize, outgoingrate;
  printf("Enter bucket size: ");
  scanf("%d", &bucketsize);
  printf("Enter outgoing rate: ");
  scanf("%d", &outgoingrate);
  leakybucket(bucketsize, outgoingrate);
  return 0;
}
```

# IP Fragmentation

```
#include <stdio.h>
#include <stdlib.h>

void fragmentIPDatagram(int packetSeqNum, int datagramSize, int mtu, int DF) {
    int headerSize = 20;  // Standard IP header size
    int fragmentDataSize = mtu - headerSize;  // Data size for each fragment (MTU - header size)
    int totalData = datagramSize - headerSize;  // Total data size (excluding header)
    int numFragments = (totalData + fragmentDataSize - 1) / fragmentDataSize;  // Number of fragments required
    int offset = 0;  // Initial offset for fragmentation

    // Printing packet sequence number
    printf("\nPacket Sequence Number: %d\n", packetSeqNum);
    printf("Frag No.\tDF\tMF\tFragment Offset\tFragment Size\n");

    // If Don't Fragment (DF) flag is set and the datagram is larger than MTU, fragmentation is not allowed
    if (DF && datagramSize > mtu) {
        printf("Fragmentation not allowed (DF = 1). Packet too large for MTU.\n");
        return;
    }

    // Fragmenting the datagram into smaller fragments
    for (int i = 1; i <= numFragments; i++) {
        int currentDataSize;

        // Calculate the current fragment's data size
        if (totalData > fragmentDataSize) {
            currentDataSize = fragmentDataSize;
        } else {
            currentDataSize = totalData;
        }

        // Print fragment details
        printf("%d\t\t%d\t%d\t%d\t\t%d\n", i,
               DF,
               (i == numFragments) ? 0 : 1,  // MF = 1 for all fragments except the last one
               offset,
               currentDataSize);

        // Update totalData and offset
        totalData -= currentDataSize;  // Reduce total data size
        offset += currentDataSize / 8;  // Increase offset by fragment size in 8-byte units
    }
}

int main() {
    int packetSeqNum, datagramSize, mtu, DF;

    // Input values
    printf("Enter the packet sequence number: ");
    scanf("%d", &packetSeqNum);

    printf("Enter the total size of the IP datagram (in bytes): ");
    scanf("%d", &datagramSize);

    printf("Enter the Maximum Transmission Unit (MTU) (in bytes): ");
    scanf("%d", &mtu);

    printf("Enter the Don't Fragment (DF) flag (0 or 1): ");
    scanf("%d", &DF);

    // Check if datagram size is valid
    if (datagramSize <= 20) {
        printf("Error: Datagram size must be greater than the header size (20 bytes).\n");
        return 1;
    }

    // Check if MTU is valid
    if (mtu <= 20) {
        printf("Error: MTU must be greater than the header size (20 bytes).\n");
        return 1;
    }

    // Fragment the IP datagram
    fragmentIPDatagram(packetSeqNum, datagramSize, mtu, DF);

