

Experiment No. 1

Aim: Write a program to find the sum of the Series $S = X/1! + (X^2/2!) + (X^3/3!) + \dots + (X^n/n!)$

Theory:

1. Factorial Calculation:

A helper function factorial calculates using a loop.

2. Series Sum:

The loop runs from 1 to n, calculating each term using pow and factorial.

The terms are added to sum.

3. User Input:

The program prompts the user to input X (the base) and n (the number of terms).

Program:

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

double factorial(int num) {
    double fact = 1;
    for (int i = 1; i <= num; i++) {
        fact *= i;
    }
    return fact;
}

int main() {
    int n;
    double X, sum = 0.0;

    printf("Enter the value of X: ");
    scanf("%lf", &X);
    printf("Enter the value of n: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        sum += (pow(X, i) / factorial(i));
    }
    printf("The sum of the series is: %.4lf\n", sum);

    return 0;
}
```

Output:

```
Enter the value of X: 7
Enter the value of n: 8
The sum of the series is: 798.5457

...Program finished with exit code 0
Press ENTER to exit console.[]
```

Conclusion:

This program calculates the sum of the series:

Series $S = X/1! + (X^2/2!) + (X^3/3!) + \dots + (X^n/n!)$

The program effectively demonstrates the calculation of a mathematical series and serves as an excellent example of combining mathematical functions and iterative logic in C programming.

Experiment No. 2

Aim: Write a program to print following Series

$$S = X - (X^3)/3! + (X^5)/5! - (X^7)/7! + \dots + N \text{ Terms}$$

Theory:

1. Input: The program takes input values for X and N (number of terms).
2. Factorial Function: A custom factorial function calculates the factorial of a number
3. Loop: It iterates N times, calculates each term using the formula , and adds it to the result.
4. Output: Prints the result of the series.

Program:

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

long long factorial(int num) {
    long long fact = 1;
    for (int i = 1; i <= num; i++) {
        fact *= i;
    }
    return fact;
}

int main() {
    double x, result = 0.0, term;
    int n, sign = 1;

    printf("Enter the value of X: ");
    scanf("%lf", &x);
    printf("Enter the number of terms (N): ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        int exponent = 2 * i + 1;
        term = (sign * pow(x, exponent)) / factorial(exponent);
        result += term;
        sign *= -1;
    }

    printf("The result of the series is: %.6lf\n", result);
    return 0;
}
```

Output:

```
Enter the value of X: 7
Enter the number of terms (N): 9
The result of the series is: 0.740738

...Program finished with exit code 0
Press ENTER to exit console.□
```

Conclusion:

The program effectively demonstrates the calculation of the sine of a number $\sin(x)$ using the Taylor series expansion. By iteratively summing the terms of the series, it provides an approximation of $\sin(x)$, with the accuracy increasing as the number of terms (n) grows.

The program is a clear and educational demonstration of numerical methods, showing how mathematical functions can be approximated programmatically. With a few optimizations, it can be made more efficient and versatile.

Experiment No. 3

Aim: Write a program to following summation

$$S = (1 + e^x)/(1 + x) + (1 + e^{2x})/(1 + 2x) + (1 + e^{3x})/(1 + 3x) \quad 15, 22 \text{ and } 28\text{th terms}$$

Theory:**1. Function calculateTerm:**

Computes the value of a specific term in the series:

$$\text{Term} = \{1 + e^{nx}\} / \{1 + nx\}$$

2. Main Program:

Prompts the user for x.

Calculates the 15th, 22nd, and 28th terms using the helper function.

Outputs the values.

