Hanoi Towers Game Practice

Report Document

Group Code: AM
Practice Teacher: Dolors Sala Batlle

NIAs of group members
NIA student 1: 218926
NIA student 2: 218863
NIA student 3: 218861

The Hanoi towers game practice submission must include the code and a report addressing the questions and issues indicated document. You can answer the questions directly here, and deliver this same document (modifying the filename for submission and grading management), or build a different report document containing the requested information.

1. Understanding of the Hanoi game: Recursion on paper

This part wants to consolidate the comprehension of the recursion concept. And to do so it asks you to start working the game by hand on a paper.

Build in paper the tree execution (explained in theory) to see graphical representation of the function's call. (Review the theory slides of Recursion chapter
Do the tree for the following cases:

a) hanoi(3, 0, 1, 2)
b) hanoi(4, 0, 1, 2)
c) hanoi(5, 0, 1, 2) [Optional]

Label each move node of the tree with the **move count** and the **depth level** for each level of the tree. Add to each move call in the tree the picture of the state of the towers. Take a picture of each of the trees and include them in your submission. Include in here the pictures or the name of the separate files included in the submission.

Note the tree execution for the case of 3 discs is given in the following site: http://www.cs.cmu.edu/~cburch/survey/recurse/hanoiex.html. But in yours, you have to add with respect to this graphical representation, for each move node in the tree:

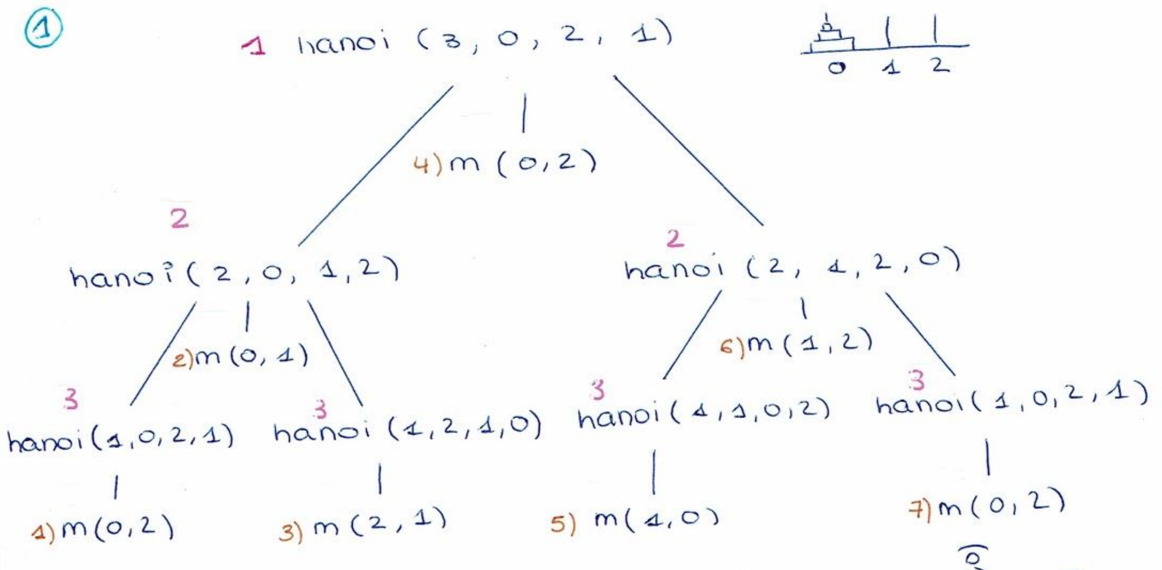
1) the move count
2) the recursion depth
3) the picture of the state of the game (picture of the 3 towers with the discs)

Answer:

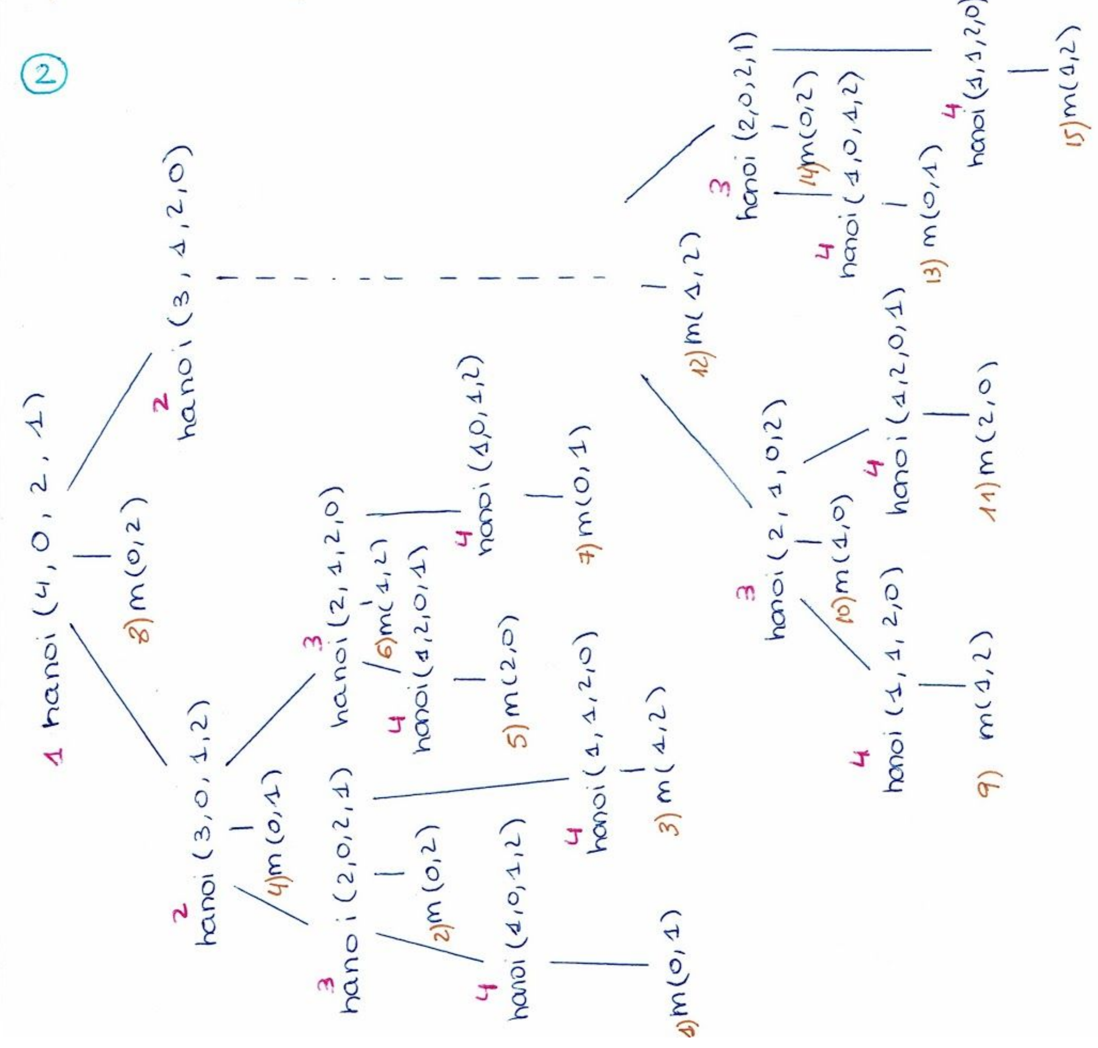
hanoi (disks, source, dest , aux) m (source, dest)

