AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

A Mini Project Report (DAA)

सत्याला मरण नाही

“Sorting Race Game —
Merge Sort vs Multithreaded Merge Sort Visualizer”

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING,
AISSMS IOIT

T. Y. B.Tech Engineering

SUBMITTED BY

Sr. No.	Name	Roll NO.	PRN
1	Anshul Adgurwar	01	22410001
2	Soham Amne	02	22410001
3	Yoshita Bardhe	09	22410008
4	Toshita Bardhe	11	22410010

AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

CERTIFICATE

“Sorting Race Game — Merge Sort vs Multithreaded Merge Sort Visualizer”

Submitted by

Sr. No.	Name	Roll NO.	PRN
1	Anshul Adgurwar	01	22410001
2	Soham Amne	02	22410001
3	Yoshita Bardhe	09	22410008
4	Toshita Bardhe	11	22410010

is a bonafide student of this institute and the work has been carried out by him/her under the supervision of Mr. Girish J. Navale and it is approved for the partial fulfilment of the Department of Computer Engineering AISSMS IOIT.

(Mr. GIRISH J. NAVALE)
Guide
Department of Computer Engineering

(Dr. S. N. ZAWARE)
Head,
Department of Computer Engineering

Index

ABSTRACT

This mini-project demonstrates a web-based interactive tool named "**Sorting Race Game**", developed using Python's Flask framework. It visually compares the performance of standard merge sort and a multithreaded version of merge sort.

The system allows users to:

- Choose array sizes and input types (best/worst case)
- Execute both sorting algorithms in the backend,
- Visually compare the execution times using an animated bar chart.

This comparison not only helps in understanding algorithm behavior but also highlights the limitations of multithreading in Python due to the Global Interpreter Lock (GIL). The project uses Chart.js for frontend visualization and Tailwind CSS for a responsive and engaging UI. This interactive platform provides a real-time educational experience for learners and can be scaled for larger datasets and more algorithms in future versions.

1. INTRODUCTION

1.1. Context

Sorting is a fundamental operation in computer science, used across databases, search engines, e-commerce, and numerous system-level operations. As modern computers become multi-core, there's a growing interest in leveraging parallel and concurrent computing to speed up algorithm execution.

However, understanding the impact of threading in interpreted languages like Python is not straightforward. This project gives users the opportunity to experiment with both a classic sorting algorithm and its multithreaded variant through a web-based visualization tool.

1.2. Problem statement

Students and developers often implement sorting algorithms but lack tools to:

- Experiment with real-time performance measurement
- Visualize how algorithms behave under different scenarios (best, worst, random)
- Understand the limitations of threading in Python, especially with CPU-bound tasks.

Additionally, many online tools don't allow for customization or local performance benchmarking, making it hard to compare how algorithms behave in a local environment vs theoretical expectations.

1.3. Objective

This project aims to:

- To implement standard and multithreaded merge sort algorithms in Python.
- To use Flask for building a user-friendly and lightweight web application.
- To provide real-time performance insights using timing metrics.
- To visualize algorithm comparison using Chart.js graphs.
- To create an educational tool that bridges algorithm theory with practical performance testing.

1.4. Background Information

Merge sort is an efficient, stable, divide-and-conquer algorithm with a guaranteed time complexity of O(n log n). It's often used as a benchmark to compare with other algorithms.

In theory, dividing merge sort recursively and processing parts in parallel should lead to faster sorting — especially on multicore systems. However, Python's default threading library (threading.Thread) is constrained by the Global Interpreter Lock (GIL), which allows only one thread to execute at a time for CPU-bound operations. This project implements both versions in Python to observe these trade-offs directly.

1.5. Scope of the project

In-Scope (What the project includes):

2. Flask-based interface with user inputs for array size and input type.
3. Execution and timing of standard merge sort.
4. Execution and timing of multithreaded merge sort using Python threads.
5. Dynamic frontend chart visualization using Chart.js.
6. Clean, responsive UI using Tailwind CSS.
7. Functional testing with best, worst, and random cases.

