NOVA Information Management School

Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa

SEMESTER PROJECT

by

Catarina Oliveira | 20211616

Inês Vieira | 20211589

Martim Serra | 20211543

Luís Soeiro | 20211536

Course Unit: Algorithms and Data Structures

Lectured by Professors Leonardo Vanneshi & Mijail Zolotov

INDEX

INDEX OF FIGURES

INTRODUCTION

During this semester we acquired more knowledge in the programing language *python*, and this project aims to test our ability to put into practice the given knowledge. The project is divided into three parts, each one evaluating a specific area of the lectured subject.

The **first** exercise is to put to test the theoretical notions of the time that it takes for different sorting algorithms, studied throughout the semester, to sort a given random list.

The **second** exercise tests our knowledge of python in a more general way, focusing particularly on the application of a sorting algorithm in a specific way and with certain rules.

The **third** and final exercise puts into practice our knowledge of understanding a premade code well enough to make the requested changes.

Adding to this, it was also requested of us to do the time complexity of the two first exercises and of the premade code of the third, once again to test our abilities in the subject that was learned during the semester.

TIME COMPLEXITY

1st Exercise

In this first exercise, the sorting algorithms used were bubble sort, heap sort, selection sort, merge sort, insertion sort, quick sort, and shell sort. For all these algorithms the **average and worst cases** were considered.

Studying it more deeply we have the following loops and functions with greater relevance to time complexity (t.c.):

```
# LIST GENERATOR
all_iterations = []  # List that will contain all the random lists to order
for i in range(iterations):
    iteration = [random.randint(0, range_size) for j in range(initial_size + i * incr_size)]
    all_iterations.insert(i, iteration)
print(all_iterations)
```

Figure 1: Iterations for loop

In figure 1, the time complexity is n^2 since there are two for loops both with n iterations (n x n).

```
# List with the length of each iteration
list_n = ["Length of list:"]  # length of each random list
list_indexes = ["Iteration:"]  # Number of iterations
for i in range(iterations):
    result = initial_size + i * incr_size
    list_n.insert(i + 1, result)
    list_indexes.append(str(i + 1))
```

Figure 2: List with the length of each iteration

In figure 2, the size of the problem is n (nº of elements stored in 'iterations'), in the for loop. Assuming that the basic operations within the for loop have constant time, this part of the program has a $\Theta(n) = n$.

```
# bubble_sort
def bubble_sort(list_to_sort):

    n = len(list_to_sort)
    for k in range(n - 1):
        for j in range(0, n - k - 1):
            if list_to_sort[j] > list_to_sort[j + 1]:
                list_to_sort[j], list_to_sort[j + 1] = list_to_sort[j + 1], list_to_sort[j]


    return list_to_sort
```

Figure 3: Bubble Sort

The bubble sort Algorithm has a time complexity of $\Theta(n) = n^2$ for both the average and worst cases.

```
def heap_sort(list_to_sort):

    n = len(list_to_sort)
    for i in range(n // 2 - 1, -1, -1):
        heapify(list_to_sort, n, i)

    # One by one extract elements
    for i in range(n - 1, 0, -1):
        swapping(list_to_sort, i, 0) # swap
        heapify(list_to_sort, i, 0)
```

Figure 4: Heap Sort

Heap Sort is composed of two functions. Since the first one is only composed of comparisons, with constant time, it is not mentioned here. That said, figure 4 represents part of the heap sort algorithm and has a time $\Theta(n) = n \log(n)$ for both average and worst cases.

```
# Selection sort
def selection_sort(list_to_sort):

    for k in range(len(list_to_sort)):
        min_idx = k
        for j in range(k + 1, len(list_to_sort)):
            if list_to_sort[min_idx] > list_to_sort[j]:
                min_idx = j
        list_to_sort[k], list_to_sort[min_idx] = list_to_sort[min_idx], list_to_sort[k]

    return list_to_sort
```

Figure 5: Selection Sort

Selection Sort has a time complexity of $\Theta(n) = n^2$ for both the average and worst cases.

```
def merge_sort(list_to_sort):

    if len(list_to_sort) > 1:
        mid = len(list_to_sort) // 2
        left = list_to_sort[:mid]
        right = list_to_sort[mid:]
        merge_sort(left)
        merge_sort(right)
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                list_to_sort[k] = left[i]
                i += 1
            else:
                list_to_sort[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            list_to_sort[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            list_to_sort[k] = right[j]
            j += 1
            k += 1
```

