THE CRC ROBOTICS
SENIOR PROGRAMMING PROBLEMS

PRELIMINARY 2

MODUEL
2026

presented by

FTAI AVIATION

A program of
AEST EAST

Version 1.0

A FEW NOTES

- The complete rules are in section 4 of the rulebook.
- You have until **Friday, December 5th, 11:59 pm** to submit your code.
- Feel free to use the programming forum on the CRC discord to ask questions and discuss the problem with other teams. It is there for that purpose!
- **We are giving you quick and easy-to-use template files for your code and the tests. You are required to use them.**

USING THE TEMPLATE FILE

- The template tests call the function to test, take the output and allow you to quickly check if your code works as intended. **All your code, except additional functions you create, should be written in the function of the part of the problem you are solving.**
- Points given in the document are indications of how difficult the section is and how many points you will get if you complete it. This preliminary problem is going to be 2% of the main challenge towards the global score of the programming competition and for more points related information consult the rulebook.

STRUCTURE

Every problem contains a small introduction like this about the basics of the problem and what is required to solve it.

<u>Input and output specification:</u>

Contains the inputs and their format, and which outputs the code is required to produce and in what format they shall be.

<u>Sample input and output:</u>

Contains a sample input, sometimes containing sub examples in the sample input, and what your program should return as an output.

<u>Explanation of the first output:</u>

Explains briefly the logic that was used to reach the first output, given the first input.

Board Games!

On rainy weekends, late nights, or even mid-day, board games with friends is always a good time. So many friendships are built around board games and everyone has their favorite. Today, we'll explore various board games that are dear to CRC organizers.

Part 1: Snakes and ladders (20 points)

Snakes and ladders is a game with 100 squares with the goal to get to square 100 while rolling a die, traditionally a 6 sided die. With ladders making you go up and snakes making you go down.

You will start outside of the board, so if on the first try you move 6 squares you will be on 6, and you need to get to square 100 in the least amount of throws to reach it.

Input and output specification:

You will receive as input an *int* for the amount of sides for the die. A list of int tuples for the ladders and the snakes in ascending order or the start of it. The ladders are from the bottom to the top and the snakes are from the top to the bottom.

You will return an int of the minimum amount of dice throw to get to the top

Sample input:

d = 6
Ladder : {(1, 38), (4, 14), (21, 42), (28, 84)}
Snakes : {(48, 26), (49,11), (62, 19), (87, 24)}

d = 8
Ladder : {(7, 50)}
Snakes : {(80, 30)}

d = 10
Ladder : {(4, 9), (41, 99)}
Snakes : {(31, 5),(50, 20)}

Sample output:

7
8
6

Explanation of the first output:

With a 6 sided die the most we can move is 6 squares. The longest ladder is (28,84), to get to that ladder, we have to skip the first ladder and go to (4,14) in one roll. We also cannot use the ladder (21,42) we have to roll to go to 28, with 3 rolls. After going up the ladder, we have to roll 3 times to get to the end. Which gives us 7 rolls.

Part 2: Scrabble (40 points)

The game of Scrabble consists of placing letters on a board in order to form words and score points. These words can be placed orthogonally either up, down, right, or left. A letter placed on the board can be shared by two words, one horizontal and one vertical. The aim of the game is to place a word so that it intersects with as many words as possible already on the board.

Input and output specification:

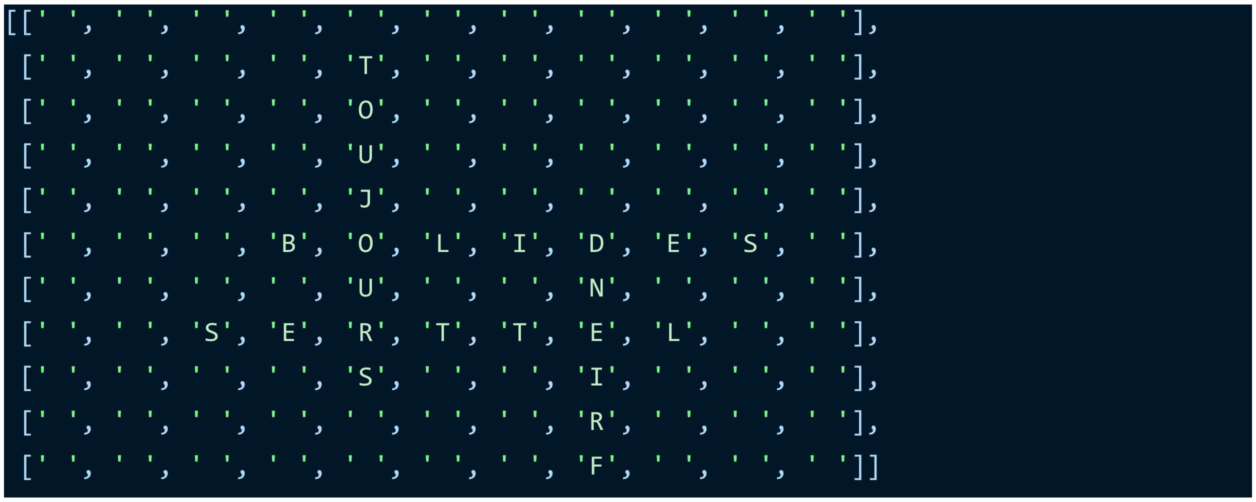
For input, you will receive a *list* containing several *lists* of *strings* which correspond to the scrabble grid's and a *string* which is the word you need to place on the board game.
For output, you need to output a list containing lists of strings which are the scrabble grid after you place the word .

Sample input:

```
[[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'T', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'O', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'U', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'J', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'O', ' ', ' ', 'D', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'U', ' ', ' ', 'N', ' ', ' ', ' '],
 [' ', ' ', 'S', 'E', 'R', 'T', 'T', 'E', 'L', ' ', ' '],
 [' ', ' ', ' ', ' ', 'S', ' ', ' ', 'I', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', 'R', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', 'F', ' ', ' ', ' ']]
```

"BOLIDE"

Sample output:

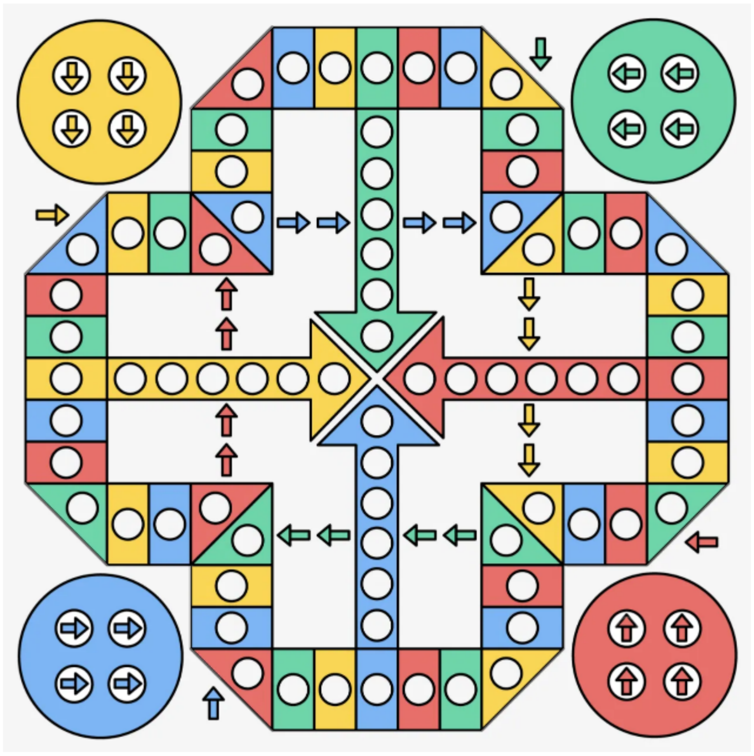


```
[[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'T', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'O', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'U', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', 'J', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', 'B', 'O', 'L', 'I', 'D', 'E', 'S', ' '],
 [' ', ' ', ' ', ' ', 'U', ' ', ' ', 'N', ' ', ' ', ' '],
 [' ', ' ', 'S', 'E', 'R', 'T', 'T', 'E', 'L', ' ', ' '],
 [' ', ' ', ' ', ' ', 'S', ' ', ' ', 'I', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', 'R', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', 'F', ' ', ' ', ' ']]
```

Explanation of the first output:

The word “BOLIDE” is placed by sharing its O with ‘Toujours’ and its D with “FRIEND,” because that is where it shares the most letters with other words already on the board.

