Midterm Exam: Solutions

Your name: CS106B Rockstar

Section leader: I don't have one

Honor Code: I hereby agree to follow both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this exam, nor will I give any. The answers I am submitting are my own work. I agree not to talk about the exam contents to anyone until a solution key is posted by the instructor.

Signature: ***CS106B Rockstar*** **← YOU MUST SIGN HERE!**

Rules: (same as posted previously to class web site)

- This exam is to be completed by each student individually, with no assistance from other students.
- You have 2 hours (120 minutes) to complete this exam.
- This test is **closed-book, closed-notes**. You may only have one 8.5x11″ double-sided sheet of notes.
- You may not use any computing devices, including calculators, cell phones, iPads, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- On code-writing problems, you do not need to write a complete program, prototypes, nor #include statements. Write only the code (function, etc.) specified in the problem statement.
- Unless otherwise specified, you can write helper functions to implement the required behavior. When asked to write a function, do not declare any global variables.
- Do not abbreviate code, such as writing ditto ("") or dot-dot-dot marks (...). Pseudo-code will be given no credit.
- If you wrote your answer on a back page or attached paper, please label this clearly to the grader.
- Do NOT staple or otherwise insert new pages into the exam. Changing the order of pages in the exam confuses our automatic scanning system.
- **You should write your name at the top of every page in the exam.** You will get 1 point for this.
- **Tear off the last page (reference sheet)** before submission. You will get 1 point for this.
- Follow the Stanford Honor Code on this exam and correct/report anyone who does not do so.

#	Description	Earned	Max
0	Names and removing last page		2
1	Big-Oh		6
2	ADT Trace		6
3	Recursion Trace		7
4	ADTs		15
5	Backtracking		14
	Total		**50**

1. Big-Oh

(6 pts) Give a tight bound of the nearest runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of variable N. (In other words, the algorithm's runtime growth rate as N grows.)

a)

```
int i = 3;
while (i < 500) {
    for (int j = 0; j < N * 4; j++) {
        cout << "i: " << "j: N" << endl;
        i += 8;
    }
}
```

Answer: O (**N**)

b)

```
Set<int> set;
for (int i = 0; i < N * 2; i++) {
    for (int j = 0; j < N; j++) {
        set.add(j);
    }
}
```

Answer: O ($\mathbf{N^2} \log \mathbf{N}$)

c)

```
Vector<int> vec;
for (int i = 0; i < 300; i++) {
    for (int j = N - 2; j <= N; j++) {
        vec.insert(0, 31);
    }
}
```

Answer: O (**1**)

2. ADT Trace

(6 pts) Consider the following code, written using Stanford library ADT implementations:

```
void collectionMystery(Stack<int> &s) {
    Queue<int> q;
    for (int i = 0; i < s.size(); i++) {
        q.enqueue(1000 + s.pop());
    }
    for (int i = 0; i < q.size(); i++) {
        s.push(100 + q.peek());
    }
    while (!s.isEmpty()) {
        q.enqueue(-1 * s.pop());
        q.enqueue(q.dequeue());
    }
    cout << q << endl;
}
```

Now write the output of the above code, given the following inputs. In the inputs shown, the leftmost entry is least recently added item, and the rightmost entry is the most recently added item. Your output should adhere to the same convention.

Hint: Try considering what each of the three loops does independently before attempting to trace the function as a whole.

a) `s = {8, 2, 4}`

{**1083, -14, -1183**}

b) `s = {14, 83, 1, 16, 7, 9}`

{**1004, -1104, 1002, -8, -1104**}

3. Recursion Trace

(7 pts) Write the output of the following function, given the inputs below:

```
void recursionMystery(string str, int n) {
    if (str.length() < 3) {
        cout << str << n;
    } else {
        recursionMystery(str.substr(0, str.length() / 3), n + 1);
        recursionMystery(str.substr(str.length() / 3), n + 1);
    }
}
```

