Assignment 1

IT851: Information and Systems Security Lab

**ANINDYA KUNDU**

IT, 8th Semester

ID 510817020

Repository:  
[github.com/meganindya/btech-assignments/information-and-systems-security/assg-1](https://github.com/meganindya/btech-assignments/tree/main/information-and-systems-security/assg-1)

08-03-2021

Part – A

Implement the following in Modular Arithmetic:

1. Additive inverse of a number

Source: A-1-additive-inverse.c

*#include* <stdio.h>

int mod\_inv\_add(int a, int m)

{

*return* m - (a % m);

}

int main(int argc, char \*argv[])

{

    int a, m;

    printf("\nCalculation of modular additive inverse\n--------\n");

    printf("Enter number (a): ");

    scanf("%d", &a);

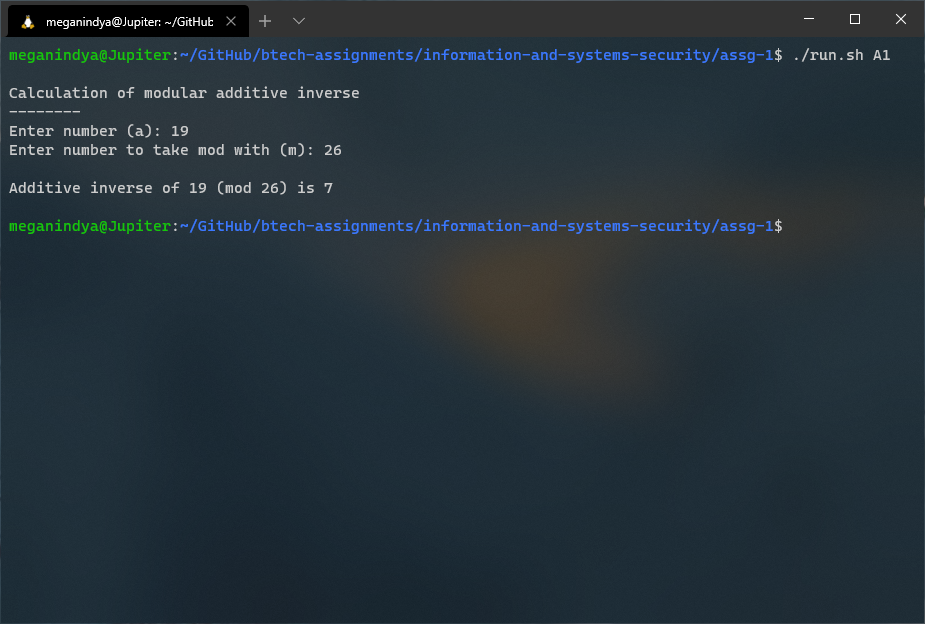
    printf("Enter number to take mod with (m): ");

    scanf("%d", &m);

    printf("\nAdditive inverse of %d (mod %d) is %d\n\n", a, m, mod\_inv\_add(a, m));

}

Sample run



1. Multiplicative inverse of a number

Source: A-2-multiplicative-inverse.c

*#include* <stdio.h>

*#include* <stdlib.h>

int mod\_inv\_mul(int a, int m)

{

*for* (int x = 1; x < m; x++)

    {

*if* (a < 0)

        {

*if* (((m - (abs(a) % m)) \* (x % m)) % m == 1)

            {

*return* x;

            }

        }

*else*

        {

*if* (((a % m) \* (x % m)) % m == 1)

            {

*return* x;

            }

        }

    }

*return* -1;

}

int main(int argc, char \*argv[])

{

    int a, m;

    printf("\nCalculation of modular multiplicative inverse\n--------\n");

    printf("Enter number (a): ");

    scanf("%d", &a);

    printf("Enter number to take mod with (m): ");

    scanf("%d", &m);

    int inv = mod\_inv\_mul(a, m);

*if* (inv == -1)

    {

        printf("\nMultiplicative inverse of %d (mod %d) does not exist\n\n", a, m);

