Computer Science 1 - Program 3
KnightsRecurse (Recursion)
Assigned: 2/8/12
Due: 2/22/12 (Wednesday) by 11:55 pm (WebCourses time)

Objective
Make use of recursion to solve problems.

The Problem
Below are a series of problems you need to solve using **recursive** functions. You will write a program that will read commands from an input file, with each command referring to one of the recursive problems to be executed. Each command will be followed (on the same line of input) by the respective parameters required for that problem.

Implementation
Each recursive function MUST have a wrapper function enclosing it where you will do input/output file processing as well as calling the actual recursive function. From Wikipedia, "A wrapper function is a function in a computer program whose main purpose is to call a second function". As an example, here is one of the wrapper functions you will use:

```
void KnightsFlip(FILE *fin, FILE *fout);
```

This would be the wrapper function for one of the recursive problems, called **KnightsFlip** (described, in detail, below). These wrapper functions will simply do three things:

1. deal with the processing of the input file
2. most importantly, the wrapper function will make the initial call to your recursive function that will actually solve the problem.
3. Finally, the wrapper functions will print to the output file. Note: printing to the output file may also occur in the actual recursive functions.

Of course, at your own discretion (meaning, your own choice), you can make auxiliary functions for your recursive functions to use, such as functions to swap values in an array, take the sum of an array of values, printing functions, etc.

> ***HINT**: I will post, on the Discussion boards, a small recursive function that uses a wrapper function as described above.

Five Problems to Solve Using Recursion:

(1) KnightsMultiply

Write a recursive function that multiplies all values from *k* up to *n* and returns the result (as a double). For example: if *k* = 5, and *n* = 10, then multiply 5 * 6 * 7 * 8 * 9 * 10 to return 151200.

(2) KnightsFlip

You have an index card with the letter K written on one side, and F on the other. You want to see ALL of the possible ways the card will land if you drop it *n* times. Write a recursive function that prints each session of dropping the cards with K's and F's. For example of you drop it 4 times in a given session, all possible ways to drop it are as follows:

```
KKKK
KKKF
KKFK
KKFF
KFKK
KFKF
KFFK
KFFF
FKKK
FKKF
FKFK
FKFF
FFKK
FFKF
FFFK
FFFF
```

*Note: The possible ways would have to print in this specific order.

(3) KnightsShape

Simply write a recursive function that prints a large X composed of smaller X's with a given "width", *n*, which is guaranteed to be odd. Example for an X of width *n* = 7:

```
X       X
 X     X
  X   X
   X X
    X
   X X
  X   X
 X     X
X       X
```

*Note: to be clear, the "width" is the length (number) of X's along one line of the large X.

(4) KnightsScotch

No, this has nothing to do with drinking Scotch! Suppose you are playing a special game of hopscotch on a line of integer numbers drawn on the ground like so:

4 4 1 5 2 6 3 4 2 0

(Although playing this game, while drinking Scotch, would certainly be entertaining!)

The number, with a square around it, indicates where you are currently standing. You can move left or right down the line by jumping the number of spaces indicated by the number you are standing on. So if you are standing on a 4, you can jump either left 4 spaces or right 4 spaces. You cannot jump past either end of the line.

For example, the first number (4) only allows you to jump right, since there are no numbers to the left that you can jump to.

The goal: you want to get to the 0 at the far end (right side) of the line. You are also guaranteed that there will be only one zero, which, again, will be at the far right side.

Here is how you do that with the above line:

Starting position **[4]** 4 1 5 2 6 3 4 2 0

Step 1: Jump right 4 4 1 5 **[2]** 6 3 4 2 0

Step 2: Jump left 4 4 **[1]** 5 2 6 3 4 2 0

Step 3: Jump right 4 4 1 **[5]** 2 6 3 4 2 0

Step 4: Jump right 4 4 1 5 2 6 3 4 **[2]** 0

Step 5: Jump left 4 4 1 5 2 6 **[3]** 4 2 0

Step 6: Jump right 4 4 1 5 2 6 3 4 2 **[0]**

Some KnightsScotch lines have multiple paths to 0 from the given starting point. Other lines have no paths to 0, such as the following:

[3] 1 2 3 0

In this line, you can jump between the 3's, but not anywhere else.

You are to write a recursive function that returns an integer 1 (for solvable) or 0 (for not solvable), indicating if you are able to get to the rightmost 0 or not.

(5) KnightsDepot

You are working on an engineering project with other students and are in need of 2x4s (2 by 4's) of various lengths. Thankfully, UCF has its very own "Depot" store: KnightsDepot…so you don't have to drive far. Unfortunately, KnightsDepot only carries 2x4s in fixed lengths. So your team will have to purchase the 2x4s and then cut them as needed. Because of your tight college student budget, you want to buy the **least** number of 2x4s in order to cover the lengths requested by your team.

The recursive function you are designing will return the minimum number of 2x4s you need to purchase, in order to cover the lengths requested by your team. Example:

Array of 2x4 lengths requested by your team: **{ 6, 4, 3, 4, 1, 3 }**
Fixed length of 2x4s at KnightsDepot: **6** (so each purchased 2x4 has a length of 6)
A possible arrangement of of 2x4s: **{ 6 } { 3, 3 } { 4, 1 } { 4 }**
Minimum number of 2x4s needed to purchase: **4**

Wrapper Functions

As mentioned, you MUST use the following five wrapper functions EXACTLY as shown:

```
void KnightsMultiply(FILE *fin, FILE *fout);
void KnightsFlip(FILE *fin, FILE *fout);
void KnightsShape(FILE *fin, FILE *fout);
void KnightsScotch(FILE *fin, FILE *fout);
void KnightsDepot(FILE *fin, FILE *fout);
```

Input File Specifications

You will read in input from a file, "KnightsRecurse.in". Have this AUTOMATED. Do not ask the user to enter "KnightsRecurse.in". You should read in this automatically (this will expedite the grading process). The file will contain an arbitrary number of lines and on each line there will be a command. Each command will be followed by relevant data as described below (and this relevant data will be on the same line as the command).

We will not tell you the number of commands in the file. So how do you know when you've read all the commands in the input file? You can use the <stdlib.h> function feof(FILE * file) to determine if you have reached the end of the file.

The commands (for the five recursive functions), and their relevant data, are described below:

`KnightsMultiply` - This command will be followed by the following information: k and then n, as described above.

`KnightsFlip` - This command will be followed by a single integer, n, as described above.

`KnightsShape` - This command will be followed by a single integer, n, as described above.

`KnightsScotch` - This command will be followed by an integer, `start`, representing the initial position on the line you are playing on (0 means the far left, the first position); `size`, the number of integers drawn on the ground; `size` will be followed by `size` number of integers, the integers drawn on the ground. (see input file for examples)

`KnightsDepot` - This command will be followed by an integer `maxlen`, which represents the maximum 2x4 length the store sells; `size`, the number of 2x4s your team needs; `size` will be followed by `size` number of integers, the length of each 2x4 your team needs. It is guaranteed that the requested lengths will all be less than or equal to the `maxlen` sold by the store.

Output Format

Your program must output to a file, called **"KnightsRecurse.out". You must follow the program specifications exactly.** Refer to sample output file for exact formatting specifications.

*****WARNING*****

Your program MUST adhere to the EXACT format shown in the sample output file (spacing capitalization, use of dollar signs, periods, punctuation, etc). The graders will use very large input files, resulting in very large output files. As such, the graders will use text comparison programs to compare your output to the correct output. If, for example, you have two spaces between in the output when there should be only one space, this will show up as an error even through you may have the program correct. You WILL get points off if this is the case, which is why this is being explained in detail. Minimum deduction will be 10% of the grade, as the graders will be forced to go to text editing of your program in order to give you an accurate grade. Again, your output MUST ADHERE EXACTLY to the sample output.

Grading Details

Your program will be graded upon the following criteria:

1) Adhering to the implementation specifications listed on this writeup.
2) Your algorithmic design.
3) Correctness.
4) **Use of recursion. If your program does not use recursion, you will NOT get credit for the assignment. Period.**
5) The frequency and utility of the comments in the code, as well as the use of white space for easy readability. (We're not kidding here. If your code is poorly commented and spaced and works perfectly, you could earn as low as 85-90% on it.)
6) Compatibility to CodeBlocks. (If your program does not compile in CodeBlocks, you will get a sizable deduction from your grade.)
7) Your output MUST adhere to the EXACT output format shown in the sample output file.

Restrictions

Name the file you create and turn in KnightsRecurse.c. Although you may use other compilers, your program must compile and run using CodeBlocks. Your program should include a header comment with the following information: your name, course number, section number, assignment title, and date. You should also include comments throughout your code, when appropriate.

Deliverables

A single source file named KnightsRecurse.c turned in through WebCourses.