UNIVERSITY INSTITUTE of COMPUTING
Asia's Fastest Growing University

NAAC GRADE A+
Accredited University

CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

DAA-Mini Project Report

On

Algorithm Analyzer & Visualizer

SUBMITTED BY:

Aditya Parmar

UID:25IMC13007

CLASS: MCA (AI&ML)

SECTION: 25MAM6-A

SUBMITTED TO:

Ms. Gurpreet Kaur

Assistant Professor

Acknowledgement

I would like to express my sincerest gratitude to all those who guided, supported, and encouraged me throughout the completion of this project.

First and foremost, I would like to thank Ms. Gurpreet Kaur, my project guide and mentor, for her invaluable guidance, continuous support, and constructive suggestions during the conceptualization, design, and implementation phases of this project. Her deep knowledge of algorithms and pedagogy helped me structure the project into clear objectives, choose suitable algorithms for demonstration, and implement effective visualization techniques that bridge theory and practice.

I am deeply grateful to the Department of Computer Science at [Chandigarh University] for providing the necessary infrastructure, laboratory facilities, and programming environment that made the development and testing of the software possible. The encouragement and prompt help from the lab assistants and staff also contributed significantly to timely completion.

I would also like to thank my classmates and friends who tested the application, provided feedback on the user interface, and helped with real-life testing scenarios. Their suggestions regarding UX flow, animation speed, and real-data examples improved the usability of the application.

Special thanks go to the authors and maintainers of the open-source libraries used in this project — particularly the Python community, NumPy, and Matplotlib — for making powerful and well-documented tools freely available. Their documentation and examples were extremely helpful during development.

Finally, I would like to thank my family for their constant encouragement and patience. Their moral support has been my backbone throughout the project timeline.

Aditya Parmar

MCA (AI & ML)

University Institute of Computing

Chandigarh University

Project Guide's

Ms. Gurpreet Kaur

Assistant Professor

Introduction

The study of algorithms forms the intellectual foundation of computer science. Every computational task, from sorting a list of names to routing data across the internet, relies on a well-designed algorithm that efficiently performs the required operations. In the subject *Design and Analysis of Algorithms (DAA)*, students are taught how to construct such algorithms, analyze their correctness, and evaluate their efficiency using mathematical tools such as time and space complexity. While this theoretical framework is extremely powerful, one major difficulty faced by learners is the lack of intuitive understanding of how algorithms actually behave step by step when executed on real data.

In most traditional classroom settings, algorithms are explained through pseudocode, flowcharts, and complexity equations. Although these are essential tools for theoretical understanding, they often fail to convey the *dynamic nature* of algorithmic processes. For example, while a student may know that Bubble Sort repeatedly compares adjacent elements and swaps them when necessary, it can still be difficult to visualize how the array changes with each iteration, or how many unnecessary passes it makes in practice. Similarly, it is challenging to mentally simulate recursive algorithms like Merge Sort or Quick Sort, where subarrays are repeatedly divided and merged, or to clearly imagine how Binary Search eliminates half of the search space at each step.

This gap between theoretical knowledge and mental visualization often leads to superficial understanding. Many students can recall pseudocode and state complexities, but they struggle to explain *why* a particular algorithm is faster or more efficient than another in specific situations. This observation forms the core motivation behind the project titled "Algorithm Visualizer Pro."

Algorithm Visualizer Pro is an educational and interactive software application that allows students to *see algorithms in action.* It is developed in Python using the Tkinter library for the graphical user interface and Matplotlib for real-time visual plotting. The application provides users with an intuitive interface where they can select an algorithm, generate a random dataset, control the execution speed, and watch as the algorithm executes step by step through animated bar charts. Each bar represents an element of the dataset, and during sorting or searching, specific bars are highlighted to indicate comparisons, swaps, or target positions.

The visual representation of algorithm execution offers learners an immediate, concrete view of abstract processes. Instead of reading dry code or tracing loops on paper, students can observe patterns, identify inefficiencies, and understand the underlying mechanics of algorithmic behavior. For instance, when visualizing Bubble Sort, the student can directly see why it has quadratic time complexity, as each pass makes multiple redundant comparisons. In contrast, the smoother, faster behavior of Merge Sort or Quick Sort vividly demonstrates the benefit of the divide-and-conquer paradigm and the $O(n\log\ n)$performance.

