Practice CS106B Midterm III Solutions

Here's one possible set of solutions for the practice midterm questions. Before reading over these solutions, please, please, please work through the problems under semi-realistic conditions (spend three hours, have a notes sheet, etc.) so that you can get a better sense of what taking a coding exam is like.

This solutions set contains some possible solutions to each of the problems in the practice exam. It's not meant to serve as "the" set of solutions to the problems – there are lots of ways you can go about solving each problem here. In other words, if your solution doesn't match ours, don't take that as a sign that you did something wrong. Feel free to stop by the CLaIR, to ask questions on Piazza, or to ask your section leader for their input!

The solutions we've included here are a lot more polished than what we'd expect most people to turn in on an exam. You'll three hours to complete this whole exam. We have a lot of time to write up these solutions, clean them up, and try to get them into a state where they'd be presentable as references. Don't worry if what you wrote isn't as clean as what we have here, but do try to see if there's anything we did here that would help you improve your own coding habits.

Problem One: Containers I (8 Points)

Here's one possible solution:

```
/* Given a set of adjectives, does it meet all the requirements? */
bool hasAllRequirements(const HashSet<string>& adjectives,
                        const Vector<string>& requirements) {
    for (string req: requirements) {
        if (!adjectives.contains(req)) {
            return false;
        }
    }
    return true;
}

HashSet<string> petsMatching(const HashMap<string, HashSet<string>>& adjectiveMap,
                             const Vector<string>& requirements) {
    HashSet<string> result;
    for (string pet: adjectiveMap) {
        if (hasAllRequirements(adjectiveMap[pet], requirements)) {
            result += pet;
        }
    }
    return result;
}
```

Why we asked this question: This question was designed to get you playing around with different container and collection types (here, `Vector`, `HashMap`, and `HashSet`) and to work with nested containers. This problem is also *significantly* easier to solve if you write a helper function – the logic to handle everything when you don't have a separate helper to check if all the requirements are met is tricky!

Problem Two: Containers II (8 Points)

Here's one possible solution, though we saw many others:

```
bool differsByOneLetter(const string& longer, const string& shorter) {
    for (int i = 0; i < longer.size(); i++) {
        if (longer.substr(0, i) + longer.substr(i + 1) == shorter) {
            return true;
        }
    }
    return false;
}

bool isShrinkingSequence(const string& word,
                         const Lexicon& words,
                         Stack<string> path) {
    /* Base Case 1: If the stack is empty, this can't be a shrinking sequence. */
    if (path.isEmpty()) {
        return false;
    }

    /* Base Case 2: If the word parameter isn't a word, the path can't possibly
     * be a shrinking path.
     */
    if (!words.contains(word)) {
        return false;
    }

    /* Get the top of the stack and confirm that it matches the word. */
    string top = path.pop();
    if (top != word) {
        return false;
    }

    /* If the word is just one character, the stack should be empty. */
    if (word.length() == 1) {
        return path.isEmpty();
    }

    /* If the stack is empty, then this can't be a shrinking path because we
     * would have returned in the previous step.
     */
    if (path.isEmpty()) {
        return false;
    }

    /* Confirm that the top word differs from the current word by exactly one
     * letter.  If not, this isn't a shrinking sequence.
     */
    if (!differsByOneLetter(word, path.peek()) {
        return false;
    }

    /* This is a shrinking sequence iff the remainder of the stack is a shrinking
     * sequence for the top of the stack.
     */
    return isShrinkingSequence(path.peek(), words, path);
}
```

Why we asked this question: This question was designed to get you engaging with a different set of containers (here, `Stack` and `Lexicon`) and to give you a chance to show off your problem-solving skills. There are a number of little details to check for, and we figured this would be a good way to let you demonstrate what you've learned about problem-solving in C++.

