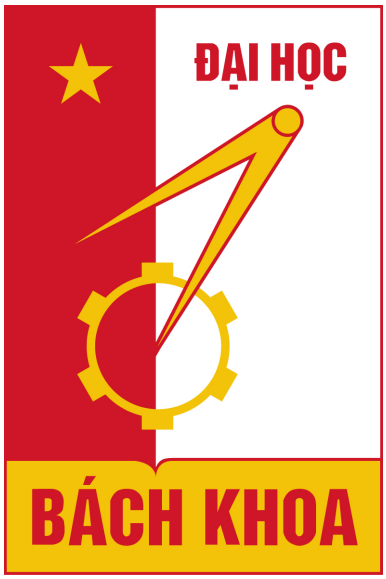
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION TECHNOLOGY AND COMMUNICATION

——————— o0o ———————



OOP PROJECT REPORT

Visualizing Sorting Algorithms: A Dynamic Learning Experience

Instructors : Prof. Tran The Hung

Group 27 : Nguyen Bui Duc Anh 20220070
Khuat The Anh - 20226008
Nguyen Tran Ngan Ha - 20225969
Vuong Le Binh Duong - 20226035
Vu Thai Hung - 20226046

Hanoi, Jun 2024

Visualizing Sorting Algorithms

A Dynamic Learning Experience

Group 11:

Nguyen Bui Duc Anh 20220070
Khuat The Anh - 20226008
Nguyen Tran Ngan Ha - 20225969
Vuong Le Binh Duong - 20226035
Vu Thai Hung - 20226046

Hanoi, June 2024

Abstract

This report explores the Sorting Visualization project, which leverages Object-Oriented Programming (OOP) methodologies. The project's aim is to create visual animations of different sorting algorithms to aid in comprehension and facilitate the educational process.

Contents

1 Problem Definition

1.1 Background

Sorting algorithms are fundamental to computer science and are extensively used in various domains both theoretical and applied. Mastery over these algorithms is crucial for students and professionals alike as they underpin many computational processes and applications. Understanding how sorting algorithms function alongside their efficiency in different scenarios forms a core part of computer science education. This report details the development of a Java Swing application designed to visually demonstrate the workings of several key sorting algorithms. By transforming abstract algorithmic concepts into dynamic visualizations the application acts as an educational tool enhancing comprehension and engagement.

1.2 Project Objectives

The primary objective of this project is to create an interactive platform that visually illustrates the execution of sorting algorithms. Specifically the application will showcase the operation of algorithms including Merge Sort, Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, and Shell Sort. By providing real-time step-by-step visualizations of these algorithms the tool aims to facilitate a deeper understanding of their mechanics and performance. Users will be able to see how data is manipulated and organized through each step of the algorithm gaining insights into the comparative speed and efficiency of these sorting techniques under various conditions. This interactive learning approach is intended to make the study of sorting algorithms more intuitive and accessible.

2 Project Functionality

The Java Swing application is designed to be user-friendly and intuitive providing a range of functionalities that support the visualization and understanding of sorting algorithms. Below is a detailed overview of its features and capabilities:

- **Main Menu:** The main menu serves as the central hub for navigating the application. It includes:
  - **Menu Bar:** This bar is prominently positioned at the top of the interface providing easy access to various options.
  - **Help Menu:** A comprehensive help section is available to guide users on how to use the application, understand the features and troubleshoot common issues.
  - **Description of Sorting Algorithms:** Detailed descriptions of each sorting algorithm are provided explaining their principles, use cases, and performance characteristics.
  - **Choose Sorting Algorithm:** Users can select from a list of sorting algorithms such as Merge Sort, Bubble Sort, Insertion Sort, and Selection Sort to visualize.

