Hanoi Towers Game Practice

Report Document

Group Code: AN
Practice Teacher: Sergio Ivan Giraldo

NIAs of group members
NIA student 1: 217447
NIA student 2: 217723
NIA student 3: 194985

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.

Remember to put the group identifier in all files submitted both inside the file and in the filenames.

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: Included in the submission folder

Analyze 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?

Answer:

In Hanoi (3,0,1,2): Destination tower is 1, depth level is 3

In Hanoi (4,0,1,2): Destination tower is 2, depth level is 4

In Hanoi (5,0,1,2): Destination tower is 1, depth level is 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?

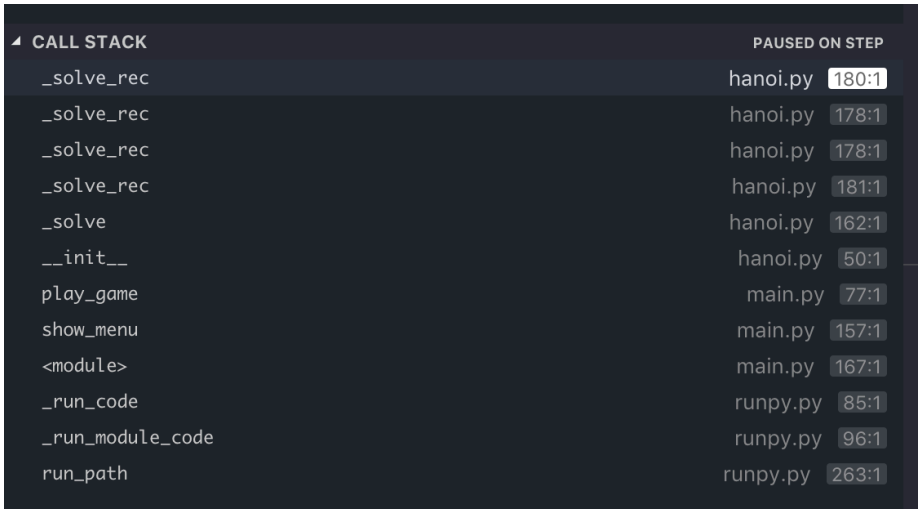
Answer: It moved the pile of discs formed by 1 and 2 from tower 0 to tower 1

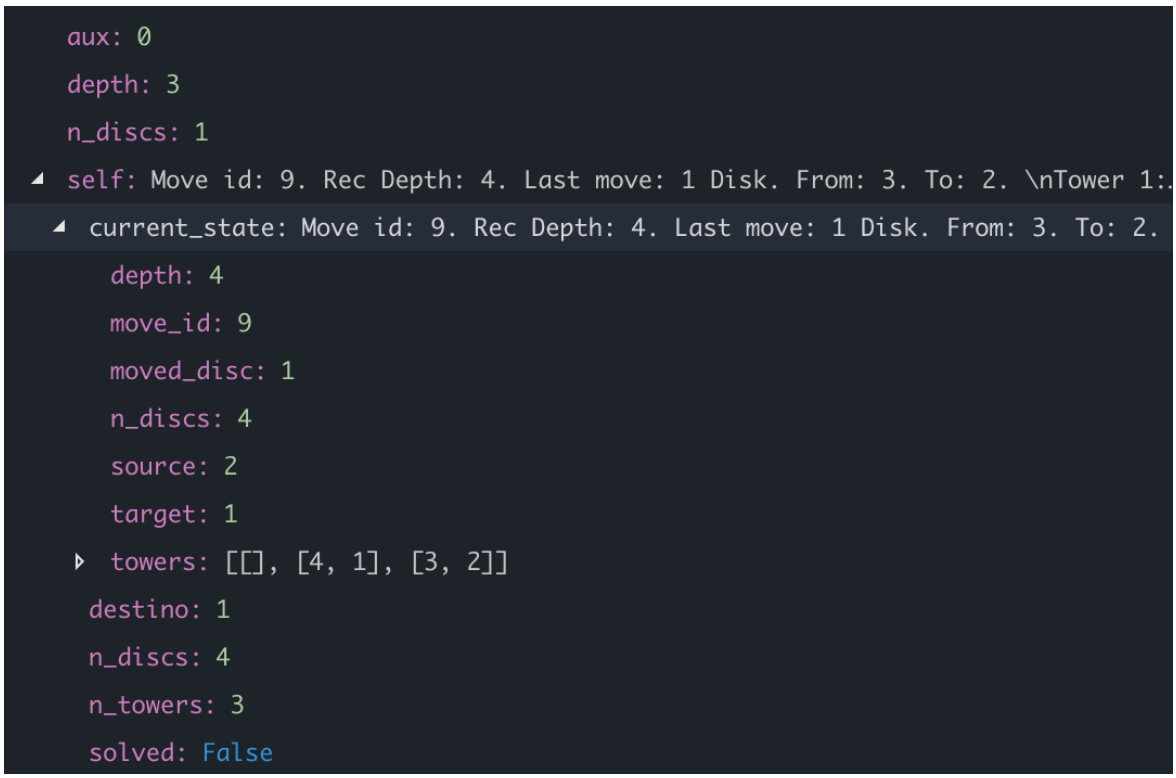
1.3 - What is the last move of the first sub-tree of depth 2 in the case b) (4 disks)? What sub-tower has it moved?