    return 0;
}
```

DVR

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MAX 50
#define INF 9999

int main() {
    int n, i, j, k;
    char router[MAX][MAX];
    int dist[MAX][MAX], nexthop[MAX][MAX];

    // Input number of nodes
    printf("Enter the number of nodes: ");
    scanf("%d", &n);

    // Input the distances between the nodes
    printf("Enter the distances between nodes (use -1 for no path):\n");
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if(i == j) {
                dist[i][j] = 0;
                continue;
            }

            printf("dist[%d][%d]: ", i, j);
            scanf("%d", &dist[i][j]);

            // If no path, set distance to INF
            if(dist[i][j] == -1) {
                dist[i][j] = INF;
            }
        }
    }

    // Input the names of the routers
    printf("Enter the names of the nodes:\n");
    for(i = 0; i < n; i++) {
        scanf("%s", router[i]);
    }

    // Initialize nexthop matrix
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            nexthop[i][j] = (dist[i][j] != INF && i != j) ? j : -1;
        }
    }

    // Implementing Floyd-Warshall algorithm to find the shortest path
    for(k = 0; k < n; k++) {
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                if(dist[i][k] != INF && dist[k][j] != INF && dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    nexthop[i][j] = nexthop[i][k];
                }
            }
        }
    }

    // Display routing table for each node
    for(i = 0; i < n; i++) {
        printf("\nRouting table for node: %s\n", router[i]);
        printf("Destination\tDistance\tNextHop\n");
        for(j = 0; j < n; j++) {
            if(i == j) {
                continue;
            }
            if(dist[i][j] == INF) {
                printf("%s\t\tINF\t\t-\n", router[j]);
            } else {
                printf("%s\t\t%d\t\t%s\n", router[j], dist[i][j], router[nexthop[i][j]]);
            }
        }
    }

    return 0;}
```

Dijikstra’s

```
#include<stdio.h>
#include<stdlib.h>

#define MAX 50
#define INF 9999

int main() {
    int n, src, dest, i, j;
    int dist[MAX][MAX], cost[MAX], visited[MAX], parent[MAX];

    // Input the number of nodes
    printf("Enter the number of nodes: ");
    scanf("%d", &n);

    // Input the distance matrix (adjacency matrix for the graph)
    printf("Enter the distance matrix (if no connection, enter -1):\n");
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if(i == j) {
                dist[i][j] = 0;  // Distance to itself is 0
                continue;
            }

            // For symmetric graph (undirected)
            if(j < i) {
                dist[i][j] = dist[j][i];
                continue;
            }

            // Input for distance matrix
            printf("dist[%d][%d]: ", i, j);
            scanf("%d", &dist[i][j]);

            // If no path between nodes, set the distance to INF
            if(dist[i][j] == -1) {
                dist[i][j] = INF;
            }
        }
    }

    // Input the source and destination nodes
    printf("\nEnter the source node (0 to %d): ", n - 1);
    scanf("%d", &src);
    printf("Enter the destination node (0 to %d): ", n - 1);
    scanf("%d", &dest);

    // Initialize cost, visited and parent arrays
    for(i = 0; i < n; i++) {
        cost[i] = INF;
        visited[i] = 0;
        parent[i] = -1;
    }

    // The cost to reach the source node is 0
    cost[src] = 0;

    // Dijkstra's Algorithm to find the shortest path
    for(i = 0; i < n; i++) {
        int min = INF, min_index = -1;

        // Find the unvisited node with the smallest cost
        for(j = 0; j < n; j++) {
            if(!visited[j] && cost[j] < min) {
                min = cost[j];
                min_index = j;
            }
        }

        // If no node was found, break (all remaining nodes are unreachable)
        if(min_index == -1) {
            break;
        }

        // Mark the current node as visited
        visited[min_index] = 1;

        // Update the cost for each neighboring node
        for(j = 0; j < n; j++) {
            if(!visited[j] && dist[min_index][j] != INF) {
                // If a shorter path is found, update the cost and parent
                if(cost[min_index] + dist[min_index][j] < cost[j]) {
                    cost[j] = cost[min_index] + dist[min_index][j];
                    parent[j] = min_index;
                }
            }
        }
    }

    // To store the path from source to destination
    int path[MAX], p = 0;
    int temp = dest;

    printf("\nThe shortest path: ");
    if(cost[dest] == INF) {
        printf("No path exists between source and destination.\n");
    } else {
        // Reconstruct the path from destination to source
        while(temp != -1) {
            path[p++] = temp;
            temp = parent[temp];
        }

        // Print the path in correct order (from source to destination)
        for(i = p - 1; i >= 0; i--) {
            printf("%d ", path[i]);
        }
        printf("\nTotal cost: %d\n", cost[dest]);
    }

    return 0;
}
```