Problem Solving in C and Python
Programming Exercises and Solutions

Part II: Functions

By Yana Kortsarts and Yulia Kempner

Table of Content

Preface

This book is a collection of exercises for the introductory programming course. This book is continuation of the book: Problem Solving in C and Python: Programming Exercises and Solutions, Part 1 (https://www.smashwords.com/books/view/879372)

In this part of the book we are focusing on C and Python programming languages and specifically on the topic of functions. The book provides a short overview of the necessary theoretical material sample solutions written in Python 3 (the most recent version of Python at the time the book was written), and a list of practice exercises in increasing order of difficulty.

Disclaimer: most of the problems are not original problems; in some cases, the appropriate references are provided, but in many cases the problems are drawn from mathematical and programming folklore.

Chapter I: Simple Functions

I. Short Theory Overview

A function is a reusable group of statements that performs a task.

Functions help to divide large task into several subtasks. This approach is a problem-solving technique known as modular programming or top-down design or stepwise refinement. Instead of writing a program as a long sequence of statements, it is systematically divided into several smaller tasks (functions/modules). These smaller functions are executed in the desired order to perform the overall task.

Functions help to design simpler code that is easier to maintain, test, and debug. Functions help to reduce the duplication of code within a program. Same functions can be used by more than one program. This benefit of using functions is known as code reuse.

1. **Function Definition and Function Call**

Function definition in Python

The syntax for user-defined function in Python is as follows:

```
def function_name(formal parameters): #function header
     statements
     return expression #return statement
```

Function header starts with `def` keyword, followed by the `function_name`, an optional list of comma-separated formal parameters enclosed in the required parentheses, and a final colon. A function can have no parameters, one parameter, or several parameters. If a function doesn't have any parameters, empty parentheses are still required. The rules for naming a function and a variable are the same. The `function_name` can contain only alphanumeric characters and underscores, and must start with the letter or an underscore. While choosing the function name, it is advisable to choose the meaningful name that is related to the function's task and avoid using Python keywords.

All function statements, including `return` statement, must be indented. Function statements can include any type of statements including other function definitions (these are called inner or nested functions). In this book we will not cover examples of inner (nested) functions.

The purpose of the `return` statement is to send function's results back to the caller and return control to the calling function. The return statement consists of the keyword `return` followed by the optional `return value` (`expression`). The `return value` could be any Python object. The `return statement` and the `return value` are optional. If a function doesn't have `return statement` or have an empty `return statement` (no `return value`), function returns special value called `None`. In Python, a function can return multiple values by separating them by commas in the single `return statement`. In Python, comma-separated values are treated as `tuples`. We will discuss `tuples` in later chapters.

<u>Function definition in C</u>

```
type_return_value function_name(formal parameters with types){
      statements

      return expression
}
```

In C, indentation is not required but highly recommended to ensure the code is easy to read.

In C, function definition starts with type of return value followed by the `function_name` and declaration of formal parameters. Names and types of formal parameters must be enclosed in the parenthesis. Function statements must be enclosed in curly braces. A function can have no parameters, one parameter, or several parameters. If there is more than one parameter, they are separated by commas. In the case of no parameters, we can keep the parentheses empty or write the word `void`.

The rules for naming a function and a variable are the same. The `function_name` can contain only alphanumeric characters and underscores, must start with the letter or an underscore, and cannot be a reserved keyword. While choosing the function name, it is advisable to choose the meaningful name that is related to the function's task.

As in Python, the purpose of the `return` statement is to send function's results back to the caller and return control to the calling function. The return statement consists of the keyword `return` followed by the optional `return value` (`expression`). Both the `return statement` and the `return value` are optional. If a function doesn't return any value, the type of the return value is `void.` In C, we can only return a single value.

Function statements can include any type of statement except other function definition. Inner or nested functions are not supported by C.

2. **Function Call**

The syntax to call a function in Python and C is as follows:

```
function_name(actual parameters)
```

A function call could be placed within other statements or on its own depending on the function's return value (see examples later in this chapter).

Formal parameters are the parameters used in the function definition, while actual parameters (arguments) are the parameters used in the function call.

3. **Passing Parameters (Arguments) to the Function**

In C, parameters (arguments) are passed to a function by value. In the call-by-value (pass-by-value) method, a function receives the value of the variable. The values of the actual parameters are copied to the function's formal parameters and stored in different memory locations. Any changes to the values of the actual parameters made inside the function have no effect outside of the function since only a copy of the variable is passed to the function. If we need to change the value of a variable through a function, it can be done using the return statement (examples will be shown later in this part of the book) or pointers, which are not covered in this part of the book.

In Python, parameters (arguments) are passed by assignment or by object reference. All data types in Python are objects, which fall into two categories: mutable and immutable. Mutable objects can be changed, whereas immutable objects cannot be changed after they are created. Built-in types like int, float, bool, and str, which we deal with in this part of the book, are immutable. When an immutable object is passed as a parameter (argument), it behaves similarly to call-by-value in C, and its value cannot be changed through a function. If we need to change the value of a variable through a function, a return statement can be used (examples will be shown later in this book). Mutable objects are not covered in this part of the book.

4. Function `main()`

Many programming languages have a special function, typically called `main()`, which serves as the starting point for program execution.

In Part I of the book (https://www.smashwords.com/books/view/879372), we already used the `main()` function while writing solutions in the C programming language. All C programs must have a `main()` function.

In Python, `main()` is not required, but for the purposes of this book we will follow the programming design approach that includes a `main()` function in each Python program. In our Python examples, the function `main()` doesn't have any parameters and doesn't return any values, and will be called first, to ensure the program execution starts from `main()`.

In C, we will use `int main()` and the main function will return 0 indicating successful execution of the program. It is possible to use `void main()` in C, and in this case, the function `main()` doesn't have any return value.

5. Function Declaration and Definition

Before a program can use any function, that function must be defined. In Python, when code is written in a single file and the `main()` function is defined to execute first, all function definitions can be written in any order before the `main()` function.

In C, there are two different approaches. One approach is similar to Python where all function definitions are written before the `main()` function. While using this approach in C, the order of function definitions is important, and they must follow the order of function calls. The second approach, which is more common, is to write all function prototypes (declarations) before `main()` in arbitrary order and function definitions after `main()`. The function prototype consists of the function header only, followed by a semi-colon. This means the function prototype includes the type of return value, function name, and the list of parameters with their types enclosed in parentheses. The names of the parameters are optional in the function prototype. It is advisable to declare all functions that are used in the program before main in arbitrary order. When the program is written in several files, the function definition is written in one file only, but the function prototype (declaration) must be included in all files where the function is used. We have already used this approach with library functions.

6. Variable Scope: Local Variables

In Python and C, the scope of a variable is the block of code in the entire program where the variable is declared and can be accessed. In C, every block enclosed within curly braces defines a new scope within the program. In Python, indentation defines the scope of the program. Formal parameters used in function definitions and variables declared inside the function are called local variables. Local variables can only be accessed within the specific function where they are defined. Different functions can have local variables with the same name. Access to local variables ends when the function execution is completed.

II. User-Defined Functions and Sample Solutions in Python and C

A function is not required to accept parameters or return values. It can perform both, either, or neither.

1. Example of the `main()` Function in Python

In Python, `main()` is not required. However, for the purposes of this book, we will follow the programming design approach that includes a `main()` function in each Python program. In our Python examples, the `main()` function doesn't have any parameters and doesn't return any values. It will be called first to ensure the program execution starts from `main()`.

Below, we show an example of the Python program using function `main()`.

Program 1:

Write a program that reads 10 integers and finds the number of even inputs.

We can write solution without using any functions as we did in Part I, but here we will demonstrate how we to use function `main()`.

```
def main():
    i=0
    count_even=0
    for i in range(10):
        num=int(input("enter an integer "))
        if(num%2==0):
            count_even+=1
    print("there are",count_even,"evens")
main()
```

2. Functions with a Single Return Value

In this section, we will see examples of programs that use one or more functions (in addition to the `main()` function), each with a single return value. To demonstrate the concept of function parameters, we will provide examples with no parameters, one parameter, and multiple parameters. Since a function is a small program, we can think of function parameters as an input and function return value as an output. Typically, there are two types of parameters: formal parameters used in function definitions, and actual parameters (arguments) used in function calls.

Program 1:

Write a function, `sum_two_ints`, that takes two integer parameters and returns their sum. Write a program to test the function with two integer inputs (solutions 1 and 2) and with 10 pairs of integers (solution 3).

<u>Solution 1 in Python</u>

```
#function definition starts with a keyword def followed by a
#function name, and a list of formal parameters.

def sum_two_ints(num1, num2):

    #num1 and num2 are formal parameters
    result=num1+num2
    #result is a local variable for this function
    return result #return statement

def main():

    #This solution demonstrates that actual and formal
    #parameters can have the same names

    #num1 and num2 are actual parameter(arguments)
    num1=int(input("enter first integer "))
    num2=int(input("enter second integer "))

    #function call
    result=sum_two_ints(num1, num2)
    print("sum of the inputs is",result)


main()#calling function main
```

<u>Solution 2 in Python</u>

```
def sum_two_ints(num1, num2):

    result=num1+num2
    return result

def main():
```

```
    #This solution demonstrates that actual and formal
    #parameters can have different names, and the function call
    #can be placed within a print statement

    #a and b are actual parameters (arguments)
    a=int(input("enter first integer "))
    b=int(input("enter second integer "))

    #function call within a print statement
    print("sum of the inputs is",sum_two_ints(a,b))


main()
```

<u>Solution 3 in Python</u>

```
def sum_two_ints(num1, num2):

    result=num1+num2
    return result

def main():

    #This solution demonstrates function call within the loop

    for i in range(10):

        a=int(input("enter first integer "))
        b=int(input("enter second integer "))
        print(a,"+",b,"=",sum_two_ints(a,b))

main()
```

<u>Solution 1 in C</u>
<u>Version 1: Function definition is written before the `main()` function.</u>

```
#include<stdio.h>
/* The function definition starts with the return value type
int, followed by the function name and a declaration of the
formal parameters enclosed in parentheses. The function body is
enclosed in curly braces. */

int sum_two_ints(int num1, int num2){

    //num1 and num2 are formal parameters
    int result;
```

```
    //result is a local variable for this function
    result = num1 + num2;
    return result; //return statement
}
int main() {

    /* This solution demonstrates that actual and formal
    parameters can have the same names */
    int num1, num2, result;
    printf("enter two integers\n");
    //num1 and num2 are actual parameters(arguments)
    scanf("%d", &num1);
    scanf("%d", &num2);

    //function call
    result = sum_two_ints(num1, num2);
    printf("sum of the inputs is %d\n", result);

}
```

Version 2: The function prototype (declaration) is written before `main()` and the function definition is written after the `main()` function.

```
#include<stdio.h>

int sum_two_ints(int num1, int num2); //function prototype

/* the function prototype is written before main()
and includes the function definition header: the type of the
return value, function name, and the list of parameters, which
includes their names (optional) and types (required). The
function prototype ends with a semicolon. The names of the
parameters in the function prototype ARE OPTIONAL, and we can
write the function prototype as follows:

int sum_two_ints(int,int);

*/

int main() {

    int num1, num2, result;
    printf("enter two integers\n");
    scanf("%d", &num1);
    scanf("%d", &num2);
    result = sum_two_ints(num1, num2);
    printf("sum of the inputs is %d\n", result);
```

```
}

//Function definition
int sum_two_ints(int num1, int num2) {
    int result;
    result = num1 + num2;
    return result;
}
```

Solution 2 in C

```
#include<stdio.h>
int sum_two_ints(int, int); //function prototype

int main() {

    /* This solution demonstrates that actual and formal
    parameters can have different names and the function
    call can be placed within print statement */

    int a, b;
    //a and b are actual parameters (arguments)
    scanf("%d", &a);
    scanf("%d", &b);

    //function call within print statement
    printf("sum of the inputs is %d\n", sum_two_ints(a, b));

}
//Function definition
int sum_two_ints(int num1, int num2) {
    int result;
    result = num1 + num2;
    return result;
}
```

Solution 3 in C

```
#include<stdio.h>
int sum_two_ints(int, int); //function prototype

int main() {


    /* This solution demonstrates a function call within the
loop */
```

```
    int i, a, b;
    for (i = 0; i < 10; i++) {

        scanf("%d", &a);
        scanf("%d", &b);
        printf("%d+%d=%d\n", a, b, sum_two_ints(a, b));
    }
    return 0;
}

//Function definition
int sum_two_ints(int num1, int num2) {
    int result;
    result = num1 + num2;
    return result;
}
```

Program 2:

Write a function `ave_3` that takes 3 integer parameters and returns their average. Write a program that repeats the following task 5 times: the program reads three integers and finds their average using the `ave_3` function.

<u>Solution in Python</u>

```
def ave_3(a,b,c):
     #a,b, and c are formal parameters
     return (a+b+c)/3
     #calculations can be performed in the return statement

def main():

     for i in range(5):
          a=int(input("enter integer "))
          b=int(input("enter integer "))
          c=int(input("enter integer "))

          #a, b, and c are actual parameters

          #we can call the function and assign the return value
          #to a variable
          res=ave_3(a,b,c)
          print("their average is", res)

main()
```

Solution in C

```c
#include<stdio.h>
//function prototype (declaration)
double ave_3(int, int, int);

int main() {
    int a, b, c, i;
    double res;
    for (i = 0; i < 5; i++) {
        scanf("%d", &a);
        scanf("%d", &b);
        scanf("%d", &c);
        /*a, b, and c are actual parameters. */

        /* we can call the function and assign the return value
        to a variable */
        res = ave_3(a, b, c);
        printf("their average is %f\n", res);

    }
    return 0;
}
// function definition
double ave_3(int a, int b, int c) {
    //a,b, and c are formal parameters

    return (a + b + c) / 3.0;
    //calculations can be performed in the return statement

}
```

Program 3:

Write a function square that takes one integer parameter and returns its square. Write a program that reads 10 integers and prints the square of each input number.

Solution in Python

```python
def square(a):
     #a is formal parameter
     return a*a #a**2
def main():
     for i in range(10):
          num=int(input("enter integer "))
          print(num,"^2=",square(num), sep='');
          #num is an actual parameter
```

```
          #function call is placed within the print statement

main()
```

Solution in C

```
#include<stdio.h>
//function prototype (declaration)
int square(int);

int main() {
    int num, i;
    for (i = 0; i < 10; i++) {
        scanf("%d", &num);
        printf("%d^2=%d\n", num, square(num));
        //num is an actual parameter
        //function call is placed within the printf statement

    }
    return 0;
}
//function definition
int square(int a) {
    //a is formal parameter
    return a * a;
}
```

Program 4:

Write a function `ave_grade` that takes one integer parameter, which represents the number of courses a student takes per semester. The function reads the final grade for each course and returns the average grade per semester. We will assume the input grade is a valid integer between 0 and 100. Write a program to test the function.

Solution in Python

```
#function definition
def ave_grade(num_courses):
        #num_courses is a formal parameter
        #we will assume that num_courses >0
        #we will check this in the main function before
        #calling the function

        sum_num=0
        for i in range(num_courses):
```

```python
                grade=int(input("enter grade "))
                sum_num += grade


        return sum_num/num_courses

def main():
        num_courses=int(input("how many courses you took "))
        if(num_courses>0):
                print("average grade ", ave_grade(num_courses))
        else:
                print("student didn’t take any courses")
main()
```

Solution in C

```c
#include<stdio.h>
//function prototype (declaration)
double ave_grade(int);

int main() {
    int num_courses;
    printf("enter number of courses\n");
    scanf("%d", &num_courses);
    if (num_courses > 0)
        printf("average=%f\n", ave_grade(num_courses));
    else
        printf("student didn’t take any courses\n");
    return 0;
}
//function definition
double ave_grade(int num_courses) {

    //num_courses is a formal parameter
    //we will assume that num_courses > 0
    //we will check this in the main function before
    //calling the function

    int sum = 0, i, grade;
    printf("enter %d grades\n", num_courses);
    for (i = 0; i < num_courses; i++) {
        scanf("%d", &grade);
        sum += grade;
    }

    return (double)(sum) / num_courses;
}
```

Program 5

This program demonstrates the use of Boolean type variables `True` and `False` (ONLY IN PYTHON). In C, the function returns integer, with `1` indicating `True` and `0` indicating `False`.

