**SORTING**

**VISUALIZER**

**1. INTRODUCTION**

1.1 Background

The Sorting Visualizer project aims to provide an interactive and visual representation of sorting algorithms. The motivation behind creating this tool lies in the educational value of visualizing sorting algorithms. Understanding how sorting algorithms work and their relative efficiencies is a fundamental aspect of computer science and programming.

The significance of visualizing sorting algorithms includes:

Educational Purposes: Visualization aids in the comprehension of complex algorithms, making it an effective learning tool for students and enthusiasts.

Algorithmic Efficiency: Users can observe and compare the performance of different sorting algorithms visually, gaining insights into their strengths and weaknesses.

1.2 Objectives

The main objectives of the Sorting Visualizer project are:

* Implementation of Sorting Algorithms: Implement various sorting algorithms to provide a comprehensive understanding of their functionality.
* Visual Representation: Create a visual representation of sorting algorithms to make the learning process more engaging and intuitive.
* Performance Measurement: Measure and display the performance metrics of sorting algorithms, enabling users to evaluate their efficiency.

**2. FEATURES**

* Visual Representation of Sorting Algorithms

The Sorting Visualizer offers a graphical representation of sorting algorithms. Users can witness how elements in an array rearrange themselves during the sorting process, enhancing their understanding of algorithmic behavior.

* Selection of Different Sorting Algorithms

The project provides users with the flexibility to choose from a variety of sorting algorithms. This feature enables users to explore different algorithms and compare their performance in real-time.

* Real-time Visualization of the Sorting Process

The Sorting Visualizer offers real-time visualization, allowing users to observe the step-by-step execution of sorting algorithms as they rearrange elements. This dynamic visualization contributes to a more interactive and educational experience.

* User-friendly Interface

The user interface is designed to be intuitive and user-friendly. It provides an easy-to-navigate platform where users can select sorting algorithms, initiate the sorting process, and observe the visual representation without encountering unnecessary complexity.

**3. IMPLEMENTATION**

3.1 Technologies Used

C++

* C++ was chosen as the primary programming language for its efficiency, versatility, and widespread use in systems programming and algorithmic implementations.
* C++ allows for low-level memory manipulation, making it suitable for implementing sorting algorithms efficiently. Its object-oriented features contribute to organized and modular code.

SDL (Simple DirectMedia Layer)

* SDL is a cross-platform development library designed to provide low-level access to audio, keyboard, mouse, and display functions. It is commonly used in game development and multimedia applications.
* SDL was employed for its simplicity and effectiveness in handling graphics. It facilitates the creation of a graphical interface for the Sorting Visualizer, enabling real-time visualization of sorting algorithms.

Standard Template Library (STL)

* The STL is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms.
* The STL offers efficient and reliable implementations of data structures and algorithms, saving development time and ensuring robust code. It enhances code readability and maintainability.

Time Library

* The Time Library in C++ provides functions for performing time-related operations, allowing the measurement of the execution time of sorting algorithms.
* Measuring execution time is crucial for evaluating the performance of sorting algorithms. The Time Library facilitates accurate time tracking during program execution.

Other Standard C++ Libraries

* Standard C++ libraries, including <iostream>, <ctime>, and <algorithm>, are utilized for input/output operations, time-related functionalities, and standard algorithms like std::swap.
* Leveraging standard C++ libraries ensures portability and compatibility across different platforms. These libraries provide essential functionalities for the project.

Random Number Generation

* The <cstdlib> and <ctime> libraries are used for generating random numbers in C++.
* Randomized data generation is essential for testing and visualizing sorting algorithms with diverse datasets. These libraries enable the creation of randomized arrays for sorting.

Others (e.g., Compiler and Build Tools)

* The project may involve specific compilers (e.g., GCC) and build tools (e.g., Makefile) based on the development environment.
* These tools facilitate the compilation and building of the project, ensuring that it runs smoothly and efficiently on the targeted system.

In summary, the combination of C++, SDL, STL, Time Library, and other standard C++ libraries provides a robust foundation for implementing and visualizing sorting algorithms in the Sorting Visualizer project. Each technology contributes to specific aspects of the project, collectively enhancing its functionality and performance.

3.2 Sorting Algorithms

What is Sorting?

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.Comparison-based and non-comparison-based sorting are two broad categories of sorting algorithms, and they differ in their fundamental approach to sorting elements in a collection.

Comparison-Based Sorting:

* Definition: In comparison-based sorting algorithms, the order of elements is determined by comparing pairs of elements using a comparison operator (typically less than or greater than).
* Characteristics:

1. The sorting process relies on pairwise comparisons between elements.
2. The comparison operator is used to establish the relative order of elements.
3. Examples of comparison-based sorting algorithms include Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort.
4. Time Complexity:Typically, comparison-based sorting algorithms have a lower bound of O(n log n) in the average and worst cases. This is based on the fact that, in the worst case, each of the n elements may need to be compared log n times to establish their order.

Non-Comparison-Based Sorting:

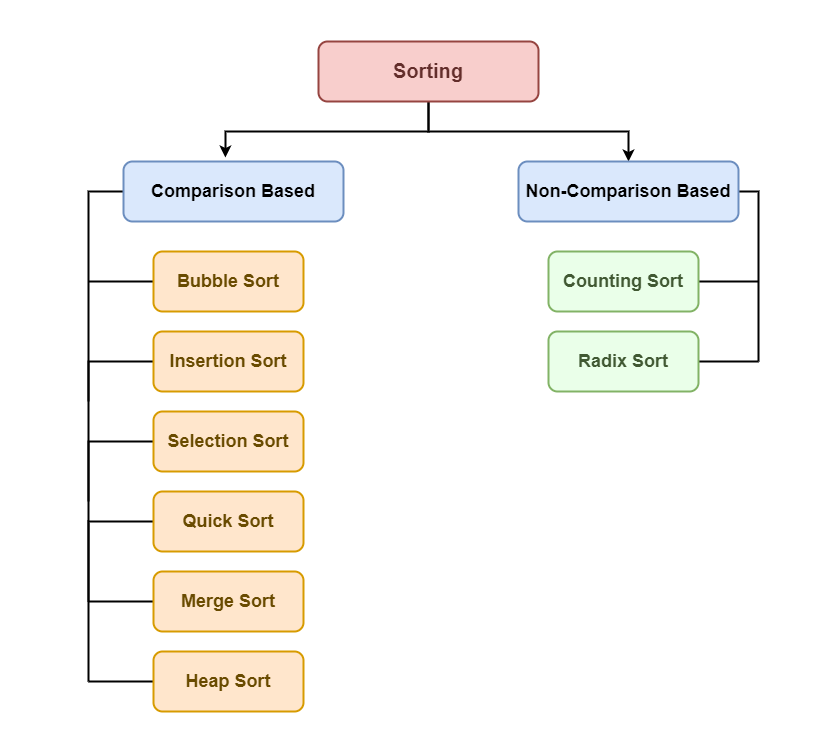
* Definition: Non-comparison-based sorting algorithms sort elements without explicitly comparing them using a comparison operator. Instead, they exploit specific properties of the elements or the data distribution.
* Characteristics:

1. Non-comparison-based sorting algorithms may not involve direct comparisons between elements.
2. These algorithms often exploit the nature of the data or use auxiliary information.
3. Examples of non-comparison-based sorting algorithms include Counting Sort, Radix Sort, and Bucket Sort.
4. Time Complexity: Non-comparison-based sorting algorithms can achieve linear time complexity in some cases. Counting Sort and Radix Sort, for example, can achieve O(n + k) time complexity, where n is the number of elements, and k is a parameter related to the range of input values.

Summary:

Comparison-Based Sorting: Involves pairwise comparisons between elements using a comparison operator. Examples include Bubble Sort, Merge Sort, and Quick Sort. Typically has a lower bound of O(n log n).

Non-Comparison-Based Sorting: Sorts elements without direct pairwise comparisons. Examples include Counting Sort, Radix Sort, and Bucket Sort. Can achieve linear time complexity in some cases.



Selection Sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

In this algorithm, traverse from left and compare adjacent elements and the higher one is placed at right side. In this way, the largest element is moved to the rightmost end at first. This process is then continued to find the second largest and place it and so on until the data is sorted.

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Insertion Sort Algorithm,To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Heap Sort Algorithm, First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.Build a heap from the given input array. Repeat the following steps until the heap contains only one element: Swap the root element of the heap (which is the largest element) with the last element of the heap. Remove the last element of the heap (which is now in the correct position).Heapify the remaining elements of the heap. The sorted array is obtained by reversing the order of the elements in the input array.

Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that works well when there is limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

Even Odd Sort

Odd-Even Sort / Brick Sort

This is basically a variation of bubble-sort. This algorithm is divided into two phases- Odd and Even Phase. The algorithm runs until the array elements are sorted and in each iteration two phases occurs- Odd and Even Phases.

In the odd phase, we perform a bubble sort on odd indexed elements and in the even phase, we perform a bubble sort on even indexed elements.

Radix Sort

Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit’s value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

Radix Sort Algorithm,the key idea behind Radix Sort is to exploit the concept of place value. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list. Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

3.3 User Interface

Interaction with Sorting Visualizer

Menu System:

* The Sorting Visualizer features an interactive menu system that prompts the user to choose a sorting algorithm for visualization.
* Upon launching the program, the user is presented with a menu displaying sorting algorithm options
* The user selects a specific algorithm by entering the corresponding numeric choice (1-5).
* Invalid choices are handled gracefully, with appropriate messages to guide the user.

