Week 4 Section Handout

This week’s section handout has practice with backtracking.

Remember our three steps to doing backtracking problems, where you need to explore multiple paths in each recursive call:

- **choose**, where you set up exploring a particular path;
- **explore**, where you recursively explore that path; and
- **un-choose**, where you undo whatever you did in the first step.

For any parameter that is passed by reference, that parameter must be the same when the function returns. You are also allowed to use helper functions for any of these problems. Remember another technique that you can use with additional parameters: keep track of information through your recursive calls.

---

1. **diceRolls**

Write a recursive function named `diceRolls` accepts an integer representing a number of 6-sided dice to roll, and output all possible combinations of values that could appear on the dice. For example, the call of `diceRolls(3)` should print:

```
{1, 1, 1}
{1, 1, 2}
{1, 1, 3}
{1, 1, 4}
{1, 1, 5}
{1, 1, 6}
{1, 2, 1}
...
{6, 6, 6}
```

If the number of digits passed is 0 or negative, print no output.

---

2. **PrintSquares**

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, the call of `printSquares(200)` should produce:

```
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 cannot be represented as a sum of squares, in which case your function should produce no output. The sum has to be formed with unique integers. Assume there already exists a function named `display` that accepts any collection of integers (such as a vector, set, stack, queue, etc.) and prints the collection's elements in the above format. (Or if you prefer, you can write such a function yourself.)

---

Thanks to CS106B and X instructors and TAs for contributing problems on this handout.
3. ListTwiddles
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.)

4. largestSum
Write a recursive function named `largestSum` that accepts a reference to a vector of integers `V` and an integer limit `N` as parameters and uses backtracking to find the largest sum that can be generated by adding elements of `V` that does not exceed `N`. For example, if you are given the vector `{7, 30, 8, 22, 6, 1, 14}` and the limit of 19, the largest sum that can be generated that does not exceed is 16, achieved by adding 7, 8, and 1. If the vector is empty, or if the limit is not a positive integer, or all of `V`'s values exceed the limit, return 0. Assume that all values in the vector are non-negative.

Each index's element in the vector can be added to the sum only once, but the same number value might occur more than once in the vector, in which case each occurrence might be added to the sum. For example, if the vector is `{6, 2, 1}` you may use up to one 6 in the sum, but if the vector is `{6, 2, 6, 1}` you may use up to two sixes.

The vector passed to your function must be back to its original state at the end of the call.

5. canBalance
You have a bag of weights and a balance. On one side of the balance is a target weight. Write a function named `canBalance` that accepts two parameters: an integer representing the target weight that you want to balance, and a vector of positive integers, which represent your weights. Your function should return true if you can balance the scales, and false otherwise.

For example, as shown in the figures below, if you had a bag of `{1, 3}` you could measure a weight of 4 or 2 by the following two arrangements:

You are allowed to modify the vector passed in as the parameter as you compute the answer, as long as you restore it to its original form by the time the overall call is finished.

Thanks to CS106B and X instructors and TAs for contributing problems on this handout.