Practice Midterm Exam

Your name: ______________________________

Section leader: ______________________________

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: ______________________________ ← **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, 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.
- **Rip off the pages of library reference in the back of the exam and do not turn them in.**
- 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	ADTs, read		6
2	Big-Oh, read		6
3	Collections and addFiles, write		10
4	Recursion, read		8
5	Recursion, write		10
6	Backtracking, write		10

Name: ____

1. ADTs (read)

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

```
void collectionMystery(Queue& q) {
    Stack s1;
    Stack s2;
    while (!q.isEmpty()) {
        s1.push(q.dequeue());
        if (!q.isEmpty()) {
            s2.push(q.dequeue());
        }
    }
    while (!s1.isEmpty()) {
        s2.push(s1.pop());
    }
    while (!s2.isEmpty()) {
        q.enqueue(s2.pop());
    }
    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.

Input	**Output**
`q = {4, 5, 7, 6, 3, 2}`	
`q = {1, 3, 5, 7, 9, 11, 13}`	

Name: ____

2. Big-Oh (read)

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.)

Part a

```
int sum1 = 0;
for (int i = 1; i <= N; i++) {
    int k = 20 * 20 * 20;
    for (int j = 1; j <= k; j++) {
        sum1++;
    }
}
cout << sum1 << endl;
```

Answer: O(__________________________)

Part b

```
Vector<int> v2;
for (int i = 1; i <= N * 2; i++) {
    for (int j = 1; j <= N / 2; j++) {
        v2.insert(0, i);
    }
    v2.clear();
}
while (!v2.isEmpty()) {
    v2.remove(v2.size() - 1);
}
cout << "done!" << endl;
```

Answer: O(__________________________)

Part c

```
Map<int, int> map3;
for (int i = 1; i <= N + N; i++) {
    map3.put(i, i/2);
}

Set<int> set3;
for (int i = 1; i <= N; i++) {
    int value = map3.get(i);
    set3.add(value);
    map3.remove(i);
}
cout << "done!" << endl;
```

Answer: O(__________________________)

Name: ______________________________________

3. Collections and File IO

```
Ernie: hey bert!
Ernie: did you eat yet?
Bert: no, let's grab lunch!
Ernie: yay :)
---
Ernie: good morning!
Big Bird: Hello, there!
Ernie: have a great day
---
Ernie: bert, yt?
Ernie: guess not!
---
Big Bird: I'm hungry.
Ernie: He he he!
Big Bird: Do not mock my pain
---
Bert: you see that cute dog?
Ernie: yeah I do! So cute
Ernie: I want a puppy now
---
C Monster: Who wants cookies?
Bert: Me!
C Monster: OK let's get some!
---
C Monster: Anybody else?
Ernie: Me!
C Monster: OK no problem!
---
Bert: Any cookies left?
Ernie: Nope. he he he!
Bert: :-(
---
Big Bird: hey ernie!
Ernie: studying 106B, ttyl
```

The input file contains a collection of chat **messages**, one per line, in the format of the example shown at right. Each chat message begins with the sender's name, followed by a colon and a space, followed by the sender's message.

Chat messages are organized into **conversations**, which are groups of chat messages separated by a line of 3 dashes. For example, the file on the right contains 9 separate conversations. You may assume that a conversation involves at most **two people**.

Your function should process the input file and print out the names of the **pair(s)** of people who participated in the largest number of conversations together. If multiple pairs of people **tie** for the largest number of conversations, **print all such pairs**. It does not matter how many individual messages the people exchange, only how many conversations they both participate in together.

For example, if the file chatlogs.txt contains the text at right, the call chatBuddies(”chatlogs.txt”); should print the following output:

```
Bert and Ernie
Big Bird and Ernie
```

The preceding is the correct output to print because both of those pairs participate in 3 conversations together. The pair of Bert and Ernie are in the first, fifth, and eighth conversations together; and the pair of Big Bird and Ernie are in the second, fourth, and ninth conversations together. The relative order in which you print the pairs, and the order of the two individual names within each pair, does not matter. So printing ”Ernie and Big Bird” or ”Ernie and Bert” or any other ordering is also acceptable.

If the file is missing or unreadable, your function should **throw a string exception**. If the file does exist, you may assume that it contains valid data; that is, it will contain at least one chat log between at least two people, and that the file will be in the format described above. Names are case sensitive, e.g. ”bert” is a different name than ”BERT” or ”Bert.”

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit

- 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 use as many **collections** as you like, but you should choose them intelligently and use them properly
- You may not define any custom classes or structs; you must use the Stanford collections to solve this problem.

Write your answer on the next page.

Name: ______________________________________

3. Collections and File IO (Writing Space)

Name: ______________________________________

3. Collections and File IO (Writing Space)

Name: ____

4. Recursion (read)

For each call to the recursive function below, write the output it would produce as it would appear on the console.