Objective of the Project

The primary goal of the **Algorithm Visualizer Pro** project is to facilitate deeper understanding of classic algorithms through interactive visualization, performance instrumentation, and comparative analysis. Below is an exhaustive list of the project's objectives, each tied to the educational aims and practical features of the application.

1. Pedagogical Clarity

- **Objective:** Translate abstract algorithmic steps from pseudocode into vivid visual representations that expose internal state changes.
- **Explanation:** For example, during Merge Sort, the split-and-merge process is animated so that learners can see how subarrays are combined. For Quick Sort, pivot selection and partitioning are visualized to illustrate why certain pivot choices cause degeneration.

2. Empirical Validation of Theoretical Complexity

- **Objective:** Enable students to compare theoretical asymptotic complexity (e.g., $O(n \log n)$ vs $O(n^2)$) with empirical runtime measurements across different input sizes.
- **Explanation:** The app records timing and allows repeated runs, enabling plotting of execution time vs input size and observing trends that match big-O predictions.

3. Interactive Experimentation

- **Objective:** Provide controls for generating various input datasets (random, sorted, reverse-sorted, nearly-sorted) to observe algorithmic behavior under different conditions.
- **Explanation:** This helps to illustrate best-case, average-case, and worst-case performance in a hands-on manner.

4. Modularity & Extensibility

- **Objective:** Build the system with modular components (algorithm engines, visualization generators, UI controller) to make adding new algorithms or visual features straightforward.
- **Explanation:** New modules such as Heap Sort, Counting Sort, or graph algorithms can be integrated with pre-existing UI hooks.

5. User-Friendliness & Accessibility

- **Objective:** Design a simple and intuitive GUI that lowers the barrier to interactive algorithm exploration for undergraduate students.
- **Explanation:** Clear labeling, speed control, start/stop buttons, and contextual messages aim to make the tool accessible even to first-time users.

6. Real-time Visual Feedback

- **Objective:** Provide near-real-time animation of algorithmic steps, with controls for speed and step-by-step execution.

Technologies Used

Programming Language

- **Python 3.8+ (recommended 3.10–3.11)**
  Python was chosen due to its readability, rapid prototyping capabilities, and rich ecosystem of visualization libraries. It allows implementing algorithms succinctly while keeping the code accessible for undergraduate students.

GUI Framework

- **Tkinter** (Python's standard GUI library)
  Tkinter is bundled with Python and provides lightweight, easy-to-use widgets (buttons, frames, Canvas). It is suitable for desktop academic projects and provides compatibility across Windows, Linux, and macOS.

Plotting & Visualization

- **Matplotlib**
  Used to draw bar charts representing arrays and to animate sorting steps when embedded within a Tkinter frame via FigureCanvasTkAgg. Matplotlib offers fine-grained control over plotting and is widely used in academic settings.

Numerical Utilities

- **NumPy** (optional but recommended)
  Useful for efficient array generation and numeric operations. Although the project does not require heavy numerical computation, NumPy simplifies random array generation and array copying.

Threading & Timing

- **Python threading module**
  To run animation loops without freezing the GUI mainloop, algorithm execution is launched in a background thread. This approach keeps the UI responsive and allows stop/resume controls.
- **Python time module**
  For precise measurement of algorithm execution time (using perf_counter) and for sleep intervals that control animation speed.

Development Tools / IDE

- **Visual Studio Code** or **PyCharm** (recommended)
  Offer debugging utilities, syntax highlighting, and easy package management.

Target Platform

- Desktop environment supporting Python 3 (Windows, Linux, macOS). No specialized hardware is required.

Installation & Setup Summary

Install Python 3.x and then (from a terminal):

pip install matplotlib numpy

(If NumPy is not desired, core functionality still works but NumPy simplifies array handling.)

Requirements

Functional Requirements

1. **Algorithm Selection:** The GUI must allow the user to choose among multiple algorithms (Bubble, Merge, Quick, Linear Search, Binary Search).
2. **Data Generation:** The system must generate randomized arrays of user-specified sizes, as well as accept optional custom input.
3. **Algorithm Execution:** The chosen algorithm should execute while the GUI animates each key step (comparison, swap, merge, pivot selection).
4. **Control Panel:** The user must be able to start, pause/stop, and control animation speed.
5. **Performance Measurement:** The system must measure and display execution time and optionally compare average times across multiple runs.