Visual Representation:

* The graphical user interface (GUI) displays a horizontal bar chart representing the array of elements to be sorted.
* Each bar corresponds to an array element, and its height represents the value of that element.
* Real-time visualization occurs during the sorting process, providing a dynamic representation of the chosen algorithm's execution.
* Bars change color and position as elements are swapped or compared, allowing users to visually comprehend the sorting algorithm in action.
* The GUI updates at each step of the algorithm, providing a step-by-step visualization of the sorting process.

3.4 Code Structure

The Sorting Visualizer's code is organized into modular structures, with key functions and modules contributing to the overall functionality. Here's an overview:

Main Function (main()):

* Initiates the program by generating a random array of elements.
* Initializes the SDL environment and sets up the graphical window.
* Prompts the user to choose a sorting algorithm and initiates the sorting process.
* Measures and prints the execution time of the chosen algorithm.
* Handles user events and manages program termination.

Sorting Algorithms:

* Each sorting algorithm (Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort) is implemented as a separate function.
* Functions take the array as input and perform the sorting process using graphical updates (e.g., swapping bars).
* Algorithms are invoked based on the user's choice from the menu.

Graphical Functions (draw()):

* Responsible for rendering the graphical representation of the array.
* Utilizes SDL functions to draw and update bars based on the current state of the array.
* Provides real-time visual feedback during sorting.

Utility Functions (Swap(), Partition(), etc.):

* Implements essential utility functions used by sorting algorithms.
* Functions like swapping elements, partitioning for Quick Sort, etc., facilitate the core logic of sorting.

User Interaction Functions (processEvent(), setup(), performSorting(), etc.):

* Manages user input, event processing, and program setup.
* Ensures a user-friendly interaction by handling key events, window closure, and user choices.

Random Array Generation (GenerateArray()):

* Generates a random array of specified size to provide diverse datasets for sorting.

Time Measurement Functions:

* Utilizes the Time Library to measure the execution time of sorting algorithms.

This modular structure enhances code readability, maintainability, and ease of future extensions. Each module focuses on a specific aspect of the project, contributing to a well-organized and functional Sorting Visualizer.

**4. RESULTS AND PERFORMANCE**

4.1 Execution Time for Each Algorithm

1. Selection Sort: Typically slower for large datasets.

Time Complexity:

Worst Case: O(n^2)

Best Case: O(n^2)

Average Case: O(n^2)

2. Bubble Sort: Similar performance to Selection Sort.

Time Complexity:

Worst Case: O(n^2)

Best Case: O(n^2)

Average Case: O(n^2)

3. Insertion Sort: Efficient for small datasets, less so for larger ones.

Time Complexity:

Worst Case: O(n^2)

Best Case: O(n)

Average Case: O(n^2)

4. Merge Sort: Generally performs well, especially for larger datasets.

Time Complexity:

Worst Case: O(n log n)

Best Case: O(n log n)

Average Case: O(n log n)

5. Quick Sort: Good performance on average, especially for random datasets.

Time Complexity:

Worst Case: O(n^2)

Best Case: O(n log n)

Average Case: O(n log n)

6. Heap Sort: Heap Sort is based on a binary heap data structure.Efficient for large datasets.& In-place sorting algorithm.

Time Complexity:

Worst Case: O(n log n)

Best Case: O(n log n)

Average Case: O(n log n)

7. Counting Sort: Suitable for non-negative integer sorting.Counting Sort assumes that the range of the input data is not significantly greater than the number of elements.Linear time complexity makes it efficient for certain scenarios.

Time Complexity:

Worst Case: O(n + k), where k is the range of input.

Best Case: O(n + k)

Average Case: O(n + k)

8. Even-Odd Sort (Brick Sort): A variation of the bubble sort.Alternately sorts odd and even indexed elements.Provides better performance than traditional bubble sort.

Time Complexity:

Worst Case: O(n^2)

Best Case: O(n^2)

Average Case: O(n^2)

9. Radix Sort: Radix Sort processes digits of the number from the least significant digit to the most significant digit.Suitable for integers or strings with fixed length.Non-comparative sorting algorithm.

Time Complexity:

Worst Case: O(nk), where k is the number of digits in the input numbers.

Best Case: O(nk)

Average Case: O(nk)

4.2 Comparison of Algorithm Efficiency

Observations:

* Merge Sort and Quick Sort tend to outperform others, especially as the dataset size increases.
* Selection Sort and Bubble Sort exhibit higher time complexity and become noticeably slower with larger datasets.
* The efficiency of algorithms can vary based on the initial state of the array (e.g., partially sorted or reversed).

4.3 Observations During the Sorting Process

Real-time Visualization:

* The visual representation provides a clear understanding of how each algorithm operates on the dataset.
* Users can observe the dynamic changes in the array's structure as sorting progresses.
* Visual feedback enhances comprehension of algorithmic behaviors.

