Chapter 5

Recursion

5.1 Motivation

When we say that "something" is **recursive**, we mean that that "something" is described or defined in such a way such that it *refers to itself within the definition.* Something that is recursive is said to be *self–similar* or *self–referential.*

Recursion occurs in nature, in languages, in (the visual) art, in mathematics and in programming. Let us have first a visual appreciation of recursion before we delve on the programming aspect. Please see the images in the following websites.

```
http://www.its.caltech.edu/~atomic/snowcrystals/photos/photos.htm
http://webecoist.com/2008/09/07/17-amazing-examples-of-fractals-in-nature/
http://www.flickr.com/photos/gadl/sets/72157594316295785/
http://profron.net/pictures/sluggo%20recursive.jpg
http://math-art.net/2008/02/14/recursive-cartoon/
```

The first two websites feature recursion in nature observable in patterns appearing on snowflakes, seashells and Romanescu brocolli. The images in the third site feature photographs which were digitally altered such that they appear to be recursive. Note that the recursion is patterned after the seashell shape seen in the second website. The last two websites feature cartoon showing a character doing an action which is repeated several times within the same cartoon.

Recursion occurs in languages as well, for example, a story, within a story within a story... The acronym "GNU" is actually a recursive. When expanded, it stands for "GNU is Not Unix". Find out what is the unofficial expansion of the abbreviation "PNG". PNG is an image file format.

5.2 Definition of Recursion in Programming

According to `http://www.itl.nist.gov/div897/sqg/dads/HTML/recursion.html`, the NIST definition for recursion is as follows:

> Recursion is an algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

Why do we need (to learn) recursion? One reason is that the problem itself is recursive. It is but natural to give a recursive solution (example: Quicksort and binary tree traversals). While a non–recursive solution may exist, it may be difficult to formulate, implement and understand. Another reason is that there are languages that do not have looping mechanisms such as LISP. In such languages, the predominant technique for computing through sequences and lists is recursion.

5.3 Bad Example – Infinite Recursion

We deliberately show first a bad example of a recursive function definition. It is bad because, in theory, the recursive function calls will never terminate resulting into an infinite program. In real life, the program will crash and terminate abruptly because it will eventually run out of memory. An infinite recursion is the counterpart of an infinite loop.[1]

Listing 5.1: Bad example of recursion (infinite recursion)

```c
#include <stdio.h>

void Bad()
{
   printf("Bad example due to infinite recursion.\n");
   Bad();  /* recursive call here */
}

int main()
{
   Bad();
   return 0;
}
```

[1]Actually, an infinite recursion is worse than an infinite loop because it eats up memory every time does a recursive call resulting into a stack overflow.

▶ Self–Directed Learning Activity ◀

1. Encode, compile and run the program in Listing 5.1. What is the output?
2. It will probably take a very long time for the program to terminate due to low memory. Try adding a local variable declaration say `double A[10]` inside `Bad()` function. This local variable will eat up memory everytime there's a recursive call. Recompile and re–run the program to see what will happen.
3. Modify the program by adding `printf("Test");` after the recursive function call. Run the program. Notice that you will not get `"Test"` as part of the output. What do you think is the reason for this?

An infinite recursion results into a logical error! The moral of the story from this bad example is that a recursive function must be defined such that it will terminate after a finite number of computational steps.

There is actually something worse than the bad example function presented above. Can you guess? Answer: a `main()` function that calls itself. This is a blunder that beginning programmers commit unknowingly. Although it is syntactically possible, unwritten rules in actual programming practice discourage a recursive `main()` function.

5.4 Concept of Base Case and Recursive Case

A recursive function should be defined such that it has two parts, namely:

1. **Base case** – this is the non–recursive portion of the function that solves the *trivial* part of the problem. The trivial part usually just return a value (often a constant value) in case the return type is not `void`. The base case leads to function termination.
2. **Recursive case** – this is the *non–trivial* part wherein the function has to perform more computation that requires it to call itself.

Note that there maybe several base and recursive cases within the same function.

The base case is separated from the recursive case via a conditional control structure, i.e., an `if` (or `switch case`), to test for a condition which will determine whether the function will compute the base case or the recursive case.

5.5 Recursive Function Definition in C

As in all C functions, the function definition must specify the data type of the value to be returned by the function, this is then followed by the function name and the parameter list. Thereafter, it is followed by the body of the function.

A typical format for a recursive function definition is as follows:

```
<return type> <function name> ( [<parameter list>] )
{
     [local variable declaration]

     if (<expression>)
        [return] <base case>
     else
        [return] <recursive case>
}
```

The `return` keyword SHOULD be written when the return type of the function is NOT `void`. Alternatively, we can invert the logic of the conditional expression such that the `recursive case` can be placed in the body of the `if` clause and the `base case` will be in the `else` clause. That is,

```
     if (<expression>)
        [return] <recursive case>
     else
        [return] <base case>
```

It is possible to define a recursive function such that it just has an `if` without a corresponding `else`. That is,

```
     if (<expression>)
        [return] <recursive case>
```

In this particular scenario, the `base case` corresponds to a function termination when the expression evaluates to false.

We discuss several variations of recursive functions in the following subsections.

5.5.1 Recursive Function Returning a Value

We consider as our first correct example the C implementatoin of the factorial function. The recurrence relation

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

is a mathematical function that is defined recursively.[2] The exclamation mark denotes the factorial function. We see that the:

1. **base case** corresponds to the trivial task of returning a value of 1
2. **recursive case** corresponds to the non–trivial task of returning `n * (n-1)!`

The C program that computes the factorial of an integer is shown in Listing 5.2. It is very interesting to note that we actually translated a definition expressed in one language, i.e., the language of mathematics to another language, i.e., the C programming language.

We can see that the language of mathematics is more compact compared with C. The C translation require that we specify the data type of parameter `n`. The exclamation mark notation had to be translated into a user–defined function name `factorial`. The fact that there is a one–to–one correspondence between the mathematical and the C language function definition facilitates seamless translation.

Listing 5.2: Recursive factorial function

```
#include <stdio.h>

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

int main()
{
```

[2]On a historical note, the function definitions (non–recursive and recursive) in modern programming languages were actually patterned from the nature of functions and relations in mathematics. Recursive functions in particular are based from recurrence relations.

```
    int n;

    printf("Input n: ");
    scanf("%d", &n);
    printf("The factorial of %d = %d", n, factorial(n));
    return 0;
}
```

Let us trace how the program works. The program starts execution from the `main()` function, which calls `printf()` which in turn calls `factorial(n)`.

Consider first the trivial case when `n` is 0. The function call `factorial(0)` returns a value of 1 which is then displayed via the `printf()` statement in `main()`. Next, consider the non–trivial case when `n > 0`. Let us say that we call `function(3)`. The following are the corresponding computation steps in sequence:

1. `main()` calls `printf()`
2. `printf()` calls `factorial(3)`
3. Recursive case `factorial(3)` calls `factorial(2)` via `return 3 * factorial(2)`
4. Recursive case `factorial(2)` calls `factorial(1)` via `return 2 * factorial(1)`
5. Recursive case `factorial(1)` calls `factorial(1)` via `return 1 * factorial(0)`
6. Base case `factorial(0) = 1`
7. Value of `factorial(0)` computed as 1, value of `factorial(1) = 1 * 1`
8. Value of `factorial(1)` computed as 1, value of `factorial(2) = 2 * 1`
9. Value of `factorial(2)` computed as 2, value of `factorial(3) = 3 * 2`
10. Value of `factorial(3)` computed as 6, return result to `printf()` of `main()`.
11. `printf()` outputs a value of 6 on display screen.
12. `main()` terminates.

Notice that the parameter in the recursive call is changed, in this case by decrementing the value by one. It should not be identical as in the original function call; otherwise, the recursive function will never terminate. IMPORTANT: A change in the state of the parameter is required for a recursive function to eventually reach its base case.

▶ Self–Directed Learning Activity ◀

1. Encode, compile and run the program in Listing 5.2. Run it with the following values for `n` and verify the results with manual computation.
   (a) 0
   (b) 1
   (c) 2
   (d) 3
2. The value of the factorial "grows" at a "fast rate" as we increase the value of `n`. Continue experimenting with the program by running it with values of n = 4, 5, 6.. Notice what happens to the printed value. To handle larger whole numbers, try adding the keyword `long` before `int` to make the data type a long integer.
3. What do you think will happen if the `return` keyword is removed from the `else` clause? Verify it by modifying and running the program.
4. Modify the conditional expression in the original program such that the recursive case will be the body of the `if` clause and the base case will be the body of the `else` clause.
5. What do you think will happen if the body of the function is written such that there is only statement which is `return n * factorial(n-1);`?

5.5.2 Recursive Function of Type `void`

Listing 5.3 shows a simple recursive function that does not return any value. The keyword `return` need not be used in the function.

The condition `n > 0` determines whether the function `Series1()` will call itself recursively or not. The associated action for the base case is to terminate the function which happens when the conditional expression evaluates to false. The recursive case is computed when the condition evaluates to true. Notice that the parameter in the recursive call is changed to reach the base case.