the move count   recursion depth   # move

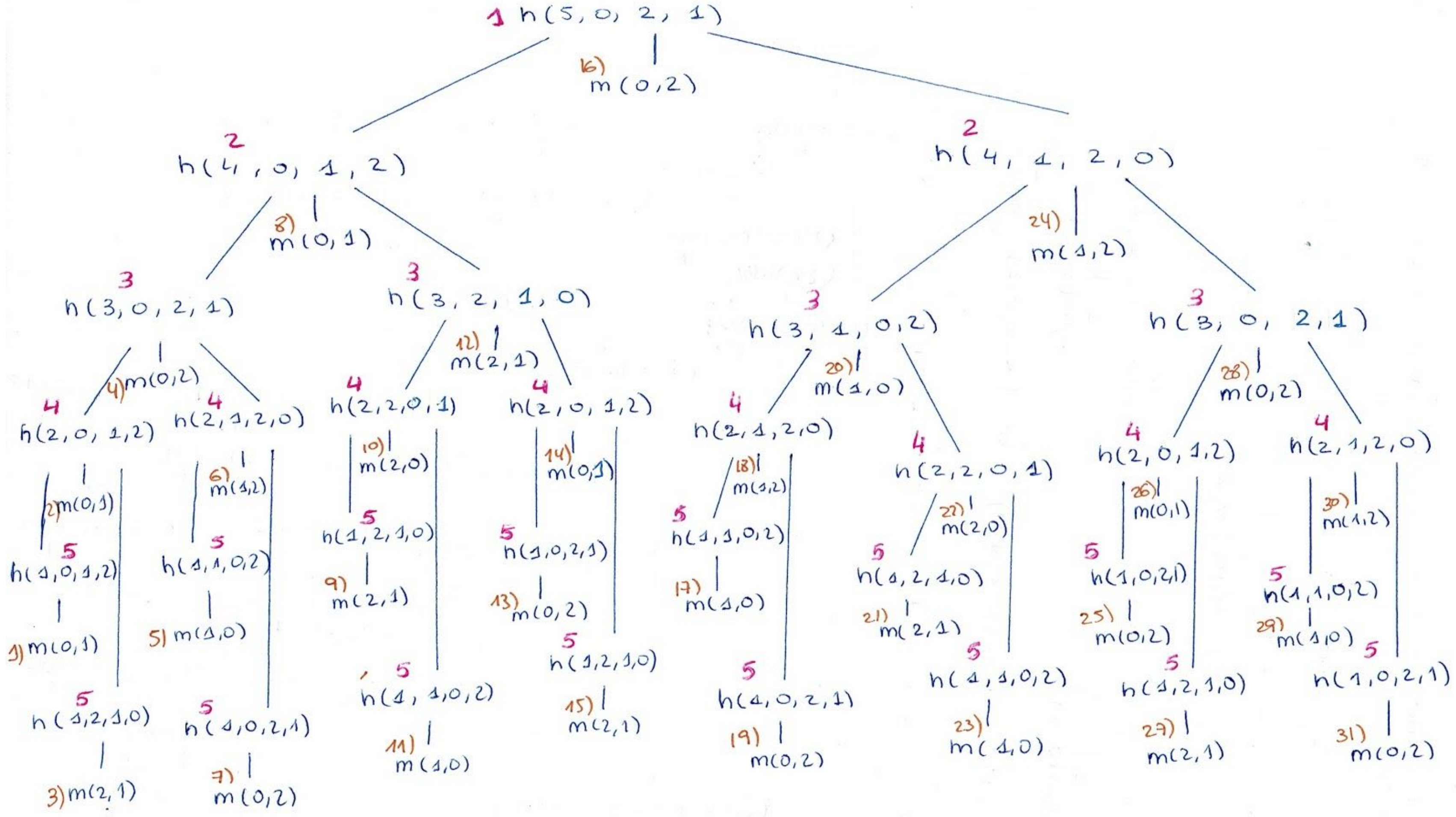
1



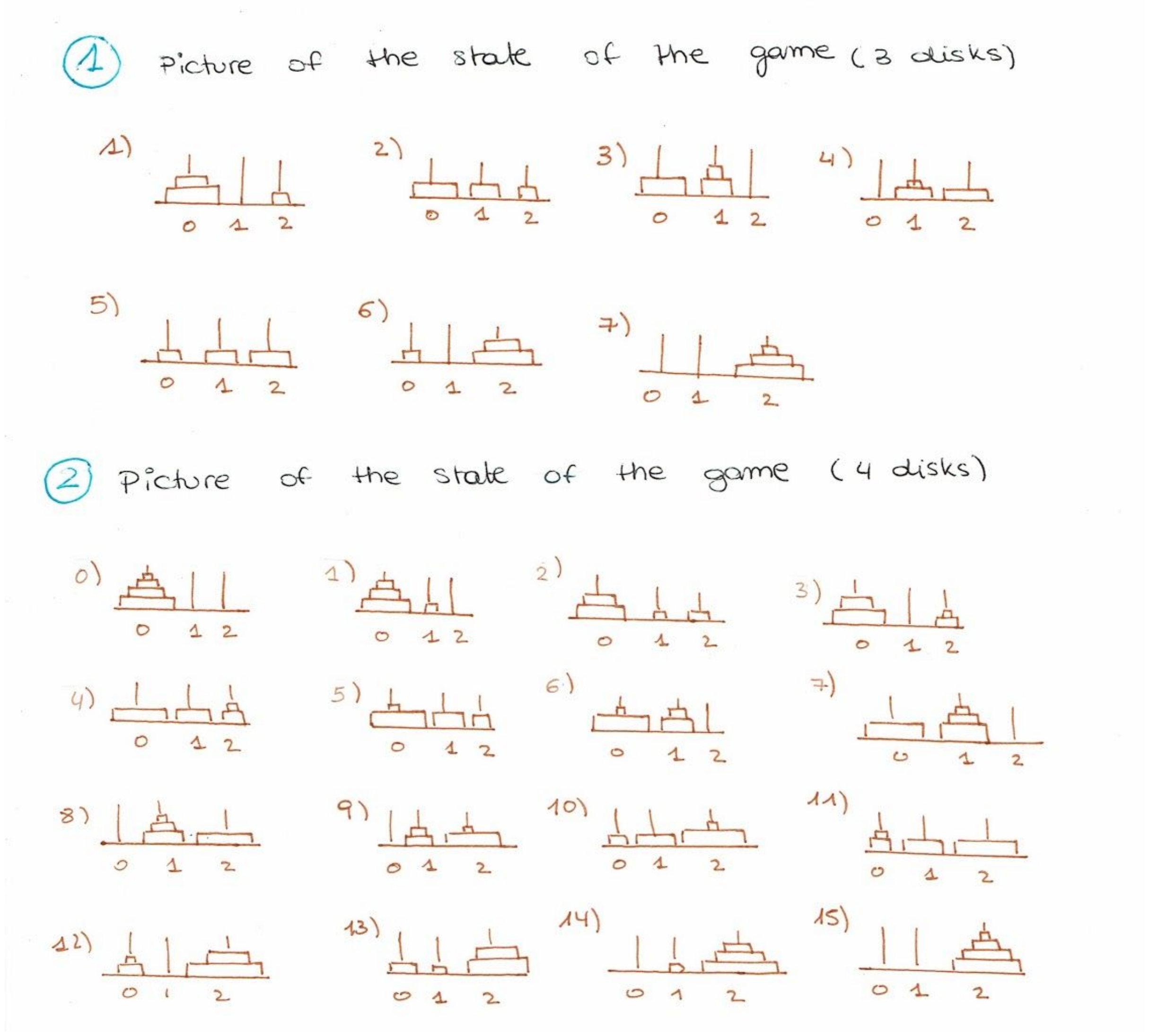
2



* Instead of hanoi(), we write h() because there's no space to write it all. (# h() = hanoi())



Universitat Pompeu Fabra Barcelona


1 Picture of the state of the game (3 disks)
1) 2) 3) 4) 5) 6) 7)
0 1 2
2 Picture of the state of the game (4 disks)
0) 1) 2) 3) 4) 5) 6) 7) 8) 9) 10) 11) 12) 13) 14) 15)
0 1 2

Analyse the maps built and answer the following questions.

Note: when you have the program running you can/should check if these are the correct answers and the program generates the same values.

1.1 - What are the destination tower and depth level of the first disc move done in each of the (three) cases?

Case 3 Disks:

Destination tower: 2

Depth level: 3

Case 4 Disks:

Destination tower: 1

Depth level: 4

Case 5 Disks:

Destination tower: 2

Depth level: 5

1.2 - The last move of the first sub-tree of depth 2 in the case a) (3 disks) is move number 3. What exact (pile of) discs (sub-tower) has moved this sub-tree when finishing the move of this sub-tree?

The pile of the first two disks.

1.3 - What is the last move of the first sub-tree of depth 2 in the case b) (4 disks)? What sub-tower has it moved?

Exactly the same as in the previous question. It has moved the pile of the first two disks (disk 1 and 2).

1.4- What is the first move of the last sub-tree of depth 2 in the case b) (4 disks) and at what depth level is it executed? When you have the code running, execute the code with the debugger and stop the execution on this level and do a screenshot of the call stack and include it here. Check and comment the values of the parameters of the different recursive calls shown by the debugger in relation to tree path of your handwritten tree.

Answer:

1.5 - How do these three simple instructions remember which move it has to do and be consistent with the previous and next moves if it does not keep information of the disks or of the state of the towers? Reason the answer.

Can we know which disk is moved in every move? If so explain how it knows it.

Think and explain how this basic algorithm should be modified/upgraded to know the state of the game, the depth and recursion value at any time of the game.

Because it does not need to know. The task of a recursive call is to move a sub pyramid from point source to point target, it's all it needs to know. If the tower is larger than 1 disk has to move the tower of 1 less disk first and if the tower is one disk it just moves it.

After this study, do you think you fully understand why the game works and provides the minimum set of moves to solve the game? Do you think you have full understanding of the recursion now? If not, remember that this understanding is needed before starting coding the solution. So working on paper is required before getting into the code and trying to understanding and extend the skeleton. Remember you can ask any type of questions in consulting hours, including the ones on paper (and not just computer/debugging questions).