Common mistakes: This problem is a lot trickier than it initially appears and there are many different cases to check for. Common mistakes included not checking that the stack wasn't empty before peeking or popping it; forgetting to check that the words in the stack were English words; counting the empty string, which isn't a word, as a shrinkable word; incorrectly checking whether adjacent words in the stack differed by exactly one letter; or forgetting to check that the stack ends on a string of length one.

A subtle but common error in this problem was writing a loop like this one:

```
for (int i = 0; i < path.size(); i++) { // <-- Careful! Doesn't work!
    string top = path.pop();
    /* … */
}
```

This code appears to iterate over the contents of the stack, but has a slight error in it. Specifically, note that on each iteration, the value of i increases and the value of `path.size()` decreases, so the total number of iterations is roughly half what it should be.

Problem Three: Recursion I (8 Points)

There are many ways to solve this problem, but the one we think is the most elegant is shown here.

```
string removeDoubledLetters(const string& str) {
    /* Base case: Any string with one or fewer letters has no duplicates. */
    if (str.length() <= 1) {
        return str;
    }
    /* Recursive case: Either the first letter is doubled or it isn't. If it's
     * doubled, then the string we want is formed by dropping the first letter
     * and then deduplicating the rest of the string.
     */
    else if (str[0] == str[1]) {
        return removeDoubledLetters(str.substr(1));
    }
    /* If that letter isn't doubled, include it in the result and deduplicate the
     * remaining characters.
     */
    else {
        return str[0] + removeDoubledLetters(str.substr(1));
    }
}
```

Why we asked this question: This question is designed to get you thinking about the insights and problem-solving strategies that come up when thinking recursively. This isn't a branching recursion problem, and there's no need to pass down extra state through the chain. Really, it's about seeing how the larger version of the problem reduces down to smaller versions.

Problem Four: Recursion II (8 Points)

The solution shown here follows the hint from the problem statement. The insight needed here is that if you have a string of n balanced parentheses, then either

- n is zero, in which case your string is empty, or
- n is nonzero. Look at the first open parenthesis and find its matching close parenthesis. What's between them must be a pair of balanced parenthesis, and let's suppose there are k pairs in that inner part. That means there are $n - k - 1$ pairs of parentheses that appear after the close parenthesis we identified.

With that in mind, we just need to assemble all the strings formed this way. Here's one way to do this:

```
HashSet<string> balancedStringsOfLength(int n) {
    /* Base Case: There is exactly one balanced string with zero parentheses. */
    if (n == 0) {
        return { "" };
    }
    /* Recursive Case: Some number of parentheses go on the inside of the first
     * pair, and some go afterward. Try all combinations and see what we get.
     */
    else {
        HashSet<string> result;

        for (int k = 0; k < n; k++) {
            for (string inner: balancedStringsOfLength(k)) {
                for (string outer: balancedStringsOfLength(n - k - 1)) {
                    result += "(" + inner + ")" + outer;
                }
            }
        }
        return result;
    }
}
```

This initial version works, but it's fairly slow because it recomputes `balancedStringsOfLength` multiple redundant times. A better approach is to compute each of the recursive calls exactly once and then aggregate things together at the end. Here's an alternative version of the recursive step:

```
    else {
        /* Produce all balanced strings of shorter lengths. */
        Vector<HashSet<string>> balanced;
        for (int i = 0; i < n; i++) {
            balanced += balancedStringsOfLength(i);
        }

        /* Glue things together. */
        HashSet<string> result;
        for (int k = 0; k < n; k++) {
            for (string inner: balanced[k]) {
                for (string outer: balanced[n - k - 1]) {
                    result += "(" + inner + ")" + outer;
                }
            }
        }
        return result;
    }
```

Why we asked this question: This question was designed to get you playing around with a novel recursive insight and how that translates into code. The key insight you need here involves looking at strings of balanced parentheses as having a particular pattern, and from there the question is how to take that insight and use it to write a recursive solution.

The strategy used here is a bit different than what we've done so far. Rather than passing down a single solution through the recursion chain, we have our function instead return up to us all solutions to the simpler problem, then combine them together in different ways to assemble all of the overall solutions.