NITTE (Deemed to be University) | NMAM INSTITUTE OF TECHNOLOGY

(An off-Campus Institution of NITTE (DEEMED TO BE UNIVERSITY), MANGALORE)

Department of Master of Computer Applications

A Task Report On

Sorting Algorithm Visualizer

First Semester Task Project, regards to the subject

DATA STRUCTURES AND ALGORITHMS

For the Academic year

2023-2024

Submitted by

Manish NNM23MC073

Kiranraj M NNM23MC067

Navaneeth NNM23MC081

Submitted To

Dr. Pallavi Shetty

(Dept. of MCA, Assistant Professor Gd-I)

1.INTRODUCTION

Sorting algorithms play a crucial role in computer science, enabling us to arrange data in a specific order efficiently.

Through this visualizer, we aim to enhance our understanding of different sorting techniques, including Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, and more.

The visual aspect of this project will allow us to witness how these algorithms operate step by step, as elements are rearranged and the data gradually becomes sorted. This will not only make it easier to comprehend the inner workings of each algorithm, but also highlight the differences in their efficiency and speed.

Furthermore, by visualizing the sorting process, we can gain valuable insights into optimizing algorithms, identifying potential performance issues, and comparing their complexity.

2.TECHNOLOGIES USED

- Software Used : IDLE Python
- Language used : Python

3.CONCEPTS INCLUDED

- Sorting Algorithm :
    - Bubble Sort
    - Insertion Sort
    - Quick Sort
    - Selection Sort
    - Merge Sort
    - Shell Sort
- Matplotlib (Python)

4.SRS

Functional Requirements

2.1 User Interface The visualizer should have an intuitive and user-friendly interface to allow users to interact with the software. The UI should display the following components: a) Input area: This area will allow users to input the data they want to sort. b) Sorting

algorithm selection: Users should be able to choose from a list of available sorting algorithms. c) Start/Stop buttons: Users can start and stop the sorting process. d) Animation speed control: Users should be able to control the speed of the animation to observe the sorting process at their desired pace. e) Display area: This area will visually represent the sorting process with animation and highlight each step taken by the algorithm.

2.2 Sorting Algorithms The visualizer should support a range of commonly used sorting algorithms, including but not limited to: a) Bubble Sort b) Insertion Sort c) Selection Sort d) Merge Sort e) Quick Sort f) shell Sort

2.3 Input Data a) The visualizer must be capable of handling various types of input data, such as integers, strings, or custom objects. b) Users should have the flexibility to input data in multiple ways, including manual entry, random generation, or loading from a file.

2.4 Animation and Visualization a) The visualization should clearly depict the sorting algorithm being executed. b) The software should highlight the current step, elements being compared, and elements being swapped for a better understanding of the sorting process. c) The visualizer should display the sorting progress in real-time, updating the display after each step.

Non-functional Requirements

3.1 Performance a) The visualizer should be capable of handling large datasets efficiently without significant performance degradation. b) The software should execute sorting algorithms within a reasonable time frame.