Answer: The last move is the number 7, which moves disk 1 from tower 0 to tower 2. The whole sub-tree moves all the disks 3,2 and 1 from tower 0 to tower 2, and after that the move 8 will change the disk 4 from tower 0 to tower 1.

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: The move 9, which moves the disk 1 from tower 2 to tower 1, and it is executed in the level of depth 4.





The relationship between the data in this image and the picture of the tree is:

- Move_id has the same name and value
- Depth has the same name and value
- N_discs: 1 inside the recursive function, the value equals to 1 (as we see above), inside the current state, it equals to the total number of discs
- Target has the same name and value
- Source has the same name and value
- Aux has the same name and value

1.5 - How do these three simple instructions know which move has to be done and how it is consistent among moves if it does not track information of the disks or 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.

Answer:

1. It does not know which move has to be done. It follows the next reasoning: The problem of moving n disks from one tower to another is the same problem as moving n-1 disks from the source tower to the auxiliary one, then moving the last disk to the destiny tower and last of all, mounting again the n-1 disks above the big one. You can do this process until only one disk is left, so the program only needs to know where to mount the tower in order to work efficiently.

2. We can know which disk Is moved in every move by the level of depth where the move is done. For instance, the disk 1 is only moved in the maximum level of depth. The disk 2 is only moved in the second maximum level of depth, and so on… We also know the disk because the pop function returns it when we start the move algorithm.

3. In order to upgrade the algorithm, you could add some parameters to the recursive call. For instance, you can know the number of moves played by the length of the array of states it creates every time a move is done and the depth where the move is done by adding one every time the function calls itself (and it will subtract one when the call ends). It can also know the state of the game because it can have parameters of the towers that are moved when the moves are done.

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 understand 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).

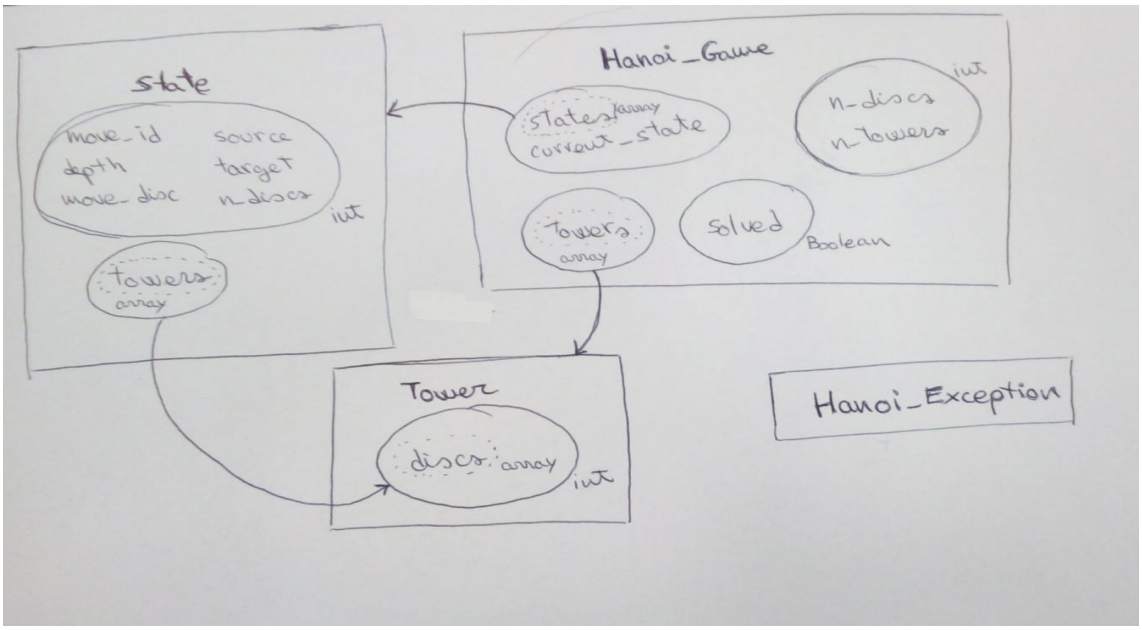
Concisely explain and justify the data structure used in the solution (explain everything including also the part provided in the skeleton).

Answer: There are 4 classes. The HanoiGame is the brain of our program. It's the one that makes the moves, calls the other classes, creates states and stores the optimal solution among many other features. It contains all the information related with the current state of the game when the user is playing, and an array of states with all the solution steps. Finally, it contains a Boolean that is used to control whether if the game has been solved by the algorithm or not.

The Tower class is the one that contains the discs of each tower. It also has a few methods that you can use to put or quit discs, as we'll see below.

The State class is the one that saves every step (like a photography) that are done in order to reach the optimal solution. It contains not only the position of the disks inside the towers, but all the metadata (depth, move_id…) of the move as well.

Last but not least, the HanoiException class has no attributes, so it's not worth mentioning it in the data structure representation. We'll cover it when we explain the structure of the program.

