AI Project

Task 1:

1. Write a program to find all Pythagorean quadruples between two given numbers. (A Pythagorean quadruple (a, b, c, d) satisfies the equation.

```
#include <stdio.h>

int main() {
    int a, b, c, d, m, n;
    printf("Enter two numbers: ");
    scanf("%d %d", &m, &n);
    for (a = m; a <= n; a++) {
        for (b = a; b <= n; b++) {
            for (c = b; c <= n; c++) {
                for (d = c; d <= n; d++) {
                    if (a*a + b*b + c*c == d*d) {
                        printf("(%d, %d, %d, %d)\n", a, b, c, d);
                    }
                }
            }
        }
    }
    return 0;
}
```

`My code :`

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

```c
int main() {
    int m, n;
    printf("Enter two numbers: ");
    scanf("%d %d", &m, &n);

    for (int a = m; a <= n; a++) {
        for (int b = a; b <= n; b++) {
            for (int c = b; c <= n; c++) {
                for (int d = c; d <= n; d++) {
                    int sumAB = a * a + b * b;
                    int sumCD = c * c + d * d;

                    if (sumAB == sumCD) {
                        printf("(%d, %d, %d, %d)\n", a, b, c, d);
                    }
                }
            }
        }
    }

    return 0;
}
```

Similarities:

Input prompt for m and n.

Nested loops iterate through combinations.

Print statement for valid combinations.

Differences:

Variable Declarations:

- Code 1: Declares all variables at the beginning.
- Code 2: Declares a, b, c, and d within loops.

Variable Scoping:

- Code 1: Variables visible throughout the main function.
- Code 2: Variables scoped within the for loops.

Sum Calculation:

- Code 2 calculates sums of squares before the conditional check.

Loop Initialization:

- Code 2 initializes loop variables within for loop headers.

2.Write a program to perform modular exponentiation of two numbers x and y modulo m. (Compute (x^y) % m efficiently using the repeated squaring met

```
#include <stdio.h>

int modular_exponentiation(int x, int y, int m) {
    int result = 1;
    x = x % m;

    while (y > 0) {
        if (y & 1) {
            result = (result * x) % m;
        }
        y = y >> 1;
        x = (x * x) % m;
    }

    return result;
}

int main() {
    int x, y, m;

    printf("Enter the values of x, y, and m: ");

    scanf("%d %d %d", &x, &y, &m);


    int result = modular_exponentiation(x, y, m);

    printf("%d^%d mod %d = %d\n", x, y, m, result);


    return 0;

}
```

My code::

```
 include <stdio.h>


int power_modulo(int base, int exponent, int modulo) {

    int result = 1;


    while (exponent > 0) {

        if (exponent % 2 == 1) {

            result = (result * base) % modulo;

        }

        base = (base * base) % modulo;

        exponent /= 2;

    }


    return result;

}


int main() {

    int x, y, m;
    printf("Enter the values of x, y, and m: ");
    scanf("%d %d %d", &x, &y, &m);

    int result = power_modulo(x, y, m);

    printf("%d^%d mod %d = %d\n", x, y, m, result);

    return 0;
}
```

Similarities:

Purpose: Both codes perform modular exponentiation and print the result.
Input: Both codes prompt the user to enter values for x,y, and m.
Loop Structure: Both codes use a while loop to iteratively calculate the modular exponentiation.
Print Statement: The printf statement is identical, printing the values of x, y, m, and the result.

Differences:

Function Name and Implementation:

- Code 1: Uses the function name modular_exponentiation and implements the exponentiation using bitwise operations.
- Code 2: Uses the function name power_modulo and implements exponentiation using the traditional modulo approach.

Conditional Check in Loop:

- Code 1: Uses bitwise AND (y & 1) for the conditional check in the loop.
- Code 2: Uses the modulo operator (exponent % 2 == 1) for the conditional check.

Loop Updates:

- Code 1: Right-shifts (y = y >> 1) the exponent in each iteration.
- Code 2: Divides (exponent /= 2) the exponent in each iteration.

Variable Naming:

- Code 1 uses result, x, and y as variable names.
- Code 2 uses result, base, and exponent as variable names.

Implementation Details:

- Code 1 utilizes bitwise operations for efficiency, which may be more optimal in some cases.
- Code 2 uses traditional arithmetic operations and may be more readable to those unfamiliar with bitwise operations.

