CS106X Instructor: Cynthia Lee

Autumn 2015 Practice Exam

PRACTICE MIDTERM EXAM

NAME (LAST, FIRST): _____

SUNET ID:_____ @stanford.edu

Problem	1	2	3	4	5	TOTAL
Topic	ADTs	Pointers, Memory	Classes	Recursion	Big-O	
Score						
Possible	30	21	30	30	9	120

Instructions:

- The time for this exam is **2 hours**. There are 120 points, or 1 point per minute. This gives you some general idea of how to pace yourself on each problem.
- Use of anything other than a pencil, eraser, pen, one 8.5x11 page (one side) of notes, and the official textbook is prohibited. In particular, **no** computers or digital devices of any kind are permitted. Blank scratch paper may be provided by proctors and does not need to be turned in.
- PLEASE rip off the pages of library reference in the back of the exam and do not turn them in.
- Please do NOT staple or otherwise insert new pages into the exam. Changing the number of pages in the exam confuses our automatic scanning system. Thanks.
- SCPD and OAE: Please call or text 760-845-7489 if you have a question.

Please sign *before* you begin:

I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

_____ _____ _____

(Signature) (Date) (Start time - HH:MM zone)

---

1. **ADTs (30pts).** For this problem, you will write a function that tries to fit a puzzle piece into a puzzle by exhaustively trying all placements, including rotating the piece 90, 180 and 270 degrees.

   `bool canFit(Grid<bool>& puzzle, Grid<bool> piece);`

   - `puzzle`: an arbitrary size Grid where places that are already filled in by a piece are set to `true`, and places that are empty and available for a piece to be placed are set to `false`.
   - `piece`: an arbitrary size Grid where the shape of the piece is given by the Grid entries that are set to `true`, and any negative space around or in the piece is indicated by `false`. You may assume that the Grid for a piece is no larger than necessary to contain the shape.
   - If a placement of the piece is found, `puzzle` should be edited accordingly to add the piece when your function returns. (If more than one possible placement exists, only edit `puzzle` to put it in one place.)
   - You may want to add a helper function.

Example 1:

Puzzle:

T	F	**T**	**T**	F
T	F	**T**	F	**T**
T	F	F	**T**	**T**

Piece:

T	F
T	F
T	**T**

canFit() returns true

- A piece may be rotated 90, 180, or 270 degrees in order to make it fit (see example 2), but not "flipped" (see example 3).To rotate a piece clockwise by 90 degrees, use the following formula: value at (row,col) goes to (col, numRows()-row). Note that 180-degree rotation is the same as two successive 90-degree rotations, and 270 is three rotations.

Example 2:

Puzzle:

T	**T**	**T**	**T**	F
T	**T**	**T**	F	**T**
T	F	F	F	**T**

Piece:

T	F
T	F
T	**T**

canFit() returns true (Piece rotates clockwise by 270 degrees to fit.)

Example 3:

Puzzle:

T	F	**T**	**T**	F
T	F	**T**	F	**T**
F	F	**T**	**T**	**T**

Piece:

T	F
T	F
T	**T**

canFit() returns false (Piece cannot fit even with rotation.)

```
bool canFit(Grid<bool>& puzzle, Grid<bool>& piece) {

}
```

2. **Pointers and Memory (21pts).** Draw the state of memory at each of the **3 marked places** in the code. Be careful in showing where your pointers originate and terminate (outer box vs inner box). If you make an error, partial credit may be assigned on the basis of assuming previous drawings were correct. **Leave uninitialized areas blank**, and **clearly mark NULL**. Draw the components in the appropriate stack and heap areas marked for you. **On the last diagram, add a label "ORPHANED"** identifying specific pieces of memory that are now orphaned, if any.

```
struct Floo {
   int powder;
   Floo *wand[2];

};
```

```
Floo network[3];
network[2].powder = 4;
network[2].wand[0] = new Floo;
network[2].wand[0]->powder = 7;
network[0].wand[0] = network[0].wand[1] = new Floo;
Floo** burrow = &(network[0].wand[0]);
                        //Draw the state of memory now (#1)
burrow[0] = NULL;
burrow++;
Floo* botts = &(network[2]);
*burrow = botts->wand[0];
                        //Draw the state of memory now (#2)
(*burrow)->wand[1] = *burrow;
network[1] = **burrow;
botts->wand[1] = botts;
                        //Draw the state of memory now (#3)
```

