Section Handout 3

Problem One: 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 two functions. The first is

```
int greedyChangeFor(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 how many coins would be needed to make change for that total using the greedy algorithm. For example, calling

```
greedyChangeFor(97, {1, 5, 10, 25, 50, 100})
```

would return 6 (these denominations are the denominations of US coins), while calling

```
greedyChangeFor(24, {1, 12, 14, 63})
```

would return 15 (these denominations are the ones those silly Recursians are using). You can write this function either iteratively or recursively, though we'd recommend doing it recursively just to get some practice. You can assume that there's always at least a 1¢ coin in the group, so you don't need to worry about the case where the total honestly can't be made with the given coins.

Next, 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.

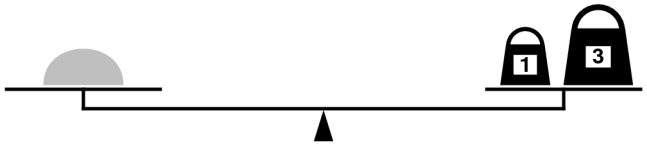
And here's a question to ponder: given a group of coins, how would you determine whether the greedy algorithm is always optimal for those coins?

Problem Two: Weights and Balances

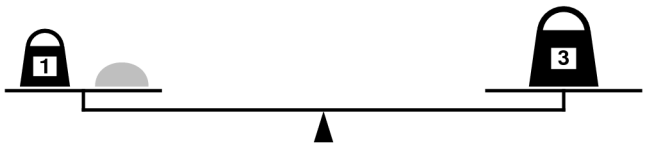
> I am the only child of parents who weighed, measured, and priced everything; for whom what could not be weighed, measured, and priced had no existence.
>
> —Charles Dickens, *Little Dorrit,* 1857

In Dickens's time, merchants measured many commodities using weights and a two-pan balance – a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:



It's more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:



Write a recursive function

```
bool isMeasurable(int target, const Vector<int>& weights)
```

that determines whether it is possible to measure out the desired target amount with a given set of weights, which is stored in the vector `weights`.

As an example, suppose that the variable `sampleWeights` has been initialized as follows:

```
Vector<int> sampleWeights = {1, 3};
```

Given these values, the function call

```
isMeasurable(2, sampleWeights)
```

should return **`true`** because it is possible to measure out two ounces using the sample weight set as illustrated in the preceding diagram. On the other hand, calling

```
isMeasurable(5, sampleWeights)
```

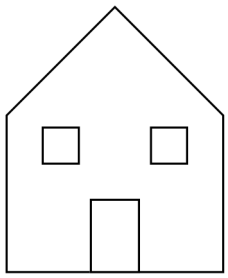
should return **`false`** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces.

Here's a function question to ponder: let's say that you get to choose *n* weights. Which ones would you pick to give yourself the best range of weights that you'd be capable of measuring?

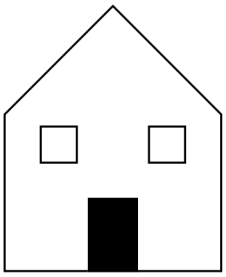
Problem Three: Filling a Region

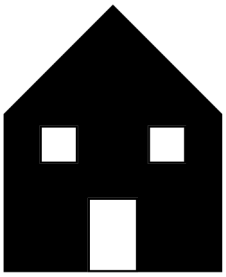
Most drawing programs make it possible to fill an enclosed region on the screen with a solid color. Typically, you invoke this operation by selecting a paint-bucket tool and then clicking the mouse, with the cursor somewhere in your drawing. When you do, the paint spreads to every part of the picture it can reach without going through a line.

For example, suppose you have just drawn the following picture of a house:

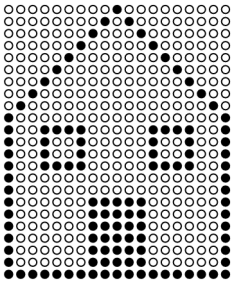


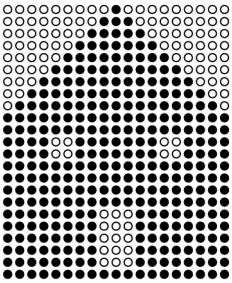
If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and doors, as shown on the right:





In order to understand how this process works, it is important to understand that the screen of the computer is actually broken down into an array of tiny dots called ***pixels***. On a monochrome display, pixels can be either white or black. The paint-fill operation consists of painting black the starting pixel (i.e., the pixel you click while using the paint-bucket tool) along with any pixels connected to that starting point by an unbroken chain of white pixels. Thus, the patterns of pixels on the screen representing the preceding two diagrams would look like this:





It is easy to represent a pixel grid using the type `Grid<bool>`. White pixels in the grid have the value `false`, and black pixels have the value `true`. Given this representation, write a function

```
void floodFillFrom(Grid<bool>& pixels, int row, int col)
```

that simulates the operation of the paintbucket tool by painting in black all white pixels reachable from the specified row and column without crossing an existing black pixel.

Just for fun – how might you rewrite this program iteratively rather than recursively? Which did you think was easier to write?

Problem Four: Weekend Hedonism

The weekend is coming up – how exciting! Being the industrious student that you are, you're hoping to cram in as much Fun and Excitement as you can. You've researched all the different events going on around campus and the larger Bay Area and have made a list of when each of those events is happening and how much total happiness it will bring you. The problem is that, try as you may, you can't seem to find a way to be in two places at the same time. Given a list of all the different things you can do, what should you do to maximize your Mirth and Whimsy?

Let's imagine we have this structure representing an event:

```
struct Event {
    string name;    // What the event is
    int happiness;  // How much happiness this will bring you. Because you can
                    //   totally quantify happiness.
    int startTime;  // When the event starts, measured in minutes since 5:00PM
                    //   on Friday.
    int endTime;    // When the event ends, measured in minutes since 5:00PM on
                    //   on Friday.
};
```

Your task is to write a function

```
Vector<Event> bestWeekendGiven(const Vector<Event>& events);
```

that takes in as input a list of all the possible events you can do over the weekend, then returns a `Vector<Event>` containing the events you should do over the weekend to maximize your total happiness.

Problem Five: 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.

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.