

Chapter 37 – Recursion

1. The Mathematics of Recursion

A recursive algorithm is an algorithm that is defined in terms of itself. In the simplest terms, recursion is a type of shorthand for defining an infinitely repeating mathematical concept without having to resort to an infinitely growing definition.

For example, the Fibonacci number series starts with the numbers 0 and 1. The next number in the series is found by adding the rightmost two known Fibonacci numbers together. This produces the series 0, 1, 1. To find the next number in the series, the two rightmost numbers in the list are then added together. This produces the series 0, 1, 1, 2. To find the next number in the series, the two rightmost two numbers are added together, which produces the series 0, 1, 1, 2, 3. This process could go on literally forever and the Fibonacci series would not be defined. In order to avoid this, the series is defined as a function with one parameter with three possible outcomes. These are

$$\begin{array}{ll}
 f(0) = 0 & \textit{pronounced f of zero is 0} \\
 f(1) = 1 & \textit{pronounced f of 1 is 1} \\
 f(n) = f(n-1) + f(n-2), & \textit{pronounced f of n is f of n-1 plus f of n-2} \\
 & \text{where n is any integer greater than 1}
 \end{array}$$

Using this definition, any Fibonacci number can be found. For example, the zeroth number, $f(0)$, in the series is defined as 0. The first Fibonacci number, $f(1)$, is defined as 1. The second Fibonacci number, $f(2)$, is defined as $f(2-1) + f(2-2)$, which is $f(1) + f(0)$. Since both $f(1)$ and $f(0)$ are known, the answer for $f(2)$ can be calculated. In this fashion, each successive number in the Fibonacci series can be derived from the definition.

2. Recursion in Programming Languages

In 3rd generation programming languages such as C++, recursion occurs when a function calls itself. In C++, as in other programming languages that support it, recursion is used as a form of repetition that does not involve iteration. Instead of repeating the same unit of code and using the same memory locations for variables, recursion involves allocating space in memory for each recursive call.

As it happens, *any problem that can be solved via iteration can be solved using recursion* and *any problem that can be solved via recursion can be solved using iteration*. Iteration is preferred by programmers for most recurring events, reserving recursion for instances where the programming solution would be greatly simplified. In a programming language, recursion involves an additional cost in terms of the space used in RAM by each recursive call to a function and in time used by the function call.

3. Recursion in C++

Most programming students will recognize the following function, which uses an iterative loop to compute and return the sum of the integers from 0 to n .

```
int sumInts(int n) {
    int sum = 0;
    for (int x=1; x<=n; x++) sum += x;
    return sum;
}
```

In recursive terms, the function *sumInts* can be defined as follows:

$$f(0) = 0$$

$$f(n) = f(n-1) + n,$$

n is any integer greater than 0

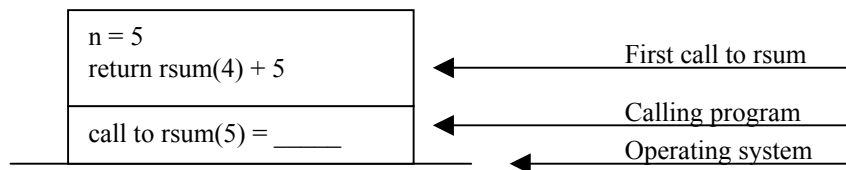
This recursive definition can be expressed in a C++ function as follows:

```
int rsum(int n) {
    if (n < 1) return 0;
    return rsum(n-1) + n;
}
```

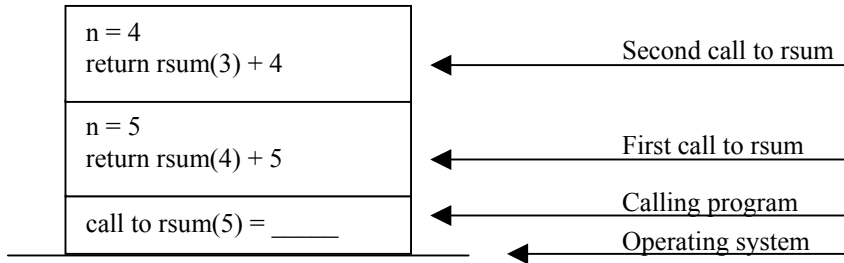
When writing a recursive function, providing a means for the recursion to halt is of primary importance. The *if* control structure insures that the recursion will halt and return 0 when the value of n is reduced to 0. If n has a value greater than 0, the *if* will fail and *rsum* will be called again with n reduced in value, which will eventually cause the recursion to halt.

