Stanford CS 106B: Boggle

This version of the assignment is by Marty Stepp. Based on previous versions built by Julie Zelenski and Eric Roberts.

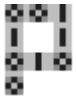
This is a **pair assignment**. You are allowed to work individually or work with a single partner. If you work as a pair, **comment both members' names** on top of every submitted code file. Only one of you should submit the assignment; do not turn in two copies.

Links:

starter code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following file. The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified. If you want to declare function prototypes, declare them at the top of your **.cpp** file, not by modifying our provided **.h** file.

- **bogglesearch.cpp**, the C++ code for your solution



turn in

When you are finished, submit your assignment using our **Paperless** web system. You can turn in all parts of the assignment together, or turn in each problem separately; it is up to you.

demo JAR

If you want to further verify the expected behavior of your program, you can download the following provided sample solution demo JAR and run it. If the behavior of our demo in any way conflicts with the information in this spec, you should favor the spec over the demo.

How to run it? (click to show)

output logs:

- log #1
- log #2
- log #3

Problem Description:

Boggle is a board game with a 4x4 square grid of letter cubes where players try to make words by connecting letters on adjacent cubes. For this part of the assignment, you will write code to search a Boggle board for words using backtracking.

We provide you with starter code in **bogglemain.cpp** that implements the text user interface of the game. You must write the functions to search the board for words in **bogglesearch.cpp**.

Letter cubes are 6-sided dice with a letter on each side rather than a number. In the real game, you shake up the letter cubes and lay them out on the board. In our version, we will ask you to type the 16 letters that make up the board.

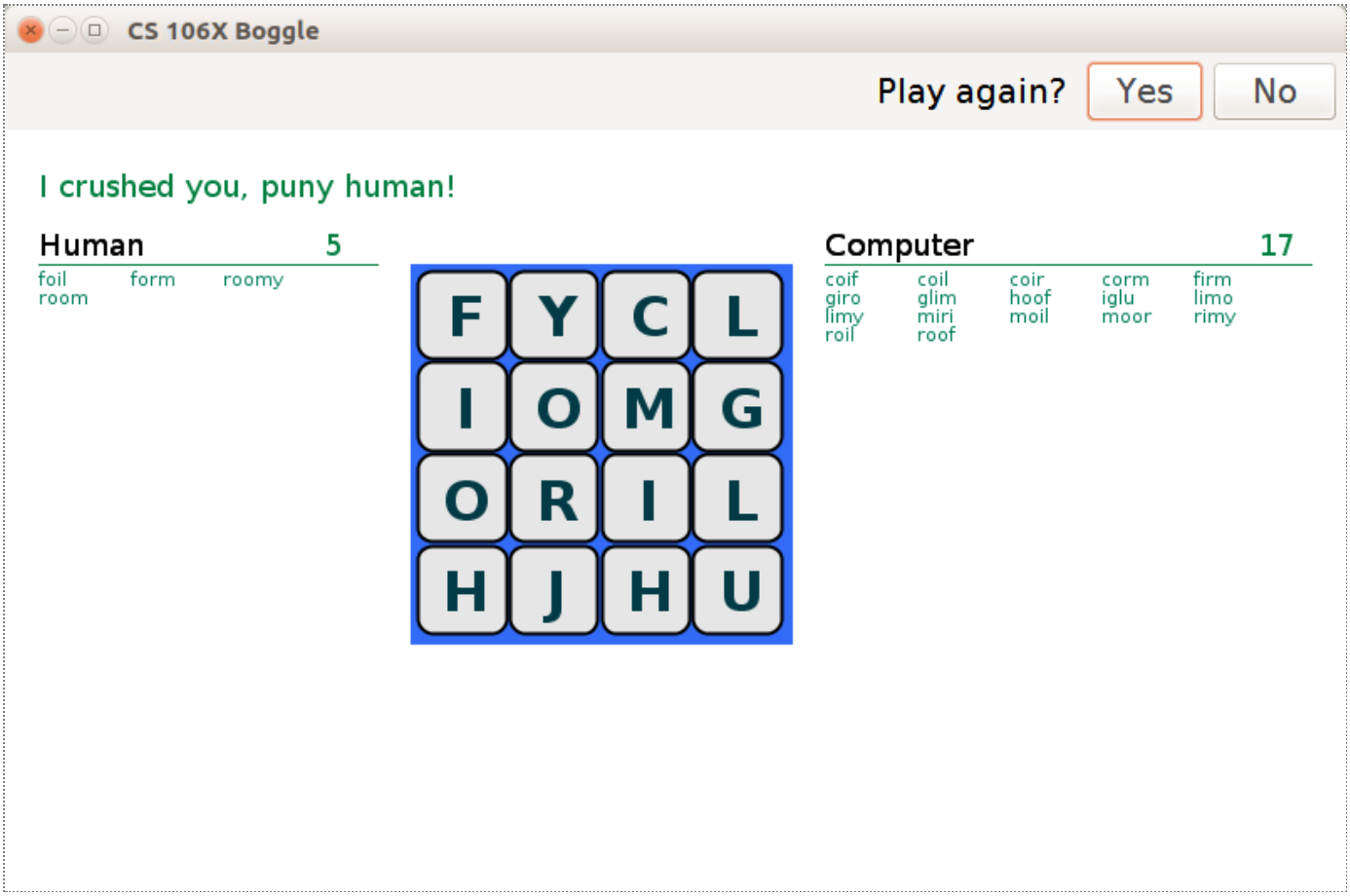
The goal is to find words on the board by tracing a path through **neighboring letters**. Two letter cubes are neighbors if they are next to each other horizontally, vertically, or diagonally. Therefore there are up to eight neighbors near a cube. Each cube can be used at most once in a given word.