Figure 6: Merge Sort

Merge Sort has a time complexity of $\Theta(n) = n \log(n)$ in both average and worst cases.

```
# Insertion sort
def insertion_sort(list_to_sort):
    for i in range(1, len(list_to_sort)):
        sort_value = list_to_sort[i]
        while list_to_sort[i - 1] > sort_value and i > 0:
            swapping(list_to_sort, i, i - 1)
            i = i - 1

    return list_to_sort
```

Figure 7: Insertion Sort

Insertion Sort has a time complexity of $\Theta(n) = n^2$ since we're considering the average and worst cases. In the very rare occasion where the best case happened time complexity would be $\Theta(n) = n$.

```python
# Quick sort
def partition(list_to_sort, low, high):
    i = (low - 1)
    pivot = list_to_sort[high]
    for j in range(low, high):
        if list_to_sort[j] <= pivot:
            i = i + 1
            swapping(list_to_sort, i, j)
    swapping(list_to_sort, i + 1, high)
    return (i + 1)

def quick_sort(list_to_sort, low, high):

    if len(list_to_sort) == 1:
        return list_to_sort
    if low < high:
        pi = partition(list_to_sort, low, high)
        quick_sort(list_to_sort, low, pi - 1)
        quick_sort(list_to_sort, pi + 1, high)
    return list_to_sort
```

Figure 8: Quick Sort

Quick Sort is also composed of two different functions. Overall, in the average case, this algorithm has a time complexity of $\Theta(n) = n \log(n)$. In the worst case, it has a time complexity of n^2.

```python
# Shell Sort
def shell_sort(list_to_sort, n):
    interval = n // 2
    while interval > 0:
        for i in range(interval, n):
            temp = list_to_sort[i]
            j = i
            while j >= interval and list_to_sort[j - interval] > temp:
                list_to_sort[j] = list_to_sort[j - interval]
                j -= interval
            list_to_sort[j] = temp
        interval //= 2
```

Figure 9: Shell Sort

Shell Sort has a time complexity of $\Theta(n) = n \log (n)$ both for the best and average cases. Since the length of the list depends on the input given by the user which is used by the function ‘shell_sort’, the time complexity of the function not only depends on how unsorted the list is but also on the result of this input. The behavior of the function turns out to be more unpredictable than other sorting algorithms which may lead to a challenging computation of the worst-case time complexity. However, it’s secure to say that the time complexity of the worst case is always less or equal than $\Theta(n) = n^2$.

Finally, the time complexity of the for loops regarding the results according to the function call is exemplified in the next figures:

```
# Bubble sort Results
bubble_results = ["bubble sort: "]
for j in range(len(list3)):
    start3 = time.time()
    bubble_sort(list3[j])
    end3 = time.time()
    results3 = "{0:.5f}".format(end3 - start3)
    bubble_results.append(results3 + "s")
```

Figure 10: Bubble Sort Results

The time complexity of figure 10 is composed of n (in the for loop) times the time complexity of bubble sort (n^2) meaning the overall time complexity of these lines of code is $n * n^2 = n^3$, $\Theta(n) = n^3$.

```
# Heap sort results
heap_results = ["heap sort:"]
for j in range(len(list4)):
    start4 = time.time()
    heap_sort(list4[j])
    end4 = time.time()
    results4 = "{0:.5f}".format(end4 - start4)
    heap_results.append(results4 + "s")
```

Figure 11: Heap Sort Results

The time complexity of the figure 11 is $n * n \log(n) = n^2 \log(n)$ following the same logic as in figure 10; $\Theta(n) = n^2 \log(n)$.

```
# Selection Sort results
selection_results = ["selection sort:"]
for j in range(len(list5)):
    start5 = time.time()
    selection_sort(list5[j])
    end5 = time.time()
    results5 = "{0:.5f}".format(end5 - start5)
    selection_results.append(results5 + "s")
```

Figure 12: Selection Sort Results

The time complexity of the figure 12 is $n * n^2 = \Theta(n) = n^3$.

```
# Merge sort results
merge_results = ["merge sort:"]
for j in range(len(list6)):
    start6 = time.time()
    merge_sort(list6[j])
    end6 = time.time()
    results6 = "{0:.5f}".format(end6 - start6)
    merge_results.append(results6 + "s")
```

Figure 13: Merge Sort Results

The time complexity of Merge Sort Results is n * n log(n) = n^2 log(n) = Θ(n).

```
# Insertion sort results
insertion_results = ["insertion sort:"]
for j in range(len(list7)):
    start7 = time.time()
    insertion_sort(list7[j])
    end7 = time.time()
    results7 = "{0:.5f}".format(end7 - start7)
    insertion_results.append(results7 + "s")
```

Figure 14: Insertion Sort Results

The time complexity of figure 14 is n * n^2 = n^3 = Θ(n) in both worst and average cases.