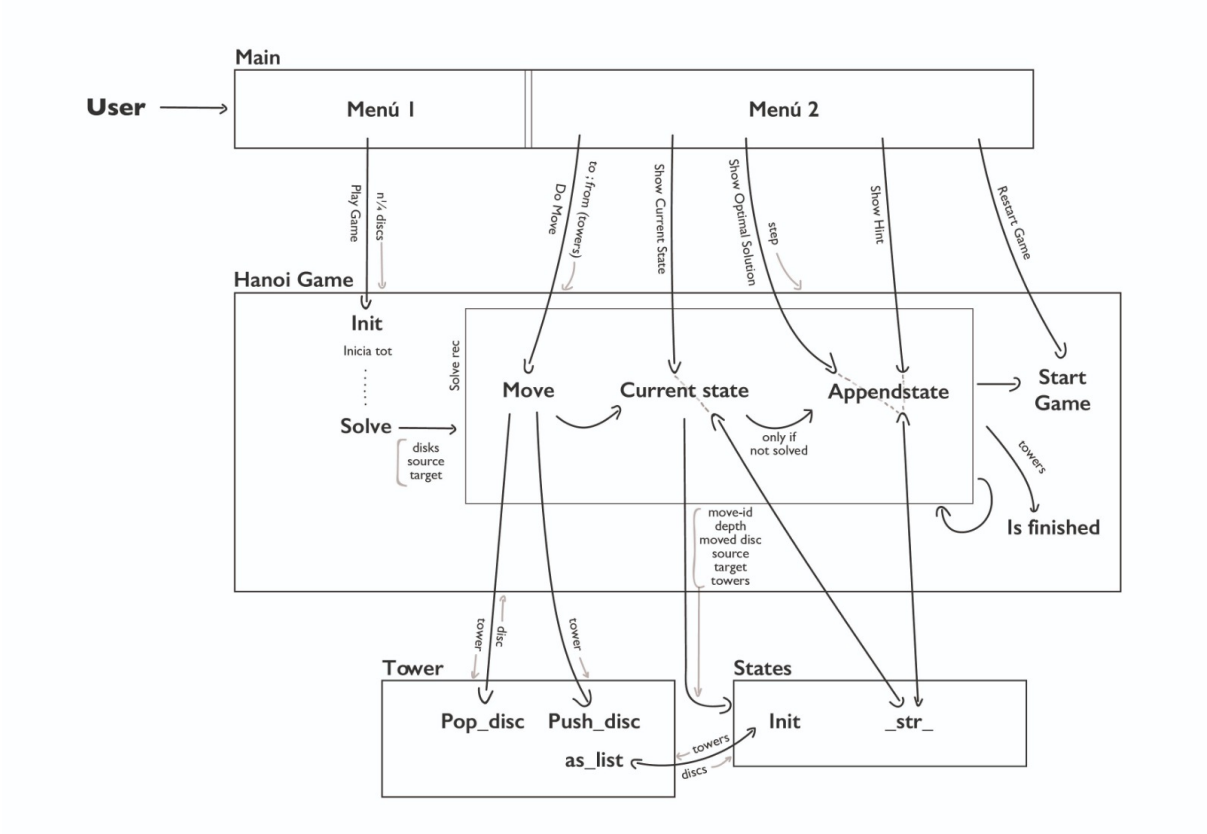


2.2 - Provide a graphical representation of the **structure of the program** you submit. The diagram has to have the functions/methods (boxes) and their interactions (arrows) indicating the data passed from one to the other (labels in arrows indicating the data passed in one direction of the arrow and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

Answer: First of all, we would like to explain why we didn't cover all the functions of the skeleton on this diagram. We wanted to keep it simple and concise when it came down about making our program understandable, and we thought that it is not needed to understand all the methods in order to understand the program itself. For this reasoning, only the most core methods are covered. This way, the reader will have a quick understanding of the functioning upon checking the diagram, and if he or she wants a deeper knowledge about the code, then we've developed an append with a detailed explanation of every function at the very bottom of this report.



The first menu gets the game started. HanoiGame then initializes the data structures such as an array for the states of the optimal solution or the towers. Then it calls the method Solve, which is the responsible of solving the game, storing the solution and reinitializing the game afterwards. Upon this process, the function solve_rec (The square inside HanoiGame) is the one that gets called recursively, and the function move in the inside is the responsible of doing the movements. The function Move interacts with the class tower providing the tower of origin in order to obtain the disk that it's moving (pop disc method), and also using the method push_disc to put the disk on the tower of destiny. After the move is done, the function solve rec creates a state of the current position (current state) and it appends to the array of states of the optimal solution.

This process will be repeated until the game is solved, and then the function Start Game will restart the game to allow the human to play and indicate the system that the game has been solved (through a Boolean). Then is when the second menu gets into play. The user will interact with the same functions that HanoiGame used to solve the game, but then the

parameters will be slightly changed. For instance, if the user calls the move function, he will be able to choose the source and destiny of the disk. However, once the disk is moved the state will be saved as a Current State as well. That state won't get appended to the array and the system will check only if the game is finished with the proper function.

Show current state is simple to understand, and it only communicates the desire to print the state to the State class. Show optimal solution does the same than show current state, but with the array of the optimal solution instead with the current state. Finally, show hint and restart the game are features that we added in a optionally way. Restart the game simply calls the same function than solve_rec uses to restart the game and it allows the user to start again at will. Show hint compares the current state with the array of the optimal solution and if we are on it, then it tells the user the next destiny tower in order to finish the game.

3. Code Implementation

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.

