CS106X
Autumn 2014 October 8, 2014

Assignment 3 - Recursion

Assignment 3 is a big one, but we are giving you two weeks to work on it. The This assignment will solidify your understanding of recursion through a number of smaller warmups followed by a larger assignment that has you implement the game of Boggle. All four warmups and Boggle are due the same day, but keep in mind that Boggle is **significantly more than 20% of the total work for this assignment**. We trust you to manage your time accordingly.

Assignment Due: Monday, October 20nd, 2014 at 5:00 pm

For the four warm-up exercises, we specify the function prototype. **Your function must exactly match that prototype** (same name, same arguments, same return type). Your function must use recursion; even if you can come up with an iterative alternative, we insist on a recursive formulation!

Note that the Boggle assignment is much more involved than these warm-up problems. In practice, you'll want to press through these problems fairly quickly and move on to Boggle pronto.

It is fine to write "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment. Some parts of the assignment essentially require a helper to implement them properly. It is up to you to decide which parts should have a helper, what parameter(s) the helpers should accept, and so on. You can declare function prototypes for any such helper functions near the top of your `recursionproblems.cpp` file.

Files:
We provide you with a ZIP archive containing a starter version of your project. Download this archive from the class web site and finish the code. Turn in only the following files:

- `recursionproblems.cpp`, the C++ code to solve all parts of the assignment
- `Boggle.cpp` / `.h` & `boggleplay.cpp`, the C++ code to implement Boggle.

The ZIP archive contains other files and libraries; you should not modify them. When grading your code, we will run your file with our own original versions of the support files, so your code must work with them

Problem 1: Human pyramid (Cynthia Lee and Keith Schwarz)

For this problem, write a recursive function to compute the weight supported by a person in a "human pyramid":

```
double weightOnKnees(int row, int col, Vector<Vector<double> >& weights)
```

A *human pyramid* is a formation of people where participants make a horizontal row along the ground, and then additional rows of people climb on top of their backs. Each row contains 1 person fewer than the row below it. The top row has a single person in it. The image at right depicts a human pyramid with four rows.

Your task is to write a function to compute the weight being supported by the knees of a given person in the human pyramid. We will define the weight on a given person *P* 's knees recursively to be *P* 's own weight, plus half of the weight on the knees of each of the two people directly above *P*. For example, in the pyramid figure at right, the weight on the knees of person I below is I's own weight, plus half of the weight on the knees of persons E and F.

The weight on the knees of persons E and F can be computed recursively using the same process; for example, the weight on the knees of person E is E's own weight, plus half of the weight on the knees of person B, plus half of the weight on the knees of person C. If a given person does not have two people directly above them, any "blank" or "missing" persons should be ignored. For example, the weight on the knees of person F in the figure at right is F's own weight, plus half of the weight on the knees of person C. No rounding occurs during any of these recursive calculations.

To represent the pyramid we will use a 2-dimensional vector of vectors, where each person's own weight in kilograms is stored as a real number. So for example, `weights[0][0]` refers to the weight of the person at the top of the pyramid, and `weights[weights.size() - 1][0]` refers to the weight of the bottom-left person in the pyramid. The table below illustrates which person's weight from the above-right figure would be stored in which index of the vector. You can think of it as a left-aligned version of the human pyramid figure.

```
    col   0  1  2  3
row
 0      {{A},
 1       {B, C},
 2       {D, E, F},
 3       {G, H, I, J}}
```

vector of weights for the above pyramid

Our provided code will create the nested vector of weights and pass it to your function. **You may assume** that the vector passed to your function is valid and matches the structure described above, such that the pyramid will always be fully filled in with values in the proper indexes. That is, if there are *n* people on the bottom row, then there are *n*-1 people on the next row up, and *n*-2 people on the next row above that, and so on. Use *exactly* the signature shown above.

Do not modify the vector passed in. If the row/column passed is outside the bounds of the vector, **return 0**.

Our starter code provides the overall program and a loop to prompt for the pyramid's size. Here is a sample output:

```
How many people are on the bottom row? 4
Each person's own weight:
51.18
55.90 131.25
69.05 133.66 132.82
53.43 139.61 134.06 121.63
Weight on each person's knees:
51.18
81.49 156.84
109.80 252.82 211.24
108.32 320.92 366.09 227.25
```

Your solution should not use any loops or additional data structures; you must solve the problem using recursion.

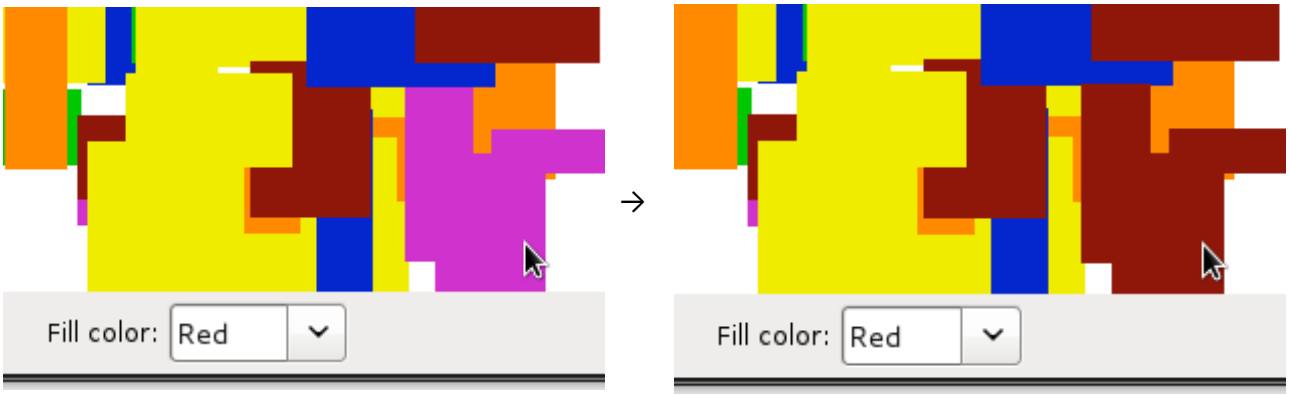
Problem 2: Flood Fill (Marty Stepp)

(This is a variation of book exercise 9.4.)

