Semester Project

Algorithms and Data Structures

Nova Information Management School

Bachelor's Degree in Data Science

2021/2022

Afonso Cadete | 20211519

Marcelo Júnior | 20211677

Rita Falcão Centeno | 20211579

Introduction:

This project was given under the context of the Algorithms and Data Structures course, and consists in the application of the concepts taught throughout the semester in three different practical exercises. This project will challenge our problem solving capabilities and encourage us to work as a team.

NOTE:

Throughout this project, in order to make the time complexity calculations cleaner and easier to understand, we chose not to include any reference to the time complexity of any line in which the value was equal to 1.

First Exercise:

During the semester we have worked with several sorting algorithms, and the aim of this exercise was to compare their running time when sorting different length lists.

All five sorting algorithms taught in classes – Insertion Sort, Selection Sort, Bubble Sort, Merge Sort and Quick Sort – were used in the code. Inspired by the last slide of the PowerPoint called *Sorting Algorithms*, available in the subject's Moodle area, we decided to search about the remaining three sorting methods presented in the table – Radix Sort, Shell Sort and Heap Sort –, and add them to our solution.

We decided to use classes in this exercise in order to structure our code in a simpler and organized way. Therefore, we started by creating a class called “Sorts” where we defined every sorting algorithm.

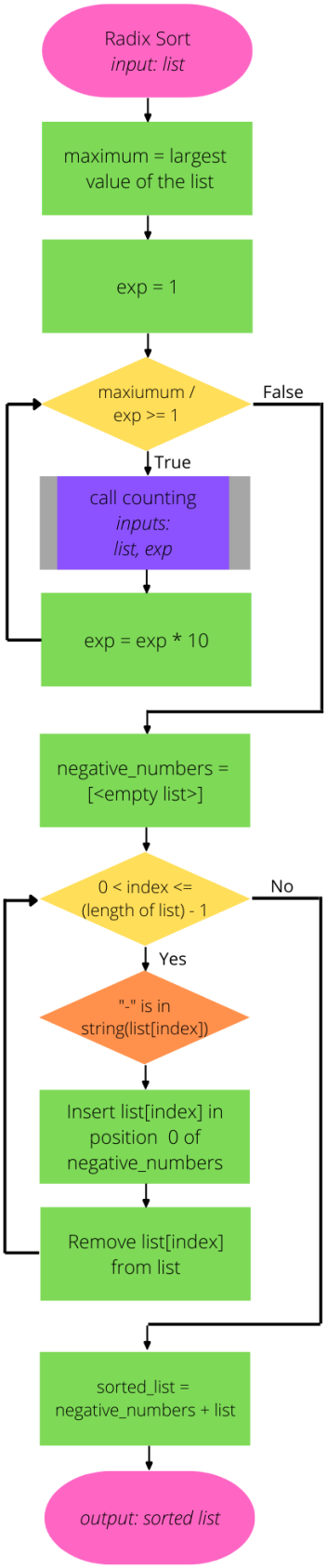
In the aforementioned class, an argument structure was defined for some sorting methods, where a default value of *None* was attributed to the *list_to_sort* argument, creating for this default case a copy of the list that was used when calling the class (*test_list*). This way, for every one of the prior methods, those being the main sorting algorithms, a copy of *test_list* was sorted, preserving the original one for further instances.

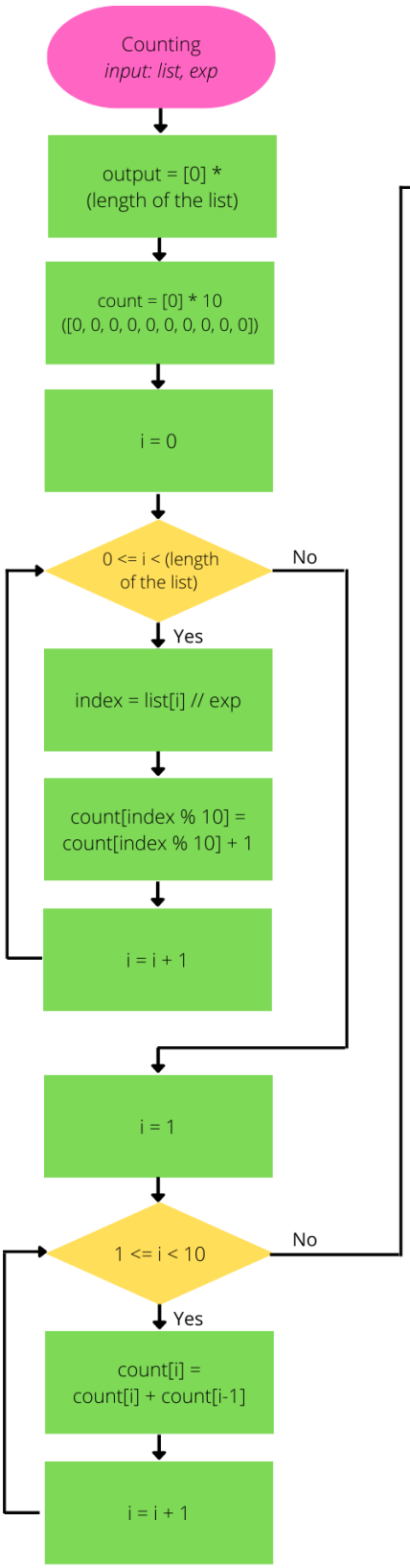
When the code is run, the user has to insert values for three different parameters: the initial size of the list, the incremental size of the list, and the number of iterations they want to be executed. Then, a random list is created with the initial size inputted. Next, the list is sorted by every sorting algorithm. Meanwhile, the time that each method takes to sort the list’s values is being measured – using the time package –, and stored in a table/matrix – built using the tabulate package. After that, the following iteration begins. However, this time, the number of elements defined by the incremental size input are added to the random list. Logically, the loop ends when all user-defined iterations are done. Finally, the table/matrix is printed.

