Assignment 6: Fun with Recursion

After endless mountains and rivers that leave doubt whether there is a path out, suddenly one encounters the shade of a willow, bright flowers and a lovely village.

— ***Lu You* (1125 - 1210), Southern Song Dynasty**

The poem is dedicated to those who work hard to learn recursive programming. Being staunch recursion programmers, we believe that, as described in the sage's poem, there will be rewards after hardships.

In this assignment, you may work alone, or pair with one partner from your class. Go and enjoy the fun of recursion!

Problem 1 Game 24

The Game 24 is an arithmetical card game in which the objective is to find a way to manipulate four cards so that the calculating result of the ranks of the cards is 24. For example, for the cards `(5C, 10S, 4H, 9C)` with the ranks 5, 10, 4, 9, one of solutions is `(((10S – 9C) + 5C) * 4H)`.

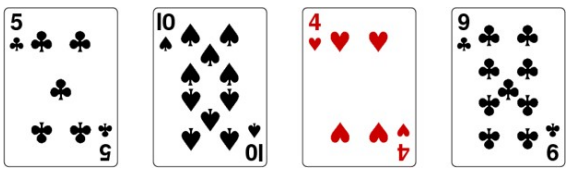
The game has been played in Shanghai since the 1960s, using playing cards. The original version of Game 24 is played with an ordinary deck of playing cards. The game proceeds by having 4 cards dealt and the first player that can achieve the number 24 exactly using only allowed operations (addition, subtraction, multiplication, division, and parentheses) wins the hand.

In the card game, the players must use all four cards, but use each card only once. Usually, there are many different ways of getting the answer 24, with aces counting as 1 and all face cards, jacks, queens, and kings counting as 11, 12, 13, respectively. Sometimes, there are also 4 - card combinations that can never make 24. One example is `(3H,9D,4S,10C)`. In some cases, such as `(AD, 5H, 5S, 5C)`, it is impossible to reach 24 only through integer operations.

Write a function

```
Set<string> countTwentyFours(Vector<Card> & cards)
```

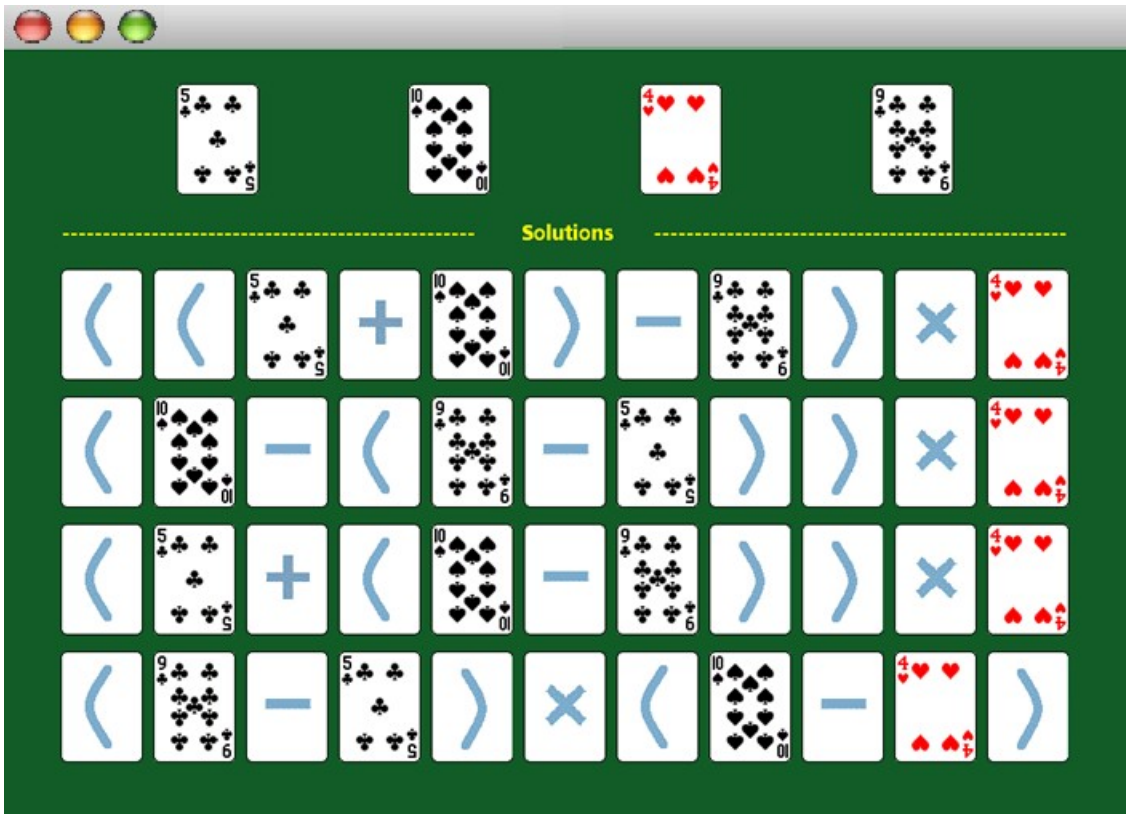
that takes a vector of `Card` objects and returns the collection of **`expression strings`** that describe the entire calculation process. For example, for the hand of cards:



the returning set of strings should contain the following expression strings:

(((5C + 10S) - 9C) * 4H)	**((10S - (9C - 5C)) * 4H)**
((5C + (10S - 9C)) * 4H)	**((9C - 5C) * (10S - 4H))**

Moreover, all these expression strings are illustrated by the following sample run:



You don't need to care about the graphical display of expression strings. To solve the problem, you must apply the **`Card`** class which is defined and implemented in **`card.h`** and **`card.cpp`**. Compared with Problem listAllFiteens in Assignment 3, the **`Card`** class has been updated in term of card **`rank`** type and **`operators (+, -, *, /)`** overloading, as well as **`expression string`**. Specially, the type of card rank is designed as double rather than integer. The overloaded operators support addition, subtraction, multiplication, and division of the ranks of cards. Furthermore, the private data member - expression string is used to describe the operation process. For example,

```
// declare four cards: 5C, 10S, 9C and 4H
Card c1("5C"), c2("10S"), c3("9C"), c4("4H");
// calculate ((5C+10S) - 9C) * 4H, and assign it to Card result
Card result = ((c1 + c2) - c3) * c4;
// output the expression string and the rank of Card result
cout << " Expression String: " << result.toExpressionString() << endl;
cout << " Calculation Result: " << result.getRank() << endl;
```

Output:

```
Expression String: ( ( ( 5C + 10S ) - 9C ) * 4H )
Calculation Result:  24
```

Your task is to implement **`countTwentyFours`** function through a recursive strategy. Don't try to construct all possible expressions from the four cards and evaluate them one-by-one to see if they can reach 24. Instead, given four cards, the recursive strategy is: try to combine pairs of cards and reduce them recursively, until there is only one card remaining, and then check if the rank of the one evaluates to 24.

For example, if we started with a hand of cards **`(AD, 5S, 5S, 5S)`**, this is roughly the thought process that occurs:

```
(AD, 5S, 5S, 5S)->(5S, Result1, 5S)->(5S, Result2)->(Result3)->success!
```

where, **`Result1 = AD / 5S`**, with rank **`0.2`** and expression string **`"( AD / 5S )"`**. **`Result2 = 5S - Result1`**, with rank **`4.8`** and expression string **`"( 5S - ( AD / 5S ) )"`**; **`Result3 = Result2 * 5S`**, with rank **`24`** and expression string **`"( ( 5S - ( AD / 5S ) ) * 5S )"`**.

As such, it becomes apparent that this is a recursive solution: solving for 4 cards is the same as choosing to combine 2 cards and solving for the remaining 3 cards. What a solid recursive step!

Extension:

If you complete the function design and test it. There are several aspects that can expand and improve your design:

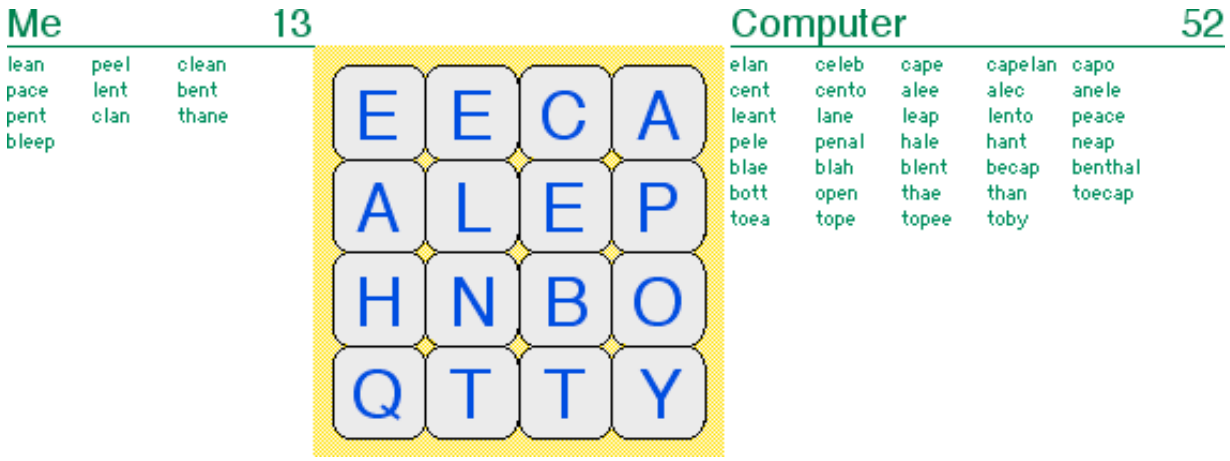
- How to eliminate the obvious duplicate results?
- The real recursive strategy is to support Game 24 with n cards. The Game 24 demo program implements at least 2 cards, up to 6 cards for Game 24. You can try to modify the program to achieve it.
- When the number of cards increases, for example, n = 5 or 6, it will be time-consuming to execute the recursive strategy. Please optimize your recursive program and don't let the user wait too long.

Accessing files

On the class web site, there is a zipped file which contains the following files:

Game24.cpp	Source file for your Game 24 implementation.
display.cpp	Source file that implements the graphics interface.
display.h	Interface file for the graphics module.
card.cpp	Source file that implements the Card class
card.h	Interface file for the Card class.
res	Folder of resource which contains the gif files of cards
Game24Demo	Folder of working program that illustrates how the game is played.

Problem 2 The Game of Boggle



The Boggle board is a 4x4 grid onto which you shake and randomly distribute 16 dice. These 6-sided dice have letters rather than numbers on the faces, creating a grid of letters from which you can form words. In the original version, the players all start together and write down all the words they can find by tracing by a path through adjoining letters. Two letters adjoin if they are next to each other horizontally, vertically, or diagonally. There are up to eight letters adjoining a cube. A letter can only be used once in the word. When time is called, duplicates are removed from the players' lists and the players receive points for their remaining words based on the word lengths.

This part of Assignment 4 is to write a program that plays a fun, graphical rendition of this little charmer, adapted to allow the human and machine to play one another. As you can see from the screen shot above, the computer basically trounces all over you, but it's fun to play anyway.

The main focus of this part of the assignment is designing and implementing the recursive algorithms required to find and verify words that appear in the Boggle board. This problem is larger than any of the four problems appearing in Assignment 3, so don't be lulled into thinking it can be done in one sitting.

How's this going to work?

You will read the letter cubes in from a file and shake the cubes up and lay them out on the board graphically. The human player gets to go first (nothing like trying to give yourself the advantage). The player proceeds to enter, one at a time, each word that she finds. **Your program is to verify that the word meets the minimum length requirement (which is 4)**, has not been guessed before, is a legal word in the English language, and can, in fact, be formed from the dice on the board. If so, the letters forming the word are highlighted, the word is added to the player's word list, and she is awarded points according to the word's length.

The player indicates that she is done entering words by hitting a lone extra carriage return. At this point, the computer gets to take a turn. The computer player searches through the board looking for words that the player didn't find and award itself points for finding all of them. The computer typically beats the player mercilessly, but the player is free to try again, you should play as many games as you want before exiting. Each time you repeat this entire process.

The Dice

The letters in Boggle are not simply chosen at random. Instead, the letter cubes are designed in such a way that common letters come up more often and it is easier to get a good mix of vowels and consonants. To recreate this, we give you a text file that contains descriptions of the actual sixteen dice used in Boggle. (We've also included a file for the 5 x 5 version of Boggle, but that's completely optional and you don't need to build the 5 x 5 version if you don't want to.) Each die description is a single line of 6 letters; they are not separated by spaces. For example, the 4 x 4 file might look something like this:

```
AAEEGN       // 4 x 4 version Boggle board, detailed in cubes16.txt
ABBJOO
ACHOPS
// 13 more lines follow this one


AAAFRS       // 5 x 5 version Boggle board, detailed in cubes25.txt
AAEEEE
AAFIRS
// 22 more lines follow this one
```

During initialization, you read the cubes files and store it into a suitable data structure for subsequent use. At the beginning of each game, "shake" the board cubes. There are two different random aspects to consider. First, the cubes themselves need to be shuffled so that the same cube is not always in the same location on the board. Second, a random side from each cube needs to be chosen to be the letter facing up.

