SORTING ALGORITHM VISUALIZER

21CSC204J SUBJECT - DESIGN & ANALYSIS OF ALGORITHMS

A MINI-PROJECT REPORT

Submitted By

Agasthya S	**RA2211042020019**
Sarathi V	**RA2211042020026**
Shakthivel K	**RA2211042020060**

in partial fulfilment for the award of the degree

of

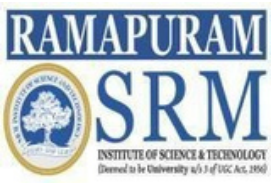
BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

RAMAPURAM, CHENNAI-600089

MAY2024

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Deemed to be University U/S 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this mini project report titled **"SORTING ALGORITHM VISUALIZER"** is the bonafide work of **"Agasthya S - RA2211042020019, Sarathi V- RA2211042020026, Shakthivel K - RA2211042020060"**

SIGNATURE

Ms. K. Aishwarya

Assistant Professor,

Computer Science and Engineering,
SRM Institute of Science and Technology,
Ramapuram, Chennai.

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

RAMAPURAM, CHENNAI -600089

DECLARATION

We hereby declare that the entire work contained in this mini project report titled **"SORTING ALGORITHM VISUALIZER"** has been carried out by **AgasthyaS RA2211042020019), Sarathi V (RA2211042020026), Shakthivel K (RA2211042020060)** at SRM Institute of Science and Technology, Ramapuram Campus, Chennai - 600089.

Place: Chennai

Date:

ABSTRACT

The Sorting Algorithm Visualizer project aims to enhance algorithm comprehension through animated visualizations. We created a web-based application that demonstrates four sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, and Merge Sort. Users can observe data being sorted in real time, improving their understanding of algorithmic processes. This project bridges the gap between theoretical knowledge and practical implementation, making complex concepts more accessible.

Keywords: Sorting Algorithms, React Visualizer, Selection Sort, Merge Sort, Bubble Sort, Insertion Sort, Heap Sort.

Additionally, other projects contribute to algorithm education through visualizations. The Algorithm Visualizer provides an interactive platform for understanding algorithms, while another paper introduces a web-based tool for visualizing searching and sorting algorithms. The ViSA (Visualization System for Algorithms) project offers step-by-step explanations and comparisons of sorting algorithms, emphasizing practical implications and educational benefits.
In summary, these projects enhance algorithm learning by making complex concepts more accessible through visual representations.

TABLE OF CONTENTS

Chapter No.	Title	Page No.
	ABSTRACT	4
1	**INTRODUCTION**	6
	1.1 Overview	6
	1.2 Problem Statement	6
	1.3 Objectives of the Project	7
	1.4 Scope	7
2	**SOFTWARE AND HARDWARE SPECIFICATION**	8
	2.1 Hardware Requirements	8
	2.2 Software Requirements	8
3	**PROBLEM DESCRIPTION**	9
	3.1 Introduction	9
	3.2 Algorithm Used	9
	3.3 Advantage of Algorithm Used	10
	3.4 Explanation of the Project	11
	3.5 Output Results	13
	3.6 Conclusion	14
4	**APPENDIX**	15
	Source Code	15

CHAPTER 1

INTRODUCTION

1.1 Overview

The sorting algorithms visualizer project is a well-structured application with separate modules for UI, rendering, and sorting algorithms. The user interface, built using Tkinter, provides intuitive controls such as dropdowns for selecting sample size, algorithms, and speed levels, along with buttons for generating samples and running sorting algorithms. The graphical representation on the canvas vividly illustrates the sorting process, highlighting elements as they are compared or swapped during sorting. Implemented algorithms like Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort offer users a comprehensive view of various sorting techniques in action. Enhancements such as adding comments for clarity and consistency in naming conventions would further refine this educational tool, making it a valuable resource for learning sorting algorithms interactively.