Non-Functional Requirements

1. **Usability:** Intuitive user interface with labels, tooltips, and a small learning curve for new users.
2. **Responsiveness:** The UI should remain responsive during algorithm execution (use of threading).
3. **Portability:** The application should run on any system with Python and required packages installed.
4. **Extensibility:** Modular architecture that allows adding more algorithms or visualization types with minimal changes.
5. **Performance:** Animations must run smoothly for small-to-moderate array sizes (recommended ≤ 100–500 for animation).

Hardware Requirements (Minimum)

- CPU: Intel i3 or equivalent
- RAM: 4 GB
- Disk: 200 MB free
- Display: 1024×768 resolution (preferably HD)

Software Requirements

- OS: Windows 10/11, Ubuntu 18.04+, or macOS
- Language: Python 3.8+
- Libraries: Matplotlib, NumPy (optional)

Source Code

```
import tkinter as tk
from tkinter import ttk, messagebox
import random, time, threading
import numpy as np
import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
def bubble_sort_gen(arr):
    a = arr.copy()
    n = len(a)
    for i in range(n):
        for j in range(0, n - i - 1):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
                yield a, (j, j+1)
    yield a, None


def merge_sort_gen_helper(a, l, r, states):
    if r - l <= 1:
        return
    m = (l + r) // 2
    merge_sort_gen_helper(a, l, m, states)
    merge_sort_gen_helper(a, m, r, states)
    left = a[l:m]; right = a[m:r]
    i = j = 0; k = l
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            a[k] = left[i]; i += 1
        else:
            a[k] = right[j]; j += 1
        states.append((a.copy(), (k,)))
        k += 1
```

```
    while i < len(left):
        a[k] = left[i]; i += 1; k += 1
        states.append((a.copy(), (k-1,)))
    while j < len(right):
        a[k] = right[j]; j += 1; k += 1
        states.append((a.copy(), (k-1,)))


def merge_sort_gen(arr):
    a = arr.copy()
    states = []
    merge_sort_gen_helper(a, 0, len(a), states)
    for s in states:
        yield s
    yield a, None


def quick_sort_gen(arr):
    a = arr.copy()
    states = []
    def qs(l, r):
        if l >= r:
            return
        pivot = a[(l+r)//2]
        i, j = l, r
        while i <= j:
            while a[i] < pivot: i += 1
            while a[j] > pivot: j -= 1
            if i <= j:
                a[i], a[j] = a[j], a[i]
                states.append((a.copy(), (i, j)))
                i += 1; j -= 1
        qs(l, j); qs(i, r)
    qs(0, len(a)-1)
    for s in states:
        yield s
    yield a, None
def linear_search(arr, key):
```