    }

*else*

    {

printf(

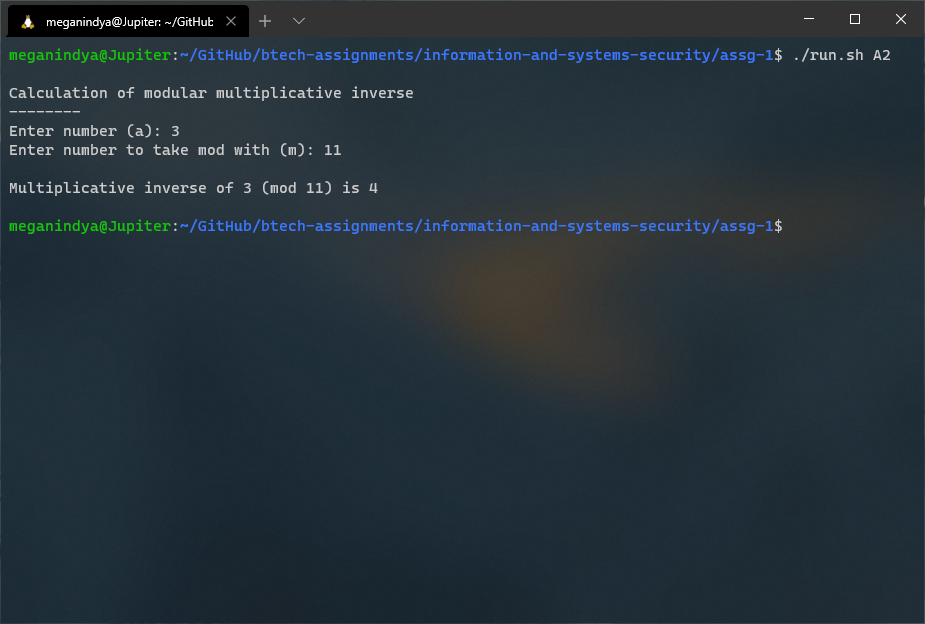
            "\nMultiplicative inverse of %d (mod %d) is %d\n\n", a, m, mod\_inv\_mul(a, m)

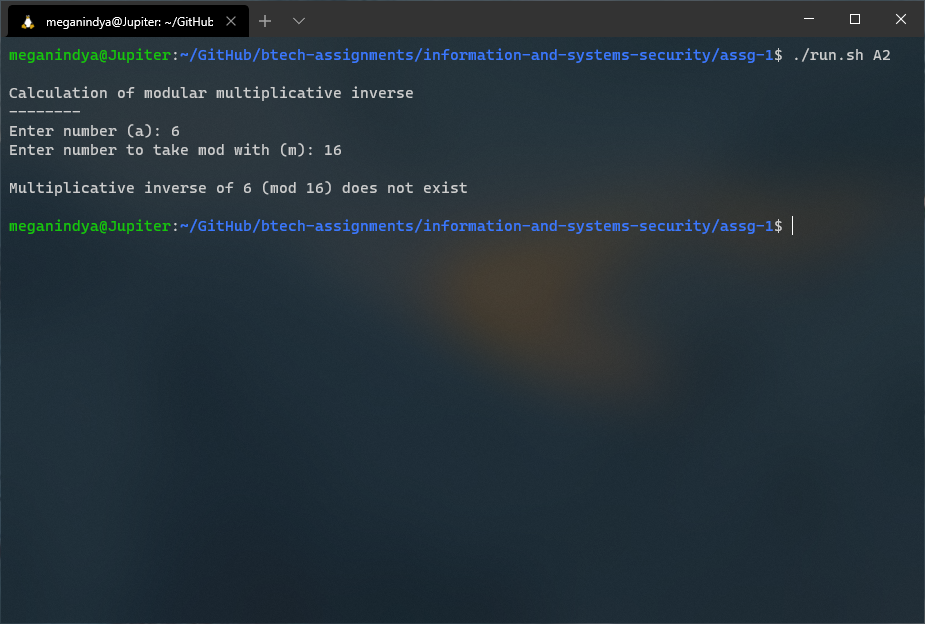
        );