1.2 Problem Statement

Despite the availability of numerous online resources and theoretical explanations, many learners struggle to grasp the intricate workings of sorting algorithms due to the lack of hands-on, interactive learning tools. Existing educational materials often fail to provide a visually engaging platform that allows users to dynamically observe and understand how different sorting algorithms operate in real-time. This gap in interactive learning tools hinders the effective comprehension and application of sorting algorithms, leading to suboptimal learning outcomes and a limited ability to translate theoretical knowledge into practical programming skills. Therefore, there is a pressing need for a sorting algorithms visualizer that offers a user-friendly interface, interactive visualization of sorting processes, and a diverse range of sorting algorithms to facilitate comprehensive learning and mastery of sorting techniques.

1.3 Aim of the Project

Develop a user-friendly sorting algorithms visualizer using Tkinter, enabling users to interactively explore Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort. The visualizer will dynamically depict sorting processes on a graphical canvas, aiding in understanding and improving programming skills.

1.4 Scopeof the Project

The project scope involves developing a user-friendly sorting algorithms visualizer using Tkinter. It includes creating an interface with dropdowns for algorithm selection, sample size, and speed adjustment. The visualizer will render sorting processes dynamically on a graphical canvas, with error handling for a smooth user experience. Documentation and testing will ensure functionality, and future enhancements may include adding more algorithms and advanced visualization techniques.

CHAPTER 2

SOFTWARE AND HARDWARE SPECIFICATIONS

2.1 Hardware Requirements

- Processor: Dual-core processor or higher
- RAM: 2 GB or more
- Display: Minimum resolution of 1024x768 pixels recommended for optimal visualization experience
- Storage: Sufficient disk space for Python installation and project files

2.2 Software Requirements

- Python 3.x (with Tkinter library)
- Operating System: Windows 7 or later, macOS, or Linux distribution
- Python packages: Tkinter (included in standard Python installations), ctypes (for DPI awareness on Windows if needed)

CHAPTER 3

PROJECT DESCRIPTION

3.1 Introduction

The sorting algorithms visualizer is a Python-based educational tool designed to provide an interactive and intuitive platform for users to explore and comprehend various sorting algorithms. Developed using the Tkinter library, this visualizer offers a user-friendly interface with dropdown menus for selecting algorithms like Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort, along with options to adjust sample sizes and speed levels. The visualizer dynamically renders sorting processes on a graphical canvas, enabling users to observe algorithmic behaviors in real-time and gain a deeper understanding of sorting techniques. With error handling mechanisms ensuring a smooth user experience, comprehensive documentation, and rigorous testing, this project aims to enhance learning and practical application of sorting algorithms for programmers and learners alike.

3.2 Algorithm Used

The sorting algorithms visualizer employs a range of classic sorting algorithms to showcase their functionalities and behaviors. These algorithms include:

1. Bubble Sort: A simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

2. Insertion Sort: Another comparison-based algorithm that builds the final sorted array one element at a time, inserting each element into its correct position in the sorted portion of the array.

3. Selection Sort: This algorithm sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.

4. Merge Sort: A divide-and-conquer algorithm that recursively divides the array into smaller subarrays, sorts them individually, and then merges them back together to achieve the final sorted array.

5. Quick Sort: Another divide-and-conquer algorithm that partitions the array into smaller subarrays based on a chosen pivot element, sorts the subarrays recursively, and combines them to obtain the sorted array.

These algorithms demonstrate various approaches to sorting data and offer insights into their efficiency, performance, and suitability for different types of input data.

3.3 Advantages of Algorithm used

1. Bubble Sort:

Simple implementation with minimal code complexity. Works well for small datasets or nearly sorted arrays. Easy to understand and suitable for educational purposes.

2. Insertion Sort:

Efficient for small datasets and arrays that are nearly sorted. Adaptive, meaning it performs well on partially sorted data. In-place sorting, requiring only a constant amount of additional memory.

3. Selection Sort:

Simple implementation with straightforward logic. Requires minimal memory usage as it performs sorting in-place. Works well for small datasets or situations where memory usage needs to be minimized.