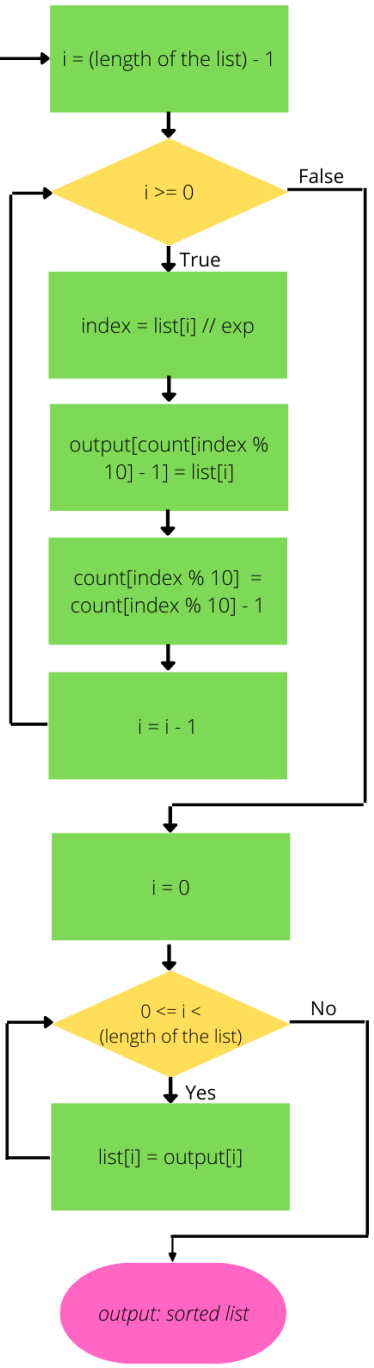
Finally, for the program to be able to run lists with a larger number of values without giving an error, we imported the package sys.

As the first five algorithms were taught in class, we consider that there is no need to explain how they work. Regarding the remaining three, we built flowcharts to help in their analysis. Next, we will explain how these functions work in the following order: Radix Sort (and Counting Sort); Shell Sort; Heap Sort (and Heapify),

Radix Sort:


Radix Sort
input: list
maximum = largest value of the list
exp = 1
maxiumum / exp >= 1
False
True
call counting
inputs:
list, exp
exp = exp * 10
negative_numbers = [<empty list>]
0 < index <= (length of list) - 1
No
Yes
"-" is in string(list[index])
Insert list[index] in position 0 of negative_numbers
Remove list[index] from list
sorted_list = negative_numbers + list
output: sorted list


Counting
input: list, exp
output = [0] * (length of the list)
count = [0] * 10
([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
i = 0
0 <= i < (length of the list)
No
Yes
index = list[i] // exp
count[index % 10] = count[index % 10] + 1
i = i + 1
i = 1
1 <= i < 10
No
Yes
count[i] = count[i] + count[i-1]
i = i + 1

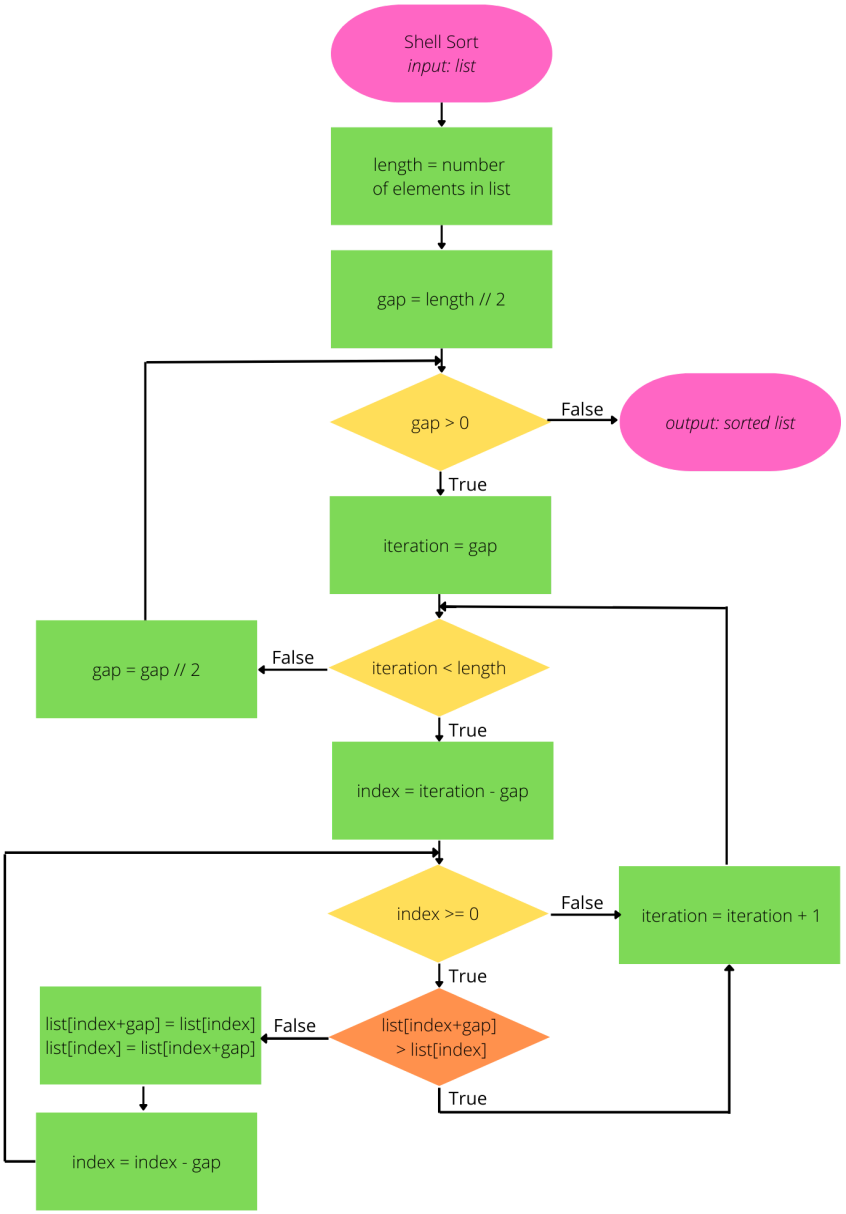

i = (length of the list) - 1
i >= 0
False
True
index = list[i] // exp
output[count[index % 10] - 1] = list[i]
count[index % 10] = count[index % 10] - 1
i = i - 1
i = 0
0 <= i < (length of the list)
No
Yes
list[i] = output[i]
output: sorted list

Our first addition to exercise 1 was Radix Sort, a non-comparative sorting algorithm that distributes a list's elements in buckets based on their *radix* (or base), which is the number of digits used to represent a number. In the decimal system, that value is 10, that is, all the numbers from 0 to 9.

It works digit by digit, sorting from least significant digit to the most significant one, using the previously mentioned Counting Sort as a subroutine. Although negative numbers would normally be put in tail of the list, by a adding a piece of code to this method, we were able to properly put them at the head of the tail, by inserting the elements from right to left in it and removing the element in their previous location (that being the original number).