The time complexity of figure 15 is n * nlog (n) which results in Θ(n) = n^2 log (n) in both worst and average cases.

```
# Quick sort results
quick_results = ["quick sort:"]
for j in range(len(list1)):
    start1 = time.time()
    quick_sort(list1[j], 0, len(list1[j]) - 1)
    end1 = time.time()
    results1 = "{0:.5f}".format(end1 - start1)
    quick_results.append(results1 + "s")
```

Figure 15: Quick Sort Results

```
# Shell sort results
shell_results = ["shell sort:"]
for j in range(len(list2)):
    len_arg = len(list2[j])
    start2 = time.time()
    shell_sort(list2[j], len_arg)
    end2 = time.time()
    results2 = "{0:.5f}".format(end2 - start2)
    shell_results.append(results2 + "s")
```

Figure 16: Shell Sort Results

As mentioned in figure 9, shell sort is not as predictable as other sorting algorithms, and so, the code in figure 16 depends on the result of the shell sort. If the average case happens then figure 16 would have a time complexity of $\Theta(n) = n^2 \log(n)$. However, when the worst-case happens the maximum time complexity might be n^3

In conclusion, in the study of the time complexity of this first exercise, as learned in theoretical classes, terms of inferior order are excluded as well as the eventual multiplicative coefficient of higher terms, and so, the time complexity of this exercise is $\Theta(n) = n^3$.

2nd Exercise

In the second exercise, the same logic will be used, only the loops and functions relevant to time complexity computation will be presented in the following figures.

```
# shuffling function
def shuffling(list):
    n = len(list)
    for i in range(n):
        j = random.randint(0, n - 1)
        element = list.pop(j)
        list.append(element)
    return list
```

Figure 17: Shuffling Function

The time complexity of the 'shuffling' function is $\Theta(n) = n$.

```
def max(lst):
    max = lst[0]
    for i in lst:
        if i > max:
            max = i
    return max
```

Figure 18: Max Function

To calculate the maximum and the minimum values in a list, max and min functions were used. As both have a similar time complexity only one is presented here and its time complexity is $\Theta(n) = n$.

```
def  counter(list1, x):
    count = 0
    for item in list1:
        if (item == x):
            count = count + 1
    return count
```

Figure 19: Counter Function

Function counter has a problem size of n, with comparisons and assignments to variables within and so time complexity is Θ(n) = n.

```
def overlap(list):

    unwanted = []

    for i in list:
        if counter(list, i) > 1:
            unwanted.append(i)

    for i in unwanted:
        if i in list and counter(unwanted, i) % 2 == 0:
            list.remove(i)

    return(list)
```

Figure 20: Overlap Function

The overlap function is composed of two for loops with a ‘counter’ function and an ‘in’ operator within. The counter has a t.c. of n as the ‘in’ operator which also has a t.c. of n even though the length of both lists is not the same (for simplicity purposes both inputs were considered n). Meaning that the time complexity of both parts is: n * n + n * n * n = $n^2 + n^3$. Therefore Θ(n) = n^3.

```
def final(matrix):
    global before_overlap
    overlap_list = []
    before_overlap = [i for i in matrix if matrix.count(i) == 1]


    # before_overlap --> matrix with all the inputs without overlapping
    # overlap_list --> inputs list to be overlapped


    for j in before_overlap:
            overlap_list.extend(j)


    overlap(overlap_list)
    # if an element was not removed on the overlap_list by overlap, it should appear only once on new_list
    new_list = []
    for i in overlap_list:
        if i not in overlap_list:
            new_list.append(i)

    overlap_list = new_list

    # overlap list -> sorted list without repeated integers
```

Figure 21: 1st Part of Function 'final'