2. Design of the submitted code (graphical representations)

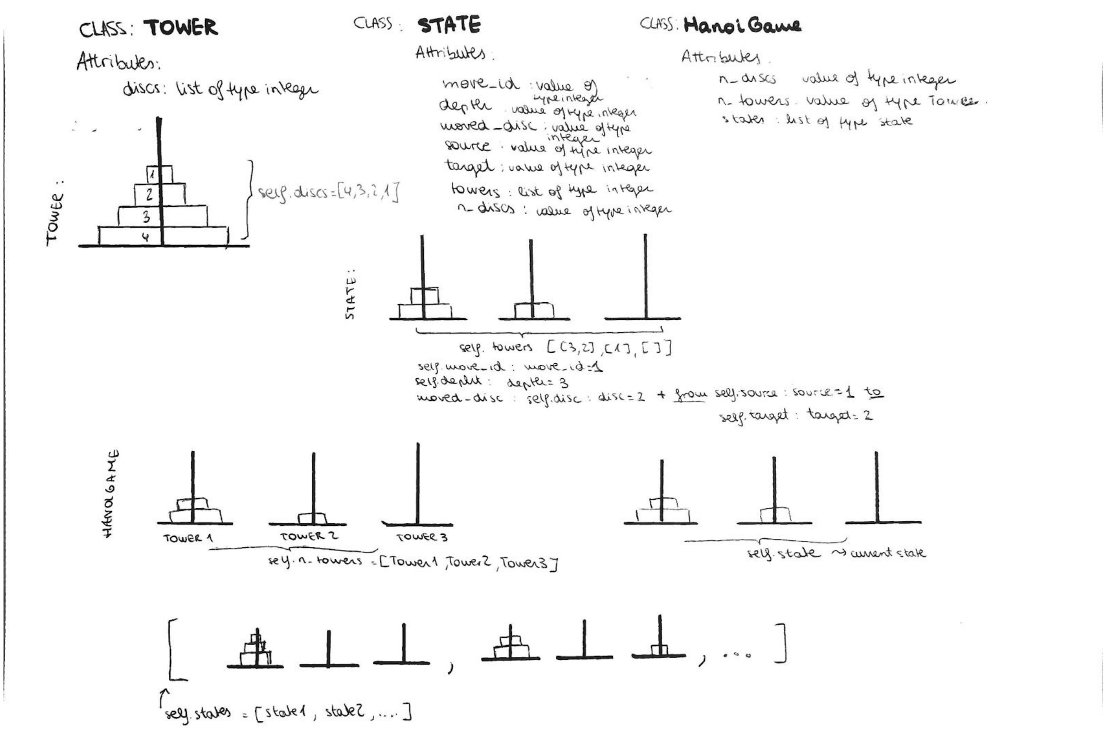
Note: There are several formal notations to provide graphical representations of a program that correspond to different view of the programs. We do not expect your graphical representations to follow any of these standards. Instead, we just want one drawing of the program that shows graphically the structure of the program to help you, and anyone, understand it. As examples, check the following link:
https://www.tutorialspoint.com/uml/uml_component_diagram.htm.

You can include the figures in the document or have separate files with the figures. If they are separate files indicate the name of the files included and what each one contains.

2.1 - Provide a graphical representation of the **data structure** used by the program. This means for every class provide a graphical picture that shows what attributes it has and how each of the attributes is defined.

An example of graphical representation of structures can be taken from the pythontutor representations (shown in some of the theory slides). Feel free to use any other drawing that visualized what information each class contains. Make sure the drawing distinguishes the design provided by the skeleton from any additional attribute of class you may add to complete the solution (if any).

Concisely explain and justify the data structure used in the solution (explain everything including also the part provided in the skeleton).

