Requirements:

* Find one or all boards that meet these requirements
* Any sequence of two or more squares touching is considered a word
* Cannot contain words shorter than 3 letters and all words must be from provided dictionary
* Only across (left to right) or down (top to bottom)
* Letter in a square can belong to either across or down or both
* Letter in a square is never part of 2+ across or 2+ down at the same time
* Dictionary file only has upper case words (one word per line) - all words in solution must be in dictionary file
* Words in dictionary are unique and words only appear one time per solution
* Use recursion in a non-trivial way
* Do some error checking when reading in input file
* Analyze performance of your algorithm using order notation (variables are dimensions of the board (w and h), number of words in dictionary (d), total number of spaces with a letter (l), total number of blacked out spaces (b), number of constraints (c), etc.) 🡪 write a concise paragraph (< 200 words) describing your answer and a simple table summarizing the running time and the number of solutions found by your program on each of the provided examples
* In your README, also include 1-3 new puzzles and at least 1 dictionary for each of your new puzzles that either helped you test corner cases or experiment with the running time of your program and describe these puzzles in your README
* Techniques covered in Lectures 1-14, HW 1-5, Labs 1-7 allowed (can’t use STL pairs, maps, set, etc.)

Arguments

* 4 baseline and 5th one for extra credit
* ./a.out [dictionary file] [initial grid file] [solution mode] [output mode] [gc]

Input file

* Comment lines start with ! and should be ignored by program
* Constraint lines start with + and are followed by a positive integer
* Lines that are neither comment nor constraint lines are one row of the puzzle’s input grid and are presented in order
* Every puzzle has one or more constraints representing a required word length in the solution
* Any legal solution must have one matching word per constraint
* Every constraint must have a matching word
* If +4 appears twice in the input file, all legal solutions must have exactly 2 four letter words

Solution mode

* Either one solution (one\_solution) or all solutions (all\_solutions) mode

Output mode

* Either count\_only (only print number of solutions found) or print\_boards (print count and all solutions and solutions may be printed in any order)
* Example of print\_boards output formatting:

Number of solution(s): 3

Board:

B#F

L#L

USE

E#E

S#S

Board:

B#F

L#L

U#E

EKE

S#S

Board:

B#F

L#L

U#E

E#E

SIS

Extra credit:

* If fifth argument gc given, then you should print only boards that are a giant component (this means that you should be able to use a series of up/down/right/left moves to reach all other letters in the board without having to go through a blacked out square)
* Examples:

#######

#LABRAT

R######

E######

D######

Legal for baseline HW 6 but not legal for extra credit

#######

#LABRAT

####E##

####D##

#######

Legal for baseline HW 6 and legal for extra credit

Pseudocode:

1. Parse dictionary file

* Push back to vector<string> words (should be lemon, melon)

1. Parse input file (example: reverse\_ell1.txt)

* Ignore lines with !
* Identify what the word constraints are
  + Push back to vector<int> constraints (should be 5, 5)
  + Read in grid to vector<vector<char>>> board

1. Start recursive crossword puzzle solver

* Brute force approach
  + Simplest case (constraints match up with dictionary - e.g. +5 +5 and lemon, melon in dictionary)
    - Start with first word (lemon)
    - Search for first l
      * Run left to right search and then top to bottom search
      * Keep recursively searching right (for left to right) and then bottom (for top to bottom) until word size = 5 (for lemon) and lemon is found OR one of the letters don’t match (e.g. third letter should be m but is a w - terminate recursive call in that case) OR you walk off board (terminate recursive call in that case)
      * Push back successful recursive call to a path to track the word where it is found to vector<vector<Point>> where Point class will have x, y
    - Continue this approach for the rest of the words in the dictionary
  + If constraints are variable, use approach of combinations to try to search for all possible solutions
    - vector<vector<string>> searchCombinations
    - We do this by having a nested set of for loops:
      * for (unsigned int i = 0; i < n; i++) {

for (unsigned int j = i + 1; j < n; j++) {

// [i][j] is one instance of a combination

}

1. Go back and change the characters in the grid not in the word paths by using all the word paths recursively by iterating through vector<vector<Point>> (nested loops scenario - 1 for grid and 2 for word path for 2D vector) and output solutions

For now, vectors can be used for everything. If we are going to do a lot of insert or erase operations, we will consider using lists instead.

Class design:

// note that the origin (0, 0) will be at the top left corner

class Point {

public:

unsigned int x, y;

}

Debugging techniques:

Invariants (for recursive calls and iterative versions [i.e. loops]) - use assert statements - what should be true at beginning and/or end of each loop/recursive call?

Well place cout statements