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. But in the version we will write, a single human player will play a single turn against an omniscient computer opponent.

The **human player** plays first, entering words one by one. Your code verifies that the word is is at least 4 letters long, then uses backtracking to see if the word can be made using letters on the board, using any cube at most once.

Once the player has found as many words as they can, the **computer player** takes a turn. The computer searches through the board using recursive backtracking to find all the possible words that can be formed. The computer's algorithm is similar to the human's, but rather than verifying the existence of a single word in the board, you are exhaustively finding the set of all possible words.

Comparing output: Your program uses an instructor-provided **graphical user interface**, and it produces only a small amount of console output. The Stanford console window does not pop up; the console output will appear in the bottom Application Output area of Qt Creator to help you debug your code. You can copy/paste your console output from the bottom of Qt Creator into our web-based Output Comparison Tool page to see if your boards' words match ours.



Implementation Details:

You must implement the following functions to complete the Boggle game. You must write the functions with exactly the headers shown here. (It is fine to have "helper functions" as needed.) None of your functions should print any output to the console.

The game also has an option where the user can enter a manual board configuration rather than choosing one randomly. 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 a useful feature for testing your code. You don't need to write any code to support the manual board configuration option; it is already provided by the GUI. The manual configuration is a loophole because it allows the user to type a board that is not possible to create using the real letter cubes, such as a board of all Qs or something like that. We won't try to catch and prevent such a thing, partly because it is useful to be able to test your code on unusual boards.

Human player's turn:

```
bool humanWordSearch(Grid<char>& board, Lexicon& dictionary, string word)
```

Implement the human turn by writing a `humanWordSearch` function with exactly the heading above. The grid of characters represents the 4x4 game board, and the string word represents the word that the player wants you to search for. Your function should determine whether this word can be formed by connecting neighboring cubes of letters on the board. If so, you should return `true`; if not, you should return `false`.

Certain cases should cause your function to *immediately* return `false` and not perform any recursive search. One such special case is if the word passed is not found in the given dictionary. Another special case is if the word passed has a length less than `BoggleGUI::MIN_WORD_LENGTH` (which defaults to 4).

Your code for this function should use **recursive backtracking** to search the board for the letters of the given word. If the word is an unsuitable length (less than the minimum just described), you should not perform the recursive search. Your method should be **case-insensitive**; you should properly search for the word whether it is passed in upper, lower, or mixed case. You may assume that the characters on the board grid are all uppercase letters from A-Z, and that the word passed contains no characters other than letters from A-Z.

Recall that algorithms that use backtracking often rely on additional parameters to keep track of choices. For either of the functions in this problem, you may write **helper functions** that accept any parameters you like, so long as your overall `humanWordSearch` function has exactly the heading above. Your function should not modify the board grid or dictionary lexicon that are passed to it. Or if you do modify it temporarily, you should put it back to its original state before your function returns.

You don't want to visit the same letter cube twice during a given exploration, so for the search algorithm to work, your code needs some way to **"mark"** whether a letter cube has been visited or not. In this problem you will need to come up with your own way to mark or remember what squares you have already explored. You will have to decide what is the best strategy to use for marking; it's up to you, as long as it is efficient and works.

Scoring: If the word is valid and can be formed on the board (if your method would return `true`), you must also indicate to the GUI that the player has scored points. You can do this using the method `BoggleGUI::scorePoints` as described later in this document. The length of the word determines the score as follows:

- Words of 4 or fewer letters are worth 1 point.
- 5-letter words are worth 2 points.
- 6-letter words are worth 3 points.
- 7-letter words are worth 5 points.
- Words longer than 7 letters are worth 11 points.

There is no penalty for trying an invalid word, but invalid words also do not count toward the player's list or score.

Graphics: As each cube is explored, you should **highlight** it in the GUI to perform an animated search (see GUI section later). That is, when your search looks at a given cube on the board and examines it to decide whether it can be included in the word you are searching for, set it to be highlighted. If you backtrack and decide not to use that letter cube in your word search, set it to be un-highlighted. At the start of your human word search, you should clear any highlighting that exists on any letter cubes from previous searches. You can do the highlighting using the methods `BoggleGUI::setHighlighted` and `BoggleGUI::clearHighlighting` as described later in this document.