```
void recursionMystery(int x) {
    if (x < 10) {
        cout << "[" << x << "] ";
    } else {
        cout << x << " ( ";
        recursionMystery(x / 2);
        cout << " , ";
        recursionMystery(x % 2);
        cout << " ) ";
    }
}
```

a) `recursionMystery(17);`

b) `recursionMystery(31);`

c) `recursionMystery(62);`

Name: ______________________________________

5. Recursion (write)

Write a recursive function named `countDuplicates` that accepts a reference to a `Stack` of integers as its parameter and returns a count of the total number of duplicate elements in the stack. You may assume that the stack's contents are a **sorted** collection of non-negative integers, and therefore that all duplicate values will be stored consecutively in the stack. For example, given a stack named `myStack` containing the following elements (left corresponds to bottom, and right corresponds to top):

$\{1, 3, 4, 7, 7, 7, 7, 9, 9, 11, 13, 14, 14, 14, 16, 16, 18, 19, 19, 19\}$

In the above example, there are 9 total duplicate values: three 7s, a 9, two 14s, a 16, and two 19s. So the call of `countDuplicates(myStack);` should return 9.

Your function should not make any externally visible changes to the stack passed in. That is, you should either not modify the stack passed, or if you do modify it, you must restore it back to its exact original state before your function returns.

Constraints: For full credit, obey the following restrictions. A solution that disobeys them may receive partial credit.

- **Do not use any loops**; you must use recursion. If you do not use recursion, you will receive almost no points.
- **Do not use any auxiliary data structures** like Vector, Map, Set, Stack, Queue, array, string, etc.
- Do not solve this problem using **"string hacks"** related to the `toString` result of a Stack, such as by calling `toString()` and then searching for commas or other patterns.
- Do not declare any global variables.
- Your code must run with a complexity of no worse than `O(N)` to receive full credit.
- You can declare as many primitive variables like `ints` as you like.
- You are allowed to define other **"helper"** functions if you like; they are subject to these same constraints.

Write your answer on the next page.

Name: ____

5. Recursion (write) - Writing Space

Name: ____________________________________

5. Recursion (write) - Writing Space

Name: ____________________________________

6. Backtracking (write)

Write a recursive function named `knightsTour` that uses backtracking to try to find a **"Knight's tour"** path on a chess board of a given size. A Knight's tour is a path on an empty chess board traveled by a knight piece that touches each square on the board exactly once. A knight chess piece moves in an "L" pattern where its row/col change by 2/1 or 1/2 in either direction, as shown in the figure at right.

You will be passed two **parameters**: a reference to a `Grid` of `ints` representing the chess board, and a starting location row/column. Your code should explore whether a knight's tour can be made starting at that location. If you find such a tour, **print** the board to show it. If not, do not print any output.

The grid passed to your function will be square (it will have the same number of rows as columns) and will initially be filled with **0s**. You should try to fill it with integers from 1 to N inclusive representing the order in which the knight should visit the squares of the board, starting from the indicated location, where N is the number of rows times columns.

To help you solve this problem, we provide you several pieces of **starter code**. Since grids often refer to individual squares with a row and column, we provide a structure type to represent a single row/column location on the board. We also provide you with a function to print a board, and a function to generate all neighboring moves from a given location. These can help simplify your solution to this problem.

```
// starter code (assume that all of this is provided)
struct Location {                         // represents one square on the board
    int row;
    int col;
};
void printBoard(Grid<int>& board);        // print a board in the format shown below
Vector<Location> getMoves(Location loc);  // all moves a knight can make from this locatio
```

Recall that you can declare and utilize a `struct` with syntax such as shown below.

```
// example of using the Location struct
Location loc1;                                // row 0, column 0
Location loc2 {3, 1};                         // row 3, column 1
cout << loc2.row << "," << loc2.col << endl;  // 3,1
```

The `getMoves` function returns all board squares that are 2/1 rows/columns away from the location you pass it. It does not exclude squares that are out of bounds on the board. For example, if your current row/col location is {1,1}, then `getMoves` returns a vector containing {{-1,0}, {0,-1}, {0,2}, {2,-1}, {2,3}, {3,0}, {3,1}, {3,2}}.

For example, if we call your function as shown below at left on a 5x5 board starting from (2, 2), one acceptable result would be to output the following knight's tour along with modifying the grid to store these integer values in its 25 squares. So, the code

Name: ______________________________________

```
Grid<int> board(5, 5);
Location start {2, 2};
knightsTour(board, start);
```

gives us:

```
21   2   7  12  23
 8  13  22  17   6
 3  20   1  24  11
14   9  18   5  16
19   4  15  10  25
```

If there are multiple valid knight's tours of the given board, you may print any of them. Regardless of which tour you print, your code **must stop** after finding/print a single solution; you should not find or print all possible solutions. Your function must return when it is complete; it is not allowed to stop your function by calling exit or other such trickery.

Efficiency: While your code is not required to have any particular Big-Oh, you may lose points if your code is extremely inefficient or repeatedly re-explores the same path multiple times. Pass data structures by reference to avoid making copies.

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- Do not declare any global variables.
- Your code can contain loops if needed to solve the problem, but to receive full credit, your overall algorithm must be recursive and must use backtracking techniques to generate the results.
- You are allowed to define other "helper" functions if you like; they are subject to these same constraints.

Write your answer on the next page.

Name: ____

6. Backtracking (write) - Writing Space

Name: ______________________________________

6. Backtracking (write) - Writing Space