Section Handout 3

Problem One: Splitting the Bill

You've gone out for coffees with a bunch of your friends and the waiter has just brought back the bill. How should you pay for it? One option would be to draw straws and have the loser pay for the whole thing. Another option would be to have everyone pay evenly. A third option would be to have everyone pay for just what they ordered. And then there are a ton of other options that we haven't even listed here!

Your task is to write a function

```
void listPossiblePayments(int total, const Set<string>& people);
```

that takes as input a total amount of money to pay (in dollars) and a set of all the people who ordered something, then lists off every possible way you could split the bill, assuming everyone pays a whole number of dollars. For example, if the bill was $4 and there were three people at the lunch (call them *A*, *B*, and *C*), your function might list off these options:

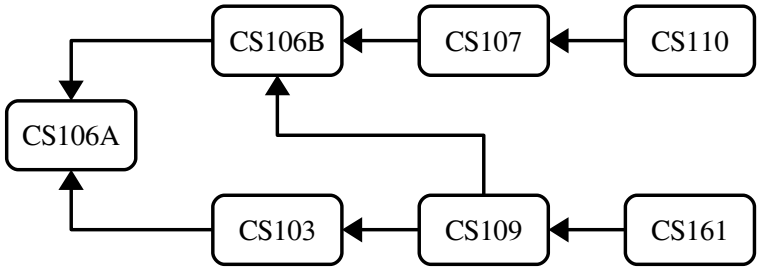
```
A: $4,    B: $0,    C: $0
A: $3,    B: $1,    C: $0
A: $3,    B: $0,    C: $1
A: $2,    B: $2,    C: $0
A: $2,    B: $1,    C: $1
A: $2,    B: $0,    C: $1
                …
A: $0,    B: $1,    C: $3
A: $0,    B: $0,    C: $4
```

Some notes on this problem:

- The total amount owed will always be nonnegative. If the total owed is negative, you should use the `error()` function to report an error.
- There is always at least one person in the set of people. If not, you should report an error.
- You can list off the possible payment options in any order that you'd like. Just don't list the same option twice.
- The output you produce should indicate which person pays which amount, but aside from that it doesn't have to exactly match the format listed above. Anything that correctly reports the payment amounts will get the job done.

Problem Two: Ordering Prerequisites

Imagine you have some collection of tasks that need to be done. Some of those tasks might depend on one another. For example, you might be navigating the CS Core, shown here:



Here, the arrows indicate prerequisites. CS106B has CS106A as a prerequisite, CS110 has CS107 as a prerequisite, CS109 has both CS106B and CS103 as prerequisites, and CS106A has no prerequisites. Assuming you can only take one CS class per quarter, what possible orderings are there for these classes that don't violate any prerequisites? Your task is to write a function

```
void listLegalOrderingsOf(const Map<string, Set<string>>& prereqs);
```

that takes as input a `Map` representing the prerequisite structure, then lists all possible orders in which you could complete those tasks without violating the prerequisites. The `prereqs` map is structured so that each key is a task and each value is the set of that task's immediate prerequisites. For example, the CS Core would be represented by the following map:

```
 "CS103" : { "CS106A" }
"CS106A" : { }
"CS106B" : { "CS106A" }
 "CS107" : { "CS106B" }
 "CS109" : { "CS103", "CS106B" }
 "CS110" : { "CS107" }
 "CS161" : { "CS109" }
```

Given this prerequisite structure, your function would then print out all of the following:

```
CS106A, CS106B, CS107, CS110, CS103, CS109, CS161
CS106A, CS103, CS106B, CS109, CS161, CS107, CS110
CS106A, CS106B, CS107, CS103, CS109, CS161, CS110
                (… many, many more …)
```

Some notes on this problem:

- Every task will be present in the `Map`. A task with no prerequisites will be represented as a key whose value is an empty `Lexicon`, as is the case for CS106A in the above example.
- Your function must not list off the same ordering twice.
- Your function must not work by simply generating all possible permutations of the tasks and then printing out just the ones that obey all the constraints. That would just be too inefficient.
- Your output doesn't have to exactly match our format. List off the orderings in whatever format you'd like. In case it helps, you can directly print a `Map`, `Set`, `Vector`, or `Lexicon` to `cout`.
- It's entirely possible that the set of tasks you're given can't actually be ordered. For example, if the task `"Learn Recursion"` depends on `"Learn Recursion"`, then no matter how you order things the prerequisites won't be satisfied. In that case, your function should just not print anything at all. Chances are, you won't need to do anything fancy to make this work. It'll just fall out naturally.

