Sorting Algorithms Performance Analysis

Abstract

This report presents an analysis of the runtime performance of two sorting algorithms for integer vectors: Improved Insertion Sort and Merge Sort. The primary objective is to examine how these algorithms perform under different input scenarios, including random, sorted, and inversely sorted vectors of varying sizes and dimensions. We measure the runtime performance of each algorithm and discuss the theoretical expectations, providing insights into their behavior.

Introduction

In this assignment, we are provided with a vector of integers (int* an array of integers) and we are asked to sort the given integer vector arrays according to their vector length using the formula $\sum_{i=1}^{n} |v_i|$.

We implemented and evaluated three sorting algorithms for integer vectors: Naïve Insertion Sort, Improved Insertion Sort and, Merge Sort. These algorithms were tested on arrays of integer vectors, and the sorting was performed based on the lengths of the vectors.

Sorting Algorithms

1. Naïve Insertion Sort

   In the naïve implementation, we start the loop from the second element and compare it with the first element. As the loop moves forward, the integers are compared with the first element. For each row, we calculate the length of the vector when it is being traversed and compare it to the previous vector's length. Calculation of the length of the vector and comparing it to the previous vector happens at the same time.

   Input:

   - A: An array of integer vectors.
   - n: The dimension of the vectors.
   - l: The left index of the subarray to be sorted.
   - r: The right index of the subarray to be sorted.

   Algorithm:

   I. For each element j from l+1 to r (inclusive):
      a. Set key as the vector at index j.
      b. Initialize i as j+1.

   II. While i is greater than or equal to l and the length of the vector at index i (determined using ivector_length) is greater than the length of key (determined using ivector_length):
      a. Move the vector at index i one position ahead (to the right) by setting A[i+1] = A[i].
      b. Decrement i by 1.

   III. Place key in its correct position within the sorted order of vectors based on their lengths by setting A[i+1] = key.

   Output:

   The array A will be sorted in ascending order of vector lengths in the subarray from index `l` to `r`.

   This algorithm sorts the given subarray of vectors in-place based on their lengths using the insertion sort technique.

2. Improved Insertion Sort

   The improved insertion sort algorithm precomputes the lengths of vectors and sorts them based on these lengths.

   Input:

   - A: An array of integer vectors.

- n: The dimension of the vectors.
- l: The left index of the subarray to be sorted.
- r: The right index of the subarray to be sorted.

Algorithm:

I. Create an integer array lens of size r+1 to store the precomputed lengths of the vectors in the range from l to r.
II. For each element I from 0 to r (inclusive):
    a. Compute and store the length of the vector at index i in the lens array using the ivector_length function.
III. For each element j from 1 to r (inclusive):
    a. Set k as the vector at index j.
    b. Set k_value as the length of the vector k obtained from the lens array.
    c. Initialize i as j - 1.
IV. While i is greater than or equal to 1 and the length of the vector at index i (retrieved from the lens array) is greater than k_value:
    a. Move the vector at index i one position ahead (to the right) by setting A[i+1] = A[i].
    b. Update the lens array as well by setting lens[i+1] = lens[i].
    c. Decrement i by 1.
V. Place k in its correct position within the sorted order of vectors based on their precomputed lengths:
    a. Set A[i+1] = k to place k at the appropriate index in the A array.
    b. Update the lens array by setting lens[i+1] = k_value.

Output:

The array A will be sorted in ascending order of precomputed vector lengths in the subarray from index l to r.

This algorithm sorts the given subarray of vectors in-place based on their precomputed lengths using the insertion sort technique.

3. Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm that also precomputes the lengths of vectors before sorting.

- Algorithm for merge_sort():

  Input:

  - `A`: An array of integer vectors.
  - `n`: The dimension of the vectors.
  - `p`: The left index of the subarray to be sorted.
  - `r`: The right index of the subarray to be sorted.

  Algorithm:

  I. Create an integer array `len` of size `r + 1` to store the precomputed lengths of the vectors in `A`.
  II. Calculate the lengths of all vectors in `A` and store them in the `len` array:
      a. For `i` from `0` to `r` (inclusive):
      b. Set `len[i]` as the length of the vector at index `i` in `A` obtained from the `ivector_length` function.
  III. Call the `merge_divide` function to start the merge sort process:
      a. Call `merge_divide(A, len, n, p, r)`.
  IV. The array `A` will now be sorted in ascending order of precomputed vector lengths in the subarray from index `p` to `r`.

- Algorithm for merge_divide():

  Input:

  - `A`: An array of integer vectors.
  - `lengths`: An array storing the precomputed lengths of the vectors in `A`.