'Final' function is a long function and because of that, it is divided between figures 21 and 22. In this figure, the fourth line, on which appears a first for loop and the function count of python, has a time complexity of n^2. The second for loop also has a time complexity of n. 'Overlap' function has a time complexity of n^3 as seen in figure 20. Finally, the last for loop has a time complexity of n^2. Overall, this first part of the function has a time complexity of $n^2 + n + n^3 + n^2 = 2n^2 + n + n^3$; $\Theta(n) = n^3$.

```
    list_of_lists = []
    one_of_the_lists = [overlap_list[0]]
    for i in range(len(overlap_list) - 1):
        if overlap_list[i + 1] == overlap_list[i] + 1:
            one_of_the_lists.append(overlap_list[i + 1])
        else:
            list_of_lists.append(one_of_the_lists)
            one_of_the_lists = [overlap_list[i + 1]]
    list_of_lists.append(one_of_the_lists)
    # list_of_lists -> matrix of the sorted elements, without shuffling

    for i in range(len(list_of_lists)):
        if i % 2 != 0:
            list_of_lists[i] = shuffling(list_of_lists[i])
    # list_of_lists shuffled

    final_final_list = []
    for i in list_of_lists:
        final_final_list.extend(i)
    # final_final_list is the only printed list


    return final_final_list
```

Figure 22: 2nd part of 'final' function

On this second part of the function ‘final’ there are three for loops in which the time complexity is $n + n * n + n = 2n + n^2$. So, overall, joining both parts of the function it has a time complexity of $\Theta(n) = n^3$.

```
# Forming new lists
def make_lists(list0):
    list0 = []

    min = int(input("Choose an integer to be the lowest number of your interval (can't be lower than -999): "))
    max = int(input("Choose an integer to be the highest value of your interval(can't be higher than 999): "))

    while min < -999:
        print('Minimum too low, choose some other values:')
        min = int(input("Choose an integer to be the lowest number of your interval (can't be lower than -999): "))

    while max > 999:
        print('Maximum too high, choose some other values:')
        max = int(input("Choose an integer to be the highest value of your interval(can't be higher than 999): "))

    for i in range(min, max + 1):
        list0.append(i)

    matriz(list0)

    return list0
```

Figure 23: 'Make Lists' Function

The function ‘make_lists’ has several while loops with a finite size problem depending on the input given by the user. The for time complexity is Θ(n) = n.

```
def merge_lists(list0, matrix, new_list):
    print(final(matrix))

    question = ''
    while not ('no' or 'yes') in question.lower():
        question = input('Do you want to register a new interval? Answer yes or no: ')
        if 'no' in question.lower():

            print(f'END OF PROGRAM, your final list is: {(final(matrix))}')
            exit()

        elif 'yes' in question.lower():

            make_lists(new_list)
```

Figure 24: 1st part of 'Merge Lists' function

```
        # sorts the new list
        for i in matrix:
            for j in matrix:
                if max(i) + 1 == min(matrix[-1]) and i != j and max(matrix[-1]) == min(j) - 1:
                    i.extend(matrix[-1][:])
                    i.extend(j)
                elif max(i) + 1 == min(matrix[-1]):
                    i.extend(matrix[-1][:])
                    matrix.pop()
                elif min(i) - 1 == max(matrix[-1]):
                    matrix[-1].extend(i)
                    matrix.remove(i)

            for i in range(len(matrix)):
                while i < len(matrix):
                    if min(matrix[-1]) <= min(matrix[i]):
                        swapping(matrix, -1, i)
                    break

        print(final(matrix))
```

Figure 25: 2nd part of 'Merge Lists' function

The function ‘merge lists’ is also long, so it is divided between figures 24 and 25. In this first part, it presents a while loop with time complexity **b** which can be defined as a finite number of iterations depending on the input chosen by the user. It also calls the function final, which has a time complexity of n^3. The last function called in this first part is ‘make lists’ which has a time complexity of n. To summarize, this part of the function has a time complexity of $b * n^3 + b * n = Θ(n) = n^3$.

On this second part of the function, there is a first while loop with time complexity **b** plus a nested for with time complexity n^2, then there is another nested for with a while loop inside, which combines in a t.c. of n^3. Finally, this function calls the 'final' function that has a time complexity of n^3. This second part of the function ends up by having a time complexity of b * n^3. Overall, the function 'merge lists' has a time complexity of $\Theta(n) = n^3$.

To conclude, by excluding all the minor terms, and the multiplicative coefficient of the higher term, the time complexity of the program presented is $\Theta(n) = n^3$.

3rd Exercise

The 3rd exercise is divided into 2 parts. In the first part, the time complexity of the original Sliding Tile Puzzle (retrieved from the "The Big Book of Small Python Projects", by AI Sweigart) was analyzed. As the variation of this puzzle is quite similar, time complexity wasn't computed again.