For this problem, write a recursive function to perform a "flood fill" on a graphical window as described below.

```
void floodFill(GBufferedImage& image, int x, int y, int color)
```

Most computer drawing programs make it possible to fill a region with a solid color. Typically you do this by selecting the "fill" tool and selecting a color, then clicking the mouse somewhere in your drawing. When you click, the paint color spreads outward from the point you clicked to every contiguous part of the image that is the same color as the pixel where you clicked. For example, if you select the Fill tool and choose Red as your active color, then if you click on a purple part of the image, the Fill operation will cause that part of the image, along with any purple part that is touching it, to become red. The screenshots below provide an illustration:



For this problem it is important to think of the graphical image on the screen as being composed of a 2-D grid of tiny square dots called *pixels* (short for picture elements). When the user clicks the mouse, whatever color that pixel was, it should be replaced by the fill color outward in all four directions: up, down, left, and right. For example, if the initial pixel color is purple, change it to red, and then also explore one pixel up, down, left, and right from there, looking for more purple pixels and changing them to red. If your exploration of a neighbor leads you to a pixel that is a different color (such as yellow rather than purple in our example), then you should stop exploring in that direction.

You can use the following functions on the `GBufferedImage` to get and set the color of individual pixels:

`int getRGB( double x, double y)`	returns the color of the given pixel as an integer
`void setRGB(double x, double y, int color)`	sets the color of the given pixel to the color as an integer.

For example, to set the color of the pixel at (4, 5) to be the same as the color at pixel (7, 9), you would write:

```
int color = image->getRGB(7, 9);     // set pixel (4, 5) to have the same
image->setRGB(4, 5, color);          // color as pixel (7, 9)
```

x/y: Your function accepts (x, y) parameters of the point the user clicked. Use these parameters as well as the functionality exported by the `GBufferedImage` to avoid going out of the bounds. If you try to get/set a pixel color that is out of bounds, it will crash.

`int color`: The colors of pixels are returned as `int` values. The exact values that map to various colors don't especially matter to your code, though you can see what color maps to what integer value by looking at the `recursionproblems.h` file. If you want to print a color out on the console for debugging, it is easier to view it in hexadecimal (base-16) mode. To do that, you'd issue a special '`hex`' command to `cout` to make it print an integer in hexadecimal rather than decimal (base-10), such as the following:

```
int color = image->getRGB(7, 9);
cout << hex << color << endl;
```

Your solution should not use any loops or data structures; you must use recursion. It is possible that on some machines, filling a very large area of pixels can cause your program to crash because of too many nested function calls ("stack overflow"). But this is out of your control and you may ignore this issue, so long as your algorithm is correct.

Problem 3: Marble Solitaire (Cynthia Lee)

The object of the Marble Solitaire game is to clear the board of marbles and leave the last remaining marble in the center position. Each move consists of a marble jumping over one of its orthogonal neighbors into an empty space. The jumped-over marble is cleared from the board. Here is a brief instructional video: http://youtu.be/vdwFTKt56hA?t=1m2s Some of you may have seen a similar game at the Cracker Barrel restaurant chain, where it is presented as a peg board game, rather than with marbles (the board layout is also different).

The provided code will allow you to play the game. Do take a moment to play the game to learn the rules for valid moves (briefly: no diagonal moves, jumps are only one marble wide, no jumping over an already empty space). You will soon discover that it is challenging to solve. After a few failed attempts, you might say to yourself (as I did over this past winter break), "I should just write a program to solve this!" Fortunately, you will be doing exactly that for this assignment. You will write code that does a **randomized depth-first/backtracking recursive search** of all possible moves, starting from the board configuration at the point where the human player gave up and asked the computer to take over.

Your only job is to implement the recursive part of the solver by filling in the body of the `solvePuzzle` function in `recursionproblems.cpp`. Although you should not edit anything else about the code, you may wish to have some decomposition of the `solvePuzzle` function by calling function(s) you add (one suggested function in particular is explained below). The `solvePuzzle` function signature is as follows:

```
bool solvePuzzle(Grid<MarbleType>& board,
                 int marblesLeft,
                 Set<uint32_t>& exploredBoards,
                 Vector<Move>& moveHistory);
```

- `board` is the current game board configuration. `MarbleType` is an `enum`, with three possible values: `MARBLE_INVALID` (for spaces that are not playable—i.e. the four corners of the board where there are no marbles), `MARBLE_OCCUPIED` (a marble is found here), and `MARBLE_EMPTY` (for spaces where a marble could be, but are currently empty).
- `marblesLeft` is a count of the marbles remaining on the board. The game is finished if only one marble remains on the board.
- `exploredBoards` is a set containing all the board configurations we have already explored. Because it is possible to reach a given board configuration via different sequences of moves, a search of all possible sequences of moves will perform some duplication of work. Your recursive function should test if `board` is found in `exploredBoards`, and if so, return `false` to avoid repeating that branch of exploration. Note that the type of `exploredBoards` is `Set<uint32_t>&`, not `Set<Grid<MarbleType>>&`, as you might expect. To save memory, we have provided you a way to encode a board configuration into a single 32-bit unsigned integer

(`uint32_t`). To do the conversion from your Grid-based `board` to the compressed `uint32_t` version, use this function found in `compression.h`:

- `uint32_t compressMarbleBoard(Grid<MarbleType>& board)`

Make sure you update the `exploredBoards` set during your recursion as you explore more boards.

