



SORTING ALGORITHM VISUALIZATION

A PROJECT REPORT

Submitted by:

Deepanshu Saini (23BCS13189)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE & ENGINEERING



6 November, 2025

TABLE OF CONTENTS

| SNO | TITLE | PAGE NUMBER |
|-----|-------------------------------|-------------|
| 1 | INTRODUCTION | 3 |
| 2 | OBJECTIVE | 4 |
| 3 | DESCRIPTION | 5-6 |
| 4 | SOURCE CODE | 7-15 |
| 5 | OUTPUT | 16-17 |
| 6 | SCOPE OF PROJECT | 18 |
| 7 | FUTURE DEVELOPMENT OF PROJECT | 19 |
| 8 | CONCLUSION | 20 |



INTRODUCTION

Sorting algorithms are fundamental in computer science, playing a key role in optimizing data processing, searching, and organizing tasks. Understanding how these algorithms work is crucial for anyone studying algorithms and data structures. This project presents a **Sorting Algorithm Visualizer**, implemented in Python using the Pygame library, designed to offer a visual and interactive understanding of various sorting techniques.

The visualizer allows users to see how different algorithms such as **Bubble Sort**, **Insertion Sort**, **Selection Sort**, **QuickSort**, and **MergeSort** sort a randomly generated list of numbers. With a user-friendly graphical interface, users can choose between ascending and descending sorting modes, reset the list to a new random set of numbers, and select from different sorting algorithms to observe how they work in real time.

Each sorting algorithm is represented visually by bars whose heights correspond to the values being sorted. As the sorting progresses, users can observe the comparison and swapping of elements through color changes, providing a clear, step-by-step depiction of the inner workings of the algorithm. Overall, this **Sorting Algorithm Visualizer** serves as an educational platform, offering a hands-on experience for learning sorting algorithms in a visually engaging and interactive way. Whether used for teaching, self-learning, or demonstration purposes, it provides a comprehensive view of the fundamental sorting mechanisms that are crucial in various computer science applications.

Overall, this Sorting Algorithm Visualizer serves as an educational platform, offering a hands-on, engaging experience for learning sorting algorithms. Whether used for teaching, self-learning, or demonstration purposes, the tool provides a comprehensive understanding of the fundamental sorting mechanisms that are critical in various computer science applications. It bridges theory and practice, making complex algorithms accessible through an interactive visual interface, ultimately fostering a deeper appreciation for algorithm design and analysis.



OBJECTIVE

The objective of this project is to develop a **GUI-based Sorting Algorithm Visualizer** using Python and the Pygame library. The tool enables users to visualize the process of sorting algorithms, allowing them to choose from several well-known algorithms, including **Bubble Sort, Insertion Sort, Selection Sort, QuickSort, and MergeSort**, and observe the sorting process step by step.

The visualizer displays the sorting process dynamically, representing elements as bars whose heights correspond to the values being sorted. As the algorithm progresses, the tool visually highlights key actions, such as comparisons and swaps, through color changes, making it easier to follow how each algorithm operates.

The user interface, designed with Pygame, offers controls to reset the list, choose the sorting order (ascending or descending), and switch between different algorithms. The tool also provides options to adjust the speed of visualization, enhancing the user's ability to study each sorting algorithm at their own pace.

The objective of this project is to provide an interactive and educational platform for users to easily understand and compare the behaviors of different sorting algorithms through visual representation, aiding in the comprehension of fundamental concepts in data structures and algorithms.

This visualizer can be a valuable educational tool in classrooms, coding bootcamps, or self-study environments. It offers a comprehensive understanding of algorithm efficiency, allowing users to compare how different algorithms perform with the same dataset. By experimenting with different algorithms and inputs, users can also observe time complexities in action, gaining insight into which algorithms perform better in various scenarios.

Overall, the objective of this project is to provide an interactive, engaging, and educational platform for users to easily understand and compare the behaviors of different sorting algorithms through visual representation. It not only aids in the comprehension of fundamental concepts in data structures and algorithms but also encourages hands-on experimentation, promoting deeper learning and critical thinking.