Write a function `is_even` that takes one integer parameter. Function returns `True` if the parameter is even, and `False` otherwise.

We will write two versions of the same function to demonstrate various locations of the return statement within the function.

Write a program that reads 10 integers and counts the number of evens and odds in the input.

The C programming language does not have Boolean data types and normally uses integers for Boolean testing. The value 0 represents `FALSE`, and any non-zero value represents `TRUE`.

Solution in Python

```
#Function definition:  Version 1
#There are two return statements in this function, BUT ONLY ONE
#WILL BE EXECUTED based on the value of the if expression.
def is_even(num):
     if(num%2==0):
          return True
     else:
          return False

#Function definition:  Version 2
#Instead of using two return statements, we are using a local
#Boolean variable result to hold the return value.

#def is_even(num):
#    if(num%2==0):
#         result=True
#    else:
#         result=False
#
#    return result

#Function definition:  Version 3
#This version eliminates the need of two return statements and
#a local variable. This is one of the most preferable solutions.

#def is_even(num):
#    return num%2==0
```

```python
#Function definition:  Version 4
#Similar to version 3. One of the most preferable solutions

#def is_even(num):
#    return (!(num%2))

def main():
     count_even=0
     count_odd=0
     print("enter 10 integers")
     for i in range(10):
          num=int(input())
          result=is_even(num)
          if(result==True):
               count_even=count_even+1
          else:
               count_odd=count_odd+1

     print("evens=",count_even,"odds=",count_odd)

     #alternatively, the code above can be written as follows:
     count_even=0
     count_odd=0
     print("enter 10 integers")
     for i in range(10):
          num=int(input())
          if(is_even(num)):
               count_even=count_even+1
          else:
               count_odd=count_odd+1

     print("evens=",count_even,"odds=",count_odd)


main()
```

<u>Solution in C</u>

```c
#include<stdio.h>
int is_even(int); //function prototype
int main() {
    int num, i, count_even = 0, count_odd = 0;
    printf("enter 10 integers\n");
    for (i = 0; i < 10; i++) {
        scanf("%d", &num);
        if (is_even(num))
            count_even++;
```

```c
        else
            count_odd++;

     }

    printf("count_even=%d\n", count_even);
    printf("count_odd=%d\n", count_odd);

    return 0;
}
/*Function definition: Version 1
There are two return statements in this function, BUT ONLY ONE
WILL BE EXECUTED based on the value of the if expression.
*/

int is_even(int num) {
    if (num % 2 == 0)
        return 1;
    else
        return 0;
}

/* Function definition:  Version 2
Instead of using two return statements, we are using a local
integer variable result to hold the return value.

int is_even(int num){
        int result;
        if(num%2==0)
                result=1;
        else
                result=0;

        return result;
}
*/

/* Function definition:  Version 3
This version eliminates the need of two return statements and
local variable. This is the most preferable solution.

int is_even(int num){
     return (!(num%2));
}
*/
```

Program 6

Write a function `gcd` that takes two integer parameters, `n` and `m`. The function finds and returns the greatest common divisor of the two parameters, which is the largest number that divides both numbers evenly. Write a program to test the function.

In this program, we will implement the Euclidean Algorithm:

```
gcd(n,m)=gcd(m, n%m), if m≠0
gcd(n,m)=n, if m=0
```

Solution in Python

```
def gcd_1(n,m): #version 1
        while(m>0):
                t=m
                m=n%m
                n=t
        return n
def gcd_2(n,m): #version 2
        while(n>0 and m>0):
                if(n>m):
                        n%=m
                else:
                        m%=n
        return m+n

def main():
        n=int(input("enter first integer "))
        m=int(input("enter second integer "))
        print("gcd =",gcd_1(n,m))
        print("gcd =",gcd_2(n,m))

main()
```

Solution in C

```
#include<stdio.h>
int gcd_1(int, int);
int gcd_2(int, int);
int main(){
    int n, m;
    printf("enter two integers\n");
    scanf("%d%d", &n, &m);
    printf("gcd = %d\n", gcd_1(n, m));
    printf("gcd = %d\n", gcd_2(n, m));
    return 0;
```

```
}
//version 1
int gcd_1(int n, int m){
    int t;
    while (m > 0) {
        t = m;
        m = n % m;
        n = t;
    }
    return n;
}
//version 2
int gcd_2(int n, int m){
    while (n > 0 && m > 0)
        if (n > m)
            n %= m;
        else
            m %= n;
    return m + n;
}
```

Program 7

Write a function `sum_divisors` that has one integer parameter and returns the sum of its divisors. Write a program to test the function.
We will demonstrate both a naive and an efficient solution. In the efficient solution we will use the following fact: if `x` is a divisor of `num` then `num/x` is also a divisor of `num`.
For example, `28 = 1 + 2 + 4 + 7 + 14 + 28` and we have pairs of divisors `(1,28=28/1), (2, 14=28/2), (4,7=28/4)`

Solution in Python

```
def sum_divisors_naive1(num):
        s=0
        for i in range(1,num+1):
                if(num%i==0):
                        s+=i
        return s

def sum_divisors_naive2(num):
        s=1
        if(num==1):
                return num
        i=2
        while(i<=num/2):
                if(num%i==0):
                        s+=i
                i+=1
```

```python
        return s+num

def sum_divisors_efficient(num):
        s=0
        i=1
        while(i*i<num):
                if(num%i==0):
                        s+=i+num//i
                i+=1
        #To account for perfect squares (9, 16, 25, 49, 81, …)
        #we need to include an if statement below to ensure we
        #are not counting the square divisor twice.
        #For example, for num=9, the sum of the divisors is:
        #1+3+9=13.
        #For example, for num=16, the sum of the divisors is:
        #1+2+4+8+16=31

        if(i*i==num):
                s+=i


        return s


def main():
        num=int(input("enter integer "))
        print("sum of the divisors",sum_divisors_naive1(num))
        print("sum of the divisors",sum_divisors_naive2(num))
        print("sum of the divisors",sum_divisors_efficient(num))

main()
```

<u>Solution in C</u>

```c
#include<stdio.h>
int sum_divisors_naive1(int num);
int sum_divisors_naive2(int num);
int sum_divisors_efficient(int num);
int main() {
    int num;
    printf("enter integer\n");
    scanf("%d", &num);
    printf("sum divisors is %d\n", sum_divisors_naive1(num));
    printf("sum divisors is %d\n", sum_divisors_naive2(num));
    printf("sum divisors is %d\n", sum_divisors_efficient(num));
    return 0;
```

```
}
int sum_divisors_naive1(int num) {
    int s = 0, i;
    for (i = 1; i <= num; i++)
        if (num % i == 0)
            s += i;
    return s;
}
int sum_divisors_naive2(int num) {
    int s = 1, i;
    if (num == 1)
        return num;
    for (i = 2; i <= num/2; i++)
        if (num % i == 0)
            s += i;
    return s + num;
}

int sum_divisors_efficient(int num) {
    int s = 0, i;
    for (i = 1; i * i < num; i++)
        if (num % i == 0)
            s += i + num / i;
/* To account for perfect squares (9, 16, 25, 49, 81, …)
we need to include the if statement below to ensure we are not
counting the square divisor twice.
For example, for num=9, the sum of the divisors is: #1+3+9=13.
For example, for num=16, the sum of the divisors is:
1+2+4+8+16=31 */

    if (i * i == num)
        s += i;

    return s;
}
```

Program 8

In number theory, a perfect number is a positive integer that equals the sum of its proper divisors (divisors that are less than a number itself). For example, `6` is a perfect number since `6 = 1+2+3,` and `28` is a perfect number since `28 = 1+2+4+7+14`. However, `12` is not a perfect number since the sum of its the proper divisors, `1+2+3+4+6=16` which is not equal to `12`.

Write a function `isPerfect` that has one integer parameter. The function returns `True (1 in C)` if the parameter is a perfect number and `False (0 in C)` otherwise. We will use function `sum_divisors` from the previous example. Write a program to test the function.

Solution in Python

```python
def sum_divisors_efficient(num):
        s=0
        i=1
        while(i*i<num):
                if(num%i==0):
                        s+=i+num//i
                i+=1
        if(i*i==num):
                s+=i;
        return s

def isPerfect(num):
        if(num==sum_divisors_efficient(num)-num):
                return True
        else:
                return False


def main():
        n=int(input("enter a positive integer "))
        if(n>0):
                if(isPerfect(n)):
                        print(n,"is a perfect number")
                else:
                        print(n,"is not a perfect number")
        else:
                print("Invalid input")
main()
```

Solution in C

```c
#include<stdio.h>
int sum_divisors_efficient(int);
int isPerfect(int num);
int main() {
    int n;
    printf("enter positive integer\n");
    scanf("%d", &n);
    if (n > 0)
        if (isPerfect(n))
            printf("%d is a perfect number\n", n);
        else
            printf("%d is not a perfect number\n", n);
    else
        printf("Invalid input\n");
```

```
    return 0;
}
int sum_divisors_efficient(int num) {
    int s = 0, i;
    for (i = 1; i * i < num; i++)
        if (num % i == 0)
            s += i + num / i;
    if (i * i == num)
        s += i;
    return s;
}
int isPerfect(int num) {
    if (num == sum_divisors_efficient(num) - num)
        return 1;
    else
        return 0;
}
```

3. **VOID/None (no return value) functions**

Program 1:

Write a function `print_square,` that has one integer parameter, `n`. The function prints an `nXn` square of stars.

For example, if `n=3` the function prints

```
***
***
***
```

If `n=5`, the function prints

```
*****
*****
*****
*****
*****
```

Write a program to test `print_square`.

Solution in Python

```
#function definition
def print_square(n):
        for i in range (n):
                for j in range(n):
                        print("*",end='')
                print()
        #this function doesn't have any return value
```

```python
def main():
        n=int(input("enter integer "))
        if(n<=0):
                print("nothing to print")
        else:
                #function call
                print_square(n)
main()
```

Solution in C

```c
#include<stdio.h>
void print_square(int);
/*function prototype
since the function doesn't have a return value, the return type
in this case is void */

int main() {
    int n;
    printf("enter integer\n");
    scanf("%d", &n);
    if (n > 0)
        //function call
        print_square(n);

    else
        printf("nothing to print\n");


    return 0;
}
void print_square(int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("*");
        }
        printf("\n");
    }
}
```

Program 2

Write a function `print_ASCII` that reads a sequence of 10 characters and prints the ASCII value of each character. Write a program to test `print_ASCII`.

Solution in Python

```
def main():

    #function call
    print_ASCII()

#function definition
def print_ASCII():
        print("enter 10 chars")
        for i in range(10):
                ch=input()
                print("ASCII of",ch,"is",(ord)(ch));
main()
```

Solution in C

```
#include<stdio.h>
void print_ASCII();
/* function prototype
 this function doesn't have any parameters and has void as its
return type */

int main() {

    //function call
    print_ASCII();
    return 0;
}
void print_ASCII() {
    int i;
    char ch;
    for (i = 0; i < 10; i++) {
        scanf("%c", &ch);
        printf("ASCII of  %c is %d\n", ch, (int)(ch));
    }
}
```

Program 3

Write a function `print_triangle` that has one integer parameter, `n`, and prints a right triangle of `n` rows as follows:

```
*
**
***
.....
```

```
***********
```

Last line has `n` stars. Write a program to test `print_triangle`.

Solution in Python

```python
#function definition
def print_triangle(n):
        for i in range(1,n+1):
                for j in range(i):
                        print('*',end='');
                print()
def main():
        print("enter integer ")
        num=int(input(""))
        if(num>0):
                #function call
                print_triangle(num)
        else:
                print("nothing to print")
main()
```

Solution in C

```c
#include<stdio.h>
//function prototype
void print_triangle(int);

int main() {
    int n;
    printf("enter a positive int\n");
    scanf("%d", &n);
    if (n > 0)
        //function call
        print_triangle(n);
    else
        printf("nothing to print\n");
    return 0;
}
void print_triangle(int n) {
    int i, j;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= i; j++)
            printf("*");
        printf("\n");
    }

}
```

Program 4

Write a function `print_upsidedown_triangle` that has one integer parameter, `n`, and prints a right upside down triangle of `n` rows as follows:

```
***********
**********
*********
…
*
```

The first line has `n` stars. Write a program to test `print_upsidedown_triangle`.

Solution in Python

```python
#function definition
def print_triangle(n):
        for i in range(n, 0, -1):
                for j in range(i):
                        print('*',end='');
                print()
def main():
        print("enter integer ")
        num=int(input(""))
        if(num>0):
                #function call
                print_triangle(num)
        else:
                print("nothing to print")
main()
```

Solution in C

```c
#include<stdio.h>
//function prototype
void print_upsidedown_triangle(int);
int main() {
    int n;
    printf("enter a positive int\n");
    scanf("%d", &n);
    if (n > 0)
        //function call
        print_upsidedown_triangle(n);
    else
        printf("nothing to print\n");
    return 0;
}
void print_upsidedown_triangle(int n) {
```

```
    int i, j;
    for (i = n; i >= 1; i--) {
        for (j = 1; j <= i; j++)
            printf("*");
        printf("\n");
    }
}
```

Program 5

Write a function `print_num_figure` that doesn't have any parameters. The function reads `9` distinct integers from `1` to `9` and prints them out in the following way: each integer will be printed the number of times equal to its numeric value. Each set of repeated outputs will be printed on a different line. The nine input integers may be input in any order. You may assume that correct input is always given. Write a program to test `print_num_figure`.
See examples below.

Examples :
Input: `1 2 3 4 5 6 7 8 9`
Output:

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Input: `2 3 9 5 4 1 7 8 6`
Output

```
22
333
999999999
55555
4444
1
7777777
88888888
666666
```

Solution in Python

```
#For a Python solution, the output will look slightly different.
#The numbers will be printed immediately after input since at
#this point we haven't covered how to enter a sequence of
#numbers on one line
```

```python
#function definition
def print_num_figure():
        print("enter 9 integers between 1 and 9");
        for i in range(9):
                n=int(input(""))
                for j in range(n):
                        print(n,end='');
                print()
def main():
        #function call
        print_num_figure()
main()
```

Solution in C

```c
#include<stdio.h>
//function prototype
void print_num_figure();
int main() {
    //function call
    print_num_figure();
    return 0;
}
void print_num_figure() {
    int i, j, n;
    printf("enter 9 integers between 1 and 9\n");
    for (i = 1; i <= 9; i++) {
        scanf("%d", &n);
        for (j = 1; j <= n; j++)
            printf("%d", n);
        printf("\n");
    }
}
```

4. **Passing Parameters (Arguments) to the Function - Example**

Example

What is the output of the following program?
Program in Python:

```python
def square(num):
     num=num*num
     return num
def main():
     num = 16
```

```
	print(square(num),"is a square of", num)

main()
```

Program in C:

```
#include<stdio.h>
int square(int);
void main()
{
	int num = 16;
	printf("%d is a square of % d\n",square(num), num);
}
int square(int num)
{
	num = num * num;
	return num;
}
```

Output and Explanation:

256 is a square of 16

Pay attention: the value of the variable `num` in `main` didn't change after the function call. Changes made in the function `square` didn't carry over after the function execution was completed.

5. **Functions with multiple return values (Python only) and alternative solutions in C**

Program 1

Write a function `sum_ave` that has one parameter, `num`, the number of items to read. The function reads `num` integers and finds and returns their sum and average. In Python, the function can return multiple values by separating them by commas in the single `return statement`. In C, the function prints the sum and average of input numbers and returns void. Write a program to test `sum_ave`. We will demonstrate several ways to call the function with multiple returns.