- `moveHistory` is the sequence of moves that led to the current `board` (not including human-played moves, if any). If this recursive function returns true because it found a solution to the board, this vector should hold the winning moves. These are saved so that if/when a winning sequence is found and the function returns, the original calling function can reproduce the sequence of moves in the graphics display.

What We Provide:

- You should use the existing `makeMove` and `undoMove` functions that are provided for you.
- You should use our system for keeping track of previously explored boards, as described above (see "`exploredBoards`" explanation).
- You should take a moment to look in `marbletypes.h` to learn about these important things you need to use:
  - You should use the `move` struct provided in marbletypes.h. Recall that a struct is a feature of C++ that you could think of as a super-lightweight object. It is a way of gathering data into one place, but (usually) without the overhead of a constructor, get/set methods, etc. The `move` struct contains the data needed to represent one move (`startRow, startCol, endRow, endCol`).
  - You will also need to use the following enum, representing the three possible states of a position in the grid that represents the board:

```
enum MarbleType {
    MARBLE_OCCUPIED, /* has a marble in it */
    MARBLE_EMPTY,    /* could hold a marble but is empty */
    MARBLE_INVALID   /* can't hold a marble (four corners) */
};
```

- You do not necessarily need to use the `isValidMove` function in the function you write. `isValidMove` is designed to check *human* moves for validity, because a human may click on any two places. Your code can be smart enough not to generate invalid moves in the first place, so they won't need extra checking. However, you may find it easier to just use the `isValidMove` function.

Important instructions:

- Your search should be depth-first, which means something roughly like this pseudocode:

```
solvePuzzle():
  for each move in the possible next moves:
    if solvePuzzle() with the board after that move finds a solution:
        this move was a good move, return true!
    otherwise:
        undo the move
  if no good moves found, return false
```

- Per the pseudocode above, you will want to have a function that finds all possible next moves given the current `board`, so you can determine where the recursion should go next. Suggested signature for this added function: `Vector<Move> findPossibleMoves(Grid<MarbleType>& board);`
- If you always return the same `Vector` of moves for the same `Grid` in `findPossibleMoves`, the game becomes rather boring since you will always see exactly the same solution. One way to get around this is by using randomization. You can either insert elements at a random index in the `Vector<Move>` that keeps track of the result or, to achieve better performance, use a shuffling algorithm like Fisher-Yates. You can look up the details online, but the algorithm iterates through the collection and swaps element `i` with a random element between the indexes `i` and `size() - 1` inclusive. You can use either approach or something else entirely as long as you randomize the set of possible moves.
- You will notice that the demo code prints out a status update message while the computer is playing, to let you know it's still "thinking." This is because it can sometimes take a long time to find a solution, so it is comforting for the user to know that progress is being made. It is not required that your code do this, but if you would like to add this feature then insert the following code in your recursive function:

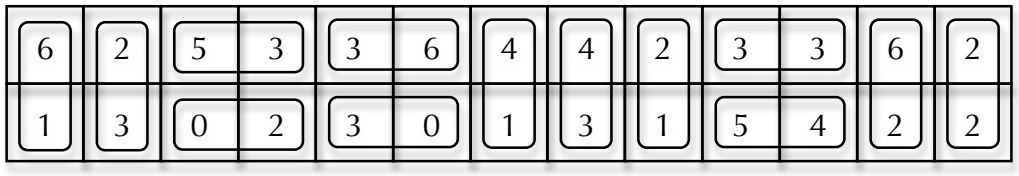
```
if (exploredBoards.size() % 10000 == 0){
    cout << "Boards evaluated: " << exploredBoards.size()
         << "\tDepth: " << moveHistory.size() << endl;
    cout.flush();
}
```

Problem 4: Finding Dominosa Solutions (Jerry Cain and Julie Zelenski)

The game of Dominosa presents a grid of small nonnegative integers, perhaps as follows:

6	2	5	3	3	6	4	4	2	3	3	6	2
1	3	0	2	3	0	1	3	1	5	4	2	2

There are always two rows of numbers, but the number of columns can, in principle, be any positive integer. The goal is to pair horizontally and vertically adjacent numbers so that every number takes part in some pair, and no two pairs include the same two numbers. As such, one solution to the above problem would pair everything as follows:



Sadly, not all boards can be solved. One small, obvious example is:

3	1
1	3

Run the sample application we've included in the collection of starter files. You'll see that the program generates random 2 x n boards (where you choose the value of n to be between 9 and 25 inclusive). For each randomly generated board, the application will animate the recursive backtracking search that determines whether some such pairing exists. The starter code provides the core of the interactive program, and it also provides a fully operational `DominosaDisplay` class that can be used to manage all aspects of the visualization. Your job is to implement the `canSolveBoard` function, which has the following prototype:

```
bool canSolveBoard(DominosaDisplay& display, Grid<int>& board);
```

You'll need to read over the `dominosa-graphics.h` file to see how the `DominosaDisplay` can be used to script the animation, which if properly implemented will do a superb job of visually confirming that your recursive backtracking algorithm is working properly.

Problem 5: Boggle (Eric Roberts and Julie Zelenski)

This assignment focuses on recursive backtracking. As a reminder, please only edit and turn in the following files:

- `Boggle.h` / `.cpp` : a `Boggle` class representing the Boggle game state
- `boggleplay.cpp` : client to perform console UI and work with `Boggle` class

We provide you with several other files to help you, but you should not modify them.

The Game of Boggle:

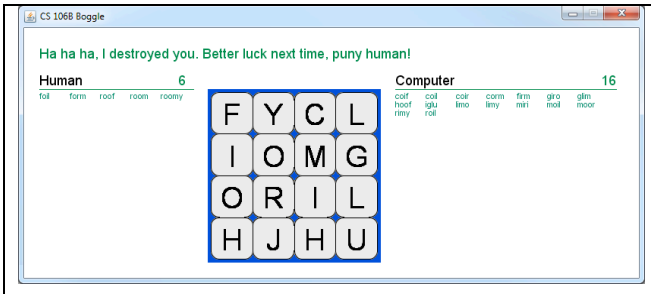
Boggle is a game played on a square grid onto which you randomly distribute a set of letter cubes. Letter cubes are 6-sided dice, except that they have a letter on each side rather than a number. The goal is to find words on the board by tracing a path through neighboring letters. Two letters are **neighbors** if they are next to each other horizontally, vertically, or diagonally. There are up to eight letters near a cube. Each cube can be used at most once in a word.

In the real-life version of this game, all players work at the same time, listing the words they find on a piece of paper. When time is called, duplicates are removed from the lists and the players receive one point for each unique word, that is, for each word that player found that no other player was able to find.

Your assignment is to write a program that plays this game, adapted for a single human player to play against a single computer opponent. Unfortunately, the computer knows **recursive backtracking**, so it can find every word on the board and destroy you every time. But it's still fun to write a program that can so soundly thrash you again and again.

To begin a game, you shake up the letter cubes and lay them out on the board. The human player plays first, entering words one by one. Your code first verifies that the word is valid, then you add it to the player's word list and award the player points according to the word's length (one point per letter ≥ 4). A word is valid if it meets all of the following conditions:

- at least 4 letters long
- is a word found in the English dictionary
- can be formed by connecting neighboring letter cubes *(using any given cube only once)*




```
Do you want to generate a random board? y
It's your turn!
FYCL
IOMG
ORIL
HJHU
...
Your words (3): {"FOIL", "FORM", "ROOF"}
Your score: 3
Type a word (or Enter to stop): room
You found a new word! "ROOM"
...

Your words (5): {"FOIL", "FORM", "ROOF",
                 "ROOM", "ROOMY"}
```

- has not already been formed by the player in this game yet *(even if there are multiple paths on the board to form the same word, the word is counted at most once)*

Once the player has found as many words as they can, the computer takes a turn. The computer searches through the board to find all the remaining words and awards itself points for those words. The computer typically beats the player, since it finds *all* words.

Your program's output format should exactly match the abridged log of execution at right. See the course web site for complete example output files.

```
Your score: 6
Type a word (or Enter to stop):

It's my turn!
My words (16): {"COIF", "COIL", "COIR",
"CORM", "FIRM", "GIRO", "GLIM", "HOOF",
"IGLU", "LIMO", "LIMY", "MIRI", "MOIL",
"MOOR", "RIMY", "ROIL"}
My score: 16
Ha ha ha, I destroyed you. Better luck
next time, puny human!
```

Setting up the Game Board:

The real Boggle game comes with sixteen letter cubes, each with particular letters on each of their six faces. The letters on each cube are not random; they were chosen in such a way that common letters come up more often and it is easier to get a good mix of vowels and consonants. We want your Boggle game to match this. The following table lists all of the letters on all six faces of each of the sixteen cubes from the original Boggle. You should decide on an appropriate way to represent this information in your program and declare it accordingly.

AAEEGN	ABBJOO	ACHOPS	AFFKPS	AOOTTW	CIMOTU	DEILRX	DELRVY
DISTTY	EEGHNW	EEINSU	EHRTVW	EIOSST	ELRTTY	HIMNQU	HLNNRZ

At the beginning of each game, "shake" the board cubes. There are two different random aspects to consider:

- A **random location** on the 4x4 game board should be chosen for each cube.
(For example, the `AAEEGN` cube should not always appear in the top-left square of the board; it should randomly appear in one of the 16 available squares with equal probability.)

- A **random side** from each cube should be chosen to be the face-up letter of that cube.
(For example, the `AAEEGN` cube should not always show `A`; it should randomly show `A` 1/3 of the time, `E` 1/3 of the time, `G` 1/6 of the time, and `N` 1/6 of the time.)

The Stanford C++ libraries have a file `shuffle.h` with a `shuffle` function you can use to rearrange the elements of an array, `Vector`, or `Grid`. See `shuffle.h` if you are curious about how the shuffling algorithm works:

Your game must also have an option where the user can enter a **manual board configuration**. In this option, rather than randomly choosing the letters to be on the board, the user enters a string of 16 characters, representing the cubes from left to right, top to bottom. (This is also a useful feature for testing your code.) Verify that the user's string is long enough to fill the board and re-prompt if it is not exactly 16 characters in length. Also re-prompt the user if any of the 16 characters is not a letter from A-Z. Your code should

work case-insensitively. You should not check whether the 16 letters typed could actually be formed from the 16 letter cubes; just accept any 16 alphabetic letters.

Human Player's Turn:

The human player enters each word she finds on the board. As described previously, for each word the user types, you must check that it is at least four letters long, contained in the English dictionary, has not already been included in the player's word list, and can be formed on the board from neighboring cubes. If any condition fails, alert the user. There is no penalty for trying an invalid word, but invalid words also do not count toward the player's list or score.

If the word is valid, you add the word to the player's word list and score. The length of the word determines the score: a 4-letter word is worth 1 point; a 5-letter word is worth 2 points; 6-letter words are worth 3; and so on. The player enters a blank line when done finding words, which signals the end of the human's turn.

Computer's Turn:

Once the human player is done entering words, the computer then searches the entire board to find the remaining words missed by the human player. The computer earns points for each remaining word found that meets the requirements (minimum length, contained in English lexicon, not already found, and can be formed on board). If the computer's resulting score is strictly greater than the human's, the computer wins. If the players tie or if the human's score exceeds the computer's, the human player wins.

You can find all words on the board using **recursive backtracking**. The idea is to start from a given letter cube, then explore neighboring cubes around it and try all partial strings that can be made, then try each neighbor's neighbor, and so on. The algorithm is roughly the following:

```
for each letter cube c:
   mark cube c as visited.                                  // choose
   for each neighboring cube next to c:
       explore all words that could start with c's letter.  // explore
   un-mark cube c as visited.                               // un-choose
```

Implementation Details:

You will write the following two sets of files. In this section we describe the expected contents of each in detail.

- `boggleplay.cpp` : client to perform console UI and work with your `Boggle` class to play a game
- `Boggle.h` / `.cpp` : files for a `Boggle` class representing the state of the current Boggle game

boggleplay.cpp: We have provided you with a file `bogglemain.cpp` that contains the program's overall `main` function. The provided code prints an introduction message about the game and then starts a loop that repeatedly calls a function called `playOneGame`. After each call to `playOneGame`, the main code prompts to play again and then exits when the user

finally says "no". The `playOneGame` function is *not* already written; you must write it in `boggleplay.cpp`. In that same file, you can place any other logic and helper functions needed to play one game. You may want to use the `getYesOrNo` function from `simpio.h` that prompts the user to type yes/no and returns a `bool`.

One aspect of the console UI is that it should "clear" the console between each word the user types, and then re-print the game state such as the board words found so far, score, etc. This makes a more pleasant UI where the game state is generally visible at the same place on the screen at all times during the game. See the provided **sample solution** for an example. Use the Stanford Library's `clearConsole();` function from `console.h` to clear the screen.

The `playOneGame` function should perform all console user interaction such as printing out the current state of the game. **This is the *only* file in which you should have any statements that read/write to cout or cin**. But `boggleplay.cpp` is *not* meant to be the place to store the majority of the game's state, logic, or algorithms.

Boggle.cpp/h: Instead, the majority of your code should be in the `Boggle.h` and `Boggle.cpp` files, which should contain the implementation of a `Boggle` class. A `Boggle` object represents the current board and state for a single Boggle game, and it should have member functions to perform most major game functions like finding words on the board and keeping score. Declare all `Boggle` class members in `Boggle.h`, and implement their bodies in `Boggle.cpp`. We provide you a skeleton that declares some required members below that your class *must* have.

Do not change the headings of any of the following functions. Do not add parameters; don't rename them. You must implement *exactly* these functions with *exactly* these headings, or you will receive a deduction.

Boggle class member	Description
`Boggle(`***`dictionary, boardText`***`)`	In this constructor you initialize your Boggle board to use the given dictionary lexicon to look up words, and use the given 16-letter string to initialize the 16 board cubes from top-left to bottom-right. If the string is empty, you should generate a random shuffled board.
`b``.getLetter(`***`row, col`***`)`	In this function you should return the character that is stored in your Boggle board at the given 0-based row and column. If the row and/or column are out of bounds, you should throw an int exception.
`b``.checkWord(`***`word`***`)`	In this function you should check whether the given word string is suitable to search for: that is, whether it is in the dictionary, long enough to be a valid Boggle word, and has *not* already been found. You should return a boolean result of true if the word is suitable, and otherwise you should return false.
`b``.humanWordSearch(`***`word`***`)`	In this function you should perform a search on the board for an individual word, and return a boolean result of whether the word can be formed. Your code for this function should use **recursive backtracking**. As each cube is explored, you should highlight it in the GUI to perform an animated search with a **100ms** delay *(see GUI*

	page). If the word is unsuitable, you should not perform the recursive search.
***b*.computerWordSearch()**	In this function you should perform a search on the board for *all* words that can be formed, and return them as a Set of strings. Your code for this function should use **recursive backtracking**. Though similar to your human search, this is different because you should look for all words and not perform any animation; therefore its code should be implemented separately from humanWordSearch.
***b*.humanScore()**	In this function you return the total number of points the human player has scored in the game so far as an integer. This total is 0 when the game begins, but whenever a successful human word search is performed, points for that word are added to the human's total score.
***b*.computerScore()**	In this function you should return the total number of points the computer player has scored in the game as an integer. This total is initially 0 when the game begins, but after a computer word search is performed, all points for those words are added to the computer's total score.
ostream& << b	You should write a << operator for printing a Boggle object to the console. The operator should print only the 4x4 grid of characters representing the board as four lines of text.

Once again for emphasis, do not modify the names or parameters of the preceding functions. Implement them as-is.

In some past assignments, we gave you an exact list of the functions to implement. In this assignment, we are asking you to come up with some of the members. The `Boggle` class members listed on the previous page represent a large fraction of that class's behavior. But you can, and must, add other members to implement all of the appropriate behavior. We also have not specified any of the private member variables that should go inside the `Boggle` class; you must decide those yourself. You must also decide what code and/or data should go in `boggleplay.cpp`, and what should go in the `Boggle` class. Part of the challenge of this assignment is learning how to design a class and console UI client effectively. Remember that each member function of your class should have a clear, coherent purpose.

Member variables: Here are some thoughts about **private data members** that you might need in your `Boggle` class:

- You'll certainly need a data structure to represent the current **game board state**, meaning the 16 letter cubes and what letter is showing on top of each cube. The exact choice of data structure is up to you, but you should make an efficient and appropriate choice from the Stanford libraries.
- It is fine to declare **additional data structures**, such as a collection of words found, etc.
- Don't make something a `private` data member if it is only needed by one function; make it local. Making a variable into a data member that could have been a local variable or parameter will hurt your Style grade.
- All data member variables inside a `Boggle` object should be `private`.

Member functions: Here are some suggestions for good **public member functions** to put in your `Boggle` class:

- Though the `boggleplay.cpp` file should do all console I/O, your `Boggle` class should have lots of convenient functions for it to call so that it doesn't need to have any complicated logic. For example, no recursion or backtracking should take place in `boggleplay`; all such recursive searching should happen in the `Boggle` class.
- The `boggleplay` code needs to be able to display various aspects of the game state, such as all words that have been found by the each player, along with the players' scores. The `Boggle` class should keep track of such things, NOT `boggleplay`. The `boggleplay` code should ask the `Boggle` class for this information by calling accessor functions on it, which should return appropriate data to the caller. Note that the `Boggle` class itself should not contain any output statements to `cout`; let `boggleplay` do that.
- Make a member function and/or parameter `const` if it does not perform modification of the object's state.
- Make a member function `private` if it is used internally and not to be called by the client *(a "helper")*.
- Do not add functions to your Boggle class that directly return internal data structures in a way that allows the client to make direct modifications to them. *(This is called "representation exposure" and is considered poor style.)*

Searching: You don't want to visit the same letter cube twice during a given exploration, so for the search algorithm to work, your `Boggle` class needs some way to "mark" whether a letter cube has been visited or not. You could use a separate structure for marking, or modify your existing board, etc. It's up to you, as long as it is efficient and works.

Efficiency is very important for this part of the program. It is important to limit the search to ensure that the process can be completed quickly. If written properly, the code to find all words on the board should run in around one second or less. To make sure your code is efficient enough, you must perform the following optimizations:

- use a `Lexicon` data structure to store the English dictionary, and do not needlessly copy the lexicon
- **prune** the tree of searches by not exploring partial paths that will be unable to form a valid word
- use efficient data structures otherwise in your program *(e.g. to represent which words are already found)*

One of the most important Boggle strategies is to **prune dead-end searches**. The `Lexicon` has a `containsPrefix` function that accepts a string and returns `true` if any word in the dictionary begins with that substring. For example, if the first cube you examine shows the letter `Z` and your algorithm tries to explore one of its neighbors that shows an `X`, your path would start with `ZX`. In this case, `containsPrefix` will inform you that there are no English words that begin with the prefix `"ZX"`. Therefore your algorithm should stop that path and move on to other combinations.

Development Strategy and Hints:

In a project of this complexity, it is important that you get an early start and work consistently toward your goal. To be sure that you're making progress, it also helps to divide up the work into manageable pieces, each of which has identifiable milestones. Here's a suggested plan of attack that breaks the problem down into six phases:

- **Task 1:** Cube setup, board drawing, cube shaking. Design your data structure for the cubes and board. As usual, no global variables. Set up and shuffle the cubes. Use the provided `shuffle` function and/or the `randomInteger` function from `random.h` to help you make random choices. Add an option for the user to force the board configuration, as illustrated by the sample application.

- **Task 2:** Human's turn *(except for finding words on the board)*. Write the loop that allows the user to enter words. Reject words that have already been entered, don't meet the minimum word length, or aren't in the lexicon. Don't worry about the recursive backtracking algorithm yet for verifying that the word can be formed from the cubes on the board; just perform the other validity checks and see if the word passes all of them.

- **Task 3:** Human's backtracking algorithm to find a given word on the board. Now use recursion to verify that a word can be formed on the board, subject to the various rules. You will employ recursive backtracking that "fails fast": as soon as you realize you can't form the word starting at a position, you backtrack.

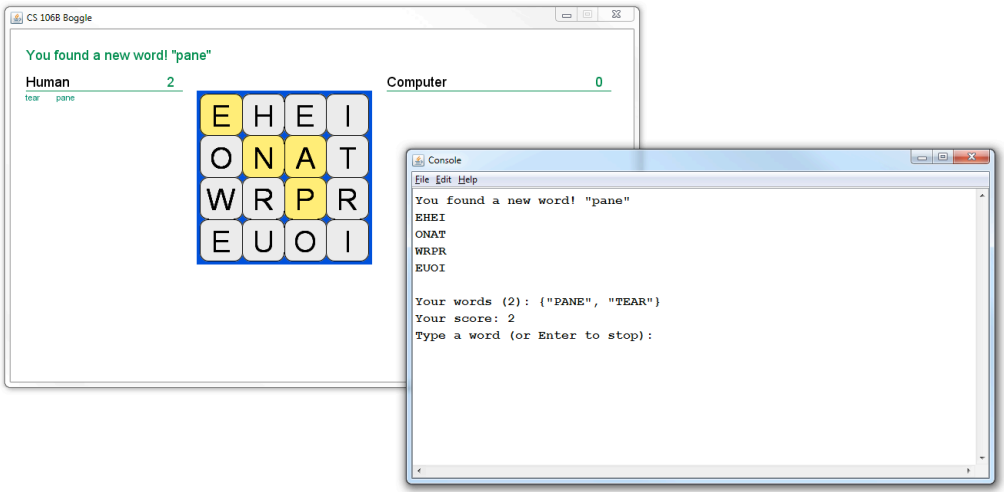
- **Task 4:** Computer's turn *(find all the words on the board)*. Now implement the killer computer player. Employing the power of recursion, your computer player traverses the board using an exhaustive search to find all remaining words. Be sure to use the lexicon prefix search to abandon searches down dead-end paths.

  NOTE: The program contains two recursive searches: one to find a specific word entered by the human player, and another to search the board exhaustively for the computer's turn. You might think that you should try to integrate the two into one combined function, by doing all word-finding at the beginning of the game, just after the board is initialized. But for full credit, **you *must* implement the human and computer player as two separate search functions**. There are enough differences between the two that they don't combine cleanly and the unified code is usually worse as a result. Focus on writing clean code that clearly communicates its algorithm.

- **Task 5:** Loop to play many games and add polish. Once you can successfully play one game, it's a snap to play many. Be sure to gracefully handle all user input so that it is not possible to break or crash the program.

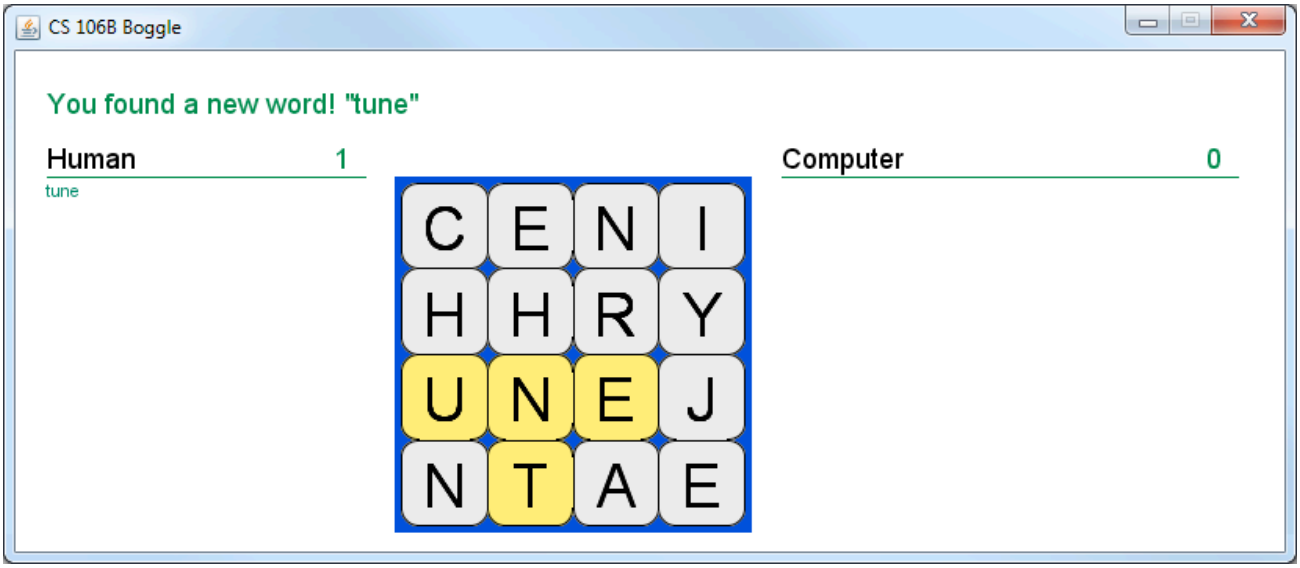
- **Task 6:** Add the graphical user interface and animation *(see next page)*. The GUI serves as a supplement to the existing text UI, not a replacement. So your text UI should still work properly in the presence of the GUI.

  Make sure to extensively **test** your program. Run the **sample solution** posted on the class web site to see the expected behavior of your program.



Graphical User Interface:

As a required part of this assignment, you must also add a graphical user interface (GUI) to your program. The GUI does not replace the console UI; it can't be clicked on to play the game, for example. It just shows a display of the current game state.



To add the GUI, include `bogglegui.h` in your code. It declares the following graphical functions that you can call from any code file:

`clearHighlight()`	Sets all letter cubes to be un-highlighted. (See `setHighlighted`.)
`initialize(`***`rows`***`, `***`cols`***`)`	Starts up the GUI and displays the graphical window. The board is drawn with empty squares and scores are 0. If called again, resets the board (see `reset`). Throws an error if ***`rows`*** or ***`cols`*** is not a positive integer from 1-6.
`isInitialized()`	Returns true if `initialize` has already been called.
`labelCube(` ***`row`***`, `***`col`***`, `***`ch`***`,` ***`highlighted`***`)`	Sets the given letter cube to display the given character. Rows and columns have 0-based indexes starting with (0, 0) at top-left. If `true` is passed for the optional ***`highlighted`*** boolean

	parameter, the cube is drawn with a colored highlight (useful to show progress of word searches). The highlight will remain until turned off. Throws an error if ***ch*** is not a letter or space.
`labelAllCubes(`***str***`)`	Sets all letter cubes to display the characters from the given string. For example, passing `"ABCDEFGHIJKLMNOP"` would label cube (0, 0) with `'A'`, cube (0, 1) with `'B'`, and so on. The string can contain other characters such as whitespace, line breaks, etc., which will be skipped over. All cubes are un-highlighted after a call to this function. Throws an error if ***str*** does not contain 16 alphabetic letters.
`recordWord(` ***word, player***`)`	Displays that the given player has found the given word string on the board. This function does not check word validity, e.g. that the word is not already shown, that it can be formed on the current board, is in the dictionary, etc. That is up to you. The ***player*** parameter indicates which player found the given word. The value you pass should be either `BoggleGUI::HUMAN` or `BoggleGUI::COMPUTER` .
`reset()`	Sets the GUI window back to its initial state, with the letter cubes blank and un-highlighted, the scores both at 0, and no solved words shown on the screen.
`setAnimationDelay(`***ms***`)`	Sets a pause/delay of the given number of milliseconds. After calling this, subsequent calls to `setHighlighted` or to `labelCube` that have ***highlight*** set to `true` will trigger a pause. This is useful for animating a word search algorithm.
`setHighlighted(` ***row, col, highlighted***`)`	Sets the given letter cube to be highlighted (`true`) or un-highlighted (`false`).
`setScore(`***score, player***`)`	Sets the GUI's score display for the given player to the given number. The GUI does not know anything about scoring rules for Boggle; it will accept any integer. The ***player*** parameter indicates which player found the given word. The value you pass should be either `BoggleGUI::HUMAN` or `BoggleGUI::COMPUTER` .
`setStatusMessage(`***str***`)`	Displays a status message in the bottom part of the window. Useful for showing messages such as telling the user that they have found a word, etc.
`shutdown()`	Closes the GUI window and frees memory associated with the GUI.

The functions of the GUI are enclosed in a *namespace* so that they do not conflict with any other global function names in your program. To call one of them, you must prefix the function's name with `BoggleGUI::` , such as:

```
// human records the word "hello"
BoggleGUI::recordWord("hello", BoggleGUI::HUMAN);
```

Style Guidelines and Grading:

In general, items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignments about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Don't forget to **cite any sources** you used in your comments. Refer to the course **Style Guide** for a more thorough discussion of good coding style.

Part of your grade will come from appropriately utilizing recursive backtracking to implement your word-finding algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Efficiency of your recursive backtracking algorithms, such as avoiding dead-end searches by pruning, is very important. Redundancy is another major grading focus; avoid repeated logic as much as possible. Your `Boggle` class will be graded on whether you make good choices about what members it should have, and other factors such as which are `public` vs. `private`, and `const`-correctness, and so on.

As for **commenting**, place a descriptive comment header on each file you submit. In your `.h` files, place detailed comment headers next to every member explaining its purpose, parameters, what it returns, any exceptions it throws, assumptions it makes, etc. You don't need to put any comment headers on those same members in the corresponding `.cpp` file, though you should place inline comments as needed on any complex code in the bodies.

For reference, our solution to the `Boggle` class has 12 public member functions, 4 private member variables (fields), and a few private helper member functions. Our `Boggle.cpp` is around 180 lines not including comments, and our `boggleplay.cpp` is around 80 lines including comments. You don't have to match these numbers or even come close to them; they are just to use as a reference and a sanity check.

Please remember to follow the **Honor Code** on this assignment. Submit your own work; do not look at others' solutions. Cite sources. Do not give out your solution; do not place a solution on a public web site or forum.

Possible Extra Features:

- **Make the Q a useful letter:** The Q is largely useless unless it is adjacent to a U, so the real Boggle prints Qu together on a single face of the cube. You use both letters together, a strategy that not only makes the Q more playable but also allows you to increase your score because the combination counts as two letters.
- **Big Boggle:** Once you have a working program, it should require only a few changes to support a variant that uses a 5 × 5 board. Word game aficionados generally agree that the original size was just a bit too small and scaling it up adds to the fun and challenge. This is a great exercise in verifying that your design is sufficiently organized and flexible to permit this

adaptation. Our starting code declares two different cube arrays, one with the 16 cubes for the standard game and another with the 25 cubes for the bigger version.

- **Embellish the GUI:** Our Boggle GUI module is supplied in source form so you can adapt into a snazzier interface. For example, the current game merely highlights the word; it might be nice if it also drew lines or arrows marking the connections. Or you could use the Stanford C++ library's `gevent.h` facilities to let the user assemble a word by clicking or dragging the mouse through the letter cubes. Make it play sounds. Etc.
- **Board exploration:** As you will learn, some Boggle boards are a lot more fruitful that others. Write some code to discover things about the possible boards. Is there an arrangement of the standard cubes that produces a board containing no words? What about an arrangement that produces a longest word, maybe even using all the cubes? What is the highest-scoring board you can construct? Recursion will be handy in trying out all the possible arrangements, but there are a lot of options (do the math on all the permutations...), so you may need to come up with some heuristics to direct your explorations.

Submitting with extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found. Since we use automated testing for part of our grading process, if your feature(s) cause your program to no longer match the expected output test cases provided, submit two versions of your files: a first one without any extra features added (or with all necessary extensions disabled or commented out), and a second one with the extensions enabled.