Answer: The first thing we will discuss is the relationship between the classes. The main class is HanoiGame. It is the one that is in charge of making the movements, storing information, solving the game ... In short, it is the brain. The HanoiGame class uses the Tower class to store the discs inside of the towers, as well as to make disk movements. Likewise, HanoiGame uses the State class to store each of the moves each time it makes a move. We could say that a State is a photograph of a play that is frozen in an album that HanoiGame is making movements, updating the information and turning the page in each iteration to complete the game. Finally, the main file is responsible for the interaction with the player and the control of the input.

One of the first decisions we had to make is about how to implement human moves. These do not have to be kept in a list so, really, they do not need to be in a state. One option was for human moves to only change the .towers attribute of HanoiGame, but in the end we came to the conclusion that, both by a reader's understanding and by internal coherence and simplicity, it was better to create states after each move, human or computer. So, we created a Boolean control (HanoiGame.solved) that was responsible of filtering whether it was to create a conventional state (with all its metadata) or a "human" state, where only the number of game disks and towers mattered, leaving all other attributes at 0. This facilitated the implementation of various functions, like trivial cases (as the function get_current_state) or to print on the screen the header of the play or not depending on whether it was moved from the solution or by a human.

We also decided that whenever a piece of data was needed that was provided by a function, this function would be called instead of repeating a line of code. This may seem logical and necessary, but we experience a considerable loss of efficiency in speed of resolution of cases with large number of disks. By going into context, we are talking about, for example, using the is_empty function (of the Towers class) instead of using the comparative

self.discs == [].

Finally, we decided that the entire program should be prepared to work with more than three towers. This is because we thought it was a good habit and, if that option was implemented in the future, just by changing the recursive solution function to adapt it to the number of towers, it would work. In addition, by conventionalism with the tests, we decided that all the indexes needed to retrieve list elements were from 0 to len (list) -1 instead of 1 to len (list), which in our opinion is not very user-friendly.

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.

Answer: The program does everything requested in the assignment. Moreover, the code passes all tests provided.

3.4 - Any additional remarks you feel useful to provide. Be concise.

Answer: One of the remarks that are needed to be done is about the towers' implementation into the state Class. We think that there are two cleaner ways to do it. The first one was importing the Tower class into the State class, and then creating three new towers and adding the discs to the new towers with the method as_list. The second way is importing the copy library and use the function deepcopy, that allows to copy an element of a class with a new reference. Instead, the program was designed to redefine the towers as an array of discs rather than elements of the tower class. Due to that, when you want to call the discs inside the tower in the HanoiGame class, you would say something like tower1.discs. If you wanted to call the same discs in the State class, you would say tower1 instead (because the element is the array itself) That is confusing, and we don't understand why it was meant like this. However, we decided that the tests were the ones who decided how things are done, so we implemented towers as arrays of discs into the State class, and we tried to be careful when calling it.

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.

Answer: Our first thought when we started this project was that something was wrong. Maybe we didn't pay enough attention in the theory classes, or maybe it was a 2nd or 3rd course class assignment that accidentally got mixed up in the educational guide. We had absolutely no clue about where to start with. But then everything changed when we came to the first afternoon class, the very next day of the assignment. We received the guidelines that we needed. I think that if we learned something valuable doing this project is, as we say in Spanish, "a sacarnos las castañas del fuego". We learned how to use the debugger, how to make profit out of an IDE, how classes work and interact with each other, how to implement methods and a lot of more useful information all by ourselves. We spent every single afternoon of an entire week with this assignment, either in support afternoon classes or in the library, and that made a click on us. We closed the gap between "Introducció a la programació" and EDA by brute force, and now we can say that we feel prepared for next assignments and challenges.

Our main struggles came from two different sources. First, the definition of towers inside the state class, as we explained in the 3.4 section. Second, the print functions and the print tests. We made several unsuccessful tries before making an auxiliary function that discomposed the strings into arrays, and then checked one by one every single position of the solution to find the mistakes. It was a bit oppressive but finding what was wrong was rewarding.

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.

Answer: We think that the difficult level was hard but necessary. Although it gave us some headaches, it was very rewarding to understand and make it work. One thing that we think it could be done in a different way is that we didn't take class (python classes) theory until two weeks before the delivery date. That made our start rough. Same with the IDE, we think we should learn how it works the very first day of the subject.

5. Submission Instructions

1. Indicate group identifier, the nia of each group members and name of the professor practice class in all code files and text files submitted.
2. Name all your document files (but not the code files) with a prefix:
   EDA1-P1-G<code_group>.
3. Create a directory called EDA1-P1-G<code_group> to pack inside your submission. Include in this directory
   a. The report in pdf format named as follows:
      EDA1-P1-G<code_group>-report.pdf.
   b. The tree execution files if included separate to this report.
   c. The design diagram files if in a separate document
4. Create a subdirectory code to put a copy of your Hanoi code. Remember this requires a sub-directory Hanoi to have several .py files for the imports.
5. Compact the whole directory in a zip (or .rar) file named as the main submission directory (EDA1-P1-G<code_group>). Submit this file in the hanoi assignment in aula global before the deadline.

Before Submission

Remember that there are eligibility criteria to fulfill to have the practice accepted for assessment. Make sure that you fulfill these requirements.

Anexo

Clase Hanoi_exception

Se crea una clase para capturar los errores que surjan tanto de errores propios del código (hacer pop en una lista vacía) como para evitar acciones ilegales.