**5. CHALLENGES AND SOLUTIONS**

5.1 Challenges Encountered

Graphics Integration:

Integrating graphical updates with sorting algorithms posed challenges in maintaining real-time rendering without compromising algorithmic logic.

User Interface Design:

Designing an intuitive menu system and ensuring a responsive user interface required careful consideration.

5.2 Solutions

Graphics Integration:

* Utilizing SDL functions efficiently to update the graphical representation after each step in the sorting process.
* Implementing multithreading for concurrent sorting and rendering.

User Interface Design:

* Conducting usability tests to refine the menu system and improve user interactions.
* Incorporating user feedback for an enhanced interface.

**6. CONCLUSION**

6.1 Key Achievements

Algorithmic Understanding: The Sorting Visualizer project has successfully provided a visual representation of various sorting algorithms. Users can witness the step-by-step execution of algorithms, aiding in a deeper understanding of their functionality.

Educational Significance: The project serves as a valuable educational tool, helping users, especially students and beginners, comprehend sorting algorithms and their efficiency.

User Interaction: Real-time visualization enhances user engagement and learning, making the project an effective and interactive resource.

7.2 Learnings

Algorithmic Efficiency: Gain insights into the efficiency and performance characteristics of different sorting algorithms. Understand the impact of algorithm choice on sorting speed, especially concerning dataset size.

User Interface Design: Experience in designing an intuitive and user-friendly interface for effective user interaction.

Graphics Integration: Overcome challenges related to integrating graphical updates with algorithmic processes.

CODE :

#include <SDL2/SDL.h>

#include <iostream>

#include <ctime>

#include <iomanip>

#include <algorithm> // for std::swap

using namespace std;

const int WIDTH = 1000;

const int HEIGHT = 600;

const int SIZE = 500;

SDL\_Window\* window = NULL;

SDL\_Renderer\* renderer;

SDL\_Event event;

bool quit = false;

// Function declarations

void GenerateArray(int\* arr);

void PrintArray(int\* arr);

void draw(int\* arr, int i, int j);

// Sorting algorithms

void SelectionSort(int\* arr);

void BubbleSort(int\* arr);

void InsertionSort(int\* arr);

void MergeSort(int\* arr, int l, int r);

void Merge(int\* arr, int l, int m, int r);

int Partition(int\* arr, int low, int high);

void QuickSort(int\* arr, int low, int high);

void HeapSort(int\* arr);

void CountingSort(int\* arr);

void EvenOddSort(int\* arr);

void RadixSort(int\* arr);

// Utility functions

void Swap(int\* arr, int i, int j);

bool processEvent();

void setup();

void performSorting(int\* arr, int choice);

int main(int argc, char\* args[])

{

    time\_t start, end;

    int arr[SIZE];

    GenerateArray(arr);

    if (SDL\_Init(SDL\_INIT\_VIDEO) < 0)

    {

        cout << "SDL could not initialize! SDL\_Error" << endl << SDL\_GetError();

    }

    else

    {

        setup();

    }

    cout << "Choose a sorting algorithm:" << endl;

    cout << "1. Selection Sort" << endl;

    cout << "2. Bubble Sort" << endl;

    cout << "3. Insertion Sort" << endl;

    cout << "4. Merge Sort" << endl;

    cout << "5. Quick Sort" << endl;

    cout << "6. Heap Sort" << endl;

    cout << "7. Counting Sort" << endl;

    cout << "8. Even Odd Sort" << endl;

    cout << "9. Radix Sort" << endl;

    int choice;

    cout << "Enter your choice (1-9): ";

    cin >> choice;

    if (choice < 1 || choice > 9)

    {

        cout << "Invalid choice. Exiting program." << endl;

        return 1;

    }

    SDL\_SetRenderDrawColor(renderer, 255, 165, 0, 255);

    for (int i = 0; i < SIZE; i++)

    {

        SDL\_Rect rect = { i \* WIDTH / SIZE, HEIGHT - arr[i], WIDTH / SIZE, HEIGHT };

        SDL\_RenderFillRect(renderer, &rect);

        SDL\_RenderPresent(renderer);

    }

    time(&start);

    // Call the appropriate sorting function based on user choice

    switch (choice)

    {

    case 1:

        SelectionSort(arr);

        break;

    case 2:

        BubbleSort(arr);

        break;

    case 3:

        InsertionSort(arr);

        break;

    case 4:

        MergeSort(arr, 0, SIZE - 1);

        break;

    case 5:

        QuickSort(arr, 0, SIZE - 1);

        break;

    case 6:

        HeapSort(arr);

        break;

    case 7:

        CountingSort(arr);

        break;

    case 8:

        EvenOddSort(arr);

        break;

    case 9:

        RadixSort(arr);

        break;

    default:

        cout << "Invalid choice." << endl;