Solution in Python

```
def sum_ave(num_items):
	sum_num=0
	print("enter",num_items,"integers")
	for i in range(num_items):
		n=int(input())
		sum_num+=n
	ave=sum_num/num_items
	return sum_num, ave
	#In Python, the function can return multiple values
```

```
        #by separating them by commas in a single return
        #statement.

        #Don't write
        #return sum_num
        #return ave
        #this is an incorrect implementation
def main():
        num_items=int(input("enter number of items "))
        if(num_items>0):

                #Version 1:
                #res1 will receive value of sum_num
                #res2 will receive value of ave

                res1, res2 = sum_ave(num_items)
                print("sum is", res1)
                print("average is", res2)



                #Version 2:
                res = sum_ave(num_items)
                print("function results are", res)

                #The output in this case will be enclosed in
                #parentheses, and the values will be separated
                #by commas. Later, we will learn that res is a
                #a tuple and we can access
                #each value of the tuple using index; in this
                #case, res[0] holds the value of sum_num
                #and res[1] holds the value of ave.
        else:
                print("invalid input")
main()
```

Solution in C

```
#include<stdio.h>
//function prototype
void sum_ave(int);
int main() {
    int n;
    printf("enter number of items\n");
    scanf("%d", &n);
    if (n > 0)
        sum_ave(n);
```

```
    else
        printf("invalid input\n");
    return 0;
}
void sum_ave(int n) {
    int sum = 0, i, num;
    double ave;
    printf("enter %d integers\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &num);
        sum += num;
    }
    ave = (double)(sum) / n;
    printf("sum=%d,ave=%f\n", sum, ave);
}
```

Program 2

Write a function `count` that reads `10` integers and finds and returns the number of positive numbers, negative numbers, and zeros in the input. In C, the function prints the number of positives, negatives, and zeros. Write a program to test `count`.

Solution in Python

```
def count():
        c_pos=0
        c_neg=0
        c_zero=0
        print("enter 10 integers")
        for i in range(10):
                n=int(input())
                if(n>0):
                        c_pos+=1
                elif(n<0):
                        c_neg+=1
                else:
                        c_zero+=1
        return c_pos, c_neg, c_zero
def main():
        #various ways to write function call
        #Version 1:
        res1, res2, res3=count()
        print("pos =",res1,"neg =",res2,"zeros =",res3)
        #Version 2
        print("counters are",count())
        #Version 3
        res=count()
        print("pos =",res[0],"neg =",res[1],"zeros =",res[2])
main()
```

Solution in C

```
#include<stdio.h>
//function prototype
void count();
int main() {
    //function call
    count();
    return 0;
}
void count(){
    int n, c_pos = 0, c_neg = 0, c_zero = 0, i;
    printf("enter 10 integers\n");
    for (i = 0; i < 10; i++) {

        scanf("%d", &n);
        if (n > 0)
            c_pos++;
        else if (n < 0)
            c_neg++;
        else
            c_zero++;
    }
    printf("pos=%d,neg=%d,zeros=%d\n",c_pos,c_neg,c_zero);
}
```

6. **Functions with multiple types of return values (Python only) and alternative solutions in C**

Program 1

Write a function `ave_odd_neg` that doesn't have any parameters. The function reads `10` integers and finds and returns the average of negative odd numbers. In there are no negative odd numbers in the input, the function returns the string: `"No odd negative numbers, no average"`. In C, the function prints the average or an error message. Write a program to test `ave_odd_neg`.

Solution in Python

```
def ave_odd_neg():
        sum=0
        count=0
        print("enter 10 integers")
        for i in range(10):
                n=int(input())
                if(n%2 and n<0):
                        sum+=n
```

```
                        count+=1
        if(count>0):
                print("average of negative odds is",end=' ')
                return sum/count

        else:
                return "No odd negative numbers, no average"
def main():
        #function call
        print(ave_odd_neg())
main()
```

Solution in C

```
#include<stdio.h>
//function prototype
void ave_odd_neg();
int main() {
    //function call
    ave_odd_neg();
    return 0;
}
void ave_odd_neg() {
    int n, sum = 0, count = 0, i;
    printf("enter 10 integers\n");
    for (i = 0; i < 10; i++) {
        scanf("%d", &n);
        if (n < 0 && n % 2) {
            sum += n;
            count++;
        }
    }
    if (count == 0)
        printf("no odd negatives, no average\n");
    else
        printf("ave=%f\n", (float)(sum) / count);
}
```

Program 2

This program is similar to program 1 but instead of returning string in case of no odd negatives the function returns None in this case.
Write a function `ave_odd_neg` that doesn't have any parameters. The function reads `10` integers and finds and returns the average of negative odd numbers. In case, and there are no negative odd numbers in the generated sequence, the function returns `None`. In C, function prints the average or an error message. Write a program to test `ave_odd_neg`.

Solution in Python

```
def ave_odd_neg():
        sum=0
        count=0
        print("enter 10 integers")
        for i in range(10):
                n=int(input())
                if(n%2!=0 and n<0):
                        sum+=n
                        count+=1
        #Version 1:
        if(count>0):
                return sum/count

        else:
                return None

        #Version 2:
        #if(count>0):
        #       return sum/count
        #else:
        #       return


        #Version 3:
        #if(count>0):
        #       return sum/count


def main():
        #function call
        res=ave_odd_neg()
        if(res==None):
                print("no negative odd integers, no average")
        else:
                print("average=",res)
main()
```

The solution in C is same as in Program 1

Program 3

This program is similar to program 1 but has multiple returns of different types.

Write a function `ave_odd_neg` that doesn't have any parameters. The function reads `10` integers and finds and returns the number of negative odd integers, their sum and average. We will demonstrate different ways to resolve the special case of no odd negative integers. In C, function prints the count, sum, and average, or an error message. Write a program to test `ave_odd_neg`.

Solution in Python

```
#we will write several versions of the same function
#to demonstrate various ways to handle the special case
#of no odd negatives
def ave_odd_neg1():
        sum=0
        count=0
        print("enter 10 integers")
        for i in range(10):
                n=int(input())
                if(n%2!=0 and n<0):
                        sum+=n
                        count+=1
        if(count>0):
                return count, sum, sum/count

        else:
                return count, sum, None
                #alternatively, we can return None,None,None
def ave_odd_neg2():
        sum=0
        count=0
        print("enter 10 integers")
        for i in range(10):
                n=int(input())
                if(n%2!=0 and n<0):
                        sum+=n
                        count+=1
        if(count>0):
                return count, sum, sum/count
def ave_odd_neg3():
        sum=0
        count=0
        print("enter 10 integers")
        for i in range(10):
                n=int(input())
                if(n%2!=0 and n<0):
                        sum+=n
                        count+=1
        if(count>0):
```

```python
                return count, sum, sum/count
        else:
                return None
        #alternatively we can write an empty return statement

def main():
        #Version 1:
        count,sum,ave=ave_odd_neg1()
        if(count==0):
                print("no negative odd integers, no average")
                #if the function returns None,None,None
                #the if statement will be if(count==None)

        else:
                print("results are",count,sum,ave)

        #Version 2:
        result=ave_odd_neg2()
        if(result==None):
                print("no negative odd integers, no average")
        else:
                print("results are",result)
                #in this case three return values
                #will be printed in parenthesis

        #Version 3:
        result=ave_odd_neg3()
        if(result==None):
                print("no negative odd integers, no average")
        else:
                print("results are",result)

        #In versions 2 and 3 we must use tuples approach.
        #The statement res1, res2, res3=ave_odd_neg2()
        #will cause an error if there are no negative odd
        #numbers
main()
```

Solution in C

```c
#include<stdio.h>
//function prototype
void ave_odd_neg();
int main() {
    //function call
    ave_odd_neg();
```

```
    return 0;
}
void ave_odd_neg() {
    int n, sum = 0, count = 0, i;
    printf("enter 10 integers\n");
    for (i = 0; i < 10; i++) {
        scanf("%d", &n);
        if (n < 0 && n % 2) {
            sum += n;
            count++;
        }
    }
    if (count == 0)
        printf("no odd negatives, no average\n");
    else
        printf("count=%d\n", count);
        printf("sum=%d\n", sum);
        printf("ave=%f\n", (float)(sum) / count);


}
```

III. Examples of Python Standard Library Modules and Sample Solutions in Python

Python standard library documentation

https://docs.python.org/3/library/index.html

1. `math` module in Python

https://docs.python.org/3/library/math.html

The `math` module provides access to the mathematical functions defined by the C standard. To use the `math` module in the current program, the `import math` statement is required at the beginning of the program.

In general, the import statement has the following syntax

```
import module_name
```

When interpreter encounters an `import` statement, it imports the module to your current program. You can use the functions from a module by using a dot(.) operator along with the module name.

```
module_name.module_member
```

For example, the `math` module has the function (member) `sqrt`.

`math.sqrt(x)` returns the square root of `x`.

Program 1:

Write a program that inputs 10 integers and prints the square root of each non-negative input and an error message otherwise.

Solution:

```
import math
def main():
     for i in range(10):
          num=int(input("enter an integer "))
          if(num>=0):
             print("the sqrt of",num,"is",math.sqrt(num))
          else:
             print("cannot find sqrt of the negative number")
main()
```

Program 2:

Write a program that inputs `10` integers and prints the factorial of each non-negative input and an error message otherwise.
The factorial of a non-negative integer n, is the product of all positive integers less than or equal to `n`. We denote n factorial as `n!` For example, `5! = 1*2*3*5=120`

Solution:

```
import math
def main():
     for i in range(10):
          num=int(input("enter an integer "))
          if(num>=0):
               print(math.factorial(num))
          else:
               print("error")

main()
```

Program 3:

Write a program that reads a pair of integers, `m` and `n`, and finds and prints m^n using the `math.pow(m,n)` function.

Solution:

```
import math
def main():
     n=int(input("enter an integer "))
     m=int(input("enter an integer "))
     if(n==0 and m<0):
```

```
            print("error, cannot divide by zero")
        else:
            print(n,"^",m,"=",math.pow(n,m))
main()
```

Program 4:

Write a function `exponent` that has one integer parameter. The function calculates and returns the estimated value of `e` using the following formula.

`e = 1/0! + 1/1! + 1/2! + 1/3! +....`

Write the program that first asks user to enter the number of terms (positive integer) in the above sum and then calls the function `exponent` to calculate the estimated value of `e`. In addition, the program prints the value of `e` stored in `math.e` constant and the value of `e` calculated using built-in function `math.exp(x)`, which returns e^x .

For example, if the input is `1`, the output will be `1`, if the input is `2`, the output will be `2`, if the input is `3`, the output will be `2.5`, if the input is `4`, the output will be approximately `2.666666`.

Solution:

```
import math
def exponent(n):
        s=0
        for i in range(n):
                s+=1/math.factorial(i)
        return s
def main():
        n=int(input("enter number of terms "))
        if(n>0):
                print("e using function exponent", exponent(n))
                print("value of e stored in math.e", math.e)
                print("using function exp(x)", math.exp(1))
        else:
                print("invalid input")
main()
```

2. **`random` module in Python**

https://docs.python.org/3/library/random.html

The `random` module provides access to pseudo-random number generators for various distributions.

To use the `random` module in the current program, an `import random` statement is required at the beginning of the program.

Program 1:

Write a function `isDiv` that has two integer parameters, `n` and `m`. The function returns `True` if `n` is divisible by `m` without a remainder, and `False` otherwise. Write a function `div_3` that has one integer parameter, `num`. The function generates `num` random integers, each from `1` to `20`, counts the number of pairs divisible by `3`, calculates their sum and average and prints the results. The function also prints randomly generated numbers for testing purposes. Write `main` to test the function `div_3`.

Solution:

```
import random
def isDiv(n,m):
        if(n%m==0):
                return True
        else:
                return False
def div_3(num):
        s=0
        count=0
        print("random data is")
        for i in range(num):
                n=random.randint(1, 20)
                print(n)
                if(isDiv(n,3)):
                        s+=n
                        count+=1
        if(count>0):
                print("there are",count,"divisible by 3")
                print("their sum",s,"average",s/count)
        else:
                print("no divisible by 3")
def main():
        num = random.randint(5, 10)
        div_3(num)

main()
```

Program 2:

Write a function `isPrime` that has one parameter. The function returns `True` if the parameter is a prime number, and `False` otherwise. Write a function `process` that generates `10` positive integers, each in the range from `1` to `100000`. For each randomly generated number the function prints `PRIME` if the number is prime and `NOT PRIME` otherwise. Write `main` to test the function `process`.

Solution:

```
import random
def isPrime(num):
```

```
        count=0
        if(num==1):
                return False
        i=2
        while(i*i <= num):
                if(num%i==0):
                        return False
                i+=1

        return True

def process():
        for i in range(10):
                 num=random.randint(1, 100000)
                 if(isPrime(num)):
                        print(num,"IS PRIME")
                 else:
                        print(num,"IS NOT PRIME")
def main():
         process()
main()
```

Program 3:

Write a function `average` that generates 10 floating-point numbers, each in the range from 1 to 10, and finds and returns their average. The function also prints randomly generated numbers for testing purposes. Write `main` to test the function `average`.

Solution:

```
import random
def average():
     sum=0.0
     print("random generated numbers are")
     for i in range(10):
          num=random.uniform(1,10)
          print(num)
          sum+=num
     return sum/10
def main():
     print("The average is",average())

main()
```

Program 4:

Write a function `process` that takes two parameters, the number of items purchased in a store and the tax rate. For each item, the function generates the price, the floating point number between 0 and 100, and the tax indicator, the integer 0 or 1, where 1 indicates taxable, and 0 indicates non-taxable.

The store is offering a discount event: items priced at $50 or more receive a 10% discount.

The function calculates and returns:

- The total amount paid after applying tax and discount.
- The average price per item.

The function also prints randomly generated numbers for testing purposes. Write `main` to generate the number of items purchased (an integer between 5 and 10) and the tax rate (an integer between 5 and 9), and then test the function `process`.

Solution:

```
import random
def main():
      num_items=random.randint(5,10)
      tax=random.randint(5,9)
      sum=0
      for i in range(num_items):
            price=random.uniform(0,100)
            indicator=random.randint(0,1)
            print("price for item",i+1,"is",format(price, "0.2f"))
            if(price>=50):
                  price*=(90/100)
                  print("discount price:",format(price,"0.2f"))
            else:
                  print("item doesn't get discount")
            if(indicator==1):
                  print("item is taxable")
                  price*=(1+tax/100)
            else:
                  print("item is not taxable")
            sum+=price
      return sum, sum/num_items

def main():
      num_items=random.randint(5,10)
      tax=random.randint(5,9)
      print("num_items",num_items,"tax",tax)
      total, average = process(num_items, tax)

      print("total price is",format(total,"0.2f"))
```

```
	print("average per item is",format(average,"0.2f"))
main()
```

Program 5:

Write a function `find_gcd` that randomly generates 10 pairs of integers, each between `1` and `100`. For each pair, the function finds and prints the greatest common divisor (`GCD`). In addition, the function counts and returns the number of relatively prime pairs. Two numbers are called relatively prime if their `GCD` is `1`. Write `main` to test the function `find_gcd`. In this program, we will use `math` and `random` modules.

```
import random
import math
def find_gcd():
	count=0
	for i in range(10):
		n1=random.randint(1,100)
		n2=random.randint(1,100)
		print("random numbers are",n1,n2)
		GCD=math.gcd(n1,n2)
		print("their gcd is",GCD)
		if(GCD==1):
			print(n1,"and",n2,"are relatively prime")
			count+=1
	return count

def main():
	count=find_gcd()
	if(count==0):
		print("no relatively prime pairs")
	else:
		print(count,"relatively prime pairs")
main()
```

Program 6:

Write a function `find_power` that randomly generates 10 pairs of integers, each between -10 and 10. For each pair the function uses function `pow(n,m)` from `math` module to find and print n^m. Write `main` to test the function `find_power`. In this program, we will use `math` and `random` modules.

Solution:

```
import math
import random
def find_power():
```

```
    print("random data is:")
    for i in range(10):
         n=random.randint(-10,10)
         m=random.randint(-10,10)
         print("n=",n,"m=",m)
         if(n==0 and m<0):
              print("error, cannot divide by zero")
         else:
              print(n,"^",m,"=",math.pow(n,m))
def main():
    find_power()
main()
```

Program 7:

Write a function `printChar` that has three parameters, two positive integers, `n` and `m`, and a printable character (a character with `ASCII` value between `33` and `126`). The function prints an `mXn` rectangle of the character. Write `main` to test the function `printChar`.

```
import random
def printChar(m,n,ch):
        for i in range(m):
                for j in range(n):
                        print(ch,end="")
                print()
def main():
        m=random.randint(5,10)
        n=random.randint(5,10)
        char=(chr)(random.randint(33,126))
        print("random data", m,n,char)
        print("rectangle is")
        printChar(m,n,char)
main()
```

IV. Examples of C built-in libraries and Sample Solutions in C

1. `math.h` library in C

The `math.h` library provides access to various mathematical functions. To use the math library in the current program, the `#include<math.h>` statement is required at the beginning of the program.