Listing 5.3: Recursive function of type `void`

```c
#include <stdio.h>

void Series1(int n)
{
   if (n > 0) {
      printf("n = %d\n", n);
      Series1(n-1);
   }
}

int main()
{
    int n;

    printf("Input n: ");
    scanf("%d", &n);
    Series1(n);
    return 0;
}
```

Let us trace how this recursive function works. The program starts execution from `main()` function, which calls `Series1(n)`. Let us consider first the trivial case when `n` is less than 1 (0 or any negative integer). Calling `Series1(0)` will result into a base case which simply terminates the function call. Next, consider the non–trivial case when `n > 0`. Let us say that we call `Series1(2)`. The following are the corresponding computation steps in sequence:

1. `main()` calls `Series1(2)`
2. Recursive case, `printf()` outputs `n = 2`, then `Series1(2)` calls `Series(1)`
3. Recursive case, `printf()` outputs `n = 1`, then `Series1(1)` calls `Series(0)`
4. Base case, `Series(0)` terminates.
5. `Series1(1)` terminates.
6. `Series1(2)` terminates.
7. `main()` terminates.

► Self–Directed Learning Activity ◄

1. Encode, compile and run the program in Listing 5.3. Run it with the following values for `n` and verify the results with manual computation.
   (a) -5
   (b) 0
   (c) 5
   (d) 10

   What do you think is the program doing?
2. Modify the program such that the last output will be zero.
3. What do you think will happen if we change the recursive call parameter from `n-1` to `n`?

5.6 Tail Recursive and Non–Tail Recursive Functions

A recursive function is said to be *tail recursive* if the last action that the function will do is to call itself. On the other hand, a recursive function that has to do some other action after calling itself is said to be *non–tail recursive.*[3]

The `Series1()` function in the previous program is an example of a tail–recursive function. The `factorial()` function is non–tail recursive. Recall the statement `return n * factorial(n-1)` in the function definition. This requires that we compute FIRST the value of `factorial(n-1)` and only AFTER that can we perform the last action which is to multiply it with `n`.

Let us look at another example of non–tail recursive function. What do you think will happen if we interchange the sequence of the `printf()` and the recursive function call in the previous program? This is actually what is shown in Listing 5.4.

Notice that `Series2()` is a non–tail recursive function because there is still another action, i.e., to execute `printf()`, waiting to be executed AFTER the recursive call has finished its task.

[3]Other books and literature use the term *tail–end recursion.*

Listing 5.4: Example of a non–tail recursive function

```c
#include <stdio.h>

void Series2(int n)
{
   if (n > 0) {
      Series2(n-1);
      printf("n = %d\n", n);
   }
}

int main()
{
    int n;

    printf("Input n: ");
    scanf("%d", &n);
    Series2(n);
    return 0;
}
```

Let us trace the program execution by calling `Series2(3)`. The following are the corresponding computation steps in sequence:

1. `main()` calls `Series2(3)`
2. Recursive case, `Series2(3)` calls `Series2(2)`. VERY IMPORTANT NOTE: the `printf()` statement cannot be executed yet because we have to finish first the task of computing `Series2(2)`.
3. Recursive case, `Series2(2)` calls `Series2(1)`. Same note applies.
4. Recursive case, `Series2(1)` calls `Series2(0)`. Same note applies.
5. Base case `Series2(0)` terminates.
6. `Series2(1)` calls `printf()` which outputs `n = 1`. `Series2(1)` terminates.
7. `Series2(2)` calls `printf()` which outputs `n = 2`. `Series2(2)` terminates.
8. `Series2(3)` calls `printf()` which outputs `n = 3`. `Series2(3)` terminates.
9. `main()` terminates.

The important lesson that we should have learned from this particular example is that any instruction written after a non–tail recursive function call will have to wait until the recursive call gets to finish its task first.

► Self–Directed Learning Activity ◄

1. Encode, compile and run the program in Listing 5.4. Run it with the following values for `n` and verify the results with manual computation.
   (a) -5
   (b) 0
   (c) 5
   (d) 10

   What do you think is the program doing?
2. Modify the program such that the first output will be zero.
3. What do you think will happen if we change the recursive call parameter from `n-1` to `n`?

5.7 More Representative Examples of Recursion

Let us look at other examples of recursive functions. Listings 5.5 and 5.6 are programs with recursive functions in charge of printing the elements of a 1D array. Listing 5.7 contains a recursive function that computes the total of array elements. Notice that each of the recursive function call passes `index + 1` in order for the program execution to approach the base case. Try to understand and trace what the programs are doing.

