Bash programs :

```
#factorial of a number
ans=1
echo "Enter i:"
read i
while [ $i -gt 0 ]
do
    ans=$(($ans * $i))
    i=$(($i - 1))
done
echo $ans
--------------------------------------------------------------
#sum of first n natural numbers
ans=0
echo "Enter n:"
read n
while [ $n -gt 0 ]
do
  ans=$(($ans + $n))
  n=$(($n - 1))
done
echo $ans
--------------------------------------------------------------
#sum of all digits in a number
ans=0
echo "Enter n:"
read n
while [ $n -gt 0 ]
do
  rem=$(($n % 10))
  ans=$(($ans + $rem))
  n=$(($n / 10))
done
echo $ans

--------------------------------------------------------------
#palindrome
echo 'Enter n:'
read n
m=$n
```

```
rev=0
while [ $n -gt 0 ]
do
  rem=$(($n % 10))
  rev=$(($rev * 10 + $rem))
  n=$(($n / 10))
done

if [ $m -eq $rev ]
then
    echo "It is a palindrome"
else
    echo "It is not a palindrome"
fi
----------------------------------------------------------------------
#gcd of two numbers
function gcd(){
  x=$1
  y=$2
  while [ $y -ne 0 ]
  do
    rem=$(($x % $y))
    x=$y
    y=$rem
  done

  echo $x
}

gcd 12 6
----------------------------------------------------------------------
#fibonacci
function fib(){
  a=0
  b=1
  num=$1
  if [ $num -eq 0 ]
  then
    echo $a
  elif [ $num -eq 1 ]
  then
    echo $a
    echo $b
  else
```

```
    echo $a
    echo $b
    num=$(($num - 2))
    while [ $num -gt 0 ]
    do
      c=$(($a + $b))
      echo $c
      a=$b
      b=$c
      num=$(($num - 1))
    done
  fi

}

echo "Enter n:"
read n
fib $n

----------------------------------------------------------------------
#to print primes upto that number
function printPrimes() {
  num=$1
  nn=2

  while [ $nn -le $num ]
  do
    ct=0
    n=$nn
    i=2

    while [ $i -le $n ]
    do
      rem=$(($n % $i))

      if [ $rem -eq 0 ]
      then
        ct=$(($ct + 1))
      fi

      i=$(($i + 1))
    done

    if [ $ct -eq 1 ]
    then
      echo $n
    fi

    nn=$(($nn + 1))
  done
}

echo "Enter n:"
read n
printPrimes $n


--------------------------------------------------------------
#armstrong numbers upto a given number
echo "Enter n:"
read n
i=1
while [ $i -le $n ]
do
 sum=0

 temp=$i
 count=0
 while [ $temp -ne 0 ]
 do
 count=` expr $count + 1 `
 temp=` expr $temp / 10 `
 done
 temp=$i
 while [ $temp -ne 0 ]
 do
 rem=` expr $temp % 10 `
rem=$(($rem ** $count))
 sum=` expr $sum + $rem `
 temp=` expr $temp / 10 `

 done
 if [ $sum -eq $i ]
 then
 echo $i
 fi
 i=` expr $i + 1 `
done
```

```
----------------------------------------------------------------
#krishnamurthy numbers upto n
echo "Enter n:"
read n
i=1
while [ $i -le $n ]
do
sum=0
 temp=$i
 while [ $temp -ne 0 ]
 do
 rem=` expr $temp % 10 `
 fact=1
 k=1
 while [ $k -le $rem ]
 do
 fact=$(($fact * $k))
 k=` expr $k + 1 `
 done
 sum=` expr $sum + $fact `
 temp=` expr $temp / 10 `
 done
 if [ $sum -eq $i ]
 then
 echo $i
 fi
 i=` expr $i + 1 `
done
```

System calls :

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {
        execlp("/bin/ls","ls",NULL);
    }
    else {
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

Messages Queues :

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#define MAX_TEXT 512 //maximum length of the message that can be sent allowed
struct my_msg{
long int msg_type;
char some_text[MAX_TEXT];
};
int main()
{
int running=1;
int msgid;
struct my_msg some_data;
char buffer[50]; //array to store user input
msgid=msgget((key_t)14534,0666|IPC_CREAT);
if (msgid == -1) // -1 means the message queue is not created
{
printf("Error in creating queue\n");
exit(0);
}
while(running)
{
printf("Enter some text:\n");
fgets(buffer,50,stdin);
some_data.msg_type=1;
strcpy(some_data.some_text,buffer);
if(msgsnd(msgid,(void *)&some_data, MAX_TEXT,0)==-1) // msgsnd returns -1 if
the message is not sent
{
printf("Msg not sent\n");
}
if(strncmp(buffer,"end",3)==0)
{
running=0;
}
}
}
```

Read program :

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
struct my_msg{
long int msg_type;
char some_text[BUFSIZ];
};
int main()
{
int running=1;
int msgid;
struct my_msg some_data;
long int msg_to_rec=0;
msgid=msgget((key_t)12345,0666|IPC_CREAT);
while(running)
{
```

```
msgrcv(msgid,(void *)&some_data,BUFSIZ,msg_to_rec,0);
printf("Data received: %s\n",some_data.some_text);
if(strncmp(some_data.some_text,"end",3)==0)
{
running=0;
}
}
msgctl(msgid,IPC_RMID,0);
}
```