    }

}

Sample run





1. Inverse of an m×m matrix with m ≤ 3

Source: A-3-matrix-inverse.c

*#include* <stdio.h>  *// printf, scanf*

*#include* <stdlib.h> *// abs*

*#include* <math.h>   *// pow*

int mod(int, int);

int mod\_inv\_mul(int, int);

int determinant\_mat(int, int \*\*);

void cofactor\_mat(int, int \*\*, int \*\*);

void transpose\_mat(int, int \*\*);

int main()

{

    printf("\nCalculation of modular matrix inverse\n--------\n");

    printf("Enter the order of the matrix: ");

*/\* Stores order of matrix. \*/*

    int ord\_mat;

    scanf("%d", &ord\_mat);

    printf("Enter the elements of the %d × %d matrix:\n", ord\_mat, ord\_mat);

*/\* Stores the matrix. \*/*

    int \*\*mat;

    mat = malloc((ord\_mat) \* sizeof \*mat);

*for* (int i = 0; i < ord\_mat; i++)

    {

        mat[i] = malloc(ord\_mat \* sizeof \*mat[i]);

    }

*for* (int i = 0; i < ord\_mat; i++)

    {

*for* (int j = 0; j < ord\_mat; j++)

        {

            scanf("%d", &mat[i][j]);

        }

    }

*/\* Stores the number to take modulus with. \*/*

    int m;

    printf("Enter the number to take mod with (m): ");

    scanf("%d", &m);

*/\* Stores the determinant of the matrix. \*/*

    int det = determinant\_mat(ord\_mat, mat);

*/\* Stores the modular multiplicative inverse of the determinant. \*/*

    int det\_inv = mod\_inv\_mul(det, m);

*if* (det\_inv == -1)

    {

        printf("\nInverse of entered matrix is not possible\n\n");

*return* 0;

    }

    int \*\*new\_mat;

    new\_mat = malloc((ord\_mat) \* sizeof \*new\_mat);

*for* (int i = 0; i < ord\_mat; i++)

    {

        new\_mat[i] = malloc(ord\_mat \* sizeof \*new\_mat[i]);

    }

    cofactor\_mat(ord\_mat, mat, new\_mat);

    transpose\_mat(ord\_mat, new\_mat);

*for* (int i = 0; i < ord\_mat; i++)

    {

*for* (int j = 0; j < ord\_mat; j++)

        {

            new\_mat[i][j] = mod(new\_mat[i][j] \* det\_inv, m);

        }

    }

    printf("\nMatrix is\n");

*for* (int i = 0; i < ord\_mat; i++)

    {

*for* (int j = 0; j < ord\_mat; j++)

        {

            printf("%3d ", mat[i][j]);

        }

        printf("\n");

    }

    printf("\nModular matrix inverse is\n");

*for* (int i = 0; i < ord\_mat; i++)

    {

*for* (int j = 0; j < ord\_mat; j++)

        {

            printf("%3d ", new\_mat[i][j]);

        }

        printf("\n");

    }

    printf("\nSide by side\n");

*for* (int i = 0; i < ord\_mat; i++)

    {

*for* (int j = 0; j < ord\_mat; j++)

        {

            printf("%3d ", mat[i][j]);

        }

        printf("    ");

*for* (int j = 0; j < ord\_mat; j++)

        {

            printf("%3d ", new\_mat[i][j]);

        }

        printf("\n");

    }

    printf("\nOn multiplication\n");

*for* (int i = 0; i < ord\_mat; i++)

    {

*for* (int j = 0; j < ord\_mat; j++)

        {

            int sum = 0;

*for* (int k = 0; k < ord\_mat; k++)

            {

                sum += mat[i][k] \* new\_mat[k][j];

            }

            printf("%3d ", sum);

        }

        printf("\n");

    }

    printf("\nTaking modulus with %d\n", m);

*for* (int i = 0; i < ord\_mat; i++)

    {

*for* (int j = 0; j < ord\_mat; j++)

        {

            int sum = 0;

*for* (int k = 0; k < ord\_mat; k++)

            {

                sum += mat[i][k] \* new\_mat[k][j];

            }

            printf("%3d ", mod(sum, m));

        }

        printf("\n");

    }

    printf("\n");

