MINI-PROJECT REPORT

PROJECT TITLE: Sorting Visualizer

Student Name: Soumyadeep Jana	**UID: 25MCC20022**
Branch: MCA CCD	**Section/Group: MCC101**
Semester: 1	**Date of Performance:**
Subject Name: DAA	**SubjectCode: 25cap-612**

Index

Problem Statement -

The problem to be addressed in this project is to design and implement a graphical tool that demonstrates how fundamental sorting algorithms operate on unsorted data. Sorting is a foundational operation in computer science and forms the backbone of many higher-level operations, yet students often find it difficult to visualize internal comparisons and swaps. This project focuses on three widely taught sorting algorithms — Bubble Sort, Selection Sort, and Insertion Sort — and provides a stepwise animation that highlights the array elements being compared and the swaps executed during the process. The visualizer should allow users to generate random datasets of variable size, select any of the provided algorithms, and observe the sorting process in real time on a canvas. The intended output is both educational and interactive: it should clearly indicate which elements are involved in each step and display the final sorted sequence. If properly implemented, this tool will enhance conceptual understanding and make it easier to relate theoretical complexity to observable behavior in practice.

Aim / Overview of the Project -

The aim of this project is to create a user-friendly Python application that visually demonstrates the internal workings of common sorting algorithms, enabling students to connect algorithmic steps with tangible effects on data. The project uses a graphical interface built with Tkinter where integer values are represented as vertical bars; the height of each bar corresponds to the numeric value. The user can generate a dataset with randomized values and choose Bubble Sort, Selection Sort, or Insertion Sort, after which the chosen algorithm executes in an animated, step-by-step manner. Visual cues such as color changes mark comparisons and swaps, so the user can easily track progress and observe algorithm-specific behavior. The tool aims to support teaching and learning by making abstract algorithmic operations concrete and observable. It also serves as a demonstration platform to compare algorithm performance qualitatively. The project emphasizes clarity and simplicity, making it suitable for introductory courses in Design and Analysis of Algorithms while being extendable to more advanced sorting techniques in the future.

Technologies Used -

This project is implemented entirely in Python, which is selected for its readability and rapid development capabilities. The graphical user interface is developed using Tkinter, the standard GUI toolkit that comes bundled with Python. Tkinter provides widgets such as frames, buttons, comboboxes and canvas, which are used to create the control panel and the visual display area. The Python `random` module is used to produce randomized datasets so that each visualization run presents a new input scenario. The sorting algorithms are implemented as generator functions to allow controlled iteration and smooth animation; using generators enables the GUI to update between algorithm steps without blocking the main event loop. The project also uses simple event-driven programming techniques: button clicks trigger dataset generation and sorting, and a timer (`root.after`) schedules repeated generator advancement for animation. No external libraries are required, ensuring portability. The combination of Python, Tkinter, and generator-based animation provides a lightweight and educational environment for algorithm visualization.

Algorithm -

1. Start the program and initialize the graphical user interface using Tkinter.

2. Enter the size of the dataset and click the “Generate” button to create a random list of numbers.

3. Display the numbers as vertical bars where the height represents the value of each element.

4. Select one sorting algorithm (Bubble Sort, Selection Sort, or Insertion Sort) from the dropdown menu.

5. Click the “Start” button to begin sorting, and highlight the elements being compared.

6. If a swap is required, interchange the elements and update the bar colors to show the change.

7. Continue the above process until all elements become sorted in ascending order.

8. Display the final sorted list in the visualizer and stop the animation. End the program.

Dataset Used -

This project does not use a fixed dataset. Instead, it creates a dataset dynamically at runtime using Python's random number generator. The user is allowed to select the size of the dataset using a Spinbox control. Based on the selected size, random integers within a suitable numeric range are generated and stored inside a list. The height of each bar displayed in the visualizer represents the numeric value of the corresponding list element. Because the dataset changes every time it is generated, the sorting behavior can be tested under different input patterns such as best case, average case, or worst case. This randomness enhances learning of how algorithm performance changes with different data distributions.

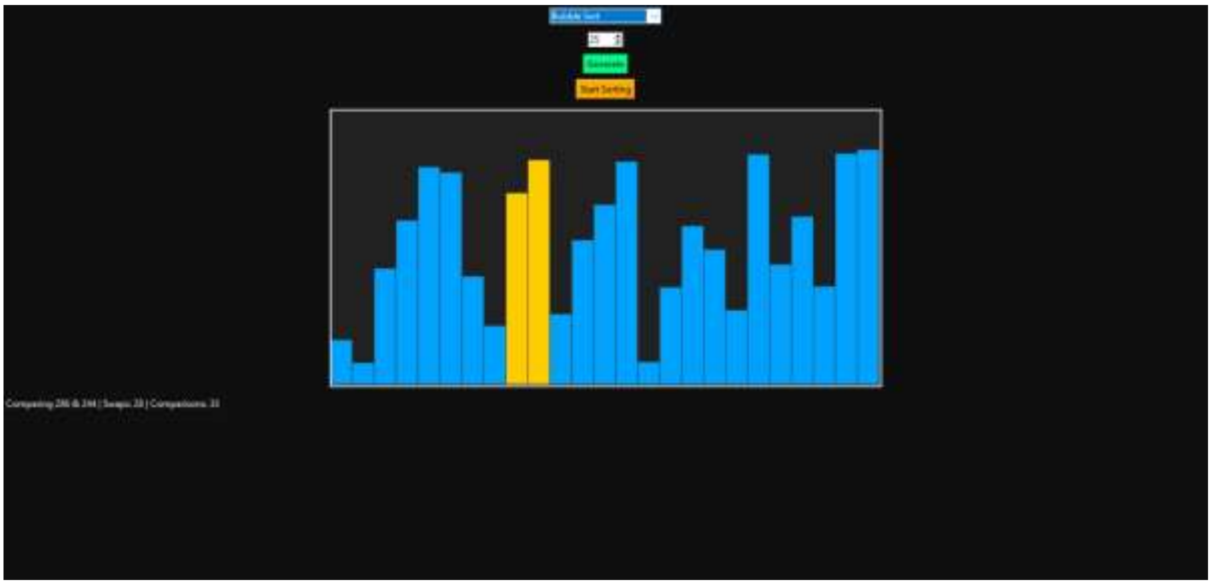
Code for Experiment -

```
import tkinter as tk
from tkinter import ttk
import random

# ---------- Globals ----------
data = []
comparisons = swaps = 0
delay = 150  # slower animation for clarity (ms)

# ---------- UI Drawin
def draw(arr, highlight=[]):
    canvas.delete("all")
    w = 700 / len(arr)
    for i, h in enumerate(arr):

        color = "#00a2ff" if i not in highlight else "#ffcc00"
        canvas.create_rectangle(i*w, 350-h, (i+1)*w, 350, fill=color, outline="#333")
    root.update_idletasks()

def update_status(text):
    status_var.set(text)
    root.update_idletasks()

# ---------- Sorting Algorithms
def bubble_sort():
    global comparisons, swaps
    n = len(data)
    for i in range(n-1):
        for j in range(n-i-1):
            comparisons += 1
            draw(data, [j, j+1])
            update_status(f"Comparing {data[j]} & {data[j+1]} | Swaps: {swaps} | Comparisons: 
{comparisons}")
            root.after(delay)
            if data[j] > data[j+1]:
                swaps += 1
                data[j], data[j+1] = data[j+1], data[j]
                draw(data, [j, j+1])
                root.after(delay)

def selection_sort():
    global comparisons, swaps
    n = len(data)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            comparisons += 1
            draw(data, [min_idx, j])
            update_status(f"Comparing {data[min_idx]} & {data[j]} | Swaps: {swaps} | Comparisons: 
{comparisons}")
            root.after(delay)
            if data[j] < data[min_idx]:
                min_idx = j
        if min_idx != i:
            swaps += 1
            data[i], data[min_idx] = data[min_idx], data[i]
            draw(data, [i, min_idx])
            root.after(delay)

def insertion_sort():
    global comparisons, swaps
    for i in range(1, len(data)):
        key = data[i]; j = i-1
        while j >= 0 and data[j] > key:
            comparisons += 1
            data[j+1] = data[j]
            swaps += 1
            draw(data, [j, j+1])
            update_status(f"Comparing {data[j]} & {key} | Swaps: {swaps} | Comparisons: {comparisons}")
            root.after(delay)
            j -= 1
        data[j+1] = key

def merge_sort(arr, l=0, r=None):
    global comparisons, swaps
    if r is None: r = len(arr)-1
    if l < r:
        m = (l+r)//2
        merge_sort(arr, l, m)
        merge_sort(arr, m+1, r)
        merge(arr, l, m, r)

def merge(arr, l, m, r):
    global comparisons, swaps
    L, R = arr[l:m+1], arr[m+1:r+1]
    i = j = 0; k = l
    while i < len(L) and j < len(R):

        comparisons += 1
        if L[i] <= R[j]:
            arr[k] = L[i]; i += 1
        else:
            arr[k] = R[j]; j += 1
            swaps += 1
        draw(data, [k]); update_status(f"Merging | Swaps: {swaps} | Comparisons: {comparisons}")
        root.after(delay); k += 1
    while i < len(L): arr[k]=L[i]; i+=1; k+=1; draw(data, [k-1]); root.after(delay)
    while j < len(R): arr[k]=R[j]; j+=1; k+=1; draw(data, [k-1]); root.after(delay)

def quick_sort(arr, low=0, high=None):
    if high is None: high = len(arr)-1
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi-1)
        quick_sort(arr, pi+1, high)

def partition(arr, low, high):
    global comparisons, swaps
    pivot = arr[high]
    i = low-1
    for j in range(low, high):
        comparisons += 1
        if arr[j] < pivot:
            i += 1; arr[i], arr[j] = arr[j], arr[i]; swaps += 1
        draw(data, [i,j,high]); update_status(f"QuickSort | Swaps: {swaps} | Comparisons: {comparisons}")
        root.after(delay)
    arr[i+1], arr[high] = arr[high], arr[i+1]; swaps += 1
    draw(data, [i+1, high]); root.after(delay)
    return i+1

def heap_sort():
    global comparisons, swaps
    n = len(data)
    def heapify(n,i):
        largest=i; l=2*i+1; r=2*i+2
        if l<n: comparisons+=1; draw(data, [i,l]); root.after(delay);
        if l<n and data[l]>data[largest]: largest=l
        if r<n: comparisons+=1; draw(data, [i,r]); root.after(delay)
        if r<n and data[r]>data[largest]: largest=r
        if largest!=i:
            data[i],data[largest]=data[largest],data[i]; swaps+=1
            draw(data,[i,largest]); root.after(delay)
            heapify(n,largest)
    for i in range(n//2-1,-1,-1): heapify(n,i)
    for i in range(n-1,0,-1): data[0],data[i]=data[i],data[0]; swaps+=1; draw(data,[0,i]); root.after(delay);
heapify(i,0)
```

```
# ---------- Run Sorting
def start_sort():
    global comparisons, swaps
    comparisons = swaps = 0
    algo = algo_menu.get()
    update_status(f"Running {algo} ...")
    root.update()
    if algo=="Bubble Sort": bubble_sort()
    elif algo=="Selection Sort": selection_sort()
    elif algo=="Insertion Sort": insertion_sort()
    elif algo=="Merge Sort": merge_sort(data)
    elif algo=="Quick Sort": quick_sort(data)
    elif algo=="Heap Sort": heap_sort()
    draw(data)
    update_status(f"Completed Swaps: {swaps} | Comparisons: {comparisons}")

# ---------- Generate Data
def generate():
    global data
    size = int(size_box.get())
    data = [random.randint(20, 300) for _ in range(size)]
    draw(data)
    update_status("Array generated ")

# ---------- UI Setup
root = tk.Tk()
root.title("Sorting Visualizer - Advanced")
root.config(bg="#111")

algo_menu = ttk.Combobox(root, values=["Bubble Sort","Selection Sort","Insertion Sort","Merge
Sort","Quick
Sort","Heap Sort"], state="readonly", width=20)
algo_menu.current(0); algo_menu.pack(pady=5)

size_box = tk.Spinbox(root, from_=5, to=50, width=5)
size_box.pack(pady=5); size_box.delete(0, tk.END); size_box.insert(0,"25")

tk.Button(root,text="Generate",command=generate,bg="#00ff88",fg="black").pack(pady=4)
tk.Button(root,text="Start Sorting",command=start_sort,bg="#ffaa00",fg="black").pack(pady=2)

canvas = tk.Canvas(root,width=700,height=350,bg="#222")
canvas.pack(pady=10)

status_var = tk.StringVar()
status_bar = tk.Label(root,textvariable=status_var,bg="#111",fg="white",anchor="w")
status_bar.pack(fill=tk.X)

generate()
root.mainloop()
```

Result / Output -

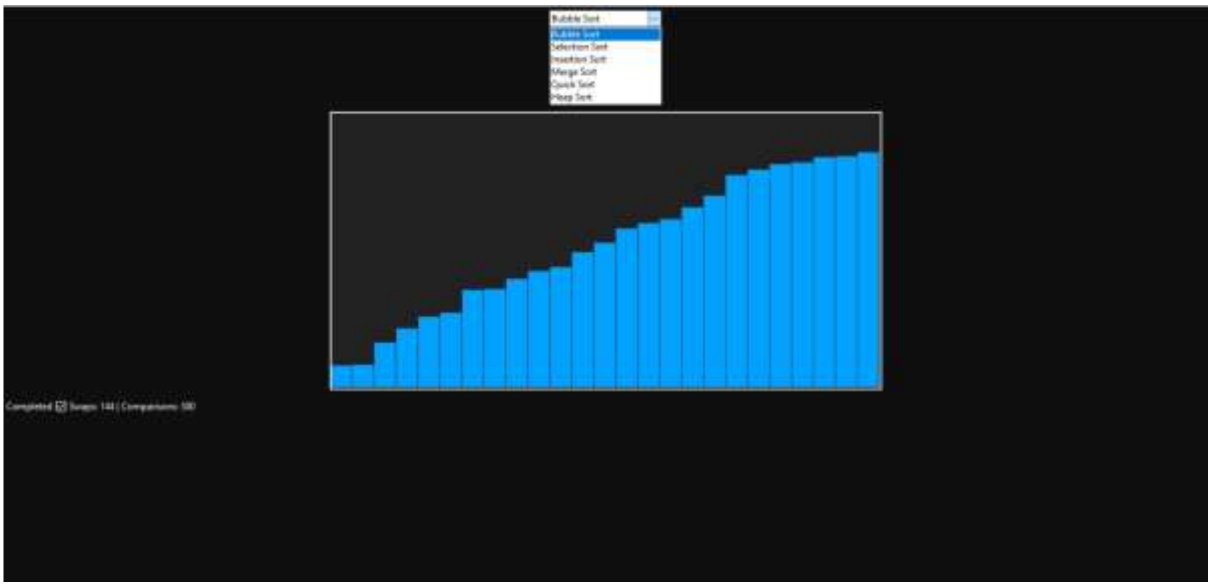
Screenshot - 1



Comparing 286 & 244 | Swaps: 20 | Comparisons: 31

Fig - 1 - Interface

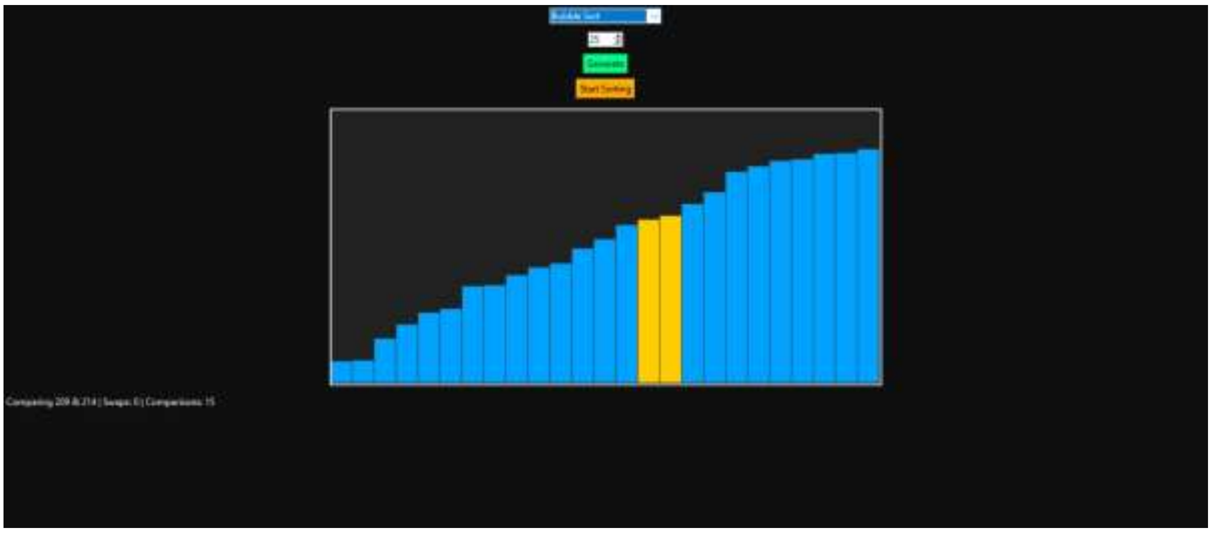
Screenshot - 2



Completed ✓ Swaps: 144 | Comparisons: 300

Fig - 2 - choosing the sorting algorithm

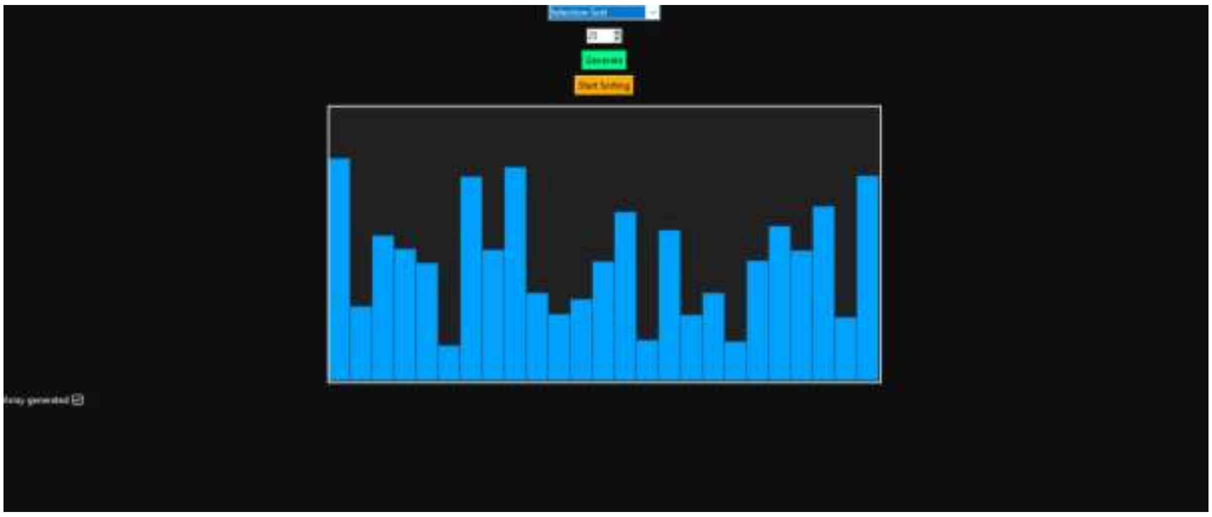
Screenshot 3 -



Comparing 209 & 214 | Swaps: 6 | Comparisons: 15

Fig - 3 - start sorting process

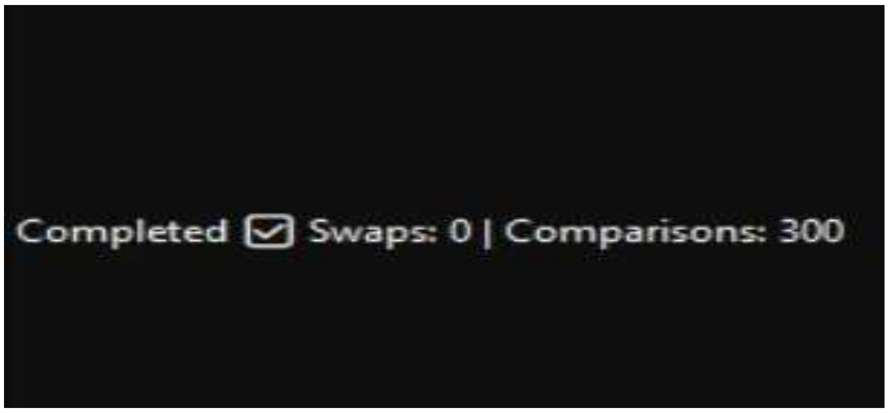
Screenshot - 4



Array generated ✅

Fig - 3 - sorting - done

Screenshot 5 -



```
Completed ☑ Swaps: 0 | Comparisons: 300
```

Fig 4 - complete message

Learning Outcomes -

Through this project, the student will gain practical and conceptual understanding of sorting algorithm mechanics and GUI-driven animation. Implementing Bubble, Selection, and Insertion Sort as generator functions reinforces knowledge about iteration control and event-driven programming paradigms. The student will learn how to map abstract array indices and operations to concrete visual elements, enabling better comprehension of algorithmic behavior. Building the interface with Tkinter develops skills in GUI layout, widget management, and canvas drawing. The use of randomized datasets offers experiential insights into best-case and worst-case scenarios, shedding light on algorithmic complexity and performance considerations. Additionally, the student will improve debugging and code-organization capabilities, following structured implementation that separates algorithm logic from visualization and control flow. Overall, the project strengthens algorithmic intuition and provides a demonstrable artifact for practical examination, viva voce discussions, and future extensions such as adding merge sort or quicksort visualizations.

Step-by-step Instructions to Run -

1. Install Python on your system if it is not already installed. Make sure Tkinter is included in the installation.

2. Open any Python IDE such as IDLE, VS Code, PyCharm, or simply use Command Prompt/Terminal.

3. Create a new Python file and name it as `sorting_visualizer.py`.

4. Copy and paste the complete Sorting Visualizer code into this file and save it.

5. Run the program. A GUI window will appear with sorting options and input controls.

6. Select the number of elements to generate using the Spinbox provided in the GUI.

7. Click on the “Generate” button to produce a new random dataset that appears as vertical bars.

8. Choose a sorting algorithm (Bubble Sort, Selection Sort, or Insertion Sort) from the dropdown list.

9. Click on the “Start” button to begin the sorting animation and observe step-by-step operations.

10. Wait until the bars are fully sorted and the final sorted result is visible on the screen.

11. Close the application window to end the program execution.

References -

1. GitHub -
2. Python Official Documentation
   Python Software Foundation. *Python 3 Documentation.* Available at: https://docs.python.org/3/
   Used for understanding Python syntax, built-in functions, and standard libraries like `random`.

3. Tkinter GUI Programming
   Tkinter documentation. *Python GUI Programming with Tkinter.* Available at: https://docs.python.org/3/library/tk.html
   Used for creating the graphical user interface, including canvas, buttons, labels, comboboxes, and Spinbox.

4. Sorting Algorithms – Concepts & Implementation
   GeeksforGeeks. *Sorting Algorithms in Python.* Available at: https://www.geeksforgeeks.org/sorting-algorithms/