Middle East Technical University

Department of Computer Engineering

CENG 111

Fall 2017

Take Home Exam

REGULATIONS

Due date: 23:59, 26 December 2017, Tuesday *(Not subject to postpone)*

Submission: Electronically. You will be submitting your program source code through a text file which you will name as `the3.py` by means of the COW system.

Team: There is **no** teaming up. This is an EXAM.

Cheating: Source(s) and Receiver(s) will receive zero and be subject to disciplinary action.

PROBLEM

In THE-3, your task is to find the placement of a subset-set $\mathbb{W}$ of N-letter words in a corpus $\mathbb{C}$, on a grid of size $N \times N$ such that the vertical readings (from top to bottom) and horizontal readings (from left to right) of letters are also words in $\mathbb{C}$. For example, assume that $\mathbb{C}$ consists of the following seven words:

`ALI, SIN, ASI, LIR, IRI, INI, KAR,`

and the grid is 3×3 (note that each row and each column can hold only one word). One possible placement of these words on a grid of 3×3 is as follows:

	column-1	column-2	column-3
row-1	A	L	I
row-2	S	I	N
row-3	I	R	I

In this placement, all the words that can be constructed from consecutive letters in a row and in a column are valid (they do not have to be meaningful) words in $\mathbb{C}$. In other words, we made use of the words `ALI` (row-1), `SIN` (row-2), `IRI` (row-3), `ASI` (column-1), `LIR` (column-2), `INI` (column-3) which are members of the set $\mathbb{C}$. Note that this placement did not make use of all the words in $\mathbb{C}$; namely, `KAR` is not a part of the solution.

SPECIFICATIONS

- You should write a function `place_words(Corpus)` which takes the corpus as a list of strings.
- You can assume that all words in the corpus have the same size, and that this size is equal to N, the size of the grid (N can be different from 3).
- If there is a solution, your function should return it as a list of words such that the i^{th} word in this list is the i^{th} row in the grid.
- If there is no solution, your program should just return `False`.
- Words can only consist of letters from the English alphabet, and your solution should be case-insensitive; i.e., the two words `CAN` and `Can` should be treated as the same words.
- Your solutions should be in uppercase no matter in which case the words in the corpus are.
- We will not test your solution with erroneous input. However, we will test your program with words which may not have a valid placement on a grid!

Example Run

```
>>> Solution = place_words(["ALI", "SIN", "ASI", "LIR", "IRI", "INI", "KAR"])
>>> print Solution
["ALI", "SIN", "IRI"]
```

Notes

- You cannot use any other method than backtracking (see Appendix for a quick intro).
- You may use recursion or iteration.
- A word may appear only once in the grid (horizontally or vertically).
- Your function will be tested with multiple data.
- Any program that performs below 30% of the total grade will enter a glass-box test (eye inspection by the grader TA). The TA will judge an overall grade in the range of [0,30].
- The glass-box test grade is not open to negotiation, discussion or explanation.
- Your function will be tested with Python interpreter (v2.7) that is installed on *inek* machines running Linux.
- You are encouraged to share input-outputs on the news group of the course.

APPENDIX: INTRODUCTION TO BACKTRACKING

Backtracking is a widely-used method in Computer Science for finding a solution to a problem incrementally. The current (partial) solution is usually kept as an ordered set $V = v_1, .., v_N$ and at each step, a new partial-solution-element v_i is added to V in such a way that the addition of v_i to V takes us one step closer to the complete solution and that v_i is not in conflict with V as far as the problem is concerned.

If $V + v_i$ is invalid, another element, if there is any, is placed in the position of v_i. If no v can be found that would make $V + v$ valid, the algorithm *backtracks*; in other words, it goes back to the previous step and tries another element v; in our scenario, this would mean removing v_{LAST} and placing some other v in v_{LAST}'s place that would not invalidate V.

The process of (i) adding a valid new partial solution, and (ii) backtracking to find a valid new partial solution is repeated until a solution is found or all the possible options are exhausted, which means that no valid solution exists for the problem.

A good example of backtracking is the n-queens problem; i.e., the placement of n queens on a $n \times n$-chessboard. We will illustrate backtracking on the 4-queens problem (Figure 1).

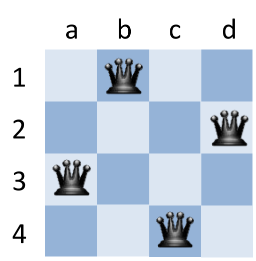

Figure 1: 4-queens problem.

The backtracking-solution to the 4-queens problem is described in Algorithm 1. Note that the solution in Algorithm 1 presents just the backbone of the solution, and the functions `get_next_empty_row()`, `get_next_queen()`, `backtrack()` and `is_valid()` need to be filled in. Another important point is that you need to keep a track of the solutions that have been tried for each row, and when you backtrack from a row, you need to erase the options that have been tried for that row.