- `n`: The dimension of the vectors.
- `p`: The left index of the subarray to be sorted.
- `r`: The right index of the subarray to be sorted.

Algorithm:

1. If `p` is less than `r`, perform the following steps recursively; otherwise, exit the function.
2. Calculate the midpoint `midpoint` as the floor of `(p + r) / 2`.
3. Recursively call `merge_divide` on the left half of the subarray:
    a. Call `merge_divide(A, lengths, n, p, midpoint)`.
4. Recursively call `merge_divide` on the right half of the subarray:
    a. Call `merge_divide(A, lengths, n, midpoint + 1, r)`.
5. Merge the two sorted subarrays into a single sorted subarray:
    a. Call `merge(A, lengths, n, p, midpoint, r)`.

Output:

- The subarray of `A` will be sorted in ascending order of precomputed vector lengths.

- Algorithm for merge():

  Input:

  - `A`: An array of integer vectors.
  - `lengths`: An array storing the precomputed lengths of the vectors in `A`.
  - `n`: The dimension of the vectors.
  - `p`: The left index of the subarray.
  - `midpoint`: The index that separates the two subarrays to be merged.
  - `r`: The right index of the subarray.

  Algorithm

  1. Calculate the sizes of the left and right subarrays:
     - `left_size` = `midpoint - p + 1`
     - `right_size` = `r - midpoint`
  2. Create two temporary arrays:
     - `left_vectors`: An array of integer vectors of size `left_size`.
     - `right_vectors`: An array of integer vectors of size `right_size`.
     - `left_lengths`: An array of lengths corresponding to the vectors in `left_vectors`.
     - `right_lengths`: An array of lengths corresponding to the vectors in `right_vectors`.
  3. Copy the vectors and their lengths from the original array `A` to the temporary arrays:
     - For `i` from `0` to `left_size - 1`:
       - Set `left_vectors[i]` as the vector at index `p + i` in `A`.
       - Set `left_lengths[i]` as the length of `left_vectors[i]` obtained from the `lengths` array.
     - For `i` from `0` to `right_size - 1`:
       - Set `right_vectors[i]` as the vector at index `midpoint + 1 + i` in `A`.
       - Set `right_lengths[i]` as the length of `right_vectors[i]` obtained from the `lengths` array.
  4. Initialize variables for indices and iterators:
     - `i` for the left subarray (starts at `0`).
     - `j` for the right subarray (starts at `0`).
     - `k` for the merged subarray (starts at `p`).

  5. Set sentinel values in both `left_lengths` and `right_lengths` arrays to ensure that the subarrays are fully merged:
     - `left_lengths[left_size]` = `numeric_limits<int>::max()`
     - `right_lengths[right_size]` = `numeric_limits<int>::max()`
  6. Merge the two subarrays into a single sorted subarray in `A` based on the lengths of vectors:

- Repeat the following until either `i` reaches `left_size` or `j` reaches `right_size`:
  - If `left_lengths[i]` is less than or equal to `right_lengths[j]`, do the following:
    - Set `lengths[k]` to `left_lengths[i]`.
    - Set `A[k]` to `left_vectors[i]`.
    - Increment `i` and `k`.
  - Otherwise, do the following:
    - Set `lengths[k]` to `right_lengths[j]`.
    - Set `A[k]` to `right_vectors[j]`.
    - Increment `j` and `k`.

7. Copy any remaining elements from the left subarray to `A` (if any):
   - Repeat the following while `i` is less than `left_size`:
     - Set `lengths[k]` to `left_lengths[i]`.
     - Set `A[k]` to `left_vectors[i]`.
     - Increment `i` and `k`.
8. Copy any remaining elements from the right subarray to `A` (if any):
   - Repeat the following while `j` is less than `right_size`:
     - Set `lengths[k]` to `right_lengths[j]`.
     - Set `A[k]` to `right_vectors[j]`.
     - Increment `j` and `k`.
9. Free the memory used by the temporary arrays `left_vectors`, `right_vectors`, `left_lengths`, and `right_lengths`.

Output:

- The subarray of `A` will be sorted in ascending order of precomputed vector lengths.

This algorithm divides the array into smaller subarrays, sorts them individually based on vector lengths, and then merges them together to achieve the final sorted result using the merge sort technique.