Part 3: Aeroplane Chess (85 points)



The Aeroplane Chess game is a popular board game, played with 2 to 4 players. Each player (referred to as a "team" later in this problem) has a color (Red, Blue, Yellow, or Green), which will be used to identify the team later, and 4 planes. The objective of the game is to be the first team to get all 4 of their planes moved ("flown") from the starting area ("hanger") on the corner of the board to the destination (the center of the board).

An image of the board for the *original* aeroplane chess game is on the left.

Other than "team", "plane", "hanger", here are some other important terms you need to know:

- The "main path" is the circular path around the board that all teams' planes must traverse, the length of which is 52 spaces, and all planes MUST follow this path in ***clockwise*** direction.
- The "final path" is the colored path that leads from the main path to the destination (center) for each team. Each final path is 6 spaces long and has the shape of a large arrow pointing to the center. Once the plane arrives at the space that is on the main path right before the final path, it MUST enter the final path on the next step. The plane is NOT allowed to go more steps than it needs to arrive at the destination (i.e. it can not overshoot).
- The "launch area" of a team is the space on the main path with an arrow of the same color as the team pointing into the space. This is where the plane enters the "main path" from the "hanger".
    - To avoid confusion, on the image, the launch areas all have right-triangular shapes and they are NOT colored according to their respective team colors.
- The "airway" is the paired ***colored*** spaces on the main path with directional arrows (of the same color) connecting between them. A plane that stops at the space with an arrow pointing to the other space will be immediately moved to the paired space, and it does NOT count as a turn.
- One "turn" is the process of a plane moving from one space to another space, which may involve the use of gadgets.

You are now playing a special version of this game **with only *one* plane per team**. You will be given a list containing 52 strings representing all the spaces on the main path, starting with the end of the Red team's final path and going clockwise around the board. Each string will have the following format:

```
[L<COLOR>]<COLOR>#<STEP>#[G<F/C/T>][<E/T>][F]
```

where:

- ***[L<COLOR>]*** may or may not be present. If it is present, it indicates that this space is the launch area space of the airway for the team with respective color. That is: ***[LR]*** indicates that this space is the launch area space of the Red team on the main path, and so on for ***[LB]***, ***[LY]***, and ***[LG]***.
- ***<COLOR>*** is one of 'R', 'B', 'Y', or 'G', representing Red, Blue, Yellow, or Green respectively.
- ***<STEP>*** is an integer between 1 and 6 (inclusive), representing the number of steps the plane on it MUST move forward in the next turn. However, if the plane is on a same-colored space (i.e. the color of the space is the same as the color of the plane), the plane MAY choose to move any number of steps between 1 and ***<STEP>*** (inclusive) instead.
- ***[G<F/C/T>]*** is a two-character code that may or may not be present. If it is present, it indicates a gadget that may affect the plane on this space.
    - ***[GF]*** is a "gadget: fan" that allows the plane to move forward an additional 3 steps on its next turn (if the plane is not on a same-colored space), or up-to an additional 3 steps on its next turn (if the plane is on a same-colored space). So, as an example, you are on a `B#6#[GF]` and you choose to use the gadget. If you are red, you must advance 9 spaces, but if you are blue, you can advance anywhere from 1 to 9 spaces in the following turn. The same logic applies to every gadget.
    - ***[GC]*** is a "gadget: core" that doubles the number of steps the plane can move on its next turn (if the plane is not on a same-colored space), or doubles the maximum number of steps the plane can move on its next turn (if the plane is on a same-colored space).
    - ***[GT]*** is a "gadget: turbine" that has the same effect as ***[GC]***, but instead of doubling, it triples the number of steps (or maximum number of steps) the plane can move on its next turn.
    - The team MAY choose not to use the gadget, but an unused gadget will NOT be kept for future turns.
- ***[E/T]*** may or may not be present. If it is present, it indicates that this space is part of an "airway" for the team with color ***<COLOR>***. The 'E' indicates the entry space of the airway, while the 'T' indicates the target space of the airway. If a plane of respective color lands on the ***[E]*** space, it will be immediately moved to the paired ***[T]*** space.