DESCRIPTION

The project is a visual tool built using the **Python programming language** and the **Pygame** library to provide users with an interactive experience of sorting algorithms. The primary objective is to allow users to select and visualize different sorting algorithms, such as **Bubble Sort**, **Insertion Sort**, **Selection Sort**, **QuickSort**, and **MergeSort**, to see how data is sorted step by step.

The GUI interface is designed to make it simple and intuitive for users to interact with. It offers several options, including the ability to generate random data sets, choose a sorting algorithm, and toggle between **ascending** or **descending** order. The sorting process is displayed as bars of varying heights, where the heights represent the values being sorted, and color changes indicate comparisons and swaps made by the algorithm.

The interface consists of various input fields and buttons, including:

- **Algorithm Selection:** Users can choose from multiple sorting algorithms such as Bubble Sort, Insertion Sort, QuickSort, and others.
- **Generate New List:** Users can reset the data set by generating a new list of random numbers to sort.
- **Sorting Order Toggle:** Users can switch between **ascending** or **descending** order for sorting.
- **Visualization Speed Control:** This feature allows users to adjust the speed of the sorting process to observe each step more closely.
- **Start/Stop Sorting:** Users can control the sorting process, starting or stopping it as needed, allowing them to closely follow how the algorithm progresses.

Each sorting algorithm's process is shown in real-time, highlighting key operations such as element comparisons, swaps, and final placements.

This enables users to understand the efficiency and behavior of different sorting algorithms in a more engaging and educational way.

In addition to its educational value, the tool also serves as a practical application for anyone studying sorting algorithms or needing a deeper understanding of their behavior in data organization.



SOURCE CODE

```
import pygame import
random import math
pygame.init()

class DrawInformation: BLACK = 0,
0, 0
WHITE = 255, 255, 255
GREEN = 0, 255, 0
RED = 255, 0, 0 BACKGROUND_COLOR = WHITE

GRADIENTS = [ (128,
128, 128),
(160, 160, 160),
(192, 192, 192)
]

FONT = pygame.font.SysFont('comicsans', 30) LARGE_FONT =
pygame.font.SysFont('comicsans', 40)

SIDE_PAD = 100
TOP_PAD = 150

def __init__(self, width, height, lst):
    self.width = width
    self.height = height

    self.window = pygame.display.set_mode((width, height))

    pygame.display.set_caption("Sorting Algorithm Visualization") self.set_list(lst)
```



```
def set_list(self, lst): self.lst = lst
    self.min_val = min(lst)
    self.max_val = max(lst)

    self.block_width = round((self.width - self.SIDE_PAD) / len(lst))
    self.block_height = math.floor((self.height - self.TOP_PAD) / (self.max_val - self.min_val))
    self.start_x = self.SIDE_PAD // 2

def draw(draw_info, algo_name, ascending):
    draw_info.window.fill(draw_info.BACKGROUND_COLOR)

    title = draw_info.LARGE_FONT.render(f'{algo_name} - '
    {'Ascending' if ascending else 'Descending'}", 1, draw_info.GREEN) draw_info.window.blit(title,
    (draw_info.width/2 -
    title.get_width()/2 , 5))

    controls = draw_info.FONT.render("R - Reset | SPACE - Start Sorting | A - Ascending | D - "
    "Descending", 1, draw_info.BLACK)
    draw_info.window.blit(controls, (draw_info.width/2 - controls.get_width()/2 ,
    45))

    sorting = draw_info.FONT.render("I - Insertion | B - Bubble | S - Selection | Q - Quick | M - "
    "Merge", 1, draw_info.BLACK)
    draw_info.window.blit(sorting, (draw_info.width/2 - sorting.get_width()/2 , 75))

    draw_list(draw_info)

pygame.display.update()
```

```
def draw_list(draw_info, color_positions={}, clear_bg=False): lst = draw_info.lst
```



```
if clear_bg:
    clear_rect=(draw_info.SIDE_PAD//2, draw_info.TOP_PAD, draw_info.width -
                draw_info.SIDE_PAD,
    draw_info.height - draw_info.TOP_PAD)
    pygame.draw.rect(draw_info.window,
draw_info.BACKGROUND_COLOR,clear_rect)

for i, val in enumerate(lst):
    x = draw_info.start_x + i * draw_info.block_width y =
    draw_info.height - (val - draw_info.min_val) *
    draw_info.block_height
    color = draw_info.GRADIENTS[i % 3] if i in
    color_positions:
        color = color_positions[i]

    pygame.draw.rect(draw_info.window, color, (x, y, draw_info.block_width,
draw_info.height))

if clear_bg: pygame.display.update()

def generate_starting_list(n, min_val, max_val): lst = []

for _ in range(n):
    val = random.randint(min_val, max_val)

    lst.append(val) return
lst

def bubble_sort(draw_info, ascending=True): lst =
    draw_info.lst

for i in range(len(lst) - 1):
```



```
for j in range(len(lst) - 1 - i): num1 = lst[j]
    num2 = lst[j + 1]

    if (num1 > num2 and ascending) or (num1 < num2 and not ascending):
        lst[j], lst[j + 1] = lst[j + 1], lst[j]
        draw_list(draw_info, {j: draw_info.GREEN, j + 1: draw_info.RED}, True)
    yield True
return lst

def insertion_sort(draw_info, ascending=True): lst =
    draw_info.lst

    for i in range(1, len(lst)): current =
        lst[i]

        while True:
            ascending_sort = i > 0 and lst[i - 1] > current and ascending
            descending_sort = i > 0 and lst[i - 1] < current and not
            ascending

            if not ascending_sort and not descending_sort:
                break
            lst[i] = lst[i - 1]
            i = i - 1
            lst[i] = current
            draw_list(draw_info, {i - 1: draw_info.GREEN, i: draw_info.RED},
            True)
    yield True
return lst

def selection_sort(draw_info, ascending=True): lst =
    draw_info.lst
```



```