3. Write a program to find all prime factors of a given number using the sieve of Eratosthenes method.

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


void sieve(int n, bool *primes) {

    primes[0] = primes[1] = false;

    for (int i = 2; i <= n; i++) {

        primes[i] = true;

    }

    for (int i = 2; i * i <= n; i++) {

        if (primes[i]) {

            for (int j = i * i; j <= n; j += i) {

                primes[j] = false;

            }

        }

    }

}


void prime_factors(int n) {

    bool primes[n + 1];

    sieve(n, primes);

    for (int i = 2; i <= n; i++) {

        if (primes[i] && n % i == 0) {

            printf("%d ", i);

            while (n % i == 0) {

                n /= i;
            }
        }
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    printf("Prime factors of %d are: ", n);
    prime_factors(n);
    return 0;
}
```

My code:

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

void sieve(int n, int *primes) {
    for (int i = 0; i <= n; i++) {
        primes[i] = 1;
    }
    primes[0] = primes[1] = 0;
    for (int i = 2; i * i <= n; i++) {
        if (primes[i]) {
            for (int j = i * i; j <= n; j += i) {
                primes[j] = 0;
            }
        }
    }
}

void prime_factors(int n) {
    int *primes = (int *)malloc((n + 1) * sizeof(int));
    sieve(n, primes);
    for (int i = 2; i <= n; i++) {
        if (primes[i] && n % i == 0) {
            printf("%d ", i);
            while (n % i == 0) {
                n /= i;
            }
        }
    }
    printf("\n");
    free(primes);
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Prime factors of %d are: ", num);
    prime_factors(num);
    return 0;
}
```

Similarities:

Purpose: Both codes aim to find and print the prime factors of a given number.

Input: Both codes prompt the user to enter a number.

Sieve Function: Both codes contain a sieve function that marks non-prime numbers up to a specified limit.

Prime Factors Function: Both codes contain a prime_factors function that uses the sieve to find and print prime factors.

Memory Management: Both codes use dynamic memory allocation to create an array for the sieve.

Differences:

Memory Allocation:

- Code 1: Uses a bool array (bool primes[]) to store whether a number is prime or not.
- Code 2: Uses an int array (int *primes) with values 0 or 1 to represent prime or non-prime.

Memory Deallocation:

- Code 1: No explicit memory deallocation is needed as it uses a statically allocated array for the sieve.
- Code 2: Uses malloc to dynamically allocate memory for the sieve, and free is called to deallocate the memory.

Array Indexing:

- Code 1: Indexing starts from 0, so primes array represents primes[0] to primes[n].
- Code 2: Indexing starts from 0, so primes array represents primes[0] to primes[n].

Array Initialization:

- Code 1: Initializes bool array elements to true for all numbers up to n initially.
- Code 2: Initializes int array elements to 1 for all numbers up to n initially.

Printf Format Specifier:

- Code 1: Uses %d as the format specifier for printf when printing integers.
- Code 2: Also uses %d as the format specifier for printf.

Variable Naming:

- Code 1: Uses n as the variable name for the limit and primes for the sieve array.
- Code 2: Uses n as the variable name for the limit, and num and primes for the sieve array.

4. Write a program to find the modulo multiplicative inverse of two numbers using the extended Euclidean algorithm.

```
#include <stdio.h>

int gcdExtended(int a, int b, int *x, int *y) {
    if (a == 0) {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1;
    int gcd = gcdExtended(b % a, a, &x1, &y1);

    *x = y1 - (b / a) * x1;
    *y = x1;

    return gcd;
}

int modInverse(int a, int m) {
    int x, y;
    int gcd = gcdExtended(a, m, &x, &y);

    if (gcd != 1) {
        printf("Inverse doesn't exist\n");
        return -1;
    } else {
        int res = (x % m + m) % m;
        return res;
    }
}
```

```
int main() {
    int a, m;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &m);

    int inverse = modInverse(a, m);
    if (inverse == -1) {
        printf("No inverse exists\n");
    } else {
        printf("The modulo multiplicative inverse of %d mod %d is %d\n", a, m, inverse);
    }

    return 0;
}
```

My code:

```
#include <stdio.h>

int extendedGCD(int a, int b, int *x, int *y) {
    if (a == 0) {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1;
    int gcd = extendedGCD(b % a, a, &x1, &y1);
    *x = y1 - (b / a) * x1;
    *y = x1;

    return gcd;
}

int modInverse(int a, int m) {
    int x, y;
    int gcd = extendedGCD(a, m, &x, &y);

    if (gcd != 1) {
        printf("Inverse doesn't exist\n");
        return -1;
    } else {
        int result = (x % m + m) % m;
        return result;
    }
}

int main() {
    int num1, num2;

    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    int inverse = modInverse(num1, num2);

    if (inverse == -1) {
        printf("No inverse exists\n");
    } else {
        printf("Modulo Multiplicative Inverse of %d mod %d is %d\n", num1, num2, inverse);
    }

    return 0;
}
```

Similarities:

Purpose: Both codes find the modular multiplicative inverse of a modulo m.

Input: Both codes prompt the user to enter two numbers a and m.

Extended GCD Function: Both codes have a function (gcdExtended or extendedGCD) that calculates the extended greatest common divisor, providing coefficients x and y such that ax+by=gcd(a,b).

Modular Inverse Function: Both codes have a function (modInverse) that uses the extended GCD to calculate the modular multiplicative inverse of a modulo m.

Error Handling: Both codes print an error message if the inverse doesn't exist (when the gcd is not 1).

Printf Format Specifier: Both codes use %d as the format specifier for printf when printing integers.