Testing Vectors:

1. **Naïve Insertion Sort**

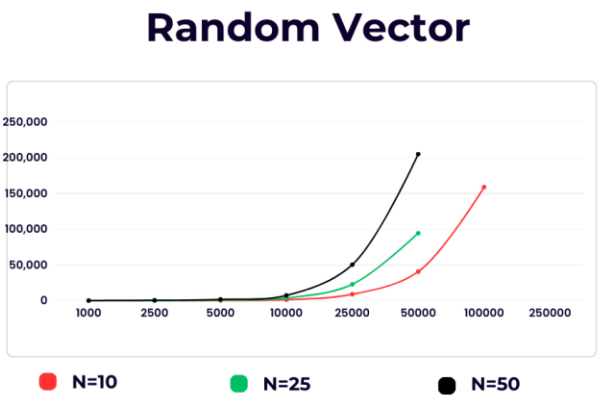
   The observations are below, where runtime is in ms.

	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	12	0	26	40	0	73	82	0	134
2500	82	0	179	240	1	467	488	1	884
5000	345	1	641	981	1	1863	1766	2	3671
10000	1364	1	2601	3745	1	7398	7429	3	14480
25000	8989	2	1605	22905	3	45178	50363	7	91671
50000	40610	3	66235	94483	6	177612	205061	13	
100000	158863	5	201535	-	12	-	-	27	-
250000	-	13	-	-	32	-	-	104	-

The Naive Insertion Sort exhibits shorter execution times with smaller vector arrays or when the vector array is already in a sorted state. However, as the vector size expands, the sorting duration using this method also grows.

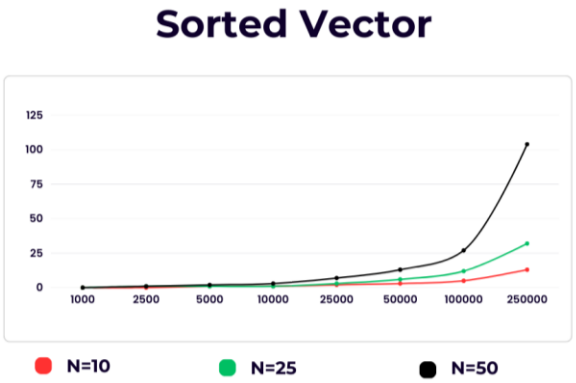
Notably, when the size reaches m = 250,000, no specific time measurement is available, indicating inefficiency in sorting larger vector arrays using this approach.

-The Input size vs Runtime for the Random vector is given below:



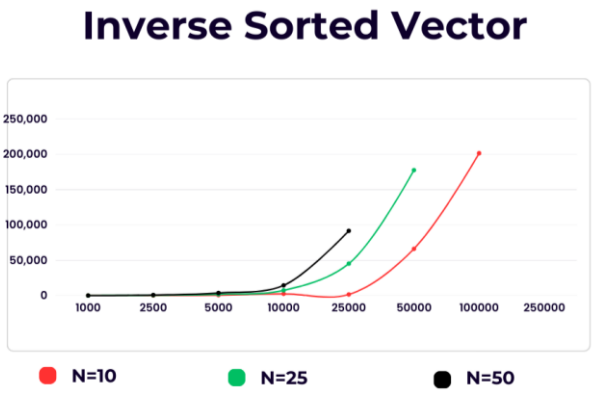
The runtime taken by random order vector increases as the size and the dimensions of the vector array increases.

-The Input size vs Runtime for the Sorted order vector is given below:



In cases involving sorted vector arrays, the naive insertion sort proves to be efficient, exhibiting shorter runtimes, especially with smaller vector sizes. However, as the vector array size grows, so does the runtime.

-The Input size vs Runtime for the Inverse sorted vector is given below:



In contrast, for inversely sorted vectors, the runtime is longer due to the repetitive satisfaction of the while loop condition, which increases time complexity.

As both the dataset size and dimensionality increase, the time required by this sorting approach also significantly escalates, rendering it inefficient for sorting large vector arrays or arrays in reverse order.

2. Improved Insertion Sort

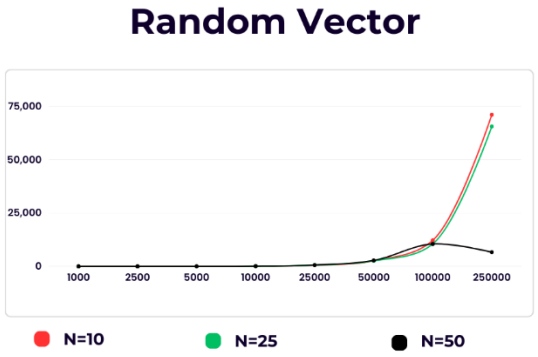
The observations are below, where runtime is in ms.

	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	2	1	0	3	2	0	2
2500	7	0	15	7	0	16	6	0	15
5000	25	0	47	35	0	55	30	1	55
10000	120	1	202	120	1	200	99	2	200
25000	570	2	1307	620	2	1286	642	4	1335
50000	2645	3	5456	2622	4	5546	2830	7	5258
100000	12200	5	22302	10510	8	21411	10524	15	21064
250000	70997	10	131801	65522	23	139779	6667	45	131008

Similar to the Naive Insertion Sort, the Improved Insertion Sort demonstrates shorter runtimes when handling small vector arrays or pre-sorted ones. However, as the vector size grows, the time required for this approach to sort the array also increases.

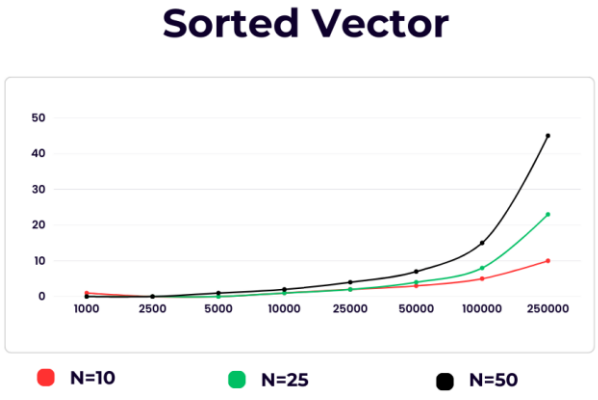
Being an enhanced technique, this method involves calculating the vector array's length before initiating the sorting process. This significantly reduces the runtime of the Improved Insertion Sort, and this improvement is evident even when the size reaches 250,000. Nonetheless, it still lacks efficiency when dealing with larger vector arrays, as it compares each element with all others in the vector until it's sorted.

-The Input size vs Runtime for the Random vector is given below:



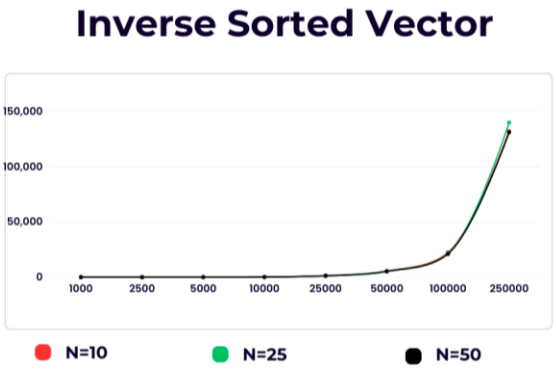
The runtime increases as both the size and dimensions of the vector array expand.

-The Input size vs Runtime for the Sorted order vector is given below:



In the context of sorted vector arrays, the Improved Insertion Sort proves efficient, resulting in shorter runtimes, especially with smaller vector sizes. Since the vector array is already sorted, the while loop condition is never met, further decreasing runtime.

-The Input size vs Runtime for the Inverse sorted vector is given below:



Conversely, the runtime is longer for inversely sorted vectors due to the repeated satisfaction of the while loop condition, which elevates time complexity.

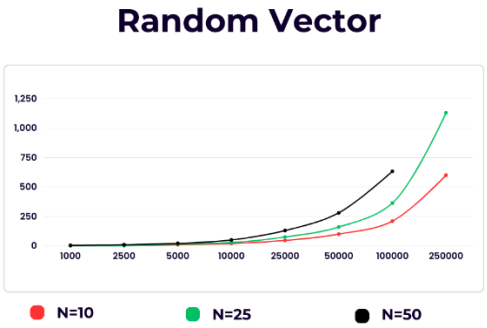
3. Merge Sort

The observations are below, where runtime is in ms

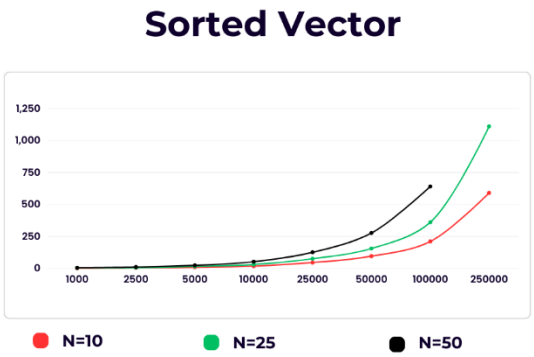
	n = 10			n = 25			n = 50		
Size (m)	**Random Vector**	**Sorted Vector**	**Inverse Sorted Vector**	**Random Vector**	**Sorted Vector**	**Inverse Sorted Vector**	**Random Vector**	**Sorted Vector**	**Inverse Sorted Vector**
1000	2	1	1	2	3	2	4	4	4
2500	4	4	4	2	5	6	10	10	10
5000	9	8	9	16	15	14	21	23	35
10000	20	17	16	28	30	28	50	52	46
25000	46	45	45	75	75	74	130	125	130
50000	100	95	95	160	155	160	280	276	281
100000	210	210	209	363	360	350	633	640	620
250000	600	590	600	1130	1110	1100	-	-	-

Similar to the Naïve Insertion Sort, this approach also calculates the vector array's length in advance, enhancing its efficiency. Merge sort exhibits nearly identical runtimes across all three scenarios. Employing a divide-and-conquer strategy, merge sort excels at sorting large vector arrays, surpassing the efficiency of the Improved Insertion Sort. Consequently, it significantly reduces runtime during sorting.

-The Input size vs Runtime for the Random vector is given below:

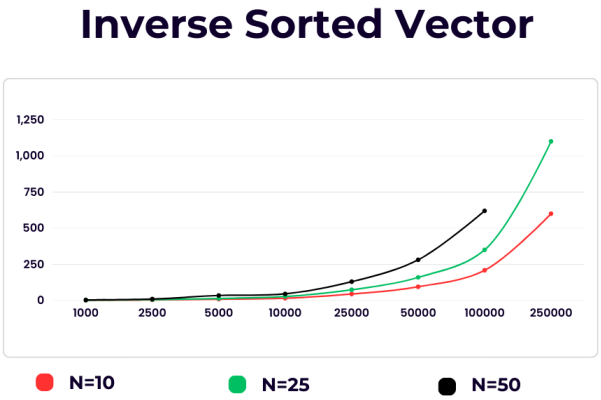


-The Input size vs Runtime for the Sorted order vector is given below:



Sorting an already sorted vector with merge sort is an inefficient approach for vector sorting. It redundantly performs all the divide-and-conquer operations, resulting in an increased runtime.

-The Input size vs Runtime for the Inverse sorted vector is given below:



Merge sort exhibits high efficiency when sorting inverse order vectors and larger-sized vectors.
It employs recursion to divide the provided vector array into subarrays, treating each subarray as a single element. Eventually, all the divided subarrays are merged together to yield a sorted vector.

Discussion:

Running Time:

- Insertion sort demonstrates efficiency when handling small vector sizes. It employs a while loop instead of a for loop, avoiding unnecessary iterations during the sorting process. Even when given an already sorted array, Insertion sort does not perform superfluous iterations. It retains its steps for the outer loop, which runs the size of the vector array, resulting in efficient time complexity in the best-case scenario. The running time of improved insertion sort decreases as it computes vector array lengths before sorting.
- Merge Sort maintains consistent time complexity across all cases by adhering to the divide-and-conquer principle on vector arrays and subsequently merging them into a single sorted array. However, Merge Sort consumes additional time when the input vector array is already sorted, as it continues to divide the array, impacting time complexity in the best-case scenario.

Limitations:

- Insertion sort takes n-squared steps for every element to be sorted, posing challenges with large input vector sizes. Therefore, Insertion sort and improved insertion sort are suitable and efficient for sorting relatively small vector arrays.

- Merge sort is less effective for sorting small vector arrays due to its divide-and-conquer technique, resulting in longer sorting times compared to insertion sort and improved insertion sort. It also requires extra memory to store the divided sub-arrays.
- Merge sort is not recommended for already sorted input vectors, as it unnecessarily applies the divide-and-conquer process, increasing time complexity.
- The vector array size (m) significantly influences the runtime. As the size increases, insertion sort and improved insertion sort also experience increased execution times and are not recommended for large vector arrays.
- Merge sort excels with larger values of m, as it can efficiently divide vector arrays of substantial size into sub-arrays for sorting, yielding better results.

Improvements:

- Enhancing merge sort can involve implementing a method to detect if the array is already sorted, reducing running time by avoiding unnecessary merge function calls.
- Exploring alternative approaches for sorting small subarrays within merge sort can slightly improve runtime efficiency.

Conclusion

The choice of a sorting algorithm should be based on the characteristics of the input vector array. For small-sized arrays or already sorted arrays, Insertion sort is the preferred and more efficient option, as evidenced by its shorter execution time. Conversely, for large-sized arrays or when the array is in reverse order, Merge sort is a more efficient choice. Merge sort divides the array into subarrays and conducts sorting operations, making it increasingly efficient as the array size grows, as indicated in the runtime table.