To rearrange the cubes on the board, you should use the following shuffling algorithm, presented here in pseudo-code form:

Copy the constant array into a vector **vec** *so you can modify it. Shuffle* **vec** *using the following approach:*

```
for (int i = 0; i < vec.size(); i++)
                                                    {  Choos
    e a random index r between i and the last element position,
    inclusive. Swap the element at positions i and r.
}
```

Fill the Boggle grid by choosing the elements of **vec** *in order.*

This code makes sure that the cubes are arranged randomly in the grid. Choosing a random side to put face-up is straightforward. Put these two together and you can shake the cubes into many different board combinations.

Alternatively, the user can choose to enter a custom board configuration. In this case, you still use your same board data structure. The only difference is where the letters come from. The user enters a string of characters, representing the cubes from left to right, top to bottom. Verify that this string is long enough to fill the board and re-prompt if it is too short. If it's too long, just ignore the ones you don't need. You do not need to verify that the entered characters are legal letters.

Once your initialization is complete, you're ready to implement the two types of recursive search: one for the user, and one for the computer.

There are two distinct types of recursion happening here. For the user, you search for a specific word and stop as soon as you find it. For the computer, you are searching for all words. While you may be tempted to try and integrate the two so they work as a single type of recursion, this is not a good idea for two reasons. One is that your program will get very messy trying to integrate the two, as they are not algorithms that can be unified well. The other is that we want you to get practice with both types. Because of this, we explicitly require that you implement two separate recursive functions, one for the human player's turn (searching for a specific word) and for the computer's turn (exhaustive search for all words).

The human player's turn

After the board is displayed, the player gets a chance to enter all the words she can find on the board. Your program must read in a list of words until the user signals the end of the list by typing a blank line. As the user enters each word, your program must check the following conditions:

- That the word is at least four letters long.
- That it is defined in the lexicon as a legal word.
- That it occurs legally on the board (i.e., it is composed of adjoining letters such that no board position is used more than once).
- That it has not already been included in the user's word list.

If any of these conditions fail, you should tell the user about it and not give any score for the word. If, however, the word satisfies all these conditions, you should add it to the user's word list and update their score appropriately. In addition, you should use the facilities provided by the **gboggle.h** interface to highlight the word. Because you don't want the highlight to remain on the screen indefinitely, you should highlight the letters in the word, pause for about a second using the **Pause** function in the extended graphics library, and then go back and remove the highlights from the letters in the word.

Word length determines point value: 1 point for the word itself and 1 additional point for every letter over the minimum. Since the minimum word length is 4, **"boot"** gets 1 point, **"smack"** gets 2, and **"frazzled"** gets 5. The functions in the **gboggle** module will help you to display the player word lists and track the scoring. The player enters a lone carriage return (blank line) when done entering words.

The computer's turn

On the computer's turn, your job is to find all of the words that the human player missed by recursively searching the board for words beginning at each square on the board. In this phase, the same conditions apply as on the user's turn, plus the additional restriction that the computer is not allowed to count any of the words already found.

As with any exponential search algorithm, it is important to limit the search as much as you can to ensure that the process can be completed in a reasonable time. **One of the most important strategies is to recognize when you are going down a dead end so you can abandon it.** For example, if you have built a path starting with the prefix "zx", you can use the Lexicon's containsPrefix method to determine that there are no English words beginning with that prefix. So,

you can stop right there and move on to more promising combinations. If you miss this optimization, you'll find yourself taking long coffee breaks while the computer is busy checking out non-existent

words like **"zxgub"**, **"zxaep"**, etc. Not what you want.

The gboggle module

As mentioned before, we have written all the fancy graphics functions for you. The functions from the **gboggle.h** interface are used to manage the appearance of the game on the display screen. It includes functions for initializing the display, labeling the cubes with letters, highlighting cubes to indicate that they are part of a word, and displaying words found by each player. Read the interface file (in the starter folder) to learn how to use the exported functions. The implementation is provided to you in source form so you can extend this code in your own novel ways.