As for Counting Sort, it creates a vector that is used to count the number of times a number, from 0 to 9, appears, each having their own position, and an output vector that will be used later.. Each occurrence is counted in the vector position of the last digit (or the remainder of the division of the number by 10). After that, the sum prior to each element is added to it. We then build the output list, corresponding to each last digit the element in the counting vector that has it as its index, subtracting 1 from the value of the count of it. Finally, we attribute the values in the output vector to the list that was used to call this method. At the end of the loop for counting, Radix Sort will have a more or less sorted list, at last using the created loop for negative numbers to have it perfectly sorted.

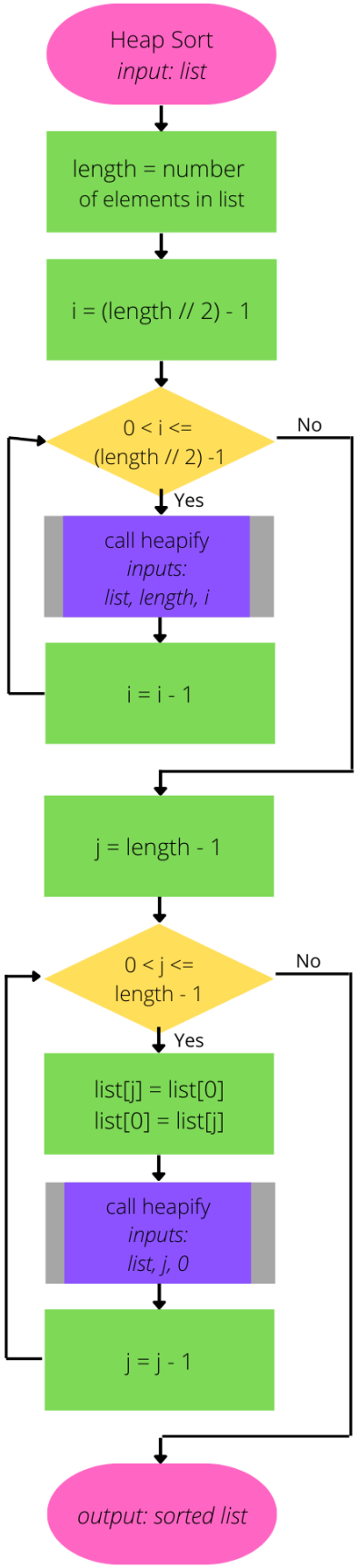
Shell Sort:

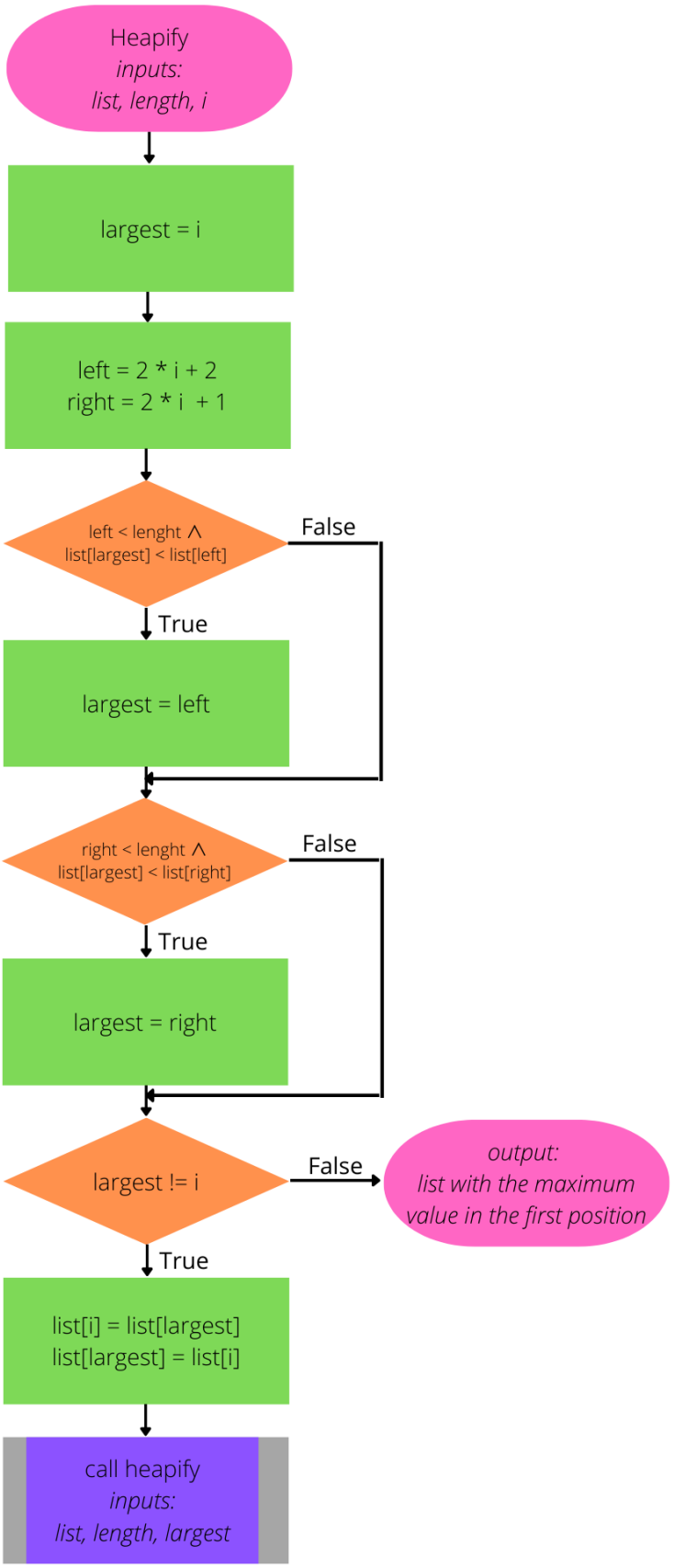


Of the three new algorithms, the only one that does not need to call another function is Shell Sort. It contains three while loops inserted into each other, along with three dependent variables.

In short, this algorithm works by gaps, being the first one the integer value of half of the list length. Each gap is ran as many times as there are elements in the list, and, during these iterations, the element at position *index* is compared with element at position *index*+*gap*. After all possible combinations by the gap are evaluated, *gap* becomes the integer value of half of itself. The loop continues until *gap* equals 0.

Heap Sort:


Heap Sort
input: list
length = number of elements in list
i = (length // 2) - 1
0 < i <= (length // 2) -1
No
Yes
call heapify
inputs:
list, length, i
i = i - 1
j = length - 1
0 < j <= length - 1
No
Yes
list[j] = list[0]
list[0] = list[j]
call heapify
inputs:
list, j, 0
j = j - 1
output: sorted list


Heapify
inputs:
list, length, i
largest = i
left = 2 * i + 2
right = 2 * i + 1
left < lenght ∧ list[largest] < list[left]
False
True
largest = left
right < lenght ∧ list[largest] < list[right]
False
True
largest = right
largest != i
False
output:
list with the maximum value in the first position
True
list[i] = list[largest]
list[largest] = list[i]
call heapify
inputs:
list, length, largest

The Heap Sort algorithm is dependent on the Heapify function.