- ➢ ***[F]*** may or may not be present. If it is present, it indicates that this space is at the beginning of the final path for the team with color ***<COLOR>***.

An example of a space string is:

[LB]R#4#[GC][T]

which indicates that this space is:

- ➢ the launch area of the airway for the Blue team,
- ➢ red-colored,
- ➢ allows a plane on it to move (up to) 4 steps forward,
- ➢ has a gadget "core" that doubles the number of steps the plane can move on its next turn,
- ➢ is the target space of the airway for the Red team.

You will also be given four 6-string long lists representing the final path spaces for each team, in the order of Red, Blue, Yellow, and Green. The formatting is the same except that there are no ***[<E/T>][F]*** at the end since no final path space is part of an airway or the beginning of a final path.

You will also be given a color that you are playing as (i.e. your team color). To maximize your chance of winning, you will have to explore the shortest possible path for your plane to reach the destination, taking into account all the gadgets and special spaces on the board. As output, show the minimum number of turns required for your plane to reach the destination.

Notes:

- ➢ The color of the spaces will strictly follow the original aeroplane chess board layout (i.e. start with 'G', then 'R', 'B, 'Y', and repeat).
- ➢ The airway space for a specific team will always have the entry at 17 spaces after the launch area space of the respective color, and the target at 29 spaces after the launch area space of the respective color.
- ➢ The space at the beginning of the final path for each team will always be the 49th space after the launch area space of the respective color.
- ➢ There are only 1 launch space, 1 airway entry space, 1 airway target space, and 1 final path entry space for each team on the main path.
- ➢ These are strictly followed by the input data and are strictly following the design of the original aeroplane chess game board. These labels are also provided in the input data for your convenience in determining the colors and positions of such special spaces.

Input and output specification:

- ➔ You will first receive *an array of 52 strings* representing the main path spaces, starting from the end of the Green team's final path and going clockwise around the board.
- ➔ Then, you will receive *four arrays of 6 strings each*, representing the final path spaces for each team, in the order of Red, Blue, Yellow, and Green.
- ➔ Finally, you will receive *a single character* representing your team color ('R', 'B', 'Y', or 'G').
- ➔ You will need to output *a single integer* representing the minimum number of turns required for your plane to reach the destination.

Sample input:

Sample Input 1

```
["G#4#[F]", "R#1#", "B#1#", "[LG]Y#4#", "G#1#", "R#4#", "B#6#[T]",
"Y#5#[E]", "G#2#", "R#2#", "B#2#", "Y#4#", "G#4#", "R#1#[GF][F]",
"B#2#", "Y#2#", "[LR]G#3#", "R#5#", "B#6#[GC]", "Y#2#[T]", "G#2#[E]",
"R#6#", "B#3#", "Y#4#", "G#2#", "R#5#", "B#2#[F]", "Y#4#", "G#4#",
"[LB]R#3#", "B#1#", "Y#3#", "G#4#[T]", "R#3#[E]", "B#1#", "Y#4#",
"G#4#[GF]", "R#3#", "B#2#", "Y#4#[F]", "G#2#[GC]", "R#6#",
"[LY]B#4#", "Y#1#", "G#4#", "R#6#[T]", "B#5#[GC][E]", "Y#4#", "G#3#",
"R#5#", "B#2#", "Y#6#"]
["R#3#[GC]", "R#4#", "R#3#", "R#6#", "R#6#", "R#5#"]
["B#6#", "B#3#", "B#4#", "B#5#", "B#6#", "B#1#"]
["Y#4#", "Y#5#", "Y#3#", "Y#4#", "Y#6#", "Y#6#"]
["G#3#", "G#4#", "G#2#", "G#2#", "G#3#", "G#2#"]
'B'
```

