International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 09 | Sept - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Implimentation Of Table_Track_Sort() Algorithm Which Classifies the Elements of an Array in Known Order. Further Examining the Time Complexity and Space Complexity of an Algorithm and Analysing the Efficacy of the Algorithm - A Case Study.

6674 – Cadet P AKILESHWER[1], 6677 – M R SANJAY[2]
Class- XII 2025-26, Sainik School Amaravathinagar
Post: Amaravathinagar,Udumalpet Taluka,Tirupur Dt,Tamilnadu State

ABSTRACT

In the field of computer science, the efficiency of a program solely depends on the time factor of processing a statement or block of statements. Further, the amount of memory it uses for processing also matters in calculating the space complexity of the program.

There are numerous sorting techniques that exist; each one has pros and cons, and these sorting techniques are used in the development of software. The table_track sorting algorithm is a unique method to sort the elements of a given list of numbers.

This manuscript implements the function table_track_sort()*, which *uses a matrix table to sort a list of numbers further. It examines the execution of the methodology with respect to time complexity and space complexity of the algorithm.

Keywords: Table Track Sort (TTS), Matrix Table(MT), Number (N), Compared Number Position(CNP), Original Number Position (ONP), Original And Compared Number Result (OCR).

1. INTRODUCTION

The Sorting is the process of arranging items into a sequence according to a specific order, such as text from A to Z, numbers from smallest to largest or from largest to smallest, or dates from oldest to newest or newest to oldest. It helps in organizing and understanding data, making it easier to visualize, find, and make decisions. While sorting can refer to simple categorization or organizing physical objects, in the context of computing, it specifically involves using sorting algorithms to reorder elements in an array or list

Some of the most popular sorting techniques are Bubble Sort, Merge Sort, Quick Sort, Counting Sort, Heap Sort, Insertion Sort, etc. Here we will see the comparison between the most popular bubble_sort() and table_track_sort(). The Bubble_sort() is a comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Here, table_track_sort() is an algorithm that sorts series in a specific order using table formation and iteration in a table with the given parameters.

2. RELATED WORK:

Related work on sorting techniques by researchers focuses on categorizing algorithms by their method (e.g., comparison-based, non-comparison-based), evaluating their performance in terms of time and space complexity (e.g., $O(n^2)$, $O(n \log n)$), and exploring their efficiency for different data sizes and types. Key areas of research include understanding fundamental algorithms like Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, and Radix Sort, as well as developing novel, more efficient sorting algorithms to handle increasingly large datasets.

3. METHODOLOGY

The ***Table_Track_Sort (TTS)*** is a comparison-driven sorting algorithm that integrates relational tracking with the help of tabular computation. Unlike traditional algorithms such as Merge_Sort and Quick_sort, which rely on the dividing and conquering archetype, ***Table_Track_Sort(TTS)*** constructs an auxiliary comparison table, which is called the ***MATRIX TABLE (MT),*** that precisely determines pairwise ordering relationships. This Matrix table forms the foundation for deriving the final sorted sequence. The methodology emphasizes transparency of intermediate decisions by explicitly recording and ordering comparisons in a structured tabular form.

The execution process of the algorithm is, once the user provides the required inputs, the ***Table_Track_Sort(TTS)*** will generate the comparison table called the **MATRIX TABLE (MT)** as described above. This very table consists of the elements with their position, which have been processed by the comparison methodology. The Matrix Table will become the blueprint to generate the list of numbers as per the order (ascending or descending) provided by the user.

n	onp	cnp	ocr
77	0	1	1
77	0	2	1
77	0	3	1
62	1	0	-1
62	1	2	-1
62	1	3	1
74	2	0	-1
74	2	1	1
74	2	3	1
24	3	0	-1
24	3	1	-1
24	3	2	-1

Further, the **Table_Track_Sort()** algorithm is implemented using one of the existing languages to check the efficacy of the algorithm.

4. STRUCTURE OF THE MATRIX TABLE:

Enter elements separated by spaces: 77 62 74 24
Enter order (asc/desc): desc

5. FUNCTIONING OF MATRIX TABLE

The system accepts four elements and the order of arranging the elements (ascending/descending). The algorithm generates a total of 12 rows in the matrix form, which contains the element, Original Number Position (ONP), Compared Number Position (CNP), and the last column, original and compared number result (OCR).

The first element will be picked by the Table_Track_Sort (TTS) algorithm, and it records

the Original Number Position (ONP). It compares the successive elements present in the given list, and the final result of the comparison, the index is recorded as shown in the Matrix Table (MT). once the Matrix table is generated, the algorithm identifies the exact position of the element and generates the sorted list in some known order.

6. ALGORITHM: Table_Track_Sort()

STEP 01: START
STEP 02: INITIALIZE AN EMPTY COMPARISON TABLE.

