CS106B Chris Gregg
Spring 2017

Section Handout #3

This week has more practice with recursion, in particular exhaustive search and recursive backtracking. For any parameter that is passed by reference, that parameter must be the same when the function returns. You're also welcome to use helper functions for any of these problems.

1. Partitionable (CodeStepByStep)
Write a recursive function named `partitionable` that takes a reference to a Vector of integers and returns whether or not it is possible to divide the values into two groups such that each group has the same sum. For example, the vector `{1,1,2,3,5}` can be split into `{1,5}` and `{1,2,3}`, both of which sum to 6. However, the vector `{1,4,5,6}` can't be split into two vectors with the same sum.

2. Make Change (CodeStepByStep)
Write a recursive function called `makeChange` that takes in a target amount of change and a Vector of coin values and prints out every way of making that amount of change, using only the coin values in coins. For example, if you need to make change using only pennies, nickels, and dimes, the coins vector would be `{1, 5, 10}`. Each way of making change should be printed as the *number of each coin used* in the coins vector. For example, if you were to use the above coins vector to make change for 15 cents, the possibilities would be

```
{15, 0, 0}, {10, 1, 0}, {5, 2, 0}, {5, 0, 1}, {0, 3, 0}, {0, 1, 1}
```

In the outputs for the example, the first element of each vector indicates the number of pennies used, the second indicates the number of nickels, and the third indicates the number of dimes.

3. Print Squares (CodeStepByStep)
Write a recursive function named `printSquares` that uses backtracking to find all ways to express an integer as a sum of squares of unique positive integers. For example you can express the integer 200 as the following sums of squares:

```
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2
1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2
1^2 + 2^2 + 5^2 + 7^2 + 11^2
1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2
1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2
2^2 + 4^2 + 6^2 + 12^2
2^2 + 14^2
3^2 + 5^2 + 6^2 + 7^2 + 9^2
6^2 + 8^2 + 10^2
```

Some numbers can't be represented in this format; if this is the case, your function should produce no output. The sum has to be formed with **unique** integers (note that in a given sum of squares, no integers are repeated).

Thanks to Aaron Broder, Marty Stepp, Victoria Kirst, Jerry Cain, and other past CS106B and X instructors / TAs for contributing content on this handout.

4. Longest Common Subsequence (CodeStepByStep)
Recall what we learned last week about subsequences. A string is a subsequence of another if it contains the same letters in the same order, but not necessarily consecutively. We're going to build on that concept this week. Write a recursive function named `longestCommonSubsequence` that takes in two strings and returns the longest string that is a subsequence of both input strings. For example,

```
longestCommonSubsequence("leslie", "wesley")              "esle"

longestCommonSubsequence("chris", "anupama")              ""

longestCommonSubsequence("she sells", "seashells")        "sesells"
```

5. Ways to Climb (CodeStepByStep)
Imagine you're standing at the base of a staircase. A small stride will move up one stair, and a large stride advances two. You want to count the number of ways to climb the staircase based on different combinations of large and small strides. Write a recursive function `waysToClimb` that takes in a positive integer value representing the number of stairs and prints out each unique way to climb a staircase of that height. For example, `waysToClimb(4)` should produce the following output:

```
{1, 1, 1, 1}
{1, 1, 2}
{1, 2, 1}
{2, 1, 1}
{2, 2}
```

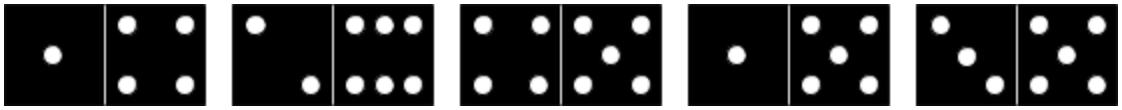
6. Twiddle (CodeStepByStep)
Write a recursive function named `listTwiddles` that accepts a string `str` and a reference to an English language `Lexicon` and uses exhaustive search and backtracking to print out all those English words that are `str`'s twiddles. Two English words are considered twiddles if the letters at each position are either the same, neighboring letters, or next-to-neighboring letters.

For instance, `"sparks"` and `"snarls"` are twiddles. Their second and second-to-last characters are different, but `'p'` is two past `'n'` in the alphabet, and `'k'` comes just before `'l'`. A more dramatic example: `"craggy"` and `"eschew"` are also twiddles. They have no letters in common, but `craggy`'s `'c'`, `'r'`, `'a'`, `'g'`, `'g'`, and `'y'` are -2, -1, -2, -1, 2, and 2 away from the `'e'`, `'s'`, `'c'`, `'h'`, `'e'`, and `'w'` in `"eschew"`. And just to be clear, `'a'` and `'z'` are not next to each other in the alphabet; there's no wrapping around at all. (Note: any word is considered to be a twiddle of itself, so it's okay to print `str` itself.)

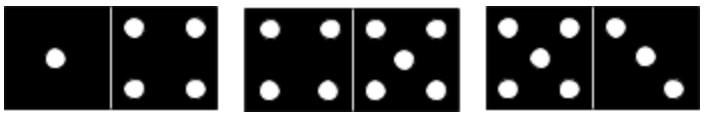
Constraints: Do not declare any global variables. You can use any data structures you like, and your code can contain loops, but the overall algorithm must be recursive and must use backtracking. You are allowed to define other "helper" functions if you like; they are subject to these same constraints. Do not modify the state of the Lexicon passed in.

7. Domino Chaining (CodeStepByStep)

The game of dominoes 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 a domino:



Dominoes are connected end-to-end to form chains, although two dominoes can only be connected if the numbers on the ends touching are the same. It's legal to rotate dominoes 180˚ so that the numbers are reversed. For example, one possible chain you could build with a subset of the above dominoes is:



Note that the domino on the right end had to be rotated.

Implement the recursive function `chainExists` that, given a list of dominoes, a start number, and an end number, returns whether or not it is possible to build a chain from the start number to the end number with some or all of the given dominoes. If two numbers are the same, then the chain already exists. Assume you are given the following struct:

```
struct domino {
  int first;
  int second;
};
```

For example, given that dominoes contained the set of dominoes above

```
chainExists(dominoes, 6, 2)                 true

chainExists(dominoes, 5, 5)                 true

chainExists(dominoes, 1, 6)                 false
```