1. **Abstract:**

We are given an integer vector represented by int\* an array of integers and we are interested in sorting the arrays of integer vectors according to the pre-defined notion of vector length which is given below.



The first given algorithm is Naïve Insertion Sort. We must improve and implement the Improved Insertion Sort algorithm and the Merge Sort algorithm for the given vector arrays of different sizes and dimensions and note the time taken by each technique for the Random order, Sorted order and Inverse Sorted order array.

1. **Result:**

**Testing Sorting Algorithms:**

Three different types of sorting algorithms were implemented and the time complexity of each one of them was measured in order to complete the given experiment. The three algorithms are as follows:

* Naive (and inefficient) Insertion Sort
* Modified Insertion Sort
* Merge Sort

**Naive Insertion Sort:**

In this type of sorting algorithm, we start from the second element and compare it with first element. Then we perform this operation for all the remaining elements. The rows are present in a multidimensional way. For each row, we are calculating the length of the corresponding vector which is compared with the length of the previous vector. We are calculating the length of vector each time when the loop for the vector is being traversed. Calculation of length and sorting of the vector takes place at the same time.

**General Algorithm:**

* Step 1: Declare an integer variable to store value of key.
* Step 2: Declare a For loop where j starts from 1 till the length of vector array.
* Step 3: Set the value of key as A[j].
* Step 4: Set the value of i as j - 1.
* Step 5: In the while loop, check whether i >= 1 && function is called twice to calculate and compare the length of vector array for A[i] element and for key.
* Step 6: Till the while condition satisfies, A[i+1] = A[i] and i is decrement by 1.
* Step 7: After while condition fails, key is assigned to A[i+1] and j is incremented for next iteration.
* Step 8: Exit.

**Improved Insertion Sort:**

In this type of sorting algorithm, we start from the second element and compare it with first element. Then we perform this operation for all the remaining elements. The rows are present in a multidimensional way. We are calculating the length of the vectors at the beginning of the program before sorting have been initiated. As the length of the vectors is calculated at the beginning and stored in an array, the array of vectors is sorted on the basis of the length of the vector. The time complexity in this type of improved insertion sort is changed to O(n2).

**General Algorithm:**

* Step 1: Declare an integer variable key\_value to store value of key and an integer pointer comp\_pro\_len pointing to an array to store the address of the length of all the vector array.
* Step 2: Declare a For loop where j starts from 0 till the length of vector array to calculate and store address of length of each vector array into an array comp\_len\_arr.
* Step 3: Declare a For loop where j starts from 1 till the length of vector array.
* Step 4: Set the value of key as A[j] and value of key\_value as comp\_len\_arr[j].
* Step 5: Set the value of i as j - 1.
* Step 6: In the while loop, check whether i >= 1 && comp\_pre\_len[i] > key\_value which contains the length of vector arrays.
* Step 7: Till the while condition satisfies, A[i+1] = A[i], comp\_len\_arr[i+1] = comp\_len\_arr[i] and i is decrement by 1.
* Step 8: After while condition fails, key is assigned to A[i+1], key\_value is assigned to comp\_len\_arr[i+1] and j is incremented for next iteration.
* Step 9: Exit.

**Merge Sort:**

This type of sorting algorithm is based on divide-and-conquer and the worst-case running time in this algorithm is better than the Insertion Sort. The given array is split into two halves until we get one element in both the half. Recursion is used in order to divide the array. The halves are the merged into one making sure that the resulted array is sorted.

**General Algorithm:**

* Step 1: Declare an integer pointer comp\_pro\_len pointing to an array to store the address of the length of all the vector array.
* Step 2: Declare a For loop where j starts from 0 till the length of vector array to calculate and store address of length of each vector array into an array comp\_len\_arr.
* Step 3: Find the middle and call the function mergeDivide(A, comp\_len\_arr, n, p, r) to divide the vector array recursively.
* Step 4: Call the function mergeSort(A, comp\_len\_arr, n, p, midpoint, r) to sort and merge the sorted arrays
* Step 5: Declare pointers for the left array, length of left array, right array and length of right array.
* Step 6: Declare a for loop where i starts from 0 till length of the left array and assign left\_length[i] = comp\_len\_arr[p + i] and left\_array[i] = A[p + i].
* Step 7: Declare a for loop where j starts from 0 till length of the right array and assign right\_length[j] = comp\_len\_arr[midpoint + j + 1] and left\_array[j] = A[midpoint + j + 1].
* Step 9: Add the remaining elements of the left and right vector array if any.
* Step 10: Exit.

