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).** You are given a board from the Game of Life (`Grid<int> board`).[1] Adjacent living cells are able to transmit a virus from one to the other (adjacent means any of the 8 neighbors of a cell). Given the coordinates of an infected cell, and the coordinates of another cell, find the length of the **shortest** path between the two cells that may be a path for virus transmission (path consists only of living cells).
   - As in your assignment, a `Grid` value of `0` indicates a dead cell, and any integer greater than zero indicates a living cell. Age of living cells does not matter for this problem.
   - Your function should have the following signature:
     ```
     int shortestLivingPath(Grid<int>& board, Point infected, Point dest)
     ```
     - The Point `struct` is:
       ```
       struct Point {
             Point(int r, int c) {row = r; col = c;};
             int row;
             int col;
       }; //you may assume an == operator works on two Points
       ```
     - The `Grid` is **not** `const`, and you are free to modify it in your function. In particular, you'll want to mark cells you've already explored so you don't re-explore them.
     - Return the length of the shortest path, i.e. the number of cells in the path, including both endpoints.
     - If there is no way to get from the infected cell to the other cell on a path of only living cells, return -1.
   - Your code should not consider paths longer than the shortest path, i.e., you should explore paths in order of increasing length. *Hint:* Remember from WordLadder that to find the shortest ladder, you did a breadth-first search of the possible ladders, and that search involved a particular data structure. You should have a similar approach here.

   Write your solution on the next page.

[1] The Game of Life was an assignment that we did in the quarter when this exam problem was originally given. We haven't done that assignment this quarter, but you aren't at any disadvantage because of that. The only things you need to know are included in the problem, namely: you're dealing with a Grid<int> where each entry of the grid is either 0 (cell is dead) or greater than zero (cell is alive—the actual value is the age of the cell but only dead/alive matters for this problem).

```
int shortestLivingPath(Grid<int>& board, Point infected, Point dest) {

}
```

2. **Pointers and Memory (21pts).** Draw the state of memory at the end of the execution of this code. Be careful in showing where your pointers originate and terminate (outer box vs. inner box). Leave uninitialized or unspecified areas blank, and clearly mark NULL (write NULL or draw a slash through the box). Draw the components in the appropriate stack and heap areas marked for you. Mark memory that has been deleted by enclosing it in a circle with a slash through it, like this: ⃠ but leave it where it is, and do not change any pointer or other values unless they are actually changed.

```
Slayer vamp[3];
Slayer* stake;
vamp[1].angel = 2;
stake = vamp;
stake->angel = 4;
stake++;
vamp[0].buffy = stake;
stake->angel = 6;
(stake+1)->angel = 8;
stake = new Slayer;
vamp[2].buffy = new Slayer;
vamp[2].buffy->buffy = NULL;
vamp[1].buffy = stake;
delete stake;
//Draw the state of memory now
```

```
struct Slayer {
   int angel;
   Slayer * buffy;
};
```

DRAWING:

Stack:

Heap:

3. **Classes (30pts).** This is a problem about using classes to organize your classes (so meta!). Given that we are halfway through the quarter, you will most likely have started to think about which classes to take next quarter. We can think about each class as storing information such as: day of the week of its lecture, lecture start time, and lecture duration. Your goal in this problem is to implement a `Lecture` class that will represent this information.

   For simplicity, we assume that all Stanford classes meet either Monday-Wednesday-Friday (MWF) or Tuesday-Thursday (TTH), and no other combinations. Your `Lecture` should represent the lecture times for exactly one class. For example, for CS106X, your `Lecture` should remember that we meet MWF at 11:00 AM for 50 minutes (which you of course already knew).

   In addition to exposing the day, time, and duration for each lecture, you should also provide a couple of methods that will help students scheduling classes. Here is the interface:

```
enum DayPattern { MWF, TTH };

class Lecture {
public:
  // You'll implement these in Part (b)
  Lecture(DayPattern daysOfWeek, int startTime, int duration);

  int startTime() const;
  int durationInMinutes() const;
  DayPattern daysOfWeek() const;

  // You'll implement these in Part (c)
  int endTime() const;
  bool overlapsWith(const Lecture& other);

private:
  // You'll fill this in in Part (a)

};
```

   You may assume that no lecture will span midnight (e.g. will not start at 23:00 and last for more than one hour). You may also assume that all times will be military time in PST (e.g. 1700 not 5:00pm), so **you can handle hours and minutes as a single number** and not worry about time zones.

a) **(5 Points)** Fill in the `private` section of the `Lecture` class. This includes any instance variables as well method prototypes for helper methods you implement.

```
class Lecture {
public:
  Lecture(DayPattern daysOfWeek, int startTime, int duration);

  int startTime() const;
  int durationInMinutes() const;
  DayPattern daysOfWeek() const;

  int endTime() const;
  bool overlapsWith(const Lecture& other);

private:
  // Fill this in










};
```

b) **(10 Points)** Fill in the constructor and getters for the start time, duration, and weekday for the Lecture.

```
Lecture::Lecture(DayPattern daysOfWeek, int startTime, int duration) {







}
int Lecture::startTime() {




}
int Lecture::durationInMinutes() const {




}
DayPattern Lecture::daysOfWeek() const {




}
```

c) **(15 Points) I**mplement the method **`endTime`** that returns the time of day that a given lecture ends. For example, if your Lecture represents a CS106X lecture this quarter where the **`startTime`** is `1100`, you should return `1150`. You are welcome to use additional helper methods, but you are not required to.

```
int Lecture::endTime() const {



}
```

Now, implement **`overlapsWith`** to make sure you can schedule classes without conflicts. The method takes another `Lecture` as a parameter and returns true if lectures overlap. If one lecture starts when another ends, they do not overlap. For example, a lecture that starts at `1100` and lasts for 50 minutes does not conflict with a lecture that starts at `1150.`You are welcome to use additional helper methods, but you are not required to.

```
bool Lecture::overlapsWith(const Lecture& other) {



}
```

4. **Recursion (30pts).** Now that we've built a container to store lecture information, we are ready to tackle a closely related problem. Your job is to write a function that uses **recursive backtracking** to determine if it's possible to take at least k units of interesting classes. Your function should have the following signature:

```
bool canTakeAtLeastKUnits(const Vector<string>& interestingClasses,
                          const Map<string, Lecture>& schedule,
                          const Map<string, int>& units,
                          int k,
                          Vector<string>& mySchedule)
```

- **`const Vector<string>& interestingClasses`**
  A `Vector` of class names (e.g. `["CS106X", "CS55N", "PSYCH1"]`)

- **`const Map<string, Lecture>& schedule`**:
  A Map from class names to lecture slots. For example, the `schedule` map could contain:
  ```
  { "CS106X": Lecture(daysOfWeek: MWF, startTime: 1100, duration: 50),
    "CS55N":  Lecture(daysOfWeek: TTH, startTime: 1100, duration: 75),
    "PSYCH1": Lecture(daysOfWeek: TTH, startTime: 1200, duration: 50)
  }
  ```

- **`const Map<string, int>& units`**:
  A Map from class names to units for the class. For example, the `units` map could contain:
  `{"CS106X": 5, "CS55N": 3, "PSYCH1": 3 }`
- **`int k:`** The smallest number of units you are willing to take.
- **`Vector<string>& mySchedule:`** If you can take k units, you should return `true` and this `Vector` should contain the *names* of non-overlapping classes that total to **at least** k units. If it is not possible to find such a solution, you should return `false` and this `Vector` should be empty. Time to do Camp Stanford!

- In the example above, it is possible to take at least 1, 2, 3, 4, 5, 6, 7, or 8 units, but it is not possible to take more than 8 since two of the classes conflict.
- You may assume that data is well formed (k and the lecture durations are both non-negative, no classes span midnight). For simplicity, assume that `interestingClasses == schedule.keys() == units.keys()`, that is all three data structures contain exactly the same courses. You may assume that the `Lecture` interface from Problem 3 is available and works (not dependent on your Problem 3 solution being correct).
- If you're trying to match the recursive backtracking template, this problem requires careful (i.e., perhaps slightly unconventional) thinking about what a "step" is in your exploration and what your "options" are for a given step. Suggestion: consider your options for a given class to be that you can take it or not take it.
- You are welcome to use wrapper and helper function(s) as desired.

```
bool canTakeAtLeastKUnits(const Vector<string>& interestingClasses,
                          const Map<string, Lecture>& schedule,
                          const Map<string, int>& units,
                          int k,
                          Vector<string>& mySchedule) {
































}
```

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(Vector<int> vec){
    int N = vec.size();
    int sum = 0;
    for(int i = 0; i < vec.size(); i += (vec.size() / 3)) {
        cout << vec[i] << endl;
        sum += vec[i];
    }
    return sum;
}
```
	O( )

```
// For this problem, treat N as N = vec.size() of the
// vec of the *original* call to recursiveFind.
bool recursiveFind(Vector<int> vec, int key) {
    if (vec.size() == 0) return false;
    if (vec[vec.size()/2] == key) return true;
    Vector<int> left;
    for (int i = 0; i < vec.size()/2; i++) {
          left.add(vec[i]);
    }
    if (recursiveFind(left, key)) return true;
    Vector<int> right;
    for (int i = vec.size()/2 + 1; i < vec.size(); i++) {
          right.add(vec[i]);
    }
    if (recursiveFind(right, key)) return true;
    return false;
}
```
	O( )

```
int myfunction(int N) {
    Grid<int> matrix(N / 2, N / 2);
    for (int i = 0; i < N/2; i++) {
        for (int j = 0; j < N/2; j++) {
           matrix[i,j] = 3;
         }
     }
     cout << “done!” << endl;
}
```
	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){…}`