1. Original Sliding Tile Puzzle

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

Figure 26: 'findBlankSpace' function

```
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)

    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]
```

Figure 27: 'makeMove' function

```
def makeRandomMove(board):
    """Perform a slide in a random direction."""
    blankx, blanky = findBlankSpace(board)
    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))
```

Figure 28: 'makeRandomMove' function

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

    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:
        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
```

Figure 29: 'askForPlayerMove' function

The time complexity of the figures 26-29 is interconnected. Starting with figure 26, 'findBlankSpace' as time complexity of n^2 due to the two for loop. 'MakeMove' (figure 27) has a time complexity of n^2 since it calls the function 'findBlankSpace' (considering that the other lines of code have a constant time complexity). The next function is the 'makeRandomMove' which calls both the function 'findBlankSpace' and 'MakeMove'. As its other lines of code have constant time complexity, its t.c. is $n^2 + n^2 = 2n^2$, which is a $\Theta(n) = n^2$.

Finally, figure 29 calls the function 'findBlankSpace' and a while loop with time complexity n and so its time complexity is $n^2 + n$; $\Theta(n) = n^2$.

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

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

Figure 30: ‘getNewPuzzle’ function

Function ‘getNewPuzzle’ calls the function ‘getNewBoard’ which has a constant t.c.. Besides that, it also contains a for loop with t.c. equal to n and calls the function ‘makeRandomMove’ which, as seen in the last paragraphs, has a t.c of n^2. So, the overall t.c. is $n * n^2 = n^3$.

```
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   ''')
    input('Press Enter to begin...')

    gameBoard = getNewPuzzle()

    while True:
        displayBoard(gameBoard)
        playerMove = askForPlayerMove(gameBoard)
        makeMove(gameBoard, playerMove)

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

Figure 31: 'main' function

‘Main’ Function is the first function that appears on the code of this exercise. Besides the lines with constant t.c., it has a while loop with t.c. equal to n, where it calls the functions ‘getNewPuzzle’, ‘displayBoard’ ($\Theta(n) = 1$), ‘askForPlayerMove’, ‘makeMove’ and ‘getNewBoard’. And so, its time complexity is $n^3 + n * 1 + n * n^2 + n * n^2 + n = n^3 + 2n + n * 2n^2$; $\Theta(n) = n^3$.

Excluding all the minor terms and multiplicative coefficients of the higher terms, the time complexity of the mentioned program was $\Theta(n) = n^3$.

´

CONCLUSION AND AUTHOR'S CONTRIBUTIONS

Summarizing what was concluded about the average time complexity of each program, the first one had a time complexity of $\Theta(n) = n^2$, the second one was also $\Theta(n) = n^2$ and finally, on the third exercise, both on the original and the variant have $\Theta(n) = n$.

Some clarification on the code used in the 5 x 5 variant of the Sliding Tile Puzzle is that in the function getNewPuzzles the moves were changed from 200 to 3 for solving purposes, in other words, a higher probability of the code solving itself was desired to check if the changes were correct and the exercise complete.

About the author's contribution, Luís Soeiro wrote the basis of the code of the first exercise, and Inês Vieira reviewed it and improved it, in order for it to work completely without any mistakes.

Catarina Oliveira and Martim Serra wrote the code of the second exercise together.

The second part of the third exercise was accomplished by Inês Vieira.

The time complexity of all of the exercises was computed by Catarina Oliveira, Inês Vieira, and Martim Serra. Being them who wrote the report.

We would also like to thank both of the professors for their contributions and availability for all of the questions related to this work, without them it wouldn't have been possible to accomplish all the objectives of this project.

REFERENCES

- Vanneschi, L. (2022). *Computational Complexity and Analysis of Algorithms.* [Supporting Slides of the Unit Course Algorithms and Data Structures, lectured at NOVA IMS]. https://elearning.novaims.unl.pt/pluginfile.php/110523/mod_resource/content/1/slides05-computational-complexity-analysis-of-algorithms.pdf
- Vanneschi, L. (2022). *Sorting Algorithms.* [Supporting Slides of the Unit Course Algorithms and Data Structures, lectured at NOVA IMS]. https://elearning.novaims.unl.pt/pluginfile.php/110525/mod_resource/content/1/slides07-sorting.pdf
- Geeks for geeks. (2022). *Python Program for Heap Sort.* https://www.geeksforgeeks.org/python-program-for-heap-sort/
- Programiz. *Shell Sort Algorithm.* https://www.programiz.com/dsa/shell-sort
- Sweigart, A.I. (2021). *The Big Book of Small Python Projects.* No Starch Press.