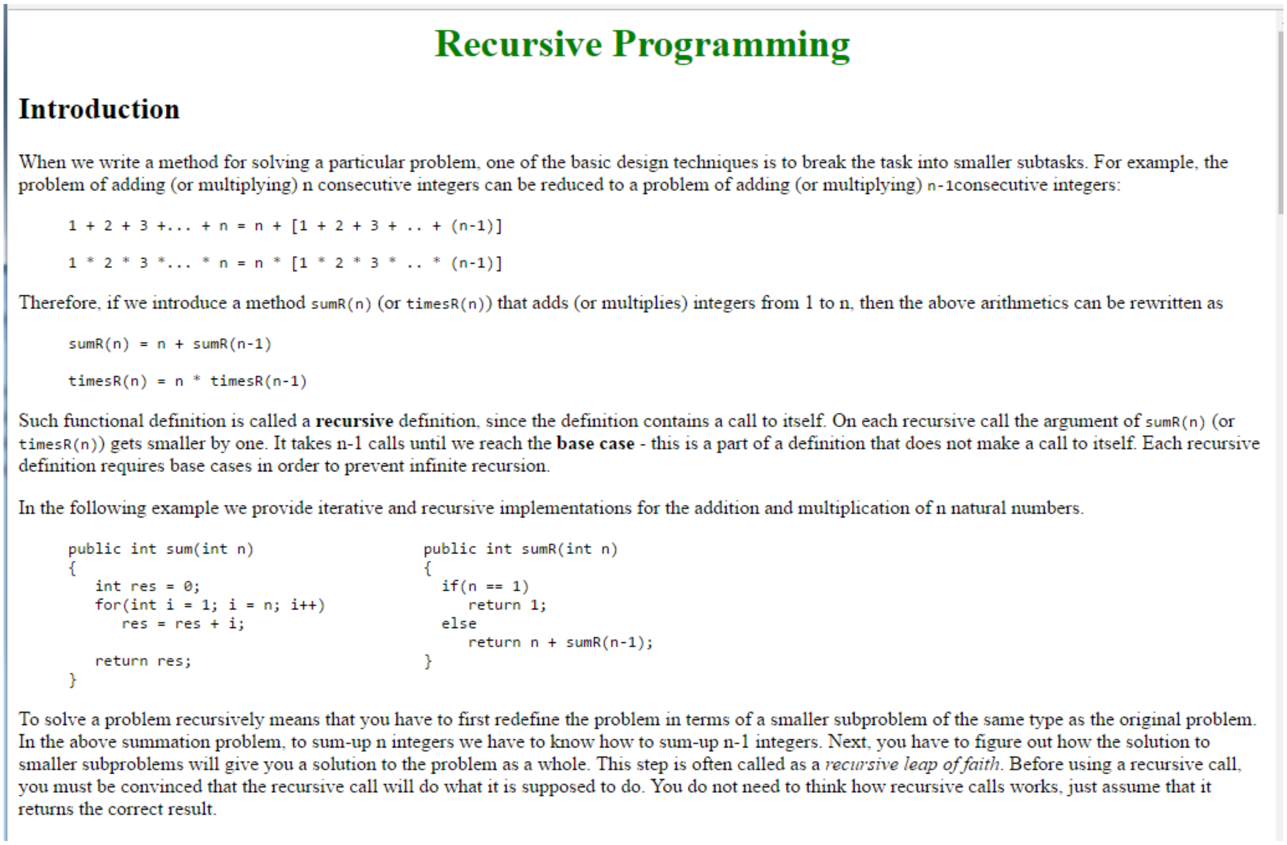
`



Recursive Programming

Introduction

When we write a method for solving a particular problem, one of the basic design techniques is to break the task into smaller subtasks. For example, the problem of adding (or multiplying) n consecutive integers can be reduced to a problem of adding (or multiplying) `n-1`consecutive integers:

```
1 + 2 + 3 +... + n = n + [1 + 2 + 3 + .. + (n-1)]

1 * 2 * 3 *... * n = n * [1 * 2 * 3 * .. * (n-1)]
```

Therefore, if we introduce a method `sumR(n)` (or `timesR(n)`) that adds (or multiplies) integers from 1 to n, then the above arithmetics can be rewritten as

```
sumR(n) = n + sumR(n-1)

timesR(n) = n * timesR(n-1)
```

Such functional definition is called a **recursive** definition, since the definition contains a call to itself. On each recursive call the argument of `sumR(n)` (or `timesR(n)`) gets smaller by one. It takes n-1 calls until we reach the **base case** - this is a part of a definition that does not make a call to itself. Each recursive definition requires base cases in order to prevent infinite recursion.

In the following example we provide iterative and recursive implementations for the addition and multiplication of n natural numbers.

```
public int sum(int n)                   public int sumR(int n)
{                                       {
   int res = 0;                           if(n == 1)
   for(int i = 1; i = n; i++)                return 1;
      res = res + i;                      else
                                             return n + sumR(n-1);
   return res;                          }
}
```

To solve a problem recursively means that you have to first redefine the problem in terms of a smaller subproblem of the same type as the original problem. In the above summation problem, to sum-up n integers we have to know how to sum-up n-1 integers. Next, you have to figure out how the solution to smaller subproblems will give you a solution to the problem as a whole. This step is often called as a *recursive leap of faith*. Before using a recursive call, you must be convinced that the recursive call will do what it is supposed to do. You do not need to think how recursive calls works, just assume that it returns the correct result.

Go to the following website and READ THOROUGHLY:

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Recursions/recursions.html