Program 1:

Write a function `sumSqrt` that doesn't have any parameters. The function reads a sequence of non-negative integers, prints the square root of each non-negative input, and finds and returns the sum of square roots. Write `main` to test the function.

```c
#include<stdio.h>
#include<math.h>
double sumSqrt();
int main() {
    double result;
    result = sumSqrt();
    printf("sum of square roots is %lf\n", result);
    return 0;
}
double sumSqrt() {
    int num;
    double sum = 0, square_root;
    printf("enter non negative integer\n");
    scanf("%d", &num);
    while (num >= 0) {
        square_root = sqrt(num);
        printf("sqrt(%d) = %lf\n", num, square_root);
        sum += square_root;
        scanf("%d", &num);
        /* it is advisable to use an auxiliary variable,
        square_root, to hold the result of the calculation
        sqrt(num) to avoid repeating the function call */

    }
    return sum;
}
```

Program 2:

Write a function `sum_abs` that reads a sequence of non-zero integers; the first zero value terminates the input. For each input, the function prints its absolute value. In addition, the function finds and returns the sum of all absolute values. Write `main` to test the function.

```c
#include<stdio.h>
#include<math.h>
int sum_abs();
int main() {
    printf("sum of absolute values is %d\n", sum_abs());
    return 0;
}
int sum_abs() {
    int num;
    int sum = 0, result;
    printf("enter non zero integer\n");
```

```
    scanf("%d", &num);
    while (num!=0) {
        result = abs(num);
        printf("abs(%d) = %d\n", num, result);
        sum += result;
        scanf("%d", &num);
    }
    return sum;
}
```

2. Pseudo-random numbers in C

We can generate pseudo-random numbers in C using the library function `int rand(void)`, declared in `stdlib.h`, which returns a pseudo-random integer between `0` and `RAND_MAX` (inclusive). `RAND_MAX` is a constant declared in `stdlib.h` and its value is compiler-dependent but at least 32767.

To generate an integer number in a specific range, say between `MIN` and `MAX`, we can use the following formula: `MIN+rand()%(MAX-MIN+1).`

To generate a floating point number between 0 and 1, we can use the following formula:
`(float)(rand())/RAND_MAX`

To generate a floating point number between 0 and `K,` we can use the following formula:
`K*(float)(rand())/RAND_MAX`

To ensure a program generates different pseudo-random numbers each time it runs, the program must use the additional library function `void srand(unsigned int seed)`, declared in `stdlib.h`, to seed the random number generator used by `rand()`. A random seed determines the start of the sequence of random numbers. We will use the UNIX timestamp, which changes every time the program runs, as the seed.

UNIX time is defined as the number of non-leap seconds that have passed since 00:00:00 UTC on Thursday, January 1, 1970. To use it in our program, we will include the following statement at the beginning:

```
srand(time(NULL));
```

The `time()` function is declared in `time.h`.

Program 1:

Write a function `isDiv` that has two integer parameters, `n` and `m`. The function returns `1` if `n` is divisible by `m` without a remainder, and `0` otherwise. Write a function `div_3` that has one integer parameter, `num`. The function generates `num` random integers, each from `1` to `20`, counts the number of pairs divisible by `3`, calculates their sum and average and prints the results. The function also prints randomly generated numbers for testing purposes. Write `main` to test the function `div_3`.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int isDiv(int, int);
void div_3(int)
int main() {
    int num;
    srand(time(NULL));
    num = 5 + rand() % 10;
    div_3(num);
    return 0;
}
int isDiv(int n, int m) {
    if (n % m == 0)
        return 1;
    else
        return 0;
}
void div_3(int num) {
    int sum = 0, count = 0, i, n;
    printf("random data is\n");
    for (i = 0; i < num; i++) {
        n = 1 + rand() % 20;
        printf("%d\n", n);
        if (isDiv(n, 3)) {
            sum += n;
            count += 1;
        }
    }
    if (count > 0) {
        printf("count=%d\n", count);
        printf("sum=%d\n", sum);
        printf("average=%f\n", (float)sum / count);
    }
    else
        printf("no divisible by 3\n");
}
```

Program 2:

Write a function `isPrime` that has one integer parameter. The function returns `1` if the parameter is a prime number, and `0` otherwise. Write a function `process` that generates `10` positive integers, each in the range from `1` to `100000`. For each randomly generated number the function prints `PRIME` if the number is prime and `NOT PRIME` otherwise. Write `main` to test the function `process`.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int isPrime(int);
void process();
int main() {
    srand(time(NULL));
    process();
    return 0;
}
int isPrime(int n) {
    int i, count = 0;
    if (n == 1)
        return 0;
    for (i = 1; i * i <= n; i++) {
        if (n % i == 0)
            return 0;
    }

    return 1;
}
void process() {
    int i, n;
    for (i = 0; i < 10; i++) {
        n = 1 + rand() % 100000;
        if (isPrime(n))
            printf("%d IS PRIME\n", n);
        else
            printf("%d IS NOT PRIME\n", n);
    }

}
```

Program 3:

Write a function `average` that generates 10 floating-point numbers, each in the range from 0 to 10, and finds and returns their average. The function also prints randomly generated numbers for testing purposes. Write `main` to test the function `average`.

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
float average()

int main() {

	printf("average=%f\n", average());
	return 0;

}
float average() {
	int count = 0, i;
	float n, sum = 0.0;
	srand(time(NULL));
	printf("random data is\n");
	for (i = 0; i < 10; i++) {
		n = 10*(float)rand()/RAND_MAX;
		printf("%f\n", n);
		sum += n;

	}
	return (float)(sum) / 10;

}
```

Program 4:

Write a function `find_power` that randomly generates 10 pairs of integers, each between -10 and 10. For each pair the function uses function `pow(n,m)` from `math` module to find and print n^m. Write `main` to test the function `find_power`.

```c
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
void find_power();

int main() {

	find_power();
	return 0;
}
void find_power() {
    int n, m, i;
    printf("random data is\n");
    for (i = 0; i < 10; i++) {
```

```
        n = -5 + rand() % (5 - (-5) + 1);
        m = -5 + rand() % (5 - (-5) + 1);
        printf("n=%d, m=%d\n", n, m);
        if (n == 0 && m < 0)
            printf("error, cannot divide by zero");
        else
            printf("n^m=%f\n", pow(n, m));
    }

}
```

Program 5:

Write a function `printChar` that has three parameters, two positive integers, `n` and `m`, and a printable character (a character with `ASCII` value between `33` and `126`). The function prints an `mXn` rectangle of the character. Write `main` to test the function `printChar`.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
void printChar(int, int, char);
int main() {
    int n, m;
    char ch;
    srand(time(NULL));
    m = 5 + rand() % 6;
    n = 5 + rand() % 6;
    ch = (char)(33 + rand() % (126 - 33 + 1));
    printf("random data is m=%d, n=%d, ch=%c\n", m, n, ch);
    printf("rectangle is\n");
    printChar(m, n, ch);
    return 0;
}
void printChar(int m, int n, char ch) {
    int i, j;
    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            printf("%c", ch);
        }
        printf("\n");
    }
}
```

V. Additional examples with solutions

Program 1

Write a function `sum_squares` that has one integer parameter, `n`, and returns the sum of squares: $1^2+2^2+3^2+...+n^2$. Write `main` to test the function `sum_squares`.

Solution in Python

```
def sum_squares(n):
      sum_result=0
      for i in range(1,n+1):
            sum_result+=i**2
      return sum_result
def main():
      num=int(input("enter integer "))
      print("sum of squares is", sum_squares(num))
main()
```

Solution in C

```
#include<stdio.h>
int sum_squares(int);
void main()
{
      int num;
      printf("enter integer\n");
      scanf("%d", &num);
      printf("sum of squares is %d\n",sum_squares(num));
}
int sum_squares(int n)
{
      int sum_result = 0, i;
      for (i = 1; i <= n; i++) {
            sum_result += i * i;
      }
      return sum_result;
}
```

Program 2

Write a function `sum_powers` that has two parameters: a floating-point number `a` and an integer `n`. The function returns the sum of powers: $a^1+a^2+a^3+...+a^n$. Write `main` to test the function `sum_powers`. We will write several versions of the function to demonstrate efficient and inefficient solutions.

Solution in Python

```
import math
def sum_powers_slow(a,n): #inefficient solution
        #number of basic operations is
        #1+2+...+n=(n+1)*n/2 = O(n^2)
        sum_result=0
        for i in range(1,n+1):
```

```python
                sum_result+=math.pow(a,i)
        return sum_result
def sum_powers_efficient(a,n):
        #Efficient solution: using the fact that a^n=a*a^(n-1)
        #number of basic operations: 2+2+2..+2=2*n=O(n)
        sum_result=0
        product=1
        for i in range(1,n+1):
                product*=a
                sum_result+=product
        return sum_result

def main():
        a=float(input("enter floating point number "))
        num=int(input("enter integer "))
        print("sum of powers is", sum_powers_slow(a,num))
        print("sum of powers is", sum_powers_efficient(a,num))
main()
```

Solution in C

```c
#include<stdio.h>
#include<math.h>
double sum_powers_slow(double, int);
double sum_powers_efficient(double, int);
int main()
{
    int num;
    double a;
    printf("enter floating point number and an integer\n");
    scanf("%lf%d", &a, &num);
    printf("sum of powers is %f\n",sum_powers_slow(a, num));
    printf("sum of powers is %f\n",sum_powers_efficient(a,num));
    return 0;
}
double sum_powers_slow(double a, int n)
{
    /*inefficient solution
    number of basic operations is
    1+2+...+n=(n+1)*n/2 = O(n^2)
    */
    double sum_result = 0;
    int  i;
    for (i = 1; i <= n; i++) {
        sum_result += pow(a, i);
    }
```

```
    return sum_result;
}
double sum_powers_efficient(double a, int n)
{
    /*Efficient solution
    using the fact that a^n=a*a^(n-1)
    number of basic operations: 2+2+2..+2=2*n=O(n)
    */
    double sum_result = 0, product = 1;
    int  i;
    for (i = 1; i <= n; i++) {
        product *= a;
        sum_result += product;
    }
    return sum_result;
}
```

Program 3

Write a function that calculates the approximate value of sin(x) using the following approximation formula:

$$\sin(x) \cong \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

The function has two parameters: a floating-point number `x` and an integer `n`. The function returns the approximate value of sin(x) using the first `n` terms in the approximation formula. Write `main` to test the function and compare the result with the built-in `math.h` library function. We will write several solutions to demonstrate efficient and inefficient ways to solve this problem.

Solution in Python

```
import math
def sin_slow(x,n):
        sin=0
        for i in range(n+1):
                sign=math.pow(-1,i)
                sin+=sign*pow(x,2*i+1)/math.factorial(2*i+1)
        return sin
def sin_efficient(x,n):
        #efficient solution:
        #using the following connection between
        #term(i) and term(i+1)
        #term(i+1)=-term(i)*x*x/((2*i)(2*i+1))
        sin=x
```

```python
        last=x
        for i in range(1,n+1):
                last*=-x*x/((2*i)*(2*i+1))
                sin+=last
        return sin

def main():
        num=int(input("enter number of terms "))
        x=float(input("enter floating point number "))
        print("sin_slow=",sin_slow(x, num));
        print("sin_efficieint=",sin_efficient(x, num));
        print("math library result", math.sin(x));

main()
```

Solution in C

```c
#include<stdio.h>
#include<math.h>
long factorial(int);
double sin_slow(double, int);
double sin_efficient(double, int);
int main()
{
    int num;
    double x;
    printf("enter floating point number and an integer\n");
    scanf("%lf%d", &x, &num);
    printf("factorial test %ld\n", factorial(num));
    printf("sin(%lf)=%f\n", x, sin_slow(x, num));
    printf("sin(%lf)=%f\n", x, sin_efficient(x, num));
    printf("math.h library result %f\n", sin(x));
    return 0;
}
long factorial(int n)
{
    long f = 1;
    for (; n > 1; --n) {
        f *= n;
    }
    return f;
}
double sin_slow(double x, int n)
{
    double sin = 0;
    int  i;
    for (i = 0; i <= n; i++) {
```

```
        sin += pow(-1,i) * pow(x, 2*i + 1) / factorial(2*i + 1);
    }
    return sin;
}
double sin_efficient(double x, int n)
{
    /* efficient solution
    using the following connection between term(i) and term(i+1)
    term(i+1)=-term(i)*x*x/((2*i)(2*i+1))
    */
    double sin = x, last = x;
    int  i;
    for (i = 1; i <= n; i++) {
        last *= -x * x / ((2 * i) * (2 * i + 1));
        sin += last;
    }
    return sin;
}
```

Program 4

Write the following functions:

`gcd_3` that has three integer parameters and returns their greatest common divisor. Note, `gcd(a, b, c) = gcd(a, gcd(b, c)) = gcd(gcd(a, b), c) = gcd(gcd(a, c), b);`

`sumCommonDiv` that has two integer parameters and returns the sum of their common divisors. For example, if parameters are `30` and `66`, the function returns `12(1+2+3+6=12)`, if the parameters are `28` and `14`, function returns `24(1+2+7+14=24)`, and if the parameters are `10` and `21`, the function returns `1` since `10` and `21` are relatively prime.

`printSimplified` that has two integer parameters, numerator and denominator of the fraction. The function checks validity of the parameters and prints the simplified fraction after the numerator and denominator are divided by their greatest common divisor.

`menu` that has one integer parameter choice, if the choice is `1`, the function tests the `gcd_3` function, if the choice is `2`, the function tests the `sumCommonDiv` function, if the choice is `3`, the function tests the `printSimplified` function. The menu function prints an error message if the choice is not `1,2,` or `3`. Write `main` to test the `menu` function. Note, we will use function `gcd` we wrote in program 6 section 2.

Solution in Python

```
def gcd(n, m):
        t = 0
        while m > 0:
```

```
                t = m
                m = n % m
                n = t
        return n

def gcd_3(a, b, c):
        return gcd(gcd(a, b), c)

def sumCommonDiv(num1, num2):
        i = 0
        min_val = 0
        max_val = 0
        sum_val = 1

#Determine the minimal and maximal values between num1 and num2

        if num1 < num2:
                min_val = num1
                max_val = num2
        else:
                min_val = num2
                max_val = num1

#if the minimal value is 1, the only common divisor is 1

        if min_val == 1:
                return 1

#if the maximal value is divisible by the minimal one,
#add the minimal value to the sum

        if max_val % min_val == 0:
                sum_val += min_val

 #iterate through possible divisors up to the square root
 #of the minimal value

        i = 2
        while i * i < min_val:

#Check if both num1 and num2 are divisible by the current
#divisor
                if min_val % i == 0:
                        if max_val % i == 0:
                                sum_val += i
                if max_val % (min_val // i) == 0:
                        sum_val += (min_val // i)
```

```
                i += 1

#checking the special case of perfect square

        if i * i == min_val and max_val % i == 0:
                sum_val += i
        return sum_val

def printSimplified(n, m):
        gcd_res = 0
        if n > 0 and m > 0:
                gcd_res = gcd(n, m)
                print("simplified fraction")
                print(n // gcd_res,"/",m //gcd_res)
        else:
                print("invalid parameters")

def menu(choice):
        if choice == 1:
                print("Enter three numbers:")
                a = int(input())
                b = int(input())
                c = int(input())
                print("gcd of three inputs",gcd_3(a, b, c))
        elif choice == 2:
                print("Enter two numbers:")
                num1 = int(input())
                num2 = int(input())
                print("sumCommonDiv =",sumCommonDiv(num1, num2))
        elif choice == 3:
                print("Enter two numbers:")
                num1 = int(input())
                num2 = int(input())
                if(num1>0 and num2>0):
                        print("Simplified fraction is")
                        printSimplified(num1,num2)
                else:
                        print("Invalid choice")
        else:
                print("Invalid choice")
def main():
        choice = int(input("enter choice "))
        menu(choice)
main()
```

