5: Recursion

One of the most common techniques used during the deriving of solutions to problems is iteration. There are some cases, however, where using an iterative approach is overly complicated. In some of those cases, it is significantly easier to use recursion.

Recursive definitions

Recursion is all about solving a problem by breaking it down into smaller versions of itself. To explain this concept, let us consider factorials.

Mathematically speaking, a factorial of a non-negative integer is defined with two equations below:

$0! = 1$ ①

$n! = n \times (n-1)!$ ②

Equation ① says that the factorial of *0* is equal to *1*, while equation ② says that the factorial of any number *n* is equal to the product of *n* and the factorial of *n-1*. You'll notice that even though we have a factorial in our definition of factorial, it is a reduced or smaller version of itself i.e. *n-1*.

Using this definition, we see that

$3! = 3 \times 2!$

But $2! = 2 \times 1!$

and $1! = 1 \times 0!$

and $0! = 1$.

Working backwards gives us

$1! = 1 \times 1 = 1$

$2! = 2 \times 1 = 2$

and $3! = 3 \times 2 = 6$

One thing to realize for this technique to work, we need a direct easy case called the **base case**, which in this case is equation ①. The other condition for this technique to work is that we need the **general case** which breaks down our solution to a simpler version of itself.

To recap:

- The recursive definition must have at least one base case.
- The general case must eventually be reduced to a base case, and
- The base case stops the recursion.

An algorithm that finds a solution to a problem by reducing it to smaller versions of itself is called a recursive algorithm. And in computer science, a function that calls itself is called a recursive

function i.e. the body of the function should have a statement that causes the same function to execute again before completing the current call.

Let's see what the factorial algorithm defined above would look like as a recursive function.

```
int fact(int num)
{
      if (num == 0)
            return 1;
      else
            return num * fact(num - 1);
}
```

Designing recursive functions

To design a recursive function, you must do the following:

- Understand the problem requirements
- Determine the limiting conditions
- Identify the base case and its direct solution
- Identify the general case and a solution to it which utilizes a smaller version of itself.

Write a recursive function to evaluate the exponent of one integer. Its signature is given below:

```
int pow(int x, int n) //It should return x^n.
```

Write a recursive function to return the n^{th} fibonacci number. The fibonacci sequence can be represented using the following mathematical description.

```
Fib(n) =  0                       if n = 0;
          1                       if n = 1;
          Fib(n-1) + Fib(n-2)     if n > 1.
```

For most cases, one can write either a recursive or normal iterative function to solve a task. There are however cases where the recursive approach is significantly easier.

Let's look at traversing a linked list from a previous lecture. The code that was provided in that lecture looked like the one below.

```
void Traverse(Node n)
{
      while (n != NULL)
      {
            System.out.print(n.getData() + " ");
            n = n.getLink();
```

```
      }
      System.out.println();
}

/* and the function would be called using a statement like:
Traverse(head);
*/
```

A function to traverse the list could be written in a recursive manner as well.

```
void Traverse(Node n)
{
      if (n == NULL)
            return;
      System.out.print(n.getData() + " ");
      Traverse(n.getLink());
}
```

As you can see, the base case is when the list is null in which case we return nothing. Otherwise, we output the data portion of the node in which we are, and then call the same function starting at the next node in the sequence.

Notice that in the iterative approach, once we move on to the next node, there is no way to get back since our variable `n` would have changed. However, in the recursive approach, since the function calls a copy of itself which has access to the next node, we technically still have access to the entire list. Recall that the function will not terminate until all function calls under it have been terminated. This means that we will have multiple copies of the `Traverse` function, each of which will have access to the list at the same time (even though we are only in one function at a time, and all the rest are kinda "on pause"). We can take advantage of this feature to print the list backward starting with the tail. To do this, we just make sure to do any processing of the nodes (such as the printing), on the way back from the base case. We do this by reversing the order of the processing and recursive call.

```
void Traverse(Node n)
{
      if (n == NULL)
            return;
      Traverse(n.getLink());
      System.out.print(n.getData() + " ");
}
```

Towers of Hanoi

The towers of Hanoi is probably the most famous example of a problem that would be solved by recursion. Towers of Hanoi is a game that involves moving different size disks from one peg to another. It typically involves 5 disks and 3 pegs. The only rule is that during the moving process, you can only move one disk at a time, and you can never place a larger disk on top of a smaller disk.

I will not go into too much detail about this game and or topic given that it was covered in CSC122.

```
void Hanoi(int n, char from, char to, char spare)
{
      if (n == 1)
      {
            System.out.println(from + " -> " + to);
```

```
            return;
        }
        Hanoi(n-1, from, spare, to);
        Hanoi(1, from, to, spare);
        Hanoi(n-1, spare, to, from);
}
```

So how many moves are required for 5 disks? 10 disks? What about 64 disks? The origin story about towers of Hanoi says that some priests have been solving a 64 disk tower of Hanoi problem and that the world will end when they finally do. If moving each disk took only 1 second, what's the shortest time it would it take to solve the 64 disk problem?