- **Sorting Algorithm Visualizer:** Each sorting algorithm has a dedicated visualizer page with the following controls:
  - **Create Array Button:** This button generates a new random array of non-negative integers to be sorted. Users can repeatedly press this button to create different arrays for comparison.
  - **Sort Button:** Once an array is generated, the sort button initiates the sorting process. Users can watch the sorting algorithm organize the data in real-time.
  - **Stop Button:** This allows users to pause the sorting visualization at any point giving them time to analyze the current state of the array.
  - **Continue Button:** After pausing, the continue button resumes the sorting process from where it left off.
  - **FPS Slider:** A slider control is available to adjust the frames per second (FPS) of the visualization. This lets users control the speed of the sorting animation making it possible to slow down or speed up the process for better observation.
- **Visualization:** The core functionality of the application is the real-time visualization of sorting algorithms:
  - **Bar Charts:** The application represents data as bar charts where each bar's height corresponds to a data element's value. This visual representation helps users intuitively understand how the sorting algorithm manipulates the data.
  - **Interactive Sorting Process:** As the algorithm progresses, users can see the bars rearrange in real-time providing a clear and dynamic view of the sorting steps.
- **Navigation and Exiting:**
  - **Back to Main Menu:** Users can easily navigate back to the main menu to select a different algorithm or adjust settings.
  - **Exit Application:** An option is provided to exit the application safely ensuring that any ongoing processes are properly terminated.

These functionalities collectively make the application a powerful educational tool allowing users to engage with and understand sorting algorithms in a visual and interactive manner.

3 Project Description

3.1 Project Requirements

To ensure the application operates smoothly and effectively conveys the sorting processes, certain requirements and constraints have been established:

- **Data Types:** The application only accepts non-negative integers as elements of the array. This restriction simplifies the sorting process and avoids complexities associated with sorting other data types or negative numbers.
- **Valid Array Size Range:** Arrays to be sorted must have sizes ranging from a minimum of 100 to a maximum of 500 elements. This range provides flexibility for users to explore the algorithms' behaviors with different data set sizes while maintaining the application's visual and performance integrity.

These requirements are in place to optimize the application's functionality and user experience, providing a clear and efficient platform for learning about sorting algorithms through visual means.

3.2 Use Case Diagram and Explanation

The use case diagram includes functionalities such as choosing a sorting algorithm, generating data, visualizing the sorting process, viewing the help menu, and navigating back to the main menu or exiting the application.

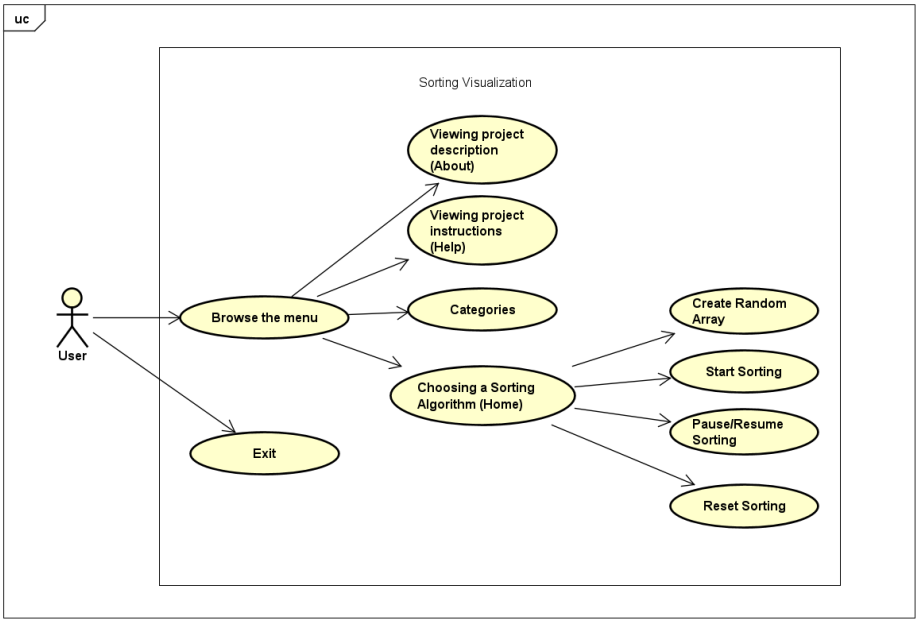


Figure 1: Usecase Diagram

3.3 Class Diagram and Explanation

The application employs object-oriented programming concepts to create a structured and easily maintainable codebase. Diagrams representing classes (substitute with placeholders for your actual diagrams) depict how different parts of the application interact. These diagrams specifically

outline the operations of modules such as Sorting, MainMenu, Bars, ButtonsPanel, SortFrame, and Visualizer.