What follows is an illustration of how *rsum* sums the numbers from 0 to n , where n has the initial value of 5.

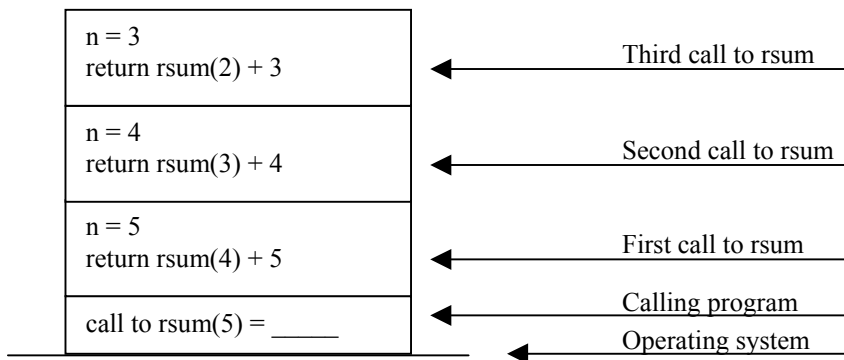
1. With the initial call of *rsum*(5), space in RAM is allocated for the code and variables of *rsum*. Since n has the value of 5, which is greater than 1, the statement of the *else* will cause this copy of *rsum* to attempt to evaluate the expression *rsum*($n-1$)+ n and return the result to the calling function. This will not be possible because the value of *rsum*($n-1$) is not known and causes a recursive call to function *rsum*.



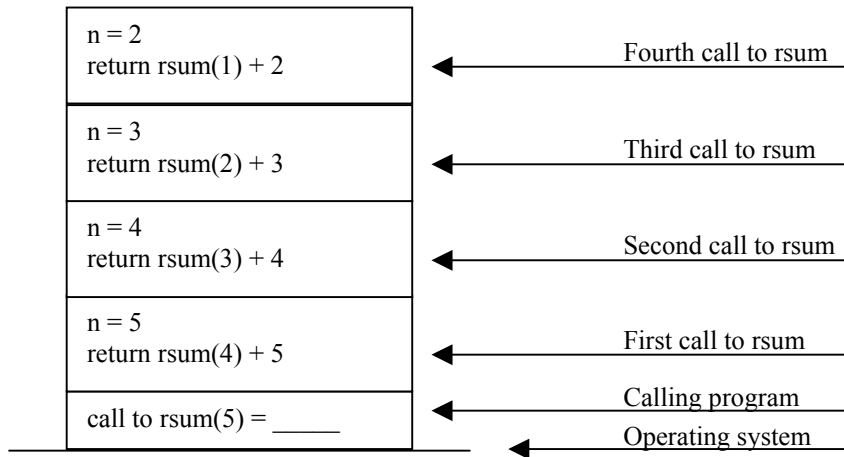
2. With the first recursive call of $rsum(4)$ (the second actual call), space in RAM is allocated for the code and variables of $rsum$ again. Since n has the value of 4, which is greater than 1, the *if* will fail, which will cause this copy of $rsum$ to attempt to evaluate the expression $rsum(n-1)+n$ and return the result to the calling function. This will not be possible because the value of $rsum(n-1)$ is not known and causes a recursive call to function $rsum$.



