Lab 3

Due 26 February

Handout 4
CSCI 136: Spring, 2007
19 February

Recursion, Recursion, Recursion, ...

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.

Each problem will have a fairly short solution, but that doesn't mean you should put this assignment off until the last minute though. 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.

1 Warm-ups

First, we present two warm-up problems. You should **come to lab with a written design for each one**. We will discuss them at the beginning of lab. Try to work through them by yourself, and if you get stuck, ask for help from me or the TA's. 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.

1.1 Digit Sum

Write a recursive method `digitSum` that takes a non-negative 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) { ... }
```

1.2 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 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). 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?

2 Lab Programs

For each problem below, you must to thoroughly test your code to verify 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.

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.

2.1 Counting Cannonballs

Before starting, copy the starter files, as described in Section 3.

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:

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

2.2 Palindromes

Write a recursive method `isPalindrome` that takes a string and returns true if 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 str)
```

You may assume the input string contains no spaces.

2.3 Balancing Parentheses

In the syntax of most programming languages, there are characters that occur only in nested pairs, 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.

For example, the string "`[(){}]`" is shown to be balanced by the following chain of calls:

```
isBalanced("[(){}]")  →
 isBalanced("[{}]")   →
  isBalanced("[]")    →
   isBalanced("")     →  true
```

2.4 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"

Printing order does not matter. 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` 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 susbstring.

2.5 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:

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.

Hint: You can identify the least significant binary digit by using the modulus operator with value 2 (ie., `number % 2`). For example, given the integer 35, the value $35\%2 = 1$ tells you that the last binary digit must be 1 (ie., this number is odd), and division by 2 gives you the remaining portion of the integer (17).

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

2.6 Extending Subset Sum

You are to write two modified versions of the `canMakeSum` method:

- 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.

  ```
  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. You do not need to print all of the subsets.

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

- **Optional Bonus Part**: Change the method to print all susbsets that sum to `targetSum`.

2.7 Cell Phone Mind Reading

Entering text using the digit keys on a phone is problematic, in that there are only 10 digits for 26 letters and thus each digit key is mapped to several letters. Some cell phones require you to "multitap": tap the 2 key once for 'a', twice for 'b' and three times for 'c', which gets tedious.

Technology like Tegic's T9 predictive text allows the user to press each digit key once and based on the user's sequence so far, it guesses which letters were intended, having found the possible completions for the sequence. For example, if the user types the digit sequence "72", there are nine possible mappings (pa, pb, pc, ra, rb, rc, sa, sb, sc). Three of these mappings seem promising (pa, ra, sa) because they are prefixes of words such as "party" and "sandwich", while the other mappings can be ignored since they lead nowhere. If the user enters "9956", there are 81 possible mappings, but you can be assured the user meant "xylo" since that is the only mapping that is a prefix of any English words.

You are to implement an algorithm to find the possible completions for a digit sequence. The `listCompletions` method is given the user's digit sequence and a `Lexicon` object. The method will print all English words that can be formed by extending that sequence. For example, here is the list of completions for "72547":

```
palisade
palisaded
palisades
palisading
palish
rakis
rakish
rakishly
rakishness
sakis
```

We will provide a `Lexicon` class that serves as a dictionary of English words for you to use. Your recursive method will take the lexicon as a parameter. Here are the important parts of the lexicon interface:

```
public class Lexicon {
  /**
   * Loads a lexicon from the specified file.
   */
  public Lexicon(String fileName)

  /**
   * Returns true if the word is contained in the lexicon.
   */
  public boolean contains(String word)

  /**
   * This method returns true if any words in the lexicon begin with the
   * specified prefix, false otherwise. A word is defined to be a prefix of
   * itself and the empty string is a prefix of everything.
   */
  public boolean containsPrefix(String prefix)

  ...
}
```

Some hints to get you started:

- Start by reviewing the `listMnemonics` method from lecture. That method lists all possible mnemonics for the input number sequence, and is a very good starting point.
- `listCompletions` consists of two recursive pieces. They require similar, but not identical, code. The first is almost the same as `listMnemonics`, which gives you a way to convert the digit sequence into letters. Here, though, rather than printing each mnemonic found, we want to pass them to the second recursive method, which will extend that prefix in an attempt to build words.

In the first case, the choices for the letters are constrained by the digit-to-letter mapping. In the second, what are the possible choices for letters that could be used to extend the sequence to built a completion?

- Be sure to take advantage of the Lexicon member function `containsPrefix`. This allows you check whether a sequence of letters is the prefix of any word contained in the lexicon. You will need to use this to avoid going down dead ends.

Your program should first create the lexicon and then pass it, along with a digit sequence to your method:

```
public static void listCompletions(String digitSequence, Lexicon lex) { ... }

public static void main(String args[]) {
  ...
  Lexicon lexicon = new Lexicon("lexicon.dat");
  listCompletions("72547", lexicon);
  ...
}
```

3 Getting Started

We provide basic starter code for this assignment. To obtain it, execute the following command to copy the recursion lab folder to your own directory:

```
cp -r /usr/mac-cs-local/share/cs136/labs/recursion .
```

The `recursion` directory contains the following files:

- `Recursion.java`: All of your code will go into this file, and you will not need to change the other files.
- `Lexicon.class`: The Lexicon library class for question 7.
- `Lexicon.dat`: The Lexicon data file.

4 Submission

Submit the file `Recursion.java` using the `turnin` utility before midnight on the due date. If you wrote additional classes for the bonus problems, 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, a description of each method with pre- and post-conditions where appropriate. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

5 Bonus Questions

The following questions are designed to show the power of recursion through a number of more challenging problems. They are not part of the required assignment, but feel free to give them a try. They will be worth a small amount of extra credit.

5.1 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 longestCommonSubsequence(String s, String t)
```

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

```
longestCommonSubsequence("moo", "cow")               →  "o"
longestCommonSubsequence("melody", "monkey")         →  "mey" (or "moy"- both are valid LCS's)
longestCommonSubsequence("recursion", "c-u-r-s-e")   →  "curs"
longestCommonSubsequence("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 `longestCommonSubsequence(s,t)` and how they may be used to find the overall solution.

5.2 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)
```

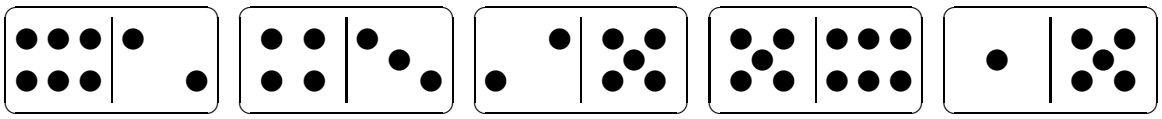
that returns whether the input string `str` matches the regular expression `pattern`. Here are some examples:

```
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
```

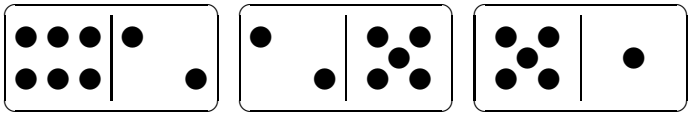
There are several different approaches you could use to recursively decompose this problem. Try to think through what smaller, similar sub-problems exist within the problem and how their solution could be useful in solving the entire problem.

5.3 Dominos

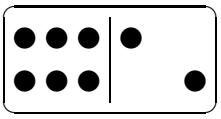
The game of dominos is played with rectangular pieces composed of two connected squares, each of which is marked with a certain number of dots. For example, each of the following rectangles represents five dominos:



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 side1, side2;
    public Domino(int side1, int side2) {
        this.side1 = side1;
        this.side2 = side2;
    }
    public int firstSide() { return side1; }
    public int secondSide() { return side2; }
}
```

Write a method

```
static public boolean chainExists(Domino 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 array `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 array 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 array 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 array must be the same after calling your method as they are beforehand. So, if you change the array or dominos during processing, you need to change them back.