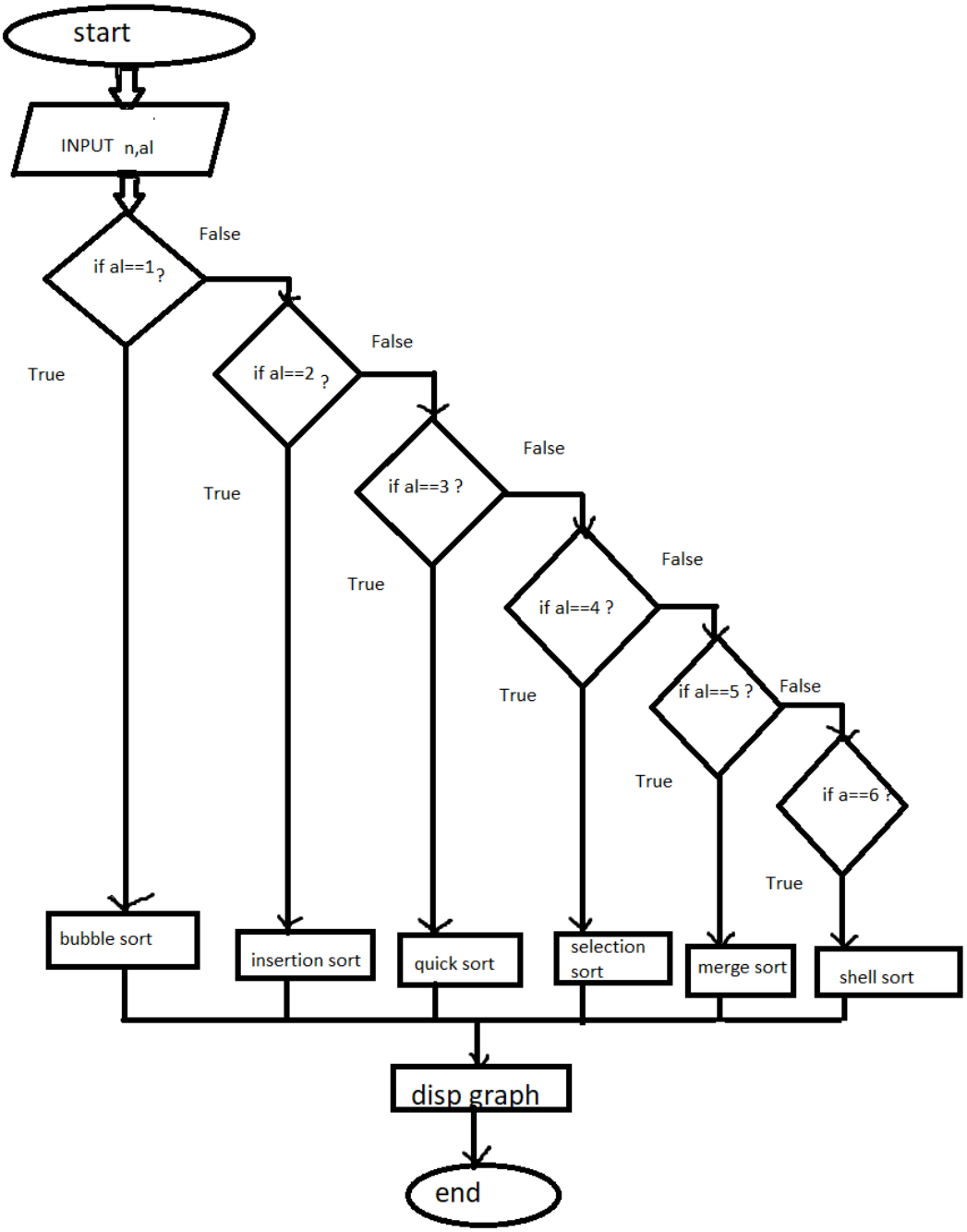
3.2 Usability a) The software should have an intuitive and easy-to-use interface, suitable for users with varying levels of technical expertise. b) Clear instructions and tooltips should be provided to guide users on how to interact with the visualizer.

3.3 Extensibility a) The structure of the visualizer should be modular and extensible, allowing for easy addition of new sorting algorithms in the future. b) Code should be well-documented, making it easier for other developers to understand and modify the visualizer.

Constraints a) The visualizer must be developed using the Python programming language. b) The software should be platform-independent and compatible with major operating systems, including Windows, macOS, and Linux.

Assumptions a) Users have a basic understanding of sorting algorithms and their general working principles. b) The visualizer will not handle extremely large datasets that may exceed memory limitations.

5.FLOWCHART/BLOCK DIAGRAM



6.POSSIBLE OUTCOME/SCOPE

- Education: It can be used as a teaching tool to visually demonstrate the working of different sorting algorithms like bubble sort, selection sort, merge sort, quick sort, etc. This can help students better understand the concepts and principles behind these algorithms.
- Learning and Practice: Students and programming enthusiasts can use this project to practice implementing different sorting algorithms and analyze their performance. They can experiment with different data sets and observe how the algorithms behave.
- Algorithm Analysis: Researchers and professionals involved in algorithm analysis can use this visualizer to compare and study the performance characteristics of different sorting algorithms. The visual representation can help in identifying the best algorithm for specific data sets and situations.
- Debugging and Optimization: Developers can utilize the visualizer for debugging purposes to identify any issues or inefficiencies in their sorting algorithm implementations. By witnessing the sorting process step by step, they can identify and address any logical errors or performance bottlenecks.
- Algorithm Design: The visualizer project can inspire developers to come up with new sorting algorithms or optimize existing ones. By observing the behavior of various sorting algorithms, one may brainstorm and experiment with new ideas to enhance performance or introduce unique features.
- Visualization and Data Analysis: The visual representation of the sorting algorithms can be useful for data visualization and analysis purposes. It can help in understanding patterns and relationships within large data sets, identifying outliers or anomalies, and gaining insights into data distributions.

