Lab objective

- The goal of this lab is to understand the concept of recursion and practice it by creating a number of recursive functions and compare them with their interactive equivalents.
- Recursion is a technique of defining a process in term of itself. More specifically, in terms of programming, recursive function is a function that calls itself.
- Recursion is often a substitution for an iterative solution

Step 1) To begin with, lets create the following C++ program and place it in a file called main.cpp

```cpp
#include <iostream>
using namespace std;

void main()
{
    system("pause");
}
```

Step 2: Let’s start with a simple recursive function that implements a factorial function. Factorial is defined as:

\[
\text{factorial} (n) = \begin{cases} 
1, & \text{if } n = 0 \\
= n \times \text{factorial} (n-1), & \text{if } n > 0 
\end{cases}
\]

To implement this function in C++ we can add the following code to the program above. (make sure to add a forward declaration for the function.)

```cpp
int factorial_recursive(int value)
{
    if (value == 0)
        return 1;
    else
        return(value * factorial_recursive(value -1));
}
```

Note that the code is essentially identical the definition of the function shown in the box above.
Now compare the **recursive code (above)** with the **iterative version (below)** that solves the same problem.

```c
int factorial_iterative(int value) {
    int result=1;
    for(int i=value; i >=1; i--) {
        result = result * i;
    }
    return(result);
}
```

Copy the above iterative version function to the bottom of your program, and then add the code below to your main() main function.

```c
int value, result;
value = 5;
result = factorial_iterative(value);
cout << "Factorial of " << value << " using (Iterative Calculation) is: " << result << endl;
result = factorial_recursive(value);
cout << "Factorial of " << value << " using (Recursive Calculation) is: " << result << endl;
```

**Screenshot 1:** Run the program and include a Screen shot of the code and output after this step.

```
Factorial of 5 using (Iterative Calculation) is: 120
Factorial of 5 using (Recursive Calculation) is: 120
```

To better understand the recursive solution, let's **modify** the recursive version as follows:

```c
int factorial_recursive(int value) {
    if (value ==0)
        return 1;
    else {
        if (debug == true) cout << value << "  *  factorial(" << value-1 << "); is called" << endl;
        return(value * factorial_recursive(value - 1));
    }
}
```

**Screenshot 2:** Run the program and include a Screen shot of the code and output after this step.

Note how `factorial(5)` is causing the following recursive function calls:
Step 3: Now, let’s look at a typical problem that we normally solve using iteration. The problem is to search an array for a value and return the location of the value in array, or return -1, if the value cannot be found.

```c
int find_value_iterative(int array[], int value, int startIndex, int endIndex)
{
    int location = -1;
    for (int i = startIndex; i <= endIndex; i++) {
        if (array[i] == value) { // Value is found.
            location = i; // remember the location
        }
    }
    return(location);
}
```

Can the above problem be solved using a recursive algorithm? Well the answer is YES! What we need to think about is a recursive strategy that follows the principles below:

- Ask yourself the following questions:
  - 1) **Base question**
     - Is there a non-recursive way out of the recursive function, and does the routine work correctly for this “base” case? In other words, can we stop the recursion?
  - 2) **Smaller Caller question**
     - Does each recursive call reduce the original problem? In other words, are we getting closer to a solution?
  - 3) **General Case question**
     - Assuming that the recursive calls are working proper, does the whole function work correctly.

Before we go any further, apply the above to the Factorial problem we just worked on. Can you identify the “**Base question**” in that code? How about the “**Smaller Caller question**”? (See below)

```c
int factorial_recursive(int value)
{
    if (value == 0)
        return 1;
    else
        return(value * factorial_recursive(value - 1));
}
```

Now, let’s try to apply these concepts to the recursive version of the find_value() function.

```c
int find_value_recursive(int array[], int value, int startIndex, int endIndex)
{
    1) I need to have a base case! (where the recursion stops!)
    2) I need to have recursive step (where we reduce the problem and call the function again!)
}
```

So, for **base case**: (non-recursive way of getting out of the function)
• I need to know when I have found what I looking for? And make sure I return out of this step without additional recursion! For example:

If (array[location] == value)  // I have found what I am looking for!
    Return(location);

• I also need to know when, I have exhausted looking through the array. In that case, I also need to return out of this step without additional recursion. For example

If (startIndex == endIndex and I have not found the value)       // return -1 to indicate the absence of value!
    Return(-1);

Now, we also need a recursive step **Smaller Caller question** (recursive function)

• where we call the function again, this time with a smaller array:

    find_value_recursive(array, value, startIndex + 1, endIndex);

Ok, we are ready to take this on! Study the code below then copy to your program:

```c
int find_value_recursive(int array[], int value, int startIndex, int endIndex)
{
    if (array[startIndex] == value) // base case, value is found.
        return (startIndex);  // Return the location
    else {
        if (startIndex == endIndex) // base case, value is not in the list
            return (-1);
        else
            return(find_value_recursive(array, value, startIndex + 1, endIndex));
    }
}
```

Now to test both versions of the “find_value” function, place the following code in your main() function.

```c
int MyData[] = {5, 9, 13, 22, 81, 143, 324, 443, 912};
int startIndex = 0;
int endIndex = sizeof(MyData) / sizeof(MyData[0]) - 1;  // Get the upperbound index of the array
int location;

// try the iterative version!
value = 22;
location = find_value_iterative(MyData, value, startIndex, endIndex);
if (location > -1)
    cout << "The value: " << value << " was found in location " << location << " in the array {5, 9, 13, 22, 81, 143, 324, 443, 912}" << endl;
else
    cout << "The value: " << value << " is NOT in the array {5, 9, 13, 22, 81, 143, 324, 443, 912}" << endl;

// Now try the recursive version!
value = 443;
location = find_value_recursive(MyData, value, startIndex, endIndex);
if (location > -1)
    cout << "The value: " << value << " was found in location " << location << " in the array {5, 9, 13, 22, 81, 143, 324, 443, 912}" << endl;
else
    cout << "The value: " << value << " is NOT in the array {5, 9, 13, 22, 81, 143, 324, 443, 912}" << endl;
```

**Screenshot 3:** Run the program and include a Screen shot of the code and output after this step.
Below, you can review some other recursive algorithms we discussed in class. Study each algorithm, then identify the base case step, and recursive/reduce step:

```
//------------------------------------------------
int sum_recursive (int MyArray[], int ArraySize)
{
    if (debug == true) {
        cout << "sum_recursive( {";
        print_array(MyArray, ArraySize);
        cout << "}, ArraySize = " << ArraySize << " )is called" " endl;
    }
    if(ArraySize == 1)
        return(MyArray[0]);
    else
        return(MyArray[0] + sum_recursive(&MyArray[1], ArraySize-1));
}

void print_array(int MyArray[], int ArraySize)
{
    for (int i = 0; i < ArraySize; i++)
        cout << MyArray[i] << " ";
}
//------------------------------------------------
int power_recursive(int x, int n)
{
    if (debug) cout << "power( " << x << " , " << n << " ) is called" " endl;
    if (n < 0) {
        cout << "Illegal argument";
        exit(1);
    }
    if (n>0)
        return (power_recursive(x, n - 1) * x);
    else
        return (1);
```
int fibonacci(int n)
{
    //cout << "calling fibonacci(" << n << ");" << endl;
    if (n == 0) {
        //cout << 0 << ";";
        return 0;
    }
    else if (n == 1) {
        //cout << 1 << ";";
        return 1;
    }
    else {
        //cout << fibonacci(n - 1) + fibonacci(n - 2) << ";";
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int binary_search_recursive(int value, int TheArray[], int Low, int High)
{
    int Middle;

    // Base case
    if (Low > High)
        return(-1); //return -1 to indicate the value was not found
    else {
        Middle = int((Low + High) / 2);
        if (value == TheArray[Middle])  //It's a match!!
            return(Middle);
        else if (value < TheArray[Middle]) {  //Search the low end of array
            High = Middle - 1;
            binary_search_recursive(value, TheArray, Low, High);
        }
        else {
            Low = Middle + 1;
            binary_search_recursive(value, TheArray, Low, High);
        }
    }