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 original problem will be divided into two pieces:
  - a piece that the function knows how to do
  - a piece that the function does not know how to do

- The second part must resemble the original problem, but should be a slightly simpler version. In order for the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, at the end converging on the simplest (base) case.

- The recursive algorithms will be similar to:

if this is a simple
case solve it

else
redefine the problem using recursion

- Recursion is used when the steps in the function will be repeated continuously until a certain limit. We can also solve such problems by the help of loops, i.e., iteratively.

- 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 4! = 5 4 3! = 5 4 3 2! = 5 4 3 2 1! = 5 4 3 2 1

- Remember that in CENG 103 we defined a factorial function using a for loop as follows:

```
double factorial(int n)
{
     int i;
     double fact;

     fact = 1;
     for (i = n; i >= 2; i--)
         fact = fact * i;
     return (fact);
}
```

- 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 factorial(int n)
{
     double fact;   // result

     // if n is 0 or 1, n! is
     1 if (n <= 1)
         fact = 1;
     // otherwise, n! is n times (n -
     1)! else
         fact = n * factorial (n - 1);
     // Return the result, thus
     n! return (fact);
}
```

- To understand how this recursive function works, suppose that it is called somewhere in the main as

```
factorial(5)
```

- Since 5 > 1, the else part of if … else will be done, and we will first have `fact = 5 * factorial(4);`

- At this stage, notice that `factorial(4)` must be computed. Thus, the function is called once more, this time with parameter 4. This call results

```
fact = 4 * factorial(3);
```

- Now `factorial(3)` must be computed. Thus, the function is this time called with parameter 3. This call results

```
fact = 3 * factorial(2);
```

- To compute `factorial(2)`, the function is called once more, with parameter 2. This call results

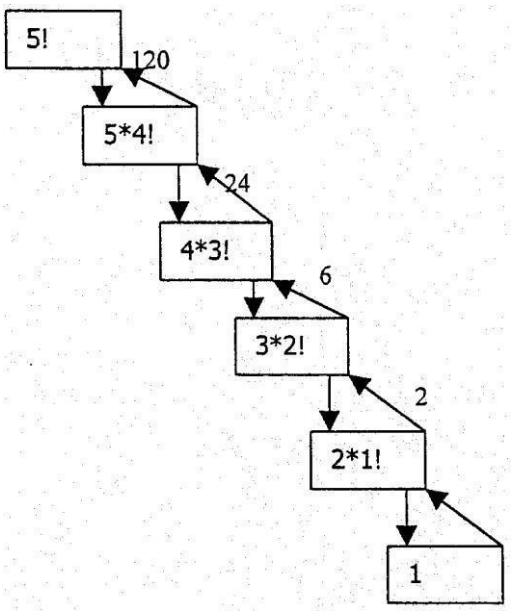
```
fact = 2 * factorial(1);
```

- Since `n` is 1, the call `factorial(1)` will return `fact` as 1, and the end of the recursion is reached. Now, the result of each call can be calculated by substituting the result of the next call, as follows:

```
fact = 2 * 1;
fact = 3 * 2;
fact = 4 * 6;
fact = 5 * 24;
```

and the result is returned as 120.

- We can represent it 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 * factorial(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 function that finds the sum of the elements of a one-dimensional array.

- This function can be written iteratively, using a for loop, as follows:

```
double array_sum(double ar[], int size)
{
     int k;
     double result = 0;
     /* Add each element of the array to the
     result */ for (k = 0; k < size; k++)
         result = result + ar[k];
     return (result);
}
```

- The same function can also be written recursively. We can formulate the sum of the array elements as

  sum of all elements = last element + sum of the other elements

- Therefore we can write the recursive step as

```
result = ar[size - 1] + array_sum(ar, size - 1);
```

- What about the stopping state? If we reach to the first element of the array, that means that there are no other elements to sum up, thus `result` is equal to the value of the first element. Hence, we can write the stopping state as

```
if (size == 1)
     result = ar[0];
```

- Now, we can define our recursive function as.

```
/* Sum of elements in an array using recursion
*/ double array_sum(double ar[], int size) {

     double result;
     if (size == 1)
         result = ar[0];
     else
         result = ar[size - 1] + array_sum(ar, size - 1);
     return (result);
}
```

- To understand how this recursive function works, suppose that it is called somewhere in the main as

```
array_sum(a, 3)
```

- Assume that `a` is an array containing three elements, as **2.5**, **1.2** and **0.5**.
- Since 3 == 1 is false, the else part of if … else will be done, and we will first have

```
result = a[3-1] + array_sum(a, 3-
1); result = 0.5 + array_sum(a, 2);
```

- At this stage, notice that `array_sum(a, 2)` must be computed. Thus, the function is called once more, this time with parameters `a` and 2. This call results

```
result = a[2-1] + array_sum(a, 2-
1); result = 1.2 + array_sum(a, 1);
```

- Since `size` is 1, the call `array_sum(a, 1)` will return `result` as `a[0]`, thus 2.5, and

  the end of the recursion is reached. Now, the result of each call can be calculated by substituting the result of the next call, as follows:

```
result = 1.2 + 2.5;
result = 0.5 + 3.7;
```

  and the function returns 4.2.

- As in the previous examples, any problem that can be solved recursively, can also be solved non-recursively, thus by using iteration (loops).

- 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.

- Let's compare recursion and iteration:

  - Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.

  - Both recursion and iteration involve repetition: Iteration explicitly uses a repetition structure; recursion achieve repetition through repeated function calls.

  - Iteration and recursion each involve a termination test: Iteration terminates when the loop condition fails; recursion terminates when a stopping state is recognized.

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

- Assume that we want to find the 4th term of the Fibonacci sequence. Since `n` = 4, the result will be

```
result = fibonacci(2) + fibonacci(3);
```

- Thus, we need to find the second and third terms of the sequence. The result of the call `fibonacci(2)` will be returned as 1. When the function is called with the parameter 3, the result will be

```
result = fibonacci(1) + fibonacci(2);
```

- The result of each of these calls will be calculated as 1 and the result of the call `fibonacci(3)` will be calculated and returned as 1 + 1 = 2. Then, the result of `fibonacci(4)` will be calculated as 1 + 2 = 3.

- Notice that, this function is very easy and very short when it is written recursively. However, the iterative solution of it is longer and more complex.

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

- 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)
      ans = 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`?

```
/* otherwise, the result is m plus the product of
    m and abs(n) - 1 */
else
    ans = m + multiply(m, abs(n) - 1);
// if the second integer is negative, negate the
answer if (n < 0)
    ans = -ans;
return (ans);
```

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

- We need to count how many times we have to subtract the divisor until the remaining number is less than the divisor itself. For instance, if we want to divide 12 by 3:

```
12 - 3 = 9
       9 - 3 = 6
             6 - 3 = 3
                   3 - 3 = 0
```

We made 4 subtractions, thus the answer is 4. If we divide 27 by 7:

```
27 - 7 = 20
       20 - 7 =  13
              13 - 7 =  6
```

We made 3 subtractions, thus the answer is 3.

- Therefore, our stopping state is

```
if (number < divisor)
     result = 0;
```

Recursive step is

```
result =  1 + divide(number - divisor, divisor);
```

- The function is

```
/* Division by subtraction using recursion
*/ int divide(int number, int divisor) {

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

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

- 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 power_raiser(double base, int power)
{
     double ans;
     if (power == 0)
         ans = 1;
     else
         ans = base * power_raiser(base, abs(power) - 1);
     // if the power is negative, take 1
     / ans if (power < 0)
         ans = 1 / ans;
     return (ans);
}
```

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

- In the recursive step we will call the function each time by removing the first character of the remaining string. If the character we remove is the same as the one we are counting, we will add 1 to the result of the function.

- We will reach to the stopping state when we reach to a string with no characters, i.e., the end of the original string. We will return 0 in that case, since there are no more characters.

```
int count(char ch, char *str)
{
   int cnt;
   if (*str == '\0')
      cnt = 0;
   else if (*str == ch)
      cnt = 1 + count(ch, str + 1);
   else
      cnt = count(ch, str + 1);
   return (cnt);
}
```

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

- Upto now, all recursive functions we defined were functions with a single result. It is also possible to write a void recursive function. The following is such a void function. Let's trace it for different `n` values and try to understand what it does:

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

- If we call it with 3, it will call itself with 3 / 2, thus with 1. Since 1 < 2, this call will display 1 and return back to the previous call, in which `n` was 3. There, 3 % 2, thus 1 will be displayed, and it will return back to the main. Therefore, the output is **11** if `n` is 3.

- If we call it with 5, it will call itself with 5 / 2, thus with 2. Since 2 is not less than 2, the function will be called once more with 2 / 2, thus with 1. Since 1 < 2, this call will display 1 and return back to the previous call. In that call `n` was 2, so 2 % 2, thus 0 will be displayed, and it will return back to the previous call. There, `n` was 5, so 5 % 2, thus 1 will be displayed, and it will return back to the main. Therefore, the output is **101** if `n` is 5. Can you understand the relation between the output and `n`?

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

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

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

- Notice that, this function could also be defined recursively. Because, 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 binary_search(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 (binary_search(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 (binary_search(ar, middle + 1, bottom, num));
    }
}
```

- Notice that, in the above function, we have two stopping cases, one for not finding, one for finding the number. We also have two recursive steps: one for continuing the search in the first half, one for continuing in the second half.

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