

Tutorial 3

MATH10017, Brendan and Song

27-28 Oct 2022, 03-10-2024

Contents

1	Recursion	1
1.1	Factorial	2
1.2	Fibonacci numbers	3
2	Replacing recursion with loops	4
2.1	Factorial	4
2.2	Fibonacci	4
3	Exercises	5
3.1	Pascal's triangle	5

Instructions. Implement all of the sample code and do the exercises.

- Make sure your code compiles.
- Don't copy and paste from the .pdf file, rewrite the code.
- Try to complete the first two sections and one problem from Section 3. Do more if you can.

1 Recursion

A recursive function is a function that calls itself in its definition.

1.1 Factorial

An example of a recursive function is *factorial*: for a non-negative integer n , we define $0! = 1$ and $n! = n \cdot (n - 1)!$ if $n > 0$. Thus:

$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 4 \cdot 3 \cdot 2 \cdot 1.$$

We can convert this into C code as follows:

```
int factorial(int n)
{
    int fact = 1;
    if (n > 0){
        fact = n * factorial(n - 1);
    }
    return fact;
}
```

Exercise 1. Create a file called "factorial.c" and put the previous code in it. Write a loop in `main()` to print the first 10 values of $n!$.

So you can see another way of writing recursive code, here is an alternate version of the code using multiple return statements:

```
int factorial(int n)
{
    if (n == 0){
        return 1;
    }
    else
        return n * factorial(n - 1);
}
```

If we call `factorial(3)`, it terminates by returning `3*factorial(2)`. But before control is returned to the caller, the expression `3*factorial(2)` must be evaluated. Similarly, before `factorial(2)` can return control to `factorial(3)`, it must wait for `factorial(1)` to return control. So we have a growing stack of functions that shrinks once we get to $n = 0$:

```
factorial(3)
3 * factorial(2)
3 * (2 * factorial(1))
3 * (2 * (1 * factorial(0)))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * (2)
6
```

1.2 Fibonacci numbers

The n th Fibonacci number is defined recursively as follows:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1.$$

Exercise 2. Create a file called `fibonacci.c`.

- Write a function that computes the n th Fibonacci number using recursion.
- Write a loop in `main()` to print the first 10 Fibonacci numbers.

```
int fibonacci(int n)
{
    int fibn;
    if (n == 0){
        fibn = 0;
    }
    else if (n == 1){
        fibn = 1;
    }
    else {
        //fibn = YOUR CODE HERE
    }
    return fibn;
}
```

Again, here is a version using multiple return statements:

```
int fibonacci(int n)
{
    if (n == 0){
        return 0;
    }
    else if (n == 1){
        return 1;
    }
    else {
        //return YOUR CODE HERE
    }
}
```

2 Replacing recursion with loops

Sometimes, it is possible to replace a recursive algorithm with a loop. This can reduce memory usage and the number of function calls.

2.1 Factorial

The code for computing $n!$ with a loop is similar to the code for computing a sum:

```
int fact_loop(int n)
{
    result = 1;
    for(int i = 1; i <= n; i++){
        result = result * i;
    }
    return result;
}
```

Exercise 3. Enter the code above in "factorial.c" and check that it computes the same values as the recursive version.

2.2 Fibonnaci

To compute a Fibonacci number, we need to know the previous two numbers. Storing "state" values often allows you to solve problems with loops; we will see this when we find maximum and minimum values.

```
// returns the n-th Fibonacci number
int fib_loop(int n)
{
    int a = 0, int b = 1;
    int temp;

    for(int i = 0; i < n; i++){
        temp = a;
        a = b;
        b = temp + b;
    }
    return a;
}
```

Exercise 4. Put the above code in "fibonacci.c". Check that it computes the same values as the recursive version.

What happens if we write

```
a = b;  
b = a + b;
```

instead of using the temporary variable?

(Note, some languages like Python let you do this without a temporary variable, e.g. `a, b = b, a+b`.)

3 Exercises

3.1 Pascal's triangle

The numbers below are rows 0 to 4 of *Pascal's triangle*.

```
    1  
   1 1  
  1 2 1  
 1 3 3 1  
1 4 6 4 1
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.

Let n denote the row of the triangle and let $0 \leq k \leq n$ denote the position in row n . (Note: n starts at 0.)

Exercise 5. Create a file called "pascal.c"

- Write a recursive function of to compute the k th term of the n th row of Pascal's triangle.
- Use a double loop to print the first 7 rows of Pascal's triangle, each row on a different line, with the numbers separated by spaces.
- Bonus: make a function that prints out the first N rows of Pascal's triangle, and make it look like a triangle. (Hint use `%3d` to print all integers as if they had at least three digits and adjust the spaces. You'll need a loop to get the blank space in front of the 1's to look right.)