Solution strategies

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 the following five phases:

- ***Task 1—Dice reading, board drawing, dice shaking.*** Design your data structure for the dice and board. It will help to group related data into sensible structures rather than pass a dozen parameters around. You should not use any global variables in this program. Read the dice file and store the dice. Create your shuffling routine. Use the **gboggle** routines to draw the starting board. Add an option for the user to specify a particular board configuration. Note that gboggle isn't object-oriented like previous graphics libraries. That's simply because the interface was defined before we introduced the **GWindow** class, and we didn't want to change the interface just because the implementation changed.It's been working too well as is.
- ***Task 2—User's turn (except for finding words on the board).*** Write the loop that allows the user to enter her words. Reject words that have already been entered or that don't meet the minimum word length or that aren't in the lexicon. Do not assume there is any upper limit on the number of words that may be found by the user. Put the **gboggle** functions to work for you adding words to the graphical display and keeping score. At this point, the words the user enters may or may not be possible to form on the board, that's comingup next.
- ***Task 3—Find a given word on the board.*** Now you will go to test your recursive talents in verifying that the user's words can actually be formed from the board. Remember that a valid word must obey the neighbor and non-duplication rules. You should search the board recursively, trying to find a legal formation of the user's word. This recursion is what you might call a "fail-fast" recursion, as soon as you realize you can't form the word starting at a position, you need to move on to the next position. Reject any word that cannot be formed from the letters currently on the board. Use the highlighting function from **gboggle** to temporarily draw attention to the letters in the word once you have verified it can be formed on the board.
- ***Task 4—Find all the words on the board (computer's turn).*** Now it's time to implement the killer computer player. Your computer player will make mincemeat of the paltry human player by traversing the board and finding every word the user missed. This recursion is an exhaustive search, so you will completely explore all positions on the board hunting for possible words. This phase is where the most difficult applications of recursion come into play. It is easy to get lost in the recursive decomposition and you should think carefully about how to proceed. Be sure to use the **Lexicon's** prefix search

to abandon searches down dead-end paths.

Requirements and suggestions

Here are a few details about our expectations for your solutions:

- Words should be considered case insensitively: **PEACE** is the same as **peace**.
- 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. They are somewhat similar, and you may be tempted to try to integrate the two into one combined function. In general, we applaud this instinct to unify similar code. However, we need to tell you that it doesn't play out well in this one case. There are enough differences between the two that they don't combine cleanly and the unified code is actually made worse, not better, as a result. Given the tricky nature of recursion, you should focus on writing exceptionally clean code that clearly communicates its algorithm and thus can be easily maintained and debugged. An important goal for this assignment is that you to learn how to employ these two different varieties of recursion into the context of a larger program, and we expect you to do so by writing two separate recursive searches.
- Our solution is about 250 lines of code (not counting comments) and is decomposed into about 30 functions.
- This is the first large program you're required to write in the course (we'll say that the Life assignment is relatively medium in size). Getting to a working solution is an excellent indication that you're on top of the technical challenges. This program is also an opportunity to demonstrate your commitment to good style. Your program should show thoughtful choices, be cleanly decomposed, be easily readable, and contain appropriate comments. If you view style only as an afterthought—a rearrangement after the fact to clean things up—you'll have missed a huge part of the benefit, which is that paying careful attention to design from the beginning results in code that is faster to write, is more likely be functionally correct, is easier to debug and modify, and requires far fewer comments.

A little more challenge: fun extras and extension ideas

As with most assignments, Boggle has many opportunities for extension. The following list may give you some ideas but is in no sense definitive. Use your imagination!

- ***Make the Q a useful letter.*** Because the **Q** is largely useless unless it is adjacent to a U, the commercial version of Boggle prints **Qu** together on a single face of the cube. You get to 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.
- ***Embellish the program with better graphics.*** The current game merely highlights the word; the words might be clearer if it also drew lines or arrows marking the connections.
- ***Use the mouse to trace the word on the board.*** The extended graphics library allows you to read the location of the mouse and determine whether the button is pressed. You can
- use these functions to allow the user to assemble a word by clicking or dragging through the letter cubes.

Accessing files

On the class web site, there is a zipped file which contains the following files:

boggle.cpp	Source file for your Boggle game implementation.
gboggle.cpp	Source file which implements the **gboggle** interface.
gboggle.h	Interface file for the **gboggle** module.
cubes16.txt	Data file containing the boggle cubes for a 4 x 4 board.
cubes25.txt	Data file containing the boggle cubes for a 5 x 5 board. (This one is completely optional.)
EnglishWords.txt	Data file containing a large word list for the lexicon in text format.
res	Folder of resource which contains cubes file, and EnglishWords.txt
BoggleDemo	Folder of working program that illustrates how the game is played.