Alexandria University
Faculty of Engineering
CSED 2025

# Discrete Lab 3
# Report

Submitted to:

**Eng. Omar Elsherif**

Faculty of Engineering Alex. University


Submitted by:

**Islam Yasser Mahmoud 20010312**

**Mkario Michel Azer 20011982**

Faculty of Engineering Alex. University


Dec. 2022

## Problem Statement:

- **Q1:**
  An implementation for sieve of Eratosthenes algorithm for finding all prime numbers up to any given limit.

- **Q2:**
  An implementation for Trial Division algorithm which finds the Prime factorization of a given number.

- **Q3:**
  An implementation for the extended Euclidean algorithm that finds the greatest common divisor d of two positive integers a and b. In addition, it outputs Bezout's coefficients s and t such that $d = s\,a + t\,b$. where d is the GCD of a, b

- **Q4:**
  An implementation for Chinese remainder theorem that takes as input m1, m2, m3, …., mn that are pairwise relatively prime and (a1, a2, …., an) and calculates x.

- **Q5:**
  An implementation for Miller's test (a probabilistic primality test) which takes a number and determine whether it is probably prime or composite.

## Used data structures:

- **Q1:**
  - Vector <bool> primes (number + 1 , true) >>>> A vector called primes with type bool with size number+1 and assign all values to true.
- **Q2:**
  - vector<bool> primes "which will store true in index i if i is prime"
  - vector<int> PrimeNumbers "which will contain all the prime numbers up to given limit"

- **Q3:**
  - vector <int> r "the remainder"
  - vector <int> q "the quotient"
  - vector <int> s "the first coefficient of Bezout's identity"
  - vector <int> t "the second coefficient of Bezout's identity"
- **Q4:**
  - vector <int> a "holds a1, a2, a3 …… an"
  - vector <int> m "holds m1, m2, m3 …… mn"
  - vector <int> M "holds M1, M2, M3 …… Mn . But the index 0 holds M which is the total product of m1 · m2 · m3 · · · · · ·mn"
  - vector <int> y "holds y1, y2, y3 …… yn"
- **Q5:**
  - No Data structures used just variables of type int, long long

## Algorithms used:

- **Q1:**
  - Get the number from the user.
  - Call the sieve function and pass the number on it.
  - Make a vector of type Boolean with size = number+1 and assign all values to True (except 0 , 1 will be false) supposing that all the numbers are primes.
  - Loop from 2:number+1 with increment +1
  - If the current number on the vector is assigned to True(meaning that it's a prime) and its square (i^2) <= the number.
    - Loop from (i^2): number with increment +i
      - Assign all the multiples of i in the vector to false.
  - Print all the primes.

- **Q2:**
  - Take the number to be factorized from the user and store it in variable number
  - Use the Sieve of Eratosthenes Algorithm to find all prime numbers until floor(sqrt(number)) ②
  - Store the prime numbers returned from Sieve algorithm in vector PrimeNumbers
  - Iterate on PrimeNumbers elements
  - Find the first prime number which divides the number
  - If found, then this prime is in the factorization of the number. so repeat step ② but instead of number perform it on **number/prime** recursively
  - if not found then the number is already a prime number, and it is the factorization of itself

- **Q3:**
  - Get the 2 numbers from the user.
  - Call the exEuclidean function and pass the numbers on it.
  - Make 4 vectors r,q,s,t of type int
  - Assign the initial values on the vectors.
  - If one of the 2 numbers was 0 the Bezouts will be with their initials and the GCD will be the other number not equal to 0.
  - If not: Loop till the remainder reaches 0 starting from counter 0
  - Assign the quotient and the remainder obtained from dividing the two numbers.
  - When the counter reaches 2, we will start evaluating the Bezouts and assigning them to their vectors.
  - A final step when the remainder reaches 0 before we exit the loop, we evaluate the last values of the Bezouts which are the final answer.
  - Print the results.
- **Q4:**
  - Get the number of equations from the user then $a_i$ followed by $m_i$
  - Compute M and M1, M2, M3 …… Mn

- Calling The Chinese_Remainder_Theorem() function Which in turn calls the Inverse function to compute $y_i$
- Compute $X = \sum a_i M_i y_i$ and returns it

- **Q5:**
  - Take the number to be tested for primality from the user and store it in variable n
  - Find two numbers m, k such that $n-1 = m \cdot 2^k$
  - Find the test variable $T = a^m \pmod n$ by fast modular exponentiation algorithm where a is random variable $1 < a < n-1$
  - If T = 1 or T = n-1 then the number n is probably prime
  - Else compute $T = T^2 \pmod n$ ②
  - If T = 1 then n is composite
  - If T = n-1 then n is probably prime
  - Else repeat step ② k times but in each time if T = 1 then n is composite else if T = n-1 then n is probably prime
  - If T never becomes 1 or n-1 in the previous k loops, then n is composite