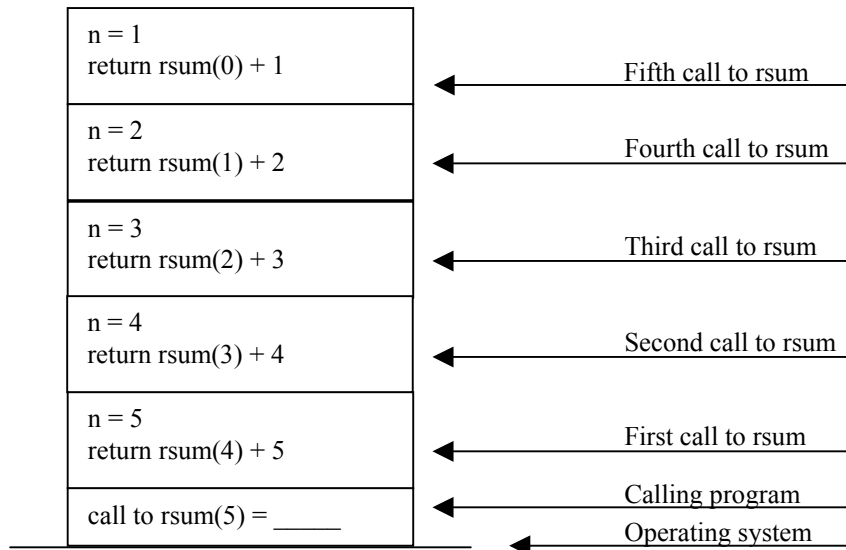
3. With the second recursive call of $rsum(3)$ (the third actual call), space in RAM is allocated for the code and variables of $rsum$ again. Since n has the value of 3, which is greater than 1, the *if* will fail, which will cause this copy of $rsum$ to attempt to evaluate the expression $rsum(n-1)+n$ and return the result to the calling function. This will not be possible because the value of $rsum(n-1)$ is not known and causes a recursive call to function $rsum$.



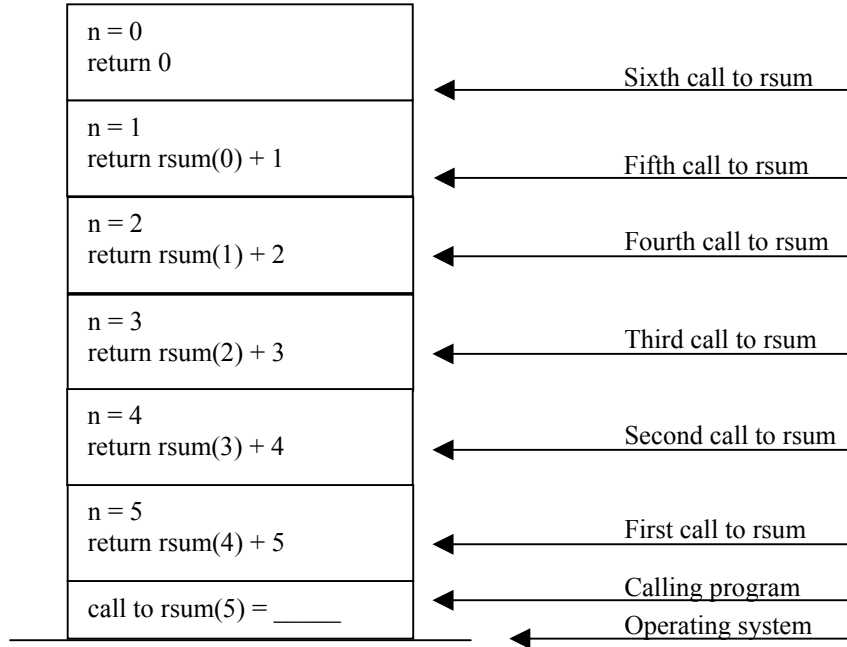
4. With the third recursive call of $rsum(2)$ (the fourth actual call), space in RAM is allocated for the code and variables of $rsum$ again. Since n has the value of 2, which is greater than 1, the *if* will fail, which will cause this copy of $rsum$ to attempt to evaluate the expression $rsum(n-1)+n$ and return the result to the calling function. This will not be possible because the value of $rsum(n-1)$ is not known and causes a recursive call to function $rsum$.