*for* (int i = 0; i < ord\_mat; i++)

    {

        free(new\_mat[i]);

    }

    free(new\_mat);

*for* (int i = 0; i < ord\_mat; i++)

    {

        free(mat[i]);

    }

    free(mat);

}

*/\**

*\* Returns the modulus of a number with another. Handles negative case.*

*\**

*\* a: number to take modulus of (can be any integer)*

*\* m: number to take modulus with (natural number)*

*\*/*

int mod(int a, int m)

{

*return* a < 0 ? m - (abs(a) % m) : a % m;

}

*/\**

*\* Calculates the modular multiplicative inverse of a number.*

*\**

*\* a: input number*

*\* m: modulus number*

*\**

*\* returns: multiplicative inverse of a (mod m)*

*\*/*

int mod\_inv\_mul(int a, int m)

{

*for* (int x = 1; x < m; x++)

    {

*if* ((mod(a, m) \* (x % m)) % m == 1)

        {

*return* x;

        }

    }

*return* -1;

}

*/\**

*\* Calculates the determinant of a matrix.*

*\**

*\* ord\_mat: order of the matrix*

*\* mat: supplied matrix*

*\**

*\* returns: determinant of mat*

*\*/*

int determinant\_mat(int ord\_mat, int \*\*mat)

{

*if* (ord\_mat == 1)

    {

*return* mat[0][0];

    }

    int det = 0;

*for* (int z = 0; z < ord\_mat; z++)

    {

        int sign = ((1 - (z & 1)) << 1) - 1;

        int \*\*sub\_mat;

        sub\_mat = malloc((ord\_mat - 1) \* sizeof \*sub\_mat);

*for* (int i = 0; i < ord\_mat - 1; i++)

        {

            sub\_mat[i] = malloc((ord\_mat - 1) \* sizeof \*sub\_mat[i]);

        }

        int kr = 0, kc = 0;

*for* (int r = 1; r < ord\_mat; r++)

        {

*for* (int c = 0; c < ord\_mat; c++)

            {

*if* (c != z)

                {

                    sub\_mat[kr][kc] = mat[r][c];

                    kc = (kc + 1) % ord\_mat;

                }

            }

            kr++;

        }

        det += sign \* mat[0][z] \* determinant\_mat(ord\_mat - 1, sub\_mat);

*for* (int i = 0; i < ord\_mat - 1; i++)

        {

            free(sub\_mat[i]);

        }

        free(sub\_mat);

    }

*return* det;

}

*/\**

*\* Calculates and fills the cofactors for a matrix.*

*\**

*\* ord\_mat: order of the matrix*

*\* mat: supplied matrix*

*\* cof\_mat: matrix to fill cofactors in*

*\*/*

void cofactor\_mat(int ord\_mat, int \*\*mat, int \*\*cof\_mat)

{

*for* (int r = 0; r < ord\_mat; r++)

    {

*for* (int c = 0; c < ord\_mat; c++)

        {

            int sign = ((1 - ((r + c) & 1)) << 1) - 1;

            int \*\*sub\_mat;

            sub\_mat = malloc((ord\_mat - 1) \* sizeof \*sub\_mat);

*for* (int i = 0; i < ord\_mat - 1; i++)

            {

                sub\_mat[i] = malloc((ord\_mat - 1) \* sizeof \*sub\_mat[i]);

            }

            int ki = 0, kj = 0;

*for* (int i = 0; i < ord\_mat; i++)

            {

*for* (int j = 0; j < ord\_mat; j++)

                {

*if* (i != r && j != c)

                    {

                        sub\_mat[ki][kj] = mat[i][j];

                        kj = (kj + 1) % (ord\_mat - 1);

                    }

                }

*if* (i != r)

                    ki++;

            }

            cof\_mat[r][c] = sign \* determinant\_mat(ord\_mat - 1, sub\_mat);

*for* (int i = 0; i < ord\_mat - 1; i++)

            {

                free(sub\_mat[i]);

            }

            free(sub\_mat);

        }

    }

}

*/\**

*\* Transposes a square matrix in place.*

*\**

*\* ord\_mat: order of the matrix*

*\* mat: supplied matrix*

*\*/*

void transpose\_mat(int ord\_mat, int \*\*mat)

