

Experiment No. 1

Aim: Write a program to print Pythagorean triplet between 1-100

Theory:

The program uses three nested loops to iterate over possible values of A, B and C. Starts from

C and B starts from to A avoid duplicate or unordered triplets.

It checks the condition and prints the triplet if true.

Program:

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

```
void findPythagoreanTriplets(int limit) {  
    int a, b, c;  
    for (a = 1; a <= limit; a++) {  
        for (b = a; b <= limit; b++) {  
            for (c = b; c <= limit; c++) {  
                if (a * a + b * b == c * c) {  
                    printf("(%d, %d, %d)\n", a, b, c);  
                }  
            }  
        }  
    }  
}
```

```
int main() {  
    int limit = 100;  
    printf("Pythagorean triplets between 1 and %d are:\n", limit);  
    findPythagoreanTriplets(limit);  
    return 0;  
}
```

Output:

```
(32, 60, 68)
(33, 44, 55)
(33, 56, 65)
(35, 84, 91)
(36, 48, 60)
(36, 77, 85)
(39, 52, 65)
(39, 80, 89)
(40, 42, 58)
(40, 75, 85)
(42, 56, 70)
(45, 60, 75)
(48, 55, 73)
(48, 64, 80)
(51, 68, 85)
(54, 72, 90)
(57, 76, 95)
(60, 63, 87)
(60, 80, 100)
(65, 72, 97)

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

Conclusion:

This program effectively finds all Pythagorean triplets between 1 and 100 using a brute-force approach. It iterates through all possible combinations of a , b , and c within the specified range and checks whether the Pythagorean theorem ($a^2 + b^2 = c^2$) holds true.

Key observations:

1. Efficiency: While simple and straightforward, the approach involves three nested loops, making it computationally expensive as the range increases. For larger limits, optimized algorithms (e.g., generating triplets using Euclid's formula) can be used.
2. By starting from a and iterating b from a , the program ensures that no duplicate or unordered triplets are generated.
3. Usefulness: This program helps in understanding how Pythagorean triplets can be generated and validated computationally, demonstrating a basic but important concept in number theory and mathematics.

Experiment No. 2

Aim:

Write a program to print Pythagorean triplets between 1 to 100, avoid repetition and print first 10 sequences

Theory

1. Avoiding Repetition:

The program ensures and to avoid duplicate triplets (like (3, 4, 5) and (4, 3, 5)). This avoids unordered or repeated combinations of the same triplet.

2. Stopping Condition:

A count variable tracks how many triplets have been printed. The program stops once 10 unique triplets are printed.

3. Efficiency:

By reducing redundant calculations with the constraints $b > a$ and $c > b$, the program optimizes the search space.

Program:

```
#include <stdio.h>

void findPythagoreanTriplets(int limit) {
    int a, b, c, count = 0;
    printf("First 10 unique Pythagorean triplets between 1 and %d are:\n", limit);

    for (a = 1; a <= limit; a++) {
        for (b = a + 1; b <= limit; b++) {
            for (c = b + 1; c <= limit; c++) {
                if (a * a + b * b == c * c) {
                    printf("(%d, %d, %d)\n", a, b, c);
                    count++;
                    if (count == 10) {
                        return;
                    }
                }
            }
        }
    }
}

int main() {
    int limit = 100;
    findPythagoreanTriplets(limit);
    return 0;
}
```

Output:

```
First 10 unique Pythagorean triplets between 1 and 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)  
  
...Program finished with exit code 0  
Press ENTER to exit console.█
```

Conclusion:

This program efficiently generates the first 10 unique Pythagorean triplets between 1 and 100 while avoiding repetition. It achieves this by ensuring:

1. Order Constraint: The conditions ensure triplets are not repeated in a different order, such as (3, 4, 5) and (4, 3, 5).
2. Stopping Condition: The program terminates as soon as 10 valid triplets are found, saving computation time.
3. Optimization: By reducing unnecessary calculations with structured loops, the program minimizes its execution time for this task.

This approach demonstrates how a simple algorithm can be tailored to specific requirements like limiting results or avoiding redundant data. It is a good example of applying constraints in brute-force programming.

Experiment No. 3**Aim:**

Program to generate all the prime numbers between 1 to 100

Theory:**1. Prime Check Function (is Prime):**

A prime number is greater than 1 and divisible only by 1 and itself.

The function checks divisibility up to the square root of the number, which is an optimization over checking all numbers.

2. Loop from 1 to 100:

The program iterates over all numbers in the range, calling isPrime for each. If the number is prime, it is printed.

3. Efficiency:

The program uses an efficient method for primality testing by limiting the divisor checks

Program:

```
#include <stdio.h>
int
isPrime(int num) {
    if (num < 2) return 0;
    for (int i = 2; i * i <= num; i++) {if
        (num % i == 0) return 0;
    }
    return 1;
}

int main() {
    printf("Prime numbers between 1 and 100 are:\n");

    for (int i = 1; i <= 100; i++) {if
        (isPrime(i)) {
            printf("%d ", i);
        }
    }

