CS 590 – B HW1 Insertion Sort and Merge Sort Analysis   
Report by: Sanjeet Vinod Jain   
  
Testing Hardware used:

* CPU: Ryzen 5600 6 core 12 thread processor at 4.0Ghz
* RAM: 16gb 3200mhz
* Storage: NVME SSD 2000MB/s read-write speed
* Operating System: Windows 11 with Virtualized Linux Bash

Keywords used:

* Default = naïve insertion sort algorithm provided
* Optimized = improved insertion sort algorithm
* m = size of array and n = dimension of each vector in the array

Testing Parameters:

* Each sorting algorithm was run with varying inputs that includes random, sorted, and inverse sorted inputs of size m = 10000; 25000; 50000; 100000; 250000; 500000; 1000000; 2500000 and vector dimension n = 10; 25; 50.
* Each combination of m and n was allowed to run for a maximum of 6 mins after which it was automatically timed out.
* Additionally, a test with n = 100,1000 is also done for sorted vector list only for insertion sort.
* **All scripts to run the code were ran parallel to finish the project on time**

Limits / Threshold for each scenario for **Insertion Sort**:

* For example, default-inverted-10 means default sorting function running with an inverted vector as input with dimension 10. The second column denotes at which array input size the function started timing out and taking longer than 5 mins.
* nil indicates that the function in that situation ran for all input array sizes and didn’t timeout

|  |  |
| --- | --- |
| Default-insertion-sort-inverted-10 | 250000 |
| default-insertion-sort -inverted-25 | 100000 |
| default-insertion-sort -inverted-50 | 100000 |
| default-insertion-sort -random-10 | 250000 |
| default-insertion-sort -random-25 | 100000 |
| default-insertion-sort -random-50 | 100000 |
| default-insertion-sort -sorted-10 | nil |
| default-insertion-sort -sorted-25 | nil |
| default-insertion-sort -sorted-50 | nil |
| optimized-insertion-sort -inverted-10 | 1000000 |
| optimized-insertion-sort -inverted-25 | 1000000 |
| optimized-insertion-sort -inverted-50 | 1000000 |
| optimized-insertion-sort -random-10 | 1000000 |
| optimized-insertion-sort -random-25 | 1000000 |
| optimized-insertion-sort -random-50 | 1000000 |
| optimized-insertion-sort -sorted-10 | nil |
| optimized-insertion-sort -sorted-25 | nil |
| optimized-insertion-sort -sorted-50 | nil |

Insertion Sort:

# Default algorithm:

*insertionSort (array A)*

*for i = 1 to length(A) - 1*

*key ← A[i]*

*j ← i - 1*

*while j >= 0 and A[j] > x*

*A [j + 1] ← A[j]*

*j ← j - 1*

*end while*

*A [j + 1] ← key*

*end for*

# Proposed Optimized Algorithm:

*function binarySearch (Array, N, key)*

*L = 0*

*R = N*

*while L < R:*

*mid = (L + R)/2*

*if Array[mid] <= key:*

*L = mid + 1*

*else:*

*R = mid*

*return L*

*end function*

*function improve\_insertion\_sort (Array)*

*for i = 1 to length (Array) do:*

*key = Array[i]*

*pos = binarySearch (Array, key, 0, i-1)*

*j = i*

*while j > pos*

*Array[j] = Array[j-1]*

*j = j-1*

*Array[pos] = key*

*end for*

*end function*

Time Complexity Analysis:

## For Default

* Best Case: Ω (m)
  + Doing insertion sort on an already sorