CS 202 – Computer Science II
Spring 2017
Chapter 15 Programming Assignment (pa15)
100 points
Due by 2:00pm, 2017-04-14

Problem description

You will write several small recursive functions for this assignment, then demonstrate the functions by calling them from your `main` function. NOTE: My restriction on one return per function does not apply to recursive functions. Recursive functions may require multiple returns.

Patterns

1. (10 points) Write a recursive function, `pattern`, that takes as its parameter a non-negative integer `n` and generates the following pattern of asterisks. If `n` is 5, then the pattern generated is:

```
*
**
***
****
*****
```

2. (10 points) Write a recursive function, `triangle`, that takes as its parameters a non-negative integer `m` (the beginning number of asterisks) and a non-negative integer `n` (the maximum number of asterisks). The first line contains `m` asterisks, the next line contains `m + 1` asterisks, and so on, up to a line with `n` asterisks. Then the pattern is repeated backwards, going back to down to `m`. For example, given the values `m = 3` and `n = 7`, the function will print the following:

```
***
****
*****
******
*******
*******
******
*****
****
***
```

Hint: Only one of the arguments changes in the recursive call. Which one?

3. (10 points) Write a recursive function, `fractal_pattern`, that takes as its parameters two non-negative integers, `n` and `i`. The parameter `n` is the number of asterisks to be printed in the longest line and is a power of two (e.g. `n` might be 1, 2, 4, 8, 16, etc.), and `i` is the column number where the asterisks should start printing, where the leftmost column is column 0. For example, the following pattern is generated when `n` is 8 and `i` is 0:

```
* 
* * 
  * 
* * * * 
    * 
    * * 
      * 
* * * * * * * * 
        * 
        * * 
          * 
        * * * * 
            * 
            * * 
              * 
```

   Hints: Think about how the pattern is a fractal. Can you find two smaller versions of the pattern within the large pattern? Here is some code that may be useful within your function. (Note that there must be exactly one space after each asterisk, including the final asterisk on each line.)

```
// A loop to print exactly i spaces:
for (k = 0; k < i; k++) outs << ' ';
// A loop to print n asterisks, each one followed by a space:
for (k = 0; k < n; k++) outs << "* ";
```

Arrays

4. (10 points) Write a recursive template function, `sum_array`, that finds and returns the sum of the elements of an array. For example, given the int array { 1, 2, 3, 4, 5 }, the function will return 15.

5. (10 points) Write a recursive template function, `print_array`, that prints an array in element order. For example, given the array { 1, 2, 3, 4, 5 }, the function would print:

```
1 2 3 4 5
```

6. (10 points) Write a recursive template function, `reverse_array`, that reverses an array *in place.* For example, given the array { 1, 2, 3, 4, 5 }, the function will transform the array so that it becomes { 5, 4, 3, 2, 1 }. Using the `print_array` function from step 4 to display the array before and after the call to `reverse_array`.

Exhaustive Searches and Backtracking

Exhaustive searches are also known as backtracking algorithms, though not all backtracking algorithms are exhaustive. The algorithm looks for every possible way to search for a solution. It is usually combined with pruning to reduce the number of items to search for. A generalized pseudocode algorithm for the exhaustive search can be defined as:

```
1. Test whether solution has been found
2. If found solution, return it
3. Else for each choice that can be made:
    a. Make that choice
    b. Recur
    c. If recursion returns a solution, return it
4. If no choices remain, return failure
```

7. (20 points) Write a recursive function with this prototype:

   ```
   void section_numbers(const string& prefix, int levels);
   ```

   The output consists of the string `prefix` followed by "section numbers" of the form 1.1., 1.2., 1.3., and so on. The `levels` argument determines how may levels the section numbers have. For example, if `levels` is 2, then the section numbers have the form x.y. If levels is 3, then section numbers have the form x.y.z. The digits permitted in each level are always '1' through '9'. As an example, if prefix is the string "CS202-" and `levels` is 2, then the function would start by printing:

   ```
   CS202-1.1.
   CS202-1.2.
   CS202-1.3.
   ```

   and end by printing:

   ```
   CS202-9.7.
   CS202-9.8.
   CS202-9.9.
   ```

   The stopping case occurs when `levels` reaches zero (in which case the `prefix` is printed once by itself followed by nothing else).

   The string class has many manipulation functions, but you'll need only the ability to make a new string which consists of `prefix` followed by another character (such as '1') and a period ('.'). If s is the string that you want to create and c is the digit character (such as '1'), then the following statement will correctly form s:

   ```
     s = prefix + c + '.';
   ```

   This new string s can be passed as a parameter to recursive calls of the function.

8. (20 points) Write a recursive function to decide whether it is *possible* to win the Jelly Bean Game. The game starts when you are given *n* jelly beans. You can then give back some jelly beans, but you must follow these rules where *n* is the number of jelly beans that you have:

   - If *n* is even, then you *may* give back exactly *n*/2 jelly beans.

- If *n* is divisible by 3 or 4, then you *may* multiply the last two digits of *n* and give back this many jelly beans. (By the way, the last digit of *n* is n % 10, and the next-to-last digit is (n % 100) / 10.)
- If *n* is divisible by 5, then you *may* give back exactly 42 jelly beans.

The goal of the game is end up with *exactly* 42 jelly beans.

For example, suppose that you start with 250 jelly beans. Then you *could* make the following moves:

- Start with 250 jelly beans
- Since 250 is divisible by 5, you *may* return 42 of the jelly beans, leaving you with 208.
- Since 208 is even, you *may* return half the jelly beans, leaving you with 104.
- Since 104 is even, you *may* return half the jelly beans, leaving you with 52.
- Since 52 is divisible by 4, you *may* multiply the last two digits (resulting in 10) and return those 10 jelly beans. This leaves you with 42.
- You have reached the goal!

The situation is more complicated than it looks. For example, suppose *n* is divisible by 60 (for example, n = 180). In this case, *n* is even, *n* is divisible by both 3 and 4, and *n* is divisible by 5. Therefore, *you may take one of three actions:* give back exactly *n*/2 jelly beans, give back *k* jelly beans (where *k* is the product of the last two digits of *n*), or give back exactly 42 jelly beans!

Your function decides whether it is possible to win the Jelly Bean Game if you start with *n* jelly beans. You do not have to figure out the exact sequence of decisions. Just determine if it is possible or not.

```
bool jelly_bean_game(int n);
```

For example, your function should return:

- `true` for n = 250
- `true` for n = 42
- `false` for n = 53
- `false` for n = 41

Submission

Write a `main` function to test/demonstrate each of your recursive functions. Name your solution **pa15.cpp**, and use the `turnin` command to submit your solution.