4. Merge Sort:

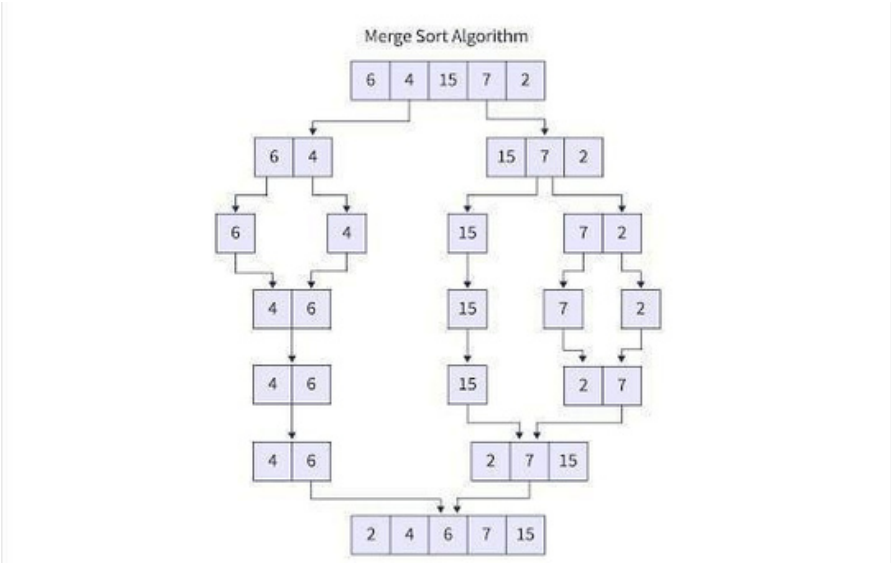
Efficient for sorting large datasets and handles large arrays well. Stable sorting algorithm, preserving the order of equal elements. Divides the sorting process into smaller subproblems, leading to improved performance.

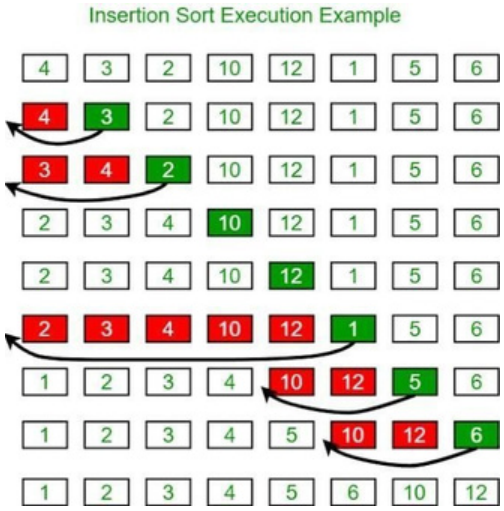
5. Quick Sort:

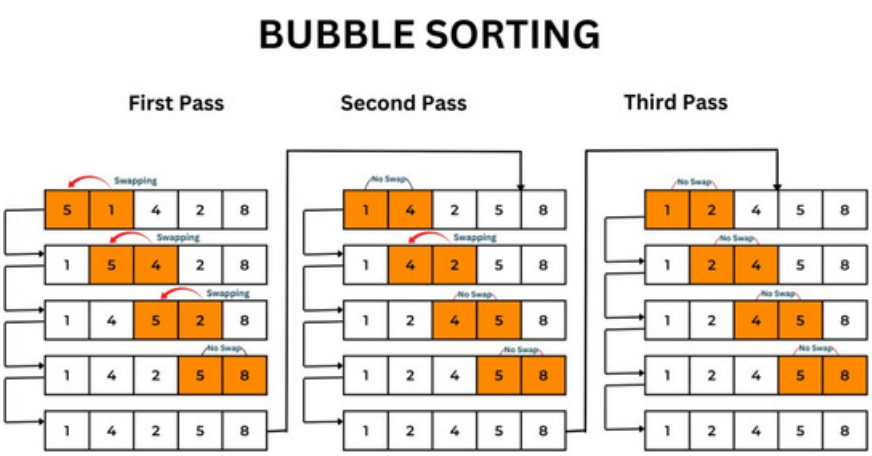
Efficient for large datasets and performs well on average and best-case scenarios. In-place sorting with relatively low memory requirements. Utilizes divide-and-conquer approach, leading to faster sorting times on average.

