Recursive Algorithms

Recursion is a form of definition and of algorithms that is very important in computer science theory **as well as in practice**. Recursive algorithms can be inefficient or efficient.[1] A recursive definition or a recursive algorithm is characterized by self-reference. Typically with recursion, a function is defined in terms of an earlier version of itself. Since this self-reference can't go on forever, there must be a termination condition. The termination condition is checked first and if it does not apply then the algorithm goes through with the self-reference.

Example The usual prototypical example of a recursive definition is the factorial function normally defined by $n! = n\cdot(n-1)\cdot(n-2)\cdot(n-3)\ldots3\cdot2\cdot1$ for positive integers and $0! = 1$. A recursive definition of n factorial is:

Definition Factorial

$$0! = 1$$

$$N! = N\cdot(N-1)!\ ; \quad N > 0$$

To evaluate 5! we have $5! = 5\cdot4!$. To evaluate 4! we must go back to the definition and we get $4! = 4\cdot 3!$ and thus $5! = 5\cdot4\cdot3!$. Similarly $3! = 3\cdot2!$ which implies $5! = 5\cdot4\cdot3\cdot2!$. We have $2! = 2\cdot1!$, and $1! = 1\cdot0!$. However the first part of the definition gives $0! = 1$. Putting all of this together we get $5! = 5\cdot4\cdot3\cdot2\cdot1\cdot1 = 120$.

Now the example just given may seem awkward but in a computing environment recursive definitions are frequently easier to write and debug than non-recursive definitions. The

[1]Computing efficiency is a topic outside of the scope of this text. This does not mean that I can't mention it or refer to it.

dirty work and record keeping is done by the compiler. The program will contain a definition that is little more than the two-line definition given above.

Example The other prototypical recursive definition is Fibonacci numbers. Fibonacci numbers show up a great deal in mathematics and computer science. However we will not make any use of them in this book other than for this example. The Fibonacci numbers are a sequence of integers denoted by $F_0, F_1, F_2, F_3, ...$ The easiest definition of Fibonacci numbers is recursive (there do exist non-recursive definitions). The recursive definition is

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}\ ;\ \ n \geq 2.$$

By this definition we get $F_2 = F_1 + F_0 = 1 + 0 = 1$. $F_3 = F_2 + F_1 = 1 + 1 = 2$. $F_4 = F_3 + F_2 = 2 + 1 = 3$. The first few entries are given in the following table:

n	Fn	n	Fn
0	0	13	233
1	1	14	377
2	1	15	610
3	2	16	987
4	3	17	1597
5	5	18	2584
6	8	19	4181
7	13	20	6765
8	21	21	10946
9	34	22	17711
10	55	23	28657
11	89	24	46368
12	144	25	75025

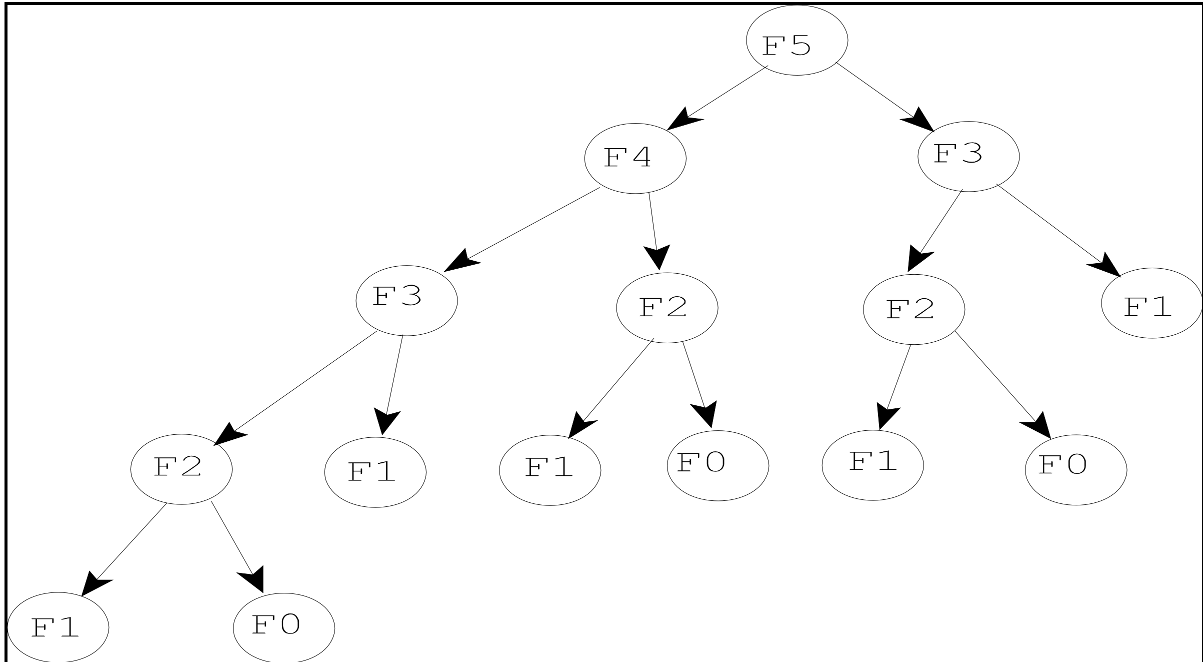

Figure 1 The Tree of Recursive Calls for F_5.

