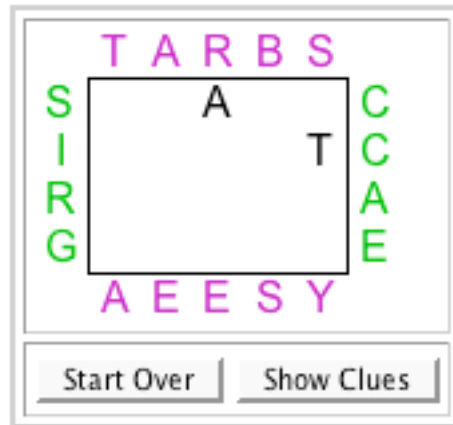


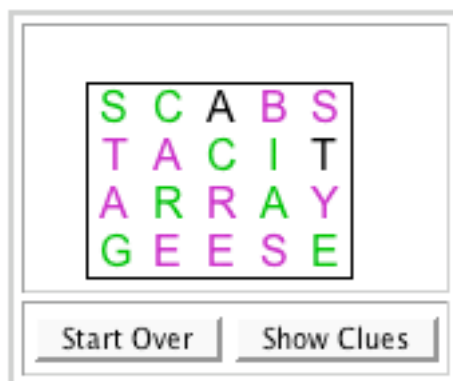
Section Handout

Problem 1: Creating Word Wreck Tangles

Several summers ago, I discovered a dashboard element for my Google Home Page that served up a new type of puzzle every week. One of these puzzles was the Word Wreck Tangle, and it looked like this:



The goal of the game is to slide each of the 18 letters around the perimeter into the board in such a way that you're left with a grid of nine overlapping words—four across and five down. Here's the solution:



Here's what I want to know! How do people find these boards in the first place? Are some people so inventive that they can just see that **scabs** stacked on top of **tacit** on top of **array** and **geese** just so happens to give you five four-letter words as well? Is it really that obvious?

Perhaps to some, but it needn't be, because we have recursive backtracking and the programmatic ability to generate a word wreck tangle, see what it gives us, and then **pretend** we didn't use a computer but just found it our own.

Your job here is actually a pretty tough one: write a recursive procedure that takes an empty **Grid<char>** (actually, it's a **Grid<char>** of space characters), and populates it with an assortment of letters that just happens to be four words across and five words down. Here's the prototype:

```
bool CreateWreckTangle(Grid<char>& board, Lexicon& lex);
```

If the top-level call to **CreateWreckTangle** returns true, then we can safely assume that the **Grid<char>** referenced by **board** was seeded with a solution that could be the goal board of the puzzle. There's no real reason to assume the grid is always 4 by 5, because it's trivial to frame the implementation in terms of whatever the grid dimensions happen to be. Here's what my solution generated (the word **stag** is big today):

v	a	s	a	l
i	x	t	l	e
t	e	a	s	e
a	l	g	o	r

You can assume you have access to:

```
void AddPermutationOfAlphabet(Vector<char>& letters);
```

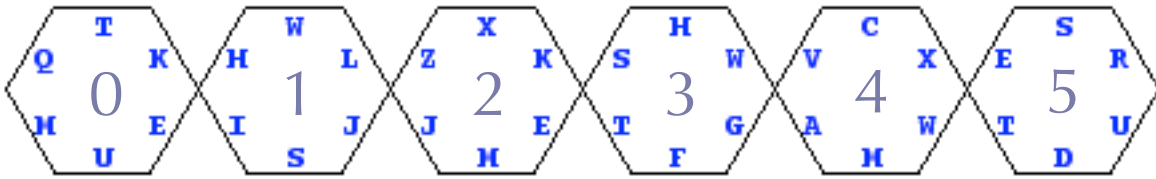
It takes the specified **Vector<char> &** and populates it with one of the 26! permutations of the alphabet. (Without it, your code would generate the same wreck tangle every time.)

As it turns out, generating a board is often a very time consuming process. What could you do to speed up the search? In other words, what heuristics beyond the use of **containsPrefix** could you do to reduce the branching factor and/or identify bad decisions more quickly?

Problem 2: Beehive Puzzle

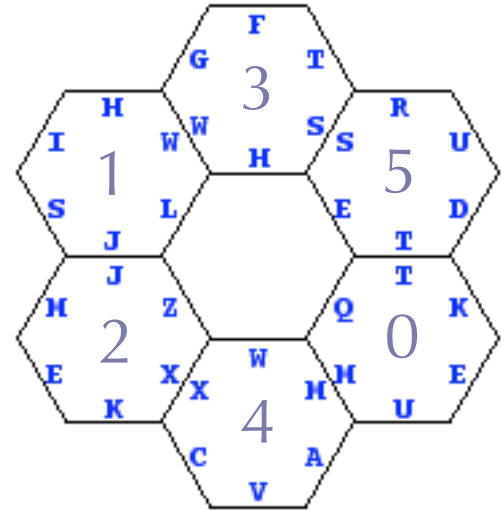
You're given six hexagonal games pieces, where each piece has a series of six letters—one per edge—around its perimeter. You're interested in finding an arrangement of the six pieces so that all six wrap around a hollow center, and all letters on adjacent edges match.

So, if you're given the following **vector** of six pieces:



you should be able to permute and rotate as necessary in order to find something like this:

Write a function called **ArrangementExists**, which takes a **vector** of six **strings**, where each **string** consists of the six characters along the perimeter of one hexagonal piece. The order of the letters dictates the clockwise ordering along the perimeter of the piece, but the piece may be rotated by any multiple of 60 degrees if it'll help to produce a solution. **ArrangementExists** reports back a **true** or a **false**—**true** if and only if some solution could be found. You don't need to return what the solution is; you only need to return a yes or a no.



```
bool ArrangementExists(Vector<string>& pieces);
```

Problem 3: Revisiting SuDoKu

The following code block was presented in class as a way of populating a SoDoKu board with an assignment that legitimately solves the puzzle. The details of the helper functions aren't important here, as you can intuit what they do based on how and where they're called.

```
static const int kUnassigned = 0;
bool SolveSudoku(Grid<int> &grid)
{
    int row, col;

    if (!FindUnassignedLocation(grid, row, col))
        return true; // success base case

    for (int num = 1; num <= 9; num++) { // consider all digits 1 to 9
        if (NoConflicts(grid, row, col, num)) { // if looks promising,
            grid[row][col] = num; // make tentative assignment
            if (SolveSudoku(grid)) return true; // recur from here: if success, yay!
            grid[row][col] = kUnassigned; // failure, unmake & try again
        }
    }

    return false; // trigger backtracking, woo
}
```

One problem of interest that the above version doesn't quite solve is whether or not the solution to the problem is **unique**. Properly constructed SuDoKu boards are such that there's only one way to solve the puzzle—not zero, not two or more—just one.

Leveraging off of the above implementation, write a function called **HasUniqueSolution**, which calls a modified version of **SolveSudoku** that returns true if and only if there's exactly one way to assign numbers to the open squares to solve the puzzle.

```
bool HasUniqueSolution(Grid<int>& board);
```

Problem 4: Minimizing Quicksort Recursion

Starting with your reader's implementation of the quicksort algorithm, modify the implementation so that at most $O(\lg n)$ recursive calls in the worst case.

Here's the reader's version of Quicksort:

```
void Quicksort(Vector<int>& vec, int start, int finish)
{
    if (start >= finish) return;
    int boundary = Partition(vec, start, finish);
    Quicksort(vec, start, boundary - 1);
    Quicksort(vec, boundary + 1, finish);
}
```