3.4 Explanation of The Project

The sorting algorithms visualizer project aims to provide an interactive and educational platform for users to explore and understand various sorting algorithms. Using Python with Tkinter for the user interface, the project offers a user-friendly environment where users can select from a range of sorting algorithms including Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort. The visualizer dynamically renders the sorting process on a graphical canvas, allowing users to observe and analyze the behavior of each algorithm in real-time. Users can adjust parameters such as sample size and speed level to further customize their learning experience. Error handling mechanisms are implemented to ensure a smooth user experience, and comprehensive documentation guides users through the functionalities of the visualizer. Through this project, users can enhance their knowledge and practical skills in sorting algorithms, bridging the gap between theoretical understanding and hands-on implementation.


Merge Sort Algorithm
6 4 15 7 2
6 4
15 7 2
6
4
15
7 2
4 6
15
7
2
4 6
15
2 7
4 6
2 7 15
2 4 6 7 15


Insertion Sort Execution Example
4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12


BUBBLE SORTING
First Pass
Second Pass
Third Pass
Swapping
No Swap

3.5 Output Results

The output result of the sorting algorithms visualizer is a dynamically updated graphical representation of the sorting process. Users can observe the step-by-step execution of sorting algorithms such as Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort on a graphical canvas. As the algorithms progress, elements are visually highlighted, swapped, or moved into their correct positions according to the sorting logic. Users can interact with the visualizer by adjusting parameters like sample size and speed level, allowing them to customize the sorting experience and observe algorithmic behaviors in real-time. The output result provides users with a clear and intuitive understanding of how each algorithm operates and how different parameters affect the sorting performance.

3.6 Conclusion

In conclusion, the sorting algorithms visualizer project offers a comprehensive and interactive platform for users to learn, explore, and understand various sorting algorithms. Through a user-friendly interface powered by Tkinter, users can select from a range of sorting algorithms and observe the sorting process visually on a graphical canvas. The project's implementation of Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort provides a diverse set of tools for users to analyze algorithmic behaviors and performance. Error handling mechanisms ensure a smooth user experience, while comprehensive documentation guides users through the functionalities of the visualizer. Overall, this project bridges the gap between theoretical knowledge and practical implementation, empowering users to enhance their understanding and proficiency in sorting algorithms in a hands-on and engaging manner.

CHAPTER 4

APPENDIX

Source Code

Main.py

```
from sorting import
start_visualizer

start_visualizer
()
```

Sorting/ init.py

```
from ._ui import
start_visualizer
```

Sorting/_ren
der.py

```
from tkinter import

Canvas, Tk from time

import sleep
def render_array(window: Tk, canva: Canvas, array: list, array_colors: list, speed:

float):

canva.delete("all")

canva_height = int(canva.cget('height'))

canva_width = int(canva.cget('width'))
unit_height = canva_height / max(array) unit_width = canva_width / len(array)

for i, value in enumerate(array):

x0 = unit_width * i

x1 = unit_width * (i + 1)

y0 = canva_height

    y1 = canva_height - (unit_height * value)

canva.create_rectangle(x0, y0, x1, y1,
fill=array_colors[i])
```

```
    window.update_idletasks()
```

