CS 137

Programming Principles

Chapter 5

Recursion

Victoria Sakhnini

Table of Contents

Recursive Functions 3

- Implementation 3
- Note — Unwinding factorial(6) 4

Classic Examples 4

- Fibonacci Sequence 4
- Number Guessing 5
- Coding 6
- Tower of Hanoi 7
- Coding 8

GCD 9

Tail and Accumulative Recursion 10

- Revisiting Fibonacci 10
- Improvement 1 — Non-Recursive (Iterative) Solution 10
- Improvement 2 — Tail-Recursive Solution 11
- Revisiting Factorial Using Accumulative Recursion 12

Extra Practice Problems 13

Recursive Functions

The C language supports a capability known as a **recursive function**. It can be effectively used to solve problems. It is usually used when a solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem.

A good simple example is the calculation of the factorial of a number. Recall that the factorial of positive integer n (written n!) is the product of the successive integers 1 through n. The factorial of 0 is a special case defined as equal to 1.

```
5! = 5 * 4 * 3 * 2 * 1 = 120
6! = 6 * 5 * 4 * 3 * 2 * 1 = 720
```

Comparing 6! with 5!, you will find that 6! = 6 * 5!.

In general: `n! = n * (n-1)!` when n > 0. Note that n! is defined in terms of the value of (n-1)! — this is called a **recursive definition**. In other words, the solution to n! is expressed as the solution to (n-1)!, which is a subset of the original problem n!.

The definition must have a **base case**, which is the trivial solution. In the case of factorial, the base case is `0! = 1`.

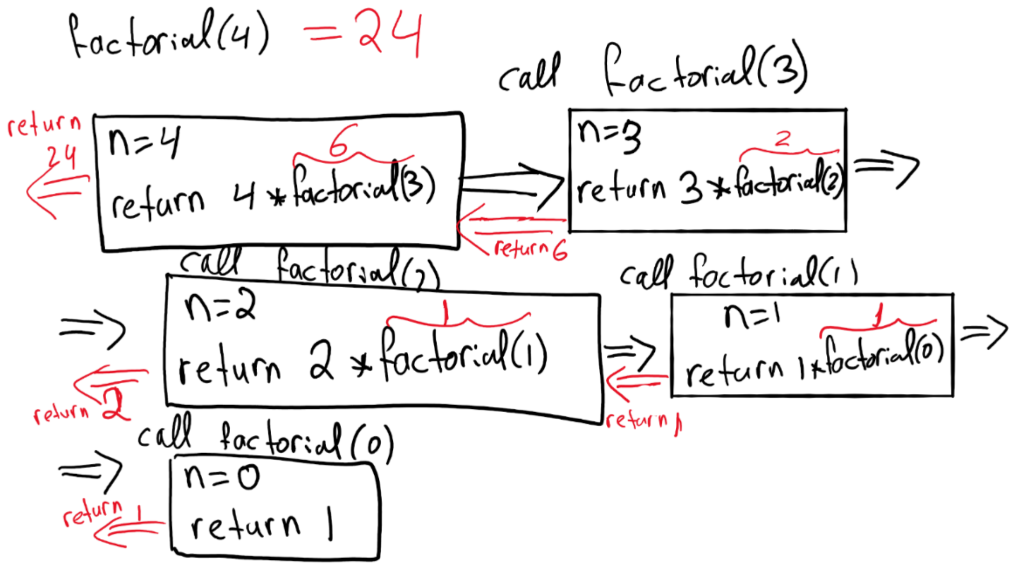
Recursion and loops are equivalent structures. Any task you can achieve with one, you can complete with the other. However, some tasks are naturally recursive, and a recursive solution is often simpler than one using loops. Recursion is most useful when breaking up your task into smaller subtasks.