for i in range(len(lst)): min_idx = i

    for j in range(i + 1, len(lst)):
        if (lst[j] < lst[min_idx] and ascending) or (lst[j] > lst[min_idx] and not ascending):
            min_idx = j

    lst[i], lst[min_idx] = lst[min_idx], lst[i]
    draw_list(draw_info, {i: draw_info.GREEN, min_idx: draw_info.RED}, True)

    yield True

return lst

def quick_sort(draw_info, ascending=True): lst =
    draw_info.lst

def quick_sort_helper(start, end):

    if start >= end: return

    pivot = lst[end]
    partition_idx =
    start

    for i in range(start, end):
        if (lst[i] < pivot and ascending) or (lst[i] > pivot and not ascending):
            lst[i], lst[partition_idx] = lst[partition_idx], lst[i]
            partition_idx += 1
            draw_list(draw_info, {i: draw_info.GREEN, partition_idx: draw_info.RED}, True)

    yield True

    lst[partition_idx], lst[end] = lst[end], lst[partition_idx]
    draw_list(draw_info, {partition_idx: draw_info.GREEN, end: draw_info.RED}, True)

    yield True
```



```
yield from quick_sort_helper(start, partition_idx - 1) yield from
quick_sort_helper(partition_idx + 1, end)

yield from quick_sort_helper(0, len(lst) - 1) def
merge_sort(draw_info, ascending=True):
    lst = draw_info.lst

def merge_sort_helper(start, end): if end - start
    + 1 <= 1:
        return

    mid = (start + end) // 2
    yield from merge_sort_helper(start, mid)

    yield from merge_sort_helper(mid + 1, end)

    left = lst[start:mid + 1] right =
    lst[mid + 1:end + 1] l_idx, r_idx =
    0, 0

    for i in range(start, end + 1):
        if l_idx < len(left) and (r_idx >= len(right) or (left[l_idx] < right[r_idx] if ascending else
left[l_idx] > right[r_idx])):
            lst[i] = left[l_idx] l_idx
            += 1
        else:
            lst[i] = right[r_idx] r_idx +=
            1

draw_list(draw_info, {i: draw_info.GREEN}, True) yield True
yield from merge_sort_helper(0, len(lst) - 1) def main():
    run = True
```



```
clock = pygame.time.Clock()

n = 50
min_val = 0
max_val = 100

lst = generate_starting_list(n, min_val, max_val)
draw_info =
DrawInformation(800, 600, lst)
sorting = False

ascending = True

sorting_algorithm = bubble_sort
sorting_algo_name =
"Bubble Sort"
sorting_algorithm_generator = None

while run:
    clock.tick(10)

    if sorting:
        try:
            next(sorting_algorithm_generator)
        except StopIteration:
            sorting = False
    else:
        draw(draw_info, sorting_algo_name, ascending)

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

        if event.type != pygame.KEYDOWN:
            continue

        if event.key == pygame.K_r:
```



```
lst=generate_starting_list(n, min_val, max_val)
draw_info.set_list(lst)
sorting = False

elif event.key == pygame.K_SPACE and sorting == False: sorting = True
    sorting_algorithm_generator =

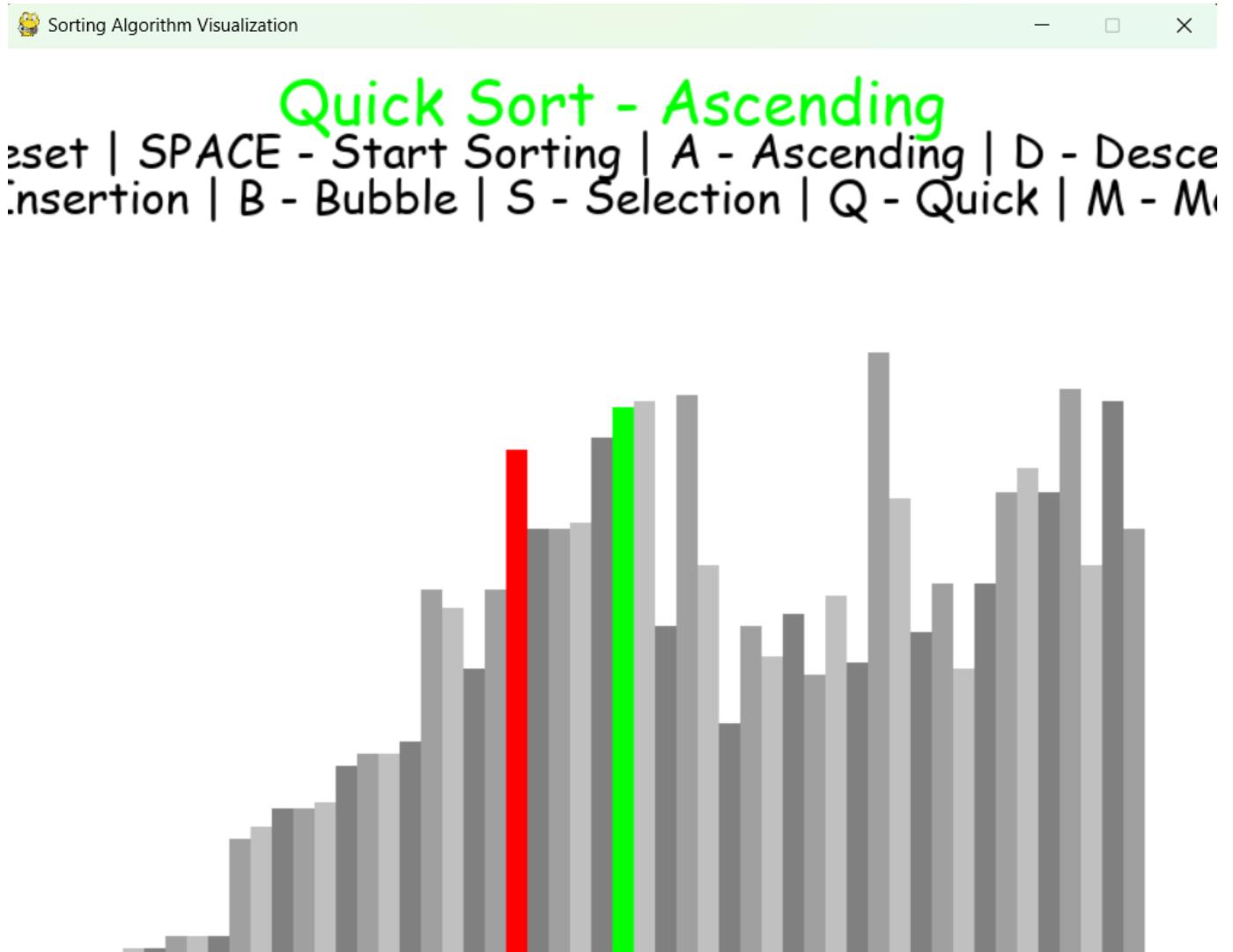
sorting_algorithm(draw_info, ascending)

    elif event.key == pygame.K_a and not sorting: ascending = True
    elif event.key == pygame.K_d and not sorting: ascending = False
    elif event.key == pygame.K_i and not sorting: sorting_algorithm =
        insertion_sort sorting_algo_name = "Insertion Sort"
    elif event.key == pygame.K_b and not sorting: sorting_algorithm
        = bubble_sort sorting_algo_name = "Bubble Sort"
    elif event.key == pygame.K_s and not sorting: sorting_algorithm =
        selection_sort sorting_algo_name = "Selection Sort"
    elif event.key == pygame.K_q and not sorting: sorting_algorithm
        = quick_sort sorting_algo_name = "Quick Sort"
    elif event.key == pygame.K_m and not sorting: sorting_algorithm
        = merge_sort sorting_algo_name = "Merge Sort"

pygame.quit()

if __name__ == "__main__":
    main()
```