## Important Assumptions and details:

- **Q1:**
  - Firstly, assume that all number less than the given number are primes then start to remove multiples of 2, 3, 5, 7 …... floor(sqrt(number))
  - Assume that the limit number can be prime number inclusively for example if the limit number is 5 then 2, 3, 5 are the primes returned from the Sieve algorithm

- **Q2:**
  - At first the Prime_Factorization() function is called, and the number is passed as a parameter then find the smallest prime number that divides the number then call the function Prime_Factorization() recursively but in this time number/prime is passed as a parameter
  - We can find the primes less than a certain number by the help of Sieve Algorithm
  - If the number is prime, then it is the Prime Factorization of itself

- **Q3:**
  - The extended Euclidean algo. uses one pass through the steps of the Euclidean algorithm to find Bezout coefficients of a and b.
  - We set s0 = 1, s1 = 0, t0 = 0, and t1 = 1 (note that s is always assigned to a and t is always assigned to b)

- **Q4:**
  - Bezout's coefficients (s, t) are declared as global variables
  - The Gcd() function finds also the bezout's coefficients by Extended Euclidean Algorithm

- The Inverse() function finds the inverse of any Linear congruence by the help of Gcd() function and Extended Euclidean Algorithm where the s Bezout's Identity is the Required Inverse

- **Q5:**
  - Miller's Test is probabilistic so the number can be composite, and the test says that it is prime but this happen with an absolutely very small probability
  - The variable test $T = a^m \pmod n$ where a is random variable between 2 and n-2 inclusively but we choose a = 2

## Code Snippets:

- **Q1:**

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

void sieve (long long number)
{
    vector<bool> primes(number+1, true);                    // suppose all numbers are primes
    primes[0] = primes[1] = false;                          // 0 , 1 are not primes
    for (long long i = 2 ; i <= number ; i++)
    {
        if (primes[i] && (i * i) <= number)
        {
            for (long long j = i * i ; j <= number ; j += i)    // removing all multiples of a prime
            {
                primes[j] = false;
            }
        }
    }
    for(long long i = 2 ; i <= number ; i++)                  // printing the primes
    {
        if(primes[i])
            cout << i << endl;
    }
}

int main()
{
    long long input;
    cout << "Enter the limit number:";
    cin >> input;
    cout << "The primes are:" << endl;
    sieve(input);
    return 0;
}
```

- **Q2:**

```
#include <bits/stdc++.h>

using namespace std;

//Perform Sieve Algorithm on the given number and return a vector of integers
//which contains all the Primes less than the passed number
vector<int> Sieve (long long number)
{
    vector<bool> primes(number+1, true);                 // suppose all numbers are primes
    for (long long i = 2 ; i <= sqrt(number) ; i++)
        if (primes[i])
            for (long long j = i * i ; j <= number ; j += i) primes[j] = false; // removing all multiples of a prime

    vector<int> PrimeNumbers;        //the vector which will contain all the prime numbers up to a given limit
    for(long long i = 2 ; i <= number ; i++)     //inserting the primes in the vector
        if(primes[i]) PrimeNumbers.push_back(i);

    return PrimeNumbers;
}
```

```
void Prime_Factorization (long long number)
{
    vector<int> PrimeNumbers = Sieve(floor(sqrt(number)));  //finding all the primes less than squareRoot the number
    for(int i = 0; i < PrimeNumbers.size(); i++){
        if(number % PrimeNumbers[i] == 0){                //if the number is divisible by a prime then print it
            cout << PrimeNumbers[i] << " ";               //and find the Prime_Factorization of number/prime
            Prime_Factorization(number/PrimeNumbers[i]);
            return;
        }
    }                               //if the number is not divisible by any prime less than sqrt(number)
    cout << number;                 //then the number is already prime so print it
}

int main()
{
    long long number;
    cout << "Enter the desired number to be factorized:";
    cin >> number;
    cout << "The factorization is:" << endl;
    Prime_Factorization(number);
    return 0;
}
```

- **Q3:**

```
.ude <iostream>
.ude <vector>

; namespace std;

 exEuclidean (int num1 , int num2)

'/ 4 vectors for the remainder , quotient , first bezout coeff. , second bezout coeff
rector <int> r;
rector <int> q;
rector <int> s;
rector <int> t;
'/ assigning the initials
.f(num1 >= num2)

    r.push_back(num1);
    r.push_back(num2);
    s.push_back(1);
    s.push_back(0);
    t.push_back(0);
    t.push_back(1);
    q.push_back(0);

:lse if (num1 < num2)

    r.push_back(num2);
    r.push_back(num1);
    t.push_back(1);
    t.push_back(0);
    s.push_back(0);
    s.push_back(1);
    q.push_back(0);

