Recursion

- *Recursion* means calling a function inside itself. Thus, a function which calls itself is named as a *recursive function*. This week, we will compare recursive and non-recursive logic. You will see the advantages of recursion, and learn how to define recursive functions.

- When a recursive function is called, the function knows to solve:

  - only the simplest case

  - or a simpler version of the original problem

- The recursive algorithms will be similar to:

  if this is a simple case

  solve it

  else

  redefine the problem using recursion

- Recursion is frequently used in mathematics. For example, consider the definition of **n!** (**n** factorial) for a nonnegative integer **n**:

  0! = 1
  1! = 1
  n! = n (n - 1)! if n > 1

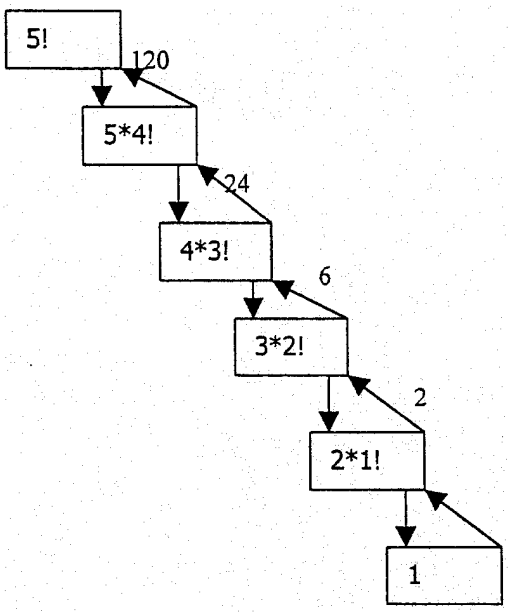
- Thus, for instance

  $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

- Remember that in CTIS 151 we defined a factorial function using a for loop. The main idea in writing a recursive function is to start from the end and go backwards towards the beginning. Thus, we will start at n. The function will terminate and return the result when we reach to 1.

```
double factorialRec(int n)
{
      // if n is 0 or 1, n! is 1
      if (n <= 1)
           return 1;
      // otherwise, n! is n times (n - 1)!
      else
           return ( n * factorialRec (n - 1) );
}
```

- We can represent the execution of this recursive function with the following figure:



- Two important comments about recursion:

1. The recursive process must have a well-defined termination. This termination is referred to as a *stopping state* (or *degenerate case*). In our example, the stopping state is:

```
if (n <= 1)
      fact = 1;
```

2. The recursive process must have well-defined steps that lead to the stopping state. These steps are usually called as *recursive steps*. In our example, these steps are:

```
      fact = n * factorialRec(n - 1);
```

- Notice that, in the recursive call, the parameter is simplified toward the stopping state. Thus, `n - 1` guarantees that `n` will reach to the value 1 at some time, and the stopping state will be reached.

Example: Define a recursive function that finds the sum of the elements of a double array.

```
/* Sum of elements in an array using recursion */
double arraySumRec(double ar[], int size)
{
      if (size == 1)
           return ar[0];
      else
           return ( ar[size - 1] + arraySumRec(ar, size - 1) );
}
```

- Recursive calls take time, and use more memory. Thus, recursion is inefficient than iteration. In addition, recursion is usually difficult to understand for beginning programmers. Then, why do we use recursion?

  - Sometimes, the recursive solution of a problem may be very apparent when compared to the iterative solution. Factorial is a good example of such a situation.

  - Some recursive solutions may be very short compared to iterative solutions.

  - Recursion is a valuable tool when working with data structures, such as stacks, queues, and linked lists.

Example: Define a recursive function that finds the n^{th} term of the Fibonacci's sequence. The Fibonacci's sequence is

1, 1, 2, 3, 5, 8, 13, 21, …

- Thus,

fibonacci (1) = 1
fibonacci (2) = 1
fibonacci (n) = fibonacci (n-2) + fibonacci (n-1) if n > 2

- This definition makes the recursive solution very apparent:

```
int fibonacci(int n)
{
      // The first and second terms of the sequence are 1
      if (n <= 2)
           return(1);
      // other terms are the sum of the two previous terms
      else
           return( fibonacci(n - 2) + fibonacci(n - 1) );
}
```

Home Exercise: Write the iterative solution.

Example: Define a recursive function for positive integer multiplication using addition.

For instance: 5 * 4 = 5 + 5 + 5 + 5 = 5 + (5 * 3)

```
/* Recursive integer multiplication using addition */
int multiply(int m, int n)
{
      /* if the second integer is 1, the result is equal to
          the first one */
      if (n == 1)
          return m;
      /* otherwise, the result is m plus the product of m
          and n-1 */
      else
          return ( m + multiply(m, n - 1) );
}
```

- If you want to multiply 2 and 100, how would you call this function? As `multiply(2, 100)` or as `multiply(100, 2)`?
- What about if `m` or `n` is 0? If `m` is 0, e.g., `multiply(0, 100)`, the result will be calculated correctly, but the function will be called 100 times unnecessarily. If `n` is 0, e.g., `multiply(100, 0)`, the function will never finish. We can add another stopping state to our function to handle those cases:

```
// if any of the integers is zero, the result is zero
if ( m == 0 || n == 0 )
    return 0;
else if (n == 1)
...
```

- What is the result of `multiply(-100, 2)`?
- What about `multiply(100, -2)` or `multiply(-100, -2)`? They cause the function to enter infinite loop. How can we change the function so that it will also work for negative `n`?

```
int multiply(int m, int n)
{
   if (m == 0 || n == 0)
      return 0;
   else if (n < 0)
      return (-multiply(m, -n));
   else if (n == 1)
      return m;
   else
      return (m + multiply(m, n - 1));
}
```

Example: Define a recursive function for integer division using subtraction.

```
/* Division by subtraction using recursion */
int divide(int number, int divisor)
{
      /* if the number is less than the divisor, the result
         is zero */
      if (number < divisor)
          return 0;
      /* otherwise, the result is one:
         one plus (number - divisor) / divisor */
      else
          return ( 1 + divide(number - divisor, divisor) );
}
```

- Let's modify the function so that it also returns the remainder of division as an output parameter:

```
/* Division by subtraction using recursion */
int divide(int number, int divisor, int *rem)
{
      /* if the number is less than the divisor, the result
         is zero, the remainder is the number */
      if (number < divisor) {
          *rem = number;
          return 0;
      }
      /* otherwise, the result is one:
         one plus (number - divisor) / divisor */
      else
          return ( 1 + divide(number-divisor, divisor, rem) );
}
```

- Notice that the above function works correctly only when both integers are positive. Otherwise, it either gives incorrect results or enters into an infinite loop.

Home Exercise: Modify the `divide` function so that it works in any case.

Example: Define a recursive function for power operation (x^y) using multiplication. `x` may be double, `y` is integer (positive or negative).

For instance: $0.5^4 = 0.5 * 0.5 * 0.5 * 0.5 = 0.5 * 0.5^3$

```
double powerRaiser(double base, int power)
{
     if (power == 0)
          return 1;
     else if (power < 0)
          return ( 1 / powerRaiser(base, -power) );
     else
          return ( base * powerRaiser(base, power - 1) );
}
```

Example: Define a recursive function that counts the number of occurrences of a given character in a given string.

```
int countRec(char ch, char *str)
{

   if (*str == '\0')
      return 0;
   else if (*str == ch)
      return( 1 + countRec(ch, str + 1) );
   else
      return( countRec(ch, str + 1) );
}
```

Home Exercise: Write the recursive version of the `strlen` function.

- Upto now, all recursive functions we defined were functions with a single result. The following is such a void function. Trace it for different `n` values and try to understand:

```
void func(int n) {
     if (n < 2)
          printf("%d", n);
     else {
          func(n / 2);
          printf("%d", n % 2);
     }
}
```

Example: Take `n` words as input and display them in reverse order on separate lines.

Input:		Output:	
	the		events
	course		human
	of		of
	human		course
	events		the

```
void reverseInputWords(int n) {
     char word[20]; // local variable for storing one word
     if (n == 0);   // no more words
     else {
          scanf("%s", word);
          reverse_input_words(n - 1);
          printf("%s\n", word);
     }
}
```

- It is also possible to write it as:

```
void reverseInputWords(int n) {
     char word[20]; // local variable for storing one word
     if (n != 0) {
          scanf("%s", word);
          reverseInputWords(n - 1);
          printf("%s\n", word);
     }
}
```

Home Exercise: Rewrite the `divide` function as a void function.

Recursive Binary Search

- Remember that, in the previous chapter, we defined the binary search function as follows:

```
int binarySearch(double ar[], int top, int bottom, double num)
{
      int middle;
      while (top <= bottom) {
          middle = (top + bottom ) / 2;
          if (num == ar[middle])
              return (middle);
          else if (ar[middle] > num)
              bottom = middle - 1;
          else
              top = middle + 1;
      }
      return (-1);
}
```

- In each repetition of the while loop, we make a binary search in only a certain part of the array. Therefore, we can call the binary search function with only that part of the array.

```
int binarySearch(double ar[], int top, int bottom, double num) {
    int middle;
    /* if top exceeds bottom, thus, if there are no more elements
       to check, return -1, because the number is not found */
    if (top > bottom)
       return (-1);
    else {
       middle = (top + bottom) / 2; // Find the middle position
       // If the number is equal to the element in the middle
       if (num == ar[middle])
          // Return the middle position
          return (middle);
       // If the number is less than the element in the middle
       else if (num < ar[middle])
          // Make a search in the first half, and return its result
          return (binarySearch(ar, top, middle - 1, num));
       // If the number is greater than the element in the  middle
       else // Make a search in the second half, and return its result
          return (binarySearch(ar, middle + 1, bottom, num));
    }
}
```

Home Exercise: Write a recursive bubble sort function and use it in a main program.