The recursive definition of Fibonacci numbers is easy to use. However it is very slow. For example, if we want to compute F_5, the recursive definition requests F_4 and F_3. But when we use the recursive definition to compute F_4 it requires F_3 and F_2. Note that we have already had to compute F_3 twice. The situation snowballs from there. The full situation for computing F_5 is shown in the tree diagram in **Figure 1**. Computing F_{200} with the recursive algorithm would severely tax a super-computer. Note that there are recursive definitions for Fibonacci numbers that are efficient but they are not as simple as the usual

definition. Lastly, it is worth mentioning that in a spreadsheet the usual recursive definition is very efficient.[1]

Define

$$\varphi = \frac{1+\sqrt{5}}{2}$$

$$\alpha = -\frac{1}{\varphi} = \frac{1-\sqrt{5}}{2}$$

where φ is the *harmonic ratio* or *golden mean*.

It can be shown using standard techniques used for difference equations that $F_n = \frac{1}{\sqrt{5}}\varphi^n - \frac{1}{\sqrt{5}}\alpha^n$. From this we can get a simpler formula: $F_n = \left\lfloor \frac{1}{\sqrt{5}}\varphi^n + .5 \right\rfloor$.

[1] The same algorithm works efficiently in one place and inefficiently elsewhere because the spreadsheet as a programming environment works very differently from a general purpose language such as C or Pascal.

An efficient non-recursive algorithm for the n'th Fibonacci number is:

```
Fibonacci Algorithm

    Input n (integer)
    If n = 0 or n = 1 then
        Begin
            Answer = n
            exit
        End
    PriorF = 0
    F = 1
    For i = 2 to n do
        Begin
            switch = F
            F = F + PriorF
            PriorF = Switch
        End
    Answer = F
```

☐ **Exercise [1]** Generalized Fibonacci numbers are defined just like Fibonacci numbers ($G_n = G_{n-1} + G_{n-2}$) except that G_0 and G_1 can be any two numbers. From this viewpoint Fibonacci numbers are just the special case of Generalized Fibonacci numbers where $G_0=0$ and $G_1=1$. Solve for G_7 when $G_0=1$ and $G_1=3$. Do the same problem for $G_0=-1$ and $G_1=2$. Do the problem for $G_0=1$ and $G_1=4$. (Note the similarity between the first and second cases.)

The case of Generalized Fibonacci numbers where $G_0 = 2$ and $G_1 = 1$ are called *Lucas numbers*. They satisfy $L_n = \varphi^n + \alpha^n$ where L_n is the n'th Lucas number and φ and α are defined as before with φ being the golden ratio.

☐ **Exercise [2]** Write a recursive algorithm using only multiplication for evaluating x^n when x is any real number and n is a non-negative integer.

Example (This example is somewhat more challenging that the previous ones. Feel free to skim it, and go on.) The n'th Bell number B_n is the number of ways of partitioning n (distinct) objects. One way of computing the Bell numbers is by using the following double recursive algorithm. This algorithm computes numbers with two arguments: B(i,j). The n'th Bell number, B_n is computed as B(n,n). For example to find B_3 compute B(3,3).[1]

B(1,1) = 1.

B(n,1) = B(n-1,n-1) for n > 1.

B(i,j) = B(i-1,j-1) + B(i,j-1) for n > 1 and 1 < j ≤ i.

---

[1]This algorithm is a translation of a table method mentioned in many texts that is similar to Pascal's triangle. However, Pascal's triangle which is covered later in this text has a simple explanation. I have been unable to find one for this algorithm. (Perhaps it is equally simple, but I sure have stumbled trying to find it.

☐ **Exercise 3** Write computer code to implement the above algorithm for Bell numbers.

☐ **Exercise 4** Use the previous exercise to compute Bell numbers B_1 through B_7.

Example A particularly wicked recursive function is the Ackerman function. Whereas it has been important to theoretical computer science since before the first digital computers, it became famous in recent years after the development of languages with recursive calls (such as Pascal and C) and the development of personal computers. It is defined as follows:

$A(0,y) \leftarrow y + 1$

$A(x+1,0) \leftarrow A(x,1)$

$A(x+1,y+1) \leftarrow A(x,A(x+1,y))$

The problem with the Ackerman function is that although it is trivial to implement in a computer language such as Pascal, it uses humongous time and memory. Computing A(10,10) will blow away any computer.

1.

n	GFN-First Case	GFN-Second Case	GFN-Third Case
0	1	-1	1
1	3	2	4
2	4	1	5
3	7	3	9
4	11	4	14
5	18	7	23
6	29	11	37
7	47	18	60

2. Input x (real number), n (integer n ≥ 0)

```
Function Power(x,n)
        if n = 0 then return 1; exit;
        return x·Power(x,n−1)
end;
```

If this is not clear to you, try to study it until you understand what is happening. You may find this kind of thinking useful elsewhere.

Note: 0^0 is not defined. Hence this algorithm is not correct for x=0 and n=0. How would you correct it? (Is it worth the effort?)

3.

```
Function B(i,j)
If i = 1 then return 1
Else if j = 1 then return B(i-1,i-1)
Else return B(i-1,j-1) + B(i,j-1)
```

4. The first seven Bell numbers are 1, 2, 5, 15, 52, 203, and 877.