PYRAMIDS DOCUMENTATION

version 1.0f

This program solves the puzzle called PYRAMIDS.

Algorithm used for solving the puzzle https://en.wikipedia.org/wiki/Backtracking

Terminology:
Guidance / Guide - List of 4 lists (each of the same length and containing only ints inside) with prompts to solve the puzzle,
N - Length of table,
Table - List of N lists containing N elements - kind of NxN matrix,
List of Possible Options - list containing values that can be interposed in the particular position of table. For every position which is not an int yet, there is a specific kind of list.

How to use

Start the program by running the game module. When a proper message appears, please type the guidance in the terminal. If the given guide is correct, the program will display a table with the solution for this puzzle. Otherwise it will display a message that says this guide is insufficient to solve it.

Solver module

class `solver.Solver` (*guide: list*)

Bases: `object`

Main class of the whole program. Contains methods which gradually solve the whole puzzle. Some of them are based on backtracking algorithm.

Parameters: **guide** *(list)* – Guide for solving the table, contains 4 lists

`check_guidance_prompts` (*table*)

Contains `counter()` methods (each of them works cognately). It compares the growth of the pyramids' height with the values from guidance. If compared values are different, returns False, so that means there are some values on the incorrect position of the game table. If values are the same, returns True.

`check_if_only_ints` (*table*)

Checks if the type of every element of the table is int (must be positive number).

`fill_table_with` (*table*, *index*, *attempt*, *tree*, *stage*)

Interposes a particular list of possible options with one of its elements and solves the table with this value.

Parameters: **table** *(list)* – current state of table
index *(list)* – indexes of position on table
attempt *(int)* – value that will be interposed
tree *(dictionary)* – tree of previous stages
stage *(int)* – stage on which program is making changes

`generate_raw_table` ()

Returns: Generates a clear table. Every position of the table becomes *0*.

Return type: list

`get_new_root` (*table*)

Backtracking algoritm can be visualized as roots of a tree and checking every possible connection to the bottom. In this method the program finds the list of possible options which contains the least elements. When it is choosing lists with fewer elements it will have fewer options to check when the program gets back.

Returns tuple containing index of list of possible options and the elements which this list contains

Return type: tuple

`get_previous_way_info` (*ways: dict, stage*)

For every stage in the *tree* it deletes the value from *options* that has been chosen to try. In that case this method appends options that have been already tried in the current state.

Parameters: **ways** (*dictionary*) – dictionary with previous moves

Returns: updated ways dictionary

Return type: dictionary

`get_stage_info` (*table, tree: dict, stage*)

While the program is going deeper in solving puzzle, it is making some choices, so it is creating copies of stages before it has to choose. At this point it is copying the current state of the table and passing most significant information further.Firstly this method is getting a dictionary (tree), which contains information of previous decisions. Then it creates the current stage, and adds information (position and its elements) of one of the lists of possible options for further solving.When program goes back (after finding wrong solution) and chooses another option it deletes stages which were on the deeper level in relation to current one.

Parameters: **table** (*list*) – current state of table
tree (*dictionary*) – tree, which contains information of previous moves/choices of program
stage (*int*) – current stage of the tree

Returns: returns tuple which contains updated tree, and last stage

Return type: tuple

`guess_solution` ()

Starting method for `try_to_fill()`. Tree and ways dictionaries are created here.

`interposer` (*table*, *in1*, *in2*, *v*)

Substitute for a more overall approach to the `solve_if()`-type methods.

Parameters: **in1** (*int*) – position indexes of the first list - which row
in2 (*int*) – position indexes of the second list - which column
v (*int*) – value that will overwrite the *0*

`is_everything_alright` (*table*)

Checks if the solved table is consistent with guidance and if there are no conflicts.

`is_table_correct` (*table=None*)

Checks if each number occurs only once in the row and in the column of the table.

Parameters: **table** (*list*) – previous table

Returns: True when given combination is correct (only one a occurence of each number in row and column), otherwise returns False.

Return type: bool

`length` ()

Returns: table's length

Return type: int

`limit_potential_solutions` (*base_table*)

Method reduces possible options that can occur. It uses two types of other methods:

1) `limiter()` *(runs first)*:

Finds values which are stated in the table. Next removes them from lists of possible options which are in the same row (or column) as these values.

For example:

[1, [1, 2, 3, 4, 5], 4, [2, 4, 5], [1, 3, 5]] -> [1, [2, 3, 5], 4, [2, 5], [3, 5]]

2) `find_unique()` *(runs after limiter)*:

If in one row (or column) there are lists of possible options and there is a value that occurs only in one of them, it must replace the list where it was.