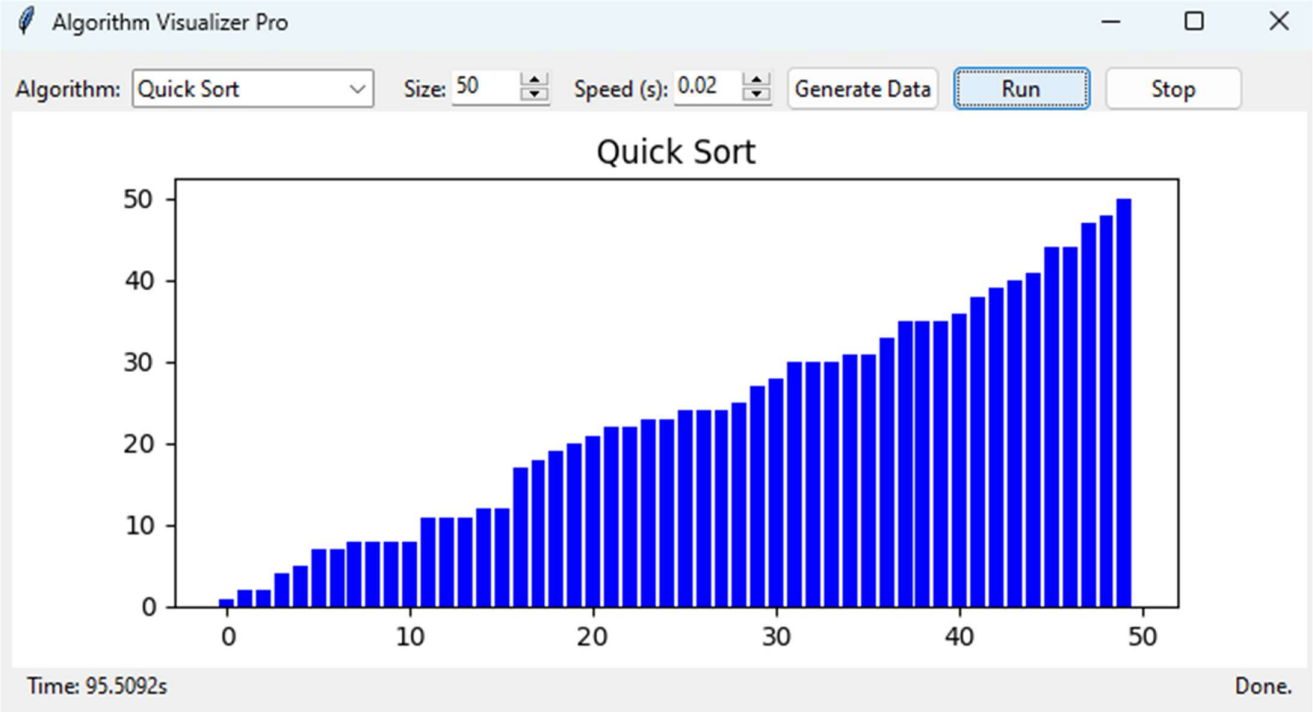
```
    for i, v in enumerate(arr):
        if v == key:
            return i
    return -1

def binary_search(arr, key):
    lo, hi = 0, len(arr)-1
    while lo <= hi:
        mid = (lo+hi)//2
        if arr[mid] == key: return mid
        if arr[mid] < key: lo = mid+1
        else: hi = mid-1
    return -1

class VisualizerApp:
    def __init__(self, root):
        self.root = root
        root.title("Algorithm Visualizer Pro")
        self.alg_var = tk.StringVar(value="Bubble Sort")
        self.size_var = tk.IntVar(value=50)
        self.speed_var = tk.DoubleVar(value=0.02)
        self.search_key_var = tk.IntVar(value=0)
        self.array = []
        self._build_ui()
        self.is_running = False

    def _build_ui(self):
        frm = ttk.Frame(self.root, padding=8)
        frm.pack(fill=tk.BOTH, expand=True)

        ctrl = ttk.Frame(frm)
        ctrl.pack(side=tk.TOP, fill=tk.X)
        ttk.Label(ctrl, text="Algorithm:").pack(side=tk.LEFT)
        ttk.Combobox(ctrl, textvariable=self.alg_var, values=[
            "Bubble Sort", "Merge Sort", "Quick Sort", "Linear Search", "Binary Search"
        ], width=18).pack(side=tk.LEFT, padx=4)
```

```
    ttk.Label(ctrl, text="Size:").pack(side=tk.LEFT, padx=(10,0))
    ttk.Spinbox(ctrl, from_=5, to=500, textvariable=self.size_var, width=6).pack(side=tk.LEFT)
    ttk.Label(ctrl, text="Speed (s):").pack(side=tk.LEFT, padx=(10,0))
    ttk.Spinbox(ctrl, from_=0.0, to=1.0, increment=0.01, textvariable=self.speed_var, width=6).pack(side=tk.LEFT)
    ttk.Button(ctrl, text="Generate Data", command=self.generate_data).pack(side=tk.LEFT, padx=6)
    ttk.Button(ctrl, text="Run", command=self.start).pack(side=tk.LEFT)
    ttk.Button(ctrl, text="Stop", command=self.stop).pack(side=tk.LEFT, padx=6)

    # canvas for matplotlib
    self.fig = Figure(figsize=(7,3))
    self.ax = self.fig.add_subplot(111)
    self.canvas = FigureCanvasTkAgg(self.fig, master=frm)
    self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    # status bar
    status = ttk.Frame(frm)
    status.pack(fill=tk.X)
    self.time_label = ttk.Label(status, text="Time: 0.000s")
    self.time_label.pack(side=tk.LEFT, padx=6)
    self.info_label = ttk.Label(status, text="")
    self.info_label.pack(side=tk.RIGHT, padx=6)

def generate_data(self):
    n = self.size_var.get()
    self.array = [random.randint(1, n) for _ in range(n)]
    self.draw_bars(self.array, color='blue')
    self.info_label.config(text=f"Generated random array of size {n}")

def draw_bars(self, arr, color='blue', highlight=None):
    self.ax.clear()
    x = list(range(len(arr)))
    self.ax.bar(x, arr, color=color)
    if highlight:
        for idx in (highlight if isinstance(highlight, (list,tuple)) else [highlight]):
            if idx is None: continue
            if 0 <= idx < len(arr):
```

```