Implementation

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

int factorial(unsigned int n)
{
    if (n == 0)            // Base case
        return 1;
    else
        return n * factorial(n - 1);
}

int main(void)
{
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(6) == 720);
}
```

Note — Unwinding factorial(6)

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
             = 6 * 120
             = 720
```



Classic Examples

Fibonacci Sequence

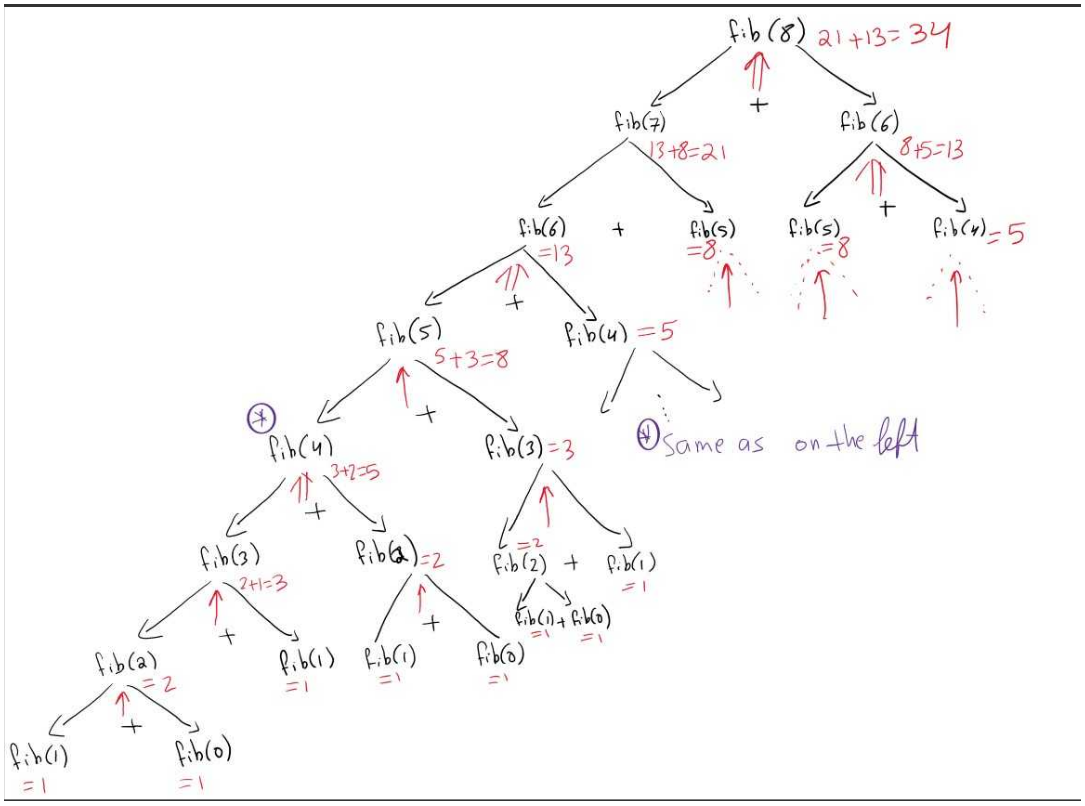
The Fibonacci sequence is a famous mathematical sequence defined recursively:

```
fib(0)  = 1
fib(1)  = 1
fib(n)  = fib(n-1) + fib(n-2)    for n >= 2
```

The sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Coding Fibonacci:

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

int fib(unsigned int n)
{
    if (n <= 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    assert(fib(0) == 1);
    assert(fib(1) == 1);
    assert(fib(2) == 2);
    assert(fib(8) == 34);
}
```



Number Guessing

We want to write a program that guesses the number the user is thinking of quickly! The rules are:

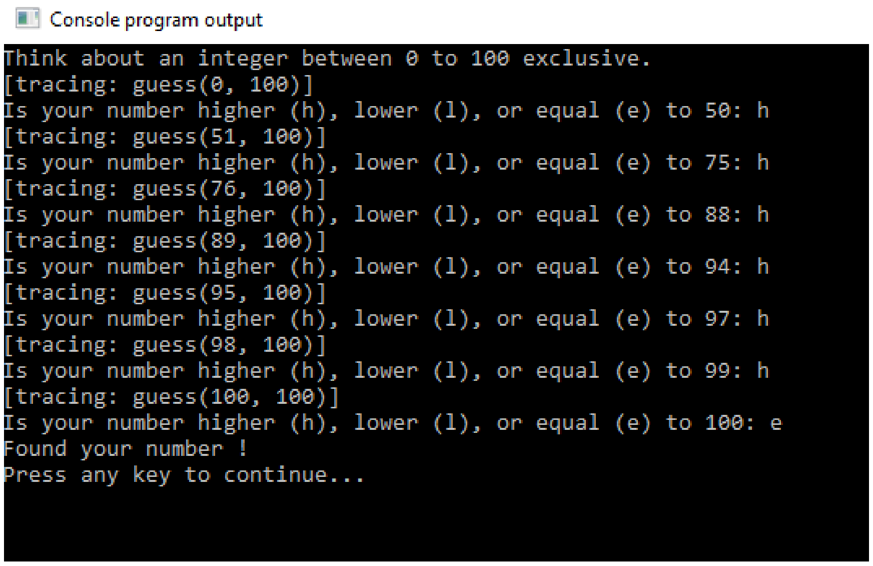
- The user picks a number between a minimum and maximum value (exclusive).
- The program guesses a number, and the user tells the program if the value they are thinking of is higher, lower, or equal to the guess.
- The game ends when the user says equal.

Let's write this function recursively!

Coding

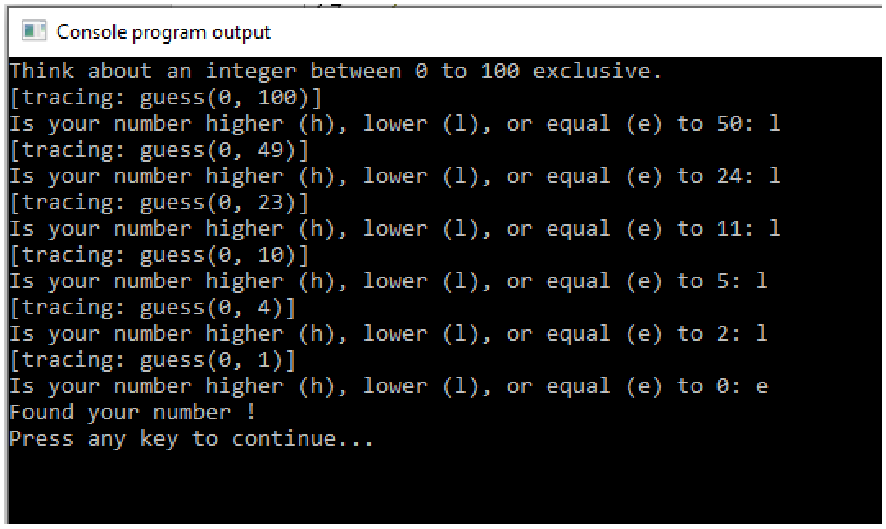
```
#include <stdio.h>

void guess(int min, int max)
{
    // The following printf is added to trace/understand the recursive calls
    printf("[tracing: guess(%d, %d)]\n", min, max);
    if (min > max)
    {
        printf("Your answers are not consistent\n");
        printf("Game is over\n");
        return;
    }
    int g = min + (max - min) / 2;
    printf("Is your number higher (h), lower (l),");
    printf(" or equal (e) to %d: ", g);
    char c;
    scanf(" %c", &c);
    if (c == 'e')
        printf("Found your number !\n");
    if (c == 'h')
        guess(g + 1, max);
    if (c == 'l')
        guess(min, g - 1);
}

int main()
{
    printf("Think about an integer between 0 to 100 exclusive.\n");
    guess(0, 100);
}
```

Adding printf statements to print variable values is a valuable tool for tracing code and understanding how it works!

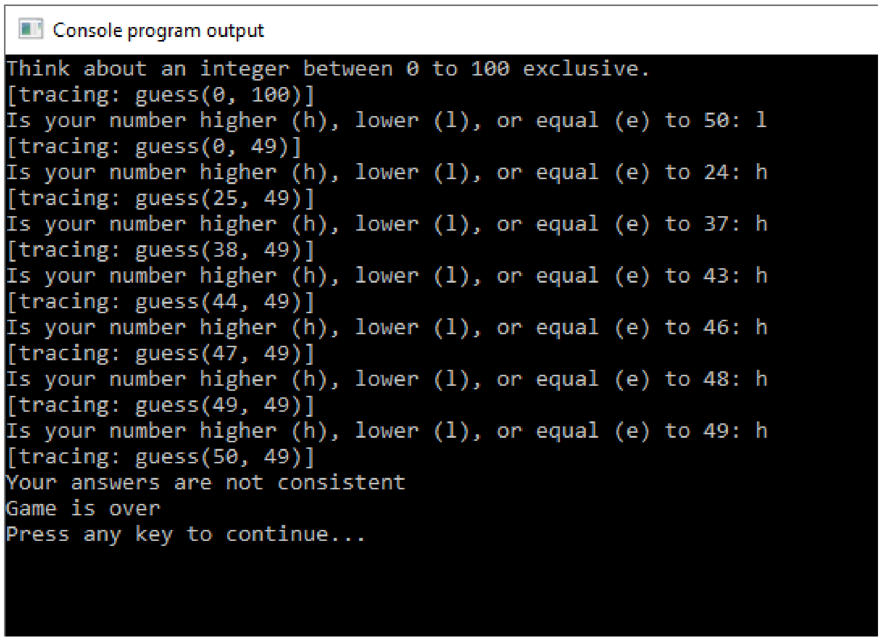


Console program output

```
Think about an integer between 0 to 100 exclusive.
[tracing: guess(0, 100)]
Is your number higher (h), lower (l), or equal (e) to 50: h
[tracing: guess(51, 100)]
Is your number higher (h), lower (l), or equal (e) to 75: h
[tracing: guess(76, 100)]
Is your number higher (h), lower (l), or equal (e) to 88: h
[tracing: guess(89, 100)]
Is your number higher (h), lower (l), or equal (e) to 94: h
[tracing: guess(95, 100)]
Is your number higher (h), lower (l), or equal (e) to 97: h
[tracing: guess(98, 100)]
Is your number higher (h), lower (l), or equal (e) to 99: h
[tracing: guess(100, 100)]
Is your number higher (h), lower (l), or equal (e) to 100: e
Found your number !
Press any key to continue...
```



```
Think about an integer between 0 to 100 exclusive.
[tracing: guess(0, 100)]
Is your number higher (h), lower (l), or equal (e) to 50: l
[tracing: guess(0, 49)]
Is your number higher (h), lower (l), or equal (e) to 24: l
[tracing: guess(0, 23)]
Is your number higher (h), lower (l), or equal (e) to 11: l
[tracing: guess(0, 10)]
Is your number higher (h), lower (l), or equal (e) to 5: l
[tracing: guess(0, 4)]
Is your number higher (h), lower (l), or equal (e) to 2: l
[tracing: guess(0, 1)]
Is your number higher (h), lower (l), or equal (e) to 0: e
Found your number !
Press any key to continue...
```



```
Think about an integer between 0 to 100 exclusive.
[tracing: guess(0, 100)]
Is your number higher (h), lower (l), or equal (e) to 50: l
[tracing: guess(0, 49)]
Is your number higher (h), lower (l), or equal (e) to 24: h
[tracing: guess(25, 49)]
Is your number higher (h), lower (l), or equal (e) to 37: h
[tracing: guess(38, 49)]
Is your number higher (h), lower (l), or equal (e) to 43: h
[tracing: guess(44, 49)]
Is your number higher (h), lower (l), or equal (e) to 46: h
[tracing: guess(47, 49)]
Is your number higher (h), lower (l), or equal (e) to 48: h
[tracing: guess(49, 49)]
Is your number higher (h), lower (l), or equal (e) to 49: h
[tracing: guess(50, 49)]
Your answers are not consistent
Game is over
Press any key to continue...
```

Tower of Hanoi

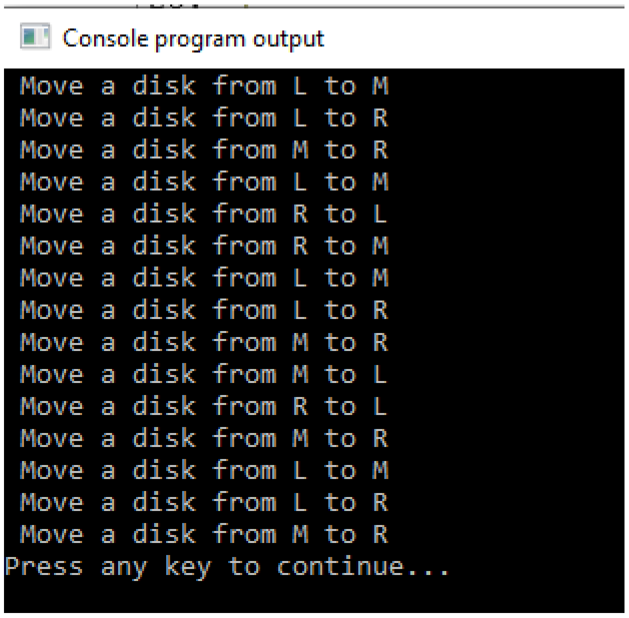
Visit `https://www.mathsisfun.com/games/towerofhanoi.html` to play the game.

The general idea: to move n disks correctly from the source (Left pole) to the destination (Right pole):

- Move n-1 disks correctly from the source pole to the middle pole.
- Move the one remaining disk from the source to the destination.
- Move n-1 disks correctly from the middle pole to the destination.

Coding

```c
#include <stdio.h>

void hanoi(int n, char src, char dst, char sp)
{
    if (n == 1)
    {
        printf(" Move a disk from %c to %c\n", src, dst);
    }
    else
    {
        hanoi(n - 1, src, sp, dst);
        hanoi(1, src, dst, sp);
        hanoi(n - 1, sp, dst, src);
    }
}

int main(void)
{
    // solution for 4 disks
    // L (Left pole), R (Right pole), M (Middle pole)
    hanoi(4, 'L', 'R', 'M');
    return 0;
}
```

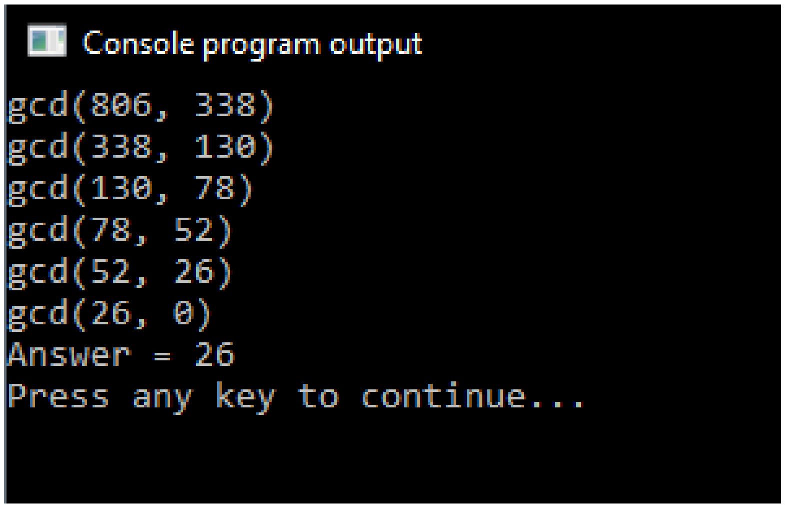

Console program output

```
 Move a disk from L to M
 Move a disk from L to R
 Move a disk from M to R
 Move a disk from L to M
 Move a disk from R to L
 Move a disk from R to M
 Move a disk from L to M
 Move a disk from L to R
 Move a disk from M to R
 Move a disk from M to L
 Move a disk from R to L
 Move a disk from M to R
 Move a disk from L to M
 Move a disk from L to R
 Move a disk from M to R
Press any key to continue...
```

GCD

Below is the recursive version of computing the GCD of two numbers.

```
#include <stdio.h>

int gcd(int a, int b)
{
    if (b == 0)
    {
        printf("gcd(%d, %d)\n", a, b);  // This line is for tracing
        return a;
    }
    else
    {
        printf("gcd(%d, %d)\n", a, b);  // This line is for tracing
        return gcd(b, a % b);
    }
}

int main(void)
{
    printf("Answer = %d\n", gcd(806, 338));
}
```


Console program output

```
gcd(806, 338)
gcd(338, 130)
gcd(130, 78)
gcd(78, 52)
gcd(52, 26)
gcd(26, 0)
Answer = 26
Press any key to continue...
```

Tail and Accumulative Recursion

Revisiting Fibonacci

The Fibonacci definition:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)   for n >= 2
```

Code translated directly from the definition:

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

int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    assert(fib(3) == 2);
    assert(fib(10) == 55);
    // fib(45) is the largest that fits in an integer. It takes some time.
    assert(fib(45) == 1134903170);
    return 0;
}
```

The running time is $O(2^n)$ — very slow.

Improvement 1 — Non-Recursive (Iterative) Solution

By using iterative structures instead of recursion, the runtime reduces to O(n).

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

int fib(int n) {
    if (n == 0)
        return 0;
    int prev = 0, cur = 1;
    for (int i = 1; i < n; i++) {
        int next = prev + cur;
        prev = cur;
        cur = next;
    }
    return cur;
}
```

```
int main(void) {
    assert(fib(3) == 2);
    assert(fib(10) == 55);
    // fib(45) is the largest that fits in an integer.
    assert(fib(45) == 1134903170);
    return 0;
}
```

Trace for fib(10):

```
n = 10, prev = 0, cur = 1
i = 1,  next = 1,  prev = 1,  cur = 1
i = 2,  next = 2,  prev = 1,  cur = 2
i = 3,  next = 3,  prev = 2,  cur = 3
i = 4,  next = 5,  prev = 3,  cur = 5
i = 5,  next = 8,  prev = 5,  cur = 8
i = 6,  next = 13, prev = 8,  cur = 13
i = 7,  next = 21, prev = 13, cur = 21
i = 8,  next = 34, prev = 21, cur = 34
i = 9,  next = 55, prev = 34, cur = 55
return 55
```

Improvement 2 — Tail-Recursive Solution

A recursive function is **tail-recursive** if the recursive call is the last thing executed by the function.

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

int fib_tr(int prev, int cur, int n) {
    if (n == 0)
        return cur;
    return fib_tr(cur, prev + cur, n - 1);
}

int fib(int n)
{
    if (n == 0)
        return 0;
    return fib_tr(0, 1, n - 1);
}

int main(void) {
    assert(fib(5) == 5);
    assert(fib(10) == 55);
    assert(fib(45) == 1134903170);
    return 0;
}
```

To use a tail-recursive solution, a helper function (fib_tr) was created with two extra parameters to accumulate the result and return it in the base case. This is also called accumulative recursion. Usually, defining a tail recursion also requires defining an accumulative one.

Tracing fib(5):

```
prev=0, cur=1, n=4
prev=1, cur=1, n=3
prev=1, cur=2, n=2
prev=2, cur=3, n=1
prev=3, cur=5, n=0  return 5
```

Tracing fib(10):

```
prev=0,  cur=1,  n=9
prev=1,  cur=1,  n=8
prev=1,  cur=2,  n=7
prev=2,  cur=3,  n=6
prev=3,  cur=5,  n=5
prev=5,  cur=8,  n=4
prev=8,  cur=13, n=3
prev=13, cur=21, n=2
prev=21, cur=34, n=1
prev=34, cur=55, n=0  return 55
```

Revisiting Factorial Using Accumulative Recursion

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

int cum_fact(int n, int res) {
    if (n <= 1) return res;
    else return cum_fact(n - 1, res * n);
}

int factorial(int n) {
    assert(n >= 0);
    return cum_fact(n, 1);
}

int main(void) {
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(6) == 720);
}
```

Tracing factorial(6):

```
cum_fact(6, 1)
cum_fact(5, 6)
cum_fact(4, 30)
cum_fact(3, 120)
cum_fact(2, 360)
cum_fact(1, 720)  return 720
```

Extra Practice Problems

Try to solve the problems and test your solutions before checking the provided solutions for some questions.

1.

Write a recursive function power that takes two non-negative integer parameters b and n, and returns b^n.

2.

Write a recursive function geometric_sum that takes two non-negative integer parameters b and n, and returns the sum:
$1 + b + b^2 + b^3 + \dots + b^n$

3.

What is the output of the following program? Trace it manually before running.

```
#include <stdio.h>

void do_it(int n)
{
    printf("%d\n", n);
    if (n == 1)
        return;
    else
        do_it(n - 1);
    printf("%d\n", n);
}

int main(void)
{
    do_it(8);
}
```

4.

Complete the following program. The function must be recursive and must not use * or /.

```
#include <stdio.h>
#include <assert.h>
// pre: a, b > 0
// recursive function, returns a*b without using the operations * or /
int mult(int a, int b) {
    // ADD YOUR CODE HERE
}

int main(void) {
    assert(mult(5, 10) == 50);
    assert(mult(8, 1) == 8);
    return 0;
}
```

5.

Complete the following program. Implement the functions using tail recursion.
'?' matches any single letter.

```
#include <string.h>
#include <stdbool.h>
#include <assert.h>

// '?' represents any letter
bool isPrefix(char *str, char *prefix) {
    // Your code
}

// counts how many times a pattern appears in a text
int countOccurrences(char *text, char *pattern) {
    // Your code
}

int main(void) {
    assert(isPrefix("CS137 is great", "CS137"));
    assert(isPrefix("CS137 is great", "CS137 is great"));
    assert(isPrefix("CS137 is great", "CS?37"));
    assert(!isPrefix("CS137 is great", "great"));
    assert(!isPrefix("CS137 is great", "cs137"));
    assert(!isPrefix("CS137", "CS138"));
    assert(countOccurrences("mama and papa", "?a?a") == 3);
    assert(countOccurrences("a", "ab") == 0);
    assert(countOccurrences("mama and papa", "dad") == 0);
    assert(countOccurrences("xxxxxxxxx", "xx") == 8);
}
```

6.

Implement a recursive function that calculates the total expected discounted reward for a two-state scenario.

Background: The Bellman equation V(s) = R(s) + discount_factor * V(s') gives the value of a state as its immediate reward plus the discounted value of the next state. The two states alternate: from state 0 you always go to state 1, and vice versa.

Implement: double calculateValueRecursive(double reward0, double reward1, double discountFactor, int currentState, int stepsRemaining)

```
int main(void)
{
    assert(0 == calculateValueRecursive(0, 0, 0, 0, 0));
    assert(0 == calculateValueRecursive(0, 0, 0, 1, 0));
    assert(5 == calculateValueRecursive(5, 0, 0, 0, 1));
    assert(10 == calculateValueRecursive(0, 10, 0, 1, 1));
    double epsilon = 0.000001;
    double valueFromState0 = calculateValueRecursive(5, 10, 0.8, 0, 5);
    double valueFromState1 = calculateValueRecursive(5, 10, 0.8, 1, 5);
    assert(23.368000 - epsilon < valueFromState0 && valueFromState0 <
23.368000 + epsilon);
    assert(27.056000 - epsilon < valueFromState1 && valueFromState1 <
27.056000 + epsilon);
    return 0;
}
```

7.

Use only recursion to solve this question. No % / * allowed.

a) int triangle(int n) — returns the nth triangle number.
Assumption: n is a positive integer.
Definition: the nth triangle number is the sum of the first n positive integers.

b) int tetrahedral(int n) — returns the nth tetrahedral number.
Assumption: n is a positive integer.
Definition: the nth tetrahedral number is the sum of the first n triangle numbers.

```
int main(void) {
    assert(triangle(2) == 3);
    assert(triangle(3) == 6);
    assert(triangle(5) == 15);
    assert(tetrahedral(2) == 4);
    assert(tetrahedral(3) == 10);
}
```

8.

Create a recursive function my_sqrt that checks if a given number is a perfect square and returns its square root.
Return -1 for invalid inputs (negative numbers) and for imperfect squares.
Do not use the sqrt function from the math library.

Example: my_sqrt(25) → 5
Example: my_sqrt(-16) → -1
Example: my_sqrt(22) → -1

9.

A memory region (unsigned integer) is rectifiable if at most one bit in its binary representation is set to 1 (a single cosmic-ray bit flip). Write a function that returns true if at most one bit is set to 1, false otherwise.

Example: n = 1 (binary: ...0001) → true
Example: n = 16 (binary: ...10000) → true
Example: n = 3 (binary: ...0011) → false

Constraints: $0 <= n <= 2^{32} - 1$

10.

You are given apples and oranges. Arrange them in a triangle: row 1 has 1 fruit, row 2 has 2 fruits, etc.
All fruits in a row must be the same type, and adjacent rows must have different types.
Return the maximum height of the triangle that can be achieved.

Example 1: apple=2, orange=4 → 3 (○ / ▨ ▨ / ○ ○ ○)
Example 2: apple=2, orange=1 → 2 (○ / ▨ ▨)

Example 3: apple=1, orange=1 → 1
Example 4: apple=10, orange=1 → 2 (🍊 / 🍎🍎)

Constraints: 1 <= apple, orange <= 100

Example 1:

Input: `apple = 2`, `orange = 4`

Output: 3

Explanation:

🍊

🍎 🍎

🍊 🍊 🍊

The only possible arrangement is shown above.

Example 2:

Input: `apple = 2`, `orange = 1`

Output: 2

Explanation:

🍊

🍎 🍎

The only possible arrangement is shown above.

Example 3:

Input: `apple = 1`, `orange = 1`

Output: 1

🍊

Or

🍎

Example 4:

Input: `apple = 10`, `orange = 1`

Output: 2

Explanation:

🍊

🍎 🍎

The only possible arrangement is shown above.

11.

You want to go up n (>= 1) stairs. At any time you walk 1 step up or jump 2 steps up.

Examples:
  2 stairs: 2 ways (1,1 or 2)
  3 stairs: 3 ways (1,1,1 or 2,1 or 1,2)

Write a recursive function that takes n and returns the number of different ways to go up n stairs.