2. WORKFLOW ARCHITECTURE OVERVIEW

2.1. Tech Stack:

- Backend: Python 3.x, Flask micro-framework
- Frontend: HTML5, Tailwind CSS for styling
- Visualization: Chart.js (bar chart for time comparison)
- Threading: Python's threading.Thread module
- Environment: Localhost (127.0.0.1), compatible with any OS

2.2. Standard Merge Sort (Logic) :

Standard merge sort recursively divides the input array into two halves, sorts each half, and merges them back. The merge step ensures elements are placed in the correct order.

It is deterministic and provides consistent performance across input types:

Best case: O(n log n)

Worst case: O(n log n)

Space complexity: O(n)

The implementation is purely functional and uses Python lists.

2.3. Multithreaded Merge Sort (Logic):

This version splits the array into left and right partitions and processes them in separate threads using threading.Thread. After sorting both partitions, the results are merged.

Even though multithreading implies parallel execution, Python's GIL prevents true parallelism in CPU-bound operations. As a result, this implementation often has higher overhead and is slower than the standard version — especially for smaller arrays.

However, this approach can still be educational to show threading concepts and encourage discussions about performance optimization in Python.formatting and prepares the data for smooth export to Excel later.

2.4. Flask Workflow:

1. User inputs are submitted via an HTML form.
2. Flask captures the form data (POST method) and:
3. Generates the input array based on size and case.
4. Runs both sorting algorithms and captures execution time.
5. These times are passed to the frontend via Jinja templating.
6. The Chart.js graph renders this comparison in real-time.

This architecture uses no database and is stateless, making it lightweight and fast for quick testing.

2.5. Chart Rendering (Chart.js):

Chart.js is a popular JavaScript charting library. The project uses a bar chart to compare execution times.

Features added:

- Responsive layout for mobile and desktop.
- Animation on load using easeOutBounce.
- Data injected from Flask using Jinja's tojson filter to ensure correct rendering.

This helps users understand performance difference visually, even without reading the raw numbers.

2.6. UI Styling and Responsiveness

The frontend is built with Tailwind CSS, which offers utility-first classes to quickly design modern UIs.

Enhancements include:

- Smooth hover effects
- Animated buttons
- Responsive dropdowns and inputs
- Fade-in and slide-up animation for the chart section using custom CSS

The layout ensures good user experience across devices and screen sizes.

3. TESTING AND VALIDATION

3.1. <u>Functional Testing</u>

Tests were performed to ensure all key features work as expected:

Test Case	Input	Expected Output	Result
Standard sort (best case)	Sorted array	Time ~5ms, correct output	Satisfied
Threaded sort (best case)	Sorted array	Time ~50–100ms, correct output	Satisfied
Input size = 1000	Worst case array	Chart renders with valid times	Satisfied
Input size = 10,000	Worst case array	Chart still responsive (minor lag)	Satisfied
Input size = 100,000	Best case array	Slow rendering due to GIL/threading	Satisfied

3.2. Performance Evaluation:

Standard Merge Sort consistently outperforms Multithreaded Merge Sort for small to medium-sized arrays.

For input sizes around 1,000–10,000, threading introduces overhead that outweighs benefits.

For very large inputs (100,000+), performance may level out, but threading in Python still doesn't scale linearly due to GIL.