                self.ax.bar(idx, arr[idx], color='red')
        self.ax.set_title(self.alg_var.get())
        self.canvas.draw()
    def start(self):
        if self.is_running:
            return
        alg = self.alg_var.get()
        if not self.array:
            self.generate_data()
        self.is_running = True
        t = threading.Thread(target=self.run_algorithm, args=(alg,))
        t.daemon = True
        t.start()
    def stop(self):
        self.is_running = False
    def run_algorithm(self, alg):
        arr = self.array.copy()
        t0 = time.perf_counter()
        if alg == "Bubble Sort":
            gen = bubble_sort_gen(arr)
            for state, highlight in gen:
                if not self.is_running: break
                self.draw_bars(state, highlight=highlight)
                time.sleep(self.speed_var.get())
        elif alg == "Merge Sort":
            gen = merge_sort_gen(arr)
            for state, highlight in gen:
                if not self.is_running: break
                self.draw_bars(state, color='green', highlight=highlight)
                time.sleep(self.speed_var.get())
        elif alg == "Quick Sort":
            gen = quick_sort_gen(arr)
            for state, highlight in gen:
                if not self.is_running: break
                self.draw_bars(state, color='purple', highlight=highlight)
                time.sleep(self.speed_var.get())
```

```
        elif alg == "Linear Search":
            key = random.choice(arr)
            found = -1
            for i, v in enumerate(arr):
                if not self.is_running: break
                self.draw_bars(arr, highlight=i)
                time.sleep(self.speed_var.get())
                if v == key:
                    found = i
                    break
            messagebox.showinfo("Linear Search", f"Key {key} found at index {found}" if found!=-1 else "Not found")
        elif alg == "Binary Search":
            arr_sorted = sorted(arr)
            key = random.choice(arr_sorted)
            lo, hi = 0, len(arr_sorted)-1
            found = -1
            while lo <= hi and self.is_running:
                mid = (lo+hi)//2
                self.draw_bars(arr_sorted, highlight=mid)
                time.sleep(self.speed_var.get())
                if arr_sorted[mid] == key:
                    found = mid; break
                elif arr_sorted[mid] < key:
                    lo = mid+1
                else:
                    hi = mid-1
            messagebox.showinfo("Binary Search", f"Key {key} found at index {found}" if found!=-1 else "Not found")
        t1 = time.perf_counter()
        self.is_running = False
        self.time_label.config(text=f"Time: {t1-t0:.4f}s")
        self.info_label.config(text="Done.")
if __name__ == "__main__":
    root = tk.Tk()
    app = VisualizerApp(root)
    root.mainloop()
```