7.CODE

```
import random
import matplotlib.pyplot as plt
import matplotlib.animation as anim


def swap(A, i, j):
    A[i],A[j]=A[j],A[i]


def sort_buble(arr):
```

```
    if (len(arr) == 1):
        return
    for i in range(len(arr) - 1):
        for j in range(len(arr) - 1 - i):
            if (arr[j] > arr[j + 1]):
                swap(arr, j, j + 1)
            yield arr

def insertion_sort(arr):
    if(len(arr)==1):
        return
    for i in range(1,len(arr)):
        j = i
        while(j>0 and arr[j-1]>arr[j]):
            swap(arr,j,j-1)
            j-=1
            yield arr

def quick_Sort(arr,p,q):
    if(p>=q):
        return
    piv = arr[q]
    pivindx = p
    for i in range(p,q):
        if(arr[i]<piv):
            swap(arr,i,pivindx)
            pivindx+=1
        yield arr
    swap(arr,q,pivindx)
    yield arr
    yield from quick_Sort(arr,p,pivindx-1)
    yield from quick_Sort(arr,pivindx+1,q)

def selection_sort(arr):
    for i in range(len(arr)-1):
        min = i
        for j in range(i+1,len(arr)):
            if(arr[j]<arr[min]):
                min=j
            yield arr
        if(min!=i):
            swap(arr,i,min)
            yield arr
```

```
def merge_sort(arr,lb,ub):
    if(ub<=lb):
        return
    elif(lb<ub):
        mid =(lb+ub)//2
        yield from merge_sort(arr,lb,mid)
        yield from merge_sort(arr,mid+1,ub)
        yield from merge(arr,lb,mid,ub)
        yield arr

def merge(arr,lb,mid,ub):
    new = []
    i = lb
    j = mid+1
    while(i<=mid and j<=ub):
        if(arr[i]<arr[j]):
            new.append(arr[i])
            i+=1
        else:
            new.append(arr[j])
            j+=1
    if(i>mid):
        while(j<=ub):
            new.append(arr[j])
            j+=1
    else:
        while(i<=mid):
            new.append(arr[i])
            i+=1
    for i,val in enumerate(new):
        arr[lb+i] = val
        yield arr

def shell_sort(arr):
    sublistcount = len(arr) // 2
    while sublistcount > 0:
      for start_position in range(sublistcount):
        yield  from gap_InsertionSort(arr, start_position, sublistcount)
      sublistcount = sublistcount // 2

def gap_InsertionSort(nlist,start,gap):
    for i in range(start+gap,len(nlist),gap):
        current_value = nlist[i]
        position = i
```

```
        while position>=gap and nlist[position-gap]>current_value:
            nlist[position]=nlist[position-gap]
            position = position-gap
            yield nlist
        nlist[position]=current_value
        yield nlist


n = int(input("Enter the number of elements : "))
print("Choose algorithm :\n 1.Bubble \n 2.Insertion \n 3.Quick \n 4.Selection \n 5.Merge
Sort \n 6.Shell \n ")
al = int(input("Enter Your choice : "))
array = [i + 1 for i in range(n)]
random.shuffle(array)

if(al==1):
    title = "Bubble Sort"
    algo = sort_buble(array)
elif(al==2):
    title = "Insertion Sort"
    algo = insertion_sort(array)
elif(al==3):
    title = "Quick Sort"
    algo = quick_Sort(array,0,n-1)
elif(al==4):
    title="Selection Sort"
    algo = selection_sort(array)
elif (al == 5):
    title = "Merge Sort"
    algo=merge_sort(array,0,n-1)
elif (al == 6):
    title = "Shell Sort"
    algo = shell_sort(array)

# Initialize fig
fig, ax = plt.subplots()
ax.set_title(title)
bar_rec = ax.bar(range(len(array)), array, align='edge', color='green')
ax.set_xlim(0, n)
ax.set_ylim(0, int(n * 1.1))
text = ax.text(0.02, 0.95, "", transform=ax.transAxes)
epochs = [0]

def update_plot(array, rec, epochs):
```

