Recursion and Applications

Akshay Raj Verma

May 15, 2015

1 Introduction

Definition. Recursion is method of defining functions where the function self reference.
A recursion consists of two parts:

1. A base case (or cases)

2. An algorithm which is used to solve instances of smaller problems, until the problem is reduced to its base case.

Example: The Fibonacci Sequence is a famous example of a recursive function
Base Case: $f(0) = 0$ and $f(1) = 1$
Formula: $f(n) = f(n-1) + f(n-2)$ for all $\mathbb{N}$ $n \geq 2$
In summary recursion is a method of defining a sequence where each successive object in the sequence is defined by the previous object in the sequence. The initial objects are predefined.

2 Applications in Mathematics

$$
\begin{gathered}
f(0) = 0 \\
f(1) = 1 \\
f(2) = f(2-1) + f(2-2) = f(1) + f(0) = 1 \\
f(3) = f(3-1) + f(3-2) = f(2) + f(1) = 2 \\
f(4) = f(4-1) + f(4-2) = f(3) + f(2) = 3 \\
f(5) = f(5-1) + f(5-2) = f(4) + f(3) = 5 \\
\ldots
\end{gathered}
$$

As shown by the above sequences solving $f(2)$ requires the use of base cases $f(1)$ and $f(0)$. Once you have the solution to both of them, you apply the Fibonacci Formula to get $f(2)$. This process is again repeated for $f(3)$ which requires the solutions to $f(2)$ which in turn as noted above requires $f(1)$ and $f(0)$. The Fibonacci Formula is again used to solve it, and so on.
We can see the use of recursion in the Fibonacci Sequence in the procedure when in order to solve the Fibonacci Sequence for n we invoke the procedure in

order to solve the problem itself.
In conclusion in order to solve the Fibonacci Sequence for $f(n+2)$ using the recursive algorithm above is not possible without first knowing the solutions for $f(n+1)$ and $f(n)$. The Fibonacci formula is used $n+1$ times to acquire the solution for $f(n+2)$.
However the nth sequence of the Fibonacci Sequence can also be produced without the use of recursion by Binet's formula.

Theorem: For all n in $\mathbb{N}$ the Fibonacci Number, we denote F_n, can be given directly by the Binet Formula(B_n) as given below:

$$B_n = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

Proof: We will prove Binet's Formula through the use of strong induction.
The Principle of Mathematical Induction states that in order to prove the statement that $F_n = B_n$, for all $\mathbb{N}$, it is sufficient to prove: $B_1 = F_1$, and if $B_n = F_n$ is true then, $B_{n+1} = F_{n+1}$.
Lemma 1. If $B_n = F_n$, then for the base cases n=0, 1 $F_0 = B_0$ and $F_1 = B_1$
Proof.
We want to show that $B_0 = F_0$ and $B_1 = F_1$
For base case $n = 0$:
First we substitute 0 for n $B_0 = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^0 - (\frac{1-\sqrt{5}}{2})^0)$
Which is equal to $\frac{1}{\sqrt{5}}((1) - (1))$
Therefore $B_0 = 0$
And so by the Transitive Property $B_0 = F_0$
For base case $n = 1$:
First we substitute 1 for n $B_1 = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^1 - (\frac{1-\sqrt{5}}{2})^1)$
So $\frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2}) - (\frac{1-\sqrt{5}}{2}))$
Which is equal to $\frac{1}{\sqrt{5}}(\frac{(1+\sqrt{5})-(1-\sqrt{5})}{2})$
Therefore $= \frac{1}{\sqrt{5}}(\frac{2*\sqrt{5}}{2})$
Simplifying further $= \frac{1}{\sqrt{5}}(\sqrt{5})$
Therefore $B_1 = 1$
And so by the Transitive Property $B_1 = F_1$
For Base Cases $n = 0$ and $n = 1$, Binet's Formula is true. ■
Lemma 2.1 If Binet's Formula is true for all $k \in \mathbb{N}$ then it is true for $k+1$, for $k > 1$.
Proof.
To demonstrate Binet's Formula is true for k+1 we must demonstrate that:

$$F_{k+1} = B_{k+1}$$
$$\text{Which is equal to } \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{k+1} - (\frac{1-\sqrt{5}}{2})^{k+1})$$

We can rewrite Binet's Formula in terms of α and β. $\alpha = \frac{1+\sqrt{5}}{2}$ and $\beta = \frac{1-\sqrt{5}}{2}$.

α can be recognized also as the Golden Ratio.
Substituting in α and β into Binet's Formula:

$$B_n = \frac{\alpha^n - \beta^n}{\sqrt{5}}$$

Lemma 2.1. $\alpha + 1 = \alpha^2$ and $\beta + 1 = \beta^2$
Proof. In order to solve Lemma 2 we want to show that α+1=α^2 and β+1 $= \beta^2$

$$\alpha + 1 = \frac{1+\sqrt{5}}{2} + 1 = \frac{1+\sqrt{5}}{2} + \frac{2}{2} = \frac{3+\sqrt{5}}{2}$$
$$\alpha^2 = (\frac{1+\sqrt{5}}{2})^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2}$$

So by the transitive property $\alpha^2 = \alpha + 1$

$$\beta + 1 = \frac{1-\sqrt{5}}{2} + 1 = \frac{1-\sqrt{5}}{2} + \frac{2}{2} = \frac{3-\sqrt{5}}{2}$$
$$\beta^2 = (\frac{1+\sqrt{5}}{2})^2 = \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2}$$

So by the transitive property $\beta^2 = \beta + 1$■

Conclusion Now we continue with the proof of Lemma 2.1, and we proceed using the definition of Fibonacci sequence and induction.

$$F_{k+1} = F_k + F_{k-1}$$
$$= \frac{1}{\sqrt{5}}(\alpha^k - \beta^k) + \frac{1}{\sqrt{5}}(\alpha^{k-1} - \beta^{k-1})$$
$$= \frac{1}{\sqrt{5}}(\alpha^k - \beta^k - \beta^k - \beta^{k-1})$$
$$= \frac{1}{\sqrt{5}}(\alpha^{k-1}(\alpha + 1) - \beta^{k-1}(\beta + 1))$$
$$= \frac{1}{\sqrt{5}}(\alpha^{k-1}\alpha^2 - \beta^{k-1}\beta^2)$$
$$\text{Using Lemma 2.1} = \frac{1}{\sqrt{5}}(\alpha^{k+1} - \beta^{k+1})$$

Using the Principle of induction we have shown that $B_n = F_n$ for all $n \in \mathbb{N}$, by proving the Base Cases of n=0 and n=1, then using induction we found that the theory is true for all $n \in \mathbb{N}$ for all $n > 1$, by applying the recursive definition of the Fibonacci sequence. ■

Conclusions:

Example Factorials are another example of a recursive function defined as $n!$ for all non negative $\mathbb{Z}$
Base Case $n = 1$, where the factorial of 1 (denoted 1!), is1. One can argue that 0! is another base case but the $0! = 1$ so it is inconsequential.
Formula $n! = n * (n - 1)!$ for all non negative $\mathbb{Z}$

Implementation

1!=1
2!=2*1!=2

$$3!=3*2!=6$$
$$4!=4*3!=24$$
$$5!=5*4!=120$$
$$6!=6*5!=720$$
$$...$$

Factorial, denoted by $n!$, is the product of all positive integers less than or equal to the positive integer n. As shown by the process above, we can see repetition in the process the factorial function uses to solve for smaller $\mathbb{Z}$ before building up to the bigger $\mathbb{Z}$.

3 Applications in Computer Science

Recursion is an important element in computer science. Not only can it be used to find mathematical quantities like Factorials, the Fibonacci Sequence but it is also used to create Data Structures to store information, or search through some structures.
Below is a program written in Pseudo code to compute factorials:

```
function:  factorial
input:     an non negative integer n whose factorial will be computed
output:    print factorial(n)

        if n is 0 return 1
        else return (n * factorial(n-1))
```

Step by Step Computing for $n = 4$
One way to imagine recursion in Computer Science is like a stack of coins. Each time the function factorial is called a new "coin" is placed on top of the stack. Each "coin" is a version of the function being called except different data is being used. When n=0 however we instead begin to remove the coins, as now the function is no longer being called rather it is returning data. We begin to "remove" the coins, one by one as each coin now "returns" the data instead of calling the function and continues to do so until no more "coins" are left.
A step by step version of the program is shown below:

1. $n = 4$ so $4 * factorial(n - 1)$
2. $n = 3$ so $3 * factorial(n - 1)$
3. $n = 2$ so $2 * factorial(n - 1)$
4. $n = 1$ so $1 * factorial(n - 1)$
5. $n = 0$ so $return 1$

6. Now $n = 0$ and no more functions are being called
7. return $1 * 1$
8. return $2 * 1$
9. return $3 * 2$
10. return $6 * 4$
11. print 24

A similar program can be written to solve the recursive version of Fibonacci Sequence.
Writing a program to solve the Fibonacci Sequence
Below is a program written in the programming language C to solve the Fibonacci Sequence

```
/* Input: non negative integer n.
Output: Fibonacci Number for n.
*/
int fibonacci(int n)
{
        if(n=0)
                return 0;
        else if(n=1)
                return 1;
        else
                return(fibonacci(n-1)+fibonacci(n-2));
        }
int main(){
        int n;
        printf(fibonacci(n));
        return 0;
}
```

Use of recursion in code is signified by the *return* statement calling the the same function it is currently in. In the *intfibonacci* function we see it in this statement:

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

Here the return statement is calling the fibonacci function, not once but twice, as opposed to the factorial program where the return statement would only be calling the factorial function once.

As we can see in the program above the function fibonacci, unlike the function factorial, is being called to return two separate times: once to calculate the Fibonacci number for $n - 1$ and again to calculate the Fibonacci number for $n - 2$. The function call to $fibonacci$ is repeated until $n = 0$,and $n = 1$ are called.

4 Recursion vs. Iteration

When we talk about recursion, we should also consider iteration. Anything done recursively can be done iteratively. But when do we choose one over another. Generally speaking, in computer science, recursion is an easier to read solution than a solution that is written iteratively. While the code may look cleaner and is more concise, what is happening as the program is running may not be as swift. First, let's consider readibility and consider an iterative Tower of Hanoi implementation. We will assume that a stack called Stack has been created and will be used for the succession of the solution. Furthermore, we will assume each edge case is being handled. And finally, we will assume all the appropriate helper functions have been implemented. Here, I will just show the iterative step.

```
public void iterativeHanoi (int numberOfDisks, struct Stack *sourceStack, struct
{
        int i;
        int totalMoves;

        for (i = numberOfDisks; i >= 1; i--)
        {
                push(sourceStack, i);
        }

        for (i = 1; i <= totalMoves; i++)
        {
                if (i$ % 3 == 1)
                {
                        moveDisks(sourceStack, finalStack);
                }

                else if (i % 3 == 2)
                {
                        moveDisks(sourceStack, transitionStack);
                }

                else if (i % 3 == 0)
                {
```

```
                moveDisks(transitionStack, finalStack);
            }
    }
}

public void recursiveHanoi (int diskSize, struct Stack *sourceStack,
structStack *transitionStack, struct Stack *finalStack)
{
    // Base Case
    if (diskSize = 0)
    {
        // done
    }

    else
    {
        recursiveHanoi(diskSize - 1, soureStack, transitionStack, finalS
        moveDisk(diskSize, finalStack);
        recursiveHanoi(diskSize - 1; transitionStack, finalStack, source
    }
}
```

5 Problems with Recursion

Recursion makes code more readable and more concise. Furthermore, in mathematics, we can use induction to prove the correctness of recursive code. But in computer science, recursion can cause performance issues in terms of time needed to complete the recursion as well as the memory required to do the recursion.In fact, in order to make recursion better on time and space, we would need to use other methods of programming such as dynamic programming to maximize efficiency.
So, what makes recursion inefficient? First, let us understand how recursion works when implemented by a compiler.

When we talk about recursion in computer science, we need to talk about call stacks and stack frames. A call stack is a data structure that is used by the program to store current information about the subroutines as the program is being executed. The call stack is essential to keep track of how subroutines delegate control once it finishes executing. A call stack is composed of stack frames and a new one is created everytime a subroutines is called. So, in recursion, everytime we call the recursive function, a new stack from is created.

So, from the above information, we can see why recursion can be a lot slower and memory inefficient. If we are creating a new stack frame for each recursive call, we clog up memory and we may run into a stack overflow. Furthemore, because we are using a stack to keep track of the recursion, the program is using a lot of resources to maintain the ever growing stack instead of just the calculations we want from the recursion, which leads into slower execution times. Furthermore, it's hard to understand how far we can go with our recursion without experiencing a stack overflow.

6 Conclusions

Recursion is an important element of Mathematics and Computer Science, used in everyday programs and calculations. Further on, it is not purely a mathematical construct, it has appearances in nature too. Snowflake patterns, crystals, and leaf patterns contain fractals which are recursive patterns. If you put two mirrors parallel to each other, an infinite number of nested images are created. This is an example of infinite recursion in nature. Although there are non recursive ways to implement otherwise Recursive formulas and/or programs, it is more inefficient to use the non recursive formula in certain cases, than it is to use the recursive, as evidenced by the Binet Formula and the Fibonacci Formula.