{

*for* (int r = 0; r < ord\_mat; r++)

    {

*for* (int c = 0; c <= ((ord\_mat - 1) >> 1); c++)

        {

            int temp = mat[r][c];

            mat[r][c] = mat[c][r];

            mat[c][r] = temp;

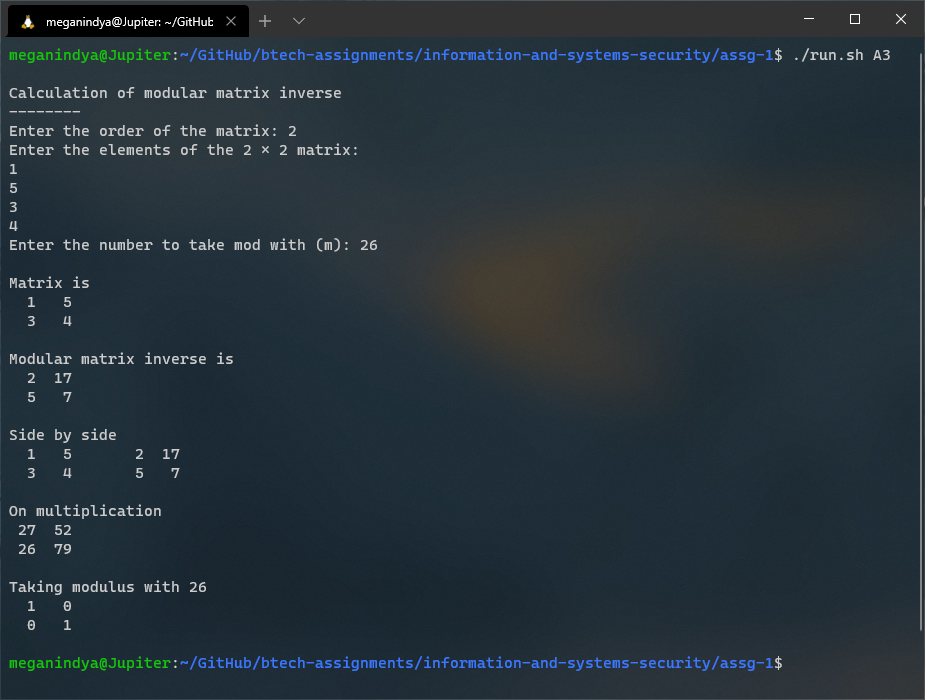
        }

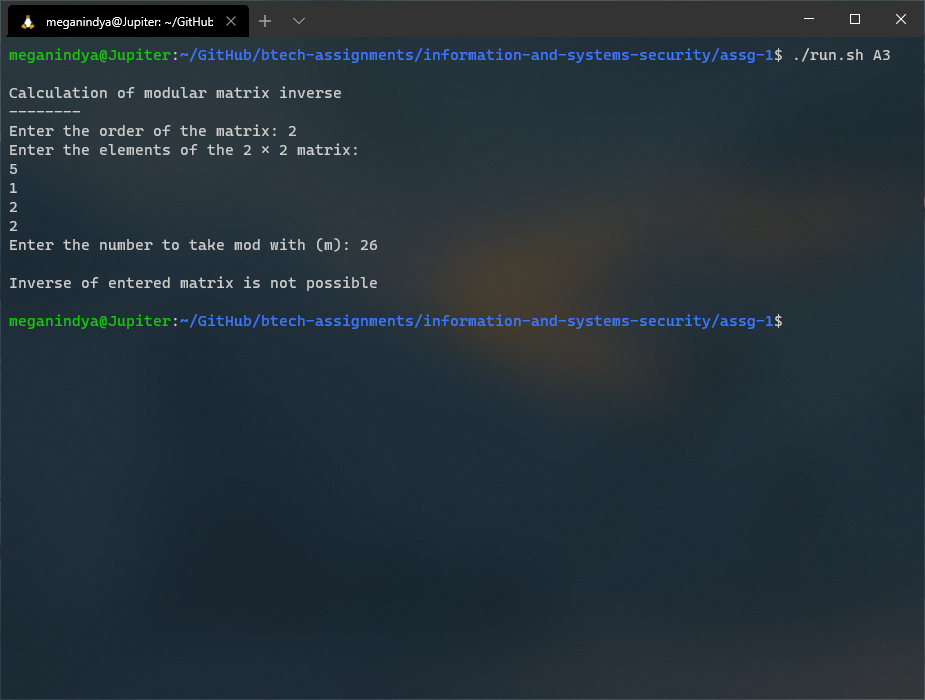
    }

}

[P.T.O.]