Differences:

Variable Naming:
- Code 1: Uses x, y, gcd, a, and m as variable names.
- Code 2: Uses x, y, gcd, a, m, result, num1, and num2 as variable names.

Printf Statement:
- Code 1: Prints "The modulo multiplicative inverse of a mod m is inverse
- Code 2: Prints "Modulo Multiplicative Inverse of num1 mod num2 is inverse".

Return Values:
- Code 1: Returns -1 if the inverse doesn't exist.
- Code 2: Returns -1 if the inverse doesn't exist.

Function Naming:
- Code 1: Uses modInverse and gcdExtended.
- Code 2: Uses modInverse and extendedGCD.

Error Message:
- Code 1: Prints "No inverse exists" when the inverse doesn't exist.
- Code 2: Prints "No inverse exists" when the inverse doesn't exist.

5. Write a program to check if two numbers are coprime using the Euclidean algorithm to find the greatest common divisor (GCD). Print pairs of coprimes between a given range.

```
#include <stdio.h>

// Function to find GCD of two numbers using Euclidean algorithm
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

// Function to check if two numbers are coprime
int are_coprime(int a, int b) {
    return gcd(a, b) == 1;
}

// Function to print pairs of coprimes between a given range
void print_coprimes(int start, int end) {
    for (int i = start; i <= end; i++) {
        for (int j = i + 1; j <= end; j++) {
            if (are_coprime(i, j)) {
                printf("(%d, %d)\n", i, j);
            }
        }
    }
}


// Driver code
int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    if (are_coprime(a, b)) {
        printf("%d and %d are coprime.\n", a, b);
    } else {
        printf("%d and %d are not coprime.\n", a, b);
    }
    int start, end;
    printf("Enter the range to print coprimes: ");
    scanf("%d %d", &start, &end);
    printf("Coprimes between %d and %d:\n", start, end);
    print_coprimes(start, end);
    return 0;
}
```

My Code:

```
#include <stdio.h>

// Function to find the GCD of two numbers using Euclidean algorithm
int findGCD(int num1, int num2) {
    while (num2 != 0) {
        int temp = num2;
        num2 = num1 % num2;
        num1 = temp;
    }
    return num1;
}

// Function to check if two numbers are coprime
int areCoprime(int num1, int num2) {
    return findGCD(num1, num2) == 1;
}

// Function to print pairs of coprimes within a given range
void printCoprimesInRange(int start, int end) {
    printf("Coprimes between %d and %d:\n", start, end);
    for (int i = start; i <= end; i++) {
        for (int j = i + 1; j <= end; j++) {
            if (areCoprime(i, j)) {
                printf("(%d, %d)\n", i, j);
            }
        }
    }
}

// Driver code
int main() {
    int input1, input2;
    printf("Enter two numbers: ");
    scanf("%d %d", &input1, &input2);

    if (areCoprime(input1, input2)) {
        printf("%d and %d are coprime.\n", input1, input2);
    } else {
        printf("%d and %d are not coprime.\n", input1, input2);
    }

    int rangeStart, rangeEnd;
    printf("Enter the range to print coprimes: ");
    scanf("%d %d", &rangeStart, &rangeEnd);

    printCoprimesInRange(rangeStart, rangeEnd);

    return 0;
}
```

Similarities:

Purpose: Both codes check if two numbers are coprime and print coprime pairs within a specified range.

GCD Calculation: Both codes use the Euclidean algorithm to calculate the greatest common divisor (GCD) of two numbers.

Coprime Check: Both codes have a function (are_coprime or areCoprime) to check if two numbers are coprime.

Print Coprime Pairs: Both codes have a function (print_coprimes or printCoprimesInRange) to print pairs of coprime numbers within a given range.

Input and Output: Both codes prompt the user to enter two numbers, check if they are coprime, and print coprime pairs within a specified range.

Printf Format Specifier: Both codes use %d as the format specifier for printf when printing integers.

Differences:

Function Naming:
- Code 1: Uses gcd, are_coprime, and print_coprimes.
- Code 2: Uses findGCD, areCoprime, and printCoprimesInRange.

Recursive vs Iterative GCD:
- Code 1: Uses a recursive approach to find the GCD (gcd function).
- Code 2: Uses an iterative approach to find the GCD (findGCD function).

Variable Naming:
- Code 1: Uses start and end for the range.
- Code 2: Uses rangeStart and rangeEnd for the range.

Driver Code:

- Code 1: Uses variables named a, b, start, and end in the driver code.
- Code 2: Uses variables named input1, input2, rangeStart, and rangeEnd in the driver code.

Error Message:

- Code 1: Does not explicitly handle the case where the range start is greater than the range end in the coprime printing function.
- Code 2: Prints "Coprimes between X and Y" even if X > Y in the coprime printing function.

Variable Scope:

- Code 1: Uses local variables for the range within the main function.
- Code 2: Uses local variables for the range within the main function.