Beyond the list, its length and an index – which will be considered the initial largest value – are inputted in *heapify*. This auxiliary function starts by defining two variables – at the first iteration, when the list has odd elements, *left* and *right* will be the last two positions of the list, in even numbers case, *left* will be the last position and *right* will be out of range. After that, the values in these positions and the actual *largest* are compared, and the biggest one keeps/becomes *largest*. If the *largest* variable does not change, the function ends. Else the elements in position *largest* and *i* are swapped; in this case, the function is called again in order to try to push another large value next to first position.

In the Heap Sort function, the first loops run from the position at the left of the central element (odd elements)/central left position (even elements) to the first element. It calls *heapify*. At the end of the loop, the largest value from the list will be in the first position.

The second loop serves to swap the first position – where is the maximum – with the last one, and, after that, ignore the maximum until the end because it is already in the right position. Then, through *heapify*, the new maximum is pushed to first position, and the same methodology is repeated until the list is sorted.

In regards to time complexity, to evaluate its value, we will have to know the time complexity of each sorting algorithm we used. The majority of them we learned in class, therefore we will just assume the value given. These are their time complexities: Insertion Sort - $\Theta(n^2)$; Selection Sort - $\Theta(n^2)$; Bubble Sort - $\Theta(n^2)$; Merge Sort - $\Theta(n \log n)$; Quick Sort - $\Theta(n \log n)$.

In what concerns the sorting algorithms we did not learn in class, these are their time complexities: *radix_sort* - $\Theta(n \log n)$; *shell_sort* - $\Theta(n \log n)$; *heap_sort* - $\Theta(n \log n)$.

Analysis of the time complexity of the main code:

```
initial_list_size = int(input("Insert initial size:\t"))
incremental_size = int(input("\nInsert incremental size:\t"))
number_iterations = int(input("\nInsert number of iterations:\t"))
random_list = [randint(-10, 10) for i in range(initial_list_size)] ⟶ n
row_names = [["Algorithm"], ["Insertion sort"], ["Bubble sort"], ["Selection sort"],
             ["Merge sort"], ["Quick sort"], ["Radix sort"], ["Shell sort"], ["Heap sort"]]
col_names = ["Iteration"] + [str(i) for i in range(1, number_iterations + 1)] ⟶ m

# Run all methods and time them.
for i in range(number_iterations): ⟶ m
    print('Random list: ', random_list)
    list_timed = Sorting(random_list)
    size = len(random_list)
    row_names[0].append(str(size))
    start = time.time()
    list_timed.insertion_sort() ⟶ n²
    row_names[1].append(f"{round(time.time() - start, 4)} sec")
    start = time.time()
    list_timed.bubble_sort() ⟶ n²
    row_names[2].append(f"{round(time.time() - start, 4)} sec")
    start = time.time()
    list_timed.selection_sort() ⟶ n²
    row_names[3].append(f"{round(time.time() - start, 4)} sec")
    start = time.time()
    list_timed.merge_sort() ⟶ n log n
    row_names[4].append(f"{round(time.time() - start, 4)} sec")
    start = time.time()
    list_timed.quick_sort(0, size - 1) ⟶ n log n
    row_names[5].append(f"{round(time.time() - start, 4)} sec")
    start = time.time()
    list_timed.radix_sort() ⟶ n log n
    row_names[6].append(f"{round(time.time() - start, 4)} sec")
    start = time.time()
    list_timed.shell_sort() ⟶ n log n
    row_names[7].append(f"{round(time.time() - start, 4)} sec")
    start = time.time()
    list_timed.heap_sort() ⟶ n log n
    row_names[8].append(f"{round(time.time() - start, 4)} sec")
    random_list += [randint(-10, 10) for a in range(incremental_size)] ⟶ n
print("\nRESULTS:\n", tabulate(row_names, headers=col_names))
```

In this part of the code, the time complexity will be the sum of all the time complexities of the functions inside the for loop, times n (the number of times the loop is executed), with n+m (number of times the first two for loops are executed) → $n+m+[n+n+n\times(n^2+n^2+n \log n+n \log n+n \log n+n \log n+n \log n)]$. As m represents a finite number of iterations selected by the user (usually smaller than the number of values the list contains [n]), this means that the time complexity of our code will be $\Theta(n^3)$.

Second Exercise:

This exercise revolves around joining several lists into a single big list, while also applying sorting and shuffling algorithms to certain subparts of the list.

The sorting algorithm we decided to use was the Merge Sort , as we figured it would be the most efficient method considering what the program had to do. Our decision was also based on the results from the previous exercise.

We approached this problem by continuously working with a matrix, and a list. The matrix would enable an easier storage and access to the different sublists inside the main list, making it easy to perform the sorting/shuffling process on its values.

While running the program the function *make_interval* is called which will ask the user to input the new interval's initial and final values. Then a current main list, containing all the values from the interval, is printed and the user is asked if he/she wants to input another list. If the answer is negative, the resulting main list is printed. If the answer is positive the user is asked to input the parameters for a new list, and the values of that new list are added to the main list. This extended list is then entirely sorted and run through a function that removes repeated values. Subsequently, this main list is transformed into a matrix, and each list within the matrix will be either sorted (if its index is odd), or shuffled (if its index is even). Finally, a final list is generated, from the union of all the sublists in the matrix, and then printed. The user is again asked about inputting another interval and the cycle is repeated.