Sample run





1. Implement Diffie Hellman Key Exchange (DHKE) Protocol, with small integer values for testing.

Source: A-4-diffie-hellman.c

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <unistd.h>

*#include* <sys/types.h>

*#include* <string.h>

*#include* <sys/wait.h>

*#include* <math.h>

*/\**

*\* Returns value of a ^ b (mod P).*

*\*/*

long long int power(long long int a, long long int b, long long int P)

{

*return* b == 1 ? a : (((long long int)pow(a, b)) % P);

}

int main(int argc, char \*argv[])

{

    printf("\n");

    long long int P = 23; *// prime number P*

    printf("Prime Number (P): %lld\n", P);

    long long int G = 9; *// primitve root for P, G*

    printf("Primitive Root (G): %lld\n\n", G);

*// pipe 1 to send from parent to child*

    int fd1[2]; *// used to store two ends of pipe 1*

*// pipe 2 to send from child to parent*

    int fd2[2]; *// used to store two ends of pipe 2*

*if* (pipe(fd1) == -1 || pipe(fd2) == -1)

    {

        fprintf(stderr, "Pipe Failed");

*return* 1;

    }

    pid\_t pid = fork();

*if* (pid < 0)

    {

        fprintf(stderr, "fork Failed");

*return* 1;

    }

*// parent process*

*else* *if* (pid > 0)

    {

*// private key of parent*

        long long int a;

        FILE \*fp = fopen("A-4-a.txt", "r");

        fscanf(fp, "%lld", &a);

        fclose(fp);

        printf("Private key of process A (a): %lld\n", a);

*// generated key of parent*

        long long int x = power(G, a, P), y;

        printf("Generated public key from process A is x = G ^ a (mod P) = %lld\n", x);

*// close reading end of pipe 1*

        close(fd1[0]);

*// write x and close writing end of pipe 1*

        write(fd1[1], &x, sizeof(x));

        close(fd1[1]);

*// wait for child to send*

        wait(NULL);

*// read from child*

        read(fd2[0], &y, sizeof(y));

*// secret key for parent*

        long long int ka = power(y, a, P);

        printf("Shared secret key at A (ka): %lld\n", ka);

*// close reading end of pipe 2*

        close(fd2[0]);

        printf("\n");

*return* 0;

    }

*// child process*

*else*

    {

*// private key of child*

        long long int b;

        FILE \*fp = fopen("A-4-b.txt", "r");

        fscanf(fp, "%lld", &b);

        fclose(fp);

        printf("Private key of process B (b): %lld\n", b);

*// generated key of child*

        long long int x, y = power(G, b, P);

        printf("\nGenerated public key from process B is y = G ^ b (mod P) = %lld\n", y);

*// read from parent*

        read(fd1[0], &x, sizeof(x));

*// secret key for child*

        long long int kb = power(x, b, P);

        printf("\nShared secret key at B (kb): %lld\n", kb);

*// close both reading ends*

        close(fd1[0]);

        close(fd2[0]);

*// write y and close writing end of pipe 2*

        write(fd2[1], &y, sizeof(y));

        close(fd2[1]);

        exit(0);