The backtracking solution to the 4-queens problem is displayed step-by-step in Figure 2.

Algorithm 1 Pseudo code for the solution to 4-queens problem.

```
Set V to {}
while True do
  q = get_next_queen()
  if q is INVALID then
    /* WE FINISHED ALL QUEENS => SOLUTION FOUND */
    Return V as solution and exit
  end if
  (row, column) = get_next_empty_row()
  /* ''Empty row'' means no queen is at (row, X) nor at (Y, column) for any X,Y nor in a diagonal with another queen */
  if row == INVALID or column == INVALID then
    /* No valid & empty (row, column) */
    backtrack() /* If there is no option to backtrack to, return NO-SOLUTION */
  else
    Set V = V + {q at (row, column)}
  end if
end while
```

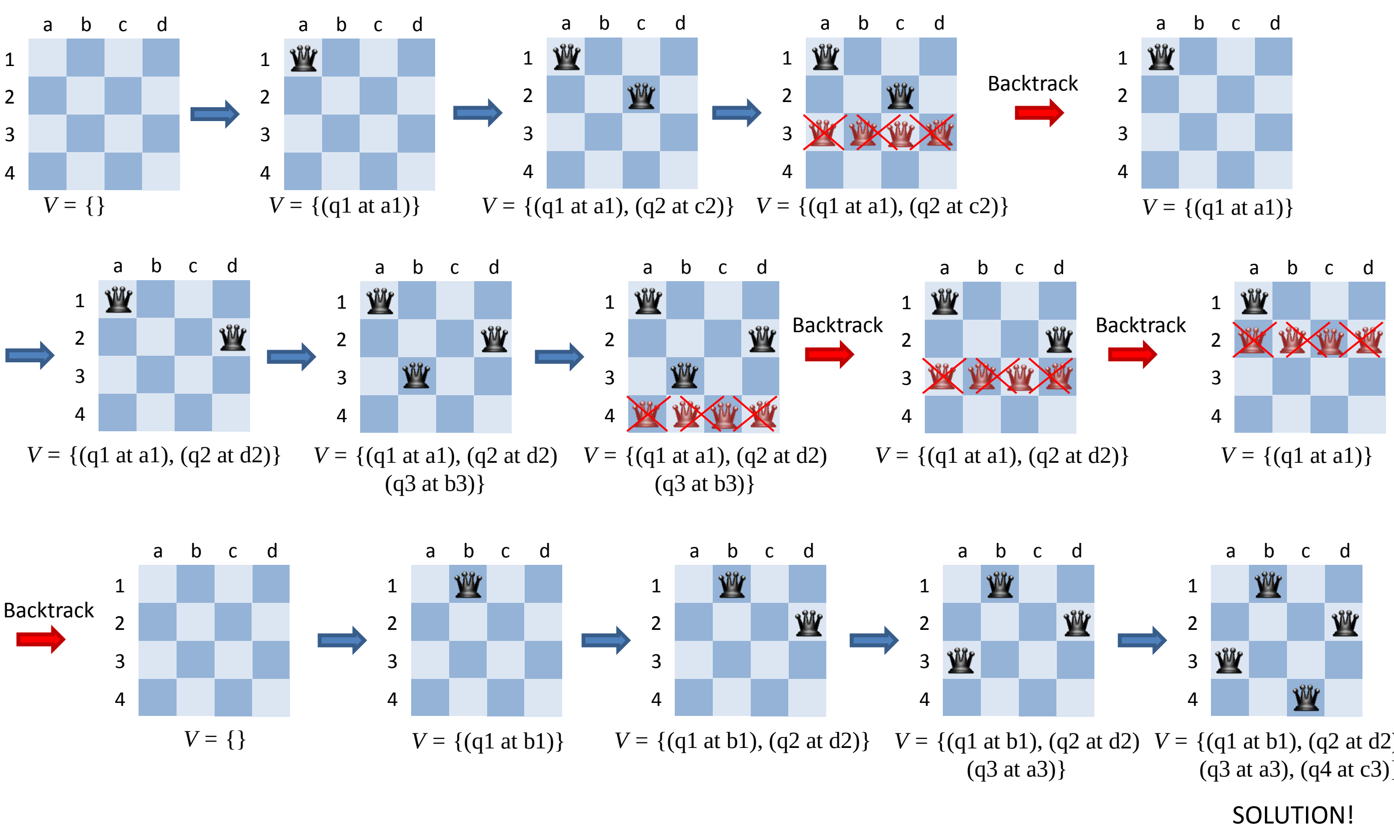

Figure 2: Step-by-step solution to the 4-queens problem.