    }

    time(&end);

    double time\_taken = double(end - start);

    cout << "Array size: " << SIZE << endl;

    cout << "Time taken by program is: " << fixed << time\_taken << setprecision(5) << " sec " << endl;

    while (!quit)

    {

        quit = processEvent();

    }

    // Destroy renderer

    SDL\_DestroyRenderer(renderer);

    // Destroy window

    SDL\_DestroyWindow(window);

    // Quit SDL subsystems

    SDL\_Quit();

    return 0;

}

void GenerateArray(int\* arr)

{

    srand(time(NULL));

    for (int i = 0; i < SIZE; i++)

    {

        arr[i] = 1 + rand() % HEIGHT;

    }

}

void PrintArray(int\* arr)

{

    for (int i = 0; i < SIZE; i++)

    {

        cout << arr[i] << ' ';

    }

}

void draw(int\* arr, int i, int j)

{

    SDL\_Rect rect;

    SDL\_SetRenderDrawColor(renderer, 0, 0, 255, 255);

    rect = { j \* WIDTH / SIZE, HEIGHT - arr[j], WIDTH / SIZE, HEIGHT };

    SDL\_RenderFillRect(renderer, &rect);

    SDL\_RenderPresent(renderer);

    SDL\_SetRenderDrawColor(renderer, 0, 0, 255, 255);

    rect = { i \* WIDTH / SIZE, HEIGHT - arr[i], WIDTH / SIZE, HEIGHT };

    SDL\_RenderFillRect(renderer, &rect);

    SDL\_RenderPresent(renderer);

    SDL\_SetRenderDrawColor(renderer, 255, 255, 255, 255);

    rect = { j \* WIDTH / SIZE, HEIGHT - arr[j], WIDTH / SIZE, HEIGHT };

    SDL\_RenderFillRect(renderer, &rect);

    SDL\_SetRenderDrawColor(renderer, 255, 165, 0, 255);

    rect = { j \* WIDTH / SIZE, HEIGHT - arr[i], WIDTH / SIZE, HEIGHT };

    SDL\_RenderFillRect(renderer, &rect);

    SDL\_RenderPresent(renderer);

    SDL\_SetRenderDrawColor(renderer, 255, 255, 255, 255);

    rect = { i \* WIDTH / SIZE, HEIGHT - arr[i], WIDTH / SIZE, HEIGHT };

    SDL\_RenderFillRect(renderer, &rect);

    SDL\_SetRenderDrawColor(renderer, 0, 0, 255, 255);

    rect = { i \* WIDTH / SIZE, HEIGHT - arr[j], WIDTH / SIZE, HEIGHT };

    SDL\_RenderFillRect(renderer, &rect);

    SDL\_RenderPresent(renderer);

}

void SelectionSort(int\* arr)

{

    int i, j, index;

    for (i = 0; i < SIZE - 1; i++)

    {

        index = i;

        for (j = i + 1; j < SIZE; j++)

        {

            if (arr[j] < arr[index]) {

                index = j;

            }

        }

        draw(arr, i, index);

        Swap(arr, i, index);

        SDL\_Delay(5);

    }

    SDL\_Rect rect;

    SDL\_SetRenderDrawColor(renderer, 0, 0, 255, 255);

    rect = { (SIZE - 1) \* (WIDTH / SIZE), HEIGHT - arr[SIZE - 1], WIDTH / SIZE, HEIGHT };

    SDL\_RenderFillRect(renderer, &rect);

    SDL\_RenderPresent(renderer);

}

void BubbleSort(int\* arr)

{

    for (int i = 0; i < SIZE - 1; i++)

    {

        for (int j = 0; j < SIZE - i - 1; j++)

        {

            if (arr[j] > arr[j + 1])

            {

                draw(arr, j, j + 1);

                Swap(arr, j, j + 1);

                SDL\_Delay(5);

            }

        }

    }

}

void InsertionSort(int\* arr)

{

    int i, key, j;

    for (i = 1; i < SIZE; i++)

    {

        key = arr[i];

        j = i - 1;

        while (j >= 0 && arr[j] > key)

        {

            draw(arr, j, j + 1);

            arr[j + 1] = arr[j];

            j = j - 1;

            SDL\_Delay(5);

        }

        arr[j + 1] = key;

    }

}

void MergeSort(int\* arr, int l, int r)

{

    if (l < r)

    {

        int m = l + (r - l) / 2;

        MergeSort(arr, l, m);

        MergeSort(arr, m + 1, r);

        Merge(arr, l, m, r);

    }

}

// ... (Your existing includes and constants)

void Merge(int\* arr, int l, int m, int r)

{

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    int i = 0;

    int j = 0;

    int k = l;

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

void QuickSort(int\* arr, int low, int high)

{

    if (low < high)