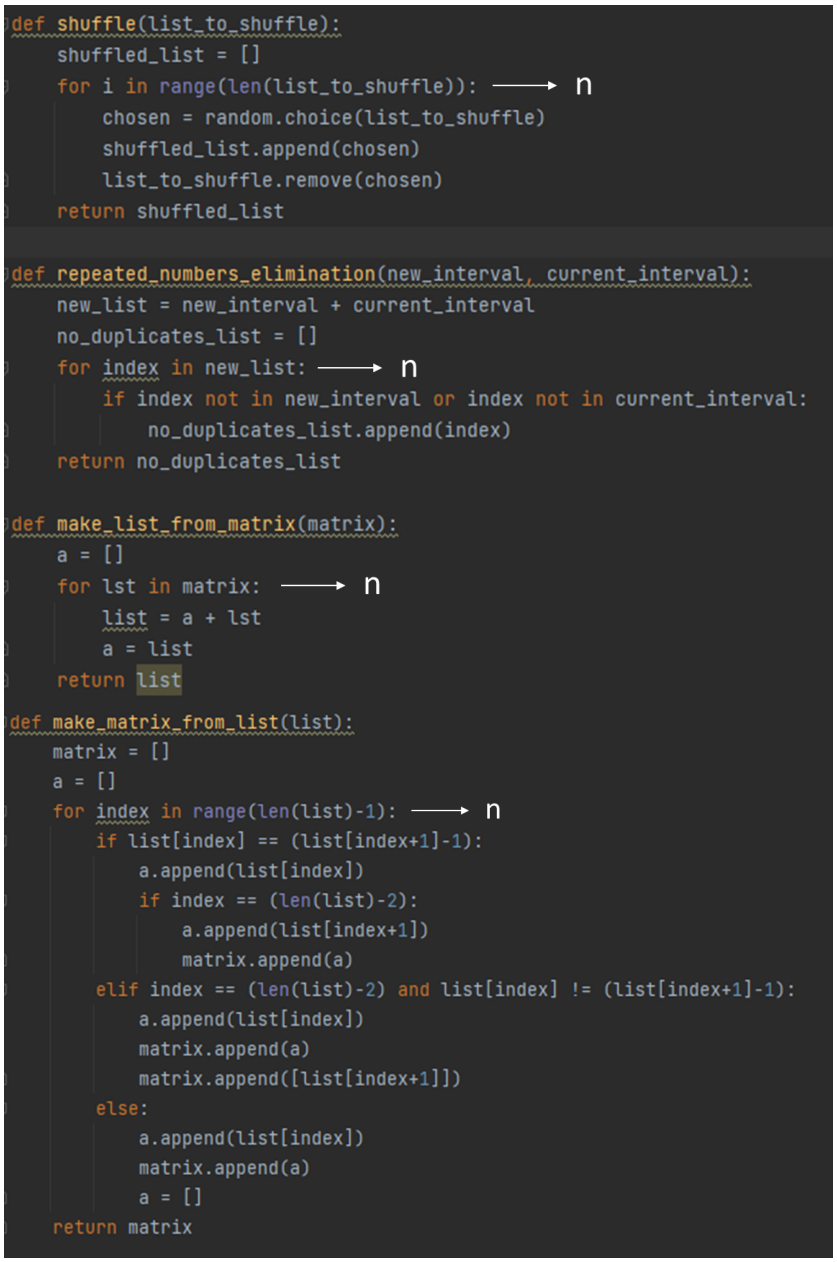
In what concerns the time complexity of this exercise, here are the calculus we made:

```
def make_interval():
    initial_number = int(input("Interval's initial value (-999 to 999):\t"))
    while not (-1000 < initial_number < 1000): ⟶ n
        initial_number = int(input("That wasn't a valid answer. Insert a valid interval's initial value (-999 to 999):\t"))
    final_number = int(input(f"Interval's final value ({initial_number} to 999):\t"))
    while not (-1000 < final_number < 1000): ⟶ n
        final_number = int(input(f"That wasn't a valid answer. Insert a valid interval's final value ({initial_number} to 999):\t"))
    user_interval = [number for number in range(initial_number, final_number + 1)] ⟶ n
    return user_interval
```

In this *make_interval* function, the time complexity will be the sum of the time complexities of the three loops. This means that it will be Θ(n).

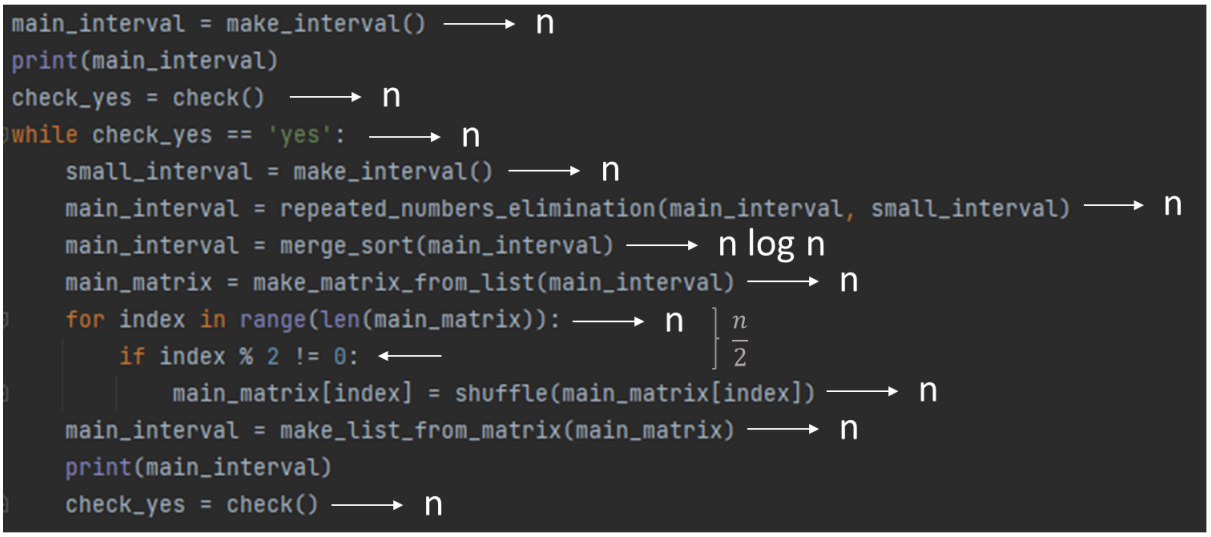
```
def check():
    checking = input("Do you want to insert another interval? (Yes/No)\t")
    while not (checking.lower() == 'yes' or checking.lower() == 'no'): ⟶ n
        checking = input("That wasn't a valid answer. Do you want to insert another interval? (Yes/No)\t")
    return checking.lower()
```

The *check* function's time complexity will only depend on the while loop, which means that the time complexity will be Θ(n).



The *shuffle*, *repeted_numbers_elimination*, *make_list_from_matrix*, and *make_matrix_from_list* functions will have a time complexity of Θ(n), because of the for loops they contain.

The sorting algorithm we used in this exercise was merge sort, therefore, the time complexity considered for the sorting algorithm was Θ(n log n).