    }

}

Source: A-4-a.txt

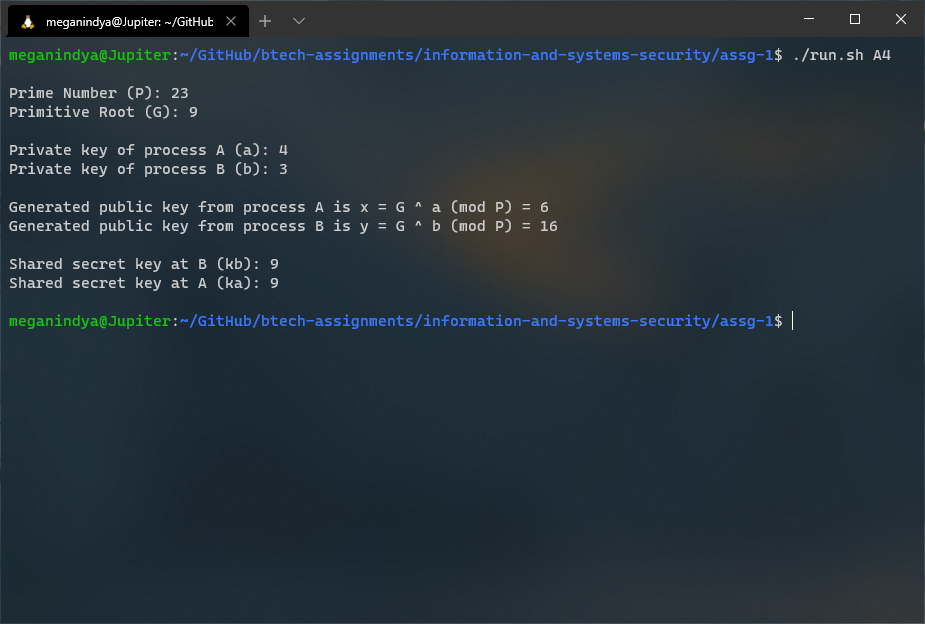
4

Source: A-4-b.txt

3

Sample run

Entry process is forked, then the two processes dynamically input data from files and interchange data via pipes.



Part – B

1. Implement the Euclidean Algorithm below, to find GCD of two numbers:

Source: B-1-euclid-gcd-algorithm.c

*#include* <stdio.h>

int gcd(int a, int b)

{

    printf("\nSteps:\n----\n");

*if* (a == 0)

    {

        printf("a: %d, b: %d\n", a, b);

*return* b;

    }

*while* (b != 0)

    {

        printf("a: %d, b: %d\n", a, b);

*if* (a > b)

            a = a - b;

*else*

            b = b - a;

    }

    printf("a: %d, b: %d\n", a, b);

*return* a;

}

int main(int argc, char \*argv[])

{

    int a, b;

    printf("\nCalculation of GCD by Euclid's Algorithm\n--------\n");

    printf("Enter number (a): ");

    scanf("%d", &a);

    printf("Enter number (b): ");

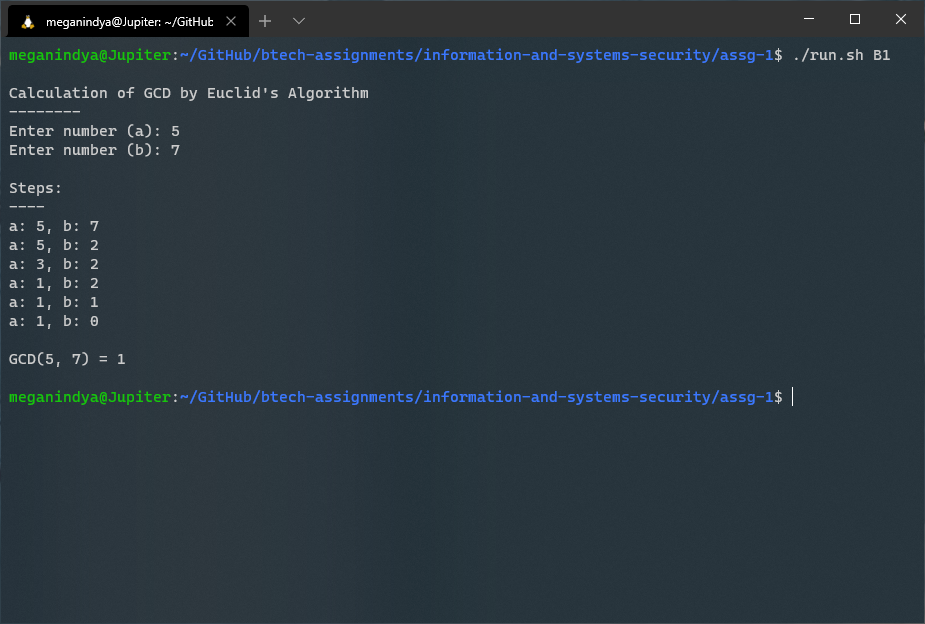
    scanf("%d", &b);

