# COMP132 – Computer Science II
# Fall 2015

## Homework #8 - Solutions
## Recursion

1. In class we traced the execution of several recursive function definitions. For example, we traced the recursive definition of `sum` as follows:

```
sum(4)  = 4 + sum(3)
        = 4 + (3 + sum(2))
        = 4 + (3 + (2 + sum(1)))
        = 4 + (3 + (2 + 1))
        = 4 + (3 + 3)
        = 4 + 6
        = 10
```

For each of the recursive function definitions given below, trace the execution of the indicated recursive function call.

i. f(x) = 2*x + f(x-1)
   f(0) = 3;

   Trace:    f(4)

```
f(4)= 2*4 + f(3)
    = 2*4 + (2*3 + f(2))
    = 2*4 + (2*3 + (2*2 + f(1)))
    = 2*4 + (2*3 + (2*2 + (2*1 + f(0))))
    = 2*4 + (2*3 + (2*2 + (2*1 + 3)))
    = 2*4 + (2*3 + (2*2 + 5))
    = 2*4 + (2*3 + 9)
    = 2*4 + 15
    = 23
```

ii. g(x) = 2*g(x-1) – g(x-2)
    g(2) = 1
    g(1) = 3

    Trace:      g(5)

```
g(5)= 2*g(4) – g(3)
    = 2*(2*g(3) – g(2)) – (2*g(2) – g(1))
    = 2*(2*(2*g(2) – g(1)) – 1) – (2*1 – 3)
    = 2*(2*(2*1 – 3) – 1) – (2*1 – 3)
    = 2*(2*(-1) - 1) – (-1)
    = 2*(-2 – 1) – (-1)
    = -6 – (-1)
    = -5
```

2. Consider the following recursive method:

```
public static void func(int num) {
  if (num == 5) {
    System.out.println("done!");
  }
  else {
    func(num+1);
    System.out.println(num);
  }
}
```

What output would be generated by the method call `func(1)`?

```
done!
4
3
2
1
```

To see this consider the sequence of method calls that would occur:

`func(1)` -> calls `func(2)`
  Note the print does not happen until after the call to `func(2)` returns.
`func(2)` -> calls `func(3)`
`func(3)` -> calls `func(4)`
`func(4)` -> calls `func(5)` which prints "`done!`"
  Then the call to `func(5)` completes and control returns to the point from which `func(5)` was called (i.e. back to the instruction following the call to `func(4)`.)
The call to `func(4)` picks up where it left off and it prints `4`. Note that the call to `func(num+1)` has not changed `num` in the current call, which is why `4` and not `5` is printed.
  When the call to `func(4)` completes control then returns to the point from which `func(4)` was called (i.e. back to the instruction following the call to `func(3)`.)
The call to `func(3)` resumes and prints `3`.
  The call to `func(3)` then completes and control returns to the point from which `func(3)` was called (i.e. back to below the call to `func(2)`.)
And so on until the initial call to `func(1)` prints `1`. At that point the original call completes and the program terminates.

3. Each of the following recursive function definitions contains a problem. Using the description of a recursive function/method/procedure given in class, briefly but fully and clearly explain what is wrong with each definition.

i. f(x) = 3*f(x-5)      where x is a positive integer.
   f(0) = 7

The problem with this definition is that for some values of x, repeated application of the recursive case does not reach a base case (e.g. x=8). In fact, any x that is not a multiple of 5 will never reach the base case, resulting in infinite recursion.

ii. g(x) = 2 - 4*g(x)      where x is a positive integer.
   g(1) = 3
   g(2) = 4

The problem with this definition is that the recursive case does not define the problem in terms a simpler version of the same problem. Repeated applications of the recursive case do not move us any closer to one of the base cases, resulting in infinite recursion.

4. Give a recursive definition for each of the problems stated below. For each problem clearly state the base case(s) and the recursive case(s). You do not need to implement these definitions in Java.

i. exp(x,n) = $x^n$   both x and n are non-negative integers.
   E.g.   exp(2,5) = 32

| | |
|---|---|
| Base Case: | exp(x,0) = 1 |
| Recursive Case: | exp(x,n) = x * exp(x, n-1) |

ii. arrSum(x) = the total of all values in x, where x is an array of length n (n >= 0).
   E.g.   arrSum({1, 4, 2, 8, 6}) = 21

| | |
|---|---|
| Base Case: | arrSum(x) = 0   if n = 0 |
| Recursive Case: | arrSum(x) = x[0] + arrSum(x[1…n-1]) |

Note: x[1…n-1] indicates a new array containing elements 1…n-1 of the array x.

iii. Assume we have a programming language that only allows conditionals (i.e. if statements), operations for +1 and -1, and recursive calls. How could the following function be defined recursively in that language?

add(a,b) = a + b  both a and b are non-negative integers.
E.g.  add(5, 7) = 12

| | |
|---|---|
| Base Case: | add(a,b) = 0 if a==0 and b==0 |
| Recursive Cases: | add(a,b) = 1 + add(0,b-1) if a==0 |
| | add(a,b) = 1 + add(a-1,0) if b==0 |
| | add(a,b) = 1 + 1 + add(a-1,b-1) |

5. Write a java method `fact(int x)` that computes the factorial of `x` recursively. Copy and paste the source code for your recursive method as your solution to this problem.

```java
public static int fact(int x) {
  if (x == 0) {
    return 1;
  }
  else {
    return x * fact(x-1);
  }
}
```

6. The `isPalindrome2` method in the `PalindromeChecker` class contained in the `comp132.examples.recursion` package of the `132SampleCode` project currently requires that a text string read exactly the same forward and backwards to be declared a palindrome. For example, "ABBA" is declared a palindrome but "Madam I'm Adam" is not. A more conventional definition of a palindrome allows for differences in spaces and punctuation such that "Madam I'm Adam" and "A man, a plan, a canal, Panama!" are considered palindromes. Modify the three-argument `isPalindrome2` method such that it ignores differences in spacing and punctuation. Your solution should not simply pre-process the string (i.e. do not just remove all spaces and punctuation and then call our original `isPalindrome` function.) Hint: The `Character` class in the JDK has a method that will determine if a character is a letter. Hint2: Consider adding some additional recursive cases.

```java
private static boolean isPalindrome2(String str, int start, int end) {
   if (end - start <= 0) {
      // Any string with 0 or 1 chars is a palindrome.
      return true;
   }
   else {
      // Get the first and last characters of the string.
      char first = Character.toLowerCase(str.charAt(start));
      char last = Character.toLowerCase(str.charAt(end));

      if (!Character.isLetter(first)) {
         // first char is not a letter so check without it.
         return isPalindrome2(str, start+1, end);
      }
      else if (!Character.isLetter(last)) {
         // last char is not a letter so check without it.
         return isPalindrome2(str, start, end-1);
      }
      else if (first == last) {
         // first and last letters are the same so check without them.
         return isPalindrome2(str, start+1, end-1);
      }
      else {
         // there must be a mismatch so it's not a palindrome.
         return false;
      }
   }
}
```