For example:

[[1, 2, 3], [1, 3, 5], [1, 3]] -> [2, 5, [1, 3]]

Whole method runs in a loop unless there are no differences in the following tables. In one loop, the method makes a copy of a table. Then reduces the possible options with `limiter()` and `find_uniqe()` functions, firstly for rows, secondly for columns. Next step is comparing the copy of the unchanged table with the changed one. If there are differences the next loop appears, if not, the method returns the last table.

`reduce_from_guide` (*table*)

Reduces possible options that can occur. If the guide prompt excludes the possibility of value to occur, then it is removed from the list of possible options.

For example:

If the guidance shows *3* and the length is 5, then: *5* cannot occur on the first and second position and *4* cannot occur on the first position.

`set_table` (*table=None*)

Checks if the table is partly solved and returns it to other methods (if correct).

Parameters: **table** (*list*, *optional*) – previous table (if exists)

Raises: **InvalidTableError** – if height of pyramid on given table is out of range

`solve_if_N (table=None)`

Method finds if there is *N* in guidance, then overwrites in succession the *0* in row or column with values in range from 1 to N on proper positions

Returns: returns overwritten table

Return type: list

Raises: **InvalidTableError** – if there is value other than *0* or *K* in the position where *K* were meant to be. (K is a value in range from 1 to N)

`solve_if_ONE (table=None)`

Method finds if there is *1* in guidance, then overwrites the *0* in table with *N* on proper position

Returns: returns overwritten table

Return type: list

Raises: **InvalidTableError** – if there is value other than *0* or *N* on position where *N* were meant to be

`sort_possible_options` (*table*)

Method runs across the table and finds lists of possible options. When one finds one, append it to the proper place in a dictionary of options - keys mean amount of elements in the list, values mean index of position in table.

Returns: Returns dictionary where keys represent amount of elements in lists of possible options and values represent indexes of position on table

Return type: dictionary

`try_to_fill` (*table, tree, ways, stage, backing=False*)

Method gets information from the last stage by `get_stage_info()` method. Then fills the table with one of the possible options from the list of these options and solves for this combination.
Next checks if the table is fully filled with ints. If not, the method is running again and again, until every position of the table will be a number.
When the table is completed, `is_everything_allright()` function tests its correctness. If so, the program returns the final solution. In other cases, the method must step some stages back and try filling the table in other way. Whole program runs this method until it finds the correct answer.

Parameters: **table** (*list*) – current state of table
tree (*dictionary*) – tree of previous stages
ways (dictionary) – dictionary with previous moves
stage (int) – stage on which program is making changes
backing (bool, optional) – information if algorithm is going back (default False)

Returns: Solved puzzle if it is solvable

| **unlist_single_value** (*table*)

If the list with possible options contains only one value, then it must be this value. Method replaces the list with this value.

| **zero_into_list** (*table=None*)

When the program finishes solving cases with *1* or *N* from the guide, then replace every *0* from the table to list. Each of these lists contain every value in range from 1 to N. List contains possible options to interpose in this position.

Project_errors_piramidy module

| *exception* **ProjectErrors**

Bases: **Exception**

Parent class for custom errors.

| *exception* **InvalidGuideError**

Bases: **project_errors_piramidy.ProjectErrors**

Raised when the format of guidance is invalid.

| *exception* **InvalidTableError**

Bases: **project_errors_piramidy.ProjectErrors**

Raised when there is conflict on the table. Usually when:

1) Some table's values exceed maximum height,

2) There are the same values in rows or columns.

| *exception* **PyraminInterposeError**

Bases: **project_errors_piramidy.ProjectErrors**

Raised when some method wants to interpose value that already exist in table with other value

Game module

| game.insert_guide()

Method that requires the guidance as an input for the program.

| game.main()

Method which contains all of the essential methods from Solver.py. Order of running methods:

1) `insert_guide()`
2) `set_table()`
3) `solve_if_ONE()`
4) `Solve_if_N()`
5) `is_table_correct()`
6) `zero_into_list()`
7) `reduce_from_guide()`
8) `check_if_only_ints()`
9) `is_everything_alright()`
10) `guess_solution()`

When the puzzle is solvable, the program prints the table with the correct answer. If there is no solution for it, then a proper message is printed.

| game.try_again()

Method that reads the input again after failed attempt