10. Output Screens



Future problems

While Algorithm Visualizer Pro establishes a strong baseline, there are many directions to extend its functionality to make it more powerful, research-ready, or classroom-ready. Below are potential future problems (features and research questions) that can be pursued:

1. Expand Algorithm Library

- Description: Add a larger set of algorithms such as Heap Sort, Radix Sort, Counting Sort, Shell Sort for sorting; Interpolation Search for searching; and graph algorithms like BFS/DFS visualization, Bellman-Ford, Floyd-Warshall, A* pathfinding.
- Challenge: Each algorithm may require a bespoke visualization scheme (e.g., heap shape for Heap Sort), which demands UI design considerations.

2. Stable vs Unstable Sort Demonstrator

- Description: Implement a mode to demonstrate stability in sorting algorithms using item labels and highlight how stable sorts preserve order while unstable sorts do not.
- Challenge: Clear visual encoding of item identity beyond value (e.g., color or tag) to illustrate order preservation.

3. Performance Scaling Dashboard

- Description: Add automated benchmarking across a range of input sizes and data distributions (random, sorted, nearly sorted, reverse) and plot runtime curves with regression lines.
- Challenge: Resource management and UI responsiveness when running multiple measurement batches; perhaps offload to background worker processes.

4. Step-Wise Execution & Debugging Mode

- Description: Provide step-forward and step-backward controls, allowing users to navigate through algorithm states like a debugger.
- Challenge: Requires checkpointing or replay log so that the state can be rolled backward reliably.

5. Parallel & External Sorting Visuals

- Description: Visualize parallel sorting strategies or external/distributed sorts (e.g., merge passes on disk blocks).
- Challenge: Visual metaphors for parallel threads/processes and I/O behavior; also more complex to implement.

Conclusion

The **Algorithm Visualizer Pro** project successfully demonstrates how algorithmic theory and practical behavior can be bridged through visualization. It provides a hands-on environment for experimenting with algorithms and understanding their stepwise execution. The application caters to different learning styles by combining textual explanation, interactive controls, and visual evidence.

This project will assist students in making the jump from reading algorithm pseudocode to developing an intuitive and practical understanding of algorithm mechanics. The combination of modular code, a responsive GUI, and measurable performance analytics ensures that the tool is both educational and extensible.

Moving forward, the project can be expanded in many dimensions to support additional algorithms, web deployment, advanced analytics, and assessment integration — making it a valuable platform for algorithm education and research.

his real-time animation serves as a powerful pedagogical tool. For example, observing how **Bubble Sort** repeatedly pushes the largest unsorted element to the end clarifies why its time complexity is $O(n^2)$. Visualizing **Merge Sort** highlights how recursive division leads to efficient merging in $O(n\log\ n)$time, while **Quick Sort** demonstrates the importance of pivot selection and partitioning. Similarly, when running **Linear Search** or **Binary Search**, students can see how the number of comparisons decreases drastically as the dataset is sorted and search intervals are halved. Through these visual experiences, theoretical complexities transform into tangible understanding.

In conclusion, **Algorithm Visualizer Pro** successfully fulfills its objectives. It offers a creative and practical approach to algorithm education by combining theoretical knowledge, coding skills, and visual communication. It redefines how students interact with algorithms — turning them from lines of code into living processes that can be observed, analyzed, and understood. Through its engaging interface and well-structured design, the project not only enhances algorithmic intuition but also builds confidence in applying these concepts to real-world problem-solving.

Ultimately, this project reflects a fundamental truth about computer science: that understanding comes not just from knowing the *result* of an algorithm, but from appreciating the *process* through which it reaches that result. **Algorithm Visualizer Pro** helps learners witness that process vividly, empowering them to become better programmers, thinkers, and problem-solvers.

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press. — A standard reference covering algorithm design paradigms, complexity analysis, and a comprehensive set of algorithms that inspired examples used in this project.
- Levitin, A. (2012). *Introduction to the Design & Analysis of Algorithms* (3rd ed.). Pearson. — Provides pedagogical explanations and problem sets for algorithm design paradigms and analysis.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley. — Particularly useful for algorithm visualizations and practical implementation details for sorting and searching.
- Python Software Foundation. (2023). *Python Language Reference, version 3.x.* https://docs.python.org/3/ — Official Python documentation for language syntax and standard libraries used.
- Hunter, J. D. (2007). *Matplotlib: A 2D Graphics Environment.* Computing in Science & Engineering, 9(3), 90–95. — Documentation and tutorial resources used for embedding charts and animations.
- Oliphant, T. E. (2006). *A Guide to NumPy.* Trelgol Publishing. — For efficient numerical operations and handling of arrays (optional but useful for larger datasets).
- TkDocs. (n.d.). *Tkinter Tutorial.* https://tkdocs.com/ — Practical guide for building Tkinter UIs and embedding Matplotlib figures.
- GeeksforGeeks, Stack Overflow, and various online algorithm visualizer resources — Used for implementation ideas, debugging help, and visualization techniques.
- "Algorithm Visualizations" and educational websites (e.g., VisuAlgo by Dr. Steven Halim) — inspiration for visual metaphors and animation techniques.