Recursion, Recursion, Recursion, ...

1 Digit Sum

Write a recursive method `digitSum` that takes a nonnegative integer in return for some of its digits. For example, `digitSum(1234)` returns $1 + 2 + 3 + 4 = 10$. Your method should take advantage of the fact that it is easy to break a number into two smaller pieces by dividing by 10 (ie, $1234/10 = 123$ and $1234\%10 = 4$).

For these methods, we do not need to construct any objects. Therefore, you can declare them to be `static` methods and call them directly from `main`:

```
public static int digitSum(int n) { ... }
```

2 Palindromes

Write a recursive method `isPalindrome` that takes a string and returns true if it is the same regardless of whether it is read forwards or backwards. For example,

```
isPalindrome("mom")    →  true
isPalindrome("cat")    →  false
isPalindrome("level")  →  true
```

The prototype for the method should be as follows:

```
public static boolean isPalindrome(String s)
```

3 Balancing Parentheses

In the syntax of most programming languages, there are characters that occur only in nested pairs, which are called bracketing operators. Java, for example, has these bracketing operators:

```
( . . . )
[ . . . ]
{ . . . }
```

In a properly formed program, these characters will be properly nested and matched. To determine whether this condition holds for a particular program, you can ignore all the other characters and look simply at the pattern formed by the parentheses, brackets, and braces. In a legal configuration, all the operators match up correctly, as shown in the following example:

```
{ ( [ ] ) ( [ ( ) ] ) }
```

The following configurations, however, are illegal for the reasons stated:

`( ( [ ] )`	The line is missing a close parenthesis.
`) (`	The close parenthesis comes before the open parenthesis.
`{ ( } )`	The parentheses and braces are improperly nested.

Write a recursive method

```
public static boolean isBalanced(String str)
```

that takes a string `str` from which all characters except the bracketing operators have been removed. The method should return `true` if the bracketing operators in `str` are *balanced*, which means that they are correctly nested and aligned. If the string is not balanced, the method returns `false`. Although there are many other ways to implement this operation, you should code your solution so that it embodies the recursive insight that a string consisting only of bracketing characters is balanced if and only if one of the following conditions holds:

- The string is empty.
- The string contains "`()`", "`[]`", or "`{}`" as a substring and is balanced if you remove that substring.

4 Print in Binary

Inside a computer system, integers are represented as a sequence of bits, each of which is a single digit in the binary number system and can therefore have only the value 0 or 1. The table below shows the first few integers represented in binary:

binary	decimal
0	0
1	1
1 0	2
1 1	3
1 0 0	4
1 0 1	5
1 1 0	6

Each entry in the left side of the table is written in its standard binary representation, in which each bit position counts for twice as much as the position to its right. For instance, you can demonstrate that the binary value 110 represents the decimal number 6 by following this logic:

$$\begin{array}{rcccccccc} \text{place value} & \rightarrow & 4 & & 2 & & 1 & & \\ & & \times & & \times & & \times & & \\ \text{binary digits} & \rightarrow & \mathtt{1} & & \mathtt{1} & & \mathtt{0} & & \\ & & \downarrow & & \downarrow & & \downarrow & & \\ & & 4 & + & 2 & + & 0 & = & 6 \end{array}$$

Basically, this is a base-2 number system instead of the decimal (base-10) system we are familiar with. Write a recursive method

```
public static void printInBinary(int number)
```

that prints the binary representation for a given integer. For example, calling `printInBinary(3)` would print `11`. Your method may assume the integer parameter is always non-negative.

The recursive insight to solve this problem is to use the fact that you can identify the least significant binary digit by using the modulus operator with value 2. For example, given the integer 35, mod by 2 tells you that the last binary digit must be 1 (i.e. this number is odd), and division by 2 gives you the remaining portion of the integer (17). What is the simplest possible number(s) to print in binary that identifies the base case that stops the recursion? One slightly tricky part about the recursive decomposition here is that you have to think about it a bit backwards. There is a straightforward way to easily identify and print the last binary digit, but you need to print that digit only after you have printed all the other binary digits. This dictates the placement of the printing relative to the recursion.

You will probably want to use the method `System.out.print` in the problem. It is just like `println`, but does not start a new line of output.

5 Substrings

Write a method

```
public static void substrings(String str)
```

that prints out all subsets of the letters in `str`. Example:
`substring("ABC")` → "", "A", "B", "C", "AB", "AC", BC", "ABC"
It does not matter what order they are printing in. You may find it useful to write a helper method

```
public static void substringHelper(String str, String soFar)
```

that is initially called as `substringHelper(str, '''')`. The variable `soFar` keeps track of the characters currently in the substring you are building. To process `str`, recursively build substrings containing the first character of `str` by adding that character to `soFar` and recursing on the remainder of `str`. Also, build all substrings not containing the first character by recursing on the remainder of `str` without adding the first character to `soFar`. In other words, you will perform two recursive calls inside `substringHelper`. The variable `soFar` will contain one possible substring to be printed when `str` has no more characters in it.

6 Subset Sum

The subset sum problem is an important and classic problem in computer theory. Given a set of integers and a target number, your goal is to find a subset of those numbers that sum to the target number. For example, given the set $\{3, 7, 1, 8, -3\}$ and the target sum 4, the subset $\{3, 1\}$ sums to 4. On the other hand, if the target sum were 2, the result is false since there is no subset that sums to 2. The prototype for this method is:

```
public static boolean canMakeSum(int setOfNums[], int targetSum)
```

Assume that the array contains `setOfNums.length` numbers (ie, it is completely full). Remember that many recursive problems are variations on the same common themes. Consider how this problem is related to the string subset example. You should be able to fairly easily adapt it to operate on an array of numbers instead of a string. Note that you are not asked to print the subset members, just return true/false. You will likely need a wrapper method to pass additional state through the recursive call what is the other information you need to track as you try various subset combinations?

7 Extending Subset Sum

You are to make two changes to the `canMakeSum` method:

- Change the method to print the members in the successful subset if one is found. Do this without adding any new data structures (i.e. don't build a second array to hold the subset). Just use the unwind of the recursive calls.

  ```
  public static boolean printSubsetSum(int nums[], int targetSum)
  ```

- Change the method to report not just whether any such subset exists, but the count of all such possible subsets. For example, in the set shown earlier, the subset 7, -3 also sums to 4, so there are two possible subsets for target 4.

  ```
  public static int countSubsetSumSolutions(int nums[], int targetSum)
  ```

8 Regular Expressions

Regular expressions are a widely-used construct for string matching. A regular expression is a succinct way of describing a pattern to match, such as all filenames that end in ".txt" or all names containing a "Z". A regular expression pattern is represented as a string. Ordinary characters within the pattern indicate where the input string must exactly match. The pattern may also contain *wildcard* characters, which specify how and where the input string is allowed to vary. Wildcard characters have the following meanings:

1. The '*' wildcard character matches any sequence of characters (empty or not).
2. The '?' wildcard character matches either zero or one characters.

You are to write the recursive method

```
public static boolean matches(String pattern, String str)
```

that returns whether the input string `str` matches the regular expression `pattern`. Here are some examples:

```
matches("*.txt", "file.txt")    →  true
matches("*.txt", ".txt")        →  true
matches("*.txt", "txt.file")    →  false
matches("moo?s", "moons")       →  true
matches("moo?s", "moos")        →  true
matches("moo?s", "moorings")    →  false
matches("c*t?r*", "computers")  →  true
```

There are several different approaches you could use to recursively decompose this problem. Try to think through what smaller, similar subproblems exist within the problem and how their solution could be useful in solving the entire problem.

9 Longest Common Subsequence

A subsequence of a string `s` contains some of the letters from `s` in the same order as they originally appeared. For example, "wam," "ills," and "wlim" are all subsequences of "williams." The longest common subsequence of two strings `s` and `t` is the longest string that is both a subsequence of `s` and a subsequence of `t`. You are to write a method

```
public static String longestCommonSub(String s, String t)
```

that computes the longest common subsequence of two strings. Here are a few examples:

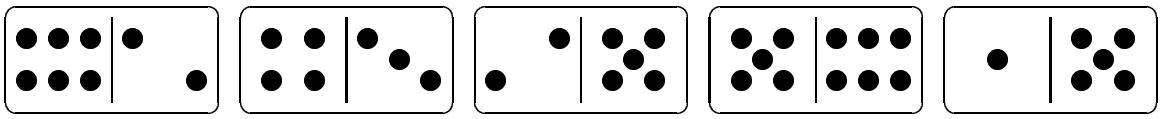
```
longestCommonSub("moo", "cow")              →  "o"
longestCommonSub("melody", "monkey")        →  "mey" (or "moy"- both are valid LCS's)
longestCommonSub("recursion", "c-u-r-s-e")  →  "curs"
longestCommonSub("cs136", "moo")            →  ""
```

If the solution is not unique, just return any of the longest common subsequences. If no common subsequence exists, simply return the empty string "".

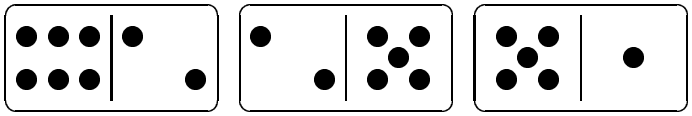
There are a number of ways to write this method recursively. Think about what smaller subproblems exist during a call to `longestCommonSub(s,t)` and how they may be used to find the overall solution.

10 Dominos

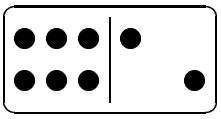
The game of dominos is played with rectangular pieces composed of two connected squares, each of which is marked with a certain number of dots. For example, each of the following rectangles represents five dominos:



Dominos are connected end-to-end to form chains, subject to the condition that two dominos can be linked together only if the numbers match, although it is legal to rotate dominos 180 degree so that the numbers are reversed. For example, you could connect the first, third, and fifth dominos in the above collection to form the following chain:



Note that the 1-5 domino had to be rotated so that it matched up correctly with the 2-5 domino. Given a set of dominos, an interesting question to ask is whether it is possible to form a chain starting at one number and ending with another. For example, the example chain shown earlier makes it clear that you can use the original set of five dominos to build a chain starting with a 6 and ending with a 1. Similarly, if you wanted to build a chain starting with a 6 and ending with a 2, you could do so using only one domino:



On the other hand, there is no way using just these five dominos to build a chain starting with a 4 and ending with a 6. Dominos can be represented in Java very easily as a pair of integers (you will probably want to create a `Domino.java` file to contain a class similar to this one):

```
public class Domino {
    protected int v1, v2;
    public Domino(int v1, int v2) {
        this.v1 = v1;
        this.v2 = v2;
    }
    public int firstSide() { return v1; }
    public int secondSide() { return v2; }
}
```

Write a method

```
static public boolean chainExists(Vector dominos, int start, int finish)
```

that returns `true` if it is possible to build a chain from `start` to `finish` using any subset of the dominos in the vector `dominos` and `false` otherwise. For example, if dominos is the domino set illustrated above, calling `chainExists` would give the following results:

```
chainExists(dominos, 6, 1)  →  true
chainExists(dominos, 6, 2)  →  true
chainExists(dominos, 4, 6)  →  false
```

Note: A domino can be used at most once in building a chain. You will need some mechanism for

either marking a domino in the vector as used or removing used dominos from the array. There are several different ways to do this (adding a field to `Domino` class, shortening the vector by removing the dominos as they are being used, etc.). You are free to take whatever approach you prefer, subject to the constraint that the values of the elements in the vector must be the same after calling your method as they are beforehand. So, if you change the vector or dominos during processing, you need to change them back.