RecursionFun

Collaboration: Solo  Work on this project alone. Do not copy any code from anywhere, other than from our website, book, or lecture notes. Do not look at another person's screen or printout. And certainly do not send you file to another.

This project asks you to solve 21 problems using recursive solutions. In each, the method must call other invocations of the same method in order to find the solution.

Work on one method at a time. Write one @Test for each method. For each method spec, sometimes assertion are given, sometimes a few test cases, and sometimes there are neither.

The class name holding all 21 public methods must be named public class RecursionFun

// Precondition: k >= 1 and n >= k
1) public int combinations(int n, int k)

Complete recursive method combinations that returns from n choose k. Method combinations is described in 17-SimpleRecursion.ppt slides 17..22 from lecture. The following recursive definition is correct for combinations (but called c here for brevity's sake). This reads that when k == 1 return n, when n == k, return 1, and c(5, 2) would return c(4, 1) + c(4, 2).

\[
c(n,k) = \begin{cases} 
  k = 1 & \Rightarrow n \\
  n = k & \Rightarrow 1 \\
  n > k & \& k > 1 & \Rightarrow c(n-1,k-1) + c(n-1,k) 
\end{cases}
\]

// Precondition: n >= 0
2) public int factorial(int n)

Implement factorial to return n! that is defined as n * n-1 * n-2 * n-3 * 2 * 1. 1! And 0! Are both defined as 1. Use recursion, no loop.

factorial(0) → 1
factorial(1) → 1
factorial(2) → 2 * 1 = 2
factorial(3) → 3 * 2 * 1 = 6
factorial(4) → 4 * 3 * 2 * 1 = 24

// Precondition: n >= 1
3) public double addReciprocals(int n)

Complete recursive method addReciprocals(int) that takes an integer as a parameter and returns the sum of the first n reciprocals. Use recursion; do not use a loop. addReciprocals(n) returns (1.0+1.0/2.0+1.0/3.0+...+1.0/n). The following assertions must pass:

addReciprocals(1) → 1.0
addReciprocals(2) → 1.0 + 1.0/2.0 = 1.5
addReciprocals(3) → 1.0 + 1.0/2.0 + 1/3.0 = 1.8333333333333333
// Precondition: m and n are not both 0. Neither can be negative.

4) public int GCD(int m, int n)

Implement the Greatest Common Divisor algorithm as recursive method GCD.

$$GCD(m,n)\begin{cases} m & \text{if } n == 0 \\
GCD(m, m\%n) & \text{if } n > 0
\end{cases}$$

Use recursion. Do not use a loop. If only one argument is 0, GCD should return the other argument. GCD is undefined when both arguments are 0 (we have no tests for this undefined case). Because the \% operator does not work with negative integers, you will need to convert the arguments to their absolute value with Math.abs(int). These assertions should pass:

assertEquals(3, rf.GCD(24, 9)); // Assume this instance variable is in the test
assertEquals(10, rf.GCD(50, 20)); // private RecursionFun rf = new RecursionFun();
assertEquals(1, rf.GCD(7, 20));

// Precondition: n > 0 && n <= 40 (takes a long time when n > 40)

5) public int fibonacci(int n)

The first 9 Fibonacci numbers are 1 1 2 3 5 8 13 21 34. Each Fibonacci number is the sum of the preceding two (except for the first two, which are both 1). Implement a recursive method named fibonacci that returns the nth Fibonacci number. Use recursion, no loops.

fibonacci(1) → 1
fibonacci(2) → 1
fibonacci(3) → 2
fibonacci(4) → 3
fibonacci(5) → 5
fibonacci(6) → 8

6) public String underScore(String str)

Given a string, compute recursively a new string where identical chars that are adjacent in the original string are separated from each other by a "_". Use recursion, no loops.

underScore("hello") → "hel_lo"
underScore("xxyy") → "x_x_y_y"
underScore("aaaa") → "a_a_a_a"

7) public boolean nestParen(String str)

Given a string, return true if it is a nesting of zero or more pairs of parenthesis, like "((())" or "((((()". Suggestion: check the first and last chars, and then recur on what's inside them. Use recursion, no loops.

nestParen( "((())" ) → true
nestParen( "((()))" ) → true
nestParen( "(((()))" ) → false
nestParen( "(" ) → false
nestParen( "x" ) → false
nestParen( "x " ) → false
nestParen( "(yy " ) → false
8) **public** String noAdjacents(String str)

Complete recursive method noAdjacents that when given a string, recursively returns a string where adjacent chars that are the same have been reduced to a single char.

```java
noAdjacents("yzzza") -> "yza"
noAdjacents("abbcdd") -> "abcd"
noAdjacents("Hello") -> "Helo"
```

// Precondition: n >= 0

9) **public** String convert(int num, int base)

Complete recursive method convert that returns a string that is the base equivalent num. For example, convert(5, 2) should return “101” base 2 (binary) and convert(16, 8) returns “20” (octal).

```java
convert(1, 2) -> "1"
convert(3, 2) -> "11"
convert(5, 2) -> "101"
convert(13, 8) -> "15"
convert(8, 8) -> "10"
convert(123, 10) -> "123"
```

// Precondition: n >= 0

10) **public** String intWithCommas(int n)

Complete recursive method intWithCommas that returns the argument as a String with commas in the correct places. The convert method from lecture with hexadecimal numbers provides a pattern.

```java
intWithCommas(999) -> "999"
intWithCommas(1234) -> "1,234"
intWithCommas(1007) -> "1,007"
intWithCommas(1023004567) -> "1,023,004,567"
```

// Preconditions: x.length >= 1, beginIndex < x.length, endIndex < x.length

11) **public** int sumArray(int[] nums, int beginIndex, int endIndex)

Complete recursive method sumArray that returns the sum of all the int elements in the given range of indexes. Use recursion, do not use a loop. You must have a recursive call in your answer. The following assertions must pass where the 2nd and 3rd arguments represent the index range:

```java
int[] x = { 1, 5, 7, 2, 3, 4 }; // Assume this instance variable exists
int[] a = { 2, 4, 6 }; // Assume this instance variable is in the test
```
public void reverseArray(int[] nums)
Write recursive method reverseArray that reverses the array elements in a filled array of ints. Use recursion. Do not use a loop. The following assertions must pass:

```java
int[] a = { 2, 4, 6 };
rf.reverseArray(a);
assertEquals(6, a[0]);
assertEquals(4, a[1]);
assertEquals(2, a[2]);
```

public int arrayRange(int[] nums)
Write recursive method arrayRange that returns the maximum integer minus the minimum integer in the filled array of ints. Use recursion; do not use a loop. The following assertions must pass. Notice the shortcut way to pass a reference to a new array—it saves your writing a bit of code.

```java
assertEquals(2, rf.arrayRange(new int[] { 1, 2, 3 }));
assertEquals(2, rf.arrayRange(new int[] { 3, 2, 1 }));
assertEquals(0, rf.arrayRange(new int[] { 3 }));
assertEquals(3, rf.arrayRange(new int[] { -3, -2, -5, -4 }));
```

public boolean isSorted(int[] nums)
Complete method isSorted to return true if the given array of ints is sorted in ascending order. Return false if not completely sorted. By definition, an empty array is sorted.

```java
int[] intsOne = {1, 2, 5, 5, 6, 6, 7};
int[] intsTwo = {1, 2, 5, 0, -1};
assertTrue(rf.isSorted(intsOne));
assertFalse(rf.isSorted(intsTwo));
```

public boolean found(String search, String[] strs)
Complete method found to return true if search is found in the array strs. If searchValue is not found, return false.

```java
String[] strs = { "Ttt", "Ccc", "Fff", "Ddd", "Hhh", "Aaa"};
assertTrue(rf.found("Aaa", strs));
assertTrue(rf.found("Hhh", strs));
assertFalse(rf.found("Not Here", strs));
```

public int binarySearch(String searchValue, String[] strings)
Complete recursive method binarySearch to return the index of the first String that equals searchValue. Use recursion, do not use a loop. Use the binary search algorithm so it run O(log n). Use a sorted array of unique strings to test binarySearch. If searchValue is not found, return -1. The following assertions must pass:

```java
String[] strs = { "Aaa", "Ccc", "Ddd", "Fff", "Hhh", "Ttt" };
assertEquals( 0, rf.binarySearch("Aaa", strs));
assertEquals( 4, rf.binarySearch("Hhh", strs));
assertEquals( -1, rf.binarySearch("Not here", strs));
Complete four methods in class LinkedList using recursion, no loops

You will need this beginning to a LinkedList class before completing methods 18..21

```java
public class LinkedList<E extends Comparable<E>> {
    private class Node {
        private E data;
        private Node next;
        public Node(E element) {
            data = element;
            next = null;
        }
    }
    private Node first;
    private int n;
    public LinkedList() {
        first = null;
        n = 0;
    }
    public int size() {
        return n;
    }
    public void addLast(E el) {
        if (first == null)
            first = new Node(el);
        else
            addLast(el, first);
        n++;
    }
    private void addLast(E el, Node ref) {
        if (ref.next == null)
            ref.next = new Node(el);
        else
            addLast(el, ref.next);
    }
    // Precondition: 0 <= index < size()
    18) public E get(int index)
    Return a reference to the element at the given index. This method may run O(n). Use recursion. Do not use a loop.

    @Test
    public void testAddLastAndGet() {
        LinkedList<String> list = new LinkedList<String>();
        list.addLast("A");
        list.addLast("B");
        list.addLast("C");
        list.addLast("D");
        assertEquals("A", list.get(0));
        assertEquals("B", list.get(1));
        assertEquals("C", list.get(2));
        assertEquals("D", list.get(3));
    }
```
19) **public** E max()
Return the largest element in this list according to the type’s `compareTo` method. Use recursion. Do not use a loop. If the list is empty (`size() == 0`) return `null`.

20) **public int** occurrencesOf(E el)
Return how often `el` occurs in this list. Use recursion. Do not use a loop. Return `0` if the list is empty.

21) **public void** duplicateAll(E el)
To the LinkedList class, add method `duplicateAll(E element)` so all elements in the singly linked structure that equals `element` are repeated next to the original. The size should increase for each element.

**Grading Criteria**

**Problem and Code Coverage 100pts**
To get 100% for these 90 points, you will need 100% problem coverage and 100% code coverage. You can get a score of 0 even though all of your tests passed in your workspace because

- WebCat reports a compile time error (look for Unknown symbol).
- One of your assertions failed on WebCat (even though it passed for you locally)
- A WebCast test caused your code to throw any exception

Please note other ways to lose points

-5 For any method that fails to properly use recursion to solve the problem. If you have a loop, consider a different way to do the problem. Do not have any loops anywhere. Use recursion.

-100 If your unit test does not end with Test, or any one of YOUR assertions fails on WebCat, or if you have any one compiletime error—and all it takes is one wrong letter in a class or method name or the wrong type or number of parameters for one compile time error. Please read all WebCat feedback.

-100 If you program times out

-100 If one of your `@Tests` do not pass.