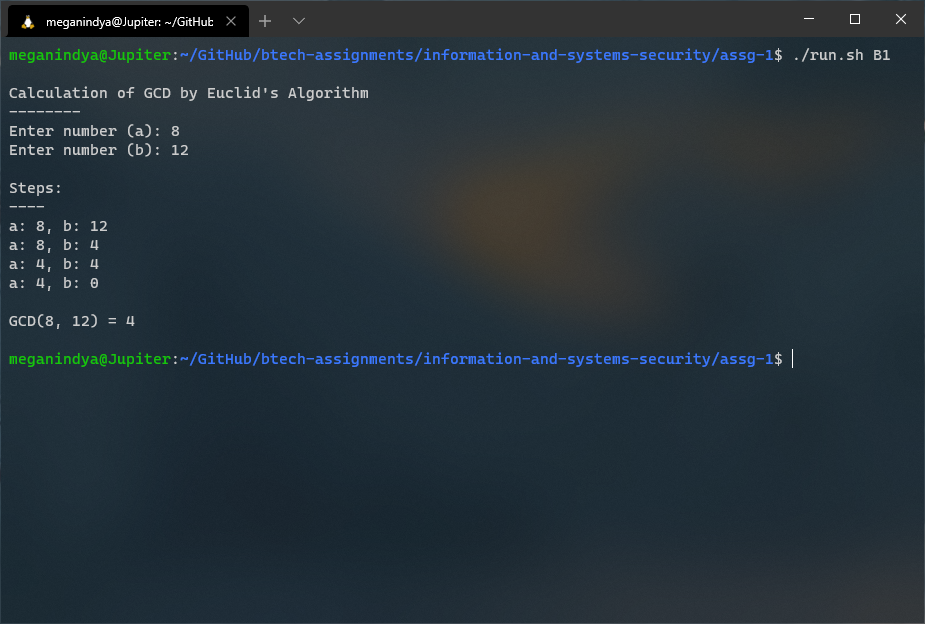
    printf("\nGCD(%d, %d) = %d\n\n", a, b, gcd(a, b));

}

[P.T.O.]

Sample run





1. Given two integers *a* and *b*, the following algorithm computes GCD (a, b) as well as b-1 mod a, when a and b are co-prime to each other. This is called the Extended Euclidean Algorithm.

Source: B-2-extended-euclid-algorithm.c

*#include* <stdio.h>

int gcd(int a, int b)

{

    printf("\nSteps:\n----\n");

    int t0 = 0;

    int t = 1;

    int s0 = 1;

    int s = 0;

    int q = a / b;

    int r = a - q \* b;

    printf(

        "t0 = %2d, t = %2d, s0 = %2d, s = %2d, q = %2d, r = %2d \n",

        t0, t, s0, s, q, r);

*while* (r > 0)

    {

        int temp = t0 - q \* t;

        t0 = t;

        t = temp;

        temp = s0 - q \* s;

        s0 = s;

        s = temp;

        a = b;

        b = r;

        q = a / b;

        r = a - q \* b;

        printf(

         "t0 = %2d, t = %2d, s0 = %2d, s = %2d, q = %2d, r = %2d \n", t0, t, s0, s, q, r);

    }

    r = b;

*return* r;

}

int main(int argc, char \*argv[])

{

    int a, b;

    printf("\nCalculation of GCD by Extended Euclidean Algorithm\n--------\n");

    printf("Enter number (a): ");

    scanf("%d", &a);

    printf("Enter number (b): ");

    scanf("%d", &b);

    printf("\nGCD(%d, %d) = %d\n\n", a, b, gcd(a, b));

}

Sample run