Listing 5.5: Accessing array elements via recursion (example 1)

```c
#include <stdio.h>

void AccessArray1(int A[], int index, int n)
{
    if (index < n) {
        printf("A[%d] = %d\n", index, A[index]);
        AccessArray1(A, index+1, n);
    }
}
```

```

int main()
{
    int A[5] = {10, 20, 30, 40, 50};

    AccessArray1(A, 0, 5);
    return 0;
}
```

Listing 5.6: Accessing array elements via recursion (example 2)

```
#include <stdio.h>
void AccessArray2(int A[], int index, int n)
{
    if (index < n) {
        AccessArray2(A, index+1, n);
        printf("A[%d] = %d\n", index, A[index]);
   }
}

int main()
{
    int A[5] = {10, 20, 30, 40, 50};

    AccessArray2(A, 0, 5);
    return 0;
}
```

Listing 5.7: Computing sum of array elements via recursion

```
#include <stdio.h>
int SumArray(int A[], int index, int n)
{
    if (index < n)
         return A[index] + SumArray(A, index+1, n);
}

int main()
{
    int A[5] = {10, 20, 30, 40, 50};

    printf("Sum of array elements = %d\n", SumArray(A, 0, 5));
    return 0;
}
```

5.8 Chapter Summary

The key ideas presented in this chapter are summarized below:

- Recursion is a technique that allows us to define functions that are self–referential. A recursive function has two parts, namely (i) base case and a (ii) recursive case.
- The base case eventually leads to program termination. The recursive case is the portion of the function where a function calls itself.
- A recursive function can either be classified as tail–recursive or non–tail recursive.
- Recursion is best used in problems that are recursive in nature, and where an iterative solution is difficult to formulate.

Problems for Chapter 5

Problem 5.1. Trace the program in Listing 5.8. Is `Mystery()` tail–end or non–tail end recursive?

Listing 5.8: `Mystery()` recursive function

```
#include <stdio.h>

int Mystery(int x, int y)
{
    if (y == 0)
       return x;
    else
       return (Mystery(y, x % y));
}

int main()
{
     printf("Mystery(2, 0) = %d\n", Mystery(2, 0));
     printf("Mystery(10, 25) = %d\n", Mystery(10, 25));
     return 0;
}
```

Problem 5.2. Trace the program in Listing 5.9. Is `DisplayDigits()` tail–end or non–tail end recursive?

Listing 5.9: `DisplayDigits()` recursive function

```
#include <stdio.h>

void DisplayDigits(int n)
{
    if (n < 10)
        printf("%d\n", n);
    else {
        printf("%d\n", n % 10);
        DisplayDigits (n/10);
    }
}

int main()
{
    int n;
```

```c

    printf("Input a positive integer: ");
    scanf("%d", &n);   /* try to input 123456 */
    DisplayDigits(n);
}
```

Problem 5.3. Trace the program in Listing 5.10. Is `BLAP()` tail–end or non–tail end recursive? Is `BLIP()` tail–end or non–tail end recursive?

Listing 5.10: `BLIP()` and `BLAP()` recursive functions

```c
#include <stdio.h>

void BLIP(int n)
{
     if (n != 0)  {
        printf("BLIP = %d\n", n);
        BLIP(n-1);
     }
}

void BLAP(int n)
{
     if (n!= 0)  {
        BLIP(n);
        BLAP(n-1);
     }
}

int main()
{
    BLAP(4);
    return 0;
}
```

Problem 5.4. The Fibonacci function is given by the following formula:

$$\text{Fibonnaci(n)} = \begin{cases} 1 & \text{if } n \text{ is 0 or } n \text{ is 1} \\ \text{Fibonacci(n-1)} + \text{Fibonacci(n-2)} & \text{if } n > 1 \end{cases}$$

- Which is the base case? recursive case?
- Evaluate the value of the following function calls

  - `Fibonacci(0)`
  - `Fibonacci(1)`
  - `Fibonacci(2)`
  - `Fibonacci(5)`

- Write a recursive C function that implements the Fibonacci function. What is the return type of the function? How many parameters are needed? What is the data type of the parameter?
- Is the function tail–recursive or non–tail recursive?
- It should be realized that although the mathematical definition is recursive, it is possible to write an iterative solution (i.e., using loop). Write an iterative solution and compare its performance with the recursive solution. Try your programs on increasing value of `n` starting from 0. Which is bettter in terms of computational performance, the recursive or the iterative solution?

Problem 5.5. The Ackermann function is the following formula (where `m` `n` are whole numbers

$$\text{Ackermann(m, n)} = \begin{cases} n+1 & \text{if } m \text{ is } 0 \\ \text{Ackermann(m-1, 1)} & \text{if } n \; 0 \\ \text{Ackermann(m-1, Ackermann(m, n-1))} & \text{otherwise} \end{cases}$$

- Which is the base case? recursive case?
- Evaluate the value of the following functions
  - `Ackermann(0, 10)`
  - `Ackermann(1, 2)`
  - `Ackermann(2, 1)`
  - `Ackermann(3, 0)`
- Write a recursive C function that implements the Fibonacci function. What is the return type of the function? How many parameters are needed? What is the data type of the parameter?
- Is the function tail–recursive or non–tail recursive?