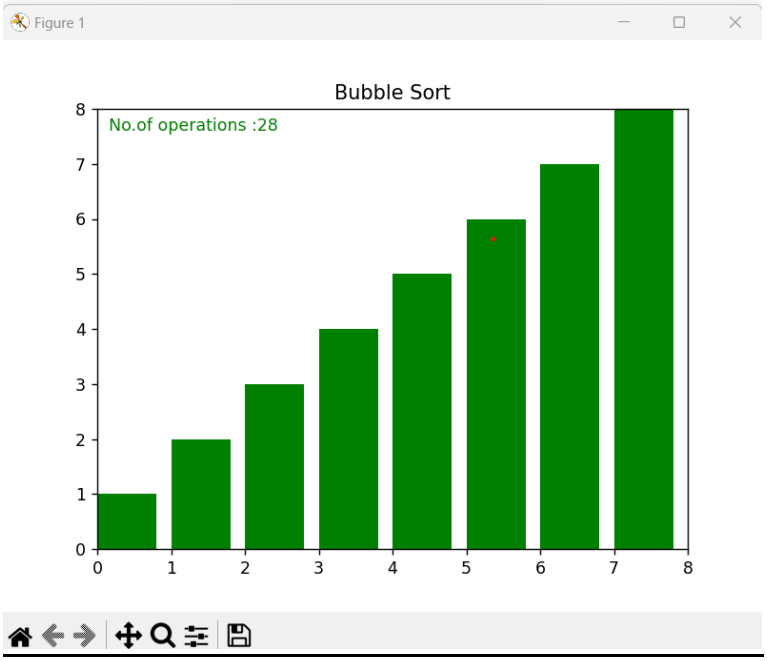
```
    for rec, val in zip(rec, array):
        rec.set_height(val)
    epochs[0]+= 1
    text.set_text("No.of operations :{}".format(epochs[0]))
    text.set_color('green')

anima = anim.FuncAnimation(fig, func=update_plot, fargs=(bar_rec, epochs), frames=algo,
interval=1, repeat=False)
plt.show()
```

8.OUTPUT

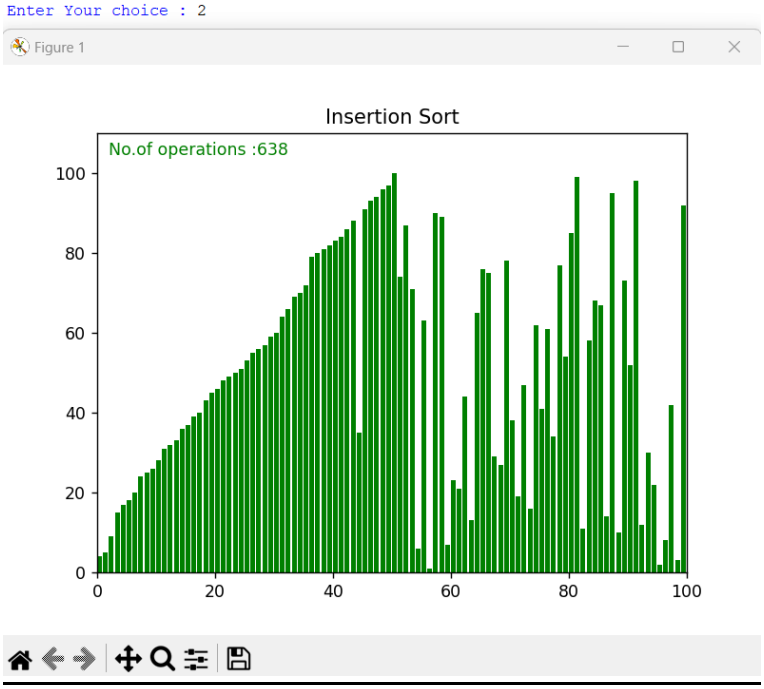
```
Enter the number of elements : 8
Choose algorithm :
 1.Bubble
 2.Insertion
 3.Quick
 4.Selection
 5.Merge Sort
 6.Shell

Enter Your choice : 1
```



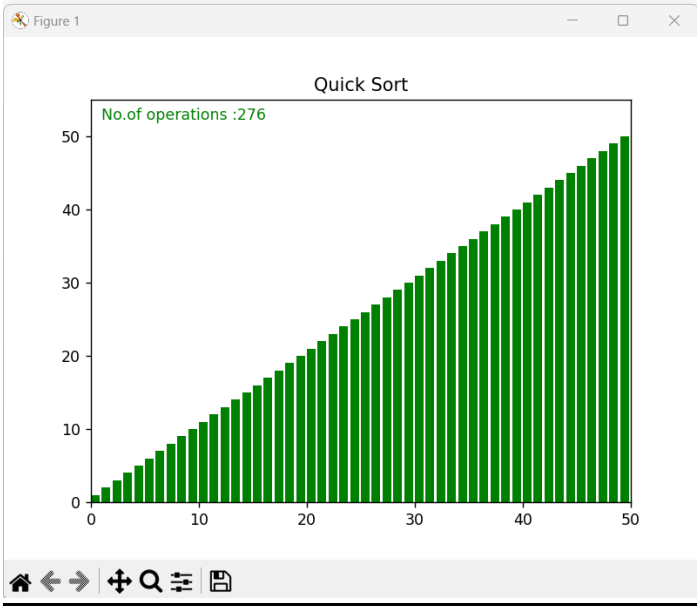
```
Enter the number of elements : 100
Choose algorithm :
 1.Bubble
 2.Insertion
 3.Quick
 4.Selection
 5.Merge Sort
 6.Shell

Enter Your choice : 2
```



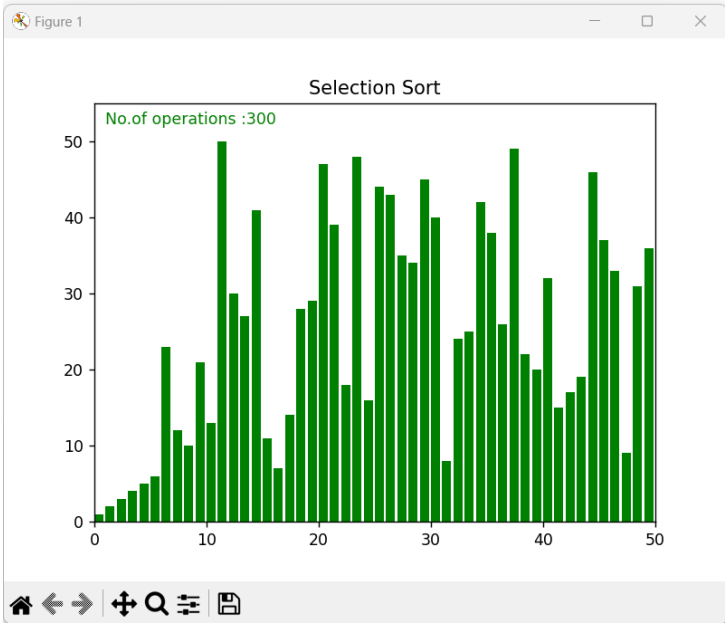

```
Enter the number of elements : 50
Choose algorithm :
 1.Bubble
 2.Insertion
 3.Quick
 4.Selection
 5.Merge Sort
 6.Shell

Enter Your choice : 3
```



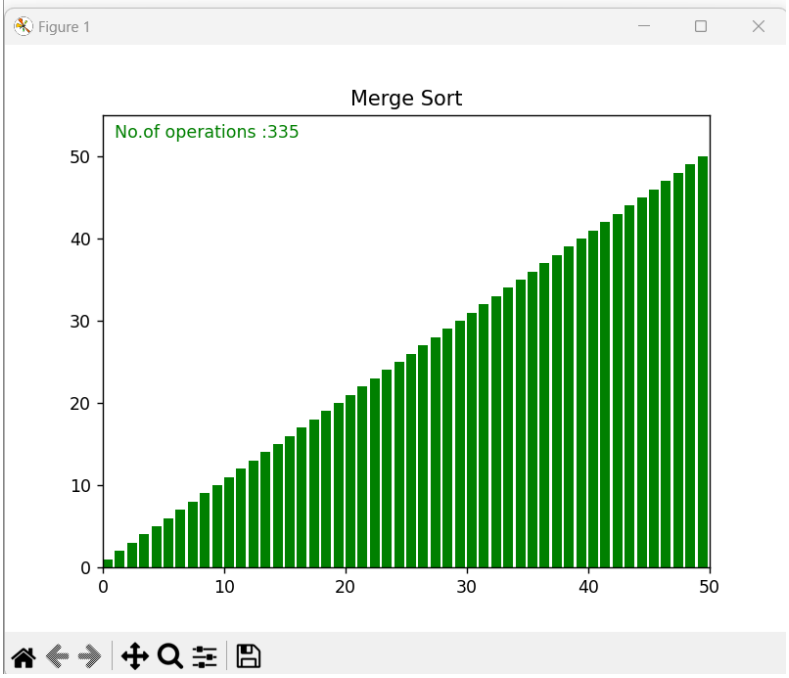
```
Enter the number of elements : 50
Choose algorithm :
 1.Bubble
 2.Insertion
 3.Quick
 4.Selection
 5.Merge Sort
 6.Shell

Enter Your choice : 4
```



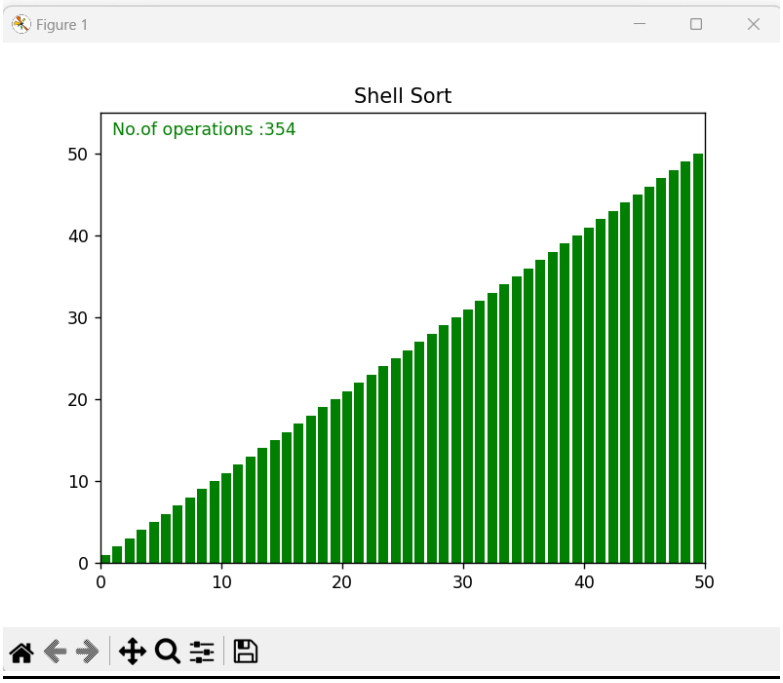

```
Enter the number of elements : 50
Choose algorithm :
 1.Bubble
 2.Insertion
 3.Quick
 4.Selection
 5.Merge Sort
 6.Shell

Enter Your choice : 5
```



```
Enter the number of elements : 50
Choose algorithm :
 1.Bubble
 2.Insertion
 3.Quick
 4.Selection
 5.Merge Sort
 6.Shell

Enter Your choice : 6
```



9.CONCLUSION

In conclusion, the Sorting Algorithm Analyzer project has successfully demonstrated the efficiency and performance of various sorting algorithms in Python. By implementing and analyzing sorting algorithms such as Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort, we were able to evaluate their time complexity and compare their execution time on different data sets.

Through careful analysis and comparison, it is clear that Merge Sort and Quick Sort outperform the other sorting algorithms in terms of time complexity, particularly when dealing with large data sets. These algorithms exhibit a time complexity of O(n log n), making them highly efficient for sorting operations.

However, it is important to consider other factors such as space complexity and stability when selecting a sorting algorithm for specific use cases. For instance, Bubble Sort and Insertion Sort, though less efficient in terms of time complexity, may be preferable for smaller data sets or when sorting stability is a priority.

Overall, this project has not only provided insights into the inner workings of various sorting algorithms but has also highlighted the significance of algorithm selection based on specific requirements. By utilizing the Sorting Algorithm Analyzer, programmers and developers can make informed decisions about which sorting algorithm to implement based

on the size and nature of the data set, ultimately optimizing the performance and efficiency of their sorting operations.

10.BIBILIOGRAPHY

- **GeeksforGeeks Sorting Algorithms** - https://www.geeksforgeeks.org/sorting-algorithms/

  A comprehensive website covering various sorting algorithms with detailed explanations and example code in Python.

- **Real Python** - Sorting Algorithms in Python - https://realpython.com/sorting-algorithms-python/

  This tutorial provides a step-by-step guide on implementing and analyzing sorting algorithms in Python.

- **Python Sorting Algorithms** - https://www.tutorialspoint.com/python_data_structure/python_sorting_algorithms.htm

- **Matplotlib** : https://www.w3schools.com/python/matplotlib_intro.asp