420-200-VA	Assignment 2B	Due by March 25, 2020
	Recursion (Chapter 13)	

1 Objectives

1. Practice recursion
2. Use helper methods to provide user friendly interface
3. Implement and use recursive methods in Java
4. Practice using the Java `ArrayList` collection

2 Background information on Recursion

Recursion is used frequently in computer science, in compilers, graphics, graph processing, operating systems, searching, sorting , and so on.

Recursion is a programming problem solving technique that solves a problem of a given size in terms of the solution(s) to instances of the same problem but of smaller size(s).

Note that the definition above requires some way measuring the "size" of the problem so that the notions such as "same problem but of smaller or larger size" are clearly defined.

The size of a problem is given by its definition. For example, the size of each of the following programming problems is some integer n, where $n \geq 1$:

A. Compute the sum of the n numbers $x_0, x_1, \cdots, x_{n-1}$.
B. Compute the minimum of n numbers.
C. Compute the maximum of n numbers.
D. Sort n names.
E. Search a list of n items,
F. Reverse a string of n characters.
G. Determine whether a string of n characters is a palindrome.
H. etc.

Note that we have solved many of such programming problems in the past. They all involve iteration, repeatedly carrying out a process, and they normally use explicit loops such as **for**, **while**, or **do-while** to implement the repetition.

2.1 An example: using iteration

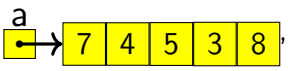
Here is an example of an *iterative* solution to the programming problem A above:

```
1  // an iterative solution to compute the sum of a given array elements
2  public static int sum(int [] array)
3  {
4    int sum = 0;
5    for(int n = 0; n < array.length; n++){ sum = sum + array[n];} // or
6  //for(int n = array.length-1; n >= 0; n--){ sum = sum + array[n];}// or
7  //for(int x : array){ sum += x; } // or,
8    return sum;
9  }
```

Note that the loop in line 5 or 7 scans the array from the first element through the last, and the loop in line 6 from the last element through the first.

However, iteration is not the only approach in town! Another is *recursion*.

2.2 An example: using recursion

Suppose, we want to find the sum of the numbers in an array 

, recursively.

Something we know for certain is that, no matter the approach, we must somehow scan the array elements, one by one. So we decide to represent the loop control variable `n` in line 5 above as a parameter into our *recursive* method:

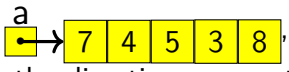
```
10 // a recursive solution to compute the sum of a given array elements
11 public static int sum(int[] a, int n)
12 {
13    // Defining a recursive method involves two steps:
14    // 1) Identify and define at least one base case;
15
16    // 2) Identify and define a general (or recursive) case,
17    // ensuring that the recursive calls ultimately reach a base case;
18
19 }
```

where

Base Case A base case is an instance of the problem, of a certain known size, which can be trivially solved without recursion.

Recursive Case A general (or recursive) case defines a relationship between a solution to a problem of a general size and solutions to one or more instances of the same problem, each of a smaller size.

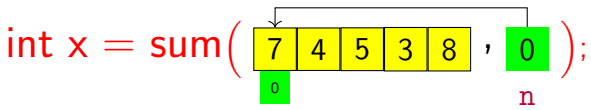
To prevent infinite recursion, the general case must make progress toward and eventually reach a base case.

In the case of our example array 

, the identification of at least one base case and a general case depend on the direction we want to scan the array elements:

A) Scan array from left to right

The base case is reached when `n == a.length-1`. We *must* require that the original call statement in the calling code to our recursive method be of the form `sum(a,0);`, which is illustrated as follows:

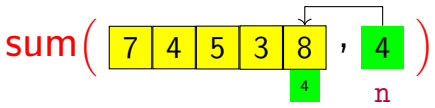


Inside our method `sum`, we reduce of the problem "size" as follows:

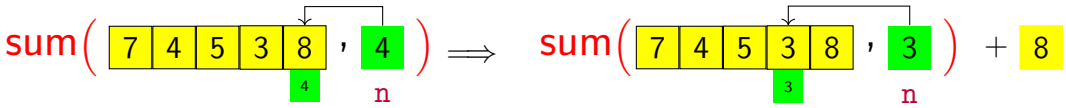
See the entire recursive process here.

B) Scan array from right to left

The base case is reached when `n == 0`. We *must* require that the original call statement in the calling code to our recursive method be of the form `sum(a,a.length-1);`, which is illustrated as follows:



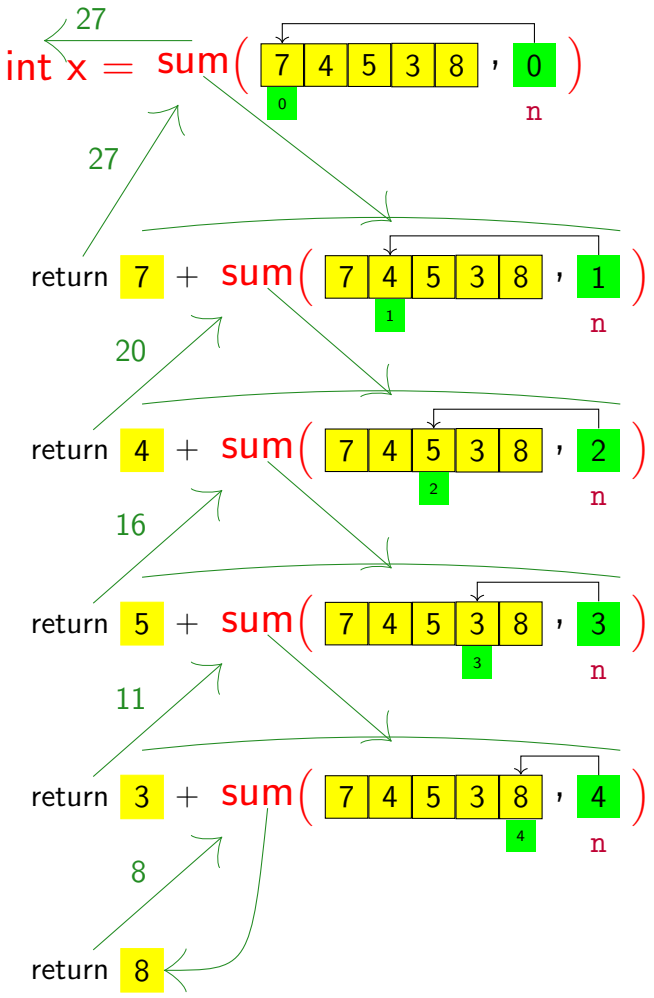
Inside our method `sum`, we reduce of the problem "size" as follows:



See the entire recursive process here.

3 Scanning the array left to right

```
1  // computes the sum of a given array elements
2  public static int sum(int[] a)
3  {  // sole purpose: to prepare and prime the actual call to our helper method below
4     return sum(a, 0); // hide implementation details from the caller
5  }
6  // a helper method that helps to avoid burdening the calling code with the details
7  // of our implementation of the actual method; note that this method is private;
8  // hence it is not accessible by code outside this class
9  private static int sum(int[] a, int n)
10 {
11    if( n == a.length-1) return a[n]; // base case
12    else return  a[n] + sum(a, n+1); // general (recursive) case
13 }
```



`sum(a,0)`, the original (non-recursive) call from the helper method; waits until an answer is sent back from this call; problem size is 5

1st time inside `sum`; reduces problem size to 4; waits until the call `sum(a,1)` returns an answer

2nd time inside `sum`; reduces problem size to 3; waits until the call `sum(a,2)` returns an answer

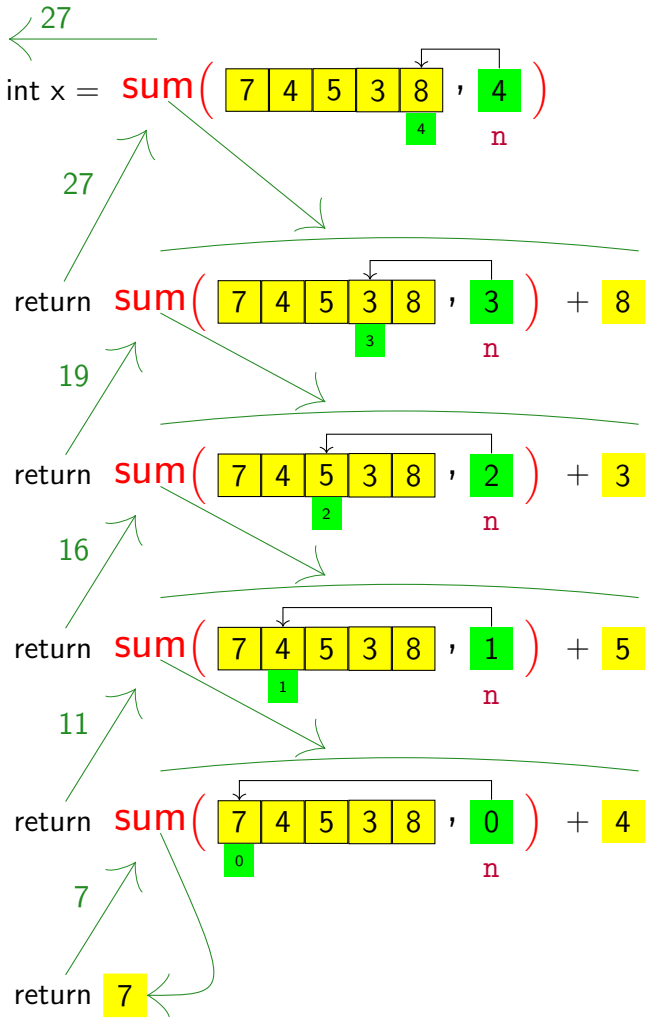
3nd time inside `sum`; reduces problem size to 2; waits until the call `sum(a,3)` returns an answer

4'th time inside `sum`; reduces problem size to 1; waits until the call `sum(a,4)` returns an answer

5'th time inside `sum`; returns 8 via **base case** (`n == a.length-1`) on line 10

4 Scanning the array right to left

```
1  // computes the sum of a given array elements
2  public static int sum(int[] a)
3  {  // sole purpose: to prepare and prime the actual call to our helper method below
4     return sum(a, a.length-1); // hide implementation details from the caller
5  }
6  // a helper method that helps to avoid burdening the calling code with the details
7  // of our implementation of the actual method; note that this method is private;
8  // hence it is not accessible by code outside this class
9  private static int sum(int[] a, int n)
10 {
11    if( n == 0) return a[n];          // base case
12    else return sum(a, n-1) + a[n];   // general (recursive) case
13 }
```



`sum(a,4)`, the original (non-recursive) call from the helper method; waits until an answer is sent back from this call; problem size is 5

1st time inside `sum`; reduces problem size to 4; waits until the call `sum(a,3)` returns an answer

2nd time inside `sum`; reduces problem size to 3; waits until the call `sum(a,2)` returns an answer

3nd time inside `sum`; reduces problem size to 2; waits until the call `sum(a,1)` returns an answer

4'th time inside `sum`; reduces problem size to 1; waits until the call `sum(a,0)` returns an answer

5'th time inside `sum`; returns 7 via **base case** (`n == 0`) on line 10

5 Your tasks

5.1 IntArrayTools

Write a class named `IntArrayTools` that has no instance variables and includes the following `static` methods.

(1) A pair of overloaded methods, named `max`:

   (a) A `public` static method `max(int[] array)` that returns the maximum value of all `array` elements. This method must delegate its task to the following recursive helper method, priming its arguments according to your implementation.

   (b) A `private` static recursive method `max` that takes an array of type `int[]` as an argument and potentially any other arguments that you need to introduce to accomplish its task. Your method must determine, recursively, and return the maximum value of all the elements in the supplied array.

(2) Again, a pair of overloaded methods, named `max`, implementing the same algorithm as used in (1) above. The difference is that they use an `ArrayList<Integer>` instead of a raw array.

   (a) A public method `max(ArrayList<Integer> list)` that returns the maximum value of all array `list` elements. This method must delegate its task to the following recursive helper method, priming its arguments according to your implementation.

   (b) A private recursive method `max` that takes an array list of type `ArrayList<Integer>` as an argument and potentially any other arguments that you need to introduce to accomplish its task. Your method must determine, recursively, and return the maximum value of all the elements in the supplied array list.

(3) A pair of overloaded methods, named `contains`:

   (a) A public method `contains(ArrayList<Integer> list, int value)` that determines whether the array `list` contains the `value`. This method must delegate its task to the following recursive helper method, priming its arguments according to your implementation.

   (b) A private recursive method that takes an array list of type `ArrayList<Integer>`, an `int value`, and potentially any other arguments that you need to introduce to accomplish the task. Your method must determine, recursively, whether the supplied array list contains the specified value.

Your program should include a client class to test your class.

5.2 charArrayTools

Write a class named `charArrayTools` that has no instance variables and includes the following `static` methods.

(1) A pair of overloaded methods, named `howMany`:

   (a) A public method `howMany(ArrayList<Character> list, char ch)` that returns the number of occurrences of `ch` in the array `list`. This method must delegate its task to the following recursive helper method, priming its arguments according to your implementation.

   (b) A private recursive helper method named `howMany` that takes an array list of type `ArrayList<Character>`, a `char ch`, and potentially any other arguments that you need to introduce to accomplish the task. Your method must determine, recursively, and return the number of occurrences of `ch` in the array list.

(2) Again, a pair of overloaded methods, named `howMany`, implementing the same algorithm as used in (1) above. The difference is that they use a `String` instead of an array list.

   (a) A public method `howMany(String str, char ch)` that returns the number of occurrences of `ch` in the string `str`. This method must delegate its task to the following recursive helper method, priming its arguments according to your implementation.

   (b) A private recursive helper method named `howMany` that takes an array list of type `ArrayList<Character>`, a `char ch`, and potentially any other arguments that you need to introduce to accomplish the task. Your method must determine, recursively, and return the number of occurrences of `ch` in the array list.

(3) A pair of overloaded methods, named `isPalindrome`.

   A palindrome is a string that is spelled the same way forward and backward. Some examples of palindromes are:

```
Able was I, ere I saw Elba
A man, a plan, a canal, Panama
Desserts, I stressed
Kayak
```

   (a) A public method `isPalindrome(ArrayList<Character> list)` that determines whether `list` stores a palindrome. This method must delegate its task to the following recursive helper method, priming its arguments according to your implementation.

   (b) A private recursive helper method that takes an array of type `ArrayList<Character>` and potentially any other arguments that you need to introduce to accomplish the task. Your method must determine, recursively, whether the supplied array list object stores a palindrome.

Your program should include a client class to test your class.

5.3 Recursive Multiplication

Problem: Write a private recursive method `int multiply(int m, int n)` that returns the product `m * n` *without* using multiplications.

Assume that both `m` and `n` are non-negative (≥ 0).

Problem size: `m` or `n`; preferably, the smaller of `m` and `n`, for efficiency

Hint Recall that multiplication can be performed as repeated addition. For example,

$$m * n = (m - 1) * n + n$$

or

$$m * n = m * (n - 1) + m$$

6 Evaluation Criteria

Evaluation Criteria

Functionality	Ability to perform as required, producing correct output for any set of input data, Proper implementation of all specified requirements, Efficiency	60%
Robustness	Ability to handle input data of wrong type or invalid value	10%
OOP style	Encapsulating only the necessary data inside objects, Information hiding, Proper use of Java constructs and facilities.	10%
Documentation	Description of purpose of program, Javadoc comment style for all methods and fields, comments on non-trivial steps in all methods	10%
Presentation	Format, clarity, completeness of output, user friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing, localizing variables	5%