a) `recursionMystery("a", 2);`

a2

b) `recursionMystery("mysteries", 0);`

m2ys2te2r3i4es4

4. ADTs

In online file systems such as Google Drive or Dropbox, users can not only store their files on the cloud, but they can also share files with their friends, coworkers, or teammates. Users have can share both files and directories (directories are the folders on your computer that store files and other folders). For the rest of this problem, we'll refer to the sample directory CS106B described below (the directories have files inside of them, indented for clarity):

```
CS106B
|---Lecture Notes
    |---Lecture 1-Introduction to CPP
        |---Lecture1.ppt
        |---Lecture1Notes.txt
    |---Lecture 2-Strings and File IO
        |---Lecture2.ppt
        |---Lecture2Notes.txt
|---Assignments
    |---Game Of Life
        |---life.pro
|---Section Notes
    |---Section1.txt
    |---Section2.txt
    |---Section3.txt
    |---Section4.txt
```

a)

(6 pts) You will write a **non-recursive** function that reads an ifstream open to a file listing these sharing permissions. Your function should return a Map from strings of file- and directory-names to a `Set<string>` representing users who have explicitly been given permission for those files. The file is structured as follows:

```
ashley:Section Notes,life.pro,Lecture1.ppt
shreya:Lecture1.ppt,Assignments
nolan:Section Notes,Lecture 1-Introduction to CPP,Lecture2Notes.txt
```

This first line of this file indicates that the directory `Section Notes` as well as the files `life.pro` and `Lecture1.ppt` have been shared with the user *ashley*. File and directory names will always be separated by commas, and the users name will always be followed by a colon.

The resulting `Map` should have the following key-value pairs:

```
Section Notes:                  {ashley, nolan}
life.pro:                       {ashley}
Lecture1.ppt:                   {ashley, shreya}
Assignments:                    {shreya}
Lecture 1-Introduction to CPP: {nolan}
Lecture2Notes.txt:              {nolan}
```

Note that Nolan has access to `Lecture1-Intro to CPP` but is not listed as having access to `Lecture1.ppt` because the file did not explicitly state that he had access to `Lecture1.ppt` (we'll fix this problem in part b!).

You will write the following function:

```
    Map<string, Set<string>> readFilePermissionsLog(ifstream &input);
```

For full credit, you solution should adhere to the following constraints:

- You should choose an efficient solution. Choose data structures intelligently and use them properly. While your code is not required to have any particular Big-Oh, you may lose points if your code is extremely inefficient.
- You may read the file only once. Do not re-open it or rewind the stream.
- You may not define any custom classes or `structs`; you must use the Stanford collections to solve this problem.
- You may not use any STL collections.

You can make the following assumptions:

- The file is well-formatted (as described above), and the `ifstream` is open at the beginning of the file.
- No file or directory names have colons or commas.
- A comma always separates two file/directory names.
- Each user is listed only once in the file, and users names are unique.
- You do not need to include any function prototypes or `#include` statements.
- No two files or directories have the same name, even if they are in different directories.

Write your solution on the following pages.

4. ADTs - Writing Space

a)

```
Map<string, Set<string>> readFilePermissionsLog(ifstream &input) {
    Map<string, Set<string>> result;
    string line;
    while (getline(input, line)) {
        Vector<string> parts = stringSplit(line, ":");
        Vector<string> files = stringSplit(parts[1], ",");
        for (string file : files) {
            result[file] += parts[0];
        }
    }
    return result;
}
```

4. ADTs

b)

(9 pts) You may have noticed that users can share directories, instead of just files. When sharing directories, the specified user would get access to all the files and subdirectories in that directory. One glaring flaw of our map from part a is that it does not handle applying permissions from a directory to its files and subdirectories. Let's take a concrete example: if you were to share `Section1.txt` with a friend, they would have access to that file, but not `Section2.txt`, `Section3.txt`, or `Section4.txt` (which is reflected in our Map above). If you were to share the directory `Lecture 1-Introduction to CPP` with your friend, however, they should get access to all of the following files: `Lecture1.ppt`, `Lecture1Notes.txt`, `Lecture2.ppt`, and `Lecture2Notes.txt`.

