Student Information

Name: Yaren Delfin Özen

Course: Programming I

Programming I Semester Project

**Sorting Algorithms Visualizer Using Tkinter**

# Index

Instructions For Users …pg. 3

Understanding the Code For Programmers …pg. 5

Instruction Manual For Users

# Introduction

The Sorting Algorithms Visualizer is a Python program that demonstrates how five different sorting algorithms work. The algorithms are Bubble Sort, Selection Sort, Insertion Sort, Merge Sort and Quick Sort. The program uses Tkinter GUI library to create a graphical interface that displays the sorting algorithms in action.

# Installation

This program requires Python 3 to run. Please ensure that you have Python 3 installed on your machine. You can download Python from the official website: <https://www.python.org/downloads/>

This program also uses the following Python modules:

1. Tkinter
2. Random
3. Time

The modules are usually pre-installed with Python, so you don’t need to install them separately. But if you are not sure whether you have Tkinter installed or not, you can run the following command in your terminal or command prompt:

import tkinter

If there is no error message, you have Tkinter installed, and you can proceed with running the code.

# Running the Code

To run the code, open your terminal or command prompt and navigate to the directory where the code is saved. Then, run the following command:

python sorting\_algorithms\_visualizer.py

Once you run the command, a window will open displaying the GUI for the sorting algorithm visualization tool.

# Using the GUI

The GUI window has three tabs, one for each sorting algorithm: bubble sort, selection sort, and insertion sort, merge sort and quick sort.

To use the GUI, follow these steps:

1. Choose the algorithm you want to visualize by clicking on the corresponding tab.
2. Click on the "Sort" button to start the visualization.
3. Watch the algorithm in action as the elements in the list are compared, swapped, or moved. The GUI window will display the current state of the list after each step.
4. Once the algorithm is done, the GUI window will display the sorted list.
5. When done with the learning process, the user can click the “Quit” button to exit the program.

# Understanding the Code for Programmers

## For Bubble Sort Algorithm

### **bubble\_sort**():

* This function generates an array ‘a' containing number\_of\_indices (which is 10) number of random integers between 0 and 500.
* Then it calls the bubble\_sort\_visualize() function to sort the array using the bubble sort algorithm.

### **bubble\_sort\_visualize**(a):

* This function takes an array ‘a’ as an input and performs the bubble sort algorithm on it.
* The algorithm works by iterating through the array and comparing adjacent elements.
* If an element is greater than its adjacent element, then they are swapped.
* This process is repeated until the entire array is sorted.
* The function also calls the draw\_bubble() function to visually display each step of the sorting process on a canvas.

### **draw\_bubble(a, curr, sw):**

* This function takes an array a, and two optional parameters, curr and sw, that represent the current element and the element to be swapped with, respectively.
* The function creates a canvas and draws rectangles for each element of the array using different colors depending on whether the element is the current element or the one being swapped.
* The rectangles are labeled with the corresponding element's value using create\_text().
* The canvas is updated with each iteration of the bubble sort algorithm.

## For Selection Sort Algorithm

### **selection\_sort():**

* This function generates an array ‘a' containing number\_of\_indices (which is 10) number of random integers between 0 and 500.
* Then it calls the selection\_sort\_visualize() function to sort the array using the selection sort algorithm.

### **selection\_sort\_visualize(a):**

* This is the main function that performs the selection sort algorithm on the ‘a’ array.
* The algorithm works as follows:

For each element in the array (starting from the first element and ending with the second-to- last element), the algorithm finds the smallest element in the remaining unsorted portion of the array and swaps it with the current element.

The algorithm keeps track of the index of the smallest element found and swaps it with the current element at the end of each iteration of the outer loop.

This process repeats until the entire array is sorted.

* During each iteration of the outer loop, the draw\_selection() function is called to visualize the current state of the array.

### **draw\_selection(a, curr, min\_index):**

* This function visualizes the current state of the a array using the canvas2 object.
* It uses the enumerate() function to iterate over each element in the array and its corresponding index.
* For each element, it draws a rectangle on the canvas with a color based on whether the element is the current element (curr) or the smallest element found so far (min\_index).
* It also draws the element value inside the rectangle using the create\_text() function.
* Finally, it updates the left\_margin variable to position the next rectangle on the canvas.

## For Insertion Sort Algorithm

### **insertion\_sort():**

* This is the main function that generates a list of random numbers using the random module and calls the insertion\_sort\_visualize() function to sort the list using insertion sort algorithm.