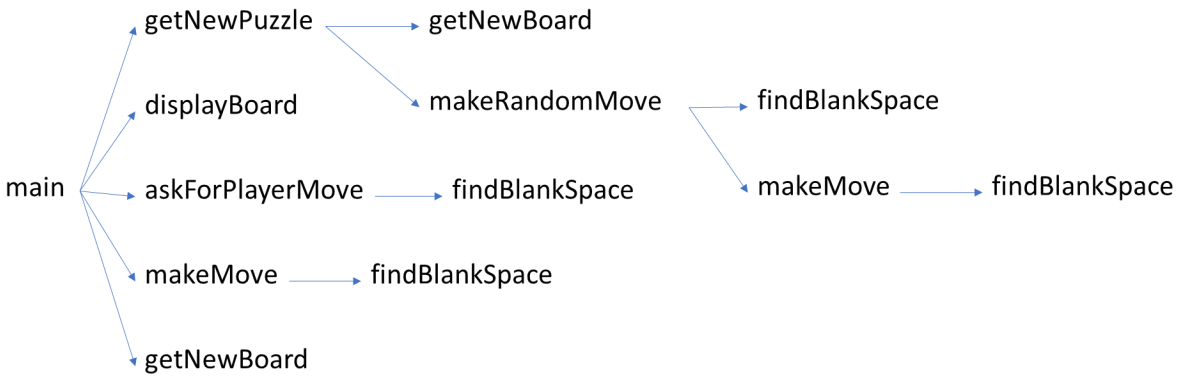


Looking at the main part of the code, we can extract that its time complexity will be of n + n + n (n + logn + n + n × (n) + n) + n. This way, we can conclude that the time complexity of this exercise is $\Theta(n^3)$.

Third Exercise:

Part 1:

The code of this 4 by 4 sliding tile game involves the chained call of several functions, therefore, for a better and clearer analysis of the time complexity, we are going to analyze in the inverse order they are called. Here is the scheme of the order they are called:




```
def findBlankSpace(board):
    """Return an (x, y) tuple of the blank space's location."""
    for x in range(4):  ⟶ n
        for y in range(4):  ⟶ n
            if board[x][y] == '  ':
                return (x, y)
```

As there is a for loop inside another for loop (which only has lines of code with time complexity of 1 inside it), the time complexity of the *findBlankSpace* function will be $\Theta(n^2)$.

```
def makeMove(board, move):
    """Carry out the given move on the given board."""
    # Note: This function assumes that the move is valid.
    bx, by = findBlankSpace(board)  ⟶ n²

    if move == 'W':
        board[bx][by], board[bx][by+1] = board[bx][by+1], board[bx][by]
    elif move == 'A':
        board[bx][by], board[bx+1][by] = board[bx+1][by], board[bx][by]
    elif move == 'S':
        board[bx][by], board[bx][by-1] = board[bx][by-1], board[bx][by]
    elif move == 'D':
        board[bx][by], board[bx-1][by] = board[bx-1][by], board[bx][by]
```

Considering that the *makeMove* function calls the *findBlankSapce* function, and that all the other lines of code have the time complexity of $\Theta(1)$, the time complexity of the *makeMove* function will be $\Theta(n^2)$.

```
def makeRandomMove(board):
    """Perform a slide in a random direction."""
    blankx, blanky = findBlankSpace(board) ⟶ n²
    validMoves = []
    if blanky != 3:
        validMoves.append('W')
    if blankx != 3:
        validMoves.append('A')
    if blanky != 0:
        validMoves.append('S')
    if blankx != 0:
        validMoves.append('D')

    makeMove(board, random.choice(validMoves)) ⟶ n²
```

The time complexity of the function *makeRandomMove* will be the sum of the time complexities of the functions *findBlankSpace* and *makeMove*. Therefore the *makeRandomMove* function time complexity is $\Theta(n^2)$.

```
def getNewBoard():
    """Return a list of lists that represents a new tile puzzle."""
    return [['1 ', '5 ', '9 ', '13'], ['2 ', '6 ', '10', '14'],
            ['3 ', '7 ', '11', '15'], ['4 ', '8 ', '12', BLANK]]
```

As the *getNewBoard* function only consists of the return of a matrix, its time complexity will be $\Theta(1)$.

```
def getNewPuzzle(moves=200):
    """Get a new puzzle by making random slides from a solved state."""
    board = getNewBoard() ⟶ 1

    for i in range(moves): ⟶ n
        makeRandomMove(board) ⟶ n²
    return board
```

The *getNewPuzzle* function's time complexity will depend on the time complexity of the functions *getNewBoard* and *makeRandomMove* (multiplied by n, because of the for loop). This way, the time complexity of *getNewPuzzle* is $\Theta(n^3)$.

```
def askForPlayerMove(board):
    """Let the player select a tile to slide."""
    blankx, blanky = findBlankSpace(board) ——→ n²

    w = 'W' if blanky != 3 else ' '
    a = 'A' if blankx != 3 else ' '
    s = 'S' if blanky != 0 else ' '
    d = 'D' if blankx != 0 else ' '

    while True:  ——→ n
        print('                              ({})'.format(w))
        print('Enter WASD (or QUIT): ({}) ({}) ({})'.format(a, s, d))

        response = input('> ').upper()
        if response == 'QUIT':
            sys.exit()
        if response in (w + a + s + d).replace(' ', ''):
            return response
```

The *askForPlayerMove* time complexity will depend on the time complexities of the function *findBlankSpace* and the while loop. However, as the first one is $\Theta(n^2)$ and the second only $\Theta(n)$, the time complexity of *askForPlayerMove* will be $\Theta(n^2)$.

```
def displayBoard(board):
    """Display the given board on the screen."""
    labels = [board[0][0], board[1][0], board[2][0], board[3][0],
              board[0][1], board[1][1], board[2][1], board[3][1],
              board[0][2], board[1][2], board[2][2], board[3][2],
              board[0][3], board[1][3], board[2][3], board[3][3]]
    boardToDraw = """
+------+------+------+------+
|      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |
+------+------+------+------+
|      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |
+------+------+------+------+
|      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |
+------+------+------+------+
|      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |
+------+------+------+------+
""".format(*labels)
    print(boardToDraw)
```

The display board function, as it only consists of basic operations, will have a time complexity of $\Theta(1)$.

```
def main():
    print('''Sliding Tile Puzzle, by Al Sweigart al@inventwithpython.com

    Use the WASD keys to move the tiles
    back into their original order:
          1  2  3  4
          5  6  7  8
          9 10 11 12
         13 14 15   ''')
    input('Press Enter to begin...')

    gameBoard = getNewPuzzle() ----> n^3

    while True: ----> n
        displayBoard(gameBoard) ----> 1
        playerMove = askForPlayerMove(gameBoard) ----> n^2
        makeMove(gameBoard, playerMove) ----> n^2

        if gameBoard == getNewBoard(): ----> 1
            print('You won!')
            sys.exit()
```