**Testing Vectors:**

1. **Naive Insertion Sort:**

* The runtime in ms for the Naive insertion sort is given below:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **n = 10** | | | **n = 25** | | | **n = 50** | | |
| **Size (m)** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** |
| 1000 | 16 | 0 | 38 | 95 | 0 | 117 | 144 | 0 | 191 |
| 2500 | 119 | 0 | 200 | 281 | 0 | 570 | 653 | 1 | 1177 |
| 5000 | 477 | 1 | 846 | 1133 | 1 | 2265 | 2442 | 2 | 4885 |
| 10000 | 981 | 1 | 1912 | 2551 | 1 | 5013 | 5390 | 2 | 10782 |
| 25000 | 6088 | 1 | 11891 | 16531 | 3 | 31862 | 37712 | 5 | 67290 |
| 50000 | 25774 | 2 | 45299 | 87703 | 6 | 84441 | 179628 | 10 | 188921 |
| 100000 | 157163 | 4 | 191535 | - | 12 | - | - | 26 | - |
| 250000 | - | 10 | - | - | 23 | - | - | 49 | - |

* The runtime taken by the Naive Insertion Sort is less when the size of the vector array is smaller or when the vector array is already sorted.
* As the size of the vector increases, the time taken by this strategy to sort the vector array increases.
* When the size is m = 250000, we can see time taken is not available and thus it is not efficient when it comes to sorting vector array of larger size.
* The runtime taken by random order vector increases as the size and the dimensions of the vector array increases.
* The Input size vs Runtime for the Random vector is given below:

Figure 1: Random Order Vector for Naive Insertion Sort

* When it comes to sorted vector array, naive insertion sort is efficient, and the runtime is less when the size of the vector array is small.
* But when the size of vector array increases, the runtime also increases.
* The Input size vs Runtime for the Sorted order vector is given below:

Figure 2: Sorted Order Vector for Naive Insertion Sort

* Runtime for inverse sorted vector is more as the while loop which is present inside is satisfied for every iteration, thus increasing the time complexity.
* As the size and dimension of the data set increases, the time taken by this sorting technique also increases a lot, thus it is not efficient when it comes to sorting large vector array or reverse order vector array
* The Input size vs Runtime for the Inverse sorted vector is given below:

Figure 3: Inverse Sorted Order Vector for Naive Insertion Sort

1. **Improved Insertion Sort:**

* The length of the vector array is precomputed in this strategy which makes this technique more efficient as compared to the Naïve Insertion Sort.
* The runtime in ms for the Improved insertion sort is given below:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **n = 10** | | | **n = 25** | | | **n = 50** | | |
| **Size (m)** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** |
| 1000 | 1 | 0 | 2 | 2 | 0 | 3 | 2 | 0 | 3 |
| 2500 | 8 | 0 | 16 | 9 | 0 | 17 | 8 | 0 | 17 |
| 5000 | 86 | 0 | 123 | 87 | 0 | 125 | 87 | 1 | 125 |
| 10000 | 190 | 1 | 344 | 198 | 1 | 342 | 199 | 1 | 349 |
| 25000 | 958 | 1 | 1897 | 965 | 3 | 1864 | 973 | 4 | 1870 |
| 50000 | 1821 | 1 | 3661 | 1821 | 3 | 3591 | 1819 | 7 | 3621 |
| 100000 | 12563 | 4 | 20905 | 12457 | 10 | 19739 | 12457 | 21 | 20901 |
| 250000 | 45132 | 7 | 90265 | 55524 | 81 | 109684 | 57595 | 101 | 109845 |

* Like the Naive Insertion Sort, the runtime taken by the Improved Insertion Sort is less when the size of the vector array is small or when the vector array is already sorted.
* As the size of the vector increases, the time taken by this strategy to sort the vector array also increases.
* As this is the improved technique, we are calculating the length of the vector array before we are beginning to sort the vectors.
* This decreases the runtime of the improved insertion sort by a big margin, and we can find the runtime when the size is m = 250000.
* It is still not efficient when it comes to sorting vector array of larger size as it compares one element with all the other elements present on the vector till it is sorted.
* The runtime taken by Random order vector increases as the size and the dimensions of the vector array increases.
* The Input size vs Runtime for the Random order vector of Improved Insertion Sort is given below:

Figure 4: Random Order Vector for Improved Insertion Sort

* In the sorted vector array, Improved Insertion Sort is efficient, and the runtime is less when the size of the vector array is of small size.
* As the vector array is already sorted, the while loop present is never satisfied which decreases the runtime of this strategy.
* As the size of vector array increases, the runtime also starts increasing.
* The Input size vs Runtime for the Sorted order vector is given below:

Figure 5: Sorted Order Vector for Improved Insertion Sort

* Runtime for Inverse sorted vector is more as the while loop which is present inside is satisfied for every iteration, thus increasing the time complexity.
* As the size and dimension of the dataset increases, the time taken by this sorting technique also increases a lot, thus it is not efficient when it comes to sorting large vector array or reverse order vector array
* The Input size vs Runtime for the Inverse sorted vector is given below:

Figure 6: Inverse Order Vector for Improved Insertion Sort

1. **Merge Sort:**

* The length of the vector array is precomputed in this strategy as well which makes this technique more efficient as compared to the Naïve Insertion Sort.
* The time taken by Merge sort is almost same in all three cases.
* Merge sort follows the divide and conquer strategy which is very efficient when it comes to sorting vector array of large size. This makes it more efficient than the Improved Insertion sort and the runtime is decreases by a big margin while sorting.
* The runtime in ms for the Merge sort is given below:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **n = 10** | | | **n = 25** | | | **n = 50** | | |
| **Size (m)** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** | **Random Vector** | **Sorted Vector** | **Inverse Sorted Vector** |
| 1000 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| 2500 | 2 | 2 | 2 | 3 | 3 | 3 | 5 | 5 | 5 |
| 5000 | 5 | 5 | 5 | 7 | 7 | 7 | 9 | 9 | 9 |
| 10000 | 11 | 10 | 11 | 15 | 14 | 15 | 37 | 36 | 37 |
| 25000 | 75 | 73 | 76 | 92 | 92 | 96 | 113 | 112 | 108 |
| 50000 | 119 | 115 | 114 | 143 | 140 | 141 | 190 | 177 | 177 |
| 100000 | 192 | 189 | 183 | 239 | 235 | 237 | 322 | 325 | 316 |
| 250000 | 420 | 407 | 400 | 544 | 546 | 549 | 650 | 745 | 743 |

* According to the runtime taken by Merge sort, we can see the time taken by Merge sort from small operations is more as compared to Insertion Sort technique. For e.g., Sorting an already sorted vector
* It goes through the whole process even if the vector is sorted.
* When it comes to larger vectors, time required by merge sort decreases by a large margin and it is most efficient among the three discussed techniques when it comes to sorting larger size vectors.
* It is faster than the Naïve Insertion Sort and the Improved Insertion Sort.
* The Input size vs Runtime for the Random sorted vector is given below:

Figure 7: Random Order Vector for Merge Sort

* Sorting the already sorted vector using the merge sort is not an efficient way to sort the vector. It repeats all the operations of divide-and-conquer thus increasing the runtime.
* The time taken to sort an already sorted vector of size m = 250000 and n = 10 is 407ms which is higher.

Figure 8: Sorted Order Vector for Merge Sort

* Merge sort is very efficient while sorting and Inverse order vector and larger size vectors.
* It uses the technique of recursion to divide the given vector array into sub arrays and then each sub array is considered as a single element.
* In the end, all the divided sub arrays are merged together and the vector is sorted.

Figure 9: Inverse Order Vector for Merge Sort

1. **Discussion:**

**Running time:**

* Insertion sort is considered as an efficient algorithm when the vector size is small. As it uses the while loop instead of for loop, the technique of sort does not perform extra iterations while sorting the vector array. Even if the passed vector array is sorted, insertion sort does not perform any extra iterations. But when we give a sorted array to an insertion sort or an improved insertion sort, it still follows its steps for the outer loop which performs size of vector array times thereby having efficient time complexity in terms of best-case scenario. The running time in case of improved insertion sort decreases as the length of the vector array are calculated before the sorting operation.
* In case of Merge Sort, the time complexity is same in for all the cases as it follows the divide-and-conquer rule on vector arrays and later again merges the divided array into one sorted array. Merge sort takes extra time when the given vector array is already sorted as it still divides the given vector array thus affecting the time complexity in terms of best-case scenario.

**Limitations:**

* As the insertion sort takes n-squared steps for every element which needs to be sorted, there is a problem when the size of vector array which is passed is large.
* Hence insertion sort and improved insertion sort can be used and can be efficient when it comes to sorting a vector array up to a limited length.
* Merge sort is not very useful when it comes to sorting vector arrays of small length. Due to its divide-and-conquer technique, time taken to sort small vector array is more as compared to insertion sort and improved insertion sort.
* Also, it requires extra space to store all the divided sub-arrays.
* Merge sort is not recommended to use when the given vector array is already sorted as it does the whole process of divide-and-conquer on the sorted vector thus increasing the time complexity.
* The factor of size(m) plays an important role in running time because as the size of the vector array increases, time taken by insertion sort and efficient insertion sort also increases and it is not advised to use them for large size of vector array.
* Merge sort is efficient on large value of m as it then divides the vector array of size m into sub-arrays in order to perform sorting and provide best result.

**Improvements:**

* The merge sort can be improved if there is a method of testing if the array is already in sorted order. We can reduce the running time by skipping the call to merge function.
* Using a different method to for sorting the small subarrays of the merge sort can improve the runtime by a small extent.

1. **Conclusion:**

The best way to choose a sorting algorithm depends on the given vector array. If the size is small or if the vector array is already sorted, then Insertion sort is the best strategy and more efficient as we saw the time taken by insertion sort is less. When the size of vector array is large or when the given vector array is in reverse order, then Merge sort is more efficient as it divides the vector array into different subarrays and performs the sorting operations. As the size of the vector array increases, merge sort becomes more and more efficient as compared to insertion sort as shown in the runtime table.