Chris Gregg
CS106B

Section #3
April 23, 2018

Section Handout #3: Recursive Backtracking

Based on handouts by various current and past CS106B/X instructors and TAs.

1. Make Change

Write a recursive function called **`makeChange`** that takes in a target amount of change and a reference to a Vector **`coins`** of coin values (integers) 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

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

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.

2. Print Squares

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
```

Thus, a call to **`printSquares(200)`** should print out every combination of positive integers whose sum of squares equals the provided number:

```
{1, 2, 3, 4, 5, 8, 9}
{1, 2, 3, 4, 7, 11}
{1, 2, 5, 7, 11}
{1, 3, 4, 5, 6, 7, 8}
{1, 3, 4, 5, 7, 10}
{2, 4, 6, 12}
{2, 14}
{3, 5, 6, 7, 9}
{6, 8, 10}
```

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).

3. Longest Common Subsequence

Write a recursive function named **`longestCommonSubsequence`** that takes in two strings and returns the longest string that is a subsequence of both input strings. A string is a subsequence of another if it contains the same letters in the same order, but not necessarily consecutively. For example:

```
longestCommonSubsequence("leslie", "wesley")          "esle"
longestCommonSubsequence("chris", "bob")              ""
longestCommonSubsequence("she sells", "seashells")    "sesells"
```

Hint: in the recursive case, think about comparing just the first characters of each string. What recursion can you perform if they are the same? What recursion can you perform if they are different?

4. Ways to Climb

Imagine you're standing at the base of a staircase. A small stride will move up one stair, and a large stride will move up 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 non-negative integer value representing the number of stairs and prints out each unique way to climb a staircase of that height, taking strides of only 1 or two stairs at a time. For example, **`waysToClimb(4)`** should produce the following output:

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

5. Twiddle

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 ok 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.

6. Password Cracking

Write a function `findPassword` that takes in the maximum length a site allows for a user's password and tries to find the password into an account by using recursive backtracking to attempt all possible passwords up to that length (inclusive). Assume you have access to the function `bool login(string password)` that returns `true` if a password is correct. You can also assume that the passwords are entirely alphabetic and case-sensitive. You should return the correct password you find, or the empty string if you cannot find the password. You should return the empty string if the maximum length passed is 0 or throw an integer exception if the length is negative.

Security note: The ease with which computers can brute-force passwords is the reason why login systems usually permit only a certain number of login attempts at a time before timing out. It's also why long passwords that contain a variety of different characters are better! Try experimenting with how long it takes to crack longer and more complex passwords.

Tip: you can find more backtracking practice problems on `codestepbystep.com`, in the *backtracking* category under *C++*.