Shared Memory :

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0);
printf("Process attached at %p\n",shared_memory);
printf("Enter some data to write to shared memory\n");
read(0,buff,100);
strcpy(shared_memory,buff);
printf("You wrote : %s\n",(char *)shared_memory);
}
```

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

```
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

Pipes :

```
#include<stdio.h>
#include<unistd.h>
int main() {
int pipefds[2];
int returnstatus;
char writemessages[2][20]={"Hi", "Hello"};
char readmessage[20];
returnstatus = pipe(pipefds);
if (returnstatus == -1) {
printf("Unable to create pipe\n");
return 1;
}
printf("Writing to pipe - Message 1 is %s\n", writemessages[0]);
write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Reading from pipe – Message 1 is %s\n", readmessage);
printf("Writing to pipe - Message 2 is %s\n", writemessages[0]);
write(pipefds[1], writemessages[1], sizeof(writemessages[0]));
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Reading from pipe – Message 2 is %s\n", readmessage);
return 0;
}
```

Fcfs :

```
#include<stdio.h>
 int main()
{
    int n,bt[30],wait_t[30],turn_ar_t[30],av_wt_t=0,avturn_ar_t=0,i,j;
    printf("Please enter the total number of processes(maximum 30):");  // the maximum process that be used to calculate is specified.
    scanf("%d",&n);
    printf("\nEnter The Process Burst Timen");
    for(i=0;i<n;i++)  // burst time for every process will be taken as input
    {
        printf("P[%d]:",i+1);
        scanf("%d",&bt[i]);
    }
    wait_t[0]=0;
    for(i=1;i<n;i++)
    {
        wait_t[i]=0;
        for(j=0;j<i;j++)
            wait_t[i]+=bt[j];
    }
    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        turn_ar_t[i]=bt[i]+wait_t[i];
        av_wt_t+=wait_t[i];
        avturn_ar_t+=turn_ar_t[i];
        printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",i+1,bt[i],wait_t[i],turn_ar_t[i]);
    }
    av_wt_t/=i;
    avturn_ar_t/=i;  // average calculation is done here
    printf("\nAverage Waiting Time:%d",av_wt_t);
    printf("\nAverage Turnaround Time:%d",avturn_ar_t);
    return 0;
}
```

Sjf non preemptive :

```
#include <stdio.h>
int findShortestProcess(int arrivalTime[], int burstTime[], int timeCounter, int n)
{
    int pid = -1, lowestBurstTime = 10000;
    for (int i = 0; i < n; i++)
    {
        if (arrivalTime[i] <= timeCounter && burstTime[i] >= 0 && burstTime[i] < lowestBurstTime)
        {
            pid = i;
            lowestBurstTime = burstTime[i];
        }
    }
    return pid;
}

float findAvg(int arr[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += arr[i];
    }
    return (float)sum / n;
}
int main()
{
    // printf("Hello World");
    int n;
    printf("Enter the number of process :");
    scanf("%d", &n);
    int burstTime[n], OriginalBurstTime[n], arrivalTime[n], waitingTime[n], turnAroundTime[n],
processOrder[n];

    for (int i = 0; i < n; i++)
    {
        printf("Enter the arrival time of process %d : ", i + 1);
        scanf("%d", &arrivalTime[i]);
        printf("Enter the burst time of process %d : ", i + 1);
        scanf("%d", &burstTime[i]);
        OriginalBurstTime[i] = burstTime[i];
    }

    int timeCounter = 0, processesCompleted = 0;

    while (processesCompleted < n)
    {
        int pid = findShortestProcess(arrivalTime, burstTime, timeCounter, n);
        if (pid == -1)
        {
            timeCounter++;
            continue;
        }
        processOrder[processesCompleted] = pid;
        waitingTime[pid] = timeCounter - arrivalTime[pid];
        turnAroundTime[pid] = waitingTime[pid] + burstTime[pid];
        timeCounter += burstTime[pid];
        burstTime[pid] = -1;
        processesCompleted++;
    }

    printf("\nprocess\tarrivalTime\tburstTime\twaitingTime\tTurnAroundTime");
    for (int i = 0; i < n; i++)
    {
        printf("\n%d\t%d\t%d\t%d\t%d", processOrder[i] + 1, arrivalTime[processOrder[i]],
OriginalBurstTime[processOrder[i]], waitingTime[processOrder[i]],
turnAroundTime[processOrder[i]]);
    }

    printf("\n\nAverage waiting Time : %f", findAvg(waitingTime, n));
    printf("\n\nAverage turnAroundTime Time : %f", findAvg(turnAroundTime, n));

    return 0;
}
```