.nt j = 0;
'/ our stopping condition is when the reminder = 0
vhile(r.back() != 0)

    q.push_back(r.at(j) / r.at(j+1));
    r.push_back(r.at(j) % r.at(j+1));
```

```cpp
        // we start to assign new values to the coeff. vectors when we reach iteration number 2
        if(j > 1)
        {
            s.push_back(s.at(j-2) - q.at(j-1)*s.at(j-1));
            t.push_back(t.at(j-2) - q.at(j-1)*t.at(j-1));
        }

        cout << j << "   " << r.at(j) << "   " <<  r.at(j+1) << "   " << q.at(j+1) << "   " << r.at(j+2) << "   " << "   " <<  s.at(j) << "   " << t.at(j) << endl;
        j++;
        // assigning the last values "Bezout's coeff.s"
        if((r.at(j+1) == 0) && (j > 1))
        {
            s.push_back(s.at(j-2) - q.at(j-1)*s.at(j-1));
            t.push_back(t.at(j-2) - q.at(j-1)*t.at(j-1));
            cout << "\t\t\t" << s.at(j) << "   " << t.at(j) << endl;
        }
    }
    cout << "GCD is:" << r.at(j) <<endl;
    cout << "Bezout's coefficients are:" << endl;
    // special case if the user insert a 0
    if(num2 == 0 || num1 == 0)
    {
        cout << "s:" << s.at(0) << endl;
        cout << "t:" << t.at(0) << endl;
    }
    else
    {
        cout << "s:" << s.back() << endl;
        cout << "t:" << t.back() << endl;
    }
}

int main()
{
    int num1 , num2;
    cout << "Enter the 2 numbers:" << endl;
    cin >> num1 >> num2;
    exEuclidean(num1,num2);
    return 0;
}
```

- **Q4:**

```cpp
#include <bits/stdc++.h>

using namespace std;

int s,t;     //Bezout's Coefficients


//returns the greatest common divisor of two numbers and sets their Bezout's identity recursively
int Gcd(int a,int b){
    if(b == 0){
        s = 1;
        t = 0;
        return a;
    }
    int gcd =  Gcd(b,a % b);
    int temp = s;
    s = t;
    t = temp - t * (a/b);
    return gcd;
}

// finds the inverse of a (mod m) where the inverse is the s bezout's identity
int Inverse(int a, int m){
    Gcd(a,m);
    return s;
}
// it takes vector a as parameter which contains a1,a2,a3, ······ an
// it takes vector M as parameter which contains M in index 0 and M1,M2,M3, ······· Mn
// vector y contains y1,y2,y3, ······ yn. where each individual y is the inverse of the corresponding M (mod m)
// vector m contains m1,m2,m3, ······ mn
// n is the number of equations of linear congruences
int Chinese_Remainder_Theorem(vector<int> a , vector<int> M , vector<int> y , vector<int> m , int n ){
    int X = 0;
    for(int i=1;i<=n;i++){
        y[i] = Inverse(M[i],m[i]);      //Calling the Inverse Function To compute yi
        X += a[i] * M[i] * y[i];        //X is the Summation of a1·M1·y1 + a2·M2·y2 ..... an·Mn·yn
    }
    return X < 0 ? X%M[0] + M[0] : X%M[0];
}
```

```cpp
int main()
{
    cout << "Enter number of equations" << endl;
    int n;
    cin >> n;              //number of equations
    vector<int> a(n+1);
    vector<int> m(n+1);
    vector<int> M(n+1);
    vector<int> y(n+1);
    M[0] = 1;         //M[0] will hold the the product of m1· m2· m3 ····· mn
    for(int i=1;i<=n;i++){
        cout << "Enter a" << i << " followed by m" <<i <<endl;
        cin >> a[i] >> m[i];                              //taking the equations from the user
        M[0] *= m[i];
    }

    for(int i=1;i<=n;i++)                //Computing M1, M2, M3 .... Mn
        M[i] = M[0]/m[i];

    int X = Chinese_Remainder_Theorem(a ,M ,y ,m ,n );          //Calling The Chinese_Remainder_Theorem function to compute X
    cout << "X = " << X << " (mod " << M[0] << ")" << endl;     //printing X
}
```

- **Q5:**

```cpp
#include <bits/stdc++.h>

using namespace std;


// computes a power m (mod n) by modular exponentiation
int Modular_Exponentiation(int a , int m , int n){
    int result = 1;
    int power = a % n;
    while(m){
        if(m & 1)                         //if ai = 1
            result = (result * power) % n;
        power = (power * power) % n;
        m >>= 1;
    }
    return result;
}
```

```cpp
// perform the Miller's Rabin Primality test on the given number n and return true if it is prime , false otherwise
bool Miller_Test(int n){
    if(n == 1) return false;
    if(n == 2 || n == 3) return true;
    long long k,m;
    for(k = 1 ; k < log2(n) ; k++){           //finds m, k such that n-1 = m * 2^k
        if((n-1) % (long long)pow(2,k) != 0 ){
            m = (n-1)/pow(2,--k);
            break;
        }
    }
    int T = Modular_Exponentiation(2, m, n);    //find the Test variable which is T = a^m (mod n) where we choose a = 2
    if(T == 1 || T == n-1) return true;         //if T = 1 or T = n-1 then n is probably prime

    for(int i = 0; i < k; i++){
        T = Modular_Exponentiation(T, 2, n);    //T = T^2 (mod n)
        if(T == 1) return false;
        if(T == n-1) return true;
    }
    return false;
}
```

```cpp
int main()
{
    cout << "Enter number to be tested for primality"<<endl;
    int n;
    cin >> n;
    if(Miller_Test(n)) cout << n << " is probably a Prime number";
    else cout << n << " is a composite number";
}
```

## Sample Runs:

- **Q1:**

```
Enter the limit number:100
The primes are:
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

```
Enter the limit number:15
The primes are:
2
3
5
7
11
13

Process returned 0 (0x0)   execution time : 4.616 s
Press any key to continue.
```

```
Enter the limit number:50
The primes are:
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47

Process returned 0 (0x0)   execution time : 3.987 s
Press any key to continue.
```

```
Enter the limit number:200
The primes are:
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
181
191
193
197
199
```

- **Q2:**

```
Enter the desired number to be factorized:
889
The factorization is:
7 127
```

```
Enter the desired number to be factorized:
1024
The factorization is:
2 2 2 2 2 2 2 2 2 2
```

```
Enter the desired number to be factorized:
12911
The factorization is:
12911
```

```
Enter the desired number to be factorized:
729
The factorization is:
3 3 3 3 3 3
```

```
Enter the desired number to be factorized:
45
The factorization is:
3 3 5
```

```
Enter the desired number to be factorized:
18
The factorization is:
2 3 3
```

- **Q3:**

```
Enter the 2 numbers:
24
5
0   24   5   4   4      1   0
1   5   4   1   1      0   1
2   4   1   4   0      1   -4
                       -1   5
GCD is:1
Bezout's coefficients are:
s:-1
t:5
```

```
Enter the 2 numbers:
66
7
0   66   7   9   3      1   0
1   7   3   2   1      0   1
2   3   1   3   0      1   -9
                       -2   19
GCD is:1
Bezout's coefficients are:
s:-2
t:19
```

```
Enter the 2 numbers:
804
14
0   804   14   57   6      1   0
1   14   6   2   2      0   1
2   6   2   3   0      1   -57
                       -2   115
GCD is:2
Bezout's coefficients are:
s:-2
t:115
```

```
Enter the 2 numbers:
7007
140
0   7007   140   50   7      1   0
1   140   7   20   0      0   1
                           1   -50
GCD is:7
Bezout's coefficients are:
s:1
t:-50
```

```
Enter the 2 numbers:
100
5
0   100   5   20   0      1   0
GCD is:5
Bezout's coefficients are:
s:0
t:1
```

```
Enter the 2 numbers:
84562
62
0   84562   62   1363   56      1   0
1   62   56   1   6      0   1
2   56   6   9   2      1   -1363
3   6   2   3   0      -1   1364
                       10   -13639
GCD is:2
Bezout's coefficients are:
s:10
t:-13639
```

- **Q4:**

```
Enter number of equations
3
Enter a1 followed by m1
2 3
Enter a2 followed by m2
3 5
Enter a3 followed by m3
2 7
X = 23 (mod 105)
```

```
Enter number of equations
4
Enter a1 followed by m1
5 7
Enter a2 followed by m2
2 19
Enter a3 followed by m3
5 11
Enter a4 followed by m4
9 23
X = 2700 (mod 33649)
```

```
Enter number of equations
5
Enter a1 followed by m1
1 3
Enter a2 followed by m2
2 5
Enter a3 followed by m3
3 7
Enter a4 followed by m4
4 11
Enter a5 followed by m5
5 13
X = 14227 (mod 15015)
```

```
Enter number of equations
3
Enter a1 followed by m1
1 2
Enter a2 followed by m2
2 3
Enter a3 followed by m3
3 5
X = 23 (mod 30)
```

- **Q5:**

```
Enter number to be tested for primality
4
4 is a composite number
```

```
Enter number to be tested for primality
541
541 is probably a Prime number
```

```
Enter number to be tested for primality
97
97 is probably a Prime number
```

```
Enter number to be tested for primality
19211
19211 is probably a Prime number
```

```
Enter number to be tested for primality
633
633 is a composite number
```