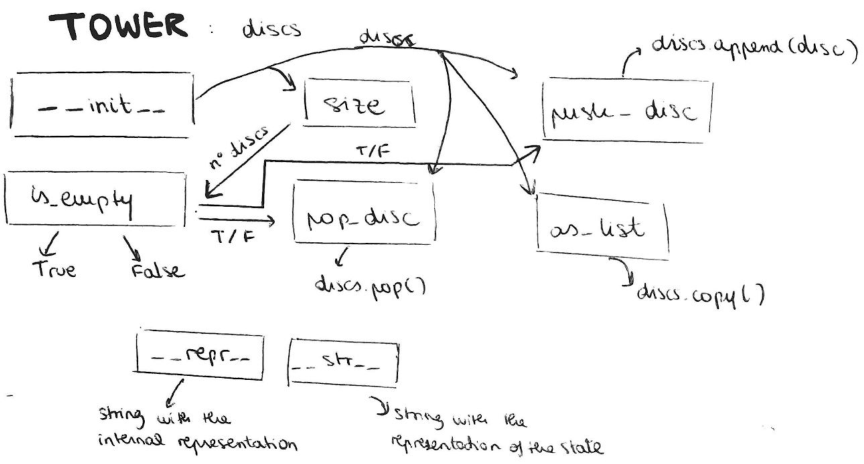

CLASS: TOWER
Attributes:
discs: list of type integer
TOWER:
1
2
3
4
self.discs=[4,3,2,1]
CLASS: STATE
Attributes:
move_id : value of type integer
depth : value of type integer
moved_disc : value of type integer
source : value of type integer
target : value of type integer
towers : list of type integer
n_discs : value of type integer
STATE:
self. towers [[3,2],[1],[ ]]
self.move_id : move_id=1
self.depth : depth= 3
moved_disc : self.disc : disc=2 + from self.source : source=1 to self.target : target= 2
CLASS: HanoiGame
Attributes:
n_discs value of type integer
n_towers : value of type Tower.
states : list of type state
HANOIGAME
TOWER 1
TOWER 2
TOWER 3
self.n_towers =[Tower1, Tower2, Tower3]
self.state ~ current state
[, , ...]
self.states = [state1, state2, ...]

We have defined the class Tower, State and HanoiGame since each one will contain a different type of data. For Tower, we have thought that the best option to store the discs that it contains, is a list, since it will keep more than one value.
For State, all the attributes are integers except towers, which is a list because there are at least 3.
For HanoiGame, n_discs must be an integer since it's a simple number. N_towers is a list of values of the class Tower (note: on the diagram it's incorrect because instead of "list" it says "value"), since it has more than one tower, and at the same time, every tower contains different data. Finally, States is a list of values of the class State, for the same reason as n_towers.

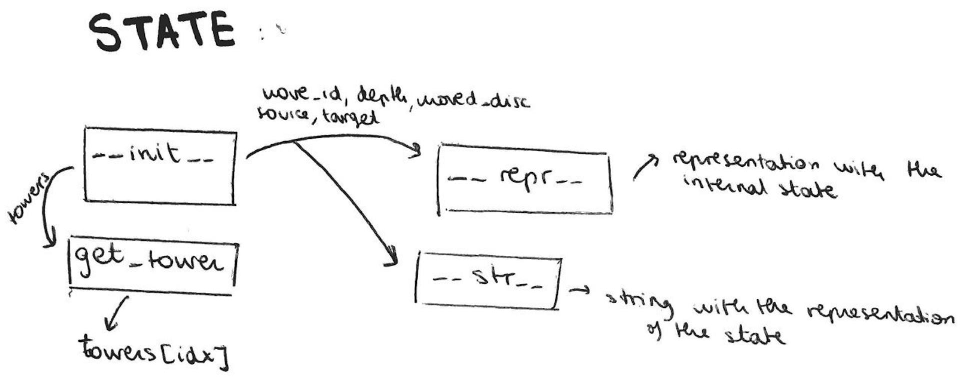
2.2 - Provide a graphical representation of the **structure of the program** you submit. The diagram has to have the functions/methods (boxes) and their interactions (arrows) indicating the data passed from one to the other (labels in arrows indicating the data passed in one direction of the arrow and the opposite direction the return).

Make sure that the diagram distinguishes and explains any difference or addition from the provided skeleton. Also indicate and represent in the diagram the functions included in the skeleton but not used in your final solution.

Concisely explain and justify the functions and methods available in your design, and how they are distributed in the files of the program (explain everything including also the part provided in the skeleton).



__init__ : is the constructor of this class, it's the function that initializes the class, so it will provide the basic information to the rest of the functions.
Size: returns the number of discs that the tower contains
Is_empty: checks wether the tower is empty or not and returns True or False
Pop_disc: takes away the last disc of the tower, and push_disc, appends to the tower a disc
As_list: makes a copy of the list of discs so that we get a copy of all the combinations that occur
__repr__ and __str__ : are used to represent the internal state of the towers, and the general state, relatively