    {

        int pi = Partition(arr, low, high);

        QuickSort(arr, low, pi - 1);

        QuickSort(arr, pi + 1, high);

    }

}

void Heapify(int\* arr, int n, int i)

{

    int largest = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

    if (left < n && arr[left] > arr[largest])

        largest = left;

    if (right < n && arr[right] > arr[largest])

        largest = right;

    if (largest != i)

    {

        draw(arr, i, largest);

        Swap(arr, i, largest);

        SDL\_Delay(5);

        Heapify(arr, n, largest);

    }

}

void HeapSort(int\* arr)

{

    for (int i = SIZE / 2 - 1; i >= 0; i--)

        Heapify(arr, SIZE, i);

    for (int i = SIZE - 1; i > 0; i--)

    {

        draw(arr, 0, i);

        Swap(arr, 0, i);

        SDL\_Delay(5);

        Heapify(arr, i, 0);

    }

}

void CountingSort(int\* arr)

{

    int maxVal = \*max\_element(arr, arr + SIZE);

    int minVal = \*min\_element(arr, arr + SIZE);

    int range = maxVal - minVal + 1;

    int\* count = new int[range]();

    int\* output = new int[SIZE];

    for (int i = 0; i < SIZE; i++)

        count[arr[i] - minVal]++;

    for (int i = 1; i < range; i++)

        count[i] += count[i - 1];

    for (int i = SIZE - 1; i >= 0; i--) {

        draw(arr, count[arr[i] - minVal] - 1, i);

        output[count[arr[i] - minVal] - 1] = arr[i];

        count[arr[i] - minVal]--;

        SDL\_Delay(5);

    }

    for (int i = 0; i < SIZE; i++) {

        arr[i] = output[i];

        draw(arr, i, i);

        SDL\_Delay(5);

    }

    delete[] count;

    delete[] output;

}

void EvenOddSort(int\* arr)

{

    bool sorted = false;

    while (!sorted)

    {

        sorted = true;

        for (int i = 1; i < SIZE - 1; i += 2)

        {

            if (arr[i] > arr[i + 1])

            {

                draw(arr, i, i + 1);

                Swap(arr, i, i + 1);

                SDL\_Delay(5);

                sorted = false;

            }

        }

        for (int i = 0; i < SIZE - 1; i += 2)

        {

            if (arr[i] > arr[i + 1])

            {

                draw(arr, i, i + 1);

                Swap(arr, i, i + 1);

                SDL\_Delay(5);

                sorted = false;

            }

        }

    }

}

void RadixSort(int\* arr)

{

    int maxVal = \*max\_element(arr, arr + SIZE);

    for (int exp = 1; maxVal / exp > 0; exp \*= 10)

    {

        int output[SIZE];

        int count[10] = { 0 };

        for (int i = 0; i < SIZE; i++)

            count[(arr[i] / exp) % 10]++;

        for (int i = 1; i < 10; i++)

            count[i] += count[i - 1];

        for (int i = SIZE - 1; i >= 0; i--)

        {

            draw(arr, count[(arr[i] / exp) % 10] - 1, i);

            output[count[(arr[i] / exp) % 10] - 1] = arr[i];

            count[(arr[i] / exp) % 10]--;

            SDL\_Delay(5);

        }

        for (int i = 0; i < SIZE; i++)

        {

            arr[i] = output[i];

            draw(arr, i, i);

            SDL\_Delay(5);

        }

    }

}

int Partition(int\* arr, int low, int high)

{

    int pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)

    {

        if (arr[j] < pivot)

        {

            i++;

            Swap(arr, i, j);

            draw(arr, i, j);

            SDL\_Delay(5);

        }

    }

    Swap(arr, i + 1, high);

    draw(arr, i + 1, high);

    SDL\_Delay(5);

    return (i + 1);

}

void Swap(int\* arr, int i, int j)

{

    int temp = arr[i];

    arr[i] = arr[j];

    arr[j] = temp;

}

bool processEvent()

{

    while (SDL\_PollEvent(&event))

    {

        switch (event.type)

        {

        case SDL\_WINDOWEVENT\_CLOSE:

            if (window)

            {

                SDL\_DestroyWindow(window);

                window = NULL;

                quit = true;

            }

            break;

        case SDL\_KEYDOWN:

            switch (event.key.keysym.sym)

            {

            case SDLK\_ESCAPE:

                quit = true;

                break;

            }

            break;

        case SDL\_QUIT:

            quit = true;

            break;

        }

    }

    return quit;

}

void setup()

{

    window = SDL\_CreateWindow("Sorting Visualizer", SDL\_WINDOWPOS\_CENTERED, SDL\_WINDOWPOS\_CENTERED, WIDTH, HEIGHT, 0);

    if (window == NULL)

    {

        cout << "Window could not be created! SDL\_Error" << endl << SDL\_GetError();