Problem Three: Change We Can Believe In

In the US, as is the case in most countries, the best way to give change for any total is to use a *greedy strategy* – find the highest-denomination coin that's less than the total amount, give one of those coins, and repeat. For example, to pay someone 97¢ in the US in cash, the best strategy would be to

- give a half dollar (50¢ given, 47¢ remain), then
- give a quarter (75¢ given, 22¢ remain), then
- give a dime (85¢ given, 12¢ remain), then
- give a dime (95¢ given, 2¢ remain), then
- give a penny (96¢ given, 1¢ remain), then
- give another penny (97¢ given, 0¢ remain).

This uses six total coins, and there's no way to use fewer coins to achieve the same total.

However, it's possible to come up with coin systems where this greedy strategy doesn't always use the fewest number of coins. For example, in the tiny country of Recursia, the residents have decided to use the denominations 1¢, 12¢, 14¢, and 63¢, for some strange reason. So suppose you need to give back 24¢ in change. The best way to do this would be to give back two 12¢ coins. However, with the greedy strategy of always picking the highest-denomination coin that's less than the total, you'd pick a 14¢ coin and ten 1¢ coins for a total of fifteen coins. That's pretty bad!

Your task is to write a function

```
int fewestCoinsFor(int cents, const Set<int>& coins)
```

that takes as input a number of cents and a `Set<int>` indicating the different denominations of coins used in a country, then returns the minimum number of coins required to make change for that total. In the case of US coins, this should always return the same number as the greedy approach, but in general it might return a lot fewer! Once you've written this function, discuss with the group whether memoization (described in the handout for Assignment 3) would be appropriate here. If so, go and add memoization to this function. If not, explain why not.

And here's a question to ponder: given a group of coins, how would you determine whether the greedy algorithm is always optimal for those coins?

You can assume that the set of coins always contains a 1¢ coin, so you never need to worry about the case where it's simply not possible to make change for some total. You can also assume that there are no coins worth exactly 0¢ or a negative number of cents, since that makes no sense. (No pun intended.) Finally, you can assume that the number of cents to make change for is nonnegative.

Problem Four: Member of the Wedding

You've been put in charge of planning a wedding! How exciting! You've gotten just about everything put together, except that you need to figure out who's supposed to sit where at dinner. You're going to have to be strategic with how you place people. The married couple, of course, needs to sit together, and after the Noodle Incident it's probably best to keep Uncle Calvin and Grandpa Hobbes at different tables. The twins Castor and Pollux probably ought to be grouped together, but your nephews Hatfield and McCoy probably ought to be kept apart. On top of all of this, each table can only hold so many people.

Your task is to write a function

```
Map<string, Vector<string>>
bestSeatingArrangementFor(const Vector<string>& guests,
                          const Vector<string>& tableNames,
                          int tableCapacity)
```

that takes as input a list of the wedding guests, a list of the names of the tables (imagine things like "table A," "table B", etc.), and the capacity of each table (assume all the tables are the same size), then returns a `Map<string, Vector<string>>` saying where each person should sit (the keys represent the names of the tables, and the values represent who's sitting at those tables) to maximize overall happiness. You can imagine you have access to a function

```
int scoreFor(const Map<string, Vector<string>>& arrangement)
```

that takes in a proposed seating arrangement and returns a numeric score indicating how good an arrangement it is, with higher numbers being better and lower numbers being worse. You can assume that `scoreFor` never returns a negative value; a score of 0 means that absolute chaos has ensued.

In the course of writing this function, you can assume that there's sufficient space to hold everyone at the wedding, so you don't need to worry about the case where you run out of tables.