5.1.1

Trace the execution of the call `mystery(4)` for the following recursive method using the technique shown in Figure 5.2. What does this method do?

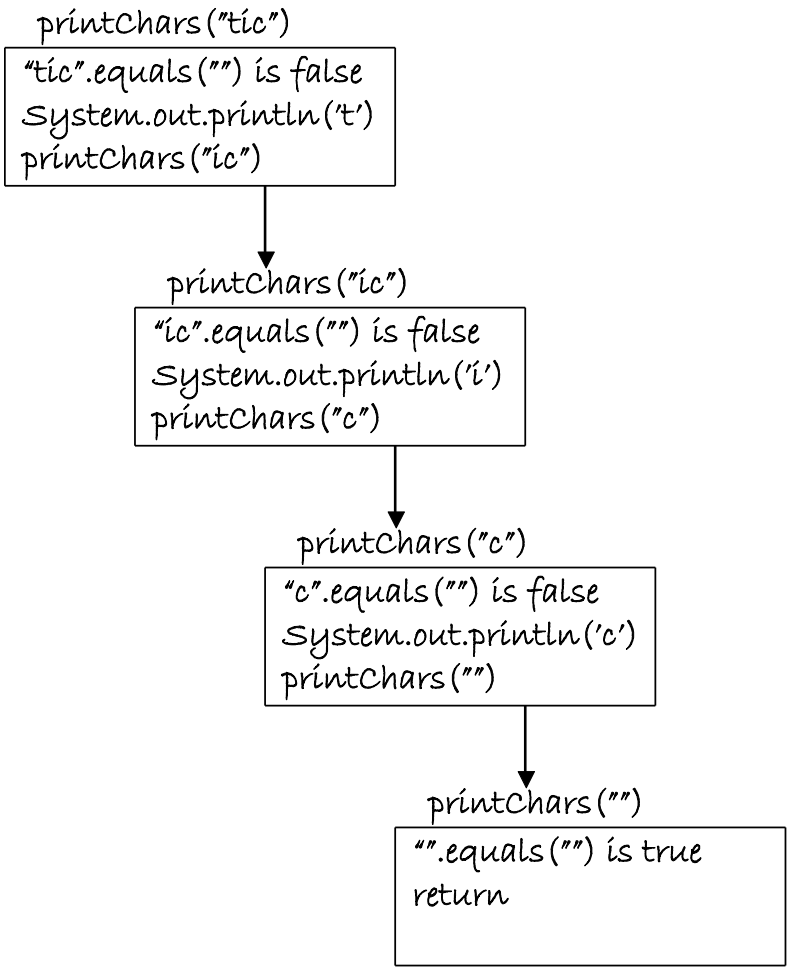
```
public static mystery(int n) {
    if (n == 0)
        return 0;
    else
    return n * n + mystery(n - 1);
}
```

```
mystery(4)   (condition false, recursive case)
|      mystery(3)  (condition false, recursive case)
|        |      mystery(2)  (condition false, recursive case)
|        |      |       mystery(1)  (condition false, recursive case)
|        |      |       |       mystery(0)  (condition true, base case)
|        |      |       |       ← return 0
|        |      |       ← return 1^2 + 0
|        |      ← return 2^2 + 1^2 + 0
|       ← return 3^2 + 2^2 + 1^2 + 0
← return 4^2 + 3^2 + 2^2 + 1^2 + 0
```

This function returns the sum of squares of all integers up to the input integer.

5.1.3

Trace the execution of `printChars("tic")` (Example 5.2) using activation frames.



5.1.5

Prove that the `printChars` method is correct.

The base case is recognized and solved correctly in the lines:

```
if (str == null || str.equals(""))
      return;
```

The recursive case does make progress towards the base by sending a smaller string for each recursive iteration:

```
printChars(str.substring(1));
```

If the smaller problems are solved correctly, the original problem is also solved:

```
System.out.println(str.charAt(0));
```

In this line, the first character is printed, and then in the next statement, the rest of the string is sent of the function.

5.1.7

Write a recursive algorithm that determines whether a specified target character is present in a string. It should return **`true`** if the target is present and **`false`** if it is not. The stopping steps should be:
a. a string reference to **`null`** or a string of length 0, the result is **`false`**
b. the first character in the string is the target, the result is **`true`**
The recursive step would involve searching the rest of the string.

isPresent(c, s)

if s is null or the empty string, return false
if the first character of s is c, return true
recursively call isPresent(c, substring of s without the first character)

5.2.1

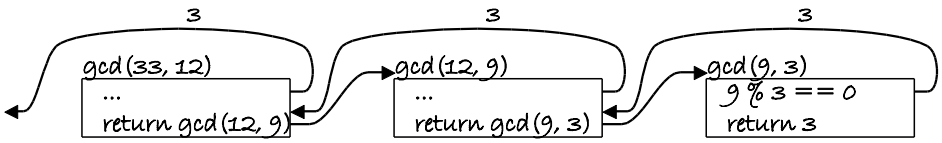
Does the recursive algorithm for raising x to the power n work for negative values of n? Does it work for negative values of x? Indicate what happens if it is called for each of these cases.

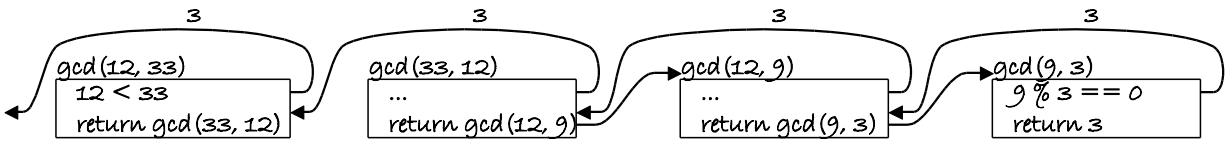
The algorithm does not work for negative values of n; it goes into an infinite loop. It works fine for negative values of x.

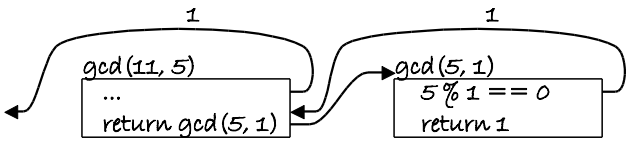
5.2.3

Trace the execution of the following using activation frames.

```
gcd(33, 12)
gcd(12, 33)
gcd(11, 5)
```







5.2.5

See for what value of *n* method `fibonacci` begins to take a long time to run on your computer (over 1 minute). Compare the performance of `fibonacciStart` and `fibo` for this same value.

On a 3.0GHz Quad Core AMD 64 fibonacci(49) == 7,778,742,049 computed in 61 seconds and fibonacciStart(49) == 7,778,742,049 computed in 4 microseconds.

5.3.1

For the array shown in Figure 5.9, show the values of `first`, `last`, `middle`, and `compResult` in successive frames when searching for a target of `"Rich"`; when searching for a target of `"Alice"`; when searching for a target of `"Daryn"`.

"Rich"

binarySearch(kidNames, "Rich")
items: kidNames
target: "Rich"
return binarySearch(kidNames, "Rich", 0, 6)

binarySearch(kidNames, "Rich", 0, 6)

items: kidNames

target: “Rich”
first: 6
last: 0
middle = (0 + 6) / 2 = 3
(0 > 6) is false
compResult is positive
return binarySearch(kidNames, “Rich”, 4, 6)

binarySearch(kidNames, “Rich”, 4, 6)

items: kidNames
target: “Rich”
first: 6
last: 4
middle = (4 + 6) / 2 = 5
compResult is positive
return binarySearch(kidNames, “Rich”, 6, 6)

binarySearch(kidNames, “Rich”, 6, 6)

items: kidNames
target: “Rich”
first: 6
last: 6
middle = (6 + 6) / 2 = 6
compResult is 0
return 6

“Alice”

binarySearch(kidNames, “Alice”)
items: kidNames
target: “Alice”
return binarySearch(kidNames, “Alice”, 0, 6)

binarySearch(kidNames, “Alice”, 0, 6)

items: kidNames
target: “Alice”
first: 6
last: 0

middle = (0 + 6) / 2 = 3
(0 > 6) is false
compResult is negative
return binarySearch(kidNames, “Alice”, 0, 2)

binarySearch(kidNames, “Alice”, 0, 2)

items: kidNames
target: “Alice”
first: 0
last: 2
middle = (0 + 2) / 2 = 1
compResult is negative
return binarySearch(kidNames, “Alice”, 0, 0)

binarySearch(kidNames, “Alice”, 0, 0)

items: kidNames
target: “Alice”
first: 0
last: 0
middle = (0 + 0) / 2 = 0
compResult is negative
return binarySearch(kidNames, “Alice”, 0, -1)

binarySearch(kidNames, “Alice”, 0, -1)

items: kidNames
target: “Alice”
first: 0
last: -1
middle = (0 - 1) / 2 = 0
(first > last)
return -1

“Daryn”

binarySearch(kidNames, “Daryn”)
items: kidNames
target: “Daryn”
return binarySearch(kidNames, “Daryn”, 0, 6)

binarySearch(kidNames, “Daryn”, 0, 6)

items: kidNames
target: “Daryn”
first: 6
last: 0
middle = (0 + 6) / 2 = 3
(0 > 6) is false
compResult is negative
return binarySearch(kidNames, “Daryn”, 0, 2)

binarySearch(kidNames, “Daryn”, 0, 2)

items: kidNames
target: “Daryn”
first: 0
last: 2
middle = (0 + 2) / 2 = 1
compResult is negative
return binarySearch(kidNames, “Daryn”, 0, 0)

binarySearch(kidNames, “Daryn”, 0, 0)

items: kidNames
target: “Daryn”
first: 0
last: 0
middle = (0 + 0) / 2 = 0
compResult is positive
return binarySearch(kidNames, “Daryn”, 1, 0)

binarySearch(kidNames, “Daryn”, 1, 0)

items: kidNames
target: “Daryn”
first: 1
last: 0
middle = (1 - 0) / 2 = 0
(first > last)
return -1

5.3.3

If there are multiple occurrences of the target item in an array, what can you say about the subscript value that will be returned by `linearSearch`? Answer the same question for `binarySearch`.

The `linearSearch` will return the first occurrence of multiple occurrences of the target. The `binarySearch` will return some occurrence, but it is not possible to predict which one.

5.3.5

Write a recursive algorithm that searches a string for a target character and returns the position of its first occurrence if it is present or -1 if it is not.

```
Search(s, t)
        search(s, t, 0)

search(s, t, i)
        if i == s.length() return -1
        else if t == s.charAt(i) return i
        else return search(s, t, i+1)
```

5.4.1

Describe the result of executing each of the following statements:

```
LinkedListRec<String> aList = new LinkedListRec<String>();
aList.add("bye");
aList.add("hello");
System.out.println(aList.size() + ", " + aList.toString());
aList.replace("hello", "welcome");
aList.add("OK");
aList.remove("bye");
aList.remove("hello");
System.out.println(aList.size() + ", " + aList.toString());
```

`LinkedListRec<String> aList = new LinkedListRec<String>();`
Creates a LinkedListRec that will hold String objects and assigns a reference to it to aList.
`aList.add("bye");`
Appends the string "bye" to the end of aList
`aList.add("hello");`
Appends the string "hello" to the end of the list
`System.out.println(aList.size() + ", " + aList.toString());`
Outputs the following: `2, bye\nhello\n\n`
`aList.replace("hello", "welcome");`
Replaces the value "hello" in the list with "welcome"
`aList.add("OK");`
Appends the string "OK" to the end of the list
`aList.remove("bye");`
Removes the entry containing "bye" from the list, returns true
`aList.remove("hello");`
Does nothing, since "hello" was replaced by "welcome" earlier, returns false.

```
System.out.println(aList.size() + ", " + aList.toString());
```
Outputs the string `2, welcome\nOK\n\n`

5.4.3

Write a recursive algorithm for method `insert(E obj, int index)` where `index` is the position of the insertion.

```
insert(E obj, int index)
        if (index == 0)
                head = new Node(obj, head)
        else
                insert(obj, head, index-1)

insert(E obj, Node<E> pred, int index)
        if (index == 0)
                pred.next = new Node(obj, head.next)
        else
                insert(obj, pred.next, index-1)
```

5.5.1

What is the big-**O** for the Towers of Hanoi as a function of *n*, where *n* represents the number of disks? Compare it to the function 2^n.

Towers of Hanoi requires $2n - 1$ moves where n is the number of disks. Thus it is $O(2^n)$

5.5.3

Provide a "trace" of the solution to a four-disk problem by showing all the calls to `showMoves` that would be generated.

```
showMoves(4, startPeg, destPeg)

showMoves(3, startPeg, otherPeg)
"Move disk 4 from peg " + startPeg + " to " + destPeg
showMoves(3, otherPeg, startPeg)

showMoves(2, startPeg, destPeg)
"Move disk 3 from peg " + startPeg + " to " + otherPeg
showMoves(2, destPeg, startPeg)
"Move disk 4 from peg " + startPeg + " to " + destPeg
showMoves(2, otherPeg, startPeg)
"Move disk 3 from peg " + otherPeg + " to " + startPeg
showMoves(2, startPeg, otherPeg)

showMoves(1, startPeg, otherPeg)
```

```
"Move disk 2 from peg " + startPeg + " to " + otherPeg
showMoves(1, otherPeg, startPeg)
"Move disk 3 from peg " + startPeg + " to " + otherPeg
showMoves(1, destPeg, startPeg)
"Move disk 2 from peg " + destPeg + " to " + startPeg
showMoves(1, startPeg, destPeg)
"Move disk 4 from peg " + startPeg + " to " + destPeg
showMoves(1, otherPeg, startPeg)
"Move disk 2 from peg " + otherPeg + " to " + startPeg
showMoves(1, startPeg, otherPeg)
"Move disk 3 from peg " + otherPeg + " to " + startPeg
showMoves(1, startPeg, otherPeg)
"Move disk 2 from peg " + startPeg + " to " + otherPeg
showMoves(1, otherPeg, startPeg)

"Move disk 1 from peg " + startPeg + " to " + otherPeg
"Move disk 2 from peg " + startPeg + " to " + otherPeg
"Move disk 1 from peg " + otherPeg + " to " + startPeg
"Move disk 3 from peg " + startPeg + " to " + otherPeg
"Move disk 1 from peg " + destPeg + " to " + startPeg
"Move disk 2 from peg " + destPeg + " to " + startPeg
"Move disk 1 from peg " + startPeg + " to " + destPeg
"Move disk 4 from peg " + startPeg + " to " + destPeg
"Move disk 1 from peg " + otherPeg + " to " + startPeg
"Move disk 2 from peg " + otherPeg + " to " + startPeg
"Move disk 1 from peg " + startPeg + " to " + otherPeg
"Move disk 3 from peg " + otherPeg + " to " + startPeg
"Move disk 1 from peg " + startPeg + " to " + otherPeg
"Move disk 2 from peg " + startPeg + " to " + otherPeg
"Move disk 1 from peg " + otherPeg + " to " + startPeg
```

5.6.1

The terminating conditions in `findMazePath` must be performed in the order specified. What could happen if the second or third condition was evaluated before the first? If the third condition was evaluated before the second condition?

We must first test to see if the cell referenced by x, y is in the grid before examining its color. If we did not do this, we might get an ArrayIndexOutOfBoundsException thrown by maze.getColor. Next we need to see if the referenced cell is the background color. It it is, we cannot move there. However, we could test the third condition before the second since the end cell is not a barrier.

5.6.3

Is the path shown in Figure 5.19 the shortest path to the exit? If not, list the cells on the shortest path.

Yes, it is the shortest path.