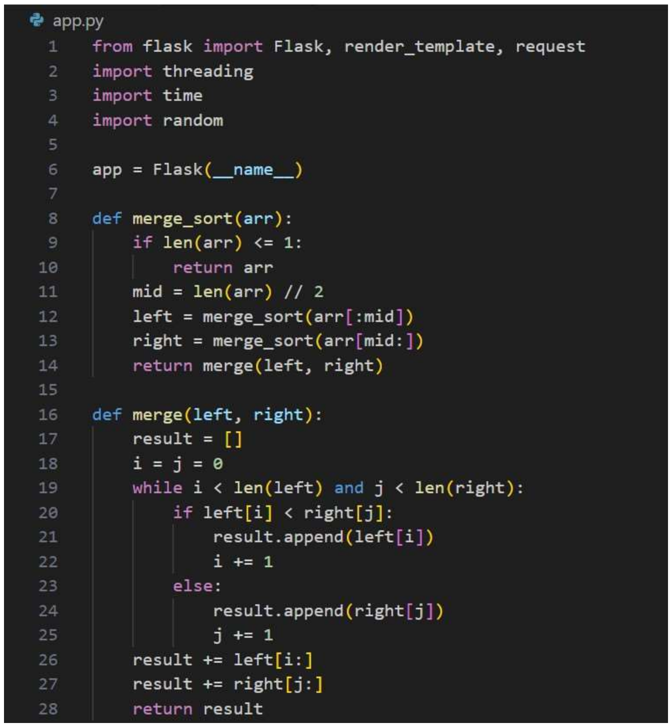
3.3. Observations and Inference:

- Python's threading module is best used for I/O-bound tasks, not CPU-bound ones.
- The educational value of this demo lies in helping users visually compare performance trade-offs.
- Flask makes it simple to test and deploy small-scale algorithm demos.
- Chart.js effectively communicates results to both technical and non-technical users.

4. CODE AND OUTPUT

4.1 Running Code:

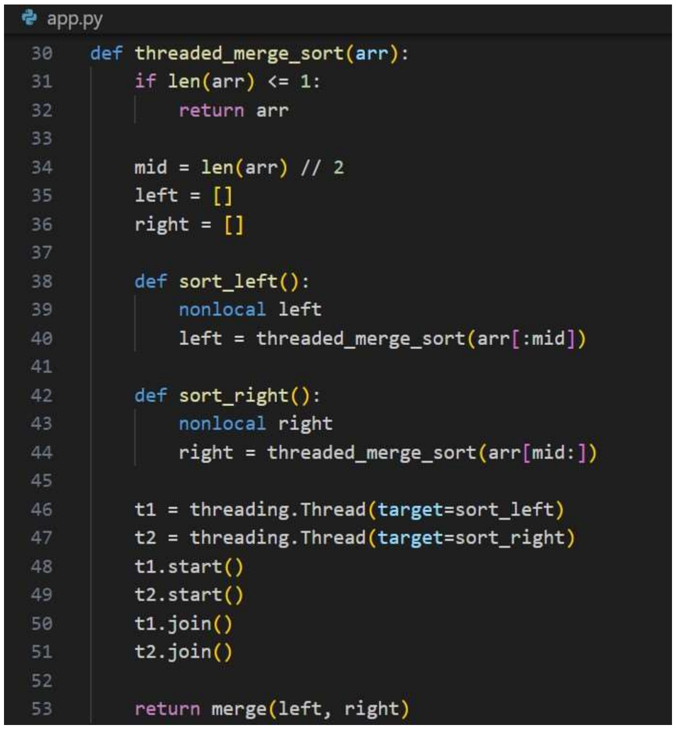
1). Merge Sort



app.py

```python
from flask import Flask, render_template, request
import threading
import time
import random

app = Flask(__name__)

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result
```