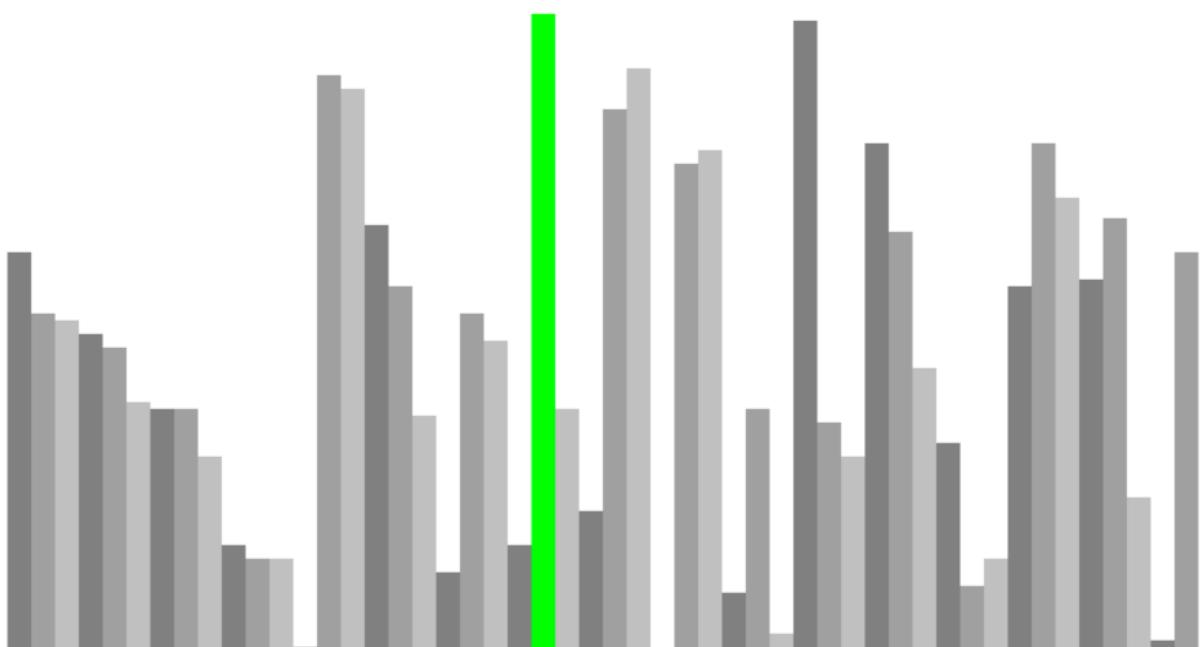
OUTPUT OF PROJECT



Sorting Algorithm Visualization

Merge Sort - Descending

Reset | SPACE - Start Sorting | A - Ascending | D - Descending | Insertion | B - Bubble | S - Selection | Q - Quick | M - Merge





SCOPE OF PROJECT

The above code is a basic implementation of a graphical user interface (GUI) for visualizing sorting algorithms using the Python programming language and the Pygame library. Its scope is to provide users with an intuitive and interactive way to understand how different sorting algorithms work by allowing them to visually follow the sorting process in real-time.