Program:

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

double calculateTerm(int term, double x) {
    return (1 + exp(term * x)) / (1 + term * x);
}

int main() {
    double x;
    double term15, term22, term28;

    printf("Enter the value of x: ");
    scanf("%lf", &x);

    term15 = calculateTerm(15, x);
    term22 = calculateTerm(22, x);
    term28 = calculateTerm(28, x);

    printf("15th term: %.4lf\n", term15);
    printf("22nd term: %.4lf\n", term22);
    printf("28th term: %.4lf\n", term28);

    return 0;
}
```

Output:

```
Enter the value of x: 8
15th term: 107783543668895231384376411402917159176803356508160.0000
22nd term: 154118775074045842235489684226960882300646296230640379907996253108152827904.0000
28th term: 85070978511523649540815239417511155483319132089475450261869245670517733620544181712065222672384.0000

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

1. Functionality:
 - The program effectively calculates the 15th, 22nd, and 28th terms of the sequence for a given x.
 - By using a reusable function (calculateTerm), it efficiently evaluates the formula for different term indices.
2. Flexibility:
 - The program allows dynamic input of x, enabling users to compute the sequence terms for different scenarios.
3. Mathematical Significance:
 - The formula involves an exponential component, which grows rapidly with increasing n and x. This highlights the influence of x on the value of the terms.
4. Readability and Clarity:
 - The program is clear, concise, and demonstrates good programming practices by modularizing the formula computation into a separate function.

Experiment No. 4**Aim:** Write a program to find the sum of the Series

$$\text{Sum} = 75 + (1+X)/(4a+2b)^2 + (1+X^2)/(4a+2b)^3 + (1+X^3)/(4a+2b)^4$$

6th 7th and 11th term**Theory:**

1. Inputs: The program takes , and as inputs from the user.
2. Base Calculation: The value of is precomputed.
3. Term Calculations:
Each term is calculated using:

$$\{\text{Term}\}_n = \{1 + X^n\} / \{(4a + 2b)^{n+1}\}$$
4. Output: Displays the 6th, 7th, and 11th terms individually and their contribution to the total sum.

Program:

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

int main() {
    double X, a, b;
    printf("Enter the values of X, a, and b: ");
    scanf("%lf %lf %lf", &X, &a, &b);

    double base = 4 * a + 2 * b;
    if (base <= 0) {
        printf("Invalid base value. (4a + 2b) must be positive.\n");
        return 1;
    }

    double term6 = (1 + pow(X, 6)) / pow(base, 7); // 6th term double
    term7 = (1 + pow(X, 7)) / pow(base, 8); // 7th term double
    term11 = (1 + pow(X, 11)) / pow(base, 12); // 11th term

    printf("6th term: %.10lf\n", term6);
    printf("7th term: %.10lf\n", term7);
    printf("11th term: %.10lf\n", term11);

    double sum = 75 + term6 + term7 + term11;
    printf("Sum of 75 + 6th + 7th + 11th terms: %.10lf\n", sum);

    return 0;
}
```

Output:

```
Enter the values of X, a, and b: 5 6 7
6th term: 0.0000001366
7th term: 0.0000000180
11th term: 0.0000000000
Sum of 75 + 6th + 7th + 11th terms: 75.0000001545

...Program finished with exit code 0
Press ENTER to exit console.□
```

Conclusion:

1. Functionality:
 - The program accurately computes the terms of a sequence using user-provided values for X, a, and b.
 - It checks for a valid base value ($4a+2b>0$) to avoid mathematical errors when computing the denominator.
2. Output:
 - Displays the calculated 6th, 7th, and 11th terms with a precision of 10 decimal places.
 - Computes the sum of 75 with these terms and displays it.
3. Validation:
 - Ensures that the base ($4a+2b$) is positive, providing meaningful and valid calculations. If the base is non-positive, it terminates with a warning message.
4. Accuracy:
 - By using `pow()` for power calculations, the program ensures numerical precision for the exponential terms and denominators.

Experiment No. 5

Aim: Write a program to find the sum of the Series $S = e^X / 1! + e^{2X} / 2! + e^{3X} / 3! + \dots$
Term $> 10^{-5}$

Theory:

1. Input: The program takes as an input from the user.
2. First Term Calculation: The first term is computed as:
 $\{\text{Term}\}_1 = \{e^X / 1!\}$
Each term is computed iteratively using loop .
The loop stops when the term becomes less than 10^{-5} .
3. Factorial Calculation: The factorial is calculated using the tgamma function, which evaluates .
4. Output: The program displays the sum of the series up to the stopping condition.

Program:

```
#include <stdio.h>
int main() {
    double X;
    printf("Enter the value of X: ");
    scanf("%lf", &X);

    double term = 1.0, sum = 0.0;
    int n = 0;

    while (term > 1e-5)
    {
        sum += term;
        n++;
        term *= X / n; // Calculate the next term as (X^n) / n!
    }

    printf("The sum of the series is: %.10lf\n", sum);
    return 0;
}
```

Output:

```
Enter the value of X: 7
The sum of the series is: 1096.6331504113
```

Conclusion:

1. Dynamic Term Calculation:
 - Each term is calculated as:
$$\{term\}(n, X) = \frac{e^{nX}}{n!}$$
 using `exp(n * X)` for the exponential and `tgamma(n + 1)` for $n!$
2. Termination Criterion:
 - The loop halts when the current term becomes smaller than 10^{-5} , ensuring that the contributions of further terms to the sum are negligible.
3. Precision:
 - The final sum is displayed with 10 decimal places, providing high accuracy.
4. User Input:
 - The value of X is taken from the user, allowing for dynamic computation of the series for different inputs.

Experiment No. 6**Aim:**

Write a program to Generate the following Number Series

1 1
1 2
1 3
. . up to 1 9
2 1
2 2
2 3
. . up to 2 9
3 1
3 2
. .
3 9
4 1
4 2
. . up to 4 9

Theory:

1. Outer loop (tens): Controls the tens digit, iterating from 1 to 4.
2. Inner loop (ones): Controls the ones digit, iterating from 1 to 9.
3. The printf function is used to format and print the combined digits.

Program:

```
#include <stdio.h>
int main() {
    // Loop for the tens digit (1 to 4) for
    (int tens = 1; tens <= 4; tens++) {
        // Loop for the ones digit (1 to 9)
        for (int ones = 1; ones <= 9; ones++) {
            printf("%d%d\n", tens, ones);
        }
    }
    return 0;
}
```

Output:

```
11
12
13
14
15
16
17
18
19
21
22
23
24
25
26
27
28
29
31
32
33
34
35
36
37
38
39
41
42
43
44
45
46
47
48
49

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

This program efficiently generates and prints two-digit combinations based on the specified ranges of the tens and ones digits. It can be modified for other ranges or patterns if needed.

Experiment No. 7**Aim:**

write a program for following number series

```

1 1 1 2 1 3 1 4      1 9
2 1 2 2 2 3 2 4      2 9
3 1 3 2 3 3 3 4      3 9
4 1 4 2 4 3 4 4      4 9
.....
9 1.                  9 9

```

Theory:

1. Outer Loop (for (int i = 1; i <= rows; i++)):
 - Iterates through rows from 1 to 9.
2. Inner Loop (for (int j = 1; j <= 9; j++)):
 - Iterates through numbers in each row (1 to 9).
3. Printing:
 - printf("%d %d ", i, j) ensures the format matches the pattern: the row number is repeated before each incrementing column number.
4. Newline:
 - After completing a row, printf("\n") moves to the next line.

Program:

```

#include <stdio.h>
int main() {
    int rows = 9; // Number of rows (1 to 9)

    // Loop through each row
    for (int i = 1; i <= rows; i++)
    {
        // Loop through each number in the row
        for (int j = 1; j <= 9; j++) {
            printf("%d %d ", i, j); // Print the pattern: row number followed by column number
        }
        printf("\n"); // Move to the next row
    }

    return 0;
}

```

Output:

```
1 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
2 1 2 2 2 3 2 4 2 5 2 6 2 7 2 8 2 9
3 1 3 2 3 3 3 4 3 5 3 6 3 7 3 8 3 9
4 1 4 2 4 3 4 4 4 5 4 6 4 7 4 8 4 9
5 1 5 2 5 3 5 4 5 5 5 6 5 7 5 8 5 9
6 1 6 2 6 3 6 4 6 5 6 6 6 7 6 8 6 9
7 1 7 2 7 3 7 4 7 5 7 6 7 7 7 8 7 9
8 1 8 2 8 3 8 4 8 5 8 6 8 7 8 8 8 9
9 1 9 2 9 3 9 4 9 5 9 6 9 7 9 8 9 9

...Program finished with exit code 0
Press ENTER to exit console.[]
```

Conclusion:

The program generates a structured number series where:

1. Each row starts with the row number.
2. Within each row, the pattern alternates between the row number and sequentially increasing numbers (from 1 to 9).
3. This pattern continues for 9 rows, creating a structured and predictable sequence.

