Nick Troccoli
CS 106X

Section #2 Solutions

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

1. Recursion code trace *(recursion, tracing)*

Call	Output
`mystery1(4, 1)`	`4`
`mystery1(8, 2)`	`16, 8, 16`
`mystery1(3, 4)`	`12, 9, 6, 3, 6, 9, 12`

2. Debugging recursion *(recursion, debugging)*

The middle index is repeated in both sub-ranges, so when the recursion gets down to a range of length 2, the recursive call doesn't actually get any smaller, so the function recurses infinitely. Fix: replace the second recursive call with `int rightMax = recursiveMax(v, middle + 1, right).`

3. cannonballs *(recursion)*

```
int cannonballs(int height) {
    if (height < 0) {
        throw height;  //can also use the Stanford library error function to add a message
    } else if (height == 0) {
        return 0;
    } else {
        return height * height + cannonballs(height – 1);
    }
}
```

4. reverseString *(recursion, strings)*

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

Bonus: A Stack. Push all characters from s onto a Stack and pop them off sequentially into a new string. This mimics the function call stack generated by our recursive code above.

5. doubleStack *(recursion, Stack)*

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

6. combinations *(recursion)*

```
// non-memoized solution
int combinations(int n, int k) {
    if (n <= 0 || k < 0 || k > n) {
        return 0;
    } else if (k == 0 || k == n) {
        return 1;
    } else {
        return combinations(n-1, k-1)
             + combinations(n-1, k);
    }
}
```

```
// bonus: memoized cool solution
int combinationsHelper(int n, int k,
HashMap<int, HashMap<int, int>>& cache) {
    if (n <= 0 || k < 0 || k > n) {
        return 0;
    } else if (k == 0 || k == n) {
        return 1;
    } else if (cache.containsKey(n)
            && cache[n].containsKey(k)) {
        return cache[n][k];
    } else {
        int result =
           combinationsHelper(n-1, k-1, cache)
         + combinationsHelper(n-1, k, cache);
        cache[n][k] = result;
        return result;
    }
}
int combinations(int n, int k) {
    HashMap<int, HashMap<int, int>> cache;
    return combinationsHelper(n, k, cache);
}
```

7. isSubsequence *(recursion)*

```
//logic: compare letters until you find every letter in small or exhaust the letters in big
bool isSubsequence(const string& big, const 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);
        }
    }
}
```

8. reverseMap *(Map, extra practice)*

```
Map<string, int> reverseMap(const Map<int, string>& map) {
    Map<string, int> rev;
    for (int key : map) {
        rev[map[key]] = key;
    }
    return rev;
}
```