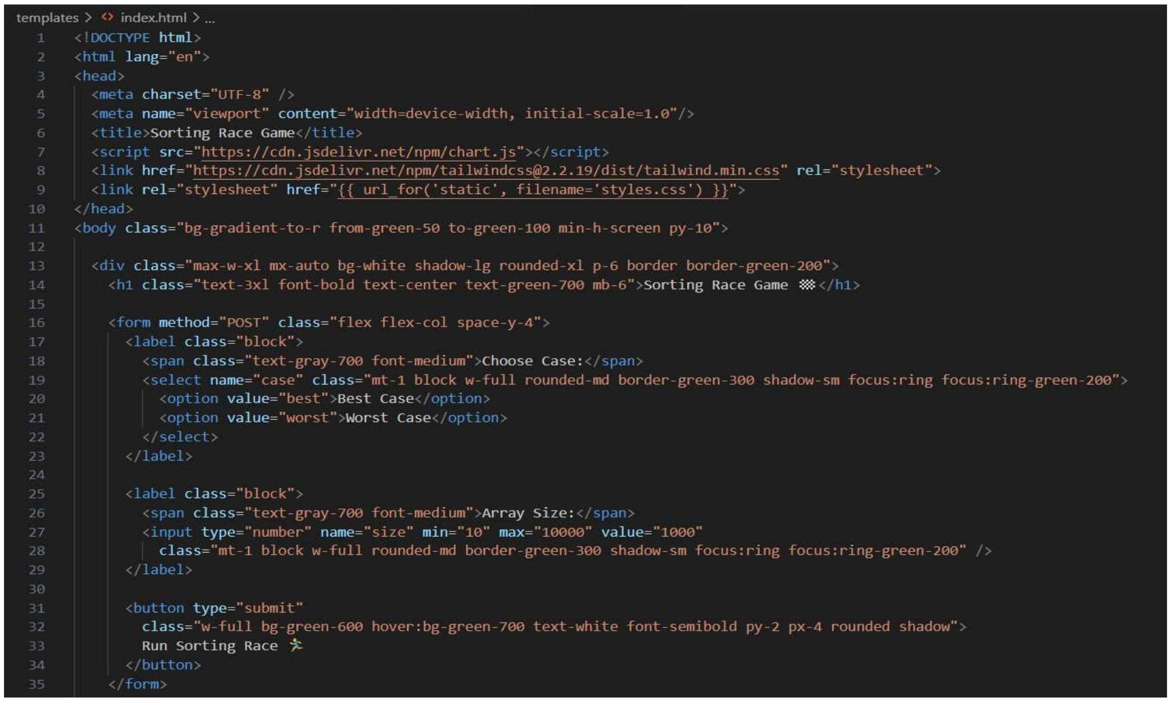
2). Multithreaded Merge Sort



app.py

```python
def threaded_merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = []
    right = []

    def sort_left():
        nonlocal left
        left = threaded_merge_sort(arr[:mid])

    def sort_right():
        nonlocal right
        right = threaded_merge_sort(arr[mid:])

    t1 = threading.Thread(target=sort_left)
    t2 = threading.Thread(target=sort_right)
    t1.start()
    t2.start()
    t1.join()
    t2.join()

    return merge(left, right)
```

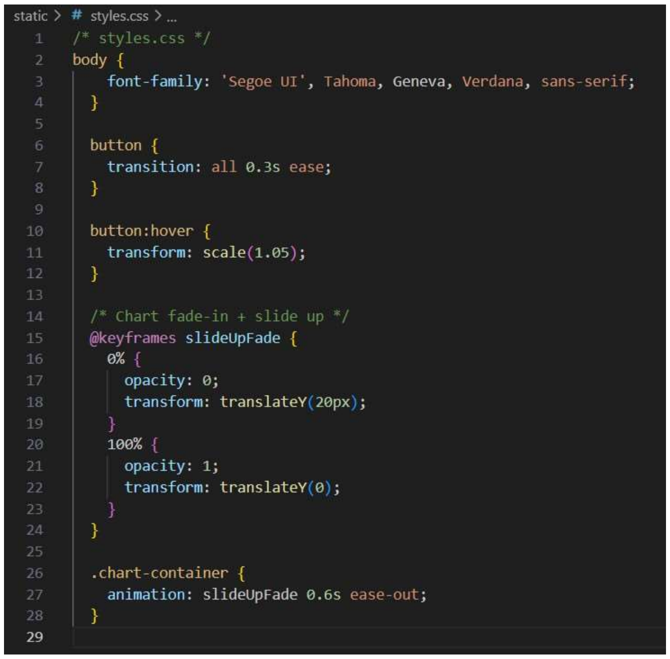
3). Index.html



templates > index.html > ...