Sample Input 2

```
["G#6#[F]", "R#6#[GF]", "B#2#[GT]", "[LG]Y#6#", "G#5#[GF]",
"R#1#[GC]", "B#6#[GC][T]", "Y#3#[E]", "G#6#[GF]", "R#4#", "B#5#",
"Y#3#", "G#6#[GT]", "R#1#[F]", "B#2#", "Y#4#", "[LR]G#5#", "R#6#",
"B#6#", "Y#1#[T]", "G#4#[GF][E]", "R#2#", "B#2#", "Y#4#[GF]", "G#5#",
"R#4#", "B#5#[GC][F]", "Y#2#", "G#2#", "[LB]R#6#", "B#6#",
"Y#3#[GF]", "G#3#[T]", "R#3#[E]", "B#4#", "Y#1#", "G#6#", "R#1#",
"B#2#", "Y#3#[F]", "G#6#", "R#2#", "[LY]B#6#[GC]", "Y#4#", "G#6#",
"R#6#[T]", "B#2#[E]", "Y#4#[GF]", "G#4#", "R#2#", "B#4#", "Y#2#"]
["R#3#", "R#5#", "R#2#[GC]", "R#3#", "R#4#[GF]", "R#3#[GF]"]
["B#3#[GC]", "B#6#", "B#1#", "B#5#[GF]", "B#6#", "B#6#"]
["Y#4#", "Y#5#", "Y#4#[GF]", "Y#1#", "Y#1#", "Y#6#"]
["G#6#", "G#1#", "G#5#[GF]", "G#2#", "G#4#", "G#2#"]
'R'
```

Sample Input 3

```
["G#4#[F]", "R#2#", "B#4#", "[LG]Y#2#", "G#3#", "R#4#", "B#4#[T]",
"Y#6#[E]", "G#1#", "R#1#", "B#5#", "Y#6#", "G#1#[GF]", "R#6#[GF][F]",
"B#1#",  "Y#2#",  "[LR]G#4#",  "R#3#",  "B#2#",  "Y#1#[T]",  "G#3#[E]",
"R#1#[GC]",  "B#2#",  "Y#3#",  "G#3#",  "R#6#",  "B#2#[F]",  "Y#6#",
"G#6#",  "[LB]R#6#",  "B#4#",  "Y#4#",  "G#3#[T]",  "R#4#[E]",  "B#1#",
"Y#1#",  "G#2#",  "R#6#",  "B#6#",  "Y#2#[F]",  "G#6#",  "R#5#",
"[LY]B#1#",  "Y#2#",  "G#5#",  "R#5#[T]",  "B#3#[E]",  "Y#3#",  "G#1#",
"R#5#", "B#2#", "Y#3#"]
["R#1#", "R#1#", "R#1#", "R#2#", "R#4#", "R#2#[GT]"]
["B#4#", "B#1#", "B#2#", "B#2#", "B#2#", "B#1#"]
["Y#3#[GF]", "Y#2#[GF]", "Y#6#[GC]", "Y#2#[GF]", "Y#5#", "Y#3#"]
["G#3#", "G#6#", "G#1#", "G#1#", "G#5#", "G#3#"]
'Y'
```

Sample output:

Sample Output 1
```
11
```

Sample Output 2
```
9
```

Sample Output 3
```
14
```

Explanation of the first output:

The blue team needs at least **11 turns** to arrive at the destination.

The path taken:

- ➢ Turn 1: From main path position 0 -> Move 3 steps to main path position 3
- ➢ Turn 2: From main path position 3 -> Move 4 steps to main path position 7
- ➢ Turn 3: From main path position 7 -> Move 4 steps to main path position 11
- ➢ Turn 4: From main path position 11 -> Move 2 steps to main path position 13
- ➢ Turn 5: From main path position 13 -> Move 4 steps to main path position 17
- ➢ **Turn 5: Automatically takes the airway to main path position 29**
- ➢ Turn 6: From main path position 29 -> Move 6 steps to main path position 35
- ➢ Turn 7: From main path position 35 -> Move 4 steps to main path position 39
- ➢ Turn 8: From main path position 39 -> Move 3 steps to main path position 42
- ➢ Turn 9: From main path position 42 -> Move 2 steps to main path position 44
- ➢ Turn 10: From main path position 44 -> Move 6 steps to final path position 0
- ➢ Turn 11: From final path position 0 -> Move 5 steps to DESTINATION

