**SORTING VISUALIZER**

A sorting visualizer is a tool or application designed to visually demonstrate the process of sorting algorithms in action. It provides an interactive or graphical representation of how different sorting algorithms sort an array of data. The primary goal of a sorting visualizer is to help users understand and compare how various sorting algorithms work by showing the step-by-step changes made to the array.

**Overview**

This C++ program demonstrates various sorting algorithms with a visual representation of the sorting process. The user can choose from different sorting algorithms to apply to an array of random integers, with each step of the algorithm being visually displayed**Components**

1. **Includes and Namespace**
   * #include <iostream>: For input and output operations.
   * #include <vector>: To use the std::vector container.
   * #include <cstdlib>: For the rand() function.
   * #include <ctime>: For the time() function to seed the random number generator.
   * #include <thread> and #include <chrono>: To introduce delays for visual effects.
2. **Utility Functions**
   * **printArray(const vector<int>& arr)**
     + Prints the array in a graphical format where each integer is represented by a line of # characters.
     + Ends the output with a separator line for clarity.
   * **generateRandomArray(int size)**
     + Generates an array of random integers between 1 and 20.
     + Returns a vector<int> containing these integers.
   * **pause(int seconds)**
     + Pauses execution for a specified number of seconds using std::this\_thread::sleep\_for.
3. **Sorting Algorithms**

Each sorting algorithm is implemented to visually display its process by calling printArray() after each significant operation. The delay is introduced by pause().

* + **bubbleSort(vector<int>& arr)**
    - Implements the Bubble Sort algorithm.
    - Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
    - After each swap, the current state of the array is printed.
  + **insertionSort(vector<int>& arr)**
    - Implements the Insertion Sort algorithm.
    - Builds the sorted array one item at a time by removing each item from the input data and finding the location it belongs in the sorted list.
    - After each insertion, the array is printed.
  + **selectionSort(vector<int>& arr)**
    - Implements the Selection Sort algorithm.
    - Divides the array into a sorted and an unsorted region. Continuously selects the smallest element from the unsorted region and moves it to the end of the sorted region.
    - The array is printed after each swap.
  + **mergeSort(vector<int>& arr, int left, int right)**
    - Implements the Merge Sort algorithm using a divide-and-conquer strategy.
    - Divides the array into two halves, recursively sorts them, and then merges the sorted halves.
    - The array is printed after each merge operation.
  + **quickSort(vector<int>& arr, int low, int high)**
    - Implements the Quick Sort algorithm.
    - Selects a pivot element and partitions the array around the pivot so that elements less than the pivot come before it and elements greater come after it.
    - Recursively applies the same process to the sub-arrays. The array is printed after partitioning.
  + **heapify(vector<int>& arr, int n, int i)**
    - A helper function used in Heap Sort.
    - Ensures the heap property is maintained at a given index i within the heap of size n.
  + **heapSort(vector<int>& arr)**
    - Implements the Heap Sort algorithm.
    - Builds a max heap from the array, then repeatedly extracts the maximum element from the heap and places it in its correct position.
    - The array is printed after each extraction.

1. **Main Function**
   * Provides a menu-driven interface for the user to interact with the program.
   * The user can choose to generate a random array or apply one of the sorting algorithms to the array.
   * The sorting algorithms are executed on the currently active array, and results are visually displayed after each step.

**Code Execution Flow**

1. **Initialization**
   * srand(static\_cast<unsigned int>(time(0))) seeds the random number generator for generating random arrays.
2. **Menu Loop**
   * Displays the main menu and processes user choices.
   * Depending on the choice, it either generates a new random array, applies a sorting algorithm, or exits the program.
3. **Sorting Execution**
   * Each sorting algorithm operates on the currently active array, and its progress is visually displayed by printing the array after each significant step.

**Notes**

* **Visual Effect**: The use of printArray() and pause() functions provides a clear visual understanding of the sorting process. However, the visualization might be slow due to the use of pause() which introduces a 1-second delay. This delay can be adjusted based on preference.
* **Array Size**: Since the maximum value of array elements is 20, the visualization will be quite compact and easy to understand.
* **Edge Cases**: The program does not handle edge cases such as sorting an empty array or very small arrays explicitly.