

Programming with C++

COMP2011: Function II — Recursion

Cecia Chan
Brian Mak
Dimitris Papadopoulos
Pedro Sander
Charles Zhang

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China



Part I

Think Recursively!
“God Help Those Who Help
Themselves.”

Example: Tower of Hanoi Game



- It consists of 3 pegs, and a stack of discs of different sizes.
- It starts with all discs stacked up on one peg with smaller discs sitting on top of bigger discs.
- The goal is to move the entire stack of discs to another peg, making use of the remaining peg.
- Rules:
 - only one disc may be moved at a time
 - no disc may be placed on top of a smaller disc

Example: Factorial Function

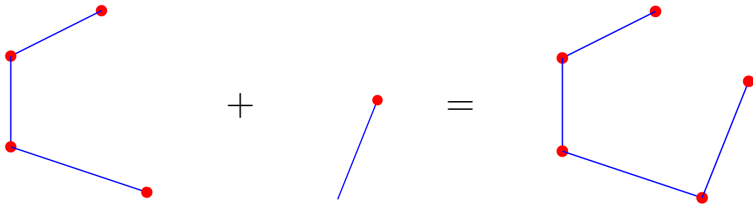
Definition of the **factorial function** you learn in high school:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots 2 \times 1$$

Recursive Definition of Factorial Function

- $0! = 1$
- $n! = n \times (n - 1)! \quad \text{if } n > 0$

Example: Polyline



A **polyline** with n line segments is equivalent to adding another line segment to a **polyline** with $n - 1$ line segments.

Example: Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

Fibonacci (1202) investigated how fast rabbits could breed:

- A **newly-born** pair of rabbits, one male, one female, are put in a field.
- Rabbits **mate at the age of one month** so that at the **end of its 2nd month**, a female can **produce another pair** of rabbits.
- Suppose that our rabbits **never die**.
- Suppose the female always produces **one new pair** (one male, one female) **every month** from the 2nd month on.
- **How many pairs will there be in one year?**

Question : What is special with the above numbers?

Answer : Except for the first 2 numbers, each number is the **sum of the last 2 numbers** in the sequence.

Thinking Recursively

- In some problems, it may be natural to **define the problem in terms of the problem itself!**
- **Recursion** is useful for problems that can be represented by a **simpler version** of the same problem.
 - ❶ To solve the **Tower of Hanoi** game that uses n discs, one first solves the **tower of hanoi** game that uses $n - 1$ discs.
 - ❷ To find the value of $n!$, first find the value of $(n - 1)!$ and then multiply the result with n .
 - ❸ To draw a **polyline** of n segments, first draw a **polyline** of $n - 1$ segments, and then add the last line segment.
 - ❹ To find the n th **Fibonacci number**, first find the $(n - 1)$ th and $(n - 2)$ th **Fibonacci numbers**, and then add them up.

Part II

Programming Recursion in C++

Implement Recursion by Recursive Function

- In programming, **recursion** means that a function calls **itself**!
- Although it looks strange in the beginning, solving a programming task by **recursion** renders the program
 - easier to **write**
 - easier to **read** (understand)
 - **shorter** (in codes).

Implement a Recursive Solution

- 1 **Decompose** the **problem** into **sub-problems** — which are smaller examples of the same problem — plus some **additional work** that “**glues**” the solutions of the sub-problems together.
- 2 The **smallest sub-problem** has a **non-recursive** solution.

Example: Factorial Function

```
int factorial(int n)      /* factorial.cpp */
{
    if (n < 0)             // Error checking
        return -1;
    else if (n == 0)       // Base case; ending case too!
        return 1;
    else                   // Recursive case
        return n * factorial(n-1);
}
```

Or, equivalently,

```
int factorial(int n)      /* factorial2.cpp */
{
    if (n < 0)             // Error checking
        return -1;
    if (n == 0)            // Base case; ending case too!
        return 1;
    return n * factorial(n-1); // Recursive case
}
```

How the Recursive Factorial Function Works?

factorial(3) :

3 < 0 false

3 == 0 false

3 * factorial(2)

factorial(2) :

2 < 0 false

2 == 0 false

2 * factorial(1)

factorial(1) :

1 < 0 false

1 == 0 false

1 * factorial(0)

factorial(0) :

0 < 0 false

0 == 0 true

return 1

return 1*1 = 1

return 2*1 = 2

return 3*2 = 6

Factorial Function: Recursive vs. Non-Recursive

```
int factorial(int n)    /* factorial.cpp */
{
    if (n < 0)          // Error checking
        return -1;
    else if (n == 0)    // Base case; ending case too!
        return 1;
    else                // Recursive case
        return n * factorial(n-1);
}
```

```
int factorial(int n)    /* non-recursive-factorial.cpp */
{
    int result = 1;      // Default value for n = 0 or 1
    if (n < 0)           // Error checking
        return -1;

    for (int j = 2; j <= n; j++) // When n >= 2
        result *= j;
    return result;
}
```

Infinite Loop!

- When we work with **loops**, we have to be careful that the **ending condition** will be **met eventually**.
- Otherwise, we will get **infinite loop!**

```
for (int j = 1; j != 10; j += 2)
{
    // Ending condition never met!
    cout << j << endl;
    ...
}
```

Infinite Recursion!

- Just like infinite loops, we have to be careful that a recursion will **eventually end up** with a **non-recursive base case**.
- Otherwise, we will get **infinite recursion!**

```
int factorial(int n)
{
    // Forget the base case, which is the ending case too!
    return n * factorial(n-1);
}
```

```
int factorial(int n)
{
    // Forget checking if n < 0
    if (n == 0)
        return 1;

    // Infinite recursion for negative n
    return n * factorial(n-1);
}
```

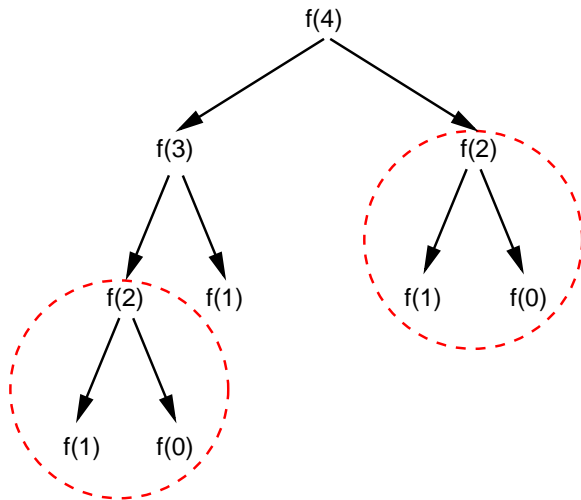
Example: Fibonacci Function as a Recursion

```
int fibonacci(int n)    /* File: fibonacci.cpp */
{
    if (n == 0)          // Base case #1
        return 0;

    if (n == 1)          // Base case #2
        return 1;

    return fibonacci(n-1) + fibonacci(n-2);
}
```

Inefficiency of Recursive Fibonacci Function



repeated computations

Example: Non-Recursive Fibonacci Function

```
int fibonacci(int n)      /* non-recursive-fibonacci.cpp */
{
    int fn;               // keep track of f(n)
    int fn_1 = 1;         // keep track of f(n-1)
    int fn_2 = 0;         // keep track of f(n-2)

    if (n == 0) return 0; // Base case #1
    if (n == 1) return 1; // Base case #2

    for (int j = 2; j <= n; j++)
    {
        fn = fn_1 + fn_2; // f(n) = f(n-1) + f(n-2)

        // Prepare for the calculation of the next fibonacci number
        fn_2 = fn_1;      // f(n-2) = f(n-1)
        fn_1 = fn;        // f(n-1) = f(n)
    }

    return fn;
}
```

Example: Counting Zeros in an Integer

- **Example:** for the integer 120809, there are 2 zeros.
- **Basic idea:**
 - Break down the number into **quotient** and **remainder**.
 - Count the number of zeros in **quotient** and **remainder**.

```
int num_zeros(int n)    /* File: num-zeros.cpp */
{
    if (n == 0)          // Base case #1
        return 1;

    else if (n < 10 && n > 0) // Base case #2
        return 0;

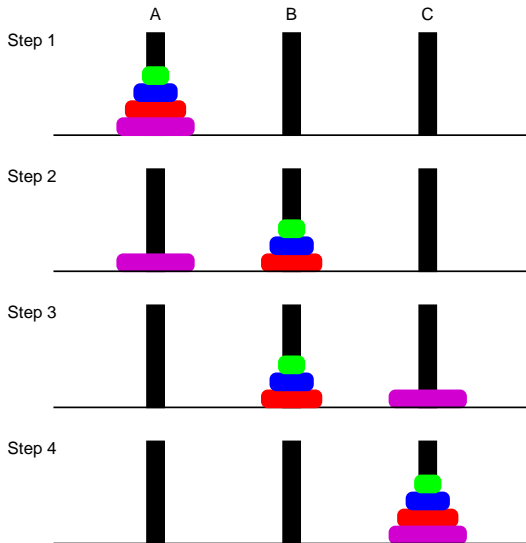
    else
        return num_zeros(n/10) + num_zeros(n%10);
}
```