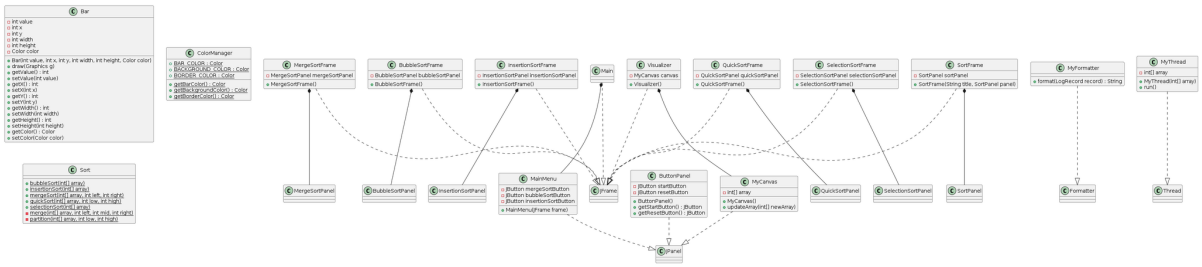


Figure 2: Class Diagram

4 Implementation

The primary operations focus on executing selected sorting algorithms (Merge Sort, Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, and Quick Sort) located in the Sorting package. Each algorithm is implemented within its dedicated class. The MainMenuApplication package manages tasks associated with the main menu and user engagement.

4.1 Class structure

Visualizer: The Visualizer class is the core component of your sorting visualization application. It is responsible for managing the graphical representation of the sorting process, providing controls for the user to interact with the visualizations and implementing various sorting algorithms.

- **Attributes:**
  - **Array and Bars:** It maintains an array of integers and a corresponding array of Bar objects that visually represent each element in the array.
  - **Colors:** It defines several colors to distinguish between original, comparing, and swapping states of the bars during the sorting process.
  - **Timing and Statistics:** It tracks the sorting process in terms of time taken, number of comparisons, and swaps made.
  - **Control Flags:** Flags for pausing and resuming the sorting operations and specific flags for controlling bubble and insertion sorts.
  - **Graphics and Buffer:** Handles graphics rendering and buffer strategy for smooth visual updates.
- **Methods:**

  - **Array Creation:** Methods to create random arrays, either with unique values or potential duplicates, and initialize their visual representations.
  - **Sorting Algorithms:** Implements classic sorting algorithms (Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, and Shell Sort) with detailed visual feedback for each step.
  - **Controls:** Methods to pause, resume, and handle graphical updates during the sorting process.
  - **Utility Functions:** Helper methods for swapping elements, coloring bars during comparisons and swaps, and redrawing the bars on the canvas.

- **Sorting Animations:**
  - **Swap:** Swaps two elements in the array and visually updates their positions and colors.
  - **Coloring:** Temporarily changes the color of bars to indicate comparison and swap actions and reverts them back after a delay.
  - **Finish Animation:** Highlights the bars when the sorting is complete and notifies the listener with sorting statistics.

- **Pause/Resume Handling:** Provides functionality to pause and resume the sorting operation, particularly useful for controlling the flow in a multi-threaded or interactive environment.

- **Listeners and Buffer Strategy:** The SortedListener interface is used to notify the application when the array is sorted and to provide access to the buffer strategy for graphical rendering.

SortFrame: The SortFrame class in Java Swing is designed to create a graphical user interface for visualizing sorting algorithms. It extends JFrame and manages several UI components for user interaction and visualization.

- **Main Features:**
  - **Full-Screen Window:** Adapts to the screen's full size but maintains a fixed position and non-resizable setup.
  - **UI Components:**
    * **Button Panel:** Contains controls for starting and managing sorting operations.
    * **Canvas (MyCanvas):** A large area for rendering the sorting process visually.
    * **Input Panel:** Includes a capacity input field and a checkbox to specify unique values in the array.
    * **Slider Panel:** Features an FPS slider to adjust the animation speed.
    * **Information Panel:** Displays real-time sorting statistics like elapsed time, comparisons, and swaps.

- **Interactivity:**
  - Listens to changes in the input field and slider to update sorting parameters.
  - Confirms with the user before closing the window.
- **Sorting Visualization:**
  - Uses a Visualizer to handle sorting logic and draw the sorted array on the canvas.
  - Ensures smooth animation using a double-buffered rendering strategy.

Bar: The Bar class represents a graphical bar in a sorting visualization designed to be drawn on a canvas in a Java Swing application. It encapsulates the properties and behaviors of an individual bar including its position, size, value (height), and color.

- **Attributes:**
  - **x and y:** Coordinates of the bar's bottom-left corner.
  - **width:** The width of the bar.
  - **value:** The height of the bar.
  - **color:** The color used to draw the bar.
  - **MARGIN:** A margin to ensure spacing between bars.
- **Constructor:** Initializes the bar's position, dimensions, value, and color.
- **Methods:**
  - **draw(Graphics g):** Draws the bar on the given Graphics object with its current attributes.
  - **clear(Graphics g):** Clears the bar from the canvas by painting over it with the background color.
  - Getter and setter methods for the bar's value and color (getValue, setValue, getColor, setColor).

ButtonPanel: The ButtonPanel class is a custom JPanel component that provides a panel of interactive buttons for a sorting visualization application. Each button is represented by a JLabel and is designed to respond to user mouse events such as clicks and hovers.

- **Key Components and Features:**
  - **Constants:**
    - BUTTON_WIDTH and BUTTON_HEIGHT: Define the dimensions of each button.
    - serialVersionUID: A version control identifier for the class.

- **Attributes:**
  - buttons: An array of JLabels representing the buttons.
  - listener: A SortButtonListener interface used to handle button click events.
  - number: Specifies the number of buttons on the panel (default is 5).
- **Constructor:** Initializes the panel and sets up each button with an icon and mouse event listeners. Adds each button to the panel with specific layout constraints.
- **Methods:**
  - initButtons(JLabel button, String name, int id): Configures each button's appearance and behavior for various mouse events (click, enter, exit, press, release). Changes the button's icon based on these interactions and notifies the listener when a button is clicked.
- **Interface:**
  - SortButtonListener: An interface that defines the sortButtonClicked method which is triggered when a button is clicked.

MyCanvas: MyCanvas is a Java class extending Canvas designed to draw graphics based on events received from a VisualizerProvider. Key features include:

- Initialization through a constructor that accepts a VisualizerProvider listener.
- Overrides paint(Graphics g) to clear the canvas and invoke listener.onDrawArray() for drawing.
- Provides a clear(Graphics g) method to set the canvas background.
- Defines the VisualizerProvider interface for callback methods, particularly onDrawArray() for drawing operations.

4.2 Algorithm Implementation

Selection Sort:

- Selection Sort is a straightforward sorting algorithm that iteratively identifies the smallest element from the unsorted part of the array and places it at the beginning.
- Implementation:

```
public class SelectionSortFrame extends SortFrame {
    public SelectionSortFrame() {
        super("Selection Sort Algorithm Visualizer");
    }
}
```

Bubble Sort:

- Bubble Sort is a basic sorting algorithm that iteratively passes through the list, compares adjacent elements and swaps them if they are in the incorrect order.
- Implementation:

```
public class BubbleSortFrame extends SortFrame {
    public BubbleSortFrame() {
        super("Bubble Sort Algorithm Visualizer");
    }
}
```

Insertion Sort:

- Insertion Sort is a basic sorting algorithm that constructs the final sorted array one element at a time by repeatedly placing elements larger than the current element ahead of their current position.
- Implementation:

```
public class InsertionSortFrame extends SortFrame {
    public InsertionSortFrame() {
        super("Insertion Sort Algorithm Visualizer");
    }
}
```

Shell Sort:

- Shell Sort is an efficient variation of insertion sort. It starts by sorting pairs of elements far apart from each other and progressively reduces the gap between elements to be compared. By the final pass it sorts adjacent elements, creating a fully sorted array.
- Implementation:

```
public class ShellSortFrame extends SortFrame {
    public ShellSortFrame() {
        super("Shell Sort Algorithm Visualizer");
    }
}
```

Merge Sort:

- Merge Sort is a divide and conquer algorithm that divides the input array recursively into two halves until each half contains only one element. It then merges these halves in a sorted manner.