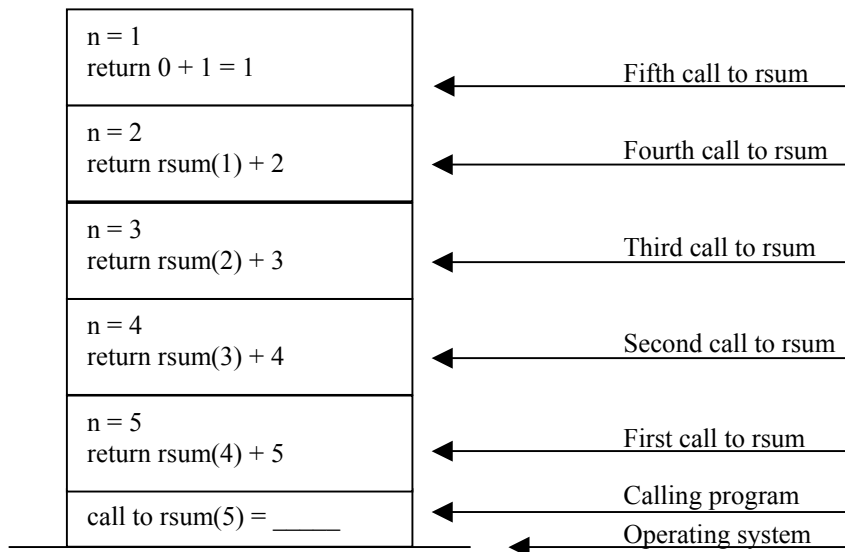
5. With the fourth recursive call of $rsum(1)$ (the fifth actual call), space in RAM is allocated for the code and variables of $rsum$ again. Since n has the value of 1, which is equal to 1, the *if* will fail, which will cause this copy of $rsum$ to attempt to evaluate the expression $rsum(n-1)+n$ and return the result to the calling function. This will not be possible because the value of $rsum(n-1)$ is not known and causes a recursive call to function $rsum$.



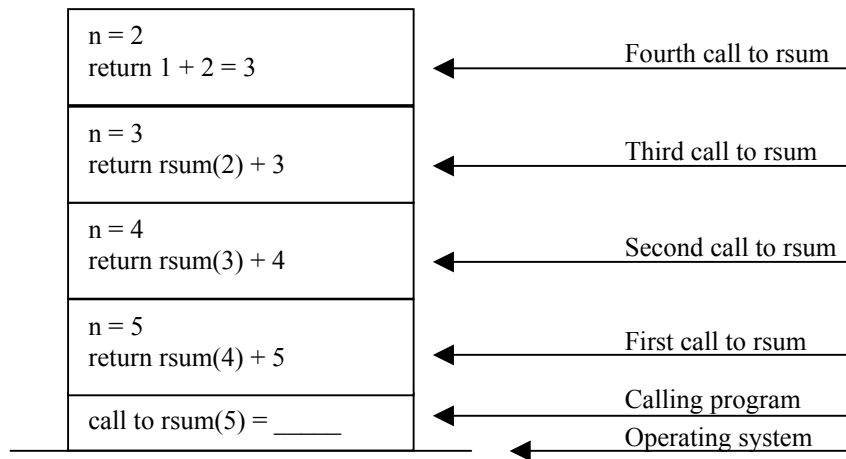
6. With the fifth recursive call of $rsum(0)$ (the sixth actual call), space in RAM is allocated for the code and variables of $rsum$ again. Since n has the value of 0, which is less than 1, the *if* will succeed, which will cause the return of 0.