Board size: Note that both of your word search functions must work properly whether you are passed a 4x4, 5x5, 6x6, or any other size board. You may assume that the board's state is valid, that every character in the grid is an uppercase letter from A-Z, and so on.

Computer player's turn:

```
Set<string> computerWordSearch(Grid<char>& board,
                               Lexicon& dictionary,
                               Set<string>& humanWords)
```

Implement the computer's turn word search by writing a `computerWordSearch` function with exactly the heading above. The grid of characters represents the game board, and the lexicon represents the set of all valid English words in the dictionary. Your code should find *every word* that can be made on the given board that is at least 4 letters long and is found in the given dictionary. Your function should return a `Set` of all such words, all in uppercase.

Your code for this function should use **recursive backtracking** to search the board for all suitable words that can be formed using its letters. The idea is to do a search starting from each of the 16 letter cubes, looking for all valid words that start with that cube's letter. You can explore that starting cube, then each of its neighboring cubes, then each neighbor's neighbor, and so on. Along the way, as you find valid words, you should gather them into a collection, which will eventually be returned at the end of all exploration.

Though similar to your human search, the code is different because you should look for all words, not just verify a single word. Therefore we insist that the code for the computer search must be implemented separately from `humanWordSearch`. **Repeat: Do not try to combine the human and computer into a single helper function, and do not have one of the wordSearch functions call the other one. Write them as completely separate algorithms.**

Your computer word search is passed a set of strings representing all of the words that the human found during his/her turn. The computer's goal is to find all of the words that the human did not already find. So if your recursive search finds a word that was already found by the human, you should not include it in the result that is returned by your computer word search.

Scoring: As with the human player's turn, you must compute the number of points scored by the computer for each word and indicate this to the GUI using the method `BoggleGUI::scorePoints`. See the section on the human word search for the list of how many points each word is worth.

Efficiency and pruning: 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` 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)

Pruning: 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, `lex.containsPrefix("ZX")` will return `false` to 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. If you do not implement this optimization, your search will be too slow.

Graphics: Unlike the human search, the computer word search **does not perform any highlighting** in the GUI. You should, however, tell the GUI to score the points for each valid word that the computer finds.

Make sure to extensively **test** your program. Run the demo solution linked from this document to see the expected behavior of your program. When in doubt, match the behavior described in this spec and/or that of the demo.

Graphical User Interface (GUI):

This problem uses an instructor-provided graphical user interface (GUI) for all user interaction rather than a console UI. The GUI is represented by the files **boggleui.h** and **boggleui.cpp**. The `BoggleGUI` class has the following static member functions that you can call as needed from your Boggle searching code:

Static Member Function	Description
`BoggleGUI::clearHighlighting();`	Sets all letter cubes to be un-highlighted.
`BoggleGUI::playSound("filename");`	plays a sound effect from the given audio file
`BoggleGUI::scorePointsComputer(points);`	Adds the given number of points to the computer player's score
`BoggleGUI::scorePointsHuman(points);`	Adds the given number of points to the human player's score
`BoggleGUI::setHighlighted(row, col, highlighted);`	Sets the letter cube at the given 0-based row/col index to be highlighted (`true`) or un-highlighted (`false`)

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1-3 specs, such as the ones about good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style contraints specific to this problem:

Recursion and backtracking: 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. Avoid **"arm's length" recursion**, which is where the true base case is not found and unnecessary code/logic is stuck into the recursive case. Efficiency of your recursive backtracking algorithms, such as avoiding dead-end searches by pruning, is very important.

Redundancy in recursive code is another major grading focus; avoid repeated logic as much as possible. As mentioned previously, it is fine (sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Variables: While this constraint is not new to this assignment, we want to stress that you should not make any **global variables** or **static variables** (unless they are constants declared with the `const` keyword). Do not use globals as a way of getting around proper recursion and parameter-passing on this assignment.

Loops/Collections: Loops and collections *are* allowed on this problem. But your fundamental algorithm must be recursive and not based on looping to perform the entire word search. You must use recursion to handle the self-similar aspects of the problem.

Commenting: Of course you should have a comment header at the top of your code file and on top of each function. But we want to remind you that you should also have **inline comments** inside functions to explain complex sections of the code. Don't forget to place descriptive inline comments as needed on any complex code in the bodies to describe nontrivial parts of your algorithms.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

Boggle FAQ (click to show)

Possible Extra Features:

Here are some ideas for extra features that you could add to your program:

- **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 x 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 starter GUI code declares multiple different cube collections, one with the 16 cubes for the standard game and another with the 25 cubes for the bigger version. You can change our GUI constants for the max board size to get started.
- **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.

- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done 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 (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name without any extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.

Survey: After you turn in the assignment, we would love for you to fill out our anonymous CS 106B homework survey to tell us how much you liked / disliked the assignment, how challenging you found it, how long it took you, etc. This information helps us improve future assignments.

Honor Code Reminder: Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.