CS 137

Recursion

Fall 2025

Victoria Sakhnini

Table of Contents

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! is calculated as 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! was defined in terms of the value of (n-1)! which is called a recursive definition. In other words, the solution to n! is expressed as the solution to (n-1)! where (n-1)! 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.

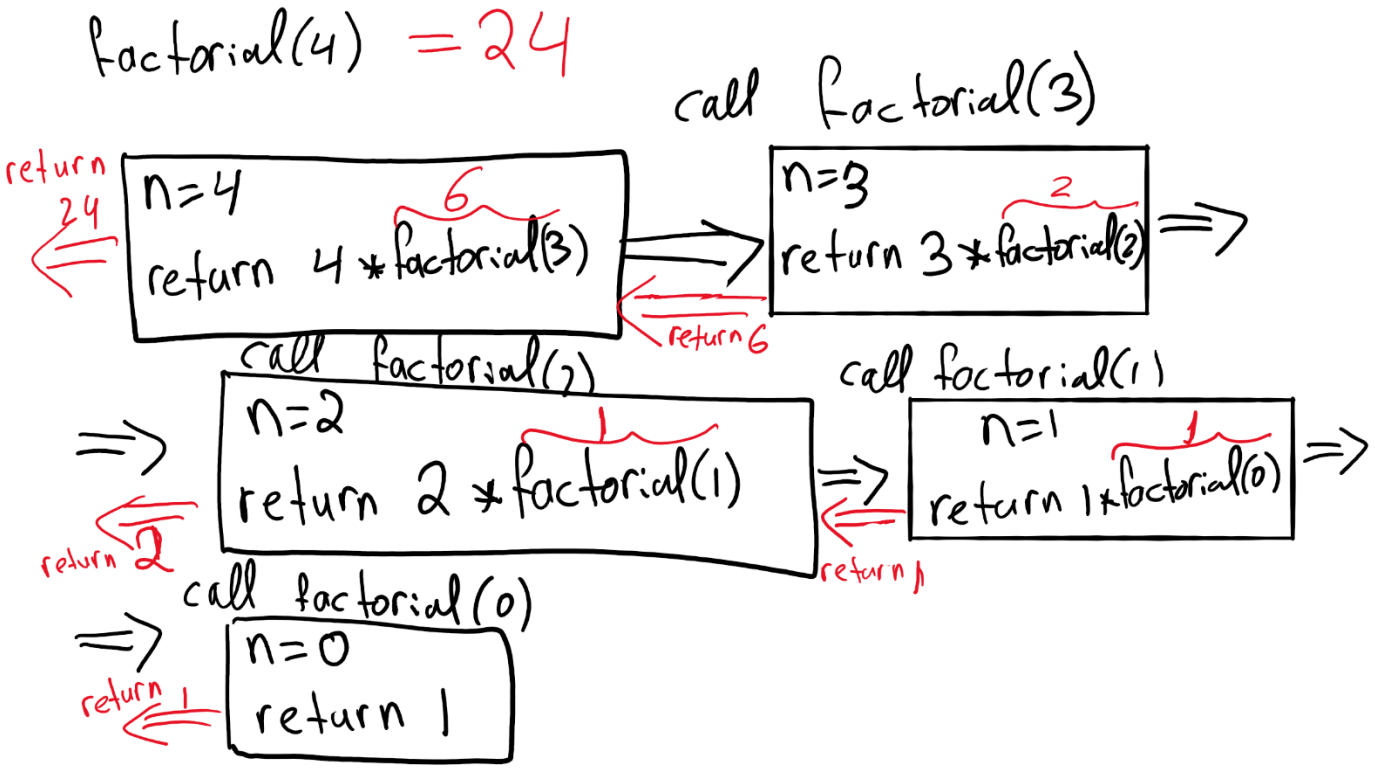
Below, you will find the factorial implementation using recursion.

```
1. #include <stdio.h>
2. #include <assert.h>
3.
4. int factorial(unsigned int n)
5. {
6.          if (n == 0)      // Base case
7.                  return 1;
8.          else
9.                  return n * factorial(n - 1);
10.  }
11.
12.  int main(void)
13.  {
14.          assert(factorial(0) == 1);
15.          assert(factorial(1) == 1);
16.          assert(factorial(6) == 720);
17.  }
```

Note:

```
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
```

```
factorial(4):
```



Classic examples

Fibonacci Sequence

The Fibonacci sequence is a famous mathematical sequence defined recursively.

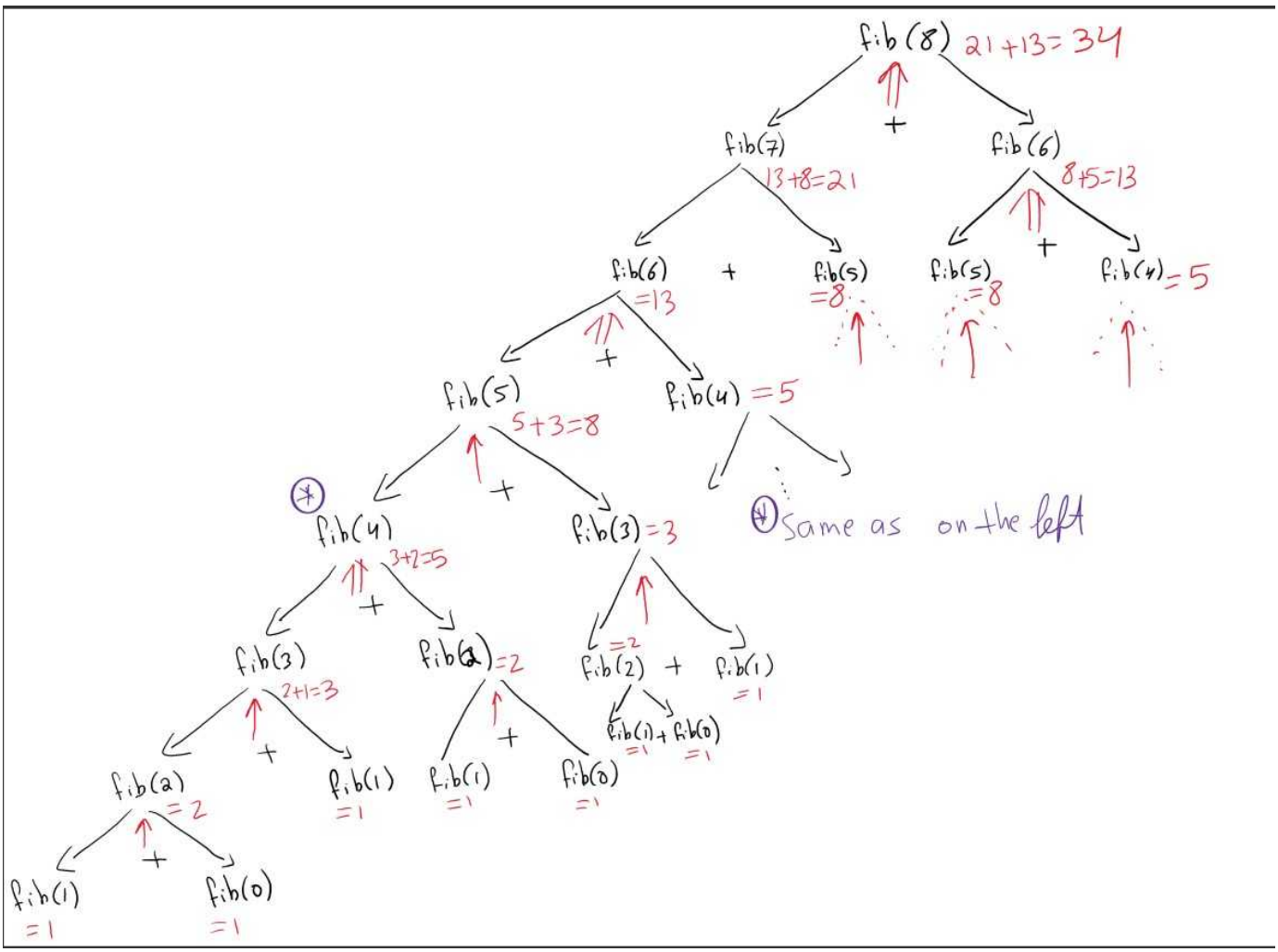
$$f(x) = \begin{cases} 1 & \text{if } x \leq 1 \\ f(n-1) + f(n-2) & \textit{otherwise} \end{cases}$$

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

Coding Fibonacci

```
1. #include <stdio.h>
2. #include <assert.h>
3.
4. int fib(unsigned int n)
5. {
6.          if (n <= 1)
7.                  return 1;
8.          return fib(n - 1) + fib(n - 2);
9. }
10.  int main()
11.  {
12.         assert(fib(0) == 1);
13.         assert(fib(1) == 1);
14.         assert(fib(2) == 2);
15.         assert(fib(8) == 34);
16.  }
```

Tracing:



Number guessing

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

- The user picks a number between a minimum and maximum value (exclusive).
- The program then guesses a number, and the user then tells the program if the value they are thinking of is
  - Higher than that number
  - Lower than that number
  - Equal to that number (Game is over)
- Let's write this function recursively!

Coding:

```c
#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);
}
```

Samples of execution:

Console program output

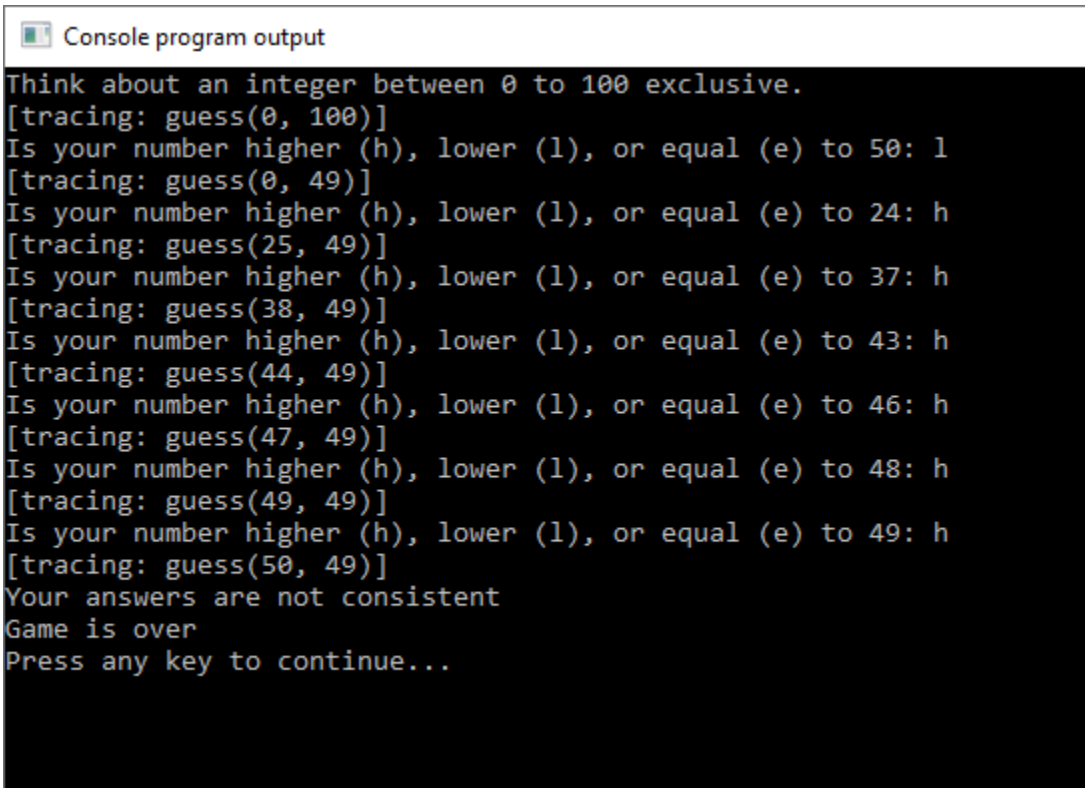
```
Think about an integer between 0 to 100 exclusive.
[tracing: guess(0, 100)]
Is your number higher (h), lower (l), or equal (e) to 50: h
[tracing: guess(51, 100)]
Is your number higher (h), lower (l), or equal (e) to 75: h
[tracing: guess(76, 100)]
Is your number higher (h), lower (l), or equal (e) 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...
```

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: 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...
```

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: l
[tracing: guess(51, 74)]
Is your number higher (h), lower (l), or equal (e) to 62: h
[tracing: guess(63, 74)]
Is your number higher (h), lower (l), or equal (e) to 68: h
[tracing: guess(69, 74)]
Is your number higher (h), lower (l), or equal (e) to 71: l
[tracing: guess(69, 70)]
Is your number higher (h), lower (l), or equal (e) to 69: h
[tracing: guess(70, 70)]
Is your number higher (h), lower (l), or equal (e) to 70: e
Found your number !
Press any key to continue...
```



Adding `printf` to print variables' values is a valuable tool for tracing code and understanding how it works!

Tower of Hanoi

Visit the following link and play the game. Have fun while paying attention to the solution and the steps.

https://www.mathsisfun.com/games/towerofhanoi.html

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

move `n-1` disks correctly (following the rules) from the source pole to the middle pole

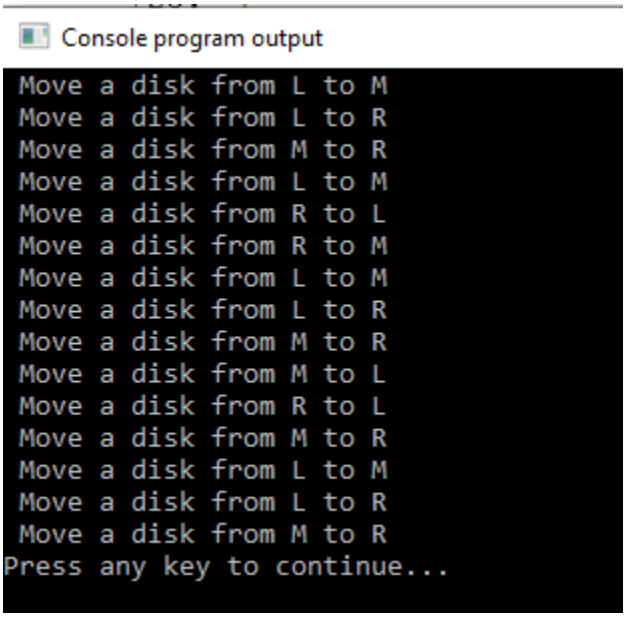
move the one disk left from the source to the destination

move `n-1` disks correctly (following the rules) from the middle pole to the destination

Coding

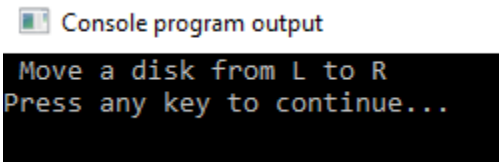
```
1. #include<stdio.h>
2.
3. void hanoi(int n, char src, char dst, char sp)
4. {
5.          if (n == 1)
6.          {
7.                  printf(" Move a disk from %c to %c\n", src, dst);
8.          }
9.          else
10.         {
11.                 hanoi(n - 1, src, sp, dst);
12.                 hanoi(1, src, dst, sp);
13.                 hanoi(n - 1, sp, dst, src);
14.         }
15. }
16.
17. int main(void)
18. {
19.         // solution for 4 disks
20.         // L (Left pole), R (Right pole), M (Middle pole)
21.         hanoi(4, 'L', 'R', 'M');
22.         return 0;
23. }
```

Output

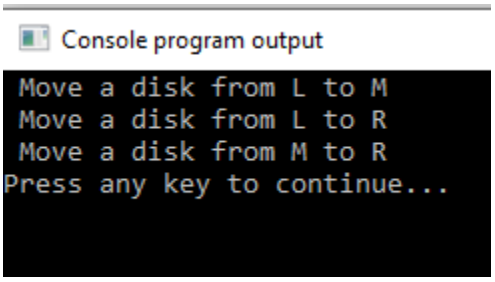


Running the program with

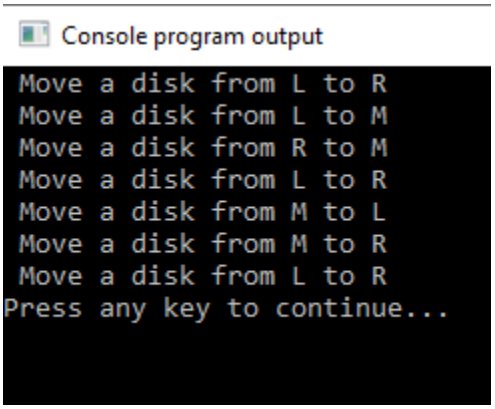
hanoi(1, 'L', 'R', 'M');



hanoi(2, 'L', 'R', 'M');



hanoi(3, 'L', 'R', 'M');

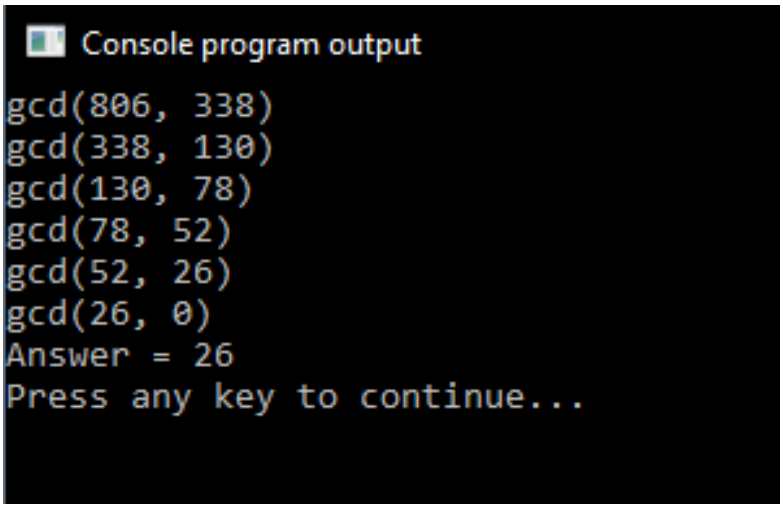


GCD

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

```
1. #include<stdio.h>
2.
3. int gcd(int a, int b)
4. {
5.          if (b == 0)
6.          {
7.                  printf("gcd(%d, %d)\n", a, b); // This line is for tracing
8.                  return a;
9.          }
10.         else
11.         {
12.                 printf("gcd(%d, %d)\n", a, b); // This line is for tracing
13.                 return gcd(b, a % b);
14.         }
15. }
16.
17. int main(void)
18. {
19.         printf("Answer = %d\n", gcd(806, 338));
20. }
```

Output



Tail and Accumulative Recursion

Back to Fibonacci Definition:

$$\mathtt{fib(n)} = \begin{cases} 0 & \text{if } n == 0 \\ 1 & \text{if } n == 1 \\ \mathtt{fib(n-1)} + \mathtt{fib(n-2)} & \text{otherwise} \end{cases}$$

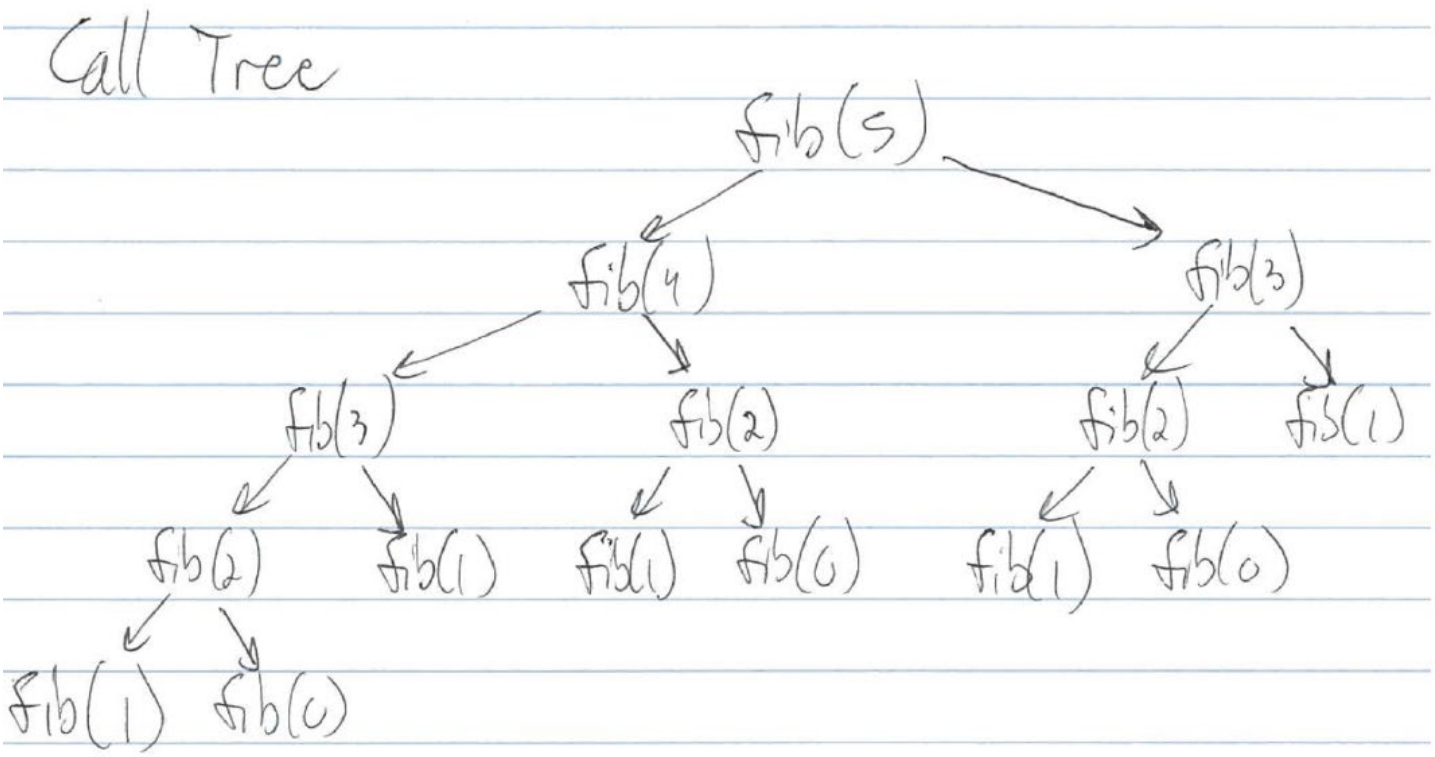
This sequence of numbers became very famous when it was seen in nature, such as in plants, pinecones, sunflowers, Rabbits, Golden spiral, Ration connections and many more examples. If you want to learn more, you should watch this video: https://www.youtube.com/watch?v=nt2OlMAJj6o or this video: https://www.youtube.com/watch?v=greG6_f7Y7Q .

The code:

[translated directly from the definition above]

```
1. #include <stdio.h>
2. #include <assert.h>
3.
4. int fib(int n)
5. {
6.          if (n == 0)
7.                  return 0;
8.          if (n == 1)
9.                  return 1;
10.         return fib(n - 1) + fib(n - 2);
11. }
12.
13. int main()
14. {
15.         assert(fib(3) == 2);
16.         assert(fib(10) == 55);
17.   // f_45 is the largest that fits in an integer. It takes some time to complete.
18.         assert(fib(45) == 1134903170);
19.         return 0;
20. }
```

A trace for n=5:



- The tree is really large, containing $O(2^n)$ many nodes (Actually grows with ϕ^n where ϕ is the golden ratio (1.618))
- Number of `fib(1)` leaves is `fib(n)`
- Summing these is $O($`fib(n)`$)$
- Thus, the code on the previous slide runs in $O($`fib(n)`$)$ which is exponential!

The running time is **$O(2^n)$. Very slow. We will learn about this later.**

Improvement1 (non-recursive solution):

This implementation of Fibonacci shouldn't take this long. After all, by hand, you could undoubtedly compute more than `fib(45)`. We could change the code to no longer call the function (create a new block in the stack) each time; instead, we're using iterative structures. This would reduce the runtime to `O(n)`. Faster than the recursive solution above.

The code:

```
1. #include <stdio.h>
2. #include <assert.h>
3.
4. int fib(int n){
5.          if (n == 0)
6.                  return 0;
7.          int prev = 0, cur = 1;
8.          for (int i = 1; i < n; i++){
9.                  int next = prev + cur;
10.                 prev = cur;
11.                 cur = next;
12.         }
13.         return cur;
14. }
15.
16. int main(void){
17.         assert(fib(3) == 2);
18.         assert(fib(10) == 55);
19.         // f_45 is the largest that fits in an integer.
20.         assert(fib(45) == 1134903170);
21.         return 0;
22. }
```

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
```

Improvement2 (Tail-recursive solution):

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

```
1. #include <stdio.h>
2. #include <assert.h>
3. int fib_tr(int prev, int cur, int n){
4.          if (n == 0)
5.                  return cur;
6.          return fib_tr(cur, prev + cur, n - 1);
7. }
8. int fib(int n)
9. {
10.         if (n == 0)
11.                 return 0;
12.         return fib_tr(0, 1, n - 1);
13.  }
14.  int main(void){
15.         assert(fib(5) == 5);
16.         assert(fib(10) == 55);
17.         assert(fib(45) == 1134903170);
18.         return 0;
19.  }
20.
```

Note: To use a tail-recursive solution, we created a helper function (`fib_tr`) and added two parameters to accumulate the result and return it in the base case. This is also called accumulative recursion.
Usually, to define a tail recursion, you might also need to define 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 function using accumulative (and tail) recursion:

```c
#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)`:

`n = 0, res = 1`

`n = 1, res = 1`

`n = 6, res = 1`

`n = 5, res = 6`

`n = 4, res = 30`

`n = 3, res = 120`

`n = 2, res = 360`

`n = 1, res`

Extra Practice Problems

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

1[i]) Write a recursive function `power` that takes two non-negative integer parameters, `b`, and `n`, and returns b^n.

2[ii]) Write a recursive function `geometric_sum` that takes two non-negative integer parameters, `b`, and `n`, and returns the sum $1 + b + b^2 + b^3 + \ldots + b^n$.

3[iii]) What is the output of the following program?

```
1. #include <stdio.h>
2.
3. void do_it(int n)
4. {
5.          printf("%d\n", n);
6.          if (n == 1)
7.                  return;
8.          else
9.                  do_it(n - 1);
10.         printf("%d\n", n);
11. }
12.
13. int main(void)
14. {
15.         do_it(8);
16. }
```

4) Complete the following program:

```
1. #include <stdio.h>
2. #include <assert.h>
3. // pre: a,b>0
4. // recursive function, returns a*b without using the operations * or /
5. int mult(int a, int b){
6.   // ADD YOUR CODE HERE
7. }
8.
9. int main(void) {
10.         assert(mult(5,10)==50);
11.         assert(mult(8,1)==8);
12.
13.  return 0;
14. }
```

5) Complete the following program. Implement the functions using tail recursion

```
#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) Develop a C program, `bellman_simple.c`, to demonstrate the application of the Bellman equation in a recursive context. The program should calculate the total expected discounted reward for a simple state-transition scenario.

Background:

The Bellman equation is a cornerstone in dynamic programming and reinforcement learning. It provides a robust framework for making optimal decisions in multi-stage problems. You will explore the Bellman equation through a recursive approach in this question.

Problem Description:

Consider a scenario with two states labeled 0 and 1. An agent transitions between these states and receives a specific reward at each state. At every step, the agent moves to the other state. The objective is to calculate the total expected discounted reward, a.k.a, the value of a given state, over a series of state transitions, starting from a given state.

The Bellman equation in this simplified scenario is:

V(s) = R(s) + discount_factor * V(s')

where V(s) is the value of state s, R(s) is the reward for state s.

Implement a recursive function, calculateValueRecursive, which calculates the total expected discounted reward from a given state over a fixed number of steps.

The function `double calculateValueRecursive(double reward0, double reward1, double discountFactor, int currentState, int stepsRemaining);` takes the following parameters:

`double reward0`: The reward received when in state 0.

`double reward1`: The reward received when in state 1.

`double discountFactor`: The discount factor (between 0 and 1) used in the calculation.

`int currentState`: The current state (0 or 1).

`int stepsRemaining`: The number of steps to be considered in the calculation.

The function should return the calculated value as a `double`.

Example of testing:

```
 1. int main(void)
 2. {
 3.     assert(0 == calculateValueRecursive(0, 0, 0, 0, 0));
 4.     assert(0 == calculateValueRecursive(0, 0, 0, 1, 0));
 5.     assert(5 == calculateValueRecursive(5, 0, 0, 0, 1));
 6.     assert(10 == calculateValueRecursive(0, 10, 0, 1, 1));
 7.     double epsilon = 0.000001;
 8.     double valueFromState0 = calculateValueRecursive(5, 10, 0.8, 0, 5);
 9.     double valueFromState1 = calculateValueRecursive(5, 10, 0.8, 1, 5);
10.     assert(23.368000 - epsilon < valueFromState0 && valueFromState0 < 23.368000 + epsilon);
11.     assert(27.056000 - epsilon < valueFromState1 && valueFromState1 < 27.056000 + epsilon);
12.     return 0;
13. }
14.
15.
```

7) Create a C program `simplex.c` that contains the following: (Use only recursion to solve this question)

a) `int triangle(int n);` The function returns the `n`th triangle number.

Assumptions: `n` is a positive integer.

Restrictions: You must NOT use any of `% / *` in your solution.

Definition: The `n`th triangle number is equal to the sum of the first `n` positive integers.

b) `int tetrahedral(int n);` The function returns `n`th tetrahedral number.

Assumptions: `n` is a positive integer.

Restrictions: You must NOT use any of `% / *` in your solution.

Definition: The `n`th tetrahedral number is equal to the sum of the first `n` triangle numbers.

The following code will help you with testing

```
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 the square root of that number without using the square root function from the `math` library. It should return -1 for invalid inputs and for imperfect squares.

Example input: 25

output: 5

Example input: -16

output: -1

Example input: 22

output: -1

9) As a NASA engineer inspecting the memory of a spacecraft, you are tasked with checking if a particular memory region (represented as an **unsigned integer**) has been affected by a cosmic ray. Typically, the memory should contain all zeros, but a cosmic ray might cause a single bit to flip to 1. The spacecraft's system can rectify this single-bit flip but cannot handle cases where more than one bit has flipped to 1.

Write a function that determines if the memory region (represented as an **unsigned integer**) can still be rectified, i.e., it has at most one bit set to 1.

Problem Restatement:

Given an **unsigned integer** n, return true if at most one bit in its binary representation is set to 1 (indicating the memory is still rectifiable). Otherwise, return false.

Examples:

Example 1:

Input: n = 1
Output: true
Explanation: The binary representation of 1 is 000...0001, which has only one bit set to 1.

Example 2:

Input: n = 16
Output: true
Explanation: The binary representation of 16 is 000...10000, which has only one bit set to 1.

Example 3:

Input: n = 3
Output: false
Explanation: The binary representation of 3 is 000...0011, which has two bits set to 1.

Constraints:

- 0 <= n <= 2^32 - 1 (for unsigned integers)

The task is to verify if the memory region can be rectified, meaning there is, at most, one bit set to 1 in the binary representation of n.

10) You are given two integers, `apple`, and `orange`, representing the count of apples and oranges. You have to arrange these fruits to form a triangle such that the 1st row will have 1 fruit, the 2nd row will have 2 fruits, the 3rd row will have 3 fruits, and so on.

All the fruits in a particular row should be of the same type, and adjacent rows should have different types of fruits.

Return the maximum height of the triangle that can be achieved.

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.

Constraints: 1 <= apple, orange <= 100

11) Assume you want to go upstairs of `n (>=1)` steps. You walk 1 step up or jump 2 steps up at any time.

For example, two different ways exist to go up two stairs (1,1, or 2).

There are three different ways to go up three stairs (1,1,1 or 2,1 or 1,2).

Write a recursive function that takes `n` and returns how many different ways you can go up `n` stairs `(n>=1)`.

Answers

---

i

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

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

int main(void)
{
    // testing
    assert(power(0, 100) == 0);
    assert(power(123, 1) == 123);
    assert(power(2, 4) == 16);
    assert(power(5, 4) == 625);
}
```

ii

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

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

int geometric_sum(int b, int n)
{
    if (n == 0)
        return 1;
    if (b == 0)
        return 1;
    return geometric_sum(b, n - 1) + power(b, n);
}

int main(void)
{
    // testing
    assert(geometric_sum(0, 100) == 1);
    assert(geometric_sum(123, 0) == 1);
    assert(geometric_sum(200, 1) == 201);
    assert(geometric_sum(10, 4) == 11111);
}
```

iii

```
Console program output
8
7
6
5
4
3
2
1
2
3
4
5
6
7
8
Press any key to continue...
```

Explanation: I did a trace for `do_it(5)` and not for `do_it(8)` 😊 (the numbers in red are the ordering of printing)