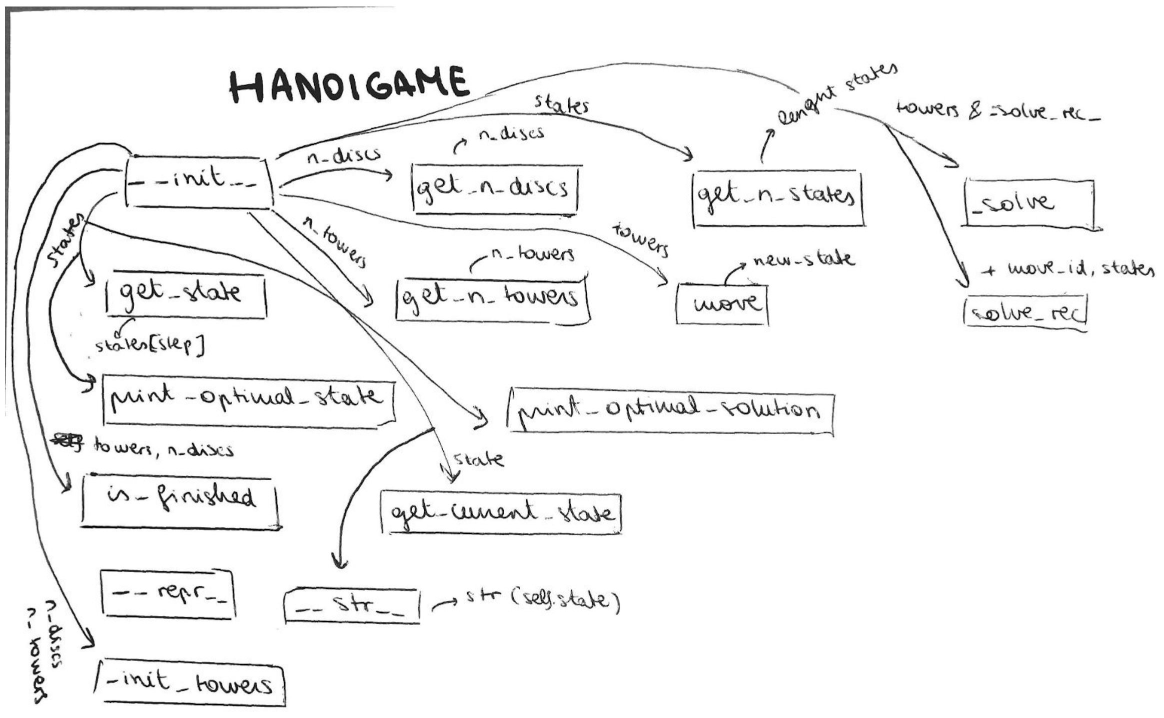

STATE :
move_id, depth, moved_disc source, target
__init__
towers
__repr__
representation with the internal state
get_tower
__str__
string with the representation of the state
towers[idx]

__init__ : Initializes a state with all the information needed to represent it.

- move_id: Identifier of the move. Ideally, the step number.

- depth: Recursion depth at which this state is generated.
- moved_disc: The disc moved to reach this state. Ideally, a disk is defined just by its size.
- source: Tower from which the disc is moved.
- target: Tower to which the disc is moved.
- towers: Towers of the game.
- n_discs: Number of discs of the game.

get_tower: returns the tower corresponding to the index given.
__repr__: returns a string with the internal representation of the state.
__str__: returns a string with the representation of the state in the requested format.



__init__: Initializes the game with n_discs and n_towers, the game can be solved and stored to consult.

get_state: Returns the state at the requested step in the optimal solution.
get_n_discs: Returns the number of disks of the game.
get_n_towers: Return the number of towers of the game.
get_n_states: Returns the number of states of the optimal solution.
move: Moves a disk from source tower to target tower.
_solve: Generates and stores the optimal solution, reinitializing the towers afterwards.
_solve_rec: Recursive call to solve the hanoi game optimally.

print_optimal_state: Prints the optimal state at the selected step in the required format.
print_optimal_solution: Prints all the states of the optimal solution in the required format.
is_finished: Checks if the interactive game is finished, returns True if is finished, False otherwise.
get_current_state: Returns the current state of the game.
__repr__: Returns a string with the internal representation of the game.
__str__: Returns a string with the representation of the current state of the game in the requested format.
_init_towers: Returns a list of Towers of length n_towers initializing the first one with n_discs discs.

3.1 - Concisely explain the submitted program in terms of the implementation of the above design diagrams. Do not replicate literal code in text. Just briefly describe what is needed to complement the code.

Hanoi.__init__(n_discs, n_towers=3):
This is one of the most important functions of the practice, first we store the given parameters (n_discs, n_towers) into a self copy so the methods of the class can use them, we also check that those parameters are correct raising an exception otherwise. Then we need a list of object Towers to represent the state of the towers of the game, we initialize them including the first tower with all the required disks. With the previous information we can create a State that we will use when we need to print the current state of the game. Finally we create a variable move_id initialized to 1 and a list of states with initially only the initial state. This list will contain all the steps of the optimal solution in order, and the function self._solve() will be in charge of fill it.

Hanoi.get_state(step):
Returns the state at the requested step in the optimal solution. We have all the states in the list self.states so we just have to access the state on the index step and return it checking if the index is in range in the process.

Hanoi.get_n_discs():
Returns the number of disks of this game. We just have to return self.n_discs.

Hanoi.get_n_towers():
Returns the number of towers of this game. Similar as before, we just have to return self.n_towers.

Hanoi.get_n_states():
Returns the number of states of the optimal solution. Ideally, it should be the size of the structure used to store the optimal solution states, so we return the length of self.states, we also added a check to see if it matches with the calculation of the optimal number of moves and if it doesn't we raise an exception.