Sjf preemptive :

```
#include <stdio.h>
int findShortestProcess(int arrivalTime[], int burstTime[], int timeCounter, int n)
{
    int pid = -1, lowestBurstTime = 10000;
    for (int i = 0; i < n; i++)
    {
        if (arrivalTime[i] <= timeCounter && burstTime[i] >= 0 && burstTime[i] < lowestBurstTime)
        {
            pid = i;
            lowestBurstTime = burstTime[i];
        }
    }
    return pid;
}

float findAvg(int arr[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += arr[i];
    }
    return (float)sum / n;
}
int main()
{
    // printf("Hello World");
    int n;
    printf("Enter the number of process :");
    scanf("%d", &n);
    int burstTime[n], OriginalBurstTime[n], arrivalTime[n], waitingTime[n], turnAroundTime[n],
processOrder[n];

    for (int i = 0; i < n; i++)
    {
        printf("Enter the arrival time of process %d : ", i + 1);
        scanf("%d", &arrivalTime[i]);
        printf("Enter the burst time of process %d : ", i + 1);
        scanf("%d", &burstTime[i]);
        OriginalBurstTime[i] = burstTime[i];
    }

    int timeCounter = 0, processesCompleted = 0;

    while (processesCompleted < n)
    {
        int pid = findShortestProcess(arrivalTime, burstTime, timeCounter, n);
        printf("%d", pid + 1);
        if (pid == -1)
        {
            timeCounter++;
            continue;
        }
        burstTime[pid]--;
        if (burstTime[pid] == 0)
        {
            processOrder[processesCompleted] = pid;
            turnAroundTime[pid] = timeCounter + 1 - arrivalTime[pid];
            waitingTime[pid] = timeCounter - arrivalTime[pid] - OriginalBurstTime[pid] + 1;
            processesCompleted++;
            burstTime[pid] = -1;
        }

        timeCounter++;
    }

    printf("\nprocess\tarrivalTime\tburstTime\twaitingTime\tTurnAroundTime");
    for (int i = 0; i < n; i++)
    {
        printf("\n%d\t%d\t%d\t%d\t%d", processOrder[i] + 1, arrivalTime[processOrder[i]],
OriginalBurstTime[processOrder[i]], waitingTime[processOrder[i]],
turnAroundTime[processOrder[i]]);
    }

    printf("\n\nAverage waiting Time : %f", findAvg(waitingTime, n));
    printf("\n\nAverage turnAroundTime Time : %f", findAvg(turnAroundTime, n));

    return 0;
}
```