STEP 03:LOOP OVER EACH ELEMENT N IN THE LIST
STEP 04: FOR EACH N, COMPARE IT WITH EVERY OTHER ELEMENT COMPARED_N

STEP 05: SKIP COMPARISON WHEN AN ELEMENT IS COMPARED WITH ITSELF.
STEP 06: DETERMINE THE COMPARISON RESULT (OCR):
STEP 07: RECORD A ROW IN THE TABLE
STEP 08: REPEAT STEPS 4–7 UNTIL ALL POSSIBLE PAIRS ARE COMPARED.
STEP 09: INITIALIZE A DICTIONARY COUNTS TO STORE WIN COUNTS FOR EACH ELEMENT.
STEP 10: TRAVERSE THE TABLE
STEP 11: INITIALIZE AN EMPTY RESULT LIST AND A USED SET.
STEP 12: WHILE THE RESULT LIST IS NOT COMPLETE, SELECT THE NEXT CANDIDATE ELEMENT:
STEP 13: APPEND THE CHOSEN ELEMENT TO THE RESULT LIST.
STEP 14: MARK THAT ELEMENT AS USED, SO IT IS NOT SELECTED AGAIN.
STEP 15: MARK THAT ELEMENT AS USED, SO IT IS NOT SELECTED AGAIN.
STEP 16: OUTPUT

7. PROGRAM: Table_Track_Sort (TTS):

 # Table_Track_Sort () function definition:

```
def Table_Track_Sort(arr, order='asc'):
    table = []


    for i in range(len(arr)):
        for j in range(len(arr)):
            if i == j:
                continue
            n = arr[i]
            compared_n = arr[j]
            ocr = 1 if n > compared_n else (-1 if n <
compared_n else 0)
            table.append({
                'n': n,
                'onp': i,
                'cnp': j,
                'ocr': ocr
            })


      counts = {}
    for row in table:
        n = row['n']
        if n not in counts:
            counts[n] = 0
        if row['ocr'] == 1:
            counts[n] += 1
    result = []
    used = set()
    while len(result) < len(arr):
        candidate = None
        for n in arr:
            if n in used:
```

```
            continue
        if candidate is None:
            candidate = n
        else:
            if order == 'asc':
                if counts[n] < counts[candidate]:
                    candidate = n
            else:
                if counts[n] > counts[candidate]:
                    candidate = n
        result.append(candidate)
        used.add(candidate)

    return result, table


def print_table(table):
    print("\nComparison Table:")
    print(f"{'n':<5}{'onp':<5}{'cnp':<5}{'ocr':<5}")
    print("-"*20)
    for row in table:

print(f"{row['n']:<5}{row['onp']:<5}{row['cnp']:<5}{ro
w['ocr']:<5}")

# main() function definition
def main():
    user_input = input("Enter elements separated by
spaces: ")
    arr = list(map(int, user_input.split()))

    order = input("Enter order (asc/desc):
").strip().lower()
    result, table = table_track_sort(arr, order=order)

    print("\nSorted Result:", result)
    print_table(table)

# Calling function
if __name__ == "__main__":
    main()
```

8. FUNCTIONING OF CODE

The code implements the Table_Track_Sort(TTS) algorithm, a comparison-based sorting technique. It takes a list of numbers and the desired order (ascending or descending) from the user. Inside the table_track_sort function, each element is compared with every other element, and the results of these comparisons are stored in a table with fields: the **Number (N)**, Its **ORIGINAL NUMBER POSITION (ONP)**, the **COMPARED NUMBER POSITION (CNP)**, and the **ORIGINAL AND COMPARED NUMBER (OCR)**.

Using this table, the algorithm counts how many times each element is greater than the others (wins) and arranges the elements accordingly. For ascending order, elements with fewer wins appear first; for descending order, elements with more wins appear first. The program then returns both the sorted result and the full comparison table. The print_table() function displays this table in a clear format, and the main function manages input, execution, and output.

9. COMPLEXITY OF ALGORITHM

In computer science, the analysis of algorithms is a crucial part. It is important to find the most efficient algorithm for solving a problem. It is possible to have many algorithms to solve a problem, but the challenge here is to choose the most efficient one.[1] There are multiple ways to design an algorithm, or considering which one to implement in an application. When thinking through this, it's crucial

to consider the algorithm's **time complexity** and **space complexity**.[2]

10. SPACE COMPLEXITY

The space complexity of an algorithm is the amount of space (or memory) taken by the algorithm to run as a function of its input length, n. Space complexity includes both auxiliary space and space used by the input.[3]

Auxiliary space is the temporary or extra space used by the algorithm while it is being executed. Space complexity of an algorithm is commonly expressed using **Big (O(n))** notation.[3]

The Space complexity is ignored in this research paper, since the space complexity of a particular problem is not considered so important.

Space complexity for the Table_Track_Sort() algorithm is : O(n)

11. TIME COMPLEXITY

The time complexity of an algorithm is the amount of time taken by an algorithm to complete its process as a function of its input length, n. The time complexity of an algorithm is commonly expressed using asymptotic notations: [2]

Big O - O(n)
Big Theta - Θ(n)
Big Omega - Ω(n)

It's valuable for a programmer to learn how to compare the performances of different algorithms and choose the best time-space complexity to solve a particular problem in the most efficient way possible.[2]

Big O notation is used in Computer Science to portray the performance or complexity of an algorithm.