DRAWING #1:

Stack:

Heap:

(Problem 7) DRAWING #2:

Stack:

Heap:

(Problem 7) DRAWING #3 (remember to label pieces of memory that are leaked, if any):

Stack:

Heap:

3. **Classes (30pts).** In this problem, you'll be implementing a Farm class. A Farm object keeps track of information about the number and types of animals on the farm. The functions in the Farm class include adding and removing animals, and tracking feed consumption.
   You are responsible for implementing the public interface given in the header file below which includes the constructor, destructor and public functions of the class. Each function has a comment which explains what it is responsible for doing as well as any special cases you need to handle. Read these function comments carefully **before** beginning to code up any part of the class. We highly suggest that you plan out how you are going to implement these functions and how they will interact with your private fields of the class before you begin writing any code on this problem.

   You should write the implementation of these public functions as if you were writing the cpp file of the class, so make sure that you use the appropriate syntax (think back to the PQueue assignment). You should put any private class fields that you would like to add into the private section of the header file below.

   Please be sure to double-check that you implement all 8 functions, as well as the relevant private data in the farm.h file.

```
//beginning of farm.h file
class Farm {
public:
   /* Create a farm with the given capacity which is the maximum number of
    * animals that the farm can support. Initially the farm has no animals and a
    * value of 0. */
   Farm(int capacity);

   /* Delete any memory that this class is responsible for */
   ~Farm();

   /* Get a list of all animal types for which there is an animal on the farm.
    * There should be no duplicates in this list. If the farm is empty, this
    * function should return an empty Vector. */
   Vector<string> getAnimalTypes();

   /* Return the total number of animals currently on the farm. */
   int getTotalNumberOfAnimals();

   /* Add an animal of the given type and cost to the farm (note that animals of
    * the same type may have different costs). If there is no room for this
    * animal (total number of animals equals capacity) then throw an error. */
   void addAnimal(string type, string cost);

   /* Return the total cost of all animals currently on the farm. */
   int getTotalCost();

//farm.h file continues on next page
```

```
//farm.h, continued

      /* Remove one animal of this type, as well as the cost it was added with. If
       * there are multiple animals of this type pick any one. If there are no
       * animals on the farm of this type, then throw an error.
       */
      void removeAnimal(string type);


      /* Return the number of animals on the farm of the given type. If none of the
       * specified animal is on the farm, return a count of 0. */
      int getNumberOfAnimalType(string type);

   private:
      // TODO: add any private fields or methods that you need for your
      // implementation









   };
//end of farm.h file
```

```
//beginning of farm.cpp file

#include "farm.h"

//TODO: Add implementations of your methods here
```

//farm.cpp file continued on next page

```
//farm.cpp file, continued
```

```
//end of farm.cpp
```

4. **Recursion (30pts). Recursive Backtracking (30pts).** For this problem, we will revisit the Boggle game from Assignment 4. Recall that the cubes in the game Boggle are marked on each of their six sides with a letter. The 16 cubes from the original game were represented in your program as 16 strings of length six, as follows:

```
const string STANDARD_CUBES[16] = {
 "AAEEGN", "ABBJOO", "ACHOPS", "AFFKPS",
 "AOOTTW", "CIMOTU", "DEILRX", "DELRVY",
 "DISTTY", "EEGHNW", "EEINSU", "EHRTVW",
 "EIOSST", "ELRTTY", "HIMNQU", "HLNNRZ"
};
```

Imagine you wish to leave a note for me on my desk. You can't find any paper or pencils in my office, but I do have a few boxes of the game Boggle, and you decide to write out the note in Boggle cubes.

Note that not all strings of characters can be generated using only Boggle cubes. Once a given cube is used for a letter, it cannot be used again for that letter nor any of the other letters on the other five sides of the cube. You can create spaces between words (as well as newlines, tabs, etc) by how you arrange the cubes on the desk, but no punctuation or digits.

Write a function `static bool canWriteInCubes(string str, int n)` that uses recursive backtracking to check whether a given string can be generated from boggle cubes (if yes, it returns `true`, otherwise returns `false`).