In the example permissions from part a, user *shreya* also has permission to view `life.pro`, but theres no obvious way to get that information from the `Map` in part a. You will write a function that, given an `ifstream` to the same file as part a, returns a `Map` listing permissions for files only. Every file should be included in your map, even those that are not shared with any users.

Specifically, for the CS106B directory and the file logging permissions from part a, the map to return should have the following key-value pairs:

```
Lecture1.ppt: {ashley, nolan, shreya}
Lecture1Notes.txt: {nolan}
Lecture2.ppt: {ashley, shreya}
Lecture2Notes.txt: {nolan}
life.pro: {ashley, shreya}
Section1.txt: {ashley, nolan}
Section2.txt: {ashley, nolan}
Section3.txt: {ashley, nolan}
Section4.txt: {ashley, nolan}
Secret.txt: {}
```

Using your function from part a, you will write the following function, which will solve the problem **recursively**:

```
Map<string, Set<string>> getFilePermissions(ifstream &input, const string
                                             &directory);
```

Even if you did not get full credit in part a, we will assume your `readFilePermissionsLog` function works as described for this part of the problem. **Our solution includes a recursive helper function.**

To receive full credit for this part, your solution should adhere to the following constraints:

- You should choose an efficient solution. Choose data structures intelligently and use them properly. While your code is not required to have any particular Big-Oh, you may lose points if your code is extremely inefficient.

- You may read the ifstream only once. Do not re-open it or rewind the stream.
- You may not define any custom classes or structs; you must use the Stanford collections to solve this problem.
- You may not use any STL collections.

You can make the following assumptions:

- **The function from part a is written correctly and available for your use for this part of the question**.
- All user names are unique, and no two files or directories share a name, even if they're in different directories.
- The ifstream adheres to the same assumptions from part a.
- You have access to all the functions in `filelib.h`. For your convenience, some of the functions you may find useful are `listDirectory`, which lists the tails of all the files and subdirectories in a directory (you can get the full filename by taking the directory name, appending a slash / and then appending the tail); `isDirectory`, which returns true if the given name is the name of a directory; and `getTail`, which strips a full filename – such as `Users/ataylor4/exam.pdf` – to its last name – such as `exam.pdf`. These functions are described in further detail in the reference sheet. As a helpful hint, the Map from part a is file **tails** to a set of permissions for those files and directories. Your Map that you return should similarly be from file **tails** to the complete permissions for those files (but should not have any directories).
- You do not need to include any function prototypes or `#include` statements.

Write your solution on the following pages.

4. ADTs - Writing Space

```
Map<string, Set<string>> getFilePermissions(ifstream &input,
                                            const string &directory) {
    Map<string, Set<string>> result;
    Map<string, Set<string>> basicFilePermissions = readFilePermissionsLog(input);
    Set<string> currPermissions;
    getFilePermissionsHelper(basicFilePermissions, result, currPermissions,
                                directory);
    return result;
}

void getFilePermissions(const Map<string, Set<string>> &basicFilePermissions,
                        Map<string, Set<string>> & result,
                        Set<string> &currPermissions, const string &name) {
    if (isFile(name)) {
        result[getTail(name)] = basicFilePermissions[getTail(name)]
                                    + currPermissions;
    } else {
        Set<string> updatedPermissions = currPermissions
                                                + basicFilePermissions[getTail(name)];
        for (string file : listDirectory(name)) {
            getFilePermissions(basicFilePermissions, result,
                                        updatedPermissions, name + "/" + file);
        }
    }
}
```

5. Backtracking

(14 points) You are the premiere event organizer for the land of Recursia and have been chosen to determine the seating plan for the Queens coronation. Unfortunately for you, the citizens of Recursia are quick to develop feuds. The Queen-to-be has requested that you avoid seating anyone next to (meaning directly in front of, behind, or to the side of) an individual they are feuding with. Your job is to successfully seat all the guests in the auditorium, represented as a Grid of strings, so that no fights will break out.