Part 4: Carcassonne (65 points)

In the game of Carcassonne based on the French town with the same name, each player picks a tile during their turn which they have to place on the board to expand the city and score points. Though, there are rules that need to be followed to place a tile: each side of the new tile has to match with the sides of the tiles surrounding it. In the base version of the game, there are 3 types of terrain a tile can have: grass, roads and castles.

The goal of your code will be to find and return a list of all the valid positions on a given board where a specific tile can be placed. To help you, we created 2 custom classes for the tiles and the board. Some functions were already made for you or we'd recommend you to complete. You can always add your own!

Input and output specification:

For your inputs, you'll receive a **Tile** object and a **Board** object already filled with tiles. A tile is represented by a string of four characters where each letter represents the type of terrain on a side of the tile. The characters always appear in the same order:

- the first one corresponds to the **top side**,
- the second one to the **right side**,
- the third one to the **bottom side**,
- and the fourth one to the **left side**.

You can access those values with the **.top**, **.right**, **.bottom** and **.left** properties. Then, each terrain has its own associated letter:

- **g** for grass,
- **r** for roads,
- **c** for castles,
- **n** to represent blank tiles.

A **Board** is simply a two dimensional list filled with **Tile** objects. Each square on the board can either be empty (a blank tile) or have a normal tile.

Sample input:

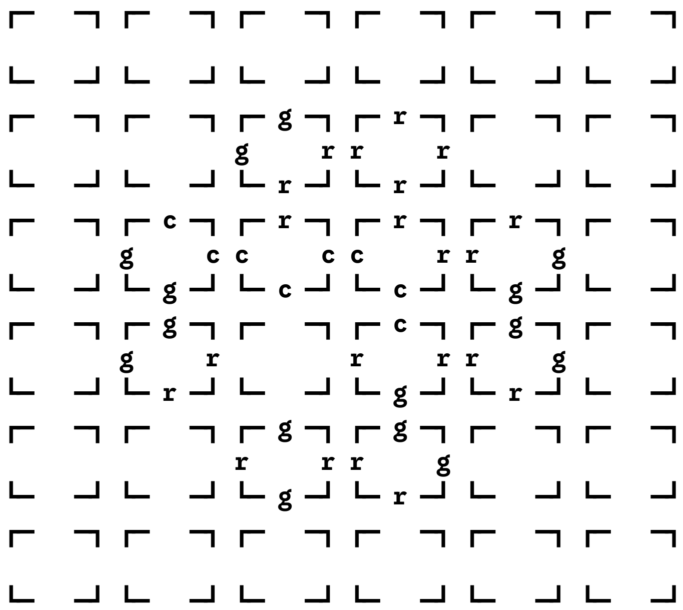
Example 1:
```
tile = Tile("ggrr")
board = [[“nnnn”, “nnnn”, “nnnn”],
         [“nnnn”, “crgr”, “nnnn”],
         [“nnnn”, “nnnn”, “nnnn”]]
```
Example 2:
```
tile = Tile("rgrr")
board = [[“nnnn”, “nnnn”, “nnnn”, “nnnn”],
         [“nnnn”, “crgr”, “ccrr”, “nnnn”],
         [“nnnn”, “nnnn”, “rgrr”, “nnnn”],
         [“nnnn”, “nnnn”, “nnnn”, “nnnn”]]
```
Example 3:
```
tile = Tile("crgr")
board = [[“nnnn”, “nnnn”, “nnnn”, “nnnn”, “nnnn”, “nnnn”],
         [“nnnn”, “nnnn”, “grrg”, “rrrr”, “nnnn”, “nnnn”],
         [“nnnn”, “ccgg”, “rccc”, “rrcc”, “rggr”, “nnnn”],
         [“nnnn”, “grrg”, “nnnn”, “crgr”, “ggrr”, “nnnn”],
         [“nnnn”, “nnnn”, “grgr”, “ggrr”, “nnnn”, “nnnn”],
         [“nnnn”, “nnnn”, “nnnn”, “nnnn”, “nnnn”, “nnnn”]]
```