    }

    else

    {

        renderer = SDL\_CreateRenderer(window, -1, 0);

        SDL\_SetRenderDrawColor(renderer, 255, 255, 255, 255);

        SDL\_RenderClear(renderer);

        SDL\_UpdateWindowSurface(window);

    }

}

void performSorting(int\* arr, int choice)

{

    switch (choice)

    {

    case 1:

        SelectionSort(arr);

        break;

    case 2:

        BubbleSort(arr);

        break;

    case 3:

        InsertionSort(arr);

        break;

    case 4:

        MergeSort(arr, 0, SIZE - 1);

        break;

    case 5:

        QuickSort(arr, 0, SIZE - 1);

        break;

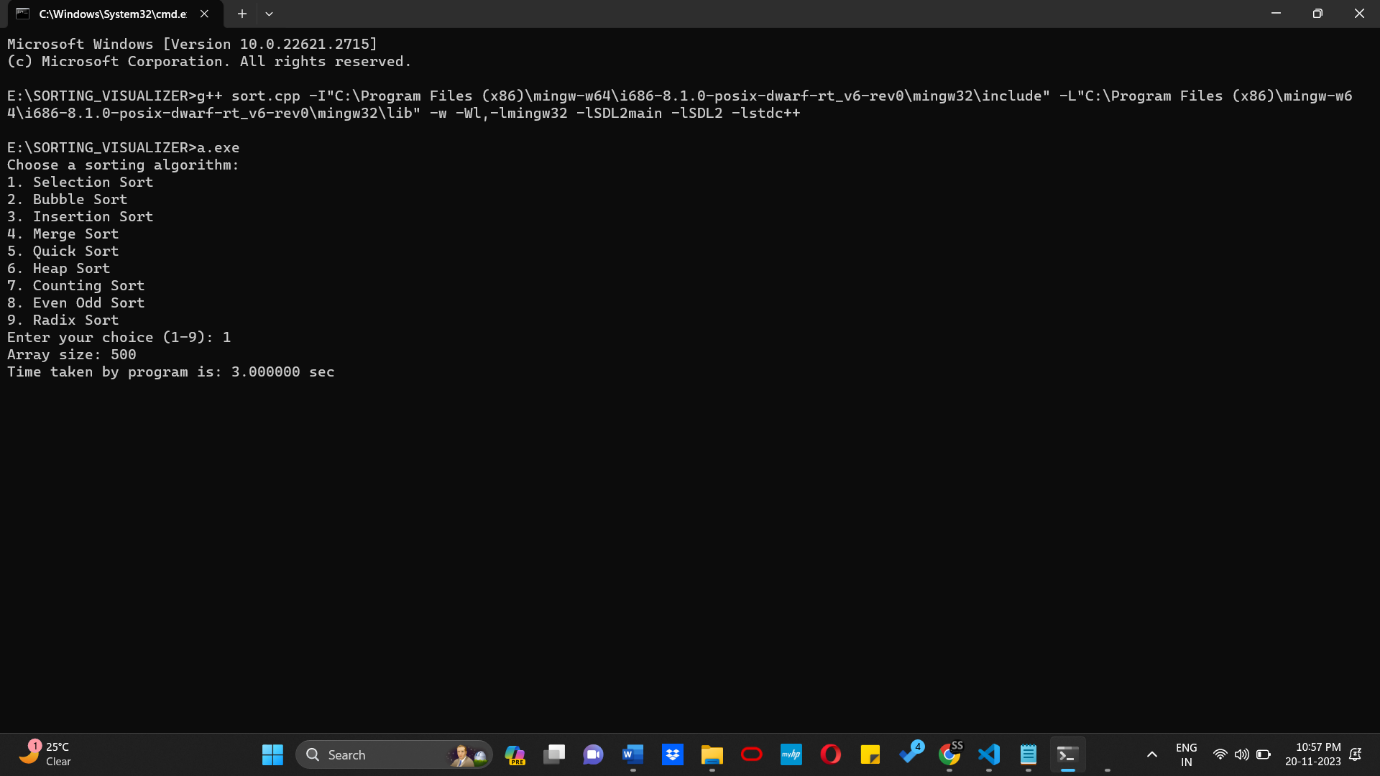
    default:

        cout << "Invalid choice." << endl;

    }

}

TERMINAL :



These commands are related to compiling and running a C++ program using the GNU Compiler Collection (GCC) on a Windows system. Let's break down each part of the commands:

g++ sort.cpp -I"C:\Program Files (x86)\mingw-w64\i686-8.1.0-posix-dwarf-rt\_v6-rev0\mingw32\include" -L"C:\Program Files (x86)\mingw-w64\i686-8.1.0-posix-dwarf-rt\_v6-rev0\mingw32\lib" -w -Wl,-lmingw32 -lSDL2main -lSDL2 -lstdc++