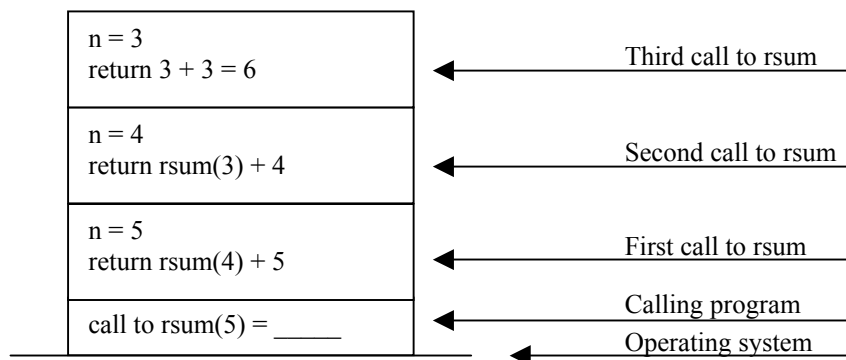
7. With the return of 0 from the fifth recursive call of $rsum(0)$ (the sixth actual call) to the fourth recursive call (the fifth actual call), the space used for the fifth recursive call is returned to RAM. This allows the expression in the fourth recursive call to finally be evaluated the answer returned.



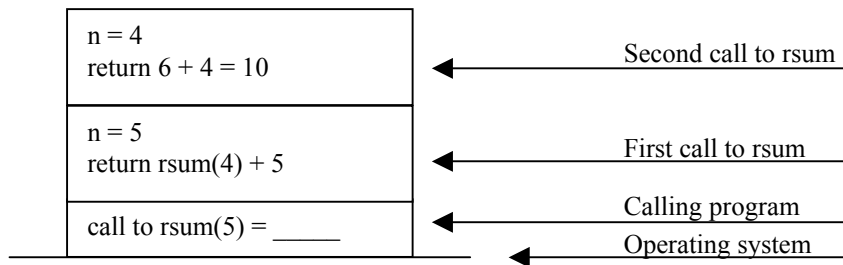
8. With the return of 1 from the fourth recursive call of $rsum(1)$ (the fifth actual call) to the third recursive call (the fourth actual call), the space used for the fourth recursive call is returned to RAM. This allows the expression in the third recursive call to finally be evaluated the answer returned.



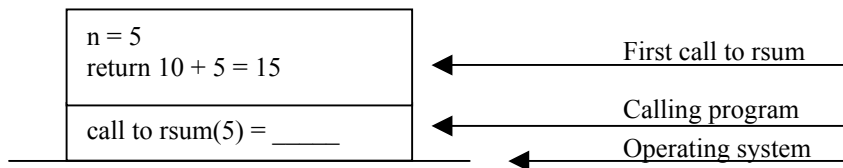
9. With the return of 3 from the third recursive call of $rsum(2)$ (the fourth actual call) to the second recursive call (the third actual call), the space used for the third recursive call is returned to RAM. This allows the expression in the second recursive call to finally be evaluated the answer returned.



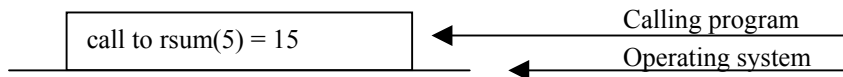
10. With the return of 6 from the second recursive call of $rsum(3)$ (the third actual call) to the first recursive call (the second actual call), the space used for the second recursive call is returned to RAM. This allows the expression in the first recursive call to finally be evaluated the answer returned.



11. With the return of 10 from the first recursive call of $rsum(4)$ (the second actual call) to the first call to $rsum$, the space used for the first recursive call is returned to RAM. This allows the expression in the first call to $rsum$ to finally be evaluated the answer returned.



12. With the return of 15 from the first call of $rsum(5)$, the original expression in the calling program receives the answer to the expression.



Programming Exercise 37.1

Write and use a recursive function to compute the factorial of any integer greater than or equal to 0.

Programming Exercise 37.2

Write and use a recursive function that receives two integer numbers (n and m), then returns the sum of the integers from n to m .

Exercise 37.1

Illustrate the recursive process when the function of programming exercise 37.2 is called with values of 3 and 6.

Programming Exercise 37.3

Write and use a recursive function to compute the n th power of m . The number m is a floating point number while n is an integer.

Programming Exercise 37.4

Write and use a recursive function to compute the n^{th} Fibonacci number.

Exercise 37.2

Illustrate the recursive process when the function of programming exercise 37.4 is called with value 7.