### **insertion\_sort\_visualize(a):**

* This function implements the insertion sort algorithm to sort the list a. It first iterates through the entire list and selects each element one by one.
* The element is temporarily saved in the variable t.
* The function then displays a message on the canvas indicating that the current element is being "lifted up" using canvas3.create\_text(), which creates a text object in the canvas.
* It then starts a loop that iterates from the current element's index to the beginning of the list. The loop moves all the elements that are larger than the current element one position to the right, making space for the current element.
* The draw\_insertion() function is called to visualize the swapping process by highlighting the two elements being swapped.
* The root.update() method updates the window so the user can see the changes on the screen.
* The time.sleep(1.0) method causes the program to pause for one second before continuing.
* After the loop completes, the current element is inserted into its correct position in the sorted list by placing it in the index position j+1.
* Again, draw\_insertion() is called to visualize the insertion process.
* This process is repeated for each element in the list until the list is fully sorted.
* Finally, draw\_insertion() is called with None values to display the fully sorted list.

### **draw\_insertion(a, curr, j):**

* This function takes the list ‘a’ and two index values as arguments.
* The curr argument represents the current element being processed, and the j argument represents the index of the element that the curr element is being compared to.
* The function begins by clearing the canvas with the canvas3.delete("all") method.
* It then sets the left\_margin variable to 15, which is the leftmost position for the first rectangle.
* The width, height, and padding variables are set to the dimensions of the rectangle and the space between the rectangles.
* The function then loops through the list a, creating a rectangle for each element.
* If the current index matches the curr argument, the rectangle is filled with the color "steel blue”.
* If it matches the j argument, the rectangle is filled with the color "red4".
* Otherwise, the rectangle is filled with the color "gray6".
* The canvas3.create\_text() method is used to place the element's value inside the rectangle.
* The left\_margin variable is updated for each rectangle so that each one is displayed next to the previous one.
* When this function is called from insertion\_sort\_visualize(), it is called multiple times to visualize the swapping and insertion processes.
* By highlighting the elements being compared and swapped, the user can see how the sorting algorithm is working.

## For Merge Sort Algorithm

### **merge\_sort():**

* This function creates an array a of random numbers between 0 and 500 using the random module, with the length of the array specified by the number\_of\_indices variable.
* It then calls the merge\_sort\_visualize() function to sort the array using merge sort algorithm.

### **merge\_sort\_visualize(a):**

* This is the main merge sort function that takes an array a as input and sorts it using the merge sort algorithm.
* The function starts by calling the draw\_merge() function with a, None, and None as arguments to draw the initial state of the array.
* It then checks if the length of the array is greater than 1, and if it is, it splits the array into two halves, left and right, and calls itself recursively on each half.
* After the recursive calls return, the function merges the two sorted halves using the standard merge algorithm, and updates the visualization by calling draw\_merge() again with the sorted array and the left and right subarrays.
* Finally, the function calls draw\_merge() again with a, None, and None to draw the final state of the array.

### **draw\_merge(a, left, right):**

* This function takes an array a, and the left and right subarrays, and draws a visualization of the current state of the merge sort algorithm on a Tkinter canvas.
* The function starts by deleting all existing canvas items using the canvas4.delete("all") method.
* It then sets up some parameters for drawing the rectangles and text on the canvas, including the left margin, rectangle width, rectangle height, and padding.
* If left and right are both None, the function sets the color to "gray6".
* Otherwise, for each element in the array a, the function checks if it belongs to left or right, and sets the color accordingly to "red4" or "chartreuse4", respectively.
* If the element doesn't belong to either left or right, the function sets the color to "gray6".
* The function then draws a rectangle and text for each element in the array using the canvas4.create\_rectangle() and canvas4.create\_text() methods, respectively.
* The position and color of each rectangle and text are determined based on the left margin, width, height, and padding parameters.

## For Quick Sort Algorithm

### **quick\_sort():**

* It creates an array a with number\_of\_indices random integers between 0 and 500.
* Then it calls the quick\_sort\_visualize function to sort the array using the QuickSort algorithm.
* Finally, it calls the draw\_quick function to draw the sorted array on the canvas.

### **quick\_sort\_visualize(a, lo, hi):**

* This is the main QuickSort function that sorts the array a using the Hoare partitioning method.
* The function takes in three parameters a, lo, and hi, where a is the array to be sorted, and lo and hi are the indices of the first and last elements of the array to be sorted, respectively.
* The function checks if the lo index is less than the hi index, if yes, it calls the partition function to partition the array into two parts, and then recursively calls quick\_sort\_visualize on the left and right partitions.

