Workshop 4 – Tools of the Trade: Part 2 – Recursion

4.0 Overview

Recursion is used throughout this unit in the development of algorithms.

This workshop covers the basic techniques of recursion. Also covered is both the power and the dangers of recursion and when you should consider using a recursive technique for solving an algorithm.

Workshop 5 will both extend and reinforce the material covered in this workshop through the study and analysis of a variety of different types of recursive algorithms.

Objectives of this workshop

After studying this workshop you should be able to:

- Understand how to think recursively
- Demonstrate an understanding of recursive methods
- Write a recursive algorithm
- Trace a recursive algorithm
- Appreciate the cost of recursion for some types of algorithms
- Recognize when tail-end recursion occurs and understand how to remove tail-end recursion.

Reading

- Goodrich & Tamassia (6th Edition) : Chapter 5
- Koffman & Wolfgang: Chapter 5
- Dale, Joyce & Weems : Chapter 3

Activities and Exercises

There are eight activities and three tasks associated with this workshop and plus a collection of exercises.

4.1.Background to recursion

So far repetition of code has been achieved by using loops (for loops and while loops).

An alternative to achieve repetition is through recursion, which occurs when a function calls itself.

Recursive techniques can be an elegant and powerful alternative for performing repetitive tasks provided it is used with care and understanding as there are some potential dangers to using recursion for the solution of some algorithms – ie recursion must always be used with care (example – see section 4.6).

4.2. Examples of recursion in life

4.2.1. A well-known recursive story.

It was a dark and stormy night
And the captain said to the mate
Tell me a story
So the mate began …

> It was a dark and stormy night
> And the captain said to the mate
> Tell me a story
> So the mate began …
>
> > It was a dark and stormy night
> > And the captain said to the mate
> > Tell me a story
> > So the mate began …
> >
> > > It was a dark and stormy night
> > > And the captain said to the mate
> > > Tell me a story
> > > So the mate began …
> > >
> > > and again and again and again and …….

Activity 4.2.1

What do you observe about this story?

Do you see any problems?

What would happen if you wrote a program like this?

Answers to this activity will be discussed during the lecture

4.2.2. The Russian Matryoshka Dolls.

Russian Matryoshka dolls are a set of identical hollow dolls of a range of sizes, which fit inside each other.

If you remove the top of one doll you will find a slightly smaller identical one inside. The smallest (innermost) doll cannot be opened.

Activity 4.2.2

What do you observe?

Why does this solve the problem of the previous example?

4.3. What is Recursion?

Recursion is a technique whereby a problem is expressed in a similar form to the original problem but smaller in scope.

Recursion is applied to problems where:

- The solution is easy to specify for certain conditions **– stopping case.**
- There are well defined rules for proceeding to a new state which is either a stopping case or eventually leads to a stopping case **– recursive steps.**

4.4. A simple recursive example

The factorial of a positive number n, denoted by n!, is defined as the product of the integers from 1 to n.

ie: for any integer n ≥ 0

$$\text{factorial}(n) = n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) * (n-3) * \ldots. 3 * 2 * 1 & \text{if } n \geq 1 \end{cases}$$

For example: factorial(5) = 5 * 4 * 3 * 2 * 1 = 120

4.4.1. Recursive definition for factorial(n):

Now: factorial(5) = 5 * (4 * 3 * 2 * 1)
= 5 * factorial(4)
and factorial(4) = 4 * (3 * 2 * 1) = 4 * factorial(3)
and so on …

Hence for a positive integer n we have factorial(n) = n * factorial(n-1)

Giving the following recursive definition

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

<– this is the stopping case.
<– these are the recursive steps

4.4.2. Recursive implementation for factorial(n):

The recursive algorithm for factorial(n) is

```
if n == 0  then
        answer is 1
else
        answer is n x factorial(n-1)
end if
```

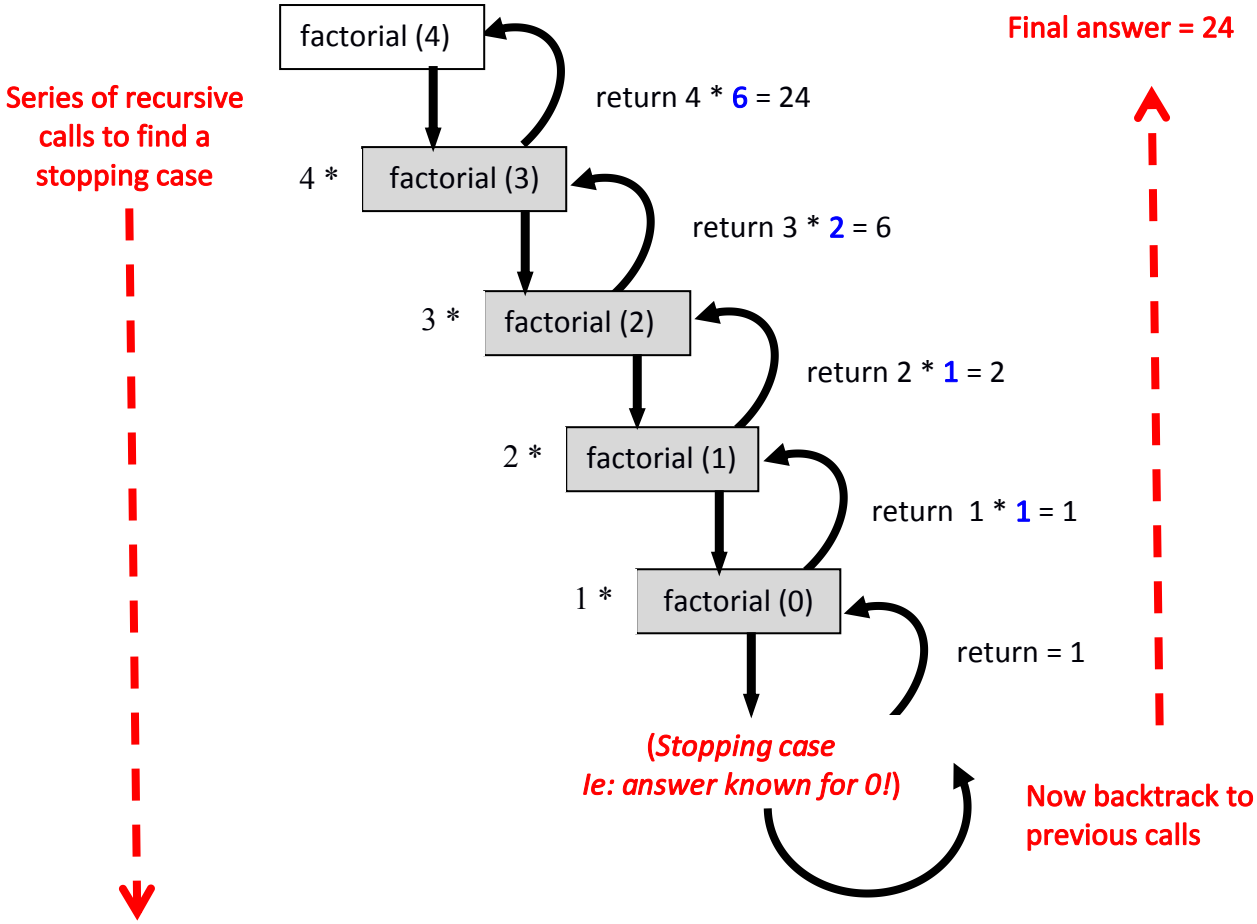
The recursive method is

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

4.4.3. Tracing a recursive algorithm

Trace for factorial (4)

Note: Simply follow the arrows down for each recursive call and then backtrack up as it unravels itself.



Activity 4.4.3(a)

(a) Discuss whether a recursive approach should be used to evaluate n!

Do you think an iterative approach is better? Justify!

(b) Give the algorithm for an iterative solution for the evaluation of n!