6. Write a program to find Euler's totient function phi(n) for a given number n. Phi(n) is the number of integers less than n that are coprime to n.

```
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

int phi(int n) {
    int result = 1;
    for (int i = 2; i < n; i++) {
        if (gcd(i, n) == 1) {
            result++;
        }
    }
    return result;
}


int main() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    printf("phi(%d) = %d\n", n, phi(n));
    return 0;
}
```

My code:

```
#include <stdio.h>


// Function to calculate the greatest common divisor (GCD) of two numbers
int calculateGCD(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}


// Function to calculate Euler's totient function (phi) for a given number
int calculatePhi(int n) {
    int result = 1; // Initialize result with 1 since 1 is always coprime
to any number


    // Iterate through numbers from 2 to n-1 and check coprimality with n
    for (int i = 2; i < n; i++) {
        if (calculateGCD(i, n) == 1) {
            result++;
        }
    }

    return result;
}

int main() {
    int userInput;
    printf("Enter a number: ");
    scanf("%d", &userInput);

    int resultPhi = calculatePhi(userInput);

    printf("Euler's totient function phi(%d) = %d\n", userInput,
resultPhi);

    return 0;
}
```

Similarities:

Purpose: Both codes calculate Euler's totient function (ϕ) for a given number n.

GCD Calculation: Both codes use a function (gcd or calculateGCD) to calculate the greatest common divisor (GCD) of two numbers.

ϕ Calculation: Both codes use a loop to iterate through numbers from 2 to n−1 and check coprimality with n to calculate ϕ(n).

Input and Output: Both codes prompt the user to enter a number, calculate ϕ(n), and print the result.

Printf Format Specifier: Both codes use %d as the format specifier for printf when printing integers.

Differences:

Function Naming:

- Code 1: Uses gcd and phi for function names.
- Code 2: Uses calculateGCD and calculatePhi for function names.

Recursive vs Iterative GCD:

- Code 1: Uses a recursive approach to find the GCD (gcd function).
- Code 2: Uses an iterative approach to find the GCD (calculateGCD function).

Variable Naming:

- Code 1: Uses n for the input number and result for the
- ϕ result.
- Code 2: Uses userInput for the input number and resultPhi for the ϕ result.

ϕ Initialization:

- Code 1: Initializes result to 1 since 1 is always coprime to any number.
- Code 2: Also initializes result to 1 with the same rationale.

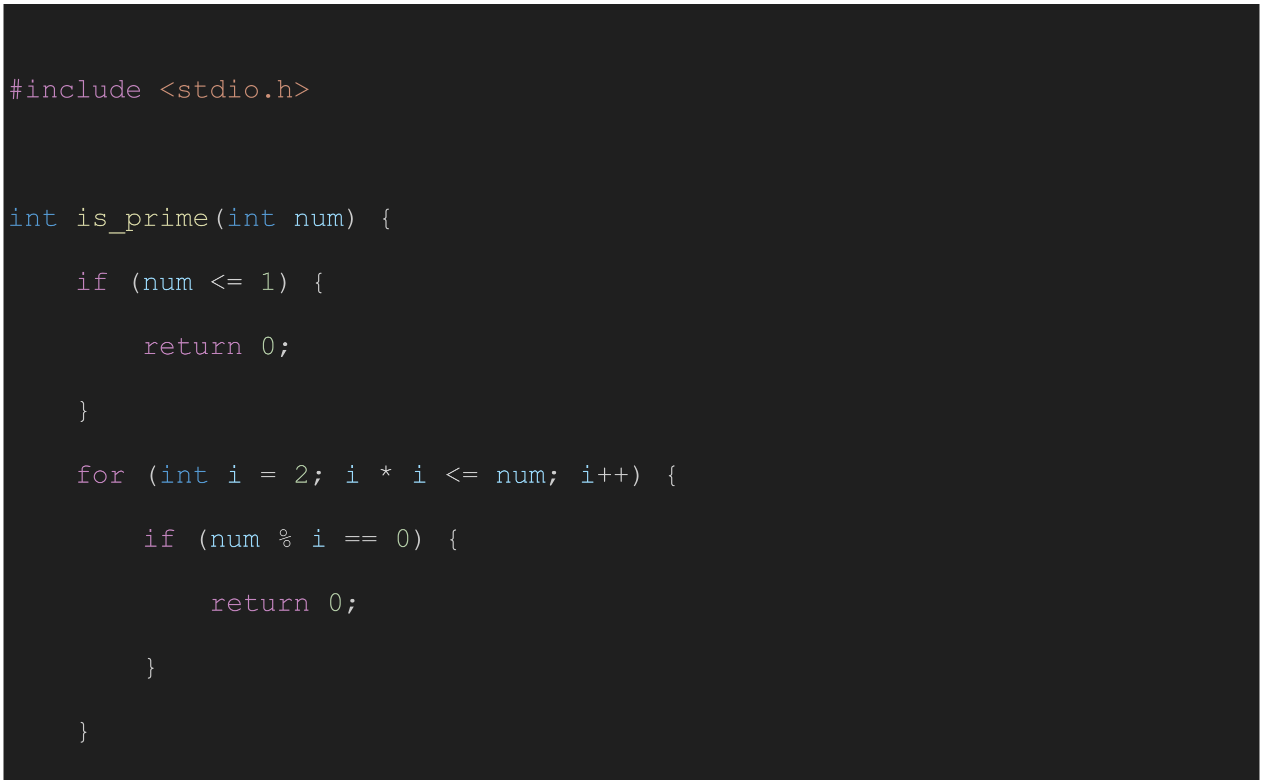
Loop Range:

- Code 1: Iterates from 2 to n−1 to check coprimality.
- Code 2: Also iterates from 2 to n−1 for the same purpose.

Error Handling:

- Both codes assume valid user input without explicit error handling for invalid inputs (e.g., negative numbers).

7. Write a program to find all prime numbers between two given numbers.



```
#include <stdio.h>

int is_prime(int num) {
    if (num <= 1) {
        return 0;
    }
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0) {
            return 0;
        }
    }
    return 1;
}


int main() {
    int start, end;
    printf("Enter the start and end numbers: ");
    scanf("%d %d", &start, &end);
    printf("Prime numbers between %d and %d are: ", start, end);
    for (int i = start; i <= end; i++) {
        if (is_prime(i)) {
            printf("%d ", i);
        }
    }
    printf("\n");
    return 0;
}
```

My code:

```
#include <stdio.h>


int is_prime(int num) {
    if (num <= 1) {
        return 0;
    }
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0) {
            return 0;
        }
    }
    return 1;
}

int main() {
    int lower_limit, upper_limit;

    printf("Enter the lower and upper limits: ");
    scanf("%d %d", &lower_limit, &upper_limit);

    printf("Prime numbers between %d and %d are: ", lower_limit, upper_limit);

    for (int num = lower_limit; num <= upper_limit; num++) {
        int flag = is_prime(num);
        if (flag) {
            printf("%d ", num);
        }
    }

    printf("\n");
    return 0;
}
```

Similarities:

Purpose: Both codes aim to find and print prime numbers within a specified range.

Prime Check Function: Both codes use a function (is_prime) to check whether a given number is prime.

Input and Output: Both codes prompt the user to enter the range and print prime numbers within that range.

Printf Format Specifier: Both codes use %d as the format specifier for printf when printing integers.

Differences:

Variable Naming:

- Code 1: Uses start and end for the range.
- Code 2: Uses lower_limit and upper_limit for the range, and num for the loop variable.

Loop Initialization:

- Code 1: Uses a loop variable i initialized to start and iterates up to end.
- Code 2: Uses a loop variable num initialized to lower_limit and iterates up to upper_limit.

Flag Usage:

- Code 1: Directly checks the result of is_prime(i) in the loop condition.
- Code 2: Uses a flag variable flag to store the result of is_prime(num).

Loop Variable Update:

- Code 1: Increments the loop variable i in the loop update.
- Code 2: Uses the loop variable num as is, with no increment in the loop update.

Variable Scope:

- Code 1: Uses local variables for the range within the main function.
- Code 2: Uses local variables for the range and loop variable within the main function.

Prompt Message:

- Code 1: Asks the user to enter "start and end numbers."
- Code 2: Asks the user to enter "lower and upper limits."

8. Write a program to find the Greatest Common Divisor (GCD) of two given numbers.

```c
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

int main() {
    int num1, num2;
    printf("Enter two numbers: ");

    scanf("%d %d", &num1, &num2);

    int result = gcd(num1, num2);

    printf("GCD of %d and %d is %d\n", num1, num2, result);

    return 0;

}
```

My code:

```
#include <stdio.h>


int calculateGCD(int a, int b) {

    if (b == 0) {

        return a;

    }

    return calculateGCD(b, a % b);

}


int main() {

    int number1, number2;

    printf("Enter two numbers: ");

    scanf("%d %d", &number1, &number2);


    int gcdResult = calculateGCD(number1, number2);


    printf("GCD of %d and %d is %d\n", number1, number2, gcdResult);


    return 0;

}
```

Similarities:

Purpose: Both codes calculate the GCD of two numbers.
GCD Function: Both codes have a GCD function (gcd or calculateGCD) that implements the Euclidean algorithm.
Input and Output: Both codes prompt the user to enter two numbers, calculate the GCD, and then print the result.
Printf Format Specifier: Both codes use %d as the format specifier for printf when printing integers.

Differences:

Variable Naming:
- Code 1: Uses num1 and num2 for the input numbers.
- Code 2: Uses number1 and number2 for the input numbers.

Function Naming:
- Code 1: Uses gcd for the GCD function.
- Code 2: Uses calculateGCD for the GCD function.

Function Invocation:
- Code 1: Invokes the GCD function as gcd(num1, num2).
- Code 2: Invokes the GCD function as calculateGCD(number1, number2).