Sorting/_ui.py

```
from tkinter import *
from tkinter import ttk, messagebox
from ._visual_algorithms import bubble_sort, insertion_sort, merge_sort, quick_sort,
selection_sort
from ._render import render_array
import random
import ctypes


try:
        ctypes.windll.shcore.SetProcessDpiAwareness(2)
except:
ctypes.windll.user32.SetProcessDPIAware()


# Colors
WHITE = "#FFFFFF"
BLACK = "#000000"
LIGHT_GRAY = "#c1c1c1"
DARK_GRAY = "#1d1d1d"
MATERIAL_BLACK =
"#121212"
BLACK = "#000000"
RED = '#d80000'
GREEN = '#12d800'
# Algorithms and Speed Levels
algorithms = {
'Bubble Sort': bubble_sort,
        'Insertion Sort': insertion_sort,
        'Selection Sort': selection_sort,
'Merge Sort': merge_sort,
```

```
    'Quick Sort': quick_sort
}

speed_levels = {
'Low': 0.1,
    'Medium': 0.05,
    'High': 0.015
}

# Visualizer
def start_visualizer():
window = Tk()
      window.title("Sorting Algorithms Visualizer")
   window.config(bg=MATERIAL_BLACK)
window.geometry("820x520")
window.resizable(width=0, height=0)

   global array
   array = []

# Top Menu
     fr_top_menu = Frame(master=window, width=900, height=300,
bg=MATERIAL_BLACK)
fr_top_menu.grid(row=0, column=0, padx=10, pady=5)

   # Sample Size
   lbl_sample_size = Label(
   master = fr_top_menu,
   text = "Sample size:",
     bg = MATERIAL_BLACK,
   fg = LIGHT_GRAY,
   )
```

```
lbl_sample_size.grid(row=0, column=0, padx=10, pady=5)

sample_size = StringVar()
cb_sample_size = ttk.Combobox(
master = fr_top_menu,
state = "readonly",
values = [*range(25, 251, 25)],
width = 12,
textvariable = sample_size
)
cb_sample_size.grid(row=1,  column=0,  padx=5,  pady=5)
cb_sample_size.current(0)

# Algorithm Selection Dropdown
lbl_algorithm = Label(
master = fr_top_menu,
text = "Algorithm:",
bg = MATERIAL_BLACK,
fg = LIGHT_GRAY
)
lbl_algorithm.grid(row=0, column=1, padx=10, pady=5)

selected_algorithm = StringVar()
cb_algo_selector = ttk.Combobox(
master = fr_top_menu,
state = "readonly",
textvariable = selected_algorithm,
values = [*algorithms.keys()],
width = 12
)
cb_algo_selector.grid(row=1, column=1, padx=5, pady=5)
```

```
cb_algo_selector.current(0)

# Speed
lbl_speed = Label(
master = fr_top_menu,
text = "Speed:",
bg = MATERIAL_BLACK,
fg = LIGHT_GRAY
)
lbl_speed.grid(row=0, column=2, padx=10, pady=5)

selected_speed = StringVar()
cb_speed_selector = ttk.Combobox(
master = fr_top_menu,
state = "readonly",
textvariable = selected_speed,
values = [*speed_levels.keys()],
width = 12
)
cb_speed_selector.grid(row=1, column=2, padx=5, pady=5)
cb_speed_selector.current(2)

# Generate Sample Button
def render_sample():
        max_size = int(sample_size.get())

    global array
    array = random.sample(range(1, (max_size + 1)),
    max_size) array_colors = [WHITE for i in array]
    render_array(window, canva, array, array_colors, 0)
```

```
btn_generate_sample = Button(
master = fr_top_menu,
text = "Generate Sample",
bg = DARK_GRAY,
fg = LIGHT_GRAY,
command = render_sample
)
btn_generate_sample.grid(row=0, column=3, padx=5, pady=5)

# Run Button
 def sort_sample():
     global array

    if len(array) == 0:
    messagebox.showerror(
    title = "Empty Array",
                message = "Array must be generated before sorting."
    )
    return

    method = selected_algorithm.get()
    speed = speed_levels[selected_speed.get()]

    if method not in [*algorithms.keys()]:
    messagebox.showerror(
    title = "Sorting Method Not Selected",
                message = "A sorting method must be chosen before sorting."
    )
    return

    algorithms[method](window, canva, array, speed)
```

```
        max_size = int(sample_size.get())
        final_speed = (0.625 / max_size)
        for i in range(len(array)):
                colors = [WHITE if j > i else (GREEN if j < i else RED) for j in range(len(array))]
        render_array(window, canva, array, colors, final_speed)


    btn_run = Button(
    master = fr_top_menu,
    text = "Run",
    bg = DARK_GRAY,
    fg = LIGHT_GRAY,
    width = 10,
    command = sort_sample
    )
    btn_run.grid(row=1, column=3, padx=10, pady=5)

    # Graphical Array Space
    canva = Canvas(
    master = window,
    width = 800,
    height = 400,
    bg = BLACK,
    highlightthickness = 0
    )
    canva.grid(row=1, column=0, padx=10, pady=5)

    window.mainloop()

if name =='main':
start_visualizer()
```