Solution in C

```c
#include<stdio.h>
int gcd(int, int);
int gcd_3(int, int, int);
int sumCommonDiv(int, int);
void printSimplified(int, int);
void menu(int);
int main() {
    int choice;
    printf("enter choice\n");
    scanf("%d", &choice);
    menu(choice);
    return 0;
}
int gcd(int n, int m) {
    int t;
    while (m > 0) {
        t = m;
        m = n % m;
        n = t;
    }
    return n;
}
int gcd_3(int a, int b, int c) {
    return (gcd(gcd(a, b), c));
}
int sumCommonDiv(int num1, int num2) {
    int i, min, max, sum = 1;

//Determine the minimal and maximal values between num1 and num2

    if (num1 < num2)
        min = num1;
    else
        min = num2;

    //statement above could be written
    //min = num1 < num2 ? num1 : num2;

    max=num1+num2-min;

    //statement above could be written
    //max = num1 > num2 ? num1 : num2;

//if the minimal value is 1, the only common divisor is 1

    if (min == 1)
        return 1;
```

```c
//if the maximal value is divisible by the minimal one,
//add the minimal value to the sum

    if (max % min == 0)
        sum += min;

//iterate through possible divisors up to the square root
//of the minimal value

    for (i = 2; i * i < min; i++) {

    //Check if both num1 and num2 are divisible by the
    //current divisor

        if (min % i == 0) {
            if (max % i == 0)
                sum += i;
            if (max % (min / i) == 0)
                sum += (min / i);
        }
    }
    //Checking the special case of perfect square

    if (i * i == min && max % i == 0)
        sum += i;
    return sum;
}
void printSimplified(int n, int m) {
    int gcd_res;
    if (n > 0 && m > 0) {
        gcd_res = gcd(n, m);
        printf("%d/%d=%d/%d\n",n,m,n/gcd_res, m/gcd_res);
    }
    else
        printf("invalid parameters\n");
}
void menu(int choice) {
    int a, b, c;
    switch (choice) {
    case 1:
        printf("enter three positive ints\n");
        scanf("%d%d%d", &a, &b, &c);
        printf("gcd(%d,%d,%d)=%d\n", a, b, c, gcd_3(a, b, c));
        break;
    case 2:
        printf("enter two positive int\n");
```

```
        scanf("%d%d", &a, &b);
        printf("sumCommonDiv is %d\n", sumCommonDiv(a, b));
        break;
    case 3:
        printf("enter two integers\n");
        scanf("%d%d", &a, &b);
        printf("simplified fraction\n");
        printSimplified(a, b);
        break;
    default:
        printf("invalid input\n");
    }
}
```

Program 5

Write the following functions:

- `minDiv`: This function has one integer parameter, `num > 1`, and returns its smallest divisor (excluding 1). If `num` is prime, the function returns `num`.
- `maxDiv`: This function has one integer parameter, `num > 1`, and returns its largest divisor (excluding `num`). If `num` is prime, the function returns `1`.
- `lcm`: This function has two integer parameters and returns their Least Common Multiple (LCM). The least common multiple of two positive integers is the smallest number that is a multiple of both. For example, LCM(4,6)=12. Multiples of 4 are 4, 8, 12, 16,...; multiples of 6 are 6, 12, 18, 24,...; and we can see that 12 is the least common multiple.
- `charToNumber`: This function has three character parameters (digits) and returns the three-digit number. For example, if the parameters are `'5', '3', '1'`, the function returns the integer `531`.
- `menu`: This function has one integer parameter, `choice`. If the `choice` is `1`, the function tests `minDiv`. If the `choice` is `2`, the function tests `maxDiv`. If the `choice` is `3`, the function tests `lcm`. If the `choice` is `4`, the function tests `charToNumber`. The `menu` function prints an error message if the `choice` is not `1, 2, 3,` or `4`. Write `main` to test the `menu` function.

Solution in Python

```
def gcd(n, m):
        t = 0
        while m > 0:
                t = m
                m = n % m
                n = t
        return n
def lcm(num1, num2):
```

```
    if (num1 == 0 or num2 == 0):
        return 0
    num1 = math.fabs(num1)
    num2 = math.fabs(num2)
    return num1 * num2 / gcd(num1, num2)

def minDiv(num):

    if (num % 2 == 0):
        return 2;
    i = 3
    while(i * i <= num):
        if (num % i == 0):
            return i
        i+=2
    return num

def maxDiv(num):

    if (num % 2 == 0):
        return num // 2

    i = 3
    while(i * i <= num):
        if (num % i == 0):
            return num//i
        i+=2
    return 1


def charToNumber(c1, c2, c3):
    num1=(ord(c1) - ord('0')) * 100
    num2=(ord(c2) - ord('0')) * 10
    num3=ord(c3) - ord('0')
    return num1 + num2 + num3

def menu(choice):

    if(choice==1):

        print("Enter positive integer ")
        num = int(input())
        print("minimal divisor is",minDiv(num))

    elif(choice==2):
        print("Enter positive integer ")
        num = int(input())
```

```python
        print("maximal divisor is",maxDiv(num))

    elif choice == 3:
        print("Enter two numbers ")
        num1 = int(input())
        num2 = int(input())
        print("LCM is",lcm(num1, num2))

    elif choice == 4:
        print("Enter 3 chars ")
        a = input()
        b = input()
        c = input()
        print("chars to number is",charToNumber(a, b, c))

    else:
        print("invalid input")

def main():
        choice = int(input("enter choice "))
        menu(choice)
main()
```

Solution in C

```c
#include<stdio.h>
int gcd(int, int);
int lcm(int, int);
int minDiv(int);
int maxDiv(int);
int charToNumber(char, char, char);
void menu(int);
int main() {
    int choice;
    printf("enter choice\n");
    scanf("%d", &choice);
    menu(choice);
    return 0;
}
int gcd(int n, int m) {
    int t;
    while (m > 0) {
        t = m;
        m = n % m;
        n = t;
    }
    return n;
}
```

```c
int lcm(int num1, int num2) {

    if (num1 == 0 || num2 == 0)
        return 0;
    num1 = num1 > 0 ? num1 : -num1;
    num2 = num2 > 0 ? num2 : -num2;
    return num1 * num2 / gcd(num1, num2);
}
int minDiv(int num)
{
    int i;
    if (num % 2 == 0)
        return 2;
    for (i = 3; i * i <= num; i += 2)
        if (num % i == 0)
            return i;
    return num;
}
int maxDiv(int num)
{
    int i;
    if (num % 2 == 0)
        return num / 2;
    for (i = 3; i * i <= num; i += 2)
        if (num % i == 0)
            return (num / i);
    return 1;
}
int charToNumber(char c1, char c2, char c3) {
    return (c1 - '0') * 100 + (c2 - '0') * 10 + (c3 - '0');
}
void menu(int choice) {
    int num1, num2;
    char a, b, c;
    switch (choice) {
    case 1:
        printf("enter positive integer\n");
        scanf("%d", &num1);
        printf("minDiv(%d)=%d\n", num1, minDiv(num1));
        break;
    case 2:
        printf("enter positive integer\n");
        scanf("%d", &num1);
        printf("maxDiv(%d)=%d\n", num1, maxDiv(num1));
        break;
    case 3:
        printf("enter two integers\n");
```

```
            scanf("%d%d", &num1, &num2);
            printf("LCM(%d,%d)=%d\n", num1, num2, lcm(num1, num2));
            break;
        case 4:
            char temp;
            //to read a char after choice value was entered
            scanf("%c", &temp);
            printf("enter 3 chars\n");
            scanf("%c%c%c", &a, &b, &c);
            printf("charToNumber(%c,%c,%c)=",a,b,c);
            printf("%d\n",charToNumber(a, b, c));
            break;
        default:
            printf("invalid input\n");
        }
}
```

VI. Practice Exercises

1. Write a program that reads a sequence of non-negative integers. The first negative integer terminates the input. For each input value, find and print the square root of the input number. Use function `sqrt` from `math.h` library.

2. Write a function `power` that has two positive integer parameters, `base` and `exponent`, and returns the value of $\texttt{base}^{\texttt{exponent}}$. For example, if the base is `2` and exponent is `3,` the power function returns `8`. Write a program that reads the sequence of positive numbers. The first negative number indicates the end of the input sequence. Assume that the number of input numbers is even. Starting from the first input number, consider each pair of inputs as `base` and `exponent`, and calculate the $\texttt{base}^{\texttt{exponent}}$ for each pair. The program prints the results on different lines, displaying the value of `base`, `exponent`, and $\texttt{base}^{\texttt{exponent}}$. If both numbers in the pair are 0, the program will print the error message "`0^0` is not defined" For example if the input is: `2, 3, 1, 4, 2, 0, 0, 5, 0, 0, 5, 2, -4 -1`.
   The program will print:
   ```
   2^3 = 8
   1^4 = 1
   2^0 = 1
   0^5 = 0
   0^0 is not defined
   5^2 = 25
   ```

3. Write a function `millionaire` that accepts two floating-point parameters: `deposit` and `interest`. You can assume that `interest` is the floating-point number representing the interest rate divided by `100`. For example, if the interest is `2%,` the parameter value will be `0.02`. The function returns the number of years it will take to reach `$1000000` starting

from `deposit`, considering the yearly `interest` Write a program that reads an even number of non-negative numbers. The first negative number indicates the end of the input sequence. For each pair of input numbers that defines the initial `deposit` and yearly `interest`, the program calculates and prints the number of years it will take to become a millionaire.

4. Write a function `average` that accepts one integer parameter, `n`, that indicates number of floating-point numbers that your function reads from the user. The function returns the average of the input numbers. For example, if the parameter is `4` and the user inputs `1.0, 2.0, 3.0, 4.0`, the function returns `2.5 ( (1.0 + 2.0 + 3.0 + 4.0)/4).` Write a program that reads an integer indicating the number of input numbers. The program uses the function `average` to find the average of the input numbers.

5. Write three functions:

   `factorial(k)`: Accepts one integer parameter `k` and returns the factorial of `k`. (`k! = 1 * 2 * ... * k` and `0! = 1`).

   `power(x, k)`: Accepts two integer parameters `x` and `k`. Calculates x^k (returns `1` if `k = 0` for any positive value of `x`).

   `valueExp(x, n)`: Accepts two integer parameters `x` and `n`. Calculates the approximate value of e^x using the formula:
   $e^x = 1 + x/(1!) + x^2/(2!) + x^3/(3!) + x^4/(4!) + ...$, where `n` is the number of terms in the sum.

   Write a program that reads a sequence of positive integers. The input terminates when the first non-positive integer is encountered. Assume that the number of input numbers is even. Each pair of input values represents a pair `(x, n)`, where:

   - `x` is the base value for the exponentiation.
   - `n` is the number of terms used to approximate e^x using the `valueExp` function.

   For each pair `(x, n)`, compute the approximate value of e^x using `n` terms of the formula provided. Print the results on separate lines, displaying the values of `x`, `n`, and the computed value of e^x.

6. Write a function `dieRes` that rolls a six-sided die one time using a random number generator. The function returns a value between `1` and `6`, indicating the number rolled. Write a program that reads one integer, indicating how many times the six-sided die will be rolled. The program calculates and returns the frequencies of each number between 1 and 6 that

appear.

7. Current USDA dietary guidelines suggest that adult men and women consume at least 10 percent of their total calories from protein. Write a function `min_protein` that takes one parameter, the number of items a person consumed on a specific day. The function reads the number of calories for each item, calculates and prints the total number of calories the person consumed, and returns the minimal amount of protein (10% of total calories) the person should consume. For example, a person who consumes 2,000 calories per day would need to consume at least 200 calories from protein each day. Write `main` to test the function. Ensure the input validity.

8. Airline Frequent Flyer Miles Programs allow passengers to earn miles based on their fare class (business (1) / economy (2)) and membership type (general (1) / gold (2) / platinum (3)). Below is a table explaining how miles are earned for each dollar spent on airfare

Earning Miles Rules			
Type of Ticket	General Membership	Gold Membership	Platinum Membership
Business Class	7 miles per dollar spent	11 miles per dollar spent	13 miles per dollar spent
Economy	5 miles per dollar spent	7 miles per dollar spent	8 miles per dollar spent

Write a function `miles_one_trip` that takes three parameters: fare class, membership type, and price. The function returns the number of miles earned for the trip. Write `main` to test the `miles_one_trip` function.

9. Instagram calculates the engagement rate for a post using the following formula:

((Likes + Comments) / Followers) x 100

Write a function `ave_rate` that has one parameter: the number of posts an Instagram user posts per week. For each post, the function reads the number of likes, the number of comments, and the number of followers at the time the post was published. The function calculates and prints the engagement rate for each post using the above formula. In addition, the function calculates and returns the average engagement rate per week.

For example, if the user posted 3 posts, and for each post the data is:

- Post 1: Likes = 14, Comments = 1, Followers = 91
- Post 2: Likes = 24, Comments = 0, Followers = 90
- Post 3: Likes = 4, Comments = 3, Followers = 94

The function prints:

- Rate for post 1: 16.483516

- Rate for post 2: 26.666667
- Rate for post 3: 7.446809

The function returns the average rate: 16.865664.

Write `main` to test the function.

10. According to the American Diabetes Association, the normal 2-hour after-meal blood sugar levels for a person without diabetes are less than 140 mg/dL. For a person with prediabetes, the range is from 140 mg/dL to 199 mg/dL. For a person with Type 1 or Type 2 diabetes, the levels are 200 mg/dL or higher.

    Write a function `sugar_level` that reads a sequence of positive integers. The first negative or zero value terminates the input. Each integer indicates the blood sugar reading 2 hours after a meal. The function calculates and prints the average after-meal blood sugar level and determines if the person doesn't have diabetes, has prediabetes, or has Type 1 or Type 2 diabetes.

    For example, if the input is: 132, 135, 145, 156, 123, 90, -1, the average after-meal blood sugar level is 130.166667, and the person doesn't have diabetes.

    Write `main` to test the function.

11. Write a function `odd_digits` that has one integer parameter and finds and prints (returns in Python) the sum and average of odd digits. The function must handle the special case of no odd digits in the number. Write `main` to test the function on a randomly generated integer.

    Example 1:

    - Input: 26448
    - Output: No odd digits

    Example 2:

    - Input: 67132
    - Output: Sum of odd digits is 11, average of odd digits is 3.6666667

12. All stores in a particular store chain assign a 6-digit positive integer number for each product; this number is called the item ID. The last 3 digits of the item ID represent the category ID.

Write a function `count_category` that has two parameters: the number of items in the store and the category ID. The function reads the 6-digit ID for each item and finds and returns the number of items in the store that belong to the specific category.

Write a function `ave_category` that has two parameters: the number of stores and the category ID. The function reads the number of items in each store, then uses the function `count_category` to find the number of items in the specific category in each store. It then calculates and returns the average number of items in that category per store.

Write `main` to read the number of stores and a 3-digit positive number representing the category ID. The program should find and print the average number of items in that category across all stores.

13. Write the following functions:

    1. `count_plus` that reads a sequence of characters. The first '*' terminates the input. The function counts and returns the number of '+' signs among the input.
    2. `printAscii` that has one character parameter, `ch`. The function prints the character on one line the number of times equal to `ASCII of ch % 10`. For example, if `ch` is 'A', the function prints 'A' 5 times, since the ASCII of 'A' is 65 and `65 % 10` is 5. If `ch` is 'd', nothing will be printed, since the ASCII of 'd' is 100 and `100 % 10` is 0.
    3. Write a function `menu` that has one integer parameter, `choice`. If `choice` is 1, the function calls `count_plus` and outputs the result. If `choice` is 2, the function reads one character and calls `printAscii` to perform the task. If `choice` is not 1 or 2, the function prints an error message.

    Write `main` to test the function `menu`.

14. Write the following functions:

    - `sum_ASCII_digits`: This function reads a sequence of characters terminated by '*'. It calculates and returns the sum of ASCII values of the input characters that are digits.
    - `maxASCII`: This function reads a sequence of characters terminated by '*'. It identifies and returns the character with the highest ASCII value. For instance, given the input 'Aad6', the function returns 'd'.

    Write a function `menu` that takes one parameter, choice. If choice is 'a', `sum_ASCII_digits` is called, and the sum of ASCII values is printed. If `choice` is 'b', `maxASCII` is called, and the character with the highest ASCII value from the input is printed. For any other value of `choice,` an error message is printed.

    Write `main` to read one character, `choice`, and call **`menu`** to execute one of the tasks.

15. Write the following functions:

Write the following functions:

1. `print_char`: This function has three parameters, a character `ch` and two integers `m` and `n`. It prints an `m` by `n` rectangle of the character `ch` (where `m` is the number of rows and `n` is the number of columns). For example, if `ch` is 'A', `m` is 3, and `n` is 5, the function prints:

   ```
   AAAAA
   AAAAA
   AAAAA
   ```

2. `ave_div_5`: This function reads a sequence of positive integers. The input terminates when the user enters the first zero or negative number. The function returns the average of the inputs that are divisible by 5. If there are no inputs divisible by 5, the function returns -1.