### **partition(a, lo, hi):**

* This is a helper function used by the quick\_sort\_visualize function to partition the array using the Hoare partitioning method.
* The function takes in three parameters a, lo, and hi, where a is the array to be sorted, and lo and hi are the indices of the first and last elements of the array to be partitioned, respectively.
* The function first selects the pivot element piv as the first element of the array.
* It initializes two pointers i and j, where i points to the first element of the array, and j points to the last element of the array.
* The function then enters into a loop where it increments i until it finds an element greater than or equal to the pivot element, and decrements j until it finds an element less than or equal to the pivot element.
* If i is greater than or equal to j, the function returns j.
* Otherwise, the function swaps the elements at indices i and j and calls the draw\_quick function to draw the current state of the array on the canvas.
* The draw\_quick function is called with four parameters a, i, j, and lo, where a is the array to be drawn, i and j are the indices of the elements being swapped, and lo is the index of the pivot element.
* The root.update() and time.sleep(1.0) statements are used to slow down the drawing of the array on the canvas.

### **draw\_quick(a, curr, compared, piv):**

* This function is used to draw the QuickSort algorithm on the canvas.
* The function takes in four parameters a, curr, compared, and piv, where a is the array to be drawn, curr is the index of the current element being sorted, compared is the index of the element being compared to the pivot element, and piv is the index of the pivot element.
* The function starts by deleting all the previous drawings on the canvas using canvas5.delete(“all").
* It then initializes some variables for the left margin, width, height, and padding of the rectangles to be drawn.
* The function then loops over all the elements of the array and checks if the current element is equal to curr, compared, or piv.
* If the current element is equal to curr, the color of the rectangle is set to "chartreuse4", which is a shade of green.
* The variable curr represents the index of the current element that is being compared in the partitioning process.
* If the current element is equal to piv, a red text is displayed on the canvas indicating that this is the current pivot element. The color of the rectangle for this element is set to "red4".
* The variable piv represents the index of the pivot element that is being used in the partitioning process.
* If the current element is equal to compared, the color of the rectangle is set to "steel blue".
* The variable compared represents the index of the element that is being compared with the pivot element.
* For all other elements, the color of the rectangle is set to "gray6", which is a shade of gray.
* Finally, the function creates a rectangle on the canvas for each element of the array using the create\_rectangle method of the canvas5 object.
* It also displays the value of each element in the rectangle using the create\_text method of the canvas5 object.
* The position of the rectangle and the text are determined by the left\_margin, width, and height variables.
* These variables are updated in each iteration of the loop to move the rectangle and the text to the right.
* The padding variable is used to add some space between the rectangles.

## Breakdown of the GUI

* **root = tk.Tk():** Creates the main window of the GUI.
* **root.title("Sorting Algorithms Visualizer"):** Sets the title of the main window to "Sorting Algorithms Visualizer”.
* **tabControl = ttk.Notebook(root):** Creates a tabbed interface within the main window using the ttk.Notebook widget.
* **tab1 = ttk.Frame(tabControl):** Creates the first tab and stores it in the tab1 variable. This is done by creating a new ttk.Frame object and passing in tabControl as the parent widget.
* **tabControl.add(tab1, text='Bubble Sort'):** Adds tab1 to the tabbed interface with the title "Bubble Sort". The same process is repeated for the other tabs.
* **ttk.Button(tab1, text="Sort", command=bubble\_sort).grid(row=1, column=0):** Creates a "Sort" button in tab1 that, when clicked, calls the bubble\_sort function. The grid method is used to position the button in the first row and first column of tab1.
* **ttk.Button(tab1, text="Quit", command=tab1.quit).grid(row=1, column=1):** Creates a "Quit" button in tab1 that, when clicked, quits the tab1 frame. The same process is repeated for the other tabs.
* **canvas = tk.Canvas(tab1, width=610, height=250, bg="grey"):** Creates a canvas widget in tab1 with a width of 610 pixels, a height of 250 pixels, and a grey background color. The same process is repeated for the other tabs.
* **canvas.grid(row=2, column=0, columnspan=2):** Positions the canvas widget in the second row of tab1 that spans both columns. The same process is repeated for the other tabs.
* **ttk.Label(tab1, text="Bubble Sort Visualization").grid(row=0, column=0):** Creates a label widget in tab1 with the text "Bubble Sort Visualization" and positions it in the first row and first column of tab1. The same process is repeated for the other tabs.
* **tabControl.pack(expand=1, fill="both"):** Packs the tabControl widget, which contains all the tabs, so that it expands to fill the main window.
* **root.mainloop():** Starts the main event loop of the GUI, which waits for user input and responds accordingly. This line must always be included at the end of a tkinter program.

Conclusion

In conclusion, this program demonstrates the visualization of five sorting algorithms: Bubble sort, Selection sort, and Insertion sort, Merge Sort and Quick Sort. The program generates an array of random numbers and sorts them using the selected sorting algorithm while visualizing the sorting process. The user can see how the elements in the array are being swapped and how the final sorted array looks. The program uses the tkinter library to draw the sorting process on a canvas. It also uses the time library to add some delay to the visualization to make it easier for the user to follow.