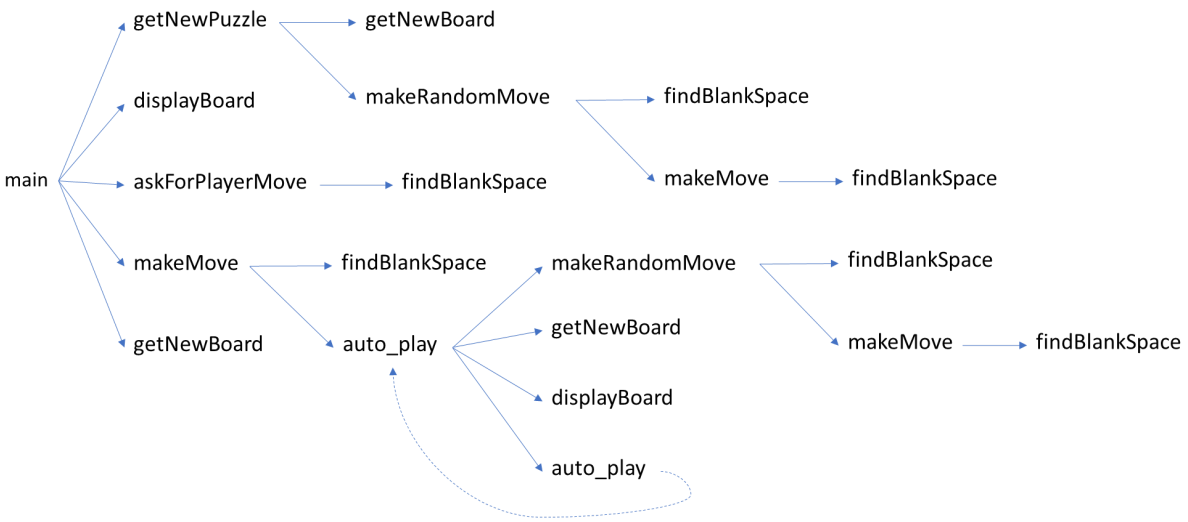
Finally, the time complexity of the *main* function's time complexity will depend on the time complexities of the function *getNewPuzzle* and the while loop, which has inside it the functions *displayBoard*, *askForPlayerMove*, *makeMove* and *getNewBoard*. Thus, summing n^3 with the multiplication of n (number of times the while is executed) by $1+n^2+n^2+1$, we will get a time complexity of $\Theta(n^3)$.

In conclusion, as the *main* function is the first to be called, we can confirm that the time complexity of this code will be $\Theta(n^3)$.

Part 2:

Starting from the code provided for the 4x4 Slide Tiles game, we changed some parameters so that we could transform it into a 5x5 game. Then we just added another function that would enable the user to, with the insertion of the value 'x', perform sets of 40 random moves in order for the program to try and complete the puzzle by itself.

To calculate the time complexity of this exercise, we will proceed with the same method of the previous exercise. Therefore here is a scheme of the order the functions are called.




```
def findBlankSpace(board):
    """Return an (x, y) tuple of the blank space's location."""
    for x in range(5): ⟶ n
        for y in range(5): ⟶ n
            if board[x][y] == '  ':
                return (x, y)
```

As there is a for loop inside another for loop (which only has lines of code with time complexity of 1 inside it), the time complexity of the *findBlankSpace* function will be $\Theta(n^2)$.

```
def makeMove(board, move):
    """Carry out the given move on the given board."""
    # Note: This function assumes that the move is valid.
    bx, by = findBlankSpace(board) ⟶ n^2

    if move == 'W':
        board[bx][by], board[bx][by+1] = board[bx][by+1], board[bx][by]
    elif move == 'A':
        board[bx][by], board[bx+1][by] = board[bx+1][by], board[bx][by]
    elif move == 'S':
        board[bx][by], board[bx][by-1] = board[bx][by-1], board[bx][by]
    elif move == 'D':
        board[bx][by], board[bx-1][by] = board[bx-1][by], board[bx][by]
    elif move == 'X':
        auto_play(board) ⟶ n^3
```

The time complexity of the *makeMove* function will depend on the way it is called. If it is called by the function *makeRandomMove* its time complexity will be of $\Theta(n^2)$. This is because the *makeRandomMove* restricts the possible valid moves that the program can run,

where the possibility ‘X’ will never be included. This means that the *auto_play* function would not be called and, therefore, not considered for the time complexity. The time complexity would then only depend on the *findBlankSpace* function.

However, this function may be called without going first through the *makeRandomMove* function first. This way the moves it can do will not be restricted. If the move chosen is ‘X’ and the last elif is executed, then, the function *auto_play* will be called and we will have to consider its time complexity for the calculations (we will afterwards come to the conclusion that it is $\Theta(n^3)$). The time complexity would then have the value $\Theta(n^3)$, which comes from the sum of time complexities of the functions *auto_play* and *findBlankSapce*.

```
def makeRandomMove(board):
    """Perform a slide in a random direction."""
    blankx, blanky = findBlankSpace(board) ⟶ n²
    validMoves = []
    if blanky != 4:
        validMoves.append('W')
    if blankx != 4:
        validMoves.append('A')
    if blanky != 0:
        validMoves.append('S')
    if blankx != 0:
        validMoves.append('D')

    makeMove(board, random.choice(validMoves)) ⟶ n²
```

The time complexity of the function *makeRandomMove* will be the sum of the time complexities of the functions *findBlankSpace* and *makeMove*. In this case, as the *makeMove* function will already have its possible moves limited, its time complexity will be $\Theta(n^2)$. Therefore the *makeRandomMove* function time complexity is $\Theta(n^2)$.

```
def getNewBoard():
    """Return a list of lists that represents a new tile puzzle."""
    return [['1 ', '6 ', '11', '16', '21'], ['2 ', '7 ', '12', '17', '22'],
            ['3 ', '8 ', '13', '18', '23'], ['4 ', '9 ', '14', '19', '24'], ['5 ', '10', '15', '20', BLANK]]
```

As the *getNewBoard* function only consists of the return of a matrix, its time complexity will be $\Theta(1)$.

```python
def getNewPuzzle(moves=16):
    """Get a new puzzle by making random slides from a solved state."""
    board = getNewBoard() ⟶ 1

    for i in range(moves): ⟶ n
        makeRandomMove(board) ⟶ n²
    return board
```

The *getNewPuzzle* function's time complexity will depend on the time complexity of the functions *getNewBoard* and *makeRandomMove* (multiplied by n, because of the for loop).