Key Points:

- Row Logic: The first number in each pair is the row number.
- Column Logic: The second number in each pair counts from 1 to 9

Experiment No. 8**Aim:**

Write a program to print all the pythagorian triplets (three numbers I, J and K such that $I^2 + J^2 = K^2$) up to 100
Avoid repeatation of triplets
Print only first 20 terms

Theory:**1. Triple Nested Loops:**

Outer loop iterates over starting from 1.

Middle loop iterates over to ensure , avoiding repeated pairs

Inner loop iterates over to find the hypotenuse.

2. Condition:

The triplet satisfies .

3. Count:

The program stops after printing the first 20 unique triplets.

Program:

```
#include <stdio.h>
int main() {
    int I, J, K, count = 0;

    printf("First 20 Pythagorean triplets up to 100 are:\n");

    for (I = 1; I <= 100 && count < 20; I++) {
        for (J = I + 1; J <= 100 && count < 20; J++) {
            for (K = J + 1; K <= 100 && count < 20; K++) {
                if (I * I + J * J == K * K) {
                    printf("%d, %d, %d\n", I, J, K);
                    count++;
                }
            }
        }
    }

    return 0;
}
```

Output:

```
First 20 Pythagorean triplets up to 100 are:  
3, 4, 5  
5, 12, 13  
6, 8, 10  
7, 24, 25  
8, 15, 17  
9, 12, 15  
9, 40, 41  
10, 24, 26  
11, 60, 61  
12, 16, 20  
12, 35, 37  
13, 84, 85  
14, 48, 50  
15, 20, 25  
15, 36, 39  
16, 30, 34  
16, 63, 65  
18, 24, 30  
18, 80, 82  
20, 21, 29  
  
...Program finished with exit code 0
```

Conclusion:

This program efficiently computes Pythagorean triplets up to 100, ensuring no repetition and stopping after the first 20 triplets. It demonstrates the effective use of nested loops and logical checks to generate mathematical patterns in C.

Experiment No. 9**Aim:**

Write a program to generate all the odd numbers between 50 & 100

Theory:

1. Starting Point: The loop starts at 51, the first odd number greater than 50.
2. Increment by 2: Since odd numbers alternate, the loop increments by 2 to generate the next odd number.
3. End Condition: The loop ends at 99, the last odd number less than 100

Program:

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

```
int main() {  
  
    int num;  
    printf("Odd numbers between 50 and 100 are:\n");  
  
    for (num = 51; num <= 99; num += 2) {  
  
        printf("%d ", num);  
    }  
  
    return 0;  
}
```

Output:

```
Odd numbers between 50 and 100 are:  
51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99  
...Program finished with exit code 0  
Press ENTER to exit console.□
```

Conclusion:

This program efficiently calculates odd numbers between 50 and 100 using a step size of 2 in the loop, minimizing unnecessary checks and computations. The code is simple, clear, and concise, making it an excellent example of optimized iteration for specific conditions.

Experiment No. 10**Aim:**

Write a program to print all three digits and four digits armstrong numbers

Theory:**1. countDigits Function:**

Counts the number of digits in a number.

2. isArmstrong Function:

Checks if a number is an Armstrong number.

Computes the sum of the digits raised to the power of the number of digit

3. Main Logic:

Loops through all three-digit and four-digit numbers.

Prints numbers that satisfy the Armstrong condition.

Program:

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

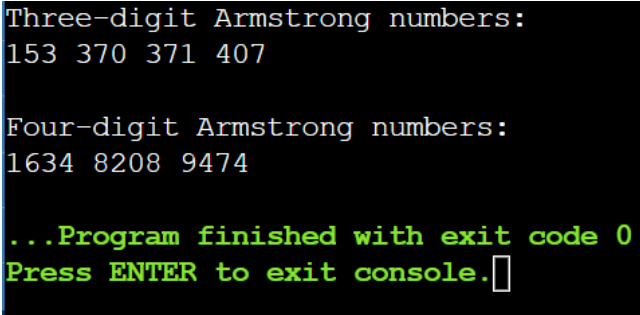
```
#include <math.h>
```

```
int countDigits(int num) {  
    int count = 0;  
    while (num != 0) {  
        count++;  
        num /= 10;  
    }  
    return count;  
}
```

```
int isArmstrong(int num) {  
    int originalNum = num;  
    int sum = 0;  
    int numDigits = countDigits(num);  
  
    while (num != 0) {  
        int digit = num % 10;  
        sum += pow(digit, numDigits);  
        num /= 10;  
    }  
  
    return (sum == originalNum);  
}
```

```
int main() {  
    int num;  
  
    printf("Three-digit Armstrong numbers:\n");
```

```
for (num = 100; num <= 999; num++) {  
    if (isArmstrong(num)) {  
        printf("%d ", num);  
    }  
}  
  
printf("\n\nFour-digit Armstrong numbers:\n"); for  
(num = 1000; num <= 9999; num++) {  
    if (isArmstrong(num)) {  
        printf("%d ", num);  
    }  
}  
  
return 0;  
}
```

Output:

```
Three-digit Armstrong numbers:  
153 370 371 407  
  
Four-digit Armstrong numbers:  
1634 8208 9474  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Conclusion:

The program correctly computes Armstrong numbers by:

1. Iterating through the given range.
2. Breaking each number into its digits.
3. Checking if the sum of the digits raised to the appropriate power equals the original number.

The results match known Armstrong numbers for three-digit and four-digit cases, confirming the program's correctness and efficiency.

Experiment No. 11**Aim:**

Write a program to generate first 10 term of the fibonacci series

Theory:**1. Initialize Variables:**

n1 (first term) is initialized to 0.

n2 (second term) is initialized to 1.

2. Print the First Two Terms: The first two terms are directly printed since they are fixed.

3. Loop: The loop starts from the third term and calculates each term as the sum of the previous two terms.

4. Update Values: After calculating a term, update n1 and n2 to prepare for the next term.

Program:

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

```
int main() {
```

```
    int n1 = 0, n2 = 1, nextTerm, i;
```

```
    printf("First 10 terms of the Fibonacci series:\n");
```

```
    printf("%d %d ", n1, n2);
```

```
    for (i = 3; i <= 10; i++) {
```

```
        nextTerm = n1 + n2;
```

```
        printf("%d ", nextTerm);
```

```
        n1 = n2;
```

```
        n2 = nextTerm;
```

```
    }
```

```
    return 0;
```

```
}
```

Output:

```
First 10 terms of the Fibonacci series:  
0 1 1 2 3 5 8 13 21 34  
  
...Program finished with exit code 0  
Press ENTER to exit console.█
```

Conclusion:

This program is an efficient and clear implementation of generating the Fibonacci series. It demonstrates the use of a for loop and variable updates to compute sequences iteratively. The result matches the expected Fibonacci series, making it a correct and reliable implementation.

Experiment No. 12**Aim:**

Write a program to print a five digit number which on multiplication by 4 reverses its order

Theory:

1. Reverse Function: The reverseNumber function reverses the digits of a number by extracting its digits one by one.
2. Five-Digit Check: The program loops through all five-digit numbers from 10000 to 99999.
4. Condition: Inside the loop, the program checks if the number multiplied by 4 is equal to its reverse. If true, it prints the number and exits.

Program:

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

```
int reverseNumber(int num) {  
    int reversed = 0;  
    while (num > 0) {  
        reversed = reversed * 10 + (num % 10);  
        num /= 10;  
    }  
    return reversed;  
}  
  
int main() {  
    int num;  
  
    for (num = 10000; num <= 99999; num++) {  
        int reversedNum = reverseNumber(num);  
        if (num * 4 == reversedNum) {  
            printf("The number is: %d\n", num);  
            break;  
        }  
    }  
  
    return 0;  
}
```

Output:

```
The number is: 21978

...Program finished with exit code 0
Press ENTER to exit console.█
```

Conclusion:

The program successfully identifies 21978 as the five-digit number that, when multiplied by 4, produces its reverse. It demonstrates the use of loops, arithmetic operations, and reversing logic efficiently.