Variable Scope:
- Code 1: Uses local variables within the main function.
- Code 2: Uses local variables within the main function.

Prompt Message:
- Code 1: Asks the user to enter "two numbers."
- Code 2: Asks the user to enter "two numbers."

Variable Initialization:
- Code 1: No explicit initialization of result is needed before calculating the GCD.
- Code 2: Initializes gcdResult to the result of calculateGCD before printing.

9. Write a program to find the Least Common Multiple (LCM) of two given numbers.

```c
#include <stdio.h>

// function to find GCD of two numbers
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

// function to find LCM of two numbers
int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}

int main() {
    int num1, num2;
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);
    printf("LCM of %d and %d is %d", num1, num2, lcm(num1, num2));
    return 0;
}
```

My Code:

```
#include <stdio.h>

// Function to find LCM using a loop
int calculateLCM(int a, int b) {
    int max = (a > b) ? a : b;

    while (1) {
        if (max % a == 0 && max % b == 0) {
            return max;
        }
        max++;
    }
}

int main() {
    int num1, num2;

    // Input two numbers from the user
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    // Calculate and display LCM
    int lcmResult = calculateLCM(num1, num2);
    printf("LCM of %d and %d is %d\n", num1, num2, lcmResult);

    return 0;
}
```

Similarities:

Purpose: Both codes aim to calculate the LCM of two numbers.
Input and Output: Both codes prompt the user to enter two numbers, calculate the LCM, and print the result.
Printf Format Specifier: Both codes use %d as the format specifier for printf when printing integers.

Differences:

LCM Calculation:

- Code 1: Calculates LCM using the formula (a×b)/gcd(a,b).
- Code 2: Calculates LCM using a loop and incrementing a variable until finding the first multiple common to both numbers.

Function Naming:

- Code 1: Uses gcd and lcm for the GCD and LCM functions, respectively.
- Code 2: Uses calculateLCM for the LCM function.

Function Invocation:

- Code 1: Invokes the LCM function as lcm(num1, num2).
- Code 2: Invokes the LCM function as calculateLCM(num1, num2).

Variable Naming:

- Code 1: Uses num1 and num2 for the input numbers.
- Code 2: Uses num1 and num2 for the input numbers, and lcmResult for the calculated LCM.

Variable Scope:

- Code 1: Uses local variables within the main function.
- Code 2: Uses local variables within the main function.

Loop vs Formula:

- Code 1: Uses a direct formula to calculate the LCM.
- Code 2: Uses a loop to find the LCM by checking multiples until a common multiple is found.

Prompt Message:

- Code 1: Asks the user to enter "two numbers."
- Code 2: Asks the user to enter "two numbers."

10. Write a program to check if two given numbers are coprime or not.

```
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

int main() {
```

```c
    int num1, num2;

    printf("Enter two numbers: ");

    scanf("%d %d", &num1, &num2);

    if (gcd(num1, num2) == 1) {

        printf("%d and %d are coprime\n", num1, num2);

    } else {

        printf("%d and %d are not coprime\n", num1, num2);

    }

    return 0;

}
```

My Code:

```c
#include <stdio.h>


// Function to check if two numbers are coprime

int areCoprime(int a, int b) {

    while (b != 0) {

        int temp = b;

        b = a % b;

        a = temp;

    }

    return (a == 1);

}


int main() {

    int number1, number2;


    // Input two numbers from the user
    printf("Enter two numbers: ");
    scanf("%d %d", &number1, &number2);

    // Check and display if numbers are coprime
    if (areCoprime(number1, number2)) {
        printf("%d and %d are coprime\n", number1, number2);
    } else {
        printf("%d and %d are not coprime\n", number1, number2);
    }

    return 0;
}
```

Similarities:

Purpose: Both codes aim to determine whether two given numbers are coprime.

Input and Output: Both codes prompt the user to enter two numbers, perform the coprimality check, and print the result.

Printf Format Specifier: Both codes use %d as the format specifier for printf when printing integers.

Differences:

Function Naming:
- Code 1: Uses gcd for the GCD function and directly checks if GCD is 1 in the main function.
- Code 2: Uses a separate function named areCoprime to check coprimality.

Function Invocation:
- Code 1: Invokes the GCD function as gcd(num1, num2) and directly checks if it's equal to 1 in the main function.
- Code 2: Invokes the areCoprime function as areCoprime(number1, number2).

Variable Naming:
- Code 1: Uses num1 and num2 for the input numbers.
- Code 2: Uses number1 and number2 for the input numbers.

Variable Scope:
- Code 1: Uses local variables within the main function.
- Code 2: Uses local variables within the main function and the areCoprime function.

Conditional Check:
- Code 1: Directly checks if the GCD is equal to 1 in the main function.

- Code 2: Uses a separate areCoprime function with a conditional check (a == 1).