Big O specifically defines the worst-case scenario of an algorithm, and can be used to describe the execution time required or the space used (e.g., in memory or on disk) by an algorithm. here, O stands for order of growth.

Big Theta(Θ) is used to represent the average case scenario of an algorithm and can be used to describe the execution time required or the space used (e.g., in memory or on disk) by an algorithm.

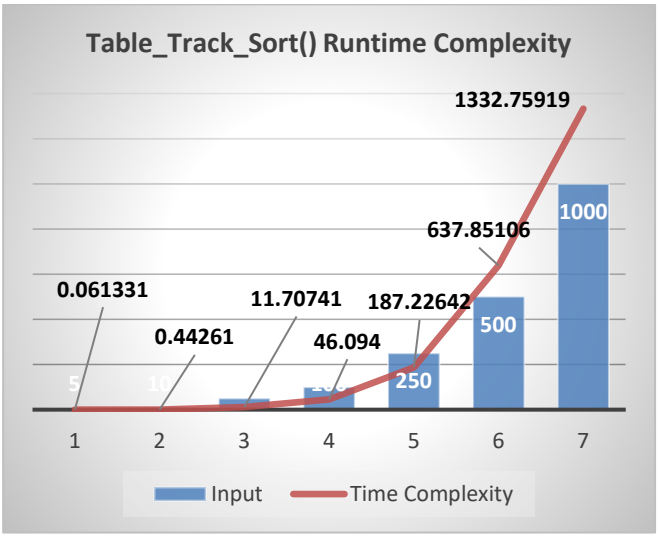
Big Omega (Ω)is used to represent the best-case scenario of an algorithm and can be used to describe the execution time required or the space used (e.g., in memory or on disk) by an algorithm.

These three methods are the most common and very popular methods of design and analysis of an algorithm and are used for finding the efficiency of the program.

12. RUNTIME COMPLEXITY OF Table_Track_Sort(TTS)

Input	Time Complexity
5	0.061331
10	0.44261
50	11.70741
100	46.094
250	187.22642
500	637.85106
1000	1332.75919

13. GRAPHICAL REPRESENTATION



14. THE RUNTIME COMPLEXITY OF TABLE_TRACK_SORT():

```
for i in range(len(arr)):
    for j in range(len(arr)):
        if i == j:
            continue
        n = arr[i]
        compared_n = arr[j]
        ocr = 1 if n > compared_n else (-1 if n <
compared_n else 0)
        table.append({
            'n': n,
            'onp': i,
            'cnp': j,
            'ocr': ocr
        })
```

For the above code the time complexity is O (n^2)

NEXT FOR LOOP

```
for row in table:
    n = row['n']
    if n not in counts:
        counts[n] = 0
    if row['ocr'] == 1:
        counts[n] += 1
result = []
used = set()
```

For the above code the time complexity is O (n)

WHILE LOOP

```
while len(result) < len(arr):
    candidate = None
    for n in arr:
        if n in used:
            continue
        if candidate is None:
            candidate = n
        else:
            if order == 'asc':
                if counts[n] < counts[candidate]:
                    candidate = n
            else:
                if counts[n] > counts[candidate]:
                    candidate = n
    result.append(candidate)
    used.add(candidate)
```

For the above code the time complexity is O (n^2)

TOTAL TIME COMPLEXITY:

Add the complexities of each part:

*** Part 1: O(n^2)**
*** Part 2: O(n)**
*** Part 3: Between O(n) and O(n^2)**

So the overall time complexity is:

Best-case: O(n^2)
Worst-case:O(n^2 + n + n^2) = O(2n^2 + n) = O(n^2)
(since highest-order term dominates)

Final Time Complexity: O(n^2)

15. CONCLUSION

There are numerous sorting techniques that exist; each one has pros and cons, and these sorting techniques are used in the development of software. The table_track_sort () sorting algorithm is a unique method to sort the elements of a given list of numbers. Unlike other sorting techniques it uses a table to sort list of numbers by recording comparisons. This is one of the distinct methodology of implementing the sorting method.

16. ACKNOWLEDGEMENT

Apart from our efforts, the success of any work or project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this research paper.

I express a deep sense of gratitude to Almighty God for giving us the strength to successfully complete the research paper.

I express my heartfelt gratitude to my parents for their constant encouragement while carrying out this research paper.

I express my deep sense of gratitude to the luminary, **The Principal Captain (Indian Navy) K Manikandan, Sainik School Amaravathinagar,** who has been continuously motivating and extending their helping hand to us.

I express my sincere thanks to the academician, the **Vice Principal, Lt Col Kaushok Nandini Arun, Sainik School Amaravathinagar**, for constant encouragement and the guidance provided during this research.

My sincere thanks to **Mr.Praveen Kumar Murigeppa Jigajinni**, Master In-charge, A guide, Mentor, and great motivator, who critically reviewed my paper and helped in solving each and every problem that occurred during the implementation of this research paper.

15. REFERENCES

[1]https://www.freecodecamp.org/news/time-complexity-of-algorithms/

[2]https://www.educative.io/edpresso/time-complexity-vs-space-complexity

[3]https://en.wikipedia.org/wiki/Space_complexity

Your Paper Index is : 202509-01-024001