Clase Tower

Nos sirve para almacenar los discos de cada torre y para implementar algunas funciones que se utilizarán a lo largo del proyecto. Por ejemplo, calcular el tamaño de una torre, quitarle un disco y devolverlo, ponerle un disco que entre por parámetro, convertirlo a lista o dibujarla por pantalla.

1. **__init__(self)** inicializa la torre. Se crea una lista vacía en el atributo de discos **self.discs**.
   is_empty (self) devuelve True si una torre está vacía o no y False en caso contrario. La torre está vacía si no tiene discos, si es una lista vacía.

2. **size(self)** devuelve el tamaño de la torre, es decir, el número de discos. Se devuelve la longitud (número de elementos) de la lista correspondiente al atributo self.discs mediante **len(self.discs).**

3. **pop_disc(self)** remueve un disco de la parte superior de la torre y lo devuelve si la torre no está vacía. En caso contrario, lanza la excepción **HanoiException** mostrando en pantalla un mensaje diciendo que la torre está vacía.

4. **push_disc(self, disc)** agrega un disco, recibido como parámetro, a la parte superior de la torre. Provoca una **HanoiException** si el disco es más grande que el disco en la parte superior de la torre. En primer lugar, miramos si la torre está vacía. En caso de que eso sea cierto, simplemente se añade el disco a esa torre mediante **.append(disc)** a la lista de discos de la torre. En caso de que la torre contenga discos, se mira si el ultimo disco de la torre es mayor que el disco a agregar. Es decir, se mira si el elemento que ocupa la posición -1 en la lista de los discos es mayor que el disco a agregar. Si el ultimo disco de la torre es mayor, se agrega a lista de discos de la torre el nuevo disco. En caso contrario, se lanza la excepción **HanoiException** mostrando en pantalla un mensaje diciendo que el movimiento no es válido.

5. **as_list (self)** devuelve los discos de la torre como una nueva lista. Por lo tanto, devuelve una copia de la lista que es la representación interna de la torre.

6. **__repr__(self)** devuelve una cadena (string) con la representación interna de la torre. Se devuelve **str(self.discs),** una cadena con los discos presentes en la torre.

7. **__str __ (self)** devuelve una cadena (string) con la representación del estado en el formato solicitado.

```
def __str__(self):
    """
    Returns a string with the representation of the state in the requested format.
    Llama un str
    :return: A string with the representation of the state in the requested format
    """
    impreso = ""
    for level in reversed(range(len(self.discs))):
        impreso+=(self.discs[0]-self.discs[level])*"."+self.discs[level]*"#"+"|"+self.discs[level]*"#"+(self.discs[0]-self.discs[level])*"."+"\n"
    return str(impreso)
```

En primer lugar, llamamos a un string vacío al que denominamos impreso y procedemos a crear la representación. Usamos el bucle for para recorrer los diferentes niveles de la torre. Se itera dependiendo de la cantidad de discos. Se recorre en el orden inverso para ir de arriba abajo, ya que está es la dirección del pintado.

Se añade a la cadena impreso un número de "." que corresponde al disco que se encuentra en la primera posición menos el disco que se encuentra en la posición del nivel. A continuación, se añade una cantidad de "#" que corresponde al disco que se encuentra en la posición del nivel. Después, se añade el símbolo "|" y se añaden, de la misma forma que anteriormente mencionada, "#" y ".", respectivamente. Finalmente, se añade un salto de línea, "\n".

Por ejemplo, si
self.discs=[3,2,1],
len(self.discs)=3,
reversed(range(len(self.discs)))=(2,1,0).

En la primera iteración level sería igual a 2. Entonces se añadiría primeramente dos "." (self.discs[0](=3) – self.discs[2](=1)=2), a continuación un "#" (self.discs[2]=1), un "|" y se añaden mediante la misma ecuación, primeramente un "#" y seguidamente dos "." Después se realizaría el mismo proceso pero con las respectivas iteraciones, en este caso dos más.

El cuerpo del bucle se realiza repetidas veces dependiendo del número de elementos del elemento que se recorre variando el valor del nivel y, consecuentemente, la representación. Se devuelve la cadena impreso que contiene la representación del estado.

Clase estado

La clase estado es la que se encarga de aglomerar el estado de las torres con sus metadatos.

En primer lugar, importamos **HanoiException** del módulo **.hanoi_exception** para cuando sea necesario lanzar una excepción.
Creamos la clase **State** cuya función es almacenar y gestionar los estados del juego y consta de diversas funciones:

1. **__init__(self, move_id, depth, moved_disc, source, target, towers, n_discs)** inicializa un estado con toda la información necesaria para representarlo en el formato solicitado. Recibe diversos elementos como parámetros:
   - **move_id** sirve para identificar el movimiento, que es el número de paso.
   - **depth** es la profundidad de recursión a la que se genera el estado.
   - **moved_disc** es el disco movido para llegar al estado, definido por su tamaño.
   - **source** es la torre des de la cual se mueve el disco.
   - **target** es la torre a la cual se mueve el disco.
   - **towers** son las torres del juego (los discos).
   - **n_discs** es el número de discos.

   En esta función inicializamos las variables con los parámetros recibidos en los atributos correspondientes.