3. `string_value`: This function has one integer parameter `size`, reads `size` characters, and returns the numeric value of the input based on the following rules: all letters (both uppercase and lowercase) will have a numeric value equal to their ASCII value, all digits will have a numeric value equal to the numeric value of the digit (for example, '0' will have a value of 0, '1' will have a value of 1, and so on), and all special characters will have a value of -1.

4. `menu`: This function has one parameter `choice` that performs the following:

- If `choice` is 1, the function randomly generates `m` and `n`, which are integers between 3 and 5, then asks the user to enter a character and calls the function `print_char`.
- If `choice` is 2, the function calls the function `ave_div_5` and prints the result.
- If `choice` is 3, the function randomly generates `size`, which is an integer between 5 and 20, then calls the function `string_value` and prints the result.
- If `choice` is not 1, 2, or 3, the function prints an error message.

5. `main`: This function randomly generates `choice`, which is an integer between 1 and 4. The program prints the randomly generated choice and tests the function `menu`.

16. To graduate, a student needs to complete a certain number of credits. Write a function `can_graduate` that has one parameter: the number of required credits a student needs to complete to graduate. The function reads a sequence of positive numbers, where each input value is the number of credits completed in one semester. The first negative or zero value terminates the input. The function finds and prints the total number of credits the student has completed overall. The function then checks if the student is eligible to graduate (i.e., if they

have completed at least the required number of credits) and returns 1 if the student is eligible and 0 otherwise. Write a main function to test your function.

Example 1: Suppose the required number of credits is 121 and the input numbers are 16, 15, 15.5, 17, 18.5, 20, 19.5, 20, -1.

The total number of completed credits is: 141.5. The student is eligible to graduate.

Example 2: Suppose the required number of credits is 124.5 and the input numbers are 16, 15, 15.5, 15, 12.5, 12, 20, -1.

The total number of completed credits is: 106. The student is not eligible to graduate.

17. The air pressure in tires must be at a certain level for different car types. For example, for cars, the recommended air pressure is between 28 and 32 PSI (pounds per square inch). It is known that changes in temperature affect tire pressure. We will use the following simple rule: a 2% pressure change for every 10-degree Fahrenheit change in temperature (tire pressure increases by 2% for every 10-degree Fahrenheit increase or decreases by 2% for every 10-degree Fahrenheit decrease).

    Write a program that asks for the number of cars, the current day's Fahrenheit temperature, and the projected Fahrenheit temperature for the next day. The program will then read the current air pressure for each car (one tire per car). It should calculate and print the air pressure for each car on the next day.

    Example 1:
    Suppose we have 3 cars, and the current tire pressures are: 32, 28, and 30.
    Today's Fahrenheit temperature is 67, and tomorrow's projected Fahrenheit temperature is 72.
    Output: 32.32, 28.28, 30.3
    Explanation: A 5-degree increase in temperature causes a 1% increase in pressure.

    Example 2:
    Suppose we have 3 cars, and the current tire pressures are: 32, 28, and 30.
    Today's Fahrenheit temperature is 67, and tomorrow's projected Fahrenheit temperature is 63.
    Output: 31.744, 27.776, 29.76
    Explanation: A 4-degree decrease in temperature causes a 0.8% decrease in pressure.

Chapter II: Simple Recursive Functions

I. Short Theory Overview

Recursion is a method of solving a problem by breaking it down into smaller instances of the same problem. A recursive function is a function that calls itself. Both Python and C programming languages support recursive functions.

A recursive function consists of three steps: a. A base case – the condition that stops the recursion call. b. A recursive call for each sub-problem. c. Combining solutions of smaller sub-problems into the solution of the original problem.

When designing a recursive solution, one needs to:

1. Define the solution of the problem in terms of solutions to smaller sub-problems of the same problem.
2. Determine the base case that stops the recursive call.

II. Sample Solutions in Python and C

Example 1: Calculating `n!` for `n≥0`

Recursive definition of factorial:

- Base cases:
    - `0! = 1`
    - `1! = 1`
- Recursive call:
    - For `n>1, n! = n×(n−1)!`

Factorial of `n` is defined as the product of n and the factorial of `(n−1)`. We define factorial recursively in terms of its smaller instances.

Example: Recursive calculation of `5!`

We have a sequence of recursive calls that stop at `1!`

```
5!=5*4!
4!=4*3!
3!=3*2!
2!=2*1!
```

As soon as the base case is reached, the values are returned to the previous steps, and the final value is calculated.

```
1!=1
2!=2*1=2
```

```
3!=3*2=6
4!=4*6=24
```
`5!=5*24=120` – final value.

Programming implementation in Python

Write a recursive function `factorial` that takes one integer parameter `n` and returns `n!`. Write a main function to test the function.

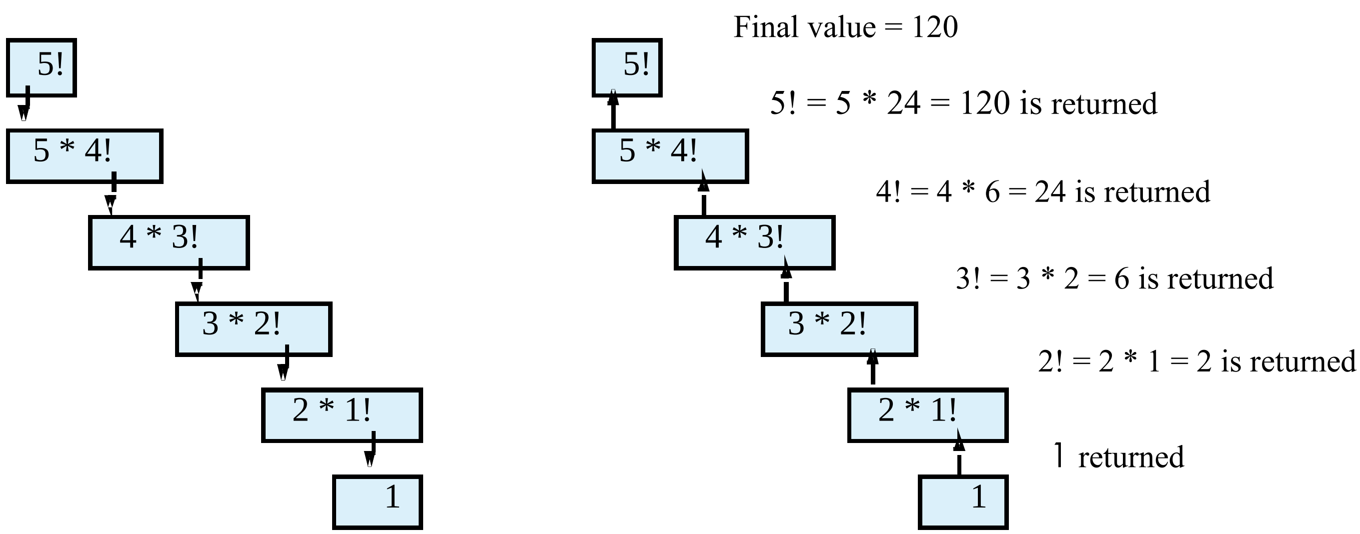
```
def factorial(n):
     if(n<=1): #base cases 0!=1, 1!=1
          return 1
     else:
          return n*factorial(n-1) #recursive call


def main():
        num=int(input("enter a positive integer "))
        if(num>0):
                print(num,end='!=')
                print(factorial(num))
        else:
                print("invalid input")

  main()
```

Programming implementation in C

```
#include <stdio.h>
int factorial(int);
int main() {
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    if (num < 0) {

        printf("Error: Input must be a positive integer.\n");
    }
    else {

        printf("Factorial of %d is %d\n", num, factorial(num));
    }

    return 0;
  }
```

```
int factorial(int n) {
    if (n<=1) {
        return 1; // Base case: 0!=1 and 1!=1
    }
    else {
        return n * factorial(n - 1); // Recursive case
    }
}
```

Example 1: calculating 5!



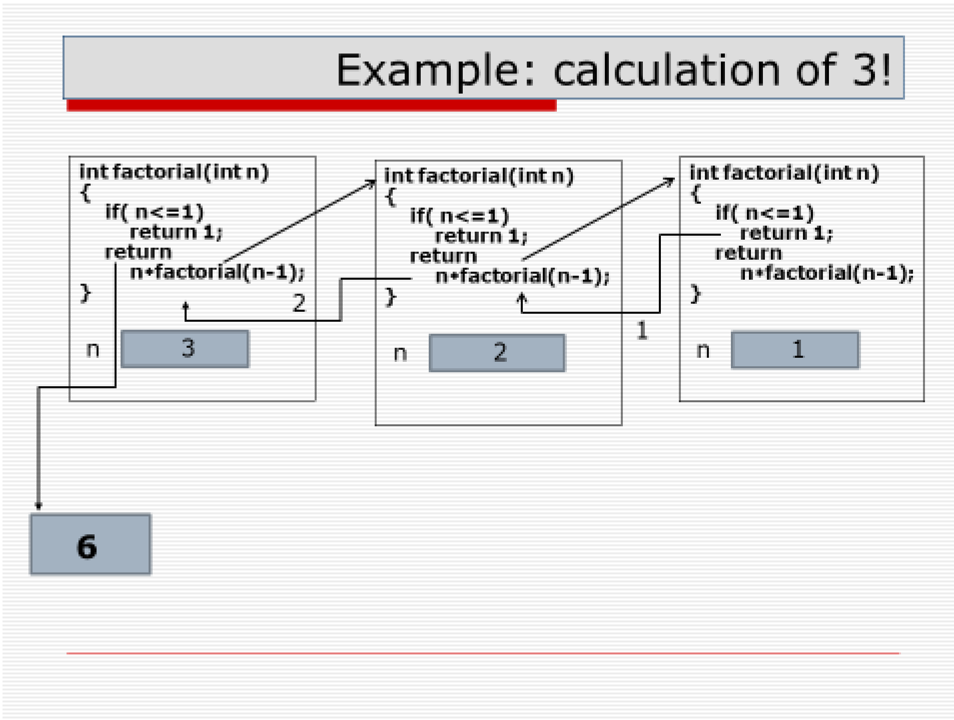
(a) Sequence of recursive calls

(b) Values returned from each recursive call

Calculating `factorial(5)` will create a following sequence of function calls:

```
5*factorial(4)
4*factorial(3)
3*factorial(2)
2*factorial(1)
```

As soon as the stopping condition is reached, `factorial(1)` returns the value 1, and `factorial(2)` is calculated and returns the value 2. Then `factorial(3)` is calculated and returns the value 6, `factorial(4)` is calculated and returns the value 24, and finally, `factorial(5)` is calculated and returns the value 120.



Example 2: Calculating a^n for $n \geq 0$

Recursive formula 1:

$a^0=1$ – base case

$a^n=a*a^{n-1}$ – recursive call

Recursive formula 2:

$a^0=1$ – base case

$a^n=a^{n/2} * a^{n/2}$, if n is even

$a^n=a^{n/2} * a^{n/2} * a$, if n is odd

We demonstrate several recursive solutions for the function. In the main function, we only show how to test one of them.

Programming implementation in Python

```
def power(a, n):
     if(n==0):
          return 1
     else:
          return a*power(a, n-1)
     #time complexity O(n)

def power1(a, n):
    """
    Computes a^n with redundant calculations.
    """
    if n == 0:
        return 1
    if(n % 2):
        return power1(a, n // 2) * power1(a, n // 2) * a
    else:
        return power1(a, n // 2) * power1(a, n // 2)

    """
    Time complexity is O(n)
    The recurrence relation for this solution is:
    T(n) = 2T(n // 2) + O(1)
    It solves to O(2^log(n)) = O(n)
    """
def power2(a, n):
    """
    Improved solution:
    To avoid redundant calculations, this version computes
    power2(a, n // 2) once and stores its result.
    """
    if n == 0:
        return 1
    temp = power2(a, n // 2)
    if n % 2 == 1:
        return temp * temp * a
    else:
        return temp * temp

    """
    Time complexity: O(log(n))
    The recurrence relation for this solution is:
    T(n) = T(n // 2) + O(1)
    It solves to O(log(n))
    """
def main():
        print("program calculates a^n")
```

```
        a=int(input("enter base a "))
        n=int(input("enter power n, a positive integer "))
        if(n>=0):
                print(a,"^",n,"=",power(a,n))
        else:
                print("invalid input")

main()
```

Programming implementation in C

```
#include <stdio.h>
int power(double, int);
int main() {
    int n;
    double a;
    printf("Enter base a and exponent n: ");
    scanf("%lf%d", &a,&n);
    if (n < 0) {

        printf("Invalid Input"\n");
    }
    else {

        printf("%f^%d=%f\n", a,n,power(a,n));
    }

    return 0;
}
double power(double a, int n) {
    if (n == 0)
        return 1;
    else
        return a * power(a, n - 1);
    /* Time complexity is O(n) */
}

double power1(double a, int n){}

    if (n == 0)
        return 1;
    if (n % 2)
        return power(a, n / 2) * power(a, n / 2) * x;
    else
        return power(a, n / 2) * power(a, n / 2);

    /* Time complexity is O(n)
       The recurrence relation for this solution is:
       T(n) = 2T(n/2) + O(1)
```

```
        It solves to O(2^log(n)) = O(n)
    */
}

double power2(double a, int n) {
    /* Improved solution:
       To avoid redundant calculations, this version computes
       power2(a, n / 2) once and stores its result.
    */
    double temp;
    if (n == 0)
        return 1;
    temp = power2(a, n / 2);
    if (n % 2)
        return temp * temp * a;
    else
        return temp * temp;

    /* Time complexity: O(log(n))
       The recurrence relation for this solution is:
       T(n) = T(n / 2) + O(1)
       It solves to O(log(n))
    */
}
```

Example 3: Calculation the sum of powers: $S_n = a^1 + a^2 + a^3 + \ldots + a^n$, for $n \geq 1$

Recursive formula 1:

$S_1 = a$

$S_n = S_{n-1} + a^n$

Recursive formula 2:

Since $a^1 + a^2 + a^3 + \ldots + a^n = a + a*(a+a^2+\ldots+a^{n-1})$

$S_n = a + a*S_{n-1}$

$S_1 = a$

We demonstrate several recursive and iterative solutions for the function. In the main function, we only show how to test one of them.

Programming implementation in Python

```
def power(a, n):
     if(n==0):
          return 1
     else:
          return a*power(a, n-1)
```

```python
def sum_powers_rec1(a, n):
     if(n==1):
          return a
     else:
          return sum_powers_rec1(a, n-1)+power(a, n)
def sum_powers1(a, n):
    sum = 0
    for i in range(1, n + 1):
        sum += a ** i
    return sum
    # The number of operations: 1+2+…+n=(1+n)*n/2
    # Time complexity: O(n^2)

def sum_powers_rec2(a, n):
    # Recursive formula 2 implementation
    if n == 1:
        return a
    return a + a * sum_powers_rec_2(a, n - 1)

def sum_powers2(a, n):
    # Iterative solution using the fact:
    # Sn = a + a(a + a(a + ... a(a + a*a) ...))
    sum = a
    for i in range(2, n + 1):
        sum = a + a * sum
    return sum
    # Time complexity: O(n)
    # The loop runs n-1 times, performing
    # a constant amount of work in each iteration

def main():
        print("program calculates: a+a^2+a^3+..a^n")
        a=int(input("enter a "))
        n=int(input("enter n, a positive integer "))
        if(n>0):
                print("Sum of powers =",sum_powers_rec1(a,n))
        else:
                print("invalid input")

main()
```

Programming implementation in C

```c
#include <stdio.h>
double power(double, int);
double sum_powers_rec1(double, int);
double sum_powers_rec2(double, int);
```

```c
double sum_powers1(double, int);
double sum_powers2(double , int);

int main() {
    int n;
    double a;
    printf("Enter a and n: ");
    scanf("%lf%d", &a,&n);
    if (n <= 0) {

        printf("Invalid Input"\n");
    }
    else {

        printf("Sum of powers = %f\n", sum_powers_rec1(a,n));
    }

    return 0;
}
double power(double a, int n) {
    if (n == 0)
        return 1;
    else
        return a * power(a, n - 1);
}

double sum_powers_rec1(double a, int n) {
    //implementation of recursive formula 1
    if (n == 1)
        return a;
    else
        return sum_powers_rec1(a, n-1) + power(a,n);
}

double sum_powers1(double a, int n)
{
    //iterative solution
    double sum = 0;
    int i;

    for (i = 1; i <= n; i++) {}
        sum += power(a, i);
    }
    return sum;
    //the number of operations: 1+2+…+n=(1+n)*n/2
    //time complexity: O(n^2)
}

double sum_powers_rec2(double a, int n)
```

```
{
    //implementation of recursive formula 2
    if (n == 1)
        return a;
    return a + a * sum_powers_rec2(a, n - 1);
}
double sum_powers2(double a, int n)
{
    //iterative solution using the fact:
    //Sn=a+a(a+a(a+…a(a+a*a)…)

    double sum = a;
    int i;
    for (i = 2; i <= n; i++)
        sum = a + a * sum;
    return sum;
    /* time complexity O(n)
    the loop runs n−1 times, performing
    constant amount of work in each iteration */

}
```

Example 4:
Calculating the nth Fibonacci number.

