Sorting Algorithm Performance Analysis

Program Description:-

In my program I used three sorting algorithms:-

Bubble Sort, Insertion Sort, and Selection Sort. On all three sorting algorithms, I ran three data sets: Random array, sorted array, and inversely sorted array. It measures two performance metrics: the number of comparisons and the execution time in microseconds, then they are compared against each other to see which algorithm uses less comparisons and which is faster. Each algorithm is tested on input sizes from 1 to 30 elements, and results are printed to the console and then put into tables in excel which are visualized by charts.

Algorithms Used:-

Bubble Sort:

Repeatedly compares elements that are next to each other and swap them if they are out of order. With each pass, the largest element moves up to its correct position. The process stops once no swaps are needed.

• Best for small datasets or when the array is nearly sorted.

Insertion Sort:

Builds a sorted array, one element at a time, starting from the second element, moving it to its correct position. Each new element is compared with already sorted ones and inserted into its correct place. Performs efficiently on small or nearly sorted datasets.

Selection Sort:

Finds the smallest element from the unsorted section and places it in the correct sorted position. This continues until all elements are sorted. It’s simple and consistent in performance. A note is that it always does the same number of comparisons no matter if the array is random, sorted or inversely sorted.

Function Breakdown:-

|  |  |
| --- | --- |
| Function | Purpose |
| bubbleSort(int arr[], int n) | Sorts the array using Bubble Sort and counts comparisons. |
| insertionSort(int arr[], int n) | Sorts the array using Insertion Sort and counts comparisons. |
| selectionSort(int arr[], int n) | Sorts the array using Selection Sort and counts comparisons. |
| printArray(int arr[], int size) | Prints the array before and after sorting. |
| test\_comparisons() | Runs all sorting algorithms on different array types, measuring time and comparisons. |
| main() | Calls test\_comparisons() to execute the program. |

Code Explained:-



This is the bubble sort function, which keeps passing through the array, comparing elements that are next to each other, then it swaps them if they are in the wrong order. Also this function will stop early if it detects that the array is already sorted, this will happen if no elements in the array were swapped during a pass of the array. The outer loop controls the number of passes through the array. The inner loop performs pair comparisons and swaps when necessary. Comparisons is a counter for the total number of comparisons.

A computer screen shot of a program code

AI-generated content may be incorrect.

This is the insertion sort function, which builds a sorted portion of the array one element at a time. Starting from the second element, each element (key) is compared to the elements before it. The algorithm shifts larger elements one position to the right until it finds the correct position for the key. The while loop does both the counter for comparisons and the moving of the elements.

A computer screen with text

AI-generated content may be incorrect.

This is the selection sort function, which sorts the array by repeatedly selecting the smallest element from the unsorted portion and placing it in its correct position. The outer for loop goes through each position (i) in the array, which is then stored in (min\_index). Then the inner loop compares it with the rest of the unsorted elements, while incrementing the comparisons counter. When it finds a smaller element is found, (min\_index) is updated. After finishing the loop, the smallest element is swapped with the element in position (i), placing it in its correct position.

A computer screen shot of text

AI-generated content may be incorrect.

The printing function simply prints the array, before and after sorting it.

A computer screen shot of a number

AI-generated content may be incorrect.

This is inside the test\_comparisons() function. It is my three base arrays, which I loop on to get all the variation (sizes 1-30) that I need.

A computer screen shot of text

AI-generated content may be incorrect.

This section determines which type of array will be used, random, sorted, or inversely sorted. A for loop runs three times, each time setting (baseArray) to one of the three array types. Depending on the value of type (0-2), it prints the corresponding array type (“Random Arrays”, “Sorted Arrays”, or “Inverse Sorted Arrays”) and assigns the matching base array for testing.

A computer screen shot of code

AI-generated content may be incorrect.

This loop runs tests for array sizes from 1 to 30. For each size, it creates a temporary array arr[] and fills it with elements from the selected base array. The original array is printed, then the time before and after sorting is recorded using (high\_resolution\_clock). The duration (in microseconds) is then calculated and printed to show how long the algorithm took for that array size. (The image is taken from bubble sort. The code is the same except the other algorithms are called instead.)

A black background with white and blue text

AI-generated content may be incorrect.

After sorting, the program prints the final sorted version of the array using the printArray() function.

Complexity Analysis:-

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Insertion Sort | O(n) | O(n²) | O(n²) |
| Selection Sort | O(n²) | O(n²) | O(n²) |

All three algorithms are implemented iteratively. If implemented recursively, the time complexity would remain the same, however the space complexity would be larger.

Excel Results:-

Two Excel files were created: First one is for the number of comparisons (not the number of swaps, meaning that if elements were compared but not swapped it would still count for the counter). The second one is for the execution time for every array, in microseconds. I didn’t calculate in seconds or milliseconds, because the array sizes are very small so the differences are extremely small so the results would be harder to read, and the charts would be nearly unreadable. Using microseconds, the results are easier on the eye, and the charts are clear. Each file includes all three algorithms and array types, and line charts to visualize each algorithm’s behavior and performance for each array type as well as compared against each other.

Comparisons Charts:

For bubble sorting algorithm, the sorted arrays made much less comparisons compared to the inversely sorted and random arrays. Inversely sorted arrays took the most comparisons.

For the insertion sorting algorithm, the sorted arrays also made much less comparisons, but the difference between the inversely sorted and random arrays was more noticeable. The inversely sorted arrays also made the most comparisons.

For the Selection sorting algorithm, the same number of comparisons was made for all three types of arrays.

For the random arrays, insertion sort had the least comparisons, followed by bubble sort and the selection sort had the most.

For the sorted arrays, insertion sort and bubble sort had the least number of comparisons, while selection sort had the most.

For the inversely sorted arrays, all three sorting algorithms had the same number of comparisons.

Execution time Charts:

For each algorithm separately (including 3 types of arrays):

Algorithms compared against each other (based on array type):

Conclusion:-

• Bubble Sort is the slowest for large or reversed arrays but efficient when nearly sorted.  
• Insertion Sort performs best for small and nearly sorted arrays.  
• Selection Sort is predictable but consistently slower overall.

• For the random arrays, insertion sort had the least number of comparisons while selection sort had the most. Bubble sort was in the middle.  
• In the sorted arrays Bubble and Insertion sort had the same number of comparisons while selection sort had a lot more.  
• For the inversely sorted arrays all three algorithms had the exact same number of comparisons.

The results for execution time were all very similar and the charts do not look like they should, but that is because our arrays are very small (max size of 30), so execution times are extremely fast. If we had much larger arrays (size 500) the charts would be more accurate as the execution time would differ a lot.

GitHub Link:-