CS 106B Section 4 (Week 5)

This week takes recursion a bit further with recursive backtracking. Remember the 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..

Recommended problems: #3, #4
Extra practice problems: 9.10, 9.11 (from Textbook)

---

1. partitionable.

Write a function named **partitionable** that takes a vector of ints and returns true if it is possible to divide the ints 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}`. However, the vector `{1,4,5,6}` can't be split into two.

```
bool partitionable(Vector<int>& nums) { ...
```

---

2. makeChange.

Write a function named **makeChange** that calculates and prints every way of making that amount of change, using the coin values in `coins`. 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 these vectors: `{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}`. Hint: in this example, there were three "choices": first, we chose the number of pennies, then the number of nickels, and finally the number of dimes.

```
void makeChange(int target, Vector<int>& coins ) { ...
```

---

3. longestCommonSubsequence.

Write a function named **longestCommonSubsequence** that returns the longest common subsequence of both 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.) Hint: in the recursive case, compare the first character of each string. What one recursive call can you make if they are the same? What two recursive calls do you try if they are different?

longestCommonSubsequence("**ma**rs", "**m**eg**a**n")	"ma"
longestCommonSubsequence("**c**hri**s**", "**cs**106b")	"cs"
longestCommonSubsequence("**s**h**e** **sells**", "**se**a**s**h**ells**")	"sesells"

```
string longestCommonSubsequence(string s1, string s2) { ...
```

CS 106B Section 4 (Week 5)

4. Debugging.

The following function takes a vector of ints and a number. It returns the largest sum we can make using elements in the vector without exceeding the number. Here are some sample calls. If everything in the vector is larger than the number or the vector is empty, return 0. Each recursive call handles one element in the vector, and that element is either in the final sum or not in the final sum, so we recursively search each of those choices. However, there is one bug.

Vector	Target Number	Return
{}	1	0
{10, **2**, **5**, **1**}	9	8
{**1**, **2**, **3**, **4**}	20	10

```
int maxSum(Vector<int>& numbers, int limit) {
    if (limit <= 0 || numbers.isEmpty()) {
        return 0;
    } else {
        // choose: remove the first element from the vector
        int first = numbers[0];
        numbers.remove(0);

        // explore: try making the final sum with and without this element
        int maximum;
        if (first > limit) {
            maximum = maxSum(numbers, limit);
        } else {
            // try making the sum with the first element, so we subtract it
            // from our limit and add it back to the result
            int with = first + maxSum(numbers, limit - first);

            // also try making the sum without the first element
            int without = maxSum(numbers, limit);
            maximum = max(with, without);
        }
        return maximum;
    }
}
```

5. Big O.

What is the Big O complexity of the follow snippets of code?

a)

```
for (int i = 0; i < 500; i++) {
  cout << i << endl;
}
```

b)

```
int sum = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < 100; j++) {
    for (int k = 0; k < j*j; k++) {
      sum++;
    }
  }
}
```

c)

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n*2; j++) {
    cout << j << endl;
  }
}
```