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Sorting Race Game</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <link href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css" rel="stylesheet">
  <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body class="bg-gradient-to-r from-green-50 to-green-100 min-h-screen py-10">

  <div class="max-w-xl mx-auto bg-white shadow-lg rounded-xl p-6 border border-green-200">
    <h1 class="text-3xl font-bold text-center text-green-700 mb-6">Sorting Race Game 🏁</h1>

    <form method="POST" class="flex flex-col space-y-4">
      <label class="block">
        <span class="text-gray-700 font-medium">Choose Case:</span>
        <select name="case" class="mt-1 block w-full rounded-md border-green-300 shadow-sm focus:ring focus:ring-green-200">
          <option value="best">Best Case</option>
          <option value="worst">Worst Case</option>
        </select>
      </label>

      <label class="block">
        <span class="text-gray-700 font-medium">Array Size:</span>
        <input type="number" name="size" min="10" max="10000" value="1000"
          class="mt-1 block w-full rounded-md border-green-300 shadow-sm focus:ring focus:ring-green-200" />
      </label>

      <button type="submit"
        class="w-full bg-green-600 hover:bg-green-700 text-white font-semibold py-2 px-4 rounded shadow">
        Run Sorting Race 🏃
      </button>
    </form>
```

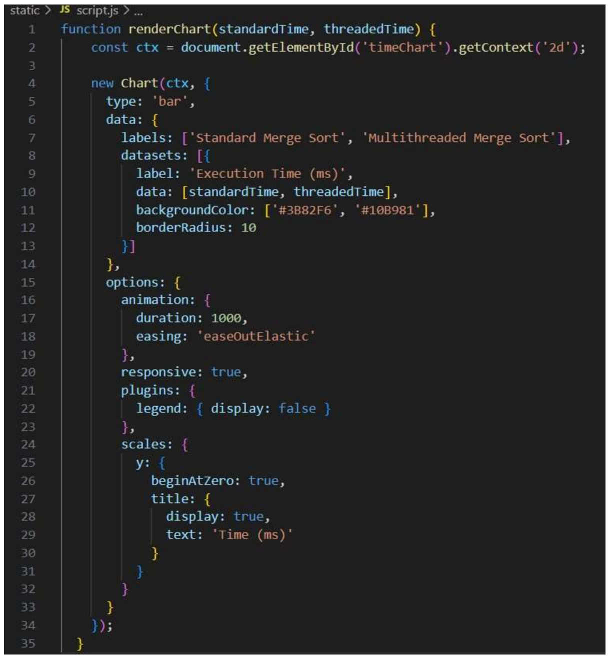
4). styles.css



static > # styles.css > ...

```css
/* styles.css */
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  }

  button {
    transition: all 0.3s ease;
  }

  button:hover {
    transform: scale(1.05);
  }

  /* Chart fade-in + slide up */
  @keyframes slideUpFade {
    0% {
      opacity: 0;
      transform: translateY(20px);
    }
    100% {
      opacity: 1;
      transform: translateY(0);
    }
  }