Definition:

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}, \ n>1$

We will demonstrate both recursive and iterative solutions.

Solution in Python (functions only)

```
def fibonacci_rec(n):
     # iterative solution

if n <= 1:
        return n
    return fibonacci_rec(n - 1) + fibonacci_rec(n - 2)


def fibonacci(n):
    # iterative solution
    if n == 0:
        return 0
    if n == 1:
```

```python
        return 1

    prev2 = 0 #F0
    prev1 = 1 #F1

    for _ in range(2, n + 1):
        current = prev1 + prev2 #F(n) = F(n-1) + F(n-2)
        prev2 = prev1           #Update F(n-2) to F(n-1)
        prev1 = current         #Update F(n-1) to F(n)

    return current
```

Solution in C (functions only)

```c
int fibonacci_rec(int n){

    // recursive solution
    if (n <= 1)
        return n;
    return fibonacci_rec(n - 1) + fibonacci_rec(n - 2);
}


int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    int prev2 = 0; // F0
    int prev1 = 1; // F1
    int current;

    for (int i = 2; i <= n; i++) {
        current = prev1 + prev2; // F(n) = F(n-1) + F(n-2)
        prev2 = prev1;          // Update F(n-2) to F(n-1)
        prev1 = current;        // Update F(n-1) to F(n)
    }

    return current;
}
```

The recursive Fibonacci solution is not efficient with exponential running time, $O(2^n)$. Each call to `fibonacci(n)` results in two additional recursive calls: `fibonacci(n - 1)` and `fibonacci(n - 2)`. This leads to a large number of redundant calculations significantly increasing the total computation time.

The time complexity of the iterative Fibonacci solution is `O(n)`. The algorithm is uses a single loop that runs `n - 1` times.

Example 5:

Calculating the greatest common divisor (GCD) of two integers.

Definition:

- `gcd(a, b) = gcd(b, a % b), where b ≠ 0`
- `gcd(a, 0) = a`

We will demonstrate both recursive and iterative solutions.

Solution in Python (functions only)

```
def gcd(a, b):
    #iterative solution
    while b > 0:
        t = b
        b = a % b
        a = t
    return a
def gcd_rec(a, b):
    # Recursive solution
    if b == 0:
        return a
    return gcd_rec(b, a % b)
```

Solution in C (functions only)

```
int gcd(int a, int b){
    //iterative solution
    int t;
    while (b > 0)
    {
        t = b;
        b = a % b;
        a = t;
    }
    return a;
}


int gcd_rec(int a, int b){
    //recursive solution
    if (b == 0)
```

```
        return a;
    return gcd_rec(b, a % b);
}
```

Example 6:

Write a function `reverse_print` that takes one integer parameter and prints its digits in reverse order. For example, if the parameter is 37856 the function prints 65873.

Recursive relationship for the given problem:

Base case: if the number is a single-digit number (i.e., less than 10), return the same number
Recursive call:

- Print the last digit of the number.
- Then, recursively print the remaining digits in reverse order (without last digit).

We demonstrate several recursive solutions.

Solution in Python (functions only)

```
def reverse_print(n):
    if n < 10:
        print(n, end="")
    else:
        print(n % 10, end="")
        reverse_print(n // 10)

def reverse_print1(n):
    # Shorter version
    print(n % 10, end="")
    if n >= 10:
        reverse_print1(n // 10)
```

Solution in C (functions only)

```
void reverse_print(int n){
    if (n < 10)
        printf("%d", n % 10);
    else
    {
        printf("%d", n % 10);
        number_rev_print(n / 10);
    }
}
void reverse_print1(int n){}
    //shorter version
```

```
    printf("%d", n % 10);
    if (n >= 10)
        number_rev_print(n / 10);
}
```

Example 7:

Write a function write_rev that reads a sequence of characters terminated by '*'. The function prints the input sequence in reverse order, i.e., the last entered character will be printed first, and the first entered character will be printed last.

Recursive relationship for the given problem:

Base case: if the input character is '*', do nothing (terminate the recursion)
Recursive call:

- Read the input character `ch`
- Then, recursively print the remaining characters in reverse order (excluding the last character).
- Finally, print the character `ch`

Solution in Python (function only)

```python
def write_rev():
    ch = input()  # Read the input character
    if ch != '*':
        write_rev()  # Recursively call for remaining characters
        print(ch, end="")  # Print the character
```

Solution in C (function only)

```c
void write_rev(void){
    char ch;
    if ((ch = getchar()) != '*'){//Read the input character
        write_rev(); //Recursively call for remaining characters
        printf("%c",ch); //Print the character
    }
}
```

Example 8:

Write a recursive function max_rec that reads a sequence of non-negative integers terminated by -1. The function finds and returns the maximal element of the sequence.

If the input is empty (only -1 was entered), the function returns -1.

Recursive relationship for the given problem:

Base Case: if the input number is -1, return -1

Recursive Call:

Computer the maximum number:

- If the input number is greater than the result of recursive call, return the input number
- Otherwise, return the result of the recursive call

Solution in Python (function only)

```
def max_rec():
    num = int(input("Enter a number: "))
    if num == -1:
        return -1
    max_num = max_rec()
            return num if num > max_num else max_num
```

Solution in C (function only)

```
int max_rec(){
     int max_num, num;
     scanf("%d", &num);
     if (num == -1)
          return -1;
     max_num = max_rec();
     return num > max_num ? num : max_num;
}
```

Example 9:

Towers of Hanoi (classic problem)

Invented by French mathematician Édouard Lucas in 1883, the Tower of Hanoi was originally called "The Tower of Brahma." It's associated with a legend where priests move disks in a temple, with the world ending when the task is complete.

The puzzle involves three rods and disks of different sizes stacked in ascending order on one rod. The goal is to move all the disks to the target rod following these rules:

1. Move one disk at a time.
2. A disk can only be placed on a larger disk or an empty rod.

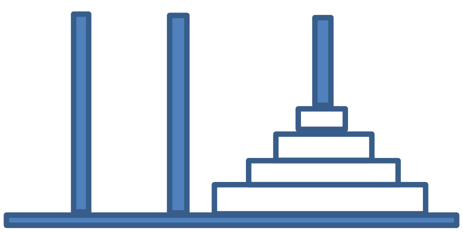
Write a recursive function hanoi that solves the Tower of Hanoi puzzle for a given number of disks. It determines and prints the sequence of moves needed to transfer all disks from a source pole to a target pole using an auxiliary pole, following the rules of the puzzle.

Function Parameters:

- `n (int)`: The number of disks.
- `sp (int)`: The source pole (where disks start).
- `tp (int)`: The target pole (where disks should end up).
- `ap (int)`: The auxiliary pole (used as a temporary storage).

Recursive relationship for this problem:

- **Base Case**:
  - If `n = 1`, move the disk from the source pole to the target pole.
- **Recursive Case**:
  - If `n > 1`:
    1. Move `n-1` disks from the source pole to the auxiliary pole, using the target pole as a temporary holding area.
    2. Move the `nth` disk from the source pole to the target pole.
    3. Move `n-1` disks from the auxiliary pole to the target pole, using the source pole as a temporary holding area.



Solution in Python

```python
def hanoi(n, sp, tp, ap):
    if n == 1:
        print(f"Move disk from pole {sp} to pole {tp}")
    else:
        hanoi(n - 1, sp, ap, tp)
        print(f"Move disk from pole {sp} to pole {tp}")
        hanoi(n - 1, ap, tp, sp)

def main():
```

```python
    n = int(input("Enter number of disks: "))
    print(f"HANOI TOWERS with {n} DISKS:\n")
    hanoi(n, 1, 3, 2)
    #Instructs the program to move n disks from pole 1 (source)
    #to pole 3 (target) using pole 2 as the auxiliary pole.

main()
```

Solution in C

```c
// SP - source pole (1), TP - target pole (3), AP - auxiliary
pole (2)
void hanoi(int n, int  sp, int  tp, int  ap)
{
     if (n == 1)
          printf("move disk from pole %d to pole %d\n", sp, tp);
     else
     {
          hanoi(n - 1, sp, ap, tp);
          printf("move disk from pole %d to pole %d\n", sp, tp);
          hanoi(n - 1, ap, tp, sp);
     }
}
void main(void)
{
     int n;
     printf("enter number of disks\n");
     scanf("%d", &n);
     printf("HANOI TOWERS with %d DISKS: \n\n", n);
     hanoi(n, 1, 3, 2);
     //instructs the program to move n
     //disks from pole 1 (source) to pole 3 (target)
     //using pole 2 as the auxiliary pole.
}
```

The time complexity of the Tower of Hanoi problem is $O(2^n)$. This is because each solution involves two recursive calls for $n - 1$ disks plus one move, leading to the recurrence $T(n) = 2T(n-1) + 1$. Solving this gives $T(n)=2^n - 1$, so the complexity is $O(2^n)$.

It would take around 584 billion years to solve the Tower of Hanoi problem for 64 disks, assuming one move per second.

Example 10 Part I: Colored Towers of Hanoi

Setup:

- **Pegs**: Three pegs labeled Peg 1 (source), Peg 2 (auxiliary), and Peg 3 (target).

- **Disks**: There are `2m` disks, consisting of `m` red disks and `m` white disks. Each disk size has one red and one white disk. Initially, all disks are stacked on Peg 1 in descending order of size, with each white disk below its corresponding red disk.

Disk Movement Rules:

- Larger disks cannot be placed on smaller disks.
- Disks of the same size can be stacked regardless of color (e.g., white can be on red and vice versa).
- Only one disk can be moved at a time.

Objective:

Move all `2m` disks from Peg 1 (source) to Peg 3 (target) following the constraints.

Recursive Solution:

1. Move `m−1` pairs from Peg 1 to Peg 2.
2. Move the largest red disk from Peg 1 to Peg 3.
3. Move `m−1` pairs from Peg 3 to Peg 2.
4. Move the largest white disk from Peg 1 to Peg 3.
5. Move `m−1` pairs from Peg 2 to Peg 1.
6. Move the red disk from Peg 2 to Peg 3.
7. Move `m−1` pairs from Peg 1 to Peg 3.

Solution in Python (functions only)

```python
def move_red(from_peg, to_peg):
    print(f"Move red disk from {from_peg} to {to_peg}")

def move_white(from_peg, to_peg):
    print(f"Move white disk from {from_peg} to {to_peg}")

def colored_hanoi(m, from_peg, to_peg):
    via_peg = 6 - from_peg - to_peg

    if m <= 0:
        return

    colored_hanoi(m - 1, from_peg, via_peg)
```

```
    move_red(from_peg, via_peg)
    colored_hanoi(m - 1, to_peg, via_peg)
    move_white(from_peg, to_peg)
    colored_hanoi(m - 1, via_peg, from_peg)
    move_red(via_peg, to_peg)
    colored_hanoi(m - 1, from_peg, to_peg)
```

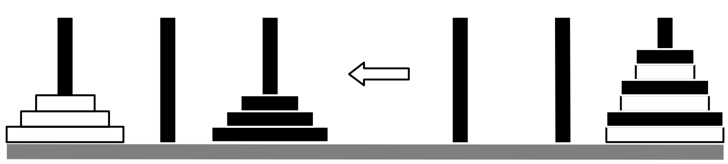
Solution in C (functions only)

```c
void movered(int from, int to) {
     printf("move red ring from %d to %d\n", from, to);
}
void movewhite(int from, int to) {
     printf("move white ring from %d to %d\n", from, to);
}
void colored_hanoi(int m, int from, int to) {
     int via = 1 + 2 + 3 - from - to;
     if (m <= 0) return;
     colored_hanoi (m - 1, from, to);
     move_red(from, via);
     colored_hanoi (m - 1, to, via);
     move_white(from, to);
     colored_hanoi (m - 1, via, from);
     move_red(via, to);
     colored_hanoi (m - 1, from, to);
}
```

Example 10 Part II: Colored Towers of Hanoi Variation

Following the colored Towers of Hanoi problem, write a function that moves m pairs of disks from the source peg to the target peg. Disks start stacked on the source peg, with each size having a red disk above a white disk. The goal is to have all white disks on the target peg (largest at the bottom) and all red disks on the source peg (largest at the bottom).

The rules are the same as those in the colored Towers of Hanoi problem.



Solution in Python

```python
def move_red(from_peg, to_peg):
    print(f"move red ring from {from_peg} to {to_peg}")
```

```python
def move_white(from_peg, to_peg):
    print(f"move white ring from {from_peg} to {to_peg}")

def colored_hanoi(m, from_peg, to_peg):
    via_peg = 6 - from_peg - to_peg
    if m <= 0:
        return

    colored_hanoi(m - 1, from_peg, to_peg)
    move_red(from_peg, via_peg)
    colored_hanoi(m - 1, to_peg, via_peg)
    move_white(from_peg, to_peg)
    colored_hanoi(m - 1, via_peg, from_peg)
    move_red(via_peg, to_peg)
    colored_hanoi(m - 1, from_peg, to_peg)

def colored_hanoi1(m, from_peg, to_peg):
    via_peg = 6 - from_peg - to_peg
    if m <= 0:
        return

    colored_hanoi(m - 1, from_peg, to_peg)
    move_red(from_peg, via_peg)
    colored_hanoi(m - 1, to_peg, via_peg)
    move_white(from_peg, to_peg)
    colored_hanoi(m - 1, via_peg, to_peg)
    move_red(via_peg, from_peg)
    colored_hanoi(m - 1, to_peg, from_peg)
    colored_hanoi1(m - 1, from_peg, to_peg)
```

Solution in C (functions only)

```c
void movered(int from, int to) {
     printf("move red ring from %d to %d\n", from, to);
}
void movewhite(int from, int to) {
     printf("move white ring from %d to %d\n", from, to);
}
void colored_hanoi(int m, int from, int to) {
     int via = 1 + 2 + 3 - from - to;
     if (m <= 0) return;
     colored_hanoi(m - 1, from, to);
     move_red(from, via);
     colored_hanoi(m - 1, to, via);
     move_white(from, to);
     colored_hanoi(m - 1, via, from);
     move_red(via, to);
```

```
      colored_hanoi(m - 1, from, to);
}
void colored_hanoi1(int m, int from, int to) {
      int via = 1 + 2 + 3 - from - to;
      if (m <= 0) return;
      colored_hanoi(m - 1, from, to);
      move_red(from, via);
      colored_hanoi(m - 1, to, via);
      move_white(from, to);
      colored_hanoi(m - 1, via, to);
      move_red(via, from);
      colored_hanoi(m - 1, to, from);
      colored_hanoi1(m - 1, from, to);
}
```

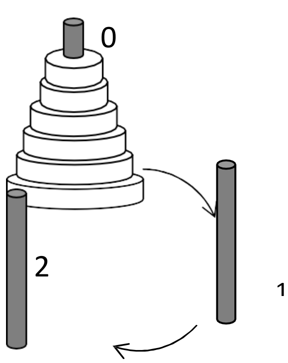
Example 11: Cyclic Hanoi Problem

You have three rods arranged in a circle. On one of these rods, there are `n` rings of different sizes stacked in a pile. Each ring (except the topmost one) is placed on a smaller ring. The rules are similar to the Towers of Hanoi problem, with the following constraints:

1. **Placement Rule**: No ring can be placed on top of a smaller ring.
2. **Movement Constraint**: Rings can only be moved between adjacent rods in a clockwise direction:
    - You can move a ring from rod 0 to rod 1.
    - You can move a ring from rod 1 to rod 2.
    - You can move a ring from rod 2 to rod 0.
    - Direct moves between rod 0 and rod 2 are not allowed.