Round robin:

```
#include <stdio.h>

int at[100], bt[100], rt[100], temp[100];
float wait_time = 0, turn_time = 0;
```

```
int main()
{
    int c, j, n, time, r, flag = 0, time_q, ltt, i, wt = 0;

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

    r = n;

    // Input arrival and burst time for each process
    for (c = 0; c < n; c++)
    {
        printf("Enter arrival time of P%d: ", c + 1);
        scanf("%d", &at[c]);
        printf("Enter burst time of P%d: ", c + 1);
        scanf("%d", &bt[c]);
        rt[c] = bt[c];
        temp[c] = bt[c];
        printf("\n");
    }

    // Input time quantum
    printf("Enter time quantum: ");
    scanf("%d", &time_q);

    printf("\n\n\tProcess\tArrival Time\tTurnaround Time\tWaiting Time\n\n");

    // Perform round robin scheduling
    for (time = 0, c = 0; r != 0;)
    {
        if (rt[c] <= time_q && rt[c] > 0)
        {
            time = time + rt[c];
            rt[c] = 0;
            flag = 1;
        }
        else if (rt[c] > 0)
        {
            rt[c] = rt[c] - time_q;
            time = time + time_q;
        }

        if (rt[c] == 0 && flag == 1)
        {
            wt = 0;
            wt = time - at[c] - bt[c];
            r--;

            // Print the turnaround time and waiting time for each process
            printf("\tP%d\t%d\t%d\t%d\n", c + 1, at[c], time - at[c], wt);

            ltt = time - at[c];
            wait_time = wait_time + time - at[c] - bt[c];
            turn_time = turn_time + time - at[c];
            flag = 0;
        }

        if (c == n - 1)
            c = 0;
        else if (at[c + 1] <= time)
            c++;
        else
            c = 0;
    }

    j = 0;

    printf("\n\n\n");
    printf("Gantt Chart ");
    printf("\n\n\n");
    printf("\t");

    // Print the Gantt Chart
    for (i = at[0]; i < time;)
    {
        if (bt[j] >= time_q)
        {
            printf("P%d\t", j + 1);
            i += time_q;
            bt[j] = bt[j] - time_q;
        }
        else if (bt[j] > 0)
        {
            printf("P%d\t", j + 1);
            i += bt[j];
            bt[j] = 0;
        }
```

```
        j++;

        if (j >= n)
        {
            j = 0;
        }
    }

    printf("\n");
    j = 0;
    printf("\t");

    // Print the time sequence
    for (i = at[0]; i < time;)
    {
        if (temp[j] >= time_q)
        {
            printf("%d\t", i + time_q);
            i += time_q;
            temp[j] = temp[j] - time_q;
        }
        else if (temp[j] > 0)
        {
            printf("%d\t", i + temp[j]);
            i += temp[j];
            temp[j] = 0;
        }
        j++;

        if (j >= n)
        {
            j = 0;
        }
    }

    printf("\n\n\n");
    printf("\nAverage waiting time = %f\n", wait_time / n);
    printf("Average turnaround time = %f\n", turn_time / n);

    return 0;
}
```

Bankers:

```
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5;                         // Number of processes
    m = 3;                          // Number of resources
    int alloc[5][3] = {{0, 1, 0},  // P0 // Allocation Matrix
                      {2, 0, 0},  // P1
                      {3, 0, 2},  // P2
                      {2, 1, 1},  // P3
                      {0, 0, 2}}; // P4

    int max[5][3] = {{7, 5, 3},  // P0 // MAX Matrix
                    {3, 2, 2},  // P1
                    {9, 0, 2},  // P2
                    {2, 2, 2},  // P3
                    {4, 3, 3}}; // P4

    int avail[3] = {3, 3, 2}; // Available Resources

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++)
    {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++)
    {
        for (i = 0; i < n; i++)
        {
            if (f[i] == 0)
            {
                int flag = 0;
                for (j = 0; j < m; j++)
                {
                    if (need[i][j] > avail[j])
                    {
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0)
                {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
    int flag = 1;
    for ( i = 0; i < n; i++)
    {
        if (f[i] == 0)
        {
            flag = 0;
            printf("The following system is not safe");
            break;
        }
    }
    if (flag == 1)
    {
        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
            printf(" P%d ->", ans[i]);
        printf(" P%d", ans[n - 1]);
    }
   return(0);
}
```

Sockets :

Server :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>


#define PORT 8080
#define MAX_BUFFER_SIZE 1024


int main()
{
    int server_socket, new_socket, valread;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[MAX_BUFFER_SIZE] = {0};
    const char *hello = "Hello from server";


    // Creating socket file descriptor
    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }


    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt)))
    {
        perror("Setsockopt failed");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    // Forcefully attaching socket to the port 8080
    if (bind(server_socket, (struct sockaddr *)&address, sizeof(address))
< 0)
    {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_socket, 3) < 0)
    {
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }
    if ((new_socket = accept(server_socket, (struct sockaddr *)&address,
(socklen_t *)&addrlen)) < 0)
    {
        perror("Accept failed");
        exit(EXIT_FAILURE);
    }


    valread = read(new_socket, buffer, MAX_BUFFER_SIZE);
    printf("Received message from client: %s\n", buffer);
    send(new_socket, hello, strlen(hello), 0);
    printf("Hello message sent to client\n");


    return 0;
}
```

Client :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>

#define PORT 8080
#define MAX_BUFFER_SIZE 1024
```

```
int main()
{
    int client_socket;
    struct sockaddr_in serv_addr;
    char buffer[MAX_BUFFER_SIZE] = {0};
    const char *message = "Hello from client";

    // Creating socket file descriptor
    if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0)
    {
        perror("Invalid address/ Address not supported");
        exit(EXIT_FAILURE);
    }

    if (connect(client_socket, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        perror("Connection Failed");
        exit(EXIT_FAILURE);
    }

    send(client_socket, message, strlen(message), 0);
    printf("Message sent to server\n");
    read(client_socket, buffer, MAX_BUFFER_SIZE);
    printf("Received message from server: %s\n", buffer);

    return 0;
}
```