Nick Troccoli
CS 106X

Section #1
Week 2

Section Handout #1: Collections, File Reading, Big-O

Based on handouts by various current and past CS106B/X instructors and TAs.

1. Mirror *(Grid)*

Write a function **`mirror`** that accepts a reference to a grid of integers as a parameter and flips the grid along its diagonal, so that each index **`[i][j]`** contains what was previously at index **`[j][i]`** in the grid. You may assume the grid is square, that is, it has the same number of rows as columns. For example, the grid below on the left would be altered to give it the new grid state on the right:

```
{{ 6,  1,  9,  4},           {{6, -2, 14, 21},
 {-2,  5,  8, 12},            {1,  5, 39, 55},
 {14, 39, -6, 18},     -->    {9,  8, -6, 73},
 {21, 55, 73, -3}}            {4, 12, 18, -3}}
```

Bonus: How would you solve this problem if the grid were not square?

2. reorder *(Stack, Queue)*

Write a function named **`reorder`** that takes a queue of integers that are already sorted by absolute value, and modifies it so that the integers are sorted normally. Use a single `Stack<int>` to help you.

For example, passing in the Queue below on the left would alter it to the new queue state on the right:

```
{1, -2, 3, 4, -5, -6, 7}   -->   {-6, -5, -2, 1, 3, 4, 7}
```

3. Stack(s) as a Queue *(Stack, Queue)*

Your job is to implement a `Stack<int>` using one or more `Queue`s. That is, you will be responsible for writing the **`push`** and **`pop`** methods for a `Stack<int>` but your internal data representation must be a `Queue`:

```
void push(Queue<int>& queue, int entry)
int pop(Queue<int>& queue)
```

Bonus: Conceptually, how would you implement a `Queue` using only one or more `Stack`s?

4. stretch *(Vector)*

Write a function named **`stretch`** that accepts a reference to a `Vector<int>` as a parameter and modifies it to be twice as large, replacing every integer with a pair of integers, each half the original. If a number in the original Vector is odd, then the first number in the new pair should be one higher than the second so that the sum equals the original number. For example, passing in the Vector **`{18, 7, 4, 24, 11}`** should modify it to contain **`{9, 9, 4, 3, 2, 2, 12, 12, 6, 5}`**.

5. removeConsecutiveDuplicates *(Vector)*

Write a function named **`removeConsecutiveDuplicates`** that accepts a reference to a `Vector` of integers as a parameter and modifies it by removing any consecutive duplicates. For example, if a vector named `v` stores **`{1, 2, 2, 2, 3, 2, 2, 3}`**, the call of **`removeConsecutiveDuplicates(v)`** should modify it to store **`{1, 2, 3, 2, 3}`**.

6. Keith Numbers *(Vector)* – Problem by Keith Schwarz

Write a function named **`isKeithNumber`** that accepts an integer and returns true if that number is a "Keith number". A "*Keith number*" is defined as any *n*-digit integer that appears in the sequence that starts off with the number's *n* digits and then continues such that each subsequent number is the sum of the preceding *n*. (This is not unlike the classic Fibonacci sequence.) All one-digit numbers are trivially Keith numbers, but there are more interesting ones as well. For example, the number 7385 is also a Keith number, because the sequence that starts with `7, 3, 8, 5` ends up back at 7385 as follows:

`7, 3, 8, 5, 23, 39, 75, 142, 279, 535, 1031, 1987, 3832, `**`7385`**

The sequence starts out 7, 3, 8, 5, because those are the digits making up 7385. Each number after that is the sum of the four numbers that precede it (four, because 7385 has four digits). So the fifth number is the sum of 7+3+8+5, or 23. The sixth number is 3+8+5+23, or 39. And so on, until we eventually get back to 7385, which makes 7385 a Keith number. This means **`isKeithNumber(7385)`** should return **`true`**.

Your function should not loop infinitely; if you become sure that the number is *not* a Keith number, stop searching and immediately return false. (How can you know this?)

Bonus: consider modifying your code and writing a function called **`findKeithNumbers`** that accepts a minimum and maximum integer as parameters and prints to the console all Keith numbers in that range, along with the sequence of integers in the Keith number sequence for each integer. For example, the call of **`findKeithNumbers(1, 500);`** would print something like:

```
1: {1}
2: {2}
3: {3}
4: {4}
5: {5}
6: {6}
7: {7}
8: {8}
9: {9}
14: {1, 4, 5, 9, 14}
19: {1, 9, 10, 19}
28: {2, 8, 10, 18, 28}
47: {4, 7, 11, 18, 29, 47}
61: {6, 1, 7, 8, 15, 23, 38, 61}
75: {7, 5, 12, 17, 29, 46, 75}
197: {1, 9, 7, 17, 33, 57, 107, 197}
```

7. Big-O Analysis

Give a tight bound on the nearest runtime complexity for each of the following code fragments, using Big-O notation, in terms of the variable N. In other words. Find the growth rate of the code's runtime as N grows. **Hint:** a `Set`'s add operation is O(logN) time.

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

```
// d)
int sum = 0;
for (int i = 1; i <= 100000; i++) {
    for (int j = 1; j <= i; j++) {
        for (int k=1; k <= j; k++) {
            sum++;
        }
    }
}
cout << sum << endl;
```

```
// e)
Vector<int> list1;
for (int i = 0; i < N; i++) {
    list1.add(i);
}
Vector<int> list2;
Set<int> set;
for (int k : list1) {
    set.add(k*N);
    list2.add(k*N);
}
cout << “done!” << endl;
```

```
// f)
Vector<int> list;
for (int i = 0; i < N; i++) {
    list.add(N - i);
}
Set<double> set;
while (!list.isEmpty()) {
    set.add(list[0] * 137.0);
    list.remove(0);
}
cout << “done!” << endl;
```

Note: although Maps and Sets will not be covered until lecture on Fri. 10/5, problems 8-11 are here to help you get practice with them.

8. friendList *(Map)*

Write a function named **`friendList`** that takes in a file name and reads friend relationships from a file and writes them to a `Map`. You should return the populated Map. Friendships are bi-directional; if Nick is friends with Zach, Zach is friends with Nick. The file contains one friend relationship per line. The names are separated by a single space. You do not have to worry about malformed entries.

If an input file named buddies.txt looked like this:

```
Nick Marty
Zach Nick
```

Then the call of **`friendList("buddies.txt")`** should return a resulting map that looks like this:

`{"Marty":{"Nick"}, "Nick":{"Marty", "Zach"}, "Zach":{"Nick"}}`

9. twice *(Set)*

Write a function named **`twice`** that takes a Vector of integers by reference and returns a `Set<int>` containing all the numbers in the Vector that appear exactly twice.

Example: passing `{1, 3, 1, 4, 3, 7, -2, 0, 7, -2, -2, 1}` returns {3, 7}.

Bonus: do the same thing using only `Set`s as auxiliary storage.

10. numInCommon *(Set)*

Write a function **`numInCommon`** that takes two `Vector<int>` parameters by reference and returns the number of unique integers that appear in both lists. You may use one or more `Set`s to help you solve this problem.

For example, given the Vectors

`v1 = {3, 7, 3, -1, 2, 3, 7, 2, 15, 15}`

`v2 = {-5, 15, 2, -1, 7, 15, 36}`

calling **`numInCommon(v1, v2)`** should return 4, because the elements -1, 2, 7, and 15 occur in both lists.

11. unionSets *(Set)*

Write a function named unionSets that takes a `Set` of `sets` of `ints`, and returns the union of all of the sets of ints. (A union is the combination of everything in each Set.)

For example, if a Set variable named `sets` stores `{{1, 3}, {2, 3, 4, 5}, {3, 5, 6, 7}}`, then the call of **`unionSets(sets)`** should return `{1, 2, 3, 4, 5, 6, 7}`.