BIQ Puzzle Project - Part 2 - due date: 04 May 2018

In this exercise the following changes and additions on top of part 1 shall be made:

[1]
Improve your code - do a code review over your code, if you think the code shall be improved please do so.

[2]
Improve the efficiency of your algorithm as far as you can.

[3]
Command-line parameters:
-input <input_file_name>
Mandatory - position of the puzzle file to solve
-output <output_file_name>
Mandatory - position of the puzzle output file (for solution or errors)
-rotate
Optional - indicating whether the puzzle pieces can be rotated - see below
-threads <num_threads>
Optional - indicating number of threads to use (including main)
if not provided default should be 4 threads

The order of parameters is not important (each parameter can appear anywhere in the command line, but if the parameter expects a value - the following parameter is assumed to be its value). In case command line is illegal for any reason: just print usage to screen.

[4]
The **-rotate** parameter:
If the parameter appears it means that the solution can rotate the puzzle elements.
In this case the solution file would look a bit differently:
The output file should be the IDs of the elements in the structure that solves the puzzle:
A single space between IDs on the same line, a new line for each row in the puzzle.
In case an element is used in rotation, the rotation angle should be added in square brackets in the following manner (suppose a puzzle of 6 elements):
1 [90] 2 5 [270]
6 4 [180] 3

Note:
- You are allowed only to rotate the elements, but not to flip them
- Rotation is clockwise, that is 90 means that the top edge becomes right
- The only possible values for rotation are: 90, 180, 270
- 0 rotation shall not be presented in the output file
- This makes a solution without rotations a special case of the generic output file

[5]
The **-threads** parameter:
If the parameter appears or not: the program should be able to use threads smartly. You are not required to manage all the threads allowed, but you cannot use more than allowed!

Errors management note:

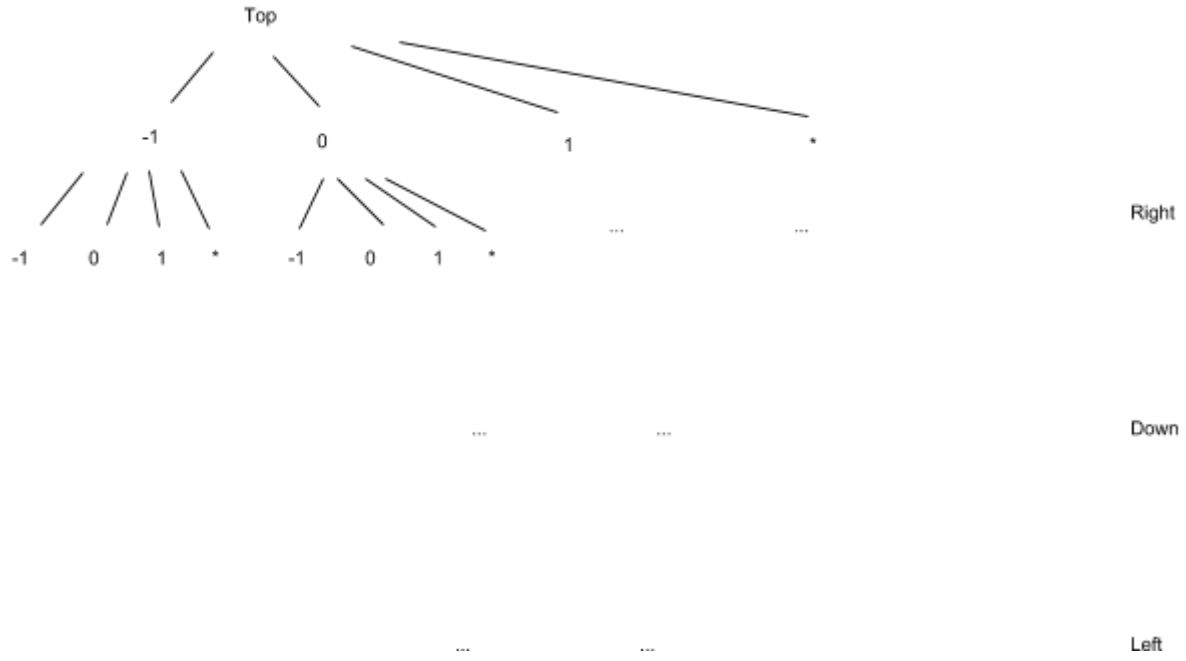
- You are allowed to take decisions and modify the errors from part 1 if they do not fit anymore the rotate option: waive validations, modify the errors or print other errors (for example: the error for missing corners become a bit complicated and maybe irrelevant for -rotate option - so you are free to make changes and even waive it)

Algorithm efficiency note:

- Regardless of the exact data structure that manages your puzzle elements, you may want a method like this in class `Puzzle`:
  `List<PuzzlePiece> getPieces(const PuzzlePieceRequirements& r);`
  The method can return all Puzzle Pieces that meets the requirements and is not in use yet, including possible rotations.

- Consider managing puzzle pieces that are “the same shape” (including - the same after rotation!) as “one”, having the list of IDs and rotations that match this “shape”.

- Consider the following data structure for managing the puzzle elements:



  The ‘*’ in the diagram above represents “there are no requirements for this edge”. With some kind of a data structure like the above you spend 256 buckets to achieve O(1) search for getting the next required element for any requirement and any given previous puzzle piece checked.

- You can postpone the efficiency to later. But have the proper design and API to allow your efficiency thoughts and plans to fit in later.

Good Luck!