6 Recursion

One of the basic rules for defining new objects or concepts is that the definition should contain only terms that have already been defined or that are obvious. Therefore, an object which is defined in terms of itself is a serious violation of this rule. However, there are many programming concepts that define themselves. As it turns out, formal restrictions imposed on definitions such as existence and uniqueness are satisfied and no violation of the rules takes place. Such definitions are called *recursive definitions* and are used primarily to define infinite sets. When defining such a set, giving a complete list of elements is impossible, and for large finite sets, it is inefficient. Thus, a more efficient way has to be devised to determine if an object belongs to a set.

A recursive definition consists of two parts. In the first part, called the *ground case*, the basic elements that are the building blocks of all other elements of the set are listed. In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects. For example, to construct the set $\mathbb{N}$ of decimal integers, the digits 0 to 9 are all singled out as basic elements (ground cases), while other values are constructed using a recursive rule:

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \in \mathbb{N}$;
2. if $n \in \mathbb{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbb{N}$;
3. there are no other objects in the set $\mathbb{N}$

Controlling Simple iteration with Recursion

Recursive definitions are frequently used to define functions and sequences of numbers. For instance, the factorial of a non-negative integer n, written $n!$, is defined as

$$n! = 1 * 2 * 3 * ... * (n-1) * n$$

However, we can also define factorial *recursively* as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (ground case)} \\ n * (n-1)! & \text{if } n > 0 \text{ (inductive step)} \end{cases}$$

This latter definition leads to the following segment of C++ code in which the function factorial is called recursively.

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

The important point to stress here is that, to use recursion appropriately, we must have a termination condition to avoid an infinite series of recursive calls.

6.0.1 Function Calls and Recursion Implementation

What happens when a function is called? If the function has formal parameters, they have to be initialized to the values passed as actual parameters. In addition, the system has to know where to resume execution of the program after the function has finished. The function can be called by other functions or by the main program (the function main()). The information indicating where it has been called from has to be remembered by the system. This could be done by storing the return address in main memory in a place set aside for return address, but we do not know in advance how much space might be needed, and allocating too much space for that purpose alone is not efficient.

For a function call more information has to be stored than just a return address. Therefore, dynamic allocation using the run-time stack is a much better solution. But what information should be preserved when a function is called? First, local variables must be stored. If function f1() which contains a declaration of a local variable x calls function f2() which also locally declares the variable x, the system has to make a distinction between these two variables x. If f2() uses a variable x, then its own x is meant, if f2() assigns a value to x, then x belonging to f1() should be left unchanged. When f2() is finished, f1() can use the value assigned to its own x before f2() was called. How does the system make a distinction between these two variables x? We could require that all variable names are unique, but, apart from this being poor software engineering, it does not solve the problem for functions that recursively call themselves.

The state of each function, including main(), is characterized by the contents of all arithmetic variables, by the values of the function's parameters, and by the return address indicating where to restart its caller. The data area containing all this information is called a *activation record* and is allocated on the run-time stack. An activation record exists for as long as a function owning it is executing. This record is a private pool of information for the function. Activation records usually have a short lifespan because they are automatically allocated at function entry and deallocated upon exiting. Only the activation record of main() outlives every other activation record. An activation record usually contains the following information:

Parameters and local variables	Return address	Return value

In order to illustrate the status of a run-time stack when a function is called, suppose that main() calls function f1(), f1() calls f2(), and f2() in turn calls f3(). While this call of f3() is being executed, the state of the run-time stack would be as follows (shown top of stack down):

Activation record of f3()		
Parameters and local variables	Return address	Return value
Activation record of f2()		
Parameters and local variables	Return address	Return value
Activation record of f1()		
Parameters and local variables	Return address	Return value
Activation record of main()		

By the nature of stacks, when the activation record for f3() is popped, f2() resumes execution and now has free access to the private pool of information necessary for reactivation of its execution. On the other hand, if f3() happens to call another function f4(), then the run-time stack increases its height since the activation record for f4() is created on the stack and the activity of f3() is suspended.

Creating an activation record whenever a function is called allows the system to handle recursion properly. Consider the following segment of code which computes 4! recursively.

```
int main()
{
      int      a, n;

      n = 4;
      a = factorial(n);
      cout << n <<”! = ” << a << endl;
      return 0;
}

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

After the first call of factorial function (from the main function) the status of run-time stack is as follows:

Activation record of factorial(4)		
Parameters and local variables	Return address	Return value
Activation record of main()		

Since the parameter n is not equal to 0 the function calls itself recursively. This process repeats until the parameter $n = 0$ is passed to the function, resulting in the following stack:

Activation record of factorial(0)		
Parameters and local variables	Return address	Return value
Activation record of factorial(1)		
Parameters and local variables	Return address	Return value
Activation record of factorial(2)		
Parameters and local variables	Return address	Return value
Activation record of factorial(3)		
Parameters and local variables	Return address	Return value
Activation record of factorial(4)		
Parameters and local variables	Return address	Return value
Activation record of main()		

In this step the recursive termination condition is reached and therefore the topmost record must be popped. The return value is 1 and the stack's status is:

Activation record of factorial(1)		
Parameters and local variables	Return address	1 * 1
Activation record of factorial(2)		
Parameters and local variables	Return address	Return value
Activation record of factorial(3)		
Parameters and local variables	Return address	Return value
Activation record of factorial(4)		
Parameters and local variables	Return address	Return value
Activation record of main()		

Each level of recursion multiplies the returned value (from the recursive call to the function) by its parameter and returns the result to the caller of the function. That is, the next status of the stack would be:

Activation record of factorial(2)		
Parameters and local variables	Return address	2 * 1 * 1 = 2
Activation record of factorial(3)		
Parameters and local variables	Return address	Return value
Activation record of factorial(4)		
Parameters and local variables	Return address	Return value
Activation record of main()		

and the status of the stack before returning control to the main function would be:

Activation record of factorial(4)		
Parameters and local variables	Return address	$4 * 3 * 2 * 1 * 1 = 24$
Activation record of main()		

6.0.2 Nested Recursion

The recursive algorithms we have examined so far have the property that, at each level of recursive execution of the algorithm, at most one recursive call will be made. The pattern of operations on the run-time stack for such recursive algorithms is that a series of stack frames is pushed, a recursive termination condition is reached, and then all stack frames are successively popped until we return to the execution level of the main program. More complex recursive algorithms involve multiple recursive calls at each level of execution. Correspondingly, the pattern of operations on the stack will no longer be a series of uninterrupted pushes followed by a series of uninterrupted pops.

An example of more complex recursive patterns is the *Towers of Hanoi Problem*, which involves moving a collection of n stone disks from one pillar, designated as pillar A, to another, designated as pillar C. The relative ordering of the disks on pillar A have to be maintained as they are moved to pillar C. Additionally, the following rules in moving disks must be observed:

1. Only one disk may be moved at a time.
2. No larger disk may ever be placed on top of a smaller disk on any pillar.
3. A third pillar may be used as an intermediate to store one or more disks while they were being moved from their original source A to their destination C.

A recursive solution to this problem could be:

1. If $n = 1$, move the disk from A to C.
2. If $n = 2$, move the top disk from A to B. Then move the remaining disk from A to C. Then move the first disk from B to C.

3. If $n = 3$, call on the technique already established in step 2 to move the first two disks from A to B using C as intermediate. Then move the third disk from A to C. Then use the technique in step 2 to move the first two disks from B to C using A as an intermediate.

   ⋮

n. For general n, use the technique in the previous step to move $n-1$ disks from A to B using C as an intermediate. Then move one disk from A to C. Then use the technique in the previous step to move $n-1$ disks from B to C using A as an intermediate.

Notice that this technique for solving the Towers of Hanoi describes itself in terms of a simpler version of itself. That is, it describes how to solve the problem for n disks in terms of a solution for $n-1$ disks. In general, this is the technique to solve any problem recursively.

Another important example of nested recursion is the Ackermann's function which is defined as follows:

$$A(n,m) = \begin{cases} m+1 & \text{if } n = 0, \\ A(n-1,1) & \text{if } n > 0 and m = 0, \\ A(n-1, A(n, m-1)) & \text{otherwise.} \end{cases}$$

This function is interesting because of its remarkably rapid growth. It grows so fast that it is guaranteed no to have a representation by a formula that uses arithmetical operations such as addition, multiplication, and exponentiation. To illustrate the rate of growth of the Ackermann function, we need only show that $A(4,1) = 2^{65536} - 3$, which exceeds even the number of atoms in the universe (which is 10^{80} according to current theories).

6.0.3 Analyzing Recursive Functions

An analysis of the time and space efficiency of a recursive algorithm is dependent on two factors. The first of these is the depth, that is, number of levels, to which recursive calls are made before reaching the recursive termination condition. Clearly, the greater the depth, the greater the number of stack frames that must be allocated and the less space efficient the algorithm becomes. The second factor affecting efficiency analyses (particularly time efficiency) of recursive algorithms is the number of recursive calls made by each level of the recursion.

We observe two principles for carrying out the analysis of recursive algorithms:

Space efficiency of a recursive algorithm: Since a stack frame must be allocated at each level of recursive execution, the space efficiency of a recursive algorithm will be proportional to the deepest level at which a recursive call is made for a particular set of values, that is, the deepest level obtained in a run-time trace of the algorithm.

Time efficiency of a recursive algorithm: Since processing time is associated with each recursive call, the time efficiency of a recursive algorithm will be proportional to the sum of the times spent processing at each level.

Efficiency Analysis of the Factorial Algorithm

Recall that computing the factorial of a non-negative integer n requires to call the factorial function n times. Thus, the space complexity of the factorial function is $O(n)$.

On the other hand, at each call the function requires to apply a multiplication (the factorial function is not a nested recursive function). Therefore, time efficiency of the algorithm is also $O(n)$.

Efficiency Analysis of the Towers of Hanoi Algorithm

In this algorithm, for any $n > 1$ a function call is made. For every extra disk (i.e. increase in the value of n by 1), the number of levels in the run-time stack increases by 1, so the space efficiency of the algorithm is $O(n)$. However, each level of call generates two further calls, so the time efficiency of the algorithm is $O(2^n)$.

Efficiency Analysis of Recursive Binary Search

Consider the following recursive version of the binary search algorithm.

```
int binary_search (list_type list, int low, int high, key_type target)
{
    int      middle;

    if (low > high)
```

```
            return -1;
        else {
            middle = (low + high) / 2;
            if (list[middle] == target)
                return middle;
            else if (list[middle] > target)
                return binary_search(list, low, middle - 1,
                target);
            else
                return binary_search(list, middle + 1, high,
                target)
            }
}
```

In this algorithm the number of operations in each call, excluding the recursive call, if any, is one, since we compare the target item to the data at the middle position. Hence the time efficiency of the algorithm is proportional to the number of levels in the run-time stack trace. The number of levels (calls) is $\log_2 n + 1$. The efficiency of the recursive version of the binary search algorithm is therefore $O(\log_2 n)$.

6.1 Trial-and-Error Backtracking

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ascertain that such a return is possible and that all paths can be tried. This technique is called *backtracking* and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position which offers the possibilities for successfully solving the problem. This technique is used in artificial intelligence (AI) and one of the well known problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other. The rules of chess say that a queen can take another piece if it lies on the same row, on the same column, or on the same diagonal as the queen. To solve this problem, we try to put the first queen on the board, then the second, so that it cannot take the first, then the third, so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a non-conflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion which is a natural implementation of backtracking. Pseudocode for this backtracking algorithm is as follows:

```
put_queen(row)
    for every position col on the same row
        if position col is not in conflict
            place a queen in position col;
            if (row < 8)
                put_queen(row + 1);
            else success;
        remove the queen from position col;
```

This algorithm finds all possible solutions without regarding that some of them are symmetrical.

Exercises (Tutorial 10)

1. Write a recursive function to compute the binomial coefficient according to the definition

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \quad \text{or} \quad k = n, \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}$$

2. Write a recursive function to add the first n terms of the series

$$1 + \frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} + \cdots$$

3. Write a recursive function GCD(n,m) that returns the greatest common divisor of two integers. According to Euclid's algorithm,

$$GCD(n,m) = \begin{cases} m & \text{if} \quad m \leq n \quad \text{and} \quad n\%m = 0, \\ GCD(m,n) & \text{if} \ \ n < m, \\ GCD(m, n\%m) & \text{otherwise} \end{cases}$$

4. Write a recursive function to print out a nonnegative integer in binary (do not use bit-wise operations).

5. Write a program to solve the eight queens problem.