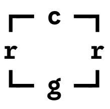
Sample output:

Exemple 1: [(1, 0), (1, 2), (2, 1)]
Exemple 2: [(1, 0), (2, 1), (2, 3), (3, 2)]
Exemple 3: [(0, 2), (0, 3), (2, 0), (2, 5), (3, 0), (3, 2), (3, 5), (4, 4), (5, 2), (5, 3)]

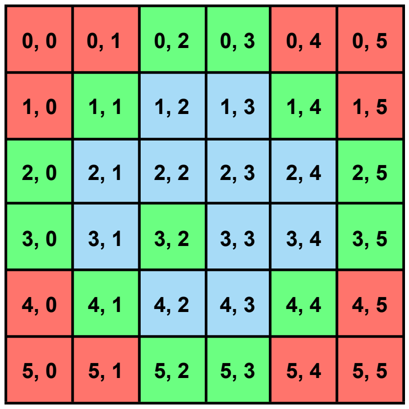
Explanation of the first output:

Here's a visual representation of the **Board** from the third example and the **Tile** to place.



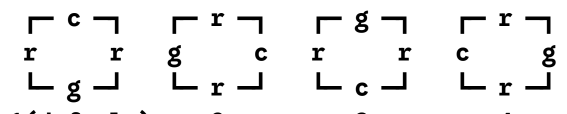


To figure out where our tile can fit, we'll go through the options by elimination. First, we have to find all the locations where any tile could be placed. The blue squares represent the tiles already on the board and the red ones are the squares with no adjacent tiles, which leaves us with the green squares where the tile could be placed.



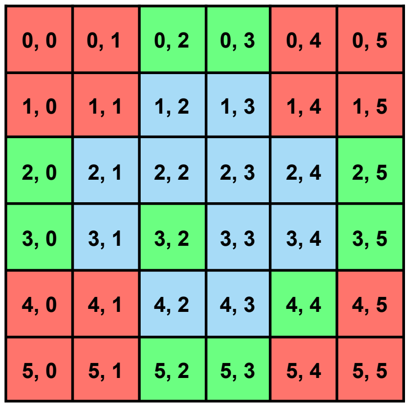
🟥 0, 0	🟥 0, 1	🟩 0, 2	🟩 0, 3	🟥 0, 4	🟥 0, 5
🟥 1, 0	🟩 1, 1	🟦 1, 2	🟦 1, 3	🟩 1, 4	🟥 1, 5
🟩 2, 0	🟦 2, 1	🟦 2, 2	🟦 2, 3	🟦 2, 4	🟩 2, 5
🟩 3, 0	🟦 3, 1	🟩 3, 2	🟦 3, 3	🟦 3, 4	🟩 3, 5
🟥 4, 0	🟩 4, 1	🟦 4, 2	🟦 4, 3	🟩 4, 4	🟥 4, 5
🟥 5, 0	🟥 5, 1	🟩 5, 2	🟩 5, 3	🟥 5, 4	🟥 5, 5

Next, we have to check if the given tile can be placed in each of those squares no matter its orientation. Here are the 4 orientations our given tile can have:



1(default) **2** **3** **4**

For the tile (0, 2), the first orientation is valid, so we append its coordinates to the list. For the tile (0, 3), the second and fourth orientation are valid, so we also append its coordinates to the list. But, for the tile (1, 1), there are no valid orientations, so we leave it out. We continue this process for each green square and we should have the final list to return.