Example: Factoring

- **Goal:** find how many times **factor** m appears in the **integer** n .
- **Example:** if $n = 48$ and $m = 4$, since $48 = 4 \times 4 \times 3$, the answer is 2.
- **Basic idea:**
 - Divide n by m **until** the remainder is non-zero.
 - **Increment** the count by 1 for every successful division.

```
int num_factors(int n, int m) /* File: factor.cpp */
{
    if (n % m != 0)          // Base case
        return 0;
    else
        return 1 + num_factors(n/m, m);
}
```

Recursive Solution of Tower of Hanoi



Example: Recursive Solution of Tower of Hanoi

```
#include <iostream>      /* File: toh.cpp */
using namespace std;

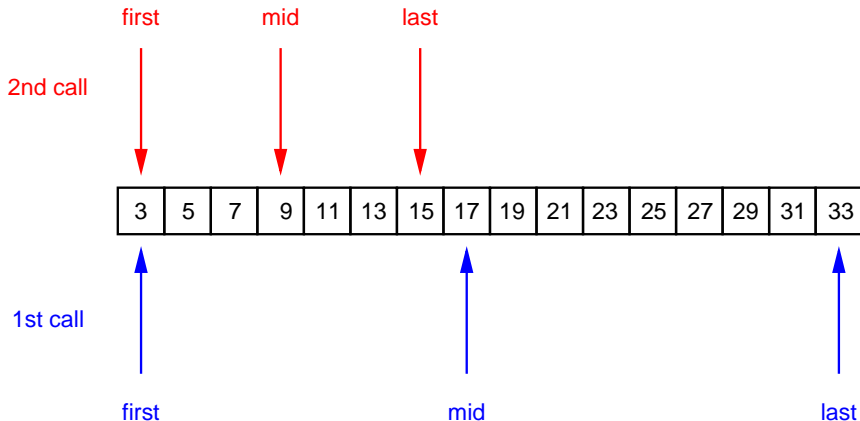
void tower_of_hanoi(int num_discs, char pegA, char pegB, char pegC)
{
    if (num_discs == 0) // Base case
        return;

    tower_of_hanoi(num_discs-1, pegA, pegC, pegB);

    cout << "move disc " << num_discs
         << " from peg " << pegA << " to peg " << pegC << endl;

    tower_of_hanoi(num_discs-1, pegB, pegA, pegC);
}
```

Binary Search



binary search for the value 9

Example: Recursive Solution of Binary Search

```
const int NOT_FOUND = -1;      /* File: bsearch.cpp */
int bsearch(const int data[ ], // sorted in ascending order
            int first,         // lower bound index
            int last,          // upper bound index
            int value)         // value to search
{
    if (last < first)           // Base case #1
        return NOT_FOUND;

    int mid = (first + last)/2;

    if (data[mid] == value)     // Base case #2
        return mid;

    else if (data[mid] > value) // Search the lower half
        return bsearch(data, first, mid-1, value);

    else                        // Search the upper half
        return bsearch(data, mid+1, last, value);
}
```

Example: Non-Recursive Solution of Binary Search

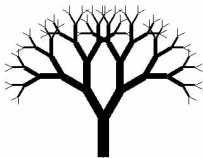
```
const int NOT_FOUND = -1;      /* File: non-recursive-bsearch.cpp */
int bsearch(const int data[ ], // sorted in ascending order
            int size,          // number of data in the array
            int value)         // value to search
{
    int first = 0;
    int last = size - 1;

    while (first <= last)
    {
        int mid = (first + last)/2;
        if (data[mid] == value)
            return mid;        // Value found!
        else if (data[mid] > value)
            last = mid - 1;    // Set up for searching the lower half
        else
            first = mid + 1;   // Set up for searching the upper half
    }

    return NOT_FOUND;
}
```


Recursion is Natural

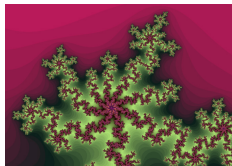
- Many natural phenomena are **recursion**: a smaller part of oneself is **embedded** in itself!



(a) tree



(b) infinite mirror
images



(c) fractal

- As a result, it is usually easier for a programmer to write a solution using recursion \Rightarrow **greater productivity**.

Disadvantages of Recursion

- The greater programming productivity is achieved at the expenses of the more computing resources. To run recursion, it usually requires
 - more memory
 - more computational time
- The reason is that whenever a function is called, the computer
 - has to memorize its **current state**, and **passes control** from the **caller** to the **callee**.
 - sets up a new data structure (you may think of it as a scratch paper for rough work) called **activation record** which contains information such as
 - **where** the **caller** stops
 - what **actual parameters** are passed to the **callee**
 - new **local variables** created by the **callee** function
 - the **return value** of the function at the end
 - removes the **activation record** of the **callee** when it finishes.
 - passes control back to the **caller**.