The program allows the user to choose a sorting algorithm, such as **Bubble Sort**, **Insertion Sort**, **Selection Sort**, **QuickSort**, or **MergeSort**, and observe how it organizes a randomly generated list of numbers. The user interface offers easy-to-use controls, such as buttons to reset the list, toggle sorting order, and start or stop the sorting process, enhancing the learning experience.

The scope of the code is focused on visualizing fundamental sorting algorithms, with features like real-time comparisons and swaps, but does not delve into advanced topics such as algorithm analysis, complexity comparison, or detailed performance metrics.

The program allows the user to choose a sorting algorithm, such as Bubble Sort, Insertion Sort, Selection Sort, QuickSort, or MergeSort, and observe how it organizes a randomly generated list of numbers. The user interface offers easy-to-use controls, such as buttons to reset the list, toggle sorting order, and start or stop the sorting process, enhancing the learning experience. Additionally, users can adjust the speed of the visualization, providing flexibility to either speed up the process for a quick overview or slow it down to analyze each step more closely.

While the program effectively demonstrates the core mechanics of sorting algorithms, it is primarily focused on visualization and interactivity, rather than in-depth performance evaluation. As such, it does not explore advanced topics like time complexity analysis, space utilization, or algorithmic trade-offs. However, by experimenting with the different algorithms, users can still get a sense of which sorting methods are faster or more efficient for small datasets. This practical, hands-on approach complements traditional learning methods, making it a useful tool for students, educators, and hobbyists.

Overall, the program serves as an educational platform for teaching essential concepts in sorting algorithms, data structures, and Python GUI development. Its simplicity and interactivity make it ideal for classroom demonstrations, coding workshops, or self-learning projects. Through direct engagement with sorting algorithms, users can gain a deeper understanding of algorithm behavior and improve their problem-solving skills, forming a solid foundation for further study in computer science and software development.



FUTURE DEVELOPMENT OF PROJECT

Certainly! There are several ways we could enhance the Sorting Algorithm Visualizer and make it more advanced and feature-rich:

1. **Add more sorting algorithms:** Currently, the program supports basic algorithms like **Bubble Sort, Insertion Sort, QuickSort, and MergeSort**. Future versions could include advanced algorithms like **Heap Sort, Shell Sort, and Radix Sort**.
2. **Algorithm performance comparison:** Adding a feature that displays the time complexity and space complexity of each sorting algorithm in real-time would help users understand how the algorithms compare in terms of efficiency.
3. **User-defined lists:** Instead of only visualizing randomly generated lists, we could allow users to input their own lists of numbers or even strings, making the tool more flexible and useful for specific cases.
4. **Different data structures:** We could expand it to work with other data structures like **linked lists, trees, or graphs**, showcasing how sorting works in different scenarios.
5. **Mobile or web-based version:** We could develop a web-based or mobile version of the visualizer, allowing users to access the tool from any device without needing to install the program.

By implementing these features, the Sorting Algorithm Visualizer could become more powerful, informative, and versatile for a broader range of educational and practical applications.

Lastly, we could implement dark mode and custom themes, enhancing the user experience and making the visualizer more appealing and comfortable for prolonged use. The ability to save and export sorting results or even share them with others could also be useful, especially for educators wanting to create assignments or demonstrations. With these enhancements, the Sorting Algorithm Visualizer would become a comprehensive educational platform catering to a variety of learning needs and practical applications in computer science.



CONCLUSION

The provided code is a sorting algorithm visualizer program built using the Python Pygame library. It allows users to choose between different sorting algorithms such as **Bubble Sort**, **Insertion Sort**, **QuickSort**, **MergeSort**, and **Selection Sort**. The program generates a visual representation of the sorting process in real-time, giving users a clear understanding of how the algorithms work by highlighting comparisons and swaps. The output is displayed in the GUI window, and users can control the sorting speed and order (ascending or descending) for further interaction.

This code serves as an excellent starting point for learning about basic sorting techniques and GUI development using Python.

Overall, the program provides a simple and interactive interface for visualizing sorting algorithms, making it a valuable educational tool. It can be expanded with more algorithms, performance analysis, and customization features based on user needs.