Answers to this activity will be discussed during lecture and in the tutorial for this workshop

Activity 4.4.3(b) – Mystery method

Given the recursion method:

```
public static int mystery (int n, int m) {
if (n == 0)
        return m;
else
        return  mystery (n-1, m) + 1;
}
```

What is output from call of mystery (4,5)?

What is mystery?

Answers to this activity will be discussed during lecture and in the tutorial for this workshop

Activity 4.4.3(c) – Summing the elements of an array recursively

Given an array A of n integers develop a **recursive** algorithm to find the sum of the n integers.

What is the stopping condition?
What are the recursive steps?
Draw a trace of your recursive algorithm for an array A consisting of 5 integers: 4, 3, 6, 2, 5.

Answers to this activity will be discussed during lecture and in the tutorial for this workshop

Activity 4.4.3(d) – Reversing the elements of an array recursively

Given an array A of n integers develop a **recursive** algorithm to reverse the contents of the array A.

Answers to this activity will be discussed during the tutorial for this workshop

4.5. Example – Fibonacci Numbers (sequence)

The Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, ...

The n^{th} Fibonacci number is sum of the previous two Fibonacci numbers.

A recursive algorithm to determine the n^{th} Fibonacci number is:

```
if N == 1 or N == 2 then
        answer is 1
else
        answer is Fibo(N-1) + Fibo(N-2)
end if
```

A non-recursive algorithm to determine the n^{th} Fibonacci number is:

```
set previous = 1
set current = 1

if N == 1 or N == 2 then
    answer = 1
else

    for J = 3 to N loop
        temp = current
        current = current + previous
        previous = temp
    end loop

    answer = current

end if
```

Observations

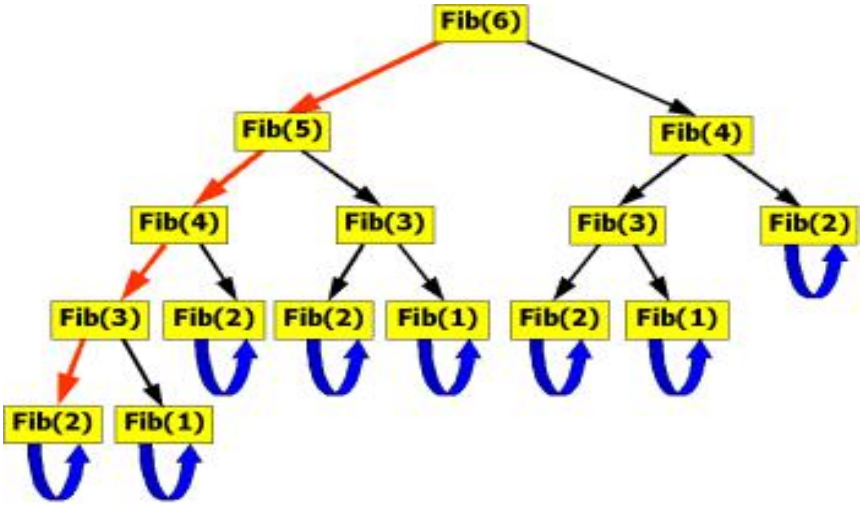
- Iterative version uses far fewer additions than is required for the recursive version.
- Iterative version for the n^{th} Fibonacci number requires n-2 additions (for n>=2).
- Recursive version requires (n^{th} Fibonacci number less one) additions!!

To calculate the 30^{th} Fibonacci number (832040) iteratively requires 28 additions, whereas the recursive version requires 832039 additions!