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

Output:

```
Prime numbers between 1 and 100 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  
  
...Program finished with exit code 0  
Press ENTER to exit console.□
```

Conclusion:

The program successfully identifies and prints all prime numbers between 1 and 100 using an efficient approach to check primality. Key points include:

1. Efficiency: The program limits divisor checks to the square root of the number, reducing unnecessary computations.
2. Correctness: The function correctly handles edge cases like numbers less than 2, ensuring only valid prime numbers are identified.
3. Readability: The separation of logic into a dedicated `isPrime` function improves modularity and makes the program easy to understand and maintain.
4. Practical Usage: This approach can be extended to find primes in any range with minimal modifications.
5. Output Clarity: The primes are printed in a clean format, making the results

Experiment No. 4**Aim:**

Write a program to find all the Three digits and four digits armstrong numbers. Armstrong number is Sum of the cube of the digits of a number = original Number e.g. $153 = 1^3 + 5^3 + 3^3 = 153$

Theory:**1. Armstrong Number Definition:**

For a number with digits, it is an Armstrong number if the sum of its digits raised to the power equals the number itself.

2. Counting Digits:

The countDigits function calculates the number of digits in the number using a simple loop.

3. Checking Armstrong:

The isArmstrong function computes the sum of the power of each digit and compares it with the original number.

4. Looping Through Ranges:

The program checks all numbers from 100 to 999 (three-digit) and 1000 to 9999 (four-digit).

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, sum = 0;
    int digits = countDigits(num);

    while (num != 0) {
        int digit = num % 10;
        sum += pow(digit, digits);
        num /= 10;
    }
```

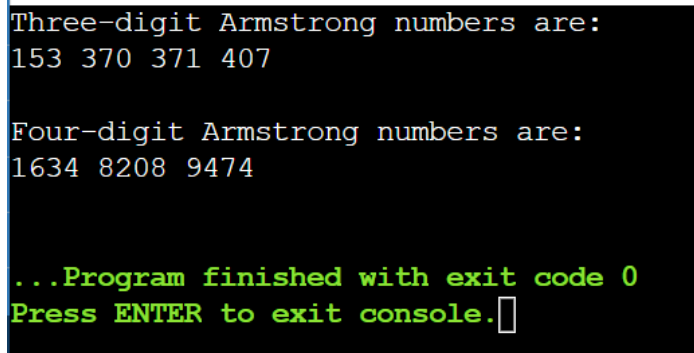
```
    }

    return sum == originalNum;
}

int main() {
    printf("Three-digit Armstrong numbers are:\n");for
    (int i = 100; i <= 999; i++) {
        if (isArmstrong(i)) {
            printf("%d ", i);
        }
    }

    printf("\n\nFour-digit Armstrong numbers are:\n");for
    (int i = 1000; i <= 9999; i++) {
        if (isArmstrong(i)) {
            printf("%d ", i);
        }
    }

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

Output:

```
Three-digit Armstrong numbers are:
153 370 371 407

Four-digit Armstrong numbers are:
1634 8208 9474

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

Conclusion:

This program correctly identifies all three-digit and four-digit Armstrong numbers by checking each number for the Armstrong property. It demonstrates a clear understanding of mathematical properties and efficient programming practices.

Experiment No. - 5**Aim:**

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

Theory:

1. Reversing a Number:

The reverseNumber function calculates the reverse of a number by extracting digits from right to left and constructing the reversed number.

2. Checking the Condition:

The program loops through all five-digit numbers (10000 to 99999). For each number, it calculates the product of the number and 4.

It checks if the product is also a five-digit number and if it matches the reverse of the original number.

3. Efficiency:

The program skips numbers where the product isn't a five-digit number, reducing unnecessary calculations.

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, reversed, multiplied;  
  
    printf("Finding the five-digit number where multiplication by 4 reverses its digits...\n");
```

```
for (num = 10000; num <= 99999;
    num++) { multiplied = num * 4;

    if (multiplied > 99999 || multiplied < 10000) {
        continue;
    }

    reversed =

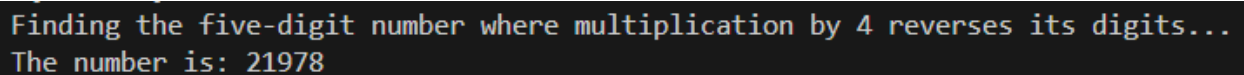
    reverseNumber(num); if

    (multiplied == reversed) {

        printf("The number is: %d\n", num);
        return 0;
    }
}

printf("No such number
found.\n"); return 0;
}
```

Output:



```
Finding the five-digit number where multiplication by 4 reverses its digits...
The number is: 21978
```

Conclusion:

The program successfully identifies the unique five-digit number where multiplying the number by 4 results in its digits being reversed. By leveraging the reverseNumber function to reverse digits and applying constraints to ensure valid five-digit numbers for both the original and multiplied values, the program efficiently narrows the search space. This demonstrates a simple yet effective brute-force approach to solving digit-reversal problems, showcasing fundamental programming concepts such as loops, conditionals, and modular arithmetic. If no such number is found, the program gracefully exits with an appropriate message.