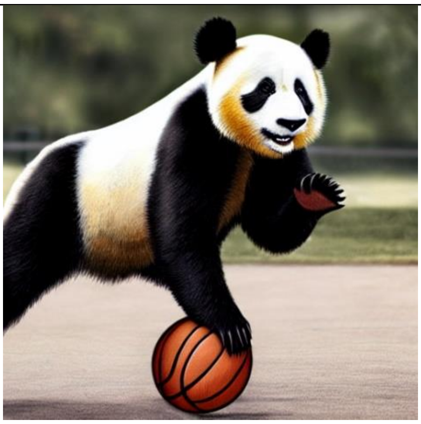
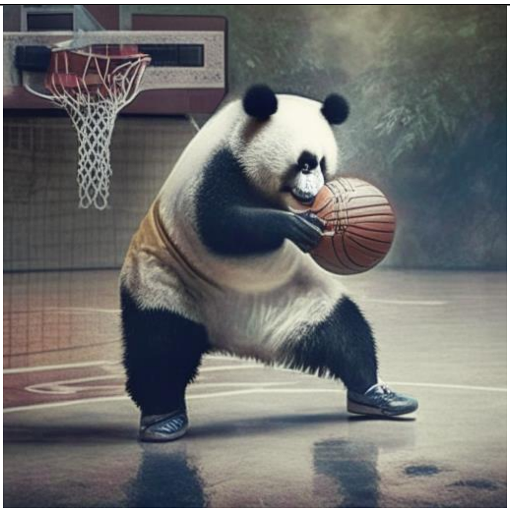
Loop vs Recursive GCD:

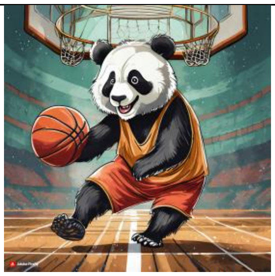
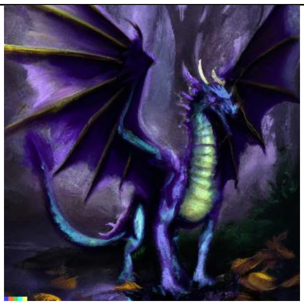
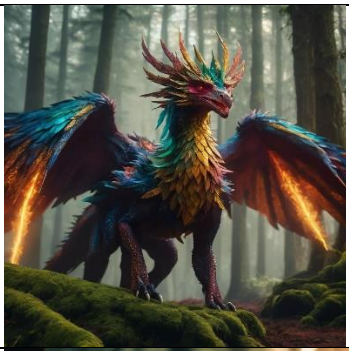
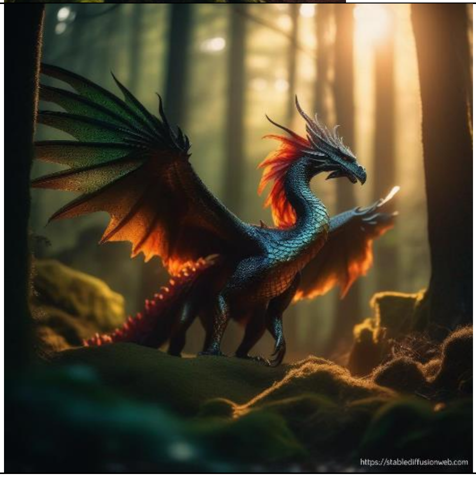
- Code 1: Uses a recursive GCD function.
- Code 2: Uses a loop to calculate GCD.

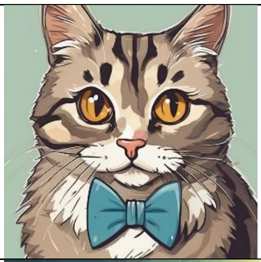
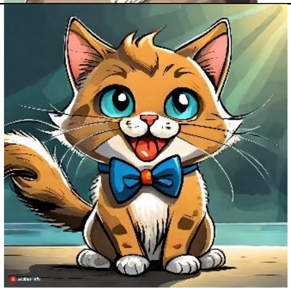
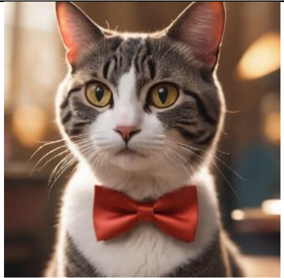
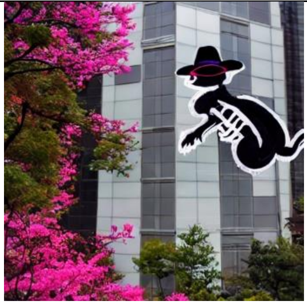
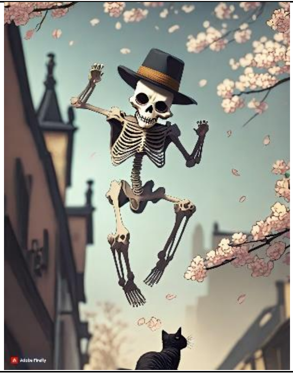
Prompt Message:

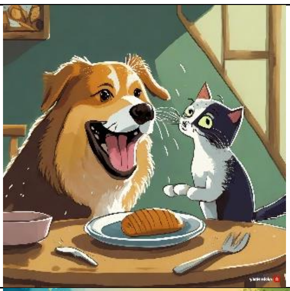
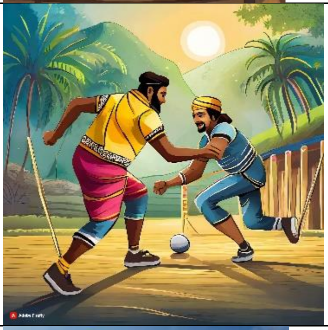
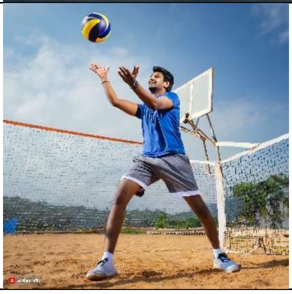
- Code 1: Asks the user to enter "two numbers."
- Code 2: Asks the user to enter "two numbers."

Task 2:

Tool	Output	Prompt	Findings
Stable Diffusion		Panda Playing Basketball	Seems like a normal photoshop rather than a image generate from scratch
Kindsky		Panda Playing Basketball	This one seemed like the best one out of the rest for this prompt, the hoop, the shoes and the size of the ball along with the expression, but it gets weird if you look closer around the ball region

Firefly		Panda Playing Basketball	A cute animated image, nails the court and everything, but firefly seems to be giving more animated images than realistic ones.
DALL-E		a mythical creature emerging from a forest. This creature should be a combination of a dragon and a phoenix, with iridescent scales and fiery wings.	Even though the fiery wings were missing, the majestic and mysterious persona looked amazing.
Gencraft			The nearest to the prompt in our opinion, nails almost everything.
Stable Diffusion			Again, missing the fiery wings but has a more realistic output.

Scribble		Generate a friendly, cartoon-style cat with a bowtie and a playful expression	I don’t know much about cats, but this isn’t playful.
Firefly		Generate a friendly, cartoon-style cat with a bowtie and a playful expression	Firefly with its animated output killing this prompt spot on.
Stable Diffusion		Generate a friendly, cartoon-style cat with a bowtie and a playful expression	Not cartoon styled and playful.
Zoo Replicate		skeleton jumping off of a roof, cherry blossom background, cat with a hat in the scene	Seemed like a poor photoshop to barely fit the prompt
Firefly		skeleton jumping off of a roof, cherry blossom background, cat with a hat in the scene	Firefly and Animated generation

Adobe firefly		dog and cat eating together	
Scribble diffusion		two person playing football	
Midjourney		a person smashing volleyball	

Text Generation:

To have some fun, we decided to ask a language model(our very own chat gpt) to generate a fun little story for us.

Prompt 1: mother Teresa commits felony, goes to jail and meets Mickey Mouse as a cellmate. he is imprisoned for arson. squidward is head correction officer. create short story

Output: In an improbable twist of fate, Mother Teresa found herself donning an orange jumpsuit, her halo replaced by a slightly askew prison cap. A misunderstanding had led to her incarceration, and now, she shared a cell with none other than Mickey Mouse. He, in his iconic red shorts, seemed to be the only one unfazed by the situation, cheerfully drawing caricatures of the prison guards.

Their cellmate dynamic took an absurd turn with Squidward Tentacles as the head correction officer. His perpetual grumpiness clashed hilariously with Mother Teresa's unwavering serenity, and Mickey's eternal optimism was like a splash of color in Squidward's monochromatic world.

Squidward's attempts at maintaining order in the prison involved playing his clarinet at the most inconvenient times, much to the annoyance of everyone, including the other guards.

As days passed, their interactions became a sitcom in the making. Mother Teresa tried to turn the prison yard into a garden, Mickey attempted to organize a talent show, and Squidward... well, Squidward mostly grumbled and played his clarinet, much to everyone's dismay.

One day, during a routine search, Squidward discovered Mickey's secret stash of cheese—yes, cheese—in a cell corner. The mouse claimed it was for emergencies, but Squidward's deadpan expression was priceless.

Deciding to make the most of their time, they orchestrated a clandestine cheese-tasting event, complete with a makeshift escape plan involving a giant cheese wheel. Mother Teresa tried to impart life lessons between bites, Mickey drew caricatures of the guards using cheese as ink, and Squidward reluctantly found himself enjoying a particularly exquisite Gouda.

Their antics didn't go unnoticed, and soon, the entire prison seemed to buzz with rumors about the odd trio's escapades. Even the toughest inmates found themselves cracking smiles at the absurdity of it all.

When their time was up, they bid farewell, Mother Teresa vowing to spread laughter and kindness, Mickey promising to host cheese-tasting events for the less fortunate, and Squidward... well, he grumbled a "goodbye" that sounded suspiciously like a fond farewell.

Their time together might have been unconventional, but their shared laughter and escapades left an indelible mark, proving that even within the confines of a prison, friendship and absurdity could thrive.