4.6. Cost of Recursion

In section 4.5 you observed that the recursive solution for the obtaining the 30th Fibonacci number required many more additions than the iterative solution – **why?**

Consider a trace to obtain the 6th Fibonacci number: Fib(6) = Fib(5) + Fib(4)



At each level the trace shows the recursive calls to evaluate a Fibonacci number.
The red arrows show the initial path taken:

Fib(6) needs the value of Fib(5)
Fib(5) needs the value of Fib(4)
Fib(4) needs the value of Fib(3)
Fib(3) needs the value of Fib(2)
Fib(2) we know has the value of 1.
At this stage we return to Fib(3) but
Fib(3) also needs the value of Fib(1)
Fib(1) we know has the value of 1.
At this stage we return to Fib(3)
Fib(3) can now be evaluated as 2
At this stage we return to Fib(4)
Fib(4) also needs the value of Fib(2) **and so on.**

Observations:

- The 4th Fibonacci number is evaluated twice
- The 3rd Fibonacci number is evaluated three times.

In this case the iterative solution (non-recursive) solution is significantly more efficient than the recursive solution even though the recursive solution is the obvious solution.

Some problems can only be solved using a recursive approach however when you have the choice you should always think about the hidden cost of recursion before solving a problem using a recursive algorithm.

4.7. How does recursion work?

Activity 4.7:

Some questions for you to find out the answers.

- What data structure is used to handle a recursive call?
- Why is this needed?
- What happens when a recursive call is made?

4.8. When not to use Recursion

You shouldn't use recursive techniques to solve the problem if the answer to any of the following questions is **NO**:

- Is the algorithm/data structure naturally suited to recursion?
- Is the recursive solution shorter and more understandable?
- Does the recursive solution run in acceptable time limits and/or space limits?

4.9. Tail End Recursion

Tail end recursion occurs when there is one recursive call in an algorithm and this call is the very last thing the method does. An example of tail end recursion is the algorithm developed in activity 4.4.3(d) (*reversing the contents of the array*). In this method having swapped the contents of the middle two items the task is completed and the method terminated – nothing is passed returned to previous methods.

Conversely the algorithm developed in activity 4.4.3(c) *(summing the elements of an array*) does **not** use tail-end recursion even though it last statement includes a recursive call. This recursive call is not the last thing the method does because after it receives the value from the recursive call it adds this to the value in the current array element and returns this sum to the calling method.

A recursive algorithm that uses tail end recursion can always be converted into an iterative non-recursive algorithm.

Activity 4.9 – Reversing the elements of an array iteratively (ie without using recursion)

Given an array A of n integers develop an **iterative** algorithm to reverse the contents of the array A.

Answers to this activity will be discussed during the tutorial for this workshop

Exercises – Recursion

1. What would be returned by the following calls:

   (a) XX(4)?

   (b) XX(10)?

   (c) XX(12)?

```
XX ( int A )
        if (A<5)
                return (3*A)
        else
                return (2 * XX(A-5) + 7)
        end if
end
```

2. This algorithm uses integer division.

   How many recursive calls are made by the following initial calls?

   (a) YY(7)?

   (b) YY(8)?

   (c) YY(15)?

```
YY( int Num )
        if (Num < 2)
                return 1
        else
                if (Num mod 2 == 0)
                        return YY(Num/2)
                else
                        return YY(3*Num + 1)
                end if
        end if
end
```

3. What would be returned by the following calls:

   a) ZZ(10,4)? b) ZZ(4,3)?

   c) ZZ(4,7)? d) ZZ(0,0)?

```
ZZ( int A,B)
        if (A>B)
                return -1
        else if (A == B)
                return 1
        else
                return (A * ZZ(A+1, B))
        end if
end
```

4. Design and write a recursive algorithm that computes the sum of all numbers from 1 to n, where n is given as parameter.

5. The *digital sum* of a number n is the sum of its digits.

   Design and write a recursive algorithm *digitalSum(n)* that takes a positive integer n and returns its digital sum.

   For example, *digitalSum(2019)* will return 12 because 2 + 0 + 1 + 9 = 12

6. Design and write a recursive program to display the integers in range (x, y).

   Example: range(2, 9) – gives output*:* 3, 4, 5, 6, 7, 8

7. Design and write a recursive algorithm to find the greatest common divisor (GCD) of two positive integer numbers where:
   - The GCD of 2 equal positive integer numbers is their common value
   - The GCD of 2 unequal positive integer numbers is the GCD of the smaller integer and their positive difference.