Write a recursive function named `CyclicHanoi` that:

- Takes the number of rings `n`, the source rod `source`, and the target rod `target` as parameters. The source and target rods are integers (`0`, `1`, or `2`) and are different (`target ≠ source`).
- Prints the sequence of moves required to transfer all the rings from the source rod to the target rod, while ensuring the movement constraints are followed.



Solution in Python

```python
def move(source, target):
    print(f"Move ring from rod {source} to rod {target}")

def next(rod):
    return (rod + 1) % 3

def CyclicHanoi(n, source, target):
    aux = 3 - source - target

    if n == 0:
        return

    if next(source) == target:
        CyclicHanoi(n - 1, source, aux)
        move(source, target)
        CyclicHanoi(n - 1, aux, target)
    else:
        CyclicHanoi(n - 1, source, target)
        move(source, aux)
        CyclicHanoi(n - 1, target, source)
        move(aux, target)
        CyclicHanoi(n - 1, source, target)
```

Solution in C (functions only)

```c
void move(unsigned int source, unsigned int target) {
    printf("Move ring from rod %u to rod %u\n", source, target);
}

unsigned int next(unsigned int rod) {
    return (rod + 1) % 3;
}

void CyclicHanoi(unsigned int n, unsigned int source, unsigned
int target) {
    unsigned int aux = 3 - source - target;
```

```
    if (n == 0) return;

    if (next(source) == target) {
        CyclicHanoi(n - 1, source, aux);
        move(source, target);
        CyclicHanoi(n - 1, aux, target);
    }
    else {
        CyclicHanoi(n - 1, source, target);
        move(source, aux);
        CyclicHanoi(n - 1, target, source);
        move(aux, target);
        CyclicHanoi(n - 1, source, target);
    }
}
```

Example 12: Part I: Numbers of Hanoi Simple Version)

Numbers of Hanoi Problem

The Numbers of Hanoi problem is similar to the classic Towers of Hanoi problem:

- **Setup**: There are 3 rods and n rings. Initially, all rings are arranged on one rod, ordered from the largest (at the bottom) to the smallest (at the top). The rods are numbered 0, 1, and 2. The rings are numbered from 1 ton, with 1 being the smallest and n being the largest.

- **Goal**: Move all rings to the rods such that each ring is located on the rod whose number is the remainder when the ring number is divided by 3. Specifically:
    - Rings numbered 3, 6, 9, ... should be on rod 0.
    - Rings numbered 1, 4, 7, ... should be on rod 1.
    - Rings numbered 2, 5, 8, ... should be on rod 2.

Rules of the Game

1. A ring may not be placed on top of a smaller ring.
2. In this simplified version, it is allowed to move multiple top rings simultaneously while maintaining their relative order.

Solution in Python (functions only)

```
def move_several(n, from_rod, to_rod):
    print(f"move {n} from {from_rod} to {to_rod}")

def hanoi2(n, loc):
```

```
    if n == 0:
        return
    if n % 3 != loc:
        move_several(n, loc, n % 3)
    hanoi2(n - 1, n % 3)
```

Solution in C (functions only)

```
void move_several(int n, int from, int to) {
    printf("move %d from %d to %d\n", n, from, to);
}
void hanoi2(int n, int loc)
{
    if (n == 0)
        return;
    if (n % 3 != loc)
        move_several(n, loc, n % 3);
    hanoi2(n - 1, n % 3);
}
```

Example 12 Part II: Numbers of Hanoi General Version

In the general version of the problem, only one ring can be moved at a time.

Solution in Python (functions only)

```
def hanoi(n, sp, tp, ap):
    if n == 1:
        move_disk(sp, tp)
    else:
        hanoi(n - 1, sp, ap, tp)
        move_disk(sp, tp)
        hanoi(n - 1, ap, tp, sp)

def hanoi_num(n, loc):
    if n == 0:
        return
    if n % 3 != loc:
        hanoi(n, loc, n % 3, 3 - loc - (n % 3))
    hanoi_num(n - 1, n % 3)
```

Solution in C (functions only)

```
void hanoi(int n, int  sp, int  tp, int  ap)
{
     if (n == 1)
```

```c
            printf("move disk from pole %d to pole %d\n", sp, tp);
        else
        {
            hanoi(n - 1, sp, ap, tp);
            printf("move disk from pole %d to pole %d\n", sp, tp);
            hanoi(n - 1, ap, tp, sp);
        }
}
void hanoi_num(int n, int loc)
{
        if (n == 0)
            return;
        if (n % 3 != loc)
            hanoi(n, loc, n % 3);
        hanoi_num(n - 1, n % 3);
}
```

Example 13:

Write a recursive function that takes an integer n>9 with distinct digits (assume this is given and does not need verification). The function should return:

- `1` if the digits of the number are in ascending order from left to right,
- `-1` if the digits are in descending order from left to right,
- `0` if the digits are neither in ascending nor descending order.

For example:

- For the number 2489, the function should return `1`.
- For the number 31, the function should return `-1`.
- For the number 98756, the function should return `0`.

Solution in Python

```python
def is_sorted_number(n):
    if n < 10:
        return 0

    d2 = (n // 10) % 10
    d1 = n % 10

    if n < 100:
        if d2 < d1:
            return 1
        else:
```

```
        return -1

    sorted_order = is_sorted_number(n // 10)

    if sorted_order == 1 and d2 < d1:
        return 1
    if sorted_order == -1 and d2 > d1:
        return -1
    return 0
```

Solution in C

```
int is_sorted_number(int n){}
     int d2 = n % 100 / 10, d1 = n % 10, sorted;
     if (n < 100)
          if (d2 < d1)
               return 1;
          else
               return -1;
     sorted = is_sorted_number(n / 10);
     if (sorted == 1 && d2 < d1)
          return 1;
     if (sorted == -1 && d2 > d1)
          return -1;
     return 0;
}
```

III. Practice Exercises

1. Write a recursive function named `count_digits` that takes an integer parameter and returns the number of digits in the integer. Then, write a `main` function to test the `count_digits` function.

2. Write a recursive function named `sum_digits` that takes an integer parameter and returns the sum of its digits. Then, write a `main` function to test the `sum_digits` function.

3. Write a recursive function named `is_sorted` that takes an integer parameter and returns `1` if the digits in the integer are sorted in increasing order, and 0 otherwise. Then, write a `main` function to test the `is_sorted` function. For example, if the parameter is 1234, the function returns 1; if the parameter is 4312 the function returns 0.

4. Write a recursive function `count` that takes a positive integer parameter `num`. The function reads a sequence of positive integers until `-1` is entered. The function finds and returns the

number of inputs that are less than `num` (excluding -1). For example, for `num = 5` and an input sequence `5, 7, 3, 8, 1, 6, -1`, the function returns `2`. Write a `main` function to test the `count` function.

5. Write a recursive function `print_num_down` that takes a positive integer parameter `num`. The function prints all numbers from `num` down to `1`. For example, if `num = 5`, the function prints: `5 4 3 2 1`. Then, write a `main` function to test the `print_num_down` function.

6. Write a recursive function `print_num_up` that takes a positive integer parameter `num`. The function prints all numbers from `1` up to `num`. For example, if `num = 5`, the function prints: `1 2 3 4 5`. Then, write a `main` function to test the `print_num_up` function.

7. Write a recursive function `print_num_down_up` that takes a positive integer parameter `num`. The function prints all numbers from `num` down to `1` and then back up to `num`, with `1` printed only once. For example, if `num = 5`, the function prints: `5 4 3 2 1 2 3 4 5`. Then, write a `main` function to test the `print_num_down_up` function.

8. Write a recursive function `diff_even_odd` that takes a positive integer parameter `num`. The function returns the difference between the number of even digits and the number of odd digits. For example, if `num = 51637021`, the function returns `-2` (since there are `3` even digits and `5` odd digits). Then, write a `main` function to test the `diff_even_odd` function.

9. Write a recursive function that takes a positive integer as a parameter, reverses its digits, and returns the reversed number. Then, write a `main` function to test this recursive function.

10. Write a recursive function that takes a positive integer as a parameter, computes the product of its digits, and returns the result. Then, write a `main` function to test this recursive function.

11. Write a recursive function that takes two integer parameters: a positive integer `num` and a one-digit positive integer `d`. The function counts and returns the occurrences of `d` in `num`. For example, if `num=12222` and `d=2`, function returns `4`. Then, write a `main` function to test this recursive function.

12. Write a recursive function that takes a positive integer as a parameter, calculates the sum of the squares of its digits, and returns the result. Then, write a `main` function to test this recursive function.

13. A palindrome is a number that reads the same forward and backward (e.g., `121` is a palindrome, but `123` is not). Write a recursive function that takes a positive integer as a parameter and returns `1` if the number is a palindrome, and `0` otherwise. Then, write a `main` function to test this recursive function.

14. Write a recursive function that takes a positive integer as a parameter and finds and returns the largest digit in the number. For example, if the parameter is `6476`, the function returns `7`, and if the parameter is `555`, the function returns `5`. Then, write a `main` function to test this recursive function.

15. Write a recursive function that takes two integer parameters: a positive integer `num` and a one-digit positive integer `d`. The function counts and returns the number of digits in `num` that are less than `d`. For example, if `num = 6543127` and `d = 6`, the function returns `5`; if `num = 7685` and `d = 1`, the function returns `0`. Then, write a `main` function to test this recursive function.

16. Write a recursive function that takes two integer parameters: a positive integer `num` and a one-digit positive integer `d`. The function counts and returns the number of digits in `num` that are greater than `d`. For example, if `num = 6543127` and `d = 6`, the function returns `1`; if `num = 7685` and `d = 1`, the function returns `4`. Then, write a `main` function to test this recursive function.

17. Write a recursive function that takes a positive integer `n` as a parameter and prints Fibonacci numbers up to the given limit `n`. `n` is not the index but the value of the numbers. For example, if `n=20`, the function prints `0, 1, 1, 2, 3, 5, 8, 13`; if `n = 21`, the function prints `0, 1, 1, 2, 3, 5, 8, 13, 21.` Then, write a `main` function to test this recursive function.

18. Write a recursive function that takes a positive integer `n` as a parameter and calculates and returns the sum of the first `n` Fibonacci numbers. For example, if `n=5`, the function returns `7`; if `n=3`, the function returns `2`. Then, write a `main` function to test this recursive function.

19. Write a recursive function that takes two positive integer parameters, `n` and `m`. The function finds and returns the Fibonacci number at position `n` modulo `m`. For example, if `n=7` and `m=3`, the function returns `2`, since the 7th Fibonacci number is `8` and `8mod 3 = 2`. Then, write a `main` function to test this recursive function.

20. Write a recursive function that takes a positive integer `n` as a parameter and calculates and returns the largest Fibonacci number less than `n`. For example, if `n=30`, the function returns `21`, since the Fibonacci numbers less than `30` are `0, 1, 1, 2, 3, 5, 8, 13, 21`.

Thus, `21` is the largest Fibonacci number less than `30`. Then, write a `main` function to test this recursive function.

21. Write a recursive function that takes a positive integer `n` as input and returns the count of Fibonacci numbers that are less than `n`. For example, if `n=30`, the function returns `9`, because there are `9` Fibonacci numbers less than `30: 0, 1, 1, 2, 3, 5, 8, 13,` and `21`. Then, write a `main` function to test this recursive function.

22. Write a recursive function that takes a positive integer `n` and returns the number of its divisors. For example, for `n=12`, the function should return `6`, as the divisors of `12` are `1, 2, 3, 4, 6,` and `12`. Then, write a `main` function to test this recursive function.

23. Write a recursive function that takes two positive integers: `n` and `m`. The function should return the number of divisors of `n` that are less than `m`. For example, if `n=30` and `t=10`, the function should return `4`, as the divisors of `30` that are less than `10` are `1, 2, 3,` and `5`. Then, write a `main` function to test this recursive function.

24. Write a recursive function that takes three positive integers: `n, a,` and `b`. The function should compute and return the sum of all divisors of `n` that are within the range `[a,b]` (inclusive). For example, if `n=30, a=2,` and `b=10`, the function should return `20`, as the divisors of `30` within this range are `2, 3, 5,` and `10`, and their sum is `20`. Then, write a `main` function to test this recursive function.

25. Write a recursive function that takes two positive integers: `n` and `m`. The function should find and return the largest divisor of `n` that is less than `m`. For example, if `n=30` and `m=10`, the function should return `6`, as `6` is the largest divisor of `30` that is less than `10`. Then, write a `main` function to test this recursive function.

26. Write a recursive function that takes a positive integer `n` and computes and returns the product of all its divisors. For example, if `n=6`, the function should return `360`, as the divisors of `6` are `1, 2, 3`, and `6`, and their product is `1 × 2 × 3 × 6 = 36`. Then, write a `main` function to test this recursive function.

27. Write a recursive function that takes a positive integer `n` and counts and returns the number of distinct ways n can be expressed as a sum of powers of 2. Each power of 2 can be used multiple times in each combination. The order of terms in the sum is not important, so different permutations of the same set are considered the same. Then, write a `main` function to test this recursive function.

Examples:

- `n = 5`
  - Distinct ways: $5 = 2^2+2^0$, $5 = 2^1+2^1+2^0$
  - Output: `2`
- `n =7`
  - Distinct ways: $7 = 2^2+2^1+2^0$, $7 = 2^2+2^0+2^0+2^0$, $7 = 2^1+2^1+2^1+2^0$
  - Output: `3`

28. Write a recursive function that takes two parameters, `n` and `k`, and determines the number of ways to distribute `n` identical items into `k` distinct bins. Then, write a `main` function to test this recursive function.

29. Write a recursive function that takes one parameter `n`, and calculates and returns the sum of the first `n` odd numbers. For example, for `n = 5`, the sum of the first `5` odd numbers is `25`, since `1+3+5+7+9=25`. Then, write a `main` function to test this recursive function.

30. Given a grid of size `m x n`, write a recursive function that takes two integer parameters `m` and `n`, and counts and returns the number of distinct paths from the top-left corner to the bottom-right corner. You can only move either down or right at any point. Then, write a `main` function to test this recursive function.

    Example:

    For a grid of size `3×3`:

    - The distinct paths are:
      - `Move Right, Right, Down, Down, Down`
      - `Move Right, Down, Right, Down, Down`
      - `Move Right, Down, Down, Right, Down`
      - `Move Down, Right, Right, Down, Down`
      - `Move Down, Right, Down, Right, Down`
      - `Move Down, Down, Right, Right, Down`
      - `Move Down, Down, Right, Down, Right`
      - `Move Down, Down, Down, Right, Right`
    - Output: 6

31. Write a recursive function that takes one parameter, a positive integer, and returns the number of 1's in its binary representation.

    Example:

    - For **5** (binary **101**), the function should return **2**.
    - For **8** (binary **1000**), the function should return **1**.

- For **15** (binary **1111**), the function should return **4**.

32. Write a recursive function that takes a single parameter, a positive integer. The function prints the binary representation of the parameter.
33. Write a recursive function that takes a single parameter, a positive integer, and returns **True (1)** if the number of 1's in its binary representation is odd, and **False (0)** otherwise. For example, for num = 6, function returns False (0) since the binary representation of 6, which is 110, contains even number of 1's.
34. Write a recursive function that takes a single parameter, a positive integer, and returns the sum of the digits at the even positions**.** For this problem, the first digit of the number is considered position 0 and will be included in the sum.
35. A number Y is called the complement of number X if Y = 9 – X. For example, 3 is the complement of 6 since 9 – 6 = 3, and 0 is the complement of 9 since 9 – 0 = 9. Write a recursive function that takes a single parameter, a positive integer. The function prints a number after replacing each digit with its complement. For example, if the input is 1234, the function prints 8765, and if the input is 998, the function prints 001.
36. Write a recursive function that takes a single parameter, a positive integer, and returns True (1) if the sum of the digits is even, and False (0) otherwise. For example, if the input is 879176, the function returns True (1), and if the input is 879076, the function returns False (0).