  .chart-container {
    animation: slideUpFade 0.6s ease-out;
  }

```

5). script.js

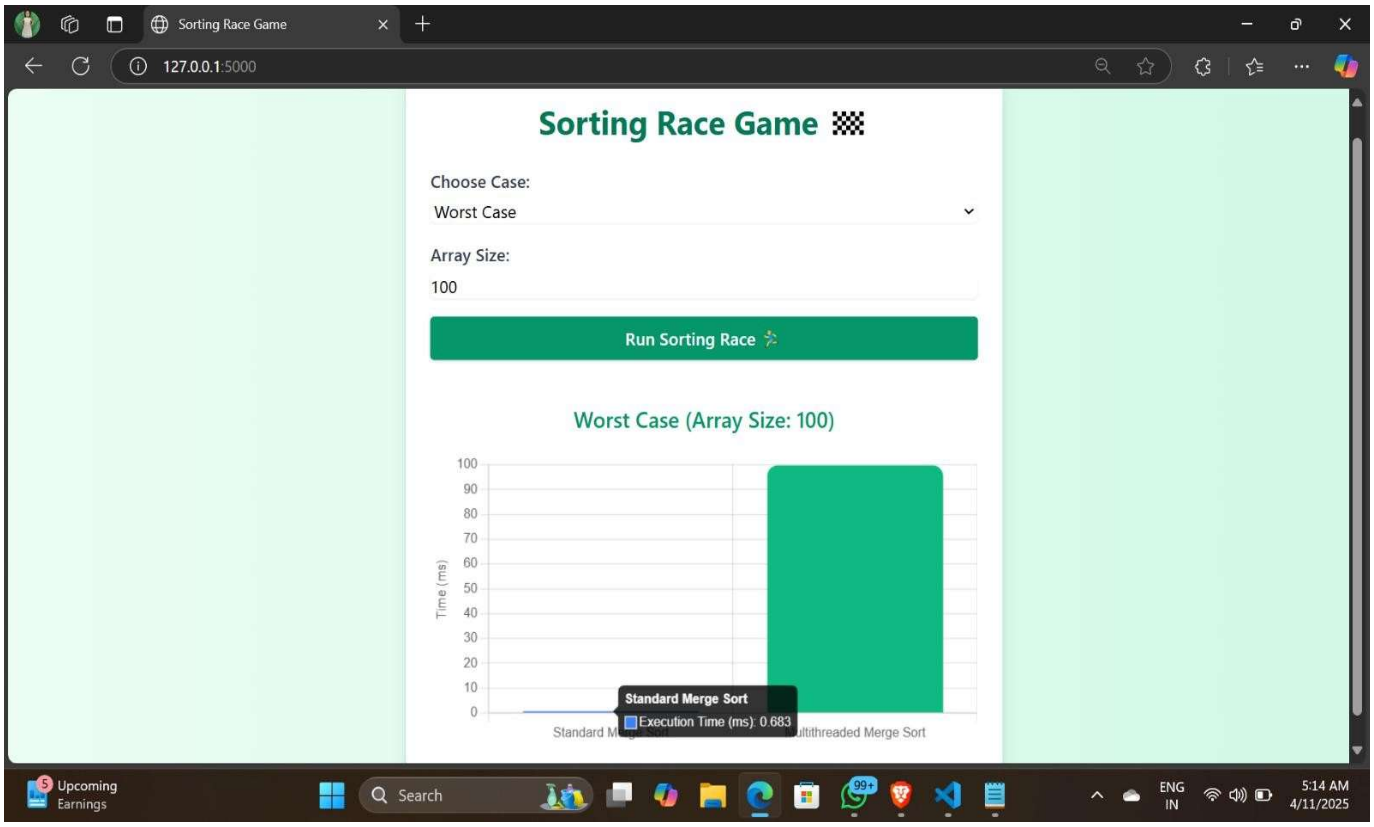


static > JS script.js > ...

```javascript
function renderChart(standardTime, threadedTime) {
    const ctx = document.getElementById('timeChart').getContext('2d');