This way, the time complexity of *getNewPuzzle* is $\Theta(n^3)$. Here we also changed the parameter previously inside the function from moves=200 to moves=16 in order for the program to be able to solve the puzzle without exceeding the maximum recursion limit of the program.

```python
def displayBoard(board):
    """Display the given board on the screen."""
    labels = [board[0][0], board[1][0], board[2][0], board[3][0], board[4][0],
              board[0][1], board[1][1], board[2][1], board[3][1], board[4][1],
              board[0][2], board[1][2], board[2][2], board[3][2], board[4][2],
              board[0][3], board[1][3], board[2][3], board[3][3], board[4][3],
              board[0][4], board[1][4], board[2][4], board[3][4], board[4][4]]
    boardToDraw = """
+------+------+------+------+------+
|      |      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |      |
+------+------+------+------+------+
|      |      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |      |
+------+------+------+------+------+
|      |      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |      |
+------+------+------+------+------+
|      |      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |      |
+------+------+------+------+------+
|      |      |      |      |      |
|  {}  |  {}  |  {}  |  {}  |  {}  |
|      |      |      |      |      |
+------+------+------+------+------+

""".format(*labels)
    print(boardToDraw)
```

The display board function, as it only consists of basic operations, will have a time complexity of $\Theta(1)$.

```
def askForPlayerMove(board):
    """Let the player select a tile to slide."""
    blankx, blanky = findBlankSpace(board) ⟶ n²

    w = 'W' if blanky != 4 else ' '
    a = 'A' if blankx != 4 else ' '
    s = 'S' if blanky != 0 else ' '
    d = 'D' if blankx != 0 else ' '
    x = 'X'

    while True: ⟶ n
        print('                          ({})'.format(w))
        print('Enter WASD (or QUIT): ({}) ({}) ({})         ({})'.format(a, s, d, x))

        response = input('> ').upper()
        if response == 'QUIT':
            sys.exit()
        if response in (w + a + s + d + x).replace(' ', ''):
            return response
```

The *askForPlayerMove* time complexity will depend on the time complexities of the function *findBlankSpace* and the while loop. However, as the first one is $\Theta(n^2)$ and the second only $\Theta(n)$, the time complexity of *askForPlayerMove* will be $\Theta(n^2)$.

```
def auto_play(curr_board):
    curr_state = [row[:] for row in curr_board] ⟶ n
    for i in range(40): ⟶ n
        makeRandomMove(curr_state) ⟶ n²

        if curr_state == getNewBoard(): ⟶ 1
            print('The algorithm won!')
            sys.exit()

    print("Final Unsuccessful State")
    displayBoard(curr_state) ⟶ 1

    # Recursion in case of failed attempt
    auto_play(curr_board) ⟶ m
```

The *auto_play* function' time complexity will depend on the number of times the for loops are executed, the time complexities of the functions *makeRandomMove*, *getNewBoard*, and *displayBoard*, and the recursive call. Therefore, the time complexity of this function will be $(n + n \times (n^2 + 1) + 1) \times m$. As m (number of times the recursive call is executed) will be a finite number, the time complexity of *auto_play* will be $\Theta(n^3)$.

```
def main():
    print('''Sliding Tile Puzzle, by Al Sweigart al@inventwithpython.com

    Use the WASD keys to move the tiles
    back into their original order:
           1  2  3  4  5
           6  7  8  9  10
           11 12 13 14 15
           16 17 18 19 20
           21 22 23 24   ''')
    input('Press Enter to begin...')

    gameBoard = getNewPuzzle()  ⟶ n^3

    while True:  ⟶ n
        displayBoard(gameBoard)  ⟶ 1
        playerMove = askForPlayerMove(gameBoard)  ⟶ n^2
        makeMove(gameBoard, playerMove)  ⟶ n^3

        if gameBoard == getNewBoard():  ⟶ 1
            print('You won!')
            sys.exit()
```

Finally, the time complexity of the *main* function's time complexity will depend on the time complexity of the function *getNewPuzzle* and the while loop, which has inside it the functions *displayBoard*, *askForPlayerMove*, *makeMove* and *getNewBoard*. Thus, summing n^3 with the multiplication of n (number of times the while is executed) by $1 + n^2 + n^3 + 1$, we will get a time complexity of $\Theta(n^4)$.

In conclusion, as the *main* function is the first to be called, we can confirm that the time complexity of this code will be $\Theta(n^4)$.

Conclusion:

In the end, by using the knowledge acquired from the Theoretical and Practical classes, we found it possible to devise a plan for each of the exercises that were presented to us, having successfully worked as a group to implement those solutions in a, from an asymptotic perspective, viable way, with none of the algorithms having reached a complexity above n^4.

With that said, we conclude our report, happy with the work we have done and the results we have achieved, having followed all the directives accordingly.

Contributions of each member to the project:

Exercise 1:

- joining all the sorting algorithms - all
- creating a code without classes - Rita & Afonso
- idea and creation of a code with classes - Marcelo
- searching for and adapting new sorting algorithms - Afonso & Marcelo
- building the table - Marcelo
- solving problems - Rita & Afonso
- testing the code - all
- time complexity - Rita
- flowchart of the additional sorting algorithms - Afonso

Exercise 2:

- base idea - Afonso
- implementation of the idea - Rita
- solving problems - all
- testing code - all
- time complexity - Rita

Exercise 3:

<u>Part 1:</u>

- time complexity - Rita

Part 2:

- changing the game to 5x5 - Rita
- building the auto_play function - Marcelo
- solving problems - all
- testing the code - all
- time complexity - all

References:

Leonardo Vanneschi (2022, June 5). *Sorting Algorithms*. NOVA Information Management School. <https://elearning.novaims.unl.pt/pluginfile.php/110525/mod_resource/content/1/slides07-sorting.pdf>

Leonardo Vanneschi (2022, June 5). *Object-Oriented Programming in Python in a Nutshell*. NOVA Information Management School. <https://elearning.novaims.unl.pt/pluginfile.php/110526/mod_resource/content/1/slides08-obj-oriented-python-nutshell.pdf>

GeeksforGeeks 2022, accessed in 5 June 2022, <https://www.geeksforgeeks.org/radix-sort/>

GeeksforGeeks 2022, accessed in 5 June 2022, <https://www.geeksforgeeks.org/shellsort/>

GeeksforGeeks 2022, accessed in 5 June 2022, <https://www.geeksforgeeks.org/heap-sort/>

Sweigart, Al. *The big book of small Python projects*. no starch press. <https://elearning.novaims.unl.pt/pluginfile.php/115469/mod_resource/content/0/Al%20Sweigart%20-%20The%20Big%20Book%20of%20Small%20Python%20Projects_%2081%20Easy%20Practice%20Programs-No%20Starch%20Press%20%282021%29.pdf>