2. **get_tower(self, idx)** devuelve la torre correspondiente al idx, índice de la torre, recibido como parámetro. Miramos si el índice es menor o igual que el número de torres. Si es así, devolvemos **self.towers[idx].discs**, la torre correspondiente al índice. Ponemos el .towers para devolver los discos y así pasar satisfactoriamente el test 7. Para ajustarnos a lo que se pide pasamos el array de los discos de dentro de la torre. En caso de que el índice sea mayor a la longitud de la torre, se lanza la excepción **HanoiException** con un mensaje indicando la cantidad de torres del juego.

3. **__repr__(self)** devuelve una cadena con la representación interna del estado. Este método representa la información en un formato diferente al solicitado. Se devuelve un string al que primero asignamos una cadena que contiene los datos de la cabecera (move id, rec depth, last move, disk from, to) seguido de un saltador de línea (\n) y a continuación le añadimos las torres. Mediante %s insertamos valores a la cadena. El %s se reemplaza por lo que se pasa a la cadena después del símbolo %, en ambos casos son remplazados por múltiples elementos dentro de una tupla de forma ordenada. Sumamos una unidad al source y al target para que encaje con el número de torres. Cuando se añaden las torres, en la tupla que sigue al símbolo %, se hace uso de repr() en los elementos ya que devuelve una cadena representativa imprimible del objeto dado.

4. **__str__(self)**

```
def __str__(self):
    """
    Returns a string with the representation of the state in the requested format.

    :return: A string with the representation of the state in the requested format
    """
    if self.move_id != 0:
        header = """\nMove id %d Rec Depth %d\nLast move: %d Disk, from %d to %d\n""" % (self.move_id,self.depth,self.moved_disc,self.source+1,self.target+1)

    else: #La cabecera no se pone en el mov_id = 0 ni en jugadas humanas
        header = "\n"

    torres_dibujo= ""
    maxdisc = self.n_discs
    for nivel in reversed(range(maxdisc)):
        for tower in self.towers:
            try:
                torres_dibujo += ((maxdisc-tower.discs[nivel])*".")+(tower.discs[nivel]*"#")+"|"+(tower.discs[nivel]*"#")+((maxdisc-tower.discs[nivel])*".")+" "
            except IndexError:
                torres_dibujo += maxdisc*"."+"|"+maxdisc*"."+" "

        torres_dibujo += "\n"

    torres_titulo= ""
    caracs_titulos = 7
    caracs_base = maxdisc*2+1 #Dos bandas a cada torre mas el palo del medio
    espacios= int((caracs_base-caracs_titulos)/2)*" "
    for tower in range(len(self.towers)):
        torres_titulo += """%sTower %d%s """ % (espacios,tower+1,espacios)
    torres_titulo+="\n"

    dibujo_final= header+torres_dibujo+torres_titulo
    return str(dibujo_final)
```

En primer lugar, creamos la cabecera. Si el turno de movimiento es el 0 o si se trata de una jugada humana, la cabecera no la escribimos (como ya hemos visto, si es jugada humana el move_id del estado siempre es igual a 0).

Seguidamente, crearemos el string de las torres. Utilizamos un doble bucle para recorrer, primero horizontal y después verticalmente, cada una de las torres y cada uno de sus niveles. Así, en primer lugar, se dibuja el primer nivel (empezando desde arriba) de cada torre y, cuando acaba se produce un salto de línea y se dibuja el siguiente nivel. Para cada nivel de cada torre, se dibuja el símbolo "." por la diferencia de los discos máximos de la torre y los discos presentes en el nivel. A continuación, se dibuja un "#" por cada disco presente en el nivel de la torre. Después, se añade el símbolo "|" y se añaden, por simetría, "#" y "." respectivamente. En el caso de que no encontremos la posición dentro

de la lista debido a que hay listas de torre más largas que otras (las que tienen más discos), entraremos en el except IndexError. En ese caso, dibujaremos solo puntos en base a la anchura de la torre, definida por el disco más ancho de la partida.

Después, crearemos el string de los títulos de las torres. El cálculo de los espacios que dejamos a cada lado de la torre es lo más complicado, y se obtiene de restar a los caracteres de la anchura de la base de cada torre el número de caracteres de los títulos (7), y dividir ese número entre dos.

Por último, sumamos las tres cadenas y hacemos el return.

Clase Hanoi

En primer lugar, importamos **logging**, **HanoiException** del módulo **.hanoi_exception, State** del módulo .**state** y **Tower** de **.tower**.
Creamos la clase **HanoiGame** que es la clase principal para la gestión de las estructuras de datos y movimientos del juego. Consta de diversas funciones:

1. **__init __ (self, n_discs, n_towers = 3)** inicializa el juego con n_discs y n_towers, que por defecto es 3, y son recibidos como parámetros. El juego se puede resolver y almacenar para consultar. Provoca una excepción HanoiException si n_discs es negativo o n_towers es menor que 3.

   Primeramente, comprobamos los parámetros y lanzamos la excepción correspondiente si no son válidos. A continuación, inicializamos los atributos de la estructura. Para ello creamos una matriz dentro de la clase de estados que contenga los diferentes estados. También creamos un atributo que contiene los discos totales, otro con el número de torres y un booleano por si se ha resuelto recursivamente ya, inicialmente tomado como falso. Después, iniciamos las torres y las llenamos con ayuda de .append() y finalmente, resolvemos y guardamos la solución óptima.