🟥 0, 0	🟥 0, 1	🟩 0, 2	🟩 0, 3	🟥 0, 4	🟥 0, 5
🟥 1, 0	🟥 1, 1	🟦 1, 2	🟦 1, 3	🟥 1, 4	🟥 1, 5
🟩 2, 0	🟦 2, 1	🟦 2, 2	🟦 2, 3	🟦 2, 4	🟩 2, 5
🟩 3, 0	🟦 3, 1	🟩 3, 2	🟦 3, 3	🟦 3, 4	🟩 3, 5
🟥 4, 0	🟥 4, 1	🟦 4, 2	🟦 4, 3	🟩 4, 4	🟥 4, 5
🟥 5, 0	🟥 5, 1	🟩 5, 2	🟩 5, 3	🟥 5, 4	🟥 5, 5

Part 5: Boggle (30 points)

Boggle is a board game where your goal is to find as many words as possible in a 4x4 grid. The way to form a word is by taking letters that are adjacent to the previous one until a word is complete (diagonals count as being adjacent). The catch is that every square can only be used once per word.

In a round of boggle, players get 3 minutes to each find as many words as possible. After 3 minutes the answers are compared and only the words that no other players have found is counted towards scoring. If the word is found by no other player, and is a valid word, you have to show on the grid how you have done it and I sometimes forget where the word is on the grid.

I need a program that can find the word I give it directly on the board and give me the position of the letters in order to form the word. The squares of the grid are counted from square 0 to square 15. The squares can only be used once and from any given letter, the next letter needs to be adjacent on the grid.

Input and output specification:

For input, you will receive an *array* of *arrays* of letters that are the grid. You will also receive the word that need to be found on the grid.

For output, you need to provide an *array* of *int* that is the positions of the letters on the grid.

P.S. All the words given will have exactly one valid solution on the grid.

Sample input:

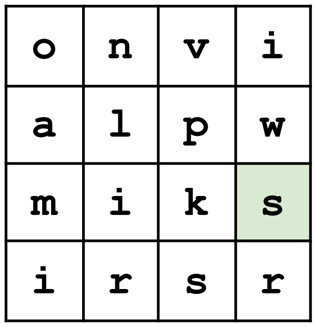
```
grid = [
['o', 'n', 'v', 'i'],
['a', 'l', 'p', 'w'],
['m', 'i', 'k', 's'],
['i', 'r', 's', 'r']]
word = "skips"

grid = [
['n', 'n', 'm', 'p'],
['o', 's', 'a', 't'],
['l', 'b', 'a', 'u'],
['i', 'z', 'f', 'n']]
word = "sablonnat"
```

```
grid = [
[‘s’, ‘a’, ‘a’, ‘u’],
[‘i’, ‘o’, ‘t’, ‘e’],
[‘w’, ‘n’, ‘e’, ‘e’],
[‘p’, ‘f’, ‘e’, ‘u’]]
word = “notais”
```

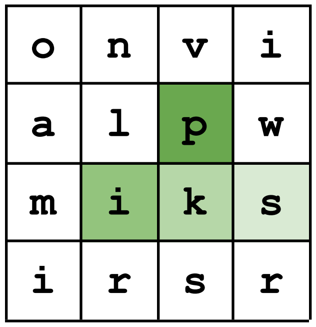
<u>Sample output:</u>

```
[14, 10, 9, 6, 11]

[5, 10, 9, 8, 4, 0, 1, 6, 7]

[9, 5, 6, 1, 4, 0]
```

<u>Explanation of the first output:</u>

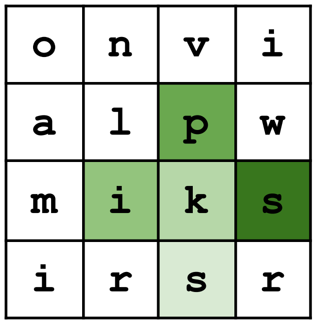


We want to find the word “skips” in the grid.
We first start by finding the first letter ‘s’ in the grid which is at the position 11.



We than take the letter ‘k’ at the 10th position (only choice of ‘k’ neighbor) an than the letter ‘i’ at the ninth position. We then get the letter ‘i’ at position 9 and ‘p’ at position 6.

In that position we try to get the last ‘s’ but the one that is neighboring has already been used so we can’t use it again. This is NOT the valid solution!!



We need to backtrack until we reach another option.

There is no other option until we reach the first letter and start the word with the other ‘s’ instead. We can now get all the letters at the positions 10, 9, 6 and 11 for our final answer.