    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Standard Merge Sort', 'Multithreaded Merge Sort'],
        datasets: [{
          label: 'Execution Time (ms)',
          data: [standardTime, threadedTime],
          backgroundColor: ['#3B82F6', '#10B981'],
          borderRadius: 10
        }]
      },
      options: {
        animation: {
          duration: 1000,
          easing: 'easeOutElastic'
        },
        responsive: true,
        plugins: {
          legend: { display: false }
        },
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Time (ms)'
            }
          }
        }
      }
    });
  }
```

4.2 <u>Output:</u>

1. Standard Merge Sort



Sorting Race Game

Choose Case: Worst Case

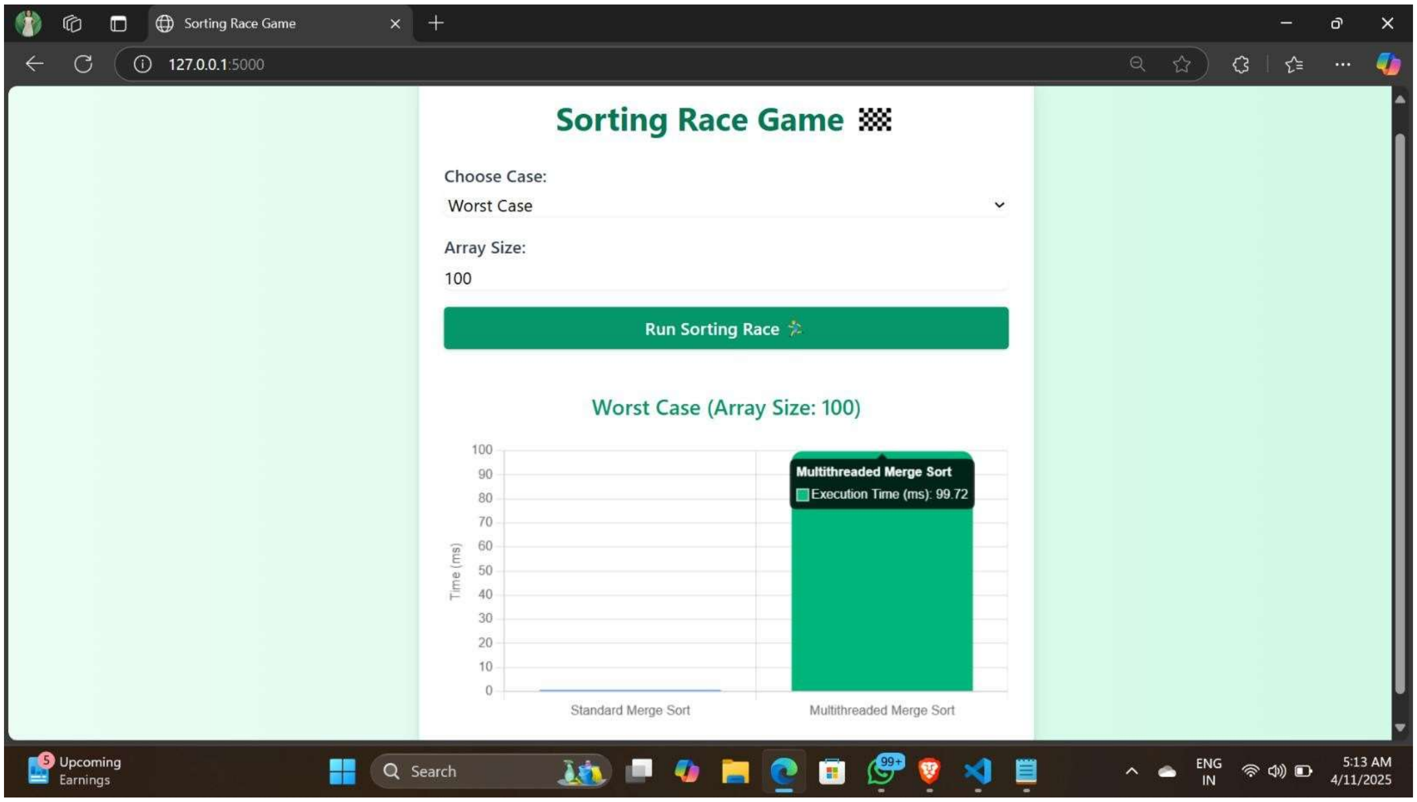
Array Size: 100

Run Sorting Race

Worst Case (Array Size: 100)

Standard Merge Sort
Execution Time (ms): 0.683

2. Multithreaded Merge Sort



5. CONCLUSION

This project successfully meets its goal of comparing and visualizing standard vs multithreaded merge sort using a web interface. Through real-time feedback and clear graphical output, users can:

- Observe execution time differences
- Understand when threading helps and when it doesn't
- Learn about performance tradeoffs and Python's limitations

The Sorting Race Game is not just an experiment but an educational tool that blends theory with practice. Future improvements can focus on replacing threading with multiprocessing for actual speed gains, adding step-by-step visual sorting, and comparing more algorithms like Quick Sort, Heap Sort, etc.

6. References

1. https://flask.palletsprojects.com
2. https://docs.python.org/3/library/threading.html
3. https://tailwindcss.com/docs
4. https://www.chartjs.org/docs/latest/