2. **start_game(self)** inicia las torres y llena la primera. También crea el estado actual.
   Creamos un array que guarde el estado actual de las torres. Añadimos las variables de la clase torre que depende del número de torres. Ponemos los discos en la primera torre. Inicializamos el estado de la partida y lo guardamos en el primer estado.

3. **get_state(self, step)** devuelve el estado en el paso solicitado en la solución óptima. Lanza una HanoiException si el índice de pasos, recibido como parámetro, es negativo o mayor que el total de estados en la solución óptima.
   Devolvemos el estado en el paso solicitado. Si se produce el error IndexError, lanzamos una HanoiException dependiendo de el caso del error, si el índice de pasos es negativo o por si es mayor que el total de estados para advertir al usuario.

4. **get_n_discs(self)** devuelve el número de discos del juego.

5. **get_n_towers(self)** devuelve el número de torres del juego.

6. **get_n_states(self)** devuelve el número de estados de la solución óptima.
   Es el tamaño de la estructura utilizada para almacenar los estados de la solución óptima por eso devolvemos **len(self.states)**.

7. **pistas(self)** devuelve la torre que debe recibir un disco de acuerdo con la solución óptima. Devuelve **False** si **current_state** no coincide con ninguno de los pasos en la solución óptima.

8. **move (self, source, target, move_id = None, depth = None)** mueve un disco de la torre de origen a la torre de destino. Provoca una excepción **HanoiException** si el origen y el destino son los mismos o si el movimiento no es válido. El movimiento no es válido si el disco movido es más grande que el último disco en la torre de destino. También devuelve el nuevo estado generado por el movimiento. La función recibe diversos elementos como parámetros:
    - **source**: torre desde la que se va a mover un disco.
    - **target**: torre a la que se va a mover un disco.
    - **move_id**: identificador del movimiento. Útil como información para el estado óptimo.
    - **depth**: profundidad de la llamada de recursión. Útil como información para el estado óptimo.

```
def move(self, source, target, move_id=None, depth=None):
    """
    Moves a disk from source tower to target tower.
    Raises a HanoiException if source and target are the same or if the move is invalid (the disk moved is bigger
    than the last disk in the target tower, the source tower is empty...)

    :param source: Tower from which a disk is going to be moved.
    :param target: Tower to which a disk is going to be moved.
    :param move_id: Identifier of the movement. Useful as information for the optimal state.
    :param depth: Depth of the recursion call. Useful as information for the optimal state.
    :return: The new state generated by the move.
    """


    disc = self.towers[source].pop_disc()

    try:
        self.towers[target].push_disc(disc)

    except:
        self.towers[source].push_disc(disc)
        raise HanoiException("A disc can not go on top of a smaller disk!")

    if self.solved == False:
        self.current_state = State(move_id,depth,disc,source,target,self.towers,self.n_discs)

    else:
        self.current_state = State(0,0,0,0,0,self.towers,self.n_discs)

    return self.current_state
```

En primer lugar, creamos un parámetro denominado disc que copia la última posición del self.towers[source]. Colocamos el disco en la torre de destino y si se produce una excepción lo devolvemos a su posición inicial y lanzamos una excepción indicando que un disco no puede ir encima de uno de menor tamaño. Miramos si el booleano creado self.solved sigue siendo False. Si es así, significa que es la máquina quien realiza la acción y no el jugador y guardamos el estado con los datos correspondientes. Si self.solved no es False, entonces guardamos el estado con move_id, depth, moved_disc, source y target iguales a 0. Cabe recalcar su importancia ya que al ser move_id=0 no se dibujará la cabecera en la representación del estado. Finalmente devolvemos el estado actual.

9. **_solve(self)** genera y almacena la solución óptima, reinicializando las torres posteriormente. Creamos, por convención con los test, tres variables destino, auxiliar y fuente. Llamamos a la función **_solve_rec()** con las variables creadas, llamamos a la función **start_game()** y cambiamos el booleano solved a True ya que se ha resuelto recursivamente ya.
_solve_rec(self, n_discs, source, target, aux, depth=0) es una llamada recursiva para resolver el juego de hanoi de manera óptima. Recibe ciertos parámetros:
    - **n_discs**: número de discos a mover.
    - **source**: torre desde la cual se moverá un disco.
    - **target**: torre a la que se va a mover un disco.

- **aux**: torre para ser utilizada como auxiliar.
- **depth**: profundidad de la llamada de recursión. Útil como información para el estado óptimo.

```
def _solve_rec(self, n_discs, source, target, aux, depth=0):
    """
    Recursive call to solve the hanoi game optimally.

    :param n_discs: Number of disks to be moved.
    :param source: Tower from which a disk is going to be moved.
    :param target: Tower to which a disk is going to be moved.
    :param aux: Tower to be used as auxiliary.
    :param depth: Depth of the recursion call. Useful as information for the optimal state.
    """
    if n_discs > 0:
        self._solve_rec(n_discs-1,source,aux,target,depth+1)

        state = self.move(source,target,move_id=len(self.states),depth=depth+1)

        self.states.append(state)

        self._solve_rec(n_discs-1,aux,target,source,depth+1)
```

Hemos creado un bucle. En primer lugar, se mira si el número de discos es mayor a 0. Volvemos a llamar a la misma función, pero siendo ahora el destino la torre auxiliar y la torre auxiliar la de destino. También disminuyendo el número de discos una unidad y aumentando la profundidad en una unidad. A continuación, creamos una variable state donde se llama a la función move y se mueve el disco de la torre fuente a la torre de destino. El move_id es la longitud de los estados (el número de estados) y la profundidad la hemos aumentado una unidad. Luego añadimos el estado a la lista de estados y volvemos a llamar a la función _solve_rec() de la misma forma que antes pero cambiando la torre auxiliar por la de origen.
print_optimal_state(self, step) imprime el estado óptimo en el paso seleccionado en el formato requerido. Parámetro:

step: paso índice de la solución óptima.
Miramos si el paso índice es mayor que la longitud de estado o si es negativo. Si se cumple, se lanza una excepción indicando que el paso no existe. Finalmente, se devuelve el estado óptimo en el paso seleccionado.
print_optimal_solution(self) imprime todos los estados de la solución óptima en el formato requerido. Imprimimos cada estado de la lista de estados.
is_finished(self) comprueba si el juego interactivo está terminado, devuelve True si está terminado, False de lo contrario.
Si los discos de la última torre son iguales a los discos iniciales del juego entonces se devuelve True.

10. **get_current_state(self)** devuelve el estado actual del juego.

11. **__repr__(self)** devuelve una cadena con la representación interna del juego.

12. **__str__(self)** devuelve una cadena con la representación del estado actual del juego en el formato solicitado.

Main

En primer lugar, importamos **HanoiException** del módulo **hanoi.hanoi_exception** y **HanoiGame** del módulo **hanoi.hanoi**. Tenemos varias funciones:

1. **get_int_value(message, range_start, range_end)** envía al usuario el mensaje de entrada y comprueba que los datos de entrada son un número entero entre range_start y range_end. Devuelve el valor si es correcto, de lo contrario, ninguno. Parámetros:
   message: mensaje para el usuario que solicita alguna entrada de entero
   range_start: primer valor en el rango válido.
   range_end: primer valor fuera del rango válido.

Creamos la variable value igual a None y entramos dentro de un while creando un bucle. Hacemos que se introduzca el mensaje deseado y se lo otorgamos a value. Si se trata de un entero y se encuentra entre el rango de inicio y el rango de final, se devuelve el valor. Si no se encuentra en el rango se lanza un mensaje indicando que debe estar en el rango. Si no se trata de un entero entonces se produce el error ValueError y se lanza un mensaje indicando de que se debe tratar de un valor válido.

do_move(hanoi_game) pregunta al usuario por un movimiento y devuelve True si el movimiento lleva a un estado final. Parámetro:

hanoi_game: la instancia del juego que está utilizando el programa.

Entramos en un bucle usando done=False. Llamamos a la función get_int_value() con un mensaje indicando de que torre desea mover con rango de entrada 0 y rango de salida 3. Si el valor introducido por el usuario no es válido se cancela el movimiento. Si es válido se vuelve a llamar a la función get_int_value() con un mensaje indicando a que torre se desea mover con el mismo rango de entrada y de salida que anteriormente. Si el valor introducido no es válido se cancela el movimiento. Si es válido se produce el movimiento llamando a la función hanoi_game.move(). Restamos una unidad a source_tower y target_tower para que coincidan con el string de torres (0,1,2). Si se introduce 0 se cancela. Cambiamos done a True para posteriormente salir del bucle. Si se produce la excepción Hanoi se muestra por pantalla la excepción correspondiente. Se sale del bucle y se devuelve si el juego ha terminado o no llamando a la función is_finished() de hanoy.py.

2. **game_options** no es una función, es una variable que contiene texto mostrando las opciones que tiene el usuario a la hora de jugar.
3. **play_game()** es el bucle principal para el juego interactivo.

Se le pregunta cuantos discos desea en el juego. El mínimo son 2 y el máximo 10. Para ello llamamos a get_int_value(). Luego llamaos a la función HanoiGame con el número de discos introducidos por el usuario como parámetro y la mostramos por pantalla. Entramos dentro de un bucle y mostramos las opciones de juego con print(game_options). Se pide al usuario que seleccione la opción que desea realizar. Si la opción elegida es la 0 se sale del bucle y se acaba el juego volviendo al menú inicial. Si se elige la 1 se llama a la función de hacer un movimiento, se muestra el estado actual y se mira si el juego ha finalizado. Si la opción introducida es la 2 se muestra el estado. Si es la 3 se llama a la función de solución óptima. Si es la 4 se llama a la función start_game() para que se vuelva a iniciar el juego. Si es la 5 se llama a la función de pistas y si es diferente de False le recomienda mover un disco de una torre a otra. Si es igual a False le dice que algún movimiento no ha sido correcto. En caso de elegir una opción de las opciones de juego no válida le dirá que no es una opción válida y que lo vuelva a intentar.

4. **header** no es una función, es una variable que contiene texto mostrando la cabecera inicial del juego.
5. **main_options** no es una función, es una variable que contiene texto mostrando las opciones del menú inicial.

6. **main_help** no es una función, es una variable que contiene texto mostrando una ayuda inicial para mostrar las restricciones.
7. **show_menu** bucle principal para el menú principal.

Mostramos la cabecera, entramos en un bucle y se muestran las opciones del menú principal. Se le pide que opción desea realizar. Si es la 0 se sale del bucle. Si es la 1 se llama a la función play_game() y se juega el juego. Si es la 2 se muestra la ayuda.
Al inicio se muestra el menú principal y se inicia el juego.