Week 4 Section

This week's section handout has practice with Recursion, Recursive Backtracking, and big Oh. You can practice all the problems on CodeStepByStep: https://codestepbystep.com/problemset/view?id=14.

---

1. Big Oh

Give a tight bound of the nearest runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of the variable *N*.

```
// a)                                                    answer:
int sum = 0;
for (int i = 1; i <= N + 2; i++) {
   sum++;
}
for (int j = 1; j <= N * 2; j++) {
   sum += 5;
}
cout << sum << endl;
```

```
// b)                                                    answer:
int sum = 0;
for (int i = 1; i <= N - 5; i++) {
   for (int j = 1; j <= N - 5; j += 2) {
       sum++;
   }
}
cout << sum << endl;
```

```
// c)                                                    answer:
int sum = N;
for (int i = 0; i < 1000000; i++) {
   for (int j = 1; j <= i; j++) {
       sum += N;
   }
   for (int j = 1; j <= i; j++) {
       sum += N;
   }
   for (int j = 1; j <= i; j++) {
       sum += N;
   }
}
cout << sum << endl;
```

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

---

2. 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.

---

3. diceSum

Write a recursive function named **`diceSum`** that accepts two parameters: an int representing a number of 6-sided dice to roll, and a desired sum, and output all possible combinations of values that could appear on the dice that would add up to exactly the given sum. For example, the call of `diceSum(3, 5);` should print all combinations of rolls of 3 dice that add up to 5:

```
{1, 1, 3}
{1, 2, 2}
{1, 3, 1}
{2, 1, 2}
{2, 2, 1}
{3, 1, 1}
```

If the number of digits passed is 0 or negative, or if the given number of dice cannot add up to exactly the given sum, print no output. Your function must use recursion, but you can use a single `for` loop if necessary.

Bonus: only print unique combinations of rolls, regardless of order.

---

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. longestCommonSubsequence

Write a recursive function named **`longestCommonSubsequence`** that returns the longest common subsequence of two strings. Recall that if a string is a subsequence of another, each of its letters occurs in the longer string in the same order, but not necessarily consecutively. For example, the following calls should return the following values:

Call	Return
`longestCommonSubsequence("marty", "megan")`	`"ma"`
`longestCommonSubsequence("hannah", "banana")`	`"anna"`
`longestCommonSubsequence("she sells", "seashells")`	`"sesells"`
`longestCommonSubsequence("janet", "cs106b")`	`""`

---

6. makeChange

You have an amount of change you need to make, and an unlimited amount of coins of various denominations. Write a recursive function named **`makeChange`** that accepts two parameters: an integer representing an amount of change, and a vector of integers representing the coins' values in cents, and calculates and prints every way of making that amount of change, using the coin values in the vector. Each way of making change should be printed as the number of each coin used in the coins vector. For example, if you need to make 15 cents using pennies, nickels, and dimes, you should print the vectors to the right:

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

Note that this result is by calling it with a coins vector of `{1, 5, 10}` representing pennies, nickels, and dimes. You may assume that the amount of change passed to your function is non-negative, but it could exceed 100.

---

7. 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.
