Course Name:	Applied Cryptography	Semester:	V
Date of Performance:	19 / 08 / 2025	DIV/ Batch No:	D2
Student Name:	Shreyans Tatiya	Roll No:	16010123325

Experiment No: 2
Title: Cryptographic Arithmetic

Aim and Objective of the Experiment:

To learn about Cryptographic Arithmetic and its implementation

1. Prime number generation
2. Primality testing
3. Prime Factorization

COs to be achieved:

CO2: Demonstrate and implement various Cryptographic Algorithms for securing systems

Books/ Journals/ Websites referred:

1. Stallings, W., Cryptography and Network Security: Principles and Practice, Second edition, Person Education
2. "Caesar Cipher in cryptography", https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/, last retrieved on Aug 01, 2023
3. "PlayFair Cipher in cryptography": https://www.geeksforgeeks.org/playfair-cipher-with-examples/, last retrieved on Aug 01, 2023
4. "Transposition cipher in cryptology, "https://www.britannica.com/topic/transposition-cipher, last retrieved on Aug 01, 2023

Theory:

Abstract:-
Cryptographic arithmetic involves using mathematical operations and techniques within cryptography to secure communication and data. It includes concepts like modular arithmetic, one-way functions, prime numbers, exponentiation, and elliptic curve cryptography, all of which are crucial for ensuring the confidentiality, integrity, and authenticity of data in cryptographic systems.

Related Theory:

1. Number Theory: Number theory forms the foundation of many cryptographic algorithms. Concepts such as prime numbers, modular arithmetic, greatest common divisors, and Euler's totient function are central to understanding and implementing cryptographic arithmetic.

2. Modular Arithmetic: Modular arithmetic is fundamental in cryptographic operations. It involves performing arithmetic operations within a finite set, called a modulus. Theorems related to modular arithmetic, like Euler's Totient Theorem, are critical for cryptographic algorithms like RSA and Diffie-Hellman.

3. Additive Inverse: Two numbers a and b are additive inverses of each other

if a + b ≡ 0 (mod n). The additive inverse of a can be calculated as b = n − a.

4. Multiplicative Inverse: Two numbers a and b are the multiplicative inverse of each other if a × b ≡ 1 (mod n)

Algorithm : (Write assigned algo)

Step 1: Initialize a list of booleans s[] representing integers up to n, all set to false.
Step 2: Mark 2 and 3 as prime numbers.
Step 3: For every pair (x, y) such that x^2 and y^2 are less than or equal to n:

- Flip s[k] for:
  - $k = 4x^2 + y^2$ if $k\%12 = 1$ or 5
  - $k = 3x^2 + y^2$ if $k\%12 = 7$
  - $k = 3x^2 - y^2$ if $x > y$ and $k\%12 = 11$

  Step 4: Eliminate multiples of squares:
- For each i where s[i] is true, mark all multiples of i^2 as false.
  Step 5: Collect all indices i where s[i] is true — these are prime numbers.

Output: List of all prime numbers ≤ n.

Solve a small numerical for assigned algorithm(Paste photograph of same) :

Q.

Let's find all **prime numbers up to 20** using the **Sieve of Atkin** algorithm.

Step 1:

Given limit **n = 20**, initialize all numbers as non-prime (false).

Step 2:

Use quadratic forms to toggle entries:

Formula	Condition	Resulting Numbers
$4x^2 + y^2$	k % 12 = 1 or 5	5, 13, 17
$3x^2 + y^2$	k % 12 = 7	7, 19
$3x^2 - y^2$	k % 12 = 11	11

Step 3:

Mark multiples of squares (like 4, 9, 16) as non-prime.

Step 4:

Collect all numbers still marked as true along with 2 and 3.

Prime numbers ≤ 20 are:
2, 3, 5, 7, 11, 13, 17, 19

Code :

```
#include <bits/stdc++.h>
using namespace std;

vector<int> atkin(int n) {
    vector<int> p;
    if (n < 2) return p;
    if (n >= 2) p.push_back(2);
    if (n >= 3) p.push_back(3);

    vector<bool> s(n + 1, false);
    int m = sqrt(n);

    for (int x = 1; x <= m; x++) {
        int x2 = x * x;
        for (int y = 1; y <= m; y++) {
            int y2 = y * y;
```

```
            int k = 4 * x2 + y2;
            if (k <= n && (k % 12 == 1 || k % 12 == 5)) s[k] = !s[k];

            k = 3 * x2 + y2;
            if (k <= n && k % 12 == 7) s[k] = !s[k];

            k = 3 * x2 - y2;
            if (x > y && k <= n && k % 12 == 11) s[k] = !s[k];
        }
    }

    for (int i = 5; i <= m; i++) {
        if (s[i]) {
            int i2 = i * i;
            for (int j = i2; j <= n; j += i2) s[j] = false;
        }
    }

    for (int i = 5; i <= n; i++) if (s[i]) p.push_back(i);
    return p;
}

int main() {
    int n;
    cin >> n;
    for (int x : atkin(n)) cout << x << " ";
    cout << "\n";
}
```

Output:

```
PS D:\KJSCE> cd "d:\KJSCE\SEM-5\AC\Lab\Expt-4\" ; if ($?) { g++ SieveofAtkin.cpp -o SieveofAtkin } ; if ($?) { .\SieveofAtkin }
30
2 3 5 7 11 13 17 19 23 29
PS D:\KJSCE\SEM-5\AC\Lab\Expt-4> cd "d:\KJSCE\SEM-5\AC\Lab\Expt-4\" ; if ($?) { g++ SieveofAtkin.cpp -o SieveofAtkin } ; if ($?) {
.\SieveofAtkin }
190
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
PS D:\KJSCE\SEM-5\AC\Lab\Expt-4>
```

Post Lab Subjective/Objective type Questions:

1. Why cryptography is strongly based on prime numbers and modulo arithmetic?

   Ans: Cryptography relies heavily on prime numbers and modular arithmetic because they form the foundation for creating secure and hard-to-break encryption systems.

- **Prime numbers** are difficult to factorize — given a product of two large primes, it's computationally expensive to find the original primes (used in RSA encryption).
- **Modular arithmetic** ensures that numbers "wrap around" after reaching a certain limit (modulus). This creates **non-reversible mathematical operations**, making it ideal for

encryption.

- Together, primes and modulo operations allow creation of **one-way functions**, which are easy to compute but extremely hard to invert without the private key.
  Example:
  RSA uses:

$$C = M^e \bmod n$$

  Where $n = p \times q$(two large primes).
  Decryption requires knowledge of $p$and q, which is practically impossible to find for large primes.

2. Discuss significance of primality testing algorithms.
   Ans: Primality testing algorithms determine whether a given number is prime.
   They are significant in cryptography and computational mathematics because:
3. **Key Generation:**
   Public-key cryptosystems like RSA require large prime numbers to generate secure keys.
4. **Efficiency:**
   Instead of checking divisibility by all numbers, primality tests (like Miller–Rabin or Fermat) quickly confirm whether a number is prime or composite.
5. **Security:**
   Using strong primality tests ensures that cryptographic keys are **truly prime**, reducing the risk of factorization attacks.
6. **Applications Beyond Cryptography:**
   Used in hashing, pseudorandom number generation, and digital signatures.
7. **Example:**
   Miller–Rabin test is a probabilistic algorithm that confirms if a number is "probably prime" with very high accuracy.

3. Compare and contrast prime number generation algorithms
Ans:

Algorithm	Type	Approach	Time Complexity	Space Complexity	Remarks
Trial Division	Basic	Divide by all smaller numbers	O(√n)	O(1)	Simple but slow for large n
Sieve of Eratosthenes	Deterministic	Marks multiples of primes	O(n log log n)	O(n)	Fast for small/medium n
Sieve of Atkin	Deterministic	Uses quadratic forms and modulo	O(n / log log n)	O(n)	Faster and more complex

		rules			than Eratosthenes
Miller–Rabin	Probabilistic	Tests compositeness using modular exponentiation	$O(k \times \log^3 n)$	$O(1)$	Used for large primes in cryptography
Fermat Test	Probabilistic	Uses Fermat's Little Theorem	$O(\log^3 n)$	$O(1)$	Less reliable; fails for Carmichael numbers

Conclusion:

Through this experiment, I learned how prime numbers and modular arithmetic form the mathematical foundation of cryptography. Implementing the Sieve of Atkin helped me understand efficient prime number generation, which is essential for secure key creation in cryptographic systems.