- Implementation:

```
public class MergeSortFrame extends SortFrame {
    public MergeSortFrame() {
        super("Merge Sort Algorithm Visualizer");
    }
}
```

Quick Sort:

- Quick Sort is a divide and conquer algorithm. It selects a "pivot" element from the array and partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot. It then recursively sorts the sub-arrays.
- Implementation:

```
public class QuickSortFrame extends SortFrame {
    public QuickSortFrame() {
        super("Quick Sort Algorithm Visualizer");
    }
}
```

5 Visualizations and Performance Analysis

The application generates real-time visualizations of the sorting process using bar charts. These visualizations allow users to observe how the algorithms arrange the data elements. Further analysis can be conducted to assess the performance of each sorting algorithm based on the visualizations generated.

1. **Merge Sort:**
   - Elapsed Time: Merge Sort has a time complexity of $O(n \log n)$ in the average and worst cases.
   - Number of Comparisons: Merge Sort typically makes $O(n \log n)$ comparisons in the worst case.
   - Number of Swaps: Merge Sort does not perform swaps in the traditional sense as it merges pre-sorted subarrays, hence it has $O(n \log n)$ merge operations.
2. **Selection Sort:**
   - Elapsed Time: Selection Sort has a time complexity of $O(n^2)$ in the average and worst cases.

- Number of Comparisons: Selection Sort makes $O(n^2)$ comparisons in the worst case
- Selection Sort makes $O(n)$ swaps in the worst case.

3. **Bubble Sort:**
   - Elapsed Time: Bubble Sort also has a time complexity of $O(n^2)$ in the average and worst cases.
   - Number of Comparisons: Bubble Sort makes $O(n^2)$ comparisons in the worst case.
   - Number of Swaps: Bubble Sort makes $O(n^2)$ swaps in the worst case.
4. **Insertion Sort:**
   - Elapsed Time: Insertion Sort has a time complexity of $O(n^2)$ in the average and worst cases but can perform better in nearly sorted or small datasets.
   - Number of Comparisons: Insertion Sort makes $O(n^2)$ comparisons in the worst case.
   - Number of Swaps: Insertion Sort makes $O(n^2)$ swaps in the worst case.
5. **Shell Sort:**
   - Elapsed Time: Shell Sort has a time complexity that varies depending on the gap sequence used but generally falls between $O(n \log^2 n)$ and $O(n^2)$.
   - Number of Comparisons: Shell Sort typically makes fewer comparisons than insertion sort but more than merge sort.
   - Number of Swaps: Shell Sort makes $O(n \log n)$ swaps in the average case.
6. **Quick Sort:**
   - Elapsed Time: Quick Sort has a time complexity of $O(n \log n)$ in the average case but $O(n^2)$ in the worst case (rare).
   - Number of Comparisons: Quick Sort makes $O(n \log n)$ comparisons on average.
   - Number of Swaps: Quick Sort makes $O(n \log n)$ swaps on average.

Performance Comparison Summary:

- Merge Sort generally outperforms Bubble Sort, Selection Sort, and Insertion Sort in terms of time complexity, especially for large datasets.
- Quick Sort is efficient for average case scenarios but may degrade to $O(n^2)$ in rare worst-case scenarios.
- Shell Sort provides a balance between complexity and performance, often faster than simple quadratic sorts like Bubble, Selection, and Insertion Sorts.

These performance metrics, combined with real-time visualizations, provide a comprehensive understanding of how each sorting algorithm behaves under different conditions and dataset sizes.

6 Conclusion

This project has effectively created an easy-to-use application that illustrates sorting algorithms visually. The interactive features of the application enhance comprehension of these algorithms, rendering it a useful resource for students and developers alike. Further enhancements could include integrating more sorting algorithms and exploring different visualization methods.