

Solutions to Midterm Exam

Problem 1: Code Reading (20 points)

- a) State of v: {10, 22, 17}, State of s: {}, return value: 27
- b) State of v: {3, 9, 5}, State of s: {}, return value: 8

Problem 2: Code Reading (20 points)

- a) $O(N^3)$
- b) $O(N)$
- c) $O(N \log N)$
- d) $O(1)$

Problem 3: Code Writing (20 points)

There are many different possible solutions to this problem – here is one of them.

```
/** This function reads in the provided, valid orders file and returns
 * a map from costume name to the number of unique customers who
 * bought that costume. **/
Map<string, int> uniqueCostumeCustomers(ifstream &file) {
    // A map from costume name to a set of customers who purchased it
    Map<string, Set<string>> costumeCustomers;

    string line;
    while (getline(file, line)) {
        // Parse the name
        int nameEnd = line.find("\\"", 1);
        string name = line.substr(1, nameEnd - 1);

        // Parse each costume one at a time
        size_t nextCostumeStart = line.find("\\"", nameEnd + 1);
        while (nextCostumeStart != string::npos) {
            int length = line.find("\\"", nextCostumeStart + 1) -
                         nextCostumeStart - 1;
            string costume = line.substr(nextCostumeStart + 1, length);
            costumeCustomers[costume] += name;
            nextCostumeStart = line.find("\\"", nextCostumeStart + length + 2);
        }
    }

    // We just need the counts of customers, not their names
    Map<string, int> costumeCustomerCounts;
    for (string key : costumeCustomers) {
        costumeCustomerCounts[key] = costumeCustomers[key].size();
    }

    return costumeCustomerCounts;
}
```

Problem 4: Code Writing (20 points)

There are many different possible solutions to this problem – here are two of them.

Solution 1:

```
/** This function recursively combines the provided sorted stacks
 * (where the lowest element is at the top) into one sorted stack
 * (where the lowest element is also at the top) and returns it.
 * This solution does not use any auxiliary data structures. */
Stack<int> combineSortedStacks(Stack<int> &s1, Stack<int> &s2) {
    if (s1.isEmpty() && s2.isEmpty()) {
        return s1;
    } else if (s1.isEmpty()) {
        return s2;
    } else if (s2.isEmpty()) {
        return s1;
    } else if (s2.peek() <= s1.peek()) {
        int element = s2.pop();
        Stack<int> newStack = combineSortedStacks(s1, s2);
        newStack.push(element);
        return newStack;
    } else {
        int element = s1.pop();
        Stack<int> newStack = combineSortedStacks(s1, s2);
        newStack.push(element);
        return newStack;
    }
}
```

Solution 2:

```
/** This function recursively combines the provided sorted stacks
 * (where the lowest element is at the top) into one sorted stack
 * (where the lowest element is also at the top) and returns it.
 * This solution uses an auxiliary data structure and
 * a wrapper function */
Stack<int> combineSortedStacks2(Stack<int>& s1, Stack<int>& s2) {
    Stack<int> returnStack;
    combineSortedStacksHelper(s1, s2, returnStack);
    return returnStack;
}

/** This function is a recursive helper function to combine
 * the provided sorted stacks. It builds the combined version
 * into the result parameter, which is passed by reference. */
void combineSortedStacksHelper(Stack<int>& s1, Stack<int>& s2,
                               Stack<int>& result) {
    if (s1.isEmpty() && s2.isEmpty()) {
        return;
    } else if (s1.isEmpty() ||
               (!s1.isEmpty() && !s2.isEmpty() && s2.peek() <= s1.peek())) {
        int element = s2.pop();
        combineSortedStacksHelper(s1, s2, result);
        result.push(element);
    } else if (s2.isEmpty() ||
               (!s1.isEmpty() && !s2.isEmpty() && s1.peek() < s2.peek())) {
```

```

        int element = s1.pop();
        combineSortedStacksHelper(s1, s2, result);
        result.push(element);
    }
}

```

Problem 5: Code Writing (20 points)

There are many different possible solutions to this problem – here is one of them.

```

/** This function attempts to find the best section assignments for
 * the provided students, where best means that we minimize
 * overall unhappiness. Unhappiness is defined as +i for each
 * time someone gets their i+1-th choice. For example, getting
 * your 1st choice means 0 unhappiness, 2nd choice +1
 * unhappiness, and so on. A student will only be placed in a section
 * they ranked. If no placement is possible, this function will
 * return an empty assignments map and INT_MAX unhappiness. **/
SectionAssignments assignSections(HashMap<string, Vector<int> >&
                                 preferences, Vector<int>& sectionSizes) {
    if (preferences.isEmpty()) {
        return {{}, 0};
    }

    // Choose a student to place
    string currStudent = preferences.front();
    Vector<int> currStudentPrefs = preferences[currStudent];
    preferences.remove(currStudent);

    // Try each preference and return the one with lowest unhappiness
    SectionAssignments bestPlacement = {{}, INT_MAX};
    for (int prefNum=0; prefNum < currStudentPrefs.size(); prefNum++) {
        int ithPreference = currStudentPrefs[prefNum];
        if (sectionSizes[ithPreference] > 0) {
            // choose
            sectionSizes[ithPreference]--;
            // explore
            SectionAssignments placement =
                assignSections(preferences, sectionSizes);
            if (placement.overallUnhappiness != INT_MAX) {
                placement.overallUnhappiness += prefNum;
                if (placement.overallUnhappiness <
                    bestPlacement.overallUnhappiness) {
                    placement.assignments[currStudent] = ithPreference;
                    bestPlacement = placement;
                }
            }
            // unchoose
            sectionSizes[ithPreference]++;
        }
    }

    // we are no longer placing this student
    preferences[currStudent] = currStudentPrefs;

```

```
    return bestPlacement;
}
```

Problem 6: Code Writing (20 points)

There are many different possible solutions to this problem – here is one of them.

```
/** This function modifies the provided queue in place to reverse
 * the order of all pairs in the first k elements. All other
 * elements remain unchanged and unmoved. If k is odd,
 * then the kth element is left in place, while all pairs before it
 * are swapped. If k is larger than the queue size, this function
 * throws an error.
 */
void swapK(Queue<int>& q, int k) {
    if (k > q.size()) error("k cannot be more than queue size");

    int kEven = k;
    if (k % 2 != 0) kEven -=1;

    for (int i = 0; i < kEven / 2; i++) {
        int elem1 = q.dequeue();
        int elem2 = q.dequeue();
        q.enqueue(elem2);
        q.enqueue(elem1);
    }

    for (int i = 0; i < q.size() - kEven; i++) {
        q.enqueue(q.dequeue());
    }
}
```