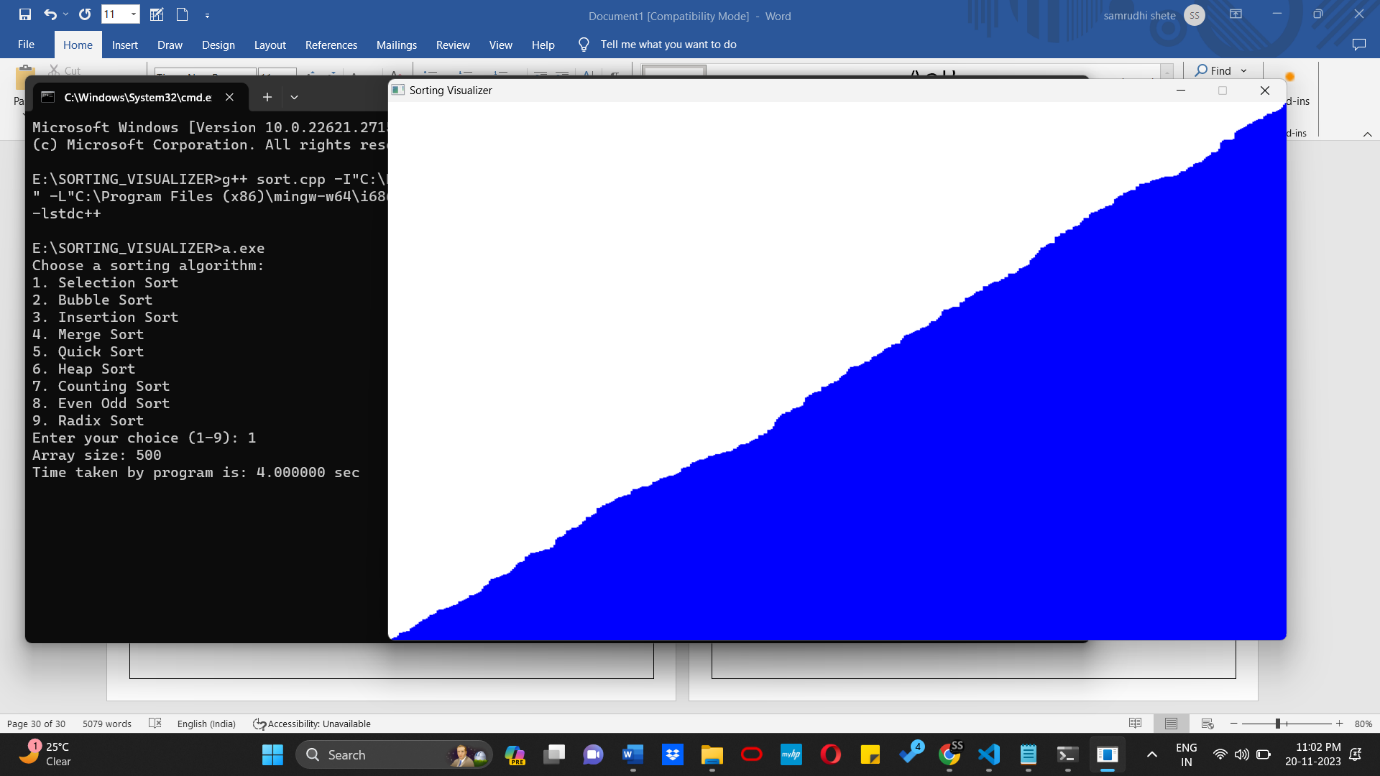
* g++: This is the command for invoking the GNU C++ compiler.
* sort.cpp: This is the source code file that you want to compile.
* -I"...": Specifies the directory to search for header files. In this case, it includes the path to the MinGW header files.
* -L"...": Specifies the directory to search for library files. It includes the path to the MinGW libraries.
* -w: Suppresses warning messages during compilation. It makes the compiler not display warning messages.
* -Wl,-lmingw32 -lSDL2main -lSDL2 -lstdc++: These are linker options specifying additional libraries to link against.
* -lmingw32: Links against the MinGW runtime library.
* -lSDL2main: Links against the SDL2 main library.
* -lSDL2: Links against the SDL2 library.
* -lstdc++: Links against the C++ standard library.

E:\SORTING\_VISUALIZER>a.exe

* E:\SORTING\_VISUALIZER>: Specifies the current working directory where the compiled executable (a.exe) is located.
* a.exe: This is the compiled executable file generated by the compilation process. Running this command executes the compiled program.

In summary, the first command is used to compile the C++ source code (sort.cpp) using the GNU C++ compiler (g++) with specific include and library paths, as well as linker options for SDL2 and other libraries. The second command runs the compiled executable (a.exe).

**OUTPUT:**



TIME AND SPACE COMPLEXITY ANALYSIS: