COSC230 Practical 5:
Recursion
(Practical for Week 9)

The purpose of this practical is to implement recursive functions, and to understand the difference between a recursive process and an iterative process – and how recursion is implemented on a computer.

Question 1: **Recursive Functions**

(a) Write a recursive function to add the first n terms of the series

$$1 + 2 + 3 + ... + n.$$

Test your function using the general formula $S_n = n(n+1)/2$.

(b) Write a recursive function to compute the value of the following function $M : \mathbf{Z}^+ \to \mathbf{Z}$:

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$$

Check your function returns the value 91 for all positive integers less than or equal to 101.

Question 2: **Solving Recurrence Relations Recursively and Iteratively**

In AMTH140, we looked at sequences defined by recurrence relations. There, we solved recurrence relations using various methods to find explicit formulae for the nth term of a sequence. Here, we will make use of the run-time (or call) stack to directly implement the recurrence relation and generate terms of the sequence. Consider the recurrence relation defining the Fibonacci sequence:

$$a_n = a_{n-1} + a_{n-2},$$

with the initial conditions

$$a_1 = a_2 = 1.$$

(a) By directly implementing the recurrence relation, give a recursive function that returns the nth term of the Fibonacci sequence.

(b) Now write an iterative function (using a loop) that returns the nth term of the Fibonacci sequence.

(c) Finally, write a recursive function that generates an iterative process – sometimes also called tail-recursive. (Hint: use a similar strategy to part (b)).

Question 3: **A Recursive Process for Decimal to Binary Conversion**

Consider the following decimal to binary converter (a specific instance of the one you wrote last week in *Practical 4*):

```
void print_decimal_to_binary_iter(int dec)
{
    stack<int> s;

    while(dec != 0)
    {
        s.push(dec % 2);
        dec = dec / 2;
    }

    while(!s.empty())
    {
        cout << s.top();
        s.pop();
    }
    cout << endl;
}
```

Rewrite this iteratively-defined function as a recursively-defined function (see the lecture recording on recursive and iterative processes). Think carefully about how you would simulate this stack in a recursive function by pushing and popping stack frames on the call stack.

Question 4: **Memoization to avoid Excessive Recursion**

The function in Question 2(a) turns out to be highly inefficent. Try evaluating both the recursive and iterative functions in Question 2 for $n = 45$. You will notice a large difference in run time between the two functions. The reason is that once you call the recursive function with $n > 3$, the function computes all of the previous terms in the sequence more than once. For example, to find `fib(6)`, the function calls `fib(5)` and `fib(4)`. When it calls `fib(5)`, it must call `fib(4)` and `fib(3)`. When it calls `fib(4)`, it calls `fib(3)` and `fib(2)`. The recursive function has therefore called `fib(4)` twice, and `fib(3)` three times. In fact, the number of function calls grows exponentially with n. However, all of the repeated function calls are simply returning the same value again and again. To avoid repeatedly calculating the same values you can implement a form of caching called memoization. Memoization is an optimization step where the return value of a function call is stored in a lookup table for later use.

In this question, use an array as a lookup table to implement memoization in the function from Question 2(a). You will need to use the `C++` keyword `static` to enable static memory allocation. Memory that is allocated statically will only be allocated once, and will then persist until the program finishes. This allows the lookup table to persist over multiple function calls. Otherwise, a new lookup table would be instantiated for each function call. Check the efficiency (run time) of your new function.

Solutions

Question 1:

(a) The function is:

```
int sum_first_ints(int n)
{
    if (n <= 0)
        throw invalid_argument("Argument must be a positive integer.");
    if (n == 1)
        return 1;
    else
        return n + sum_first_ints(n - 1);
}
```

(b) The function is:

```
int M(int n)
{
    if (n <= 0)
        throw invalid_argument("Argument must be a positive integer.");
    if (n > 100)
        return n - 10;
    else
        return M(M(n + 11));
}
```

Question 2:

(a) A simple recursive function for the nth term of the Fibonacci sequence is:

```
int fib_rec(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return fib_rec(n - 1) + fib_rec(n - 2);
}
```

(b) An iterative function for the nth term of the Fibonacci sequence is:

```
int fib_loop(int n)
{
    int a1, a2, a3;
    a1 = 1, a2 = 1;

    if (n == 1 || n == 2)
        return 1;
    else {
        for ( ; n > 2; n--) {
            a3 = a1 + a2;
            a1 = a2;
            a2 = a3;
        }
        return a3;
    }
}
```

(c) A tail-recursive function for the nth term of the Fibonacci sequence is:

```
int fib_iter(int n, int a1, int a2)
{
    if (n == 1)
        return a1;
    else
        return fib_iter(n - 1, a2, a1 + a2);
}
```

Question 3:

The corresponding recursive function is:

```
void print_decimal_to_binary_rec(int dec)
{
    if (dec != 0) {
        print_decimal_to_binary_rec(dec / 2);
        cout << dec % 2;
    }
}
```

Note that both the recursive and iterative implementations of this function generate a recursive process.

Question 4:

The memoized recursive function for the nth term of the Fibonacci sequence is:

```
int fib_mem(int n)
{
    static int a[10000]; // zero-initialized lookup table
    a[0] = 1, a[1] = 1;

    if (!a[n-1]) // if not in lookup table, do expensive function call
        a[n-1] = fib_mem(n - 1) + fib_mem(n - 2);

    return a[n-1];
}
```