Prelab Preparation. Please read through this lab handout before lab.

This week’s lab is structured as several small problems that can be solved in isolation. Recursion can be a difficult concept to master and one that is worth concentrating on separately before using it in large programs. The goals of this lab are:

• To practice writing recursive programs; and
• To solve a variety of interesting algorithmic problems.

Recursive solutions can often be formulated in just a few concise, elegant lines, but they can be very subtle and hard to get right. Recursion is a tricky topic so don’t be dismayed if you can’t immediately sit down and code these perfectly the first time. Take time to figure out how each problem is recursive in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the same problem. You will need to depend on a recursive “leap of faith” to write the solution in terms of a problem you haven’t solved yet. Be sure to take care of your base case(s) lest you end up in infinite recursion. The great thing about recursion is that once you learn to think recursively, recursive solutions to problems seem very intuitive. Spend some time on these problems and you’ll be much better prepared when you are faced with more sophisticated recursive problems.

Use the debugger as often as possible.

Warm-ups. To help get you started, we present two warm-up problems, digit sum and subset sum. You should come to lab with a written design for each one. Feel free to discuss the details of the warm-ups with other students as well as the staff. The goal of the warm-ups is to practice recursion fundamentals before lab on Wednesday.

Digit Sum. Write a recursive method digitSum that takes a non-negative integer in return for the sum 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 (that is, 1234/10 = 123 and 1234%10 = 4). Recall that integer division in Java truncates (not rounds) decimal digits.

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:

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

```java
public static void main (String args[]) { ... }
```

```
int sum = digitSum(1234);
```

Subset Sum. Subset Sum 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 is:

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

Assume that the array contains setOfNums.length numbers (i.e., it is completely full). 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 calls. What additional state would be useful to track?
Getting Started. We provide basic starter code for this assignment. To obtain it, download the file Recursion.java from the course webpage. All of your code will go into this file.

Lab Programs. For each problem below, you must thoroughly test your code to verify that it correctly handles all the necessary cases. For example, for the “Digit Sum” warm-up, you could use test code to call your method in a loop on the first 50 integers or use a loop to allow the user to repeatedly enter numbers that are fed to your method until you are satisfied. Testing is necessary to be sure you have handled all the different cases. You can leave your testing code in the file you submit—there is no need to remove it this week.

For each exercise, we specify the method signature. **Your method must exactly match that prototype** (same name, same arguments, and same return type). You will want to add additional helper methods for a number of these questions. **Your solutions must be recursive**, even if you can come up with an iterative alternative. Before starting, copy the starter files.

Counting Cannonballs. Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth. Write a recursive method `countCannonballs` that takes as its argument the height of a pyramid of cannonballs and returns the number of cannonballs it contains. The prototype for the method should be as follows:

```java
public static int countCannonballs(int height)
```

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

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

The prototype for the method should be as follows:

```java
public static boolean isPalindrome(String str)
```

You may assume the input string contains no spaces.

Substrings. Write a method:

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

that prints out all subsets of the letters in `str`. For example:

- `substring("ABC") → "", "A", "B", "C", "AB", "AC", "BC", "ABC"`

Printing order does not matter. You may find it useful to write a helper method:

```java
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` you must: 1) build all substrings containing the first character (which you do by including that character in `soFar`), and 2) build all substrings not including the first character. When `str` has no more characters in it, it will be one possible substring.

Print in Binary. Computers represent integers as sequences of bits. A bit 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:
<table>
<thead>
<tr>
<th>binary</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>3</td>
</tr>
<tr>
<td>1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>1 0 1</td>
<td>5</td>
</tr>
<tr>
<td>1 1 0</td>
<td>6</td>
</tr>
</tbody>
</table>

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:

\[
\begin{array}{ccc}
\text{place value} & \rightarrow & 4 & 2 & 1 \\
\times \times \times \\
\text{binary digits} & \rightarrow & 1 & 1 & 0 \\
\downarrow \downarrow \downarrow \\
4 + 2 + 0 &= 6
\end{array}
\]

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

```java
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.

Hint: You can identify the least significant binary digit by using the modulus operator with value 2 (i.e., `number % 2`). This tells you the remainder after dividing by 2. For example, given the integer 35, the value 35%2 = 1 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).

**Extending Subset Sum.** You are to write a modified version of the `canMakeSum` method as follows. Change the method to print the members in a 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.

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

**Extra Credit: Music Generation.** A guitarist is writing a song, and has written a chord sequence, but needs a single note accompaniment (a bass or vocal line, for example). Your task is to generate such an accompaniment.

The standard Western musical scale is made up of a sequence of notes that correspond to 12 unique tones within an **octave**; these tones are represented by a repeating sequence of notes denoted by letters from A to G. There is an A note that has twice the frequency of another A note (and therefore has a higher pitch). Furthermore, between each note exists a half-step called a **flat** or **sharp** respectively indicated by a b or #. b and # indicate that a note is lowered or raised by a half-step respectively. The spacing between all notes is not even. The following are the 12 unique tones within one octave: A A#/Bb B B#/C C#/Db D D#/Eb E E#/F F#/Gb G G#/Ab. Notes with slashes can be represented in both forms and are equivalent notes.

A major scale, given a starting note X, is the sequence of notes (counted in half-steps): X, X+2, X+4, X+5, X+7, X+9, X+11, X+12. Note that X+12 is the same note as X one octave higher. A harmonic minor scale consists of the same notes, except the third and sixth notes in the sequence are lowered by one half-step. **Chords** are sets of notes played simultaneously that are associated with a particular major or minor scale.

A simple accompaniment can consist of a sequence of notes on a major or minor scale that corresponds with the current chord being played. An accompaniment is uninteresting if it simply plays a scale along with
a chord sequence; in general, certain transitions are usually more favorable than others. For example, given a note, the transition from the first note to the fifth note on the scale is often considered to be good.

Given a chord sequence, transitions and their weights, and the number of notes to be generated to accompany each chord, you are to output the best possible (or one of the best, if multiple exist) accompaniment (sequence of notes) and its total weight, all separated by whitespace; the best such sequence is the one with maximum total weight. Assume the input allows for at least one sequence.

The first line of input will contain a chord sequence. Minor chords are denoted with a m after the chord name. The second line of input will contain the number of notes you are to generate to accompany each chord. The third line of input will contain the transition values. Each transition value is a pair of integer values X Y where X is the transition distance, $-11 \leq X \leq 11$, that represents the number of half-steps of the transition, and Y, a positive value that represents the weight of the transition. The line may contain any number of pairs separated by whitespace.

When generating a sequence for a particular chord, only the transitions that move from the current note to another note on the appropriate scale are considered valid. This means that if you are generating a sequence for C major, and are on the note C, the transition 2 10 can be used to move up to D, but if you are on the note E, the same transition cannot be used to move up to F#, because F# is not on the C major scale.

Place your code in the provided file MusicGen.java.

Sample input:

```
C G Am F
4
-7 3 -4 6 -2 2 7 4 2 5
```

Sample output:

```
E C D E C D E C Ab E C Ab E C D Bb 85
```

**Submission.** Submit the file Recursion.java using the turnin utility before midnight on the due date. If you wrote additional code or classes for the extra credit problem, submit them as well.

As in all labs, you will be graded on design, documentation, style, and correctness. Be sure to document your program with appropriate comments, including a general description at the top of the file and a description of each method. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

Since we are learning about running times and big-O analysis in lecture, for this lab, **include the running time (in big-O notation) for each method in the comments above the method.**