Hanoi.move(source, target, move_id=None, depth=None):
Moves a disk from source tower to target tower. For that we have to make use of the methods offered by the Tower class, in particular the pop_disc() and the push_disc(). We first extract the disc of the source tower, source is an integer ranging from 0 to 2, so the tower corresponding to the index source is just self.towers[source], then we call the function pop_disc() of that tower and store the return value, which is the disk pop'd, in a variable. Then we take the target tower which will be self.towers[target] and call the push_disc() function of that specific tower passing it the disk we pop'd previously. Finally we create a new state with the disk moved and the source and target parameters that we received as well as the list of towers so the State() constructor can make a copy of them and in the case of move() being called by in the recursive solution the move_id and the recursion depth values. If is not the case of being called by the recursive solution we update the current state self.state to that new state that we just created. Finally we return it.

Hanoi._solve():
Generates and stores the optimal solution, reinitializing the towers afterwards. In this function we just have to make some preparations after and before calling self._solve_rec(). There is almost no preparation needed, just make sure to pass the correct parameters to the recursive function and after the recursive function has executed we have to reinitialize the towers since the stat will be the corresponding of a solved state.

Hanoi._solve_rec(n_discs, source, target, aux, depth=0):
This is the recursive function. It has to call itself first of all for n_discs-1, to move the tower of one less disk from the source to the auxiliar, we also increment de depth by 1 . Then move the disk from the source to the target, the move function returns the new state generated so we append it to the list of states of the optimal solution and then increment the self.move_id by 1 and finally we have to move the tower of n_dics-1 agan from the aux to the target.

Hanoi.print_optimal_state(step):
We get the state of the optimal solution of the required step by using the previous implemented function self.get_state(step) and just print it with print().

Hanoi.print_optimal_solution():
We iterate over the list of states and print each one with print().

Hanoi.is_finished():
We have to check if all the disks are on the last tower. For that we access the last tower that we have on self.towers, the last one corresponds to len(self.towers) - 1 or self.n_towers -1 or just using the python list index [-1], then we check if the size of that tower corresponds to the number of total disks (self.n_diks).

Hanoi.get_current_state():
Just return the current state that is saved in self.state.

Hanoi.__repr__():

The repr of Hanoi is the same as the str so we return str(self).

Hanoi.__str__():
We just have to call the string representation of the current state, so we call str() of self.get_current_state().

Hanoi._init_towers():
This is the only auxiliary function we have added. It just creates a list of towers and it fills the first one with all the disks and returns the whole list. We put this code in an auxiliary function because it was repeated twice, one in the __init__ and the other after the recursion call in self._solve().

3.2 - State of the delivery: Indicate if the program does everything requested in the assignment and if your code passes all tests provided.

If the submission is not finished indicate what is completed in terms of functionalities requested in the assignment, and also indicate what parts of the design diagram is done and which part is not completed or not working. Provide a brief description of the non-working part indicating as much as possible the detail you know at this point of why it is not working and what you have done to try to get it work

All of this has to give a clear view of the current state of the submission and the work done to do it.

All of the three classes are fully implemented and work correctly together, all the functionalities called by the main and the test function correctly.

The program does everything that is requested, and the code passes all the tests, too.

4. Learning process

4.1 - Learning process: Think about what you have learned while doing this practice and explain your lessons learned. Include also explanation of things you have tried and have not been successful as the good learning also derived from unsuccessful decisions. The evaluation of the submission will also depend on the subjective evaluation of your experience.

We have learned especially about python Classes, how to create them, how to use them and how to relate them with each other (something that, at the beginning, took us a little time to properly understand). Also, we have learned about how to structure properly a program longer than what we are used to. Finally, seeing that one of the members has more knowledge about programming, the other members, we have learned some techniques to code more efficiently and in less lines.

4.2 - Final Evaluation Experience: Provide a constructive evaluation of the quality of this assignment with respect to meet the learning objectives of the related theory concepts. Comment from all perspectives you feel relevant (motivation, interest, usefulness, difficulty, efficiency, detail, duration, what is missing or too much? Or any other...) to improve it.

It is true that we have learned a lot with this practice. Not only about Classes, recursion, data structures, etc. but also to find our own resources.
We want to comment, that maybe this practice would have been easier if we had done first all the theory about recursion, classes, data structures, etc. and then started with the practice. Probably, instead of taking 5 weeks, we could have done it in 2.
We consider that it is useful, but we think that if instead of making one long practice, we had done more exercises (and shorter, with increasing difficulty), maybe we could have assimilated the knowledge faster and better.