To aid you in your feat, you are given a Map of individuals to a `Set` of of all their enemies, represented as a `Map<string, Set<string>>`. Under no circumstances can anyone be seated next to their enemy (you may assume that in Recursia, enemyships, like friendships, are mutual, meaning if Voldemort is enemies with Harry, Harry is enemies with Voldemort). You are also given a guest list, represented as a `Vector<string>`, as well as the dimensions of the auditorium.

As an example, assume we are seating the guests Chidi, Eleanor, Tahani, and Jason in a 2x2 auditorium. Chidi and Jason don't have any enemies, but Eleanor and Tahani are enemies. The Map would be as follows:

```
Chidi:   {},
Eleanor: {Tahani},
Tahani:  {Eleanor},
Jason:   {}
```

One acceptable seating chart (there are multiple) would be:

Eleanor	Chidi
Jason	Tahani

Note that in this situation, the two enemies, Eleanor and Tahani, are not next to each other; for the purposes of this problem, diagonals do not count. If Eleanor also becomes enemies with Jason, there are no acceptable seating charts, and your function should return a `Grid` of empty strings.

You should use **recursive backtracking** to solve this problem. As a hint, you will need to try a lot of possibilities before you find the correct one - the people of Recursia are a finicky bunch, after all. You should write code to implement the following function:

```
Grid<string> assignSeats(Map<string, Set<string>> &enemies, Vector<string>
                         &guests, int numRows, int numCols);
```

To receive full credit, your solution should adhere to the following constraints:

- You should choose an efficient solution. Choose data structures intelligently and use them properly. While your code is not required to have any particular Big-Oh, you may lose points if your code is extremely inefficient.

- Every individual is included in the enemies `Map` even if they don't have any enemies.
- You may not define any custom classes or `structs`; you must use the Stanford collections to solve this problem.
- All parameters should be unchanged at the end of your function.

You can make the following assumptions:

- The number of guests is exactly equal to the number of seats in the auditorium.
- All citizens' names are unique.
- Enemyships are mutual, and all enemyships will be represented twice in the map (i.e. if Harry is in the Set of Voldemort's enemies, Voldemort will be in the set of Harry's enemies).
- You may use a recursive helper function if you'd like.
- You do not need to include any function prototypes or `#include` statements.

Write your solution on the next page.

5. Backtracking - Writing Space

```
Grid<string> assignSeats(Map<string, Set<string>> &enemies,
                         Vector<string> &guests, int numRows, int numCols) {
    Grid<string> seatingChart(numRows, numCols);
    assignSeatingHelper(besties, enemies, guests, seatingChart, 0, 0);
    return seatingChart;
}


bool checkNeighbors(Map<string, Set<string>> &enemies, Grid<string> &seatingChart,
                    int currRow, int currCol, const string &guest) {
    // check left neighbor
    if (seatingChart.inBounds(currRow - 1, currCol)) {
        if (enemies[seatingChart[currRow - 1][currCol]].contains(guests[i])) {
            return false;
        }
    }

    // check north neighbor
    if (seatingChart.inBounds(currRow, currCol - 1)) {
        if (enemies[seatingChart[currRow][currCol - 1]].contains(guests[i])) {
            return false;
        }
    }
    return true;
}

bool assignSeatsHelper(Map<string, Set<string>> &enemies, Vector<string> &guests,
                       Grid<string> &seatingChart, int currRow, int currCol) {
    if (guests.isEmpty()) {
        return true;
    }
    if (currCol == seatingChart.numCols()) {
        currCol = 0;
        currRow += 1;
    }
    for (int i = 0; i < guests.size(); i++) {
        string guest = guests[i];
if (checkNeighbors(enemies, seatingChart, currRow, currCol, guest)) {
            seatingChart[currRow][currCol] = guest;
            guests.remove(i);
            if (assignSeatsHelper(enemies, guests, seatingChart,
                                  currRow, currCol + 1) {
                guests.insert(i, guest);
                return true;
            }
            guests.insert(i, guest);
            seatingChart[currRow][currCol] = "";
        }
    }
    return false;
}
```