- `str`: the input string. For simplicity, **you may assume** the input string `str` consists only of uppercase alphabet characters and white space. To skip over whitespace (because it can be represented by spacing the cubes on the desk), use the `isspace()` function (from `<cctype>`). It takes a single `char` and returns `true` if the `char` is a whitespace character (space, newline, tab, etc), otherwise `false`.
- `n`: the number of Boggle game sets. Your function should generalize to accommodate variable numbers of Boggle games—in other words, `n` complete sets of the game's 16 cubes are available. Return `false` if `n` is negative or zero.
- Suggestion: solve it for `n=1` and insert code to generalize if you have time.
- You may use the required function with signature given above as a wrapper for an actual recursive version that takes more parameters.
- Use the `const STANDARD_CUBES` above in your code (assume it is already defined).

```
static bool canWriteInCubes(string str, unsigned int n) {




}
```

5. **Big-O (9pts).** Give a tight bound of the nearest runtime complexity class (worst-case) for each of the following code fragments in Big-O notation, in terms of variable N. As a reminder, when doing Big-O analysis, we write a simple expression that gives only a power of N, such as $O(N^2)$ or O(log N), *not* an exact calculation. Write your answer in the blanks on the right side.

Question (3pts each)	Answer

`int myfunction(int N) {`
`  Grid<int> matrix(10 * N, 10 * N);`
`  for (int i = 0; i < matrix.numRows(); i++) {`
`  for (int j = 0; j < matrix.numCols(); j++) {`
`  matrix[i][j] = 0;`
`  }`
`  }`
`}`
	O( )

`// For this problem, treat N as N = vec.size()`
`for (int i = 0; i < vec.size(); i++) {`
`  int minPosition = i;`
`  for (int j = i; j < vec.size(); j++) {`
`  if (vec[j] < vec[minPosition]) {`
`  minPosition = j`
`  }`
`  }`
`  temp = vector[i];`
`  vector[i] = vector[minPosition];`
`  vector[minPosition] = temp;`
`}`
	O( )

`// For this problem, treat N as N = vec.size(), assume first`
`// call uses start = 0, end = vec.size() – 1.`
`void recursiveFun(Vector<int> &vec, int start, int end) {`
`  if (start > end) return;`
`  if (start == end) {`
`  cout << vec[start];`
`  return;`
`  }`
`  recursiveFun(vec, start, (start + end) / 2);`
`  recursiveFun(vec, ((start + end) / 2) + 1, end);`
`}`
	O( )

Summary of Relevant Data Types

We tried to include the most relevant member functions and operators for the exam, but not all are listed. You are free to use ones not listed here that you know exist. ***You do <u>not</u> need to do #include for these.***

```
class string {
 bool empty() const;
 int size() const;
 int find(char ch) const;
 int find(char ch, int start) const;
 string substr(int start) const;
 string substr(int start, int length) const;
 char& operator[](int index);
 const char& operator[](int index) const;
};

class Vector {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem); // operator+= used similarly
 void insert(int pos, const Type& elem);
 void remove(int pos);
 Type& operator[](int pos);
};

class Grid {
 int numRows() const;
 int numCols() const;
 bool inBounds(int row, int col) const;
 Type get(int row, int col) const; // cascade of operator[] also works
 void set(int row, int col, const Type& elem);
};

class Stack {
 bool isEmpty() const;
 void push(const Type& elem);
 Type pop();
};

class Queue {
 bool isEmpty() const;
 void enqueue(const Type& elem);
 Type dequeue();
};
```

```
class Map {
 bool isEmpty() const;
 int size() const;
 void put(const Key& key, const Value& value);
 bool containsKey(const Key& key) const;
 Value get(const Key& key) const;
 Value& operator[](const Key& key);
};
```
Example range-based for: `for (Key key : mymap){…}`

```
class Set {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem); // operator+= also adds elements
 bool contains(const Type& elem) const;
};
```
Example range-based for: `for (Type elem : mymap){…}`

```
class Lexicon {
  int size() const;
  bool isEmpty() const;
  void clear();
  void add(std::string word);
  bool contains(std::string word) const;
  bool containsPrefix(std::string prefix) const;
};
```
Example range-based for: `for (string str : english){…}`