Sorting/_visual-algorithms.py

```
from tkinter import *

from ._render import render_array


# Colors

RED = '#d80000'

YELLOW = '#fbff00'

WHITE = '#ffffff'

PURPLE = '#5800ca'


# Sorting Algorithms


# Bubble Sort

def bubble_sort(window: Tk, canva: Canvas, array: list, speed: float) -> None:

n = len(array)

for i in range(n):

swapped = False

for j in range(0, n - 1 - i):

colors = [WHITE if k != j else RED for k in range(len(array))]

if array[j] > array[j + 1]:

swapped = True

array[j], array[j + 1] = array[j + 1], array[j]

                colors = [WHITE if k not in (j, j + 1) else YELLOW for k in range(len(array))]
```

```
            render_array(window, canva, array, colors, speed)
        if not swapped:
        break


# Insertion Sort
def insertion_sort(window: Tk, canva: Canvas, array: list, speed: float) -> None:
n = len(array)
for j in range(1, n):
colors = [WHITE if k != j else RED for k in range(n)]
while j > 0 and array[j - 1] > array[j]:
array[j], array[j - 1] = array[j - 1], array[j]
j-=1
colors = [WHITE if k not in (j, j - 1) else YELLOW for k in range(n)]
render_array(window, canva, array, colors, speed)
render_array(window, canva, array, colors, speed)


# Selection Sort
def selection_sort(window: Tk, canva: Canvas, array: list, speed: float) -> None:
n = len(array)
for i in range(n):
smallest_value = array[i]
smallest_index = i
for j in range(i, n):
if array[j] < smallest_value:
```

```
smallest_value = array[j]
smallest_index = j
            colors = [WHITE if k not in (j, smallest_index) else (RED if k == j else PURPLE)
for k in range(n)]
render_array(window, canva, array, colors, speed)
if smallest_index != i:
array[smallest_index], array[i] = array[i], array[smallest_index]
colors = [WHITE if k not in (i, smallest_index) else YELLOW for k in range(n)]
render_array(window, canva, array, colors, speed)


# Merge Sort
def _merge(array: list, start: int, mid: int, end: int):
temp = []
i, j = start, mid + 1
while i <= mid and j <= end:
if array[i] <= array[j]:
temp.append(array[i])
i+=1
else:
temp.append(array[j])
j+=1
while i <= mid:
temp.append(array[i])
i+=1
```

```
    while j <= end:
        temp.append(array[j])
    j+=1
    for k in range(len(temp)):
        array[start + k] = temp[k]


def merge_sort(window: Tk, canva: Canvas, array: list, speed: float, start: int = 0, end: int = None):
if end is None:
end = len(array) - 1
if start < end:
mid = (start + end) // 2
merge_sort(window, canva, array, speed, start, mid)
merge_sort(window, canva, array, speed, mid + 1, end)
_merge(array, start, mid, end)
            colors = [WHITE if x < start or x > end else (RED if x in (start, end) else YELLOW) for x in range(len(array))]
render_array(window, canva, array, colors, speed)


# Quick Sort
def _partition(array, start, end):
pivot = array[end]
i = start - 1
for j in range(start, end):
```

```
    if array[j] <= pivot:
    i+=1
    array[i], array[j] = array[j], array[i]
    array[i + 1], array[end] = array[end], array[i + 1]
    return i + 1


def quick_sort(window: Tk, canva: Canvas, array: list, speed: float, start: int = 0, end: int = None):
if end is None:
end = len(array) - 1
if start < end:
p = _partition(array, start, end)
quick_sort(window, canva, array, speed, start, p - 1)
quick_sort(window, canva, array, speed, p + 1, end)
colors = [WHITE if (x < start) or (x > end) else (RED if x in (start, end) else
YELLOW) for x in range(len(array))]
render_array(window, canva, array, colors, speed)
```