Section Handout #2 Solutions

If you have any questions about the solutions to the problems in this handout, feel free to reach out to your section leader, Aaron, or Chris for more information.

1. Reorder

```cpp
void reorder(Queue<int> &q) {
    Stack<int> s;
    int size = q.size();
    for (int i = 0; i < size; i++) { // separate positive and negative numbers
        int n = q.dequeue();
        if (n < 0) {
            s.push(n);
        } else {
            q.enqueue(n);
        }
    }

    size = q.size();               // enqueue negative numbers in reverse order
    while (!s.isEmpty()) {
        q.enqueue(s.pop());
    }

    for (int i = 0; i < size; i++) { // move positive numbers to end
        q.enqueue(q.dequeue());
    }
}
```

2. Stacks or Queues?

// solution 1: with an extra queue
```cpp
void push(Queue<int> &q, int entry) {
    Queue<int> q2;
    q2.enqueue(entry);
    while (!q.isEmpty()) {
        q2.enqueue(q.dequeue());
    }
    while (!q2.isEmpty()) {
        q.enqueue(q2.dequeue());
    }
}
int pop(Queue<int> &q) {
    return q.dequeue();
}
```

// solution 2: without auxiliary structures
```cpp
void push(Queue<int> &q, int entry) {
    q.enqueue(entry);
    for (int i = 0; i < q.size() - 1; i++) {
        q.enqueue(q.dequeue());
    }
}
int pop(Queue<int> &q) {
    return q.dequeue();
}
```
3. Twice

// solution 1: with additional structures
Set<int> twice(Vector<int> &v) {
    Map<int, int> counts;
    for (int n : v) {
        counts[n]++;
    }
    Set<int> twice;
    for (int k : counts) {
        if (counts[k] == 2) {
            twice += k;
        }
    }
    return twice;
}

// solution 2: without auxiliary structures
Set<int> twice(Vector<int> &v) {
    Set<int> one, two, more;
    for (int n : v) {
        if (one.contains(n)) {
            one.remove(n);
            two.add(n);
        } else if (two.contains(n)) {
            two.remove(n);
            more.add(n);
        } else if (!more.contains(n)) {
            one.add(n);
        }
    }
    return two;
}

4. Friend List

Map<string, Vector<string>> friendList(string &filename) {
    ifstream infile(filename.c_str());
    Map<string, Vector<string>> friends;
    string s1, s2;
    while(infile >> s1 >> s2) {
        friends[s1] += s2;
        friends[s2] += s1;
    }
    return friends;
}

5. Tracing a Mystery

<table>
<thead>
<tr>
<th>Call</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>mystery1(4, 1)</td>
<td>4</td>
</tr>
<tr>
<td>mystery1(8, 2)</td>
<td>16, 8, 16</td>
</tr>
<tr>
<td>mystery1(3, 4)</td>
<td>12, 9, 6, 3, 6, 9, 12</td>
</tr>
</tbody>
</table>

6. Sum of Squares

int sumOfSquares(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n * n + sumOfSquares(n - 1);
    }
}
7. Reverse

```cpp
string reverse(string &s) {
    if (s == "") {
        return "";
    } else {
        return reverse(s.substr(1)) + s[0];
    }
}
```

8. Star String

There are multiple ways to handle invalid input. In our solution below, we chose to throw an exception. However, you can also have a well-defined value that you return in the event of invalid input. It's important that whatever you choose, you document it well.

```cpp
string starString(int n) {
    if (n < 0) {
        throw "Invalid input.";
    } else if (n == 0) {
        return "*";
    } else {
        string stars = starString(n - 1);
        return stars + stars;
    }
}
```

Note that another possible solution is to return the result `starString(n - 1) + starString(n - 1)`. Why might you not want to use this solution?

9. Stutter Stack

```cpp
void stutterStack(Stack<int> &s) {
    if (!s.isEmpty()) {
        int next = s.pop();
        stutterStack(s);
        s.push(next);
        s.push(next);
    }
}
```

10. Subsequence

```cpp
bool isSubsequence(string &big, string &small) {
    if (small == "") {
        return true;
    } else if (big == "") {
        return false;
    } else {
        if (big[0] == small[0]) {
            return isSubsequence(big.substr(1), small.substr(1));
        } else {
            return isSubsequence(big.substr(1), small);
        }
    }
```
11. Tower of Hanoi
Solving the Tower of Hanoi problem with \( n \) disks is composed of three steps:

1. Move all the disks but the largest (or \( n-1 \) disks) from the source peg to a spare peg in order to expose the largest disk.
2. Move the largest disk to the destination peg
3. Move all the disks but the largest (or \( n-1 \) disks) from the spare peg to the destination peg

You might notice that the first and third steps can be expressed as their own Tower of Hanoi problem, which means we can implement them using recursive calls!

```cpp
void hanoi(int disks, int source, int dest) {
    if (disks > 0) {
        int thirdPeg = 6 - source - dest; // the three peg indices sum to 6 (1 + 2 + 3) which means you can always find the third peg by subtracting the source and dest from 6
        hanoi(disks - 1, source, thirdPeg);
        cout << "move disk " << disks << " from peg " << source " to peg " << dest << endl;
        hanoi(disks - 1, thirdPeg, dest);
    }
}
```

12. Edit Distance

```cpp
int editDistance(string &s1, string &s2) {
    if (s1 == "") {
        return s2.length();
    } else if (s2 == "") {
        return s1.length();
    }

    // try three possibilities for the "zeroth" character:
    int add = 1 + editDistance(s1, s2.substr(1));
    int del = 1 + editDistance(s1.substr(1), s2);
    int sub = editDistance(s1.substr(1), s2.substr(1));
    if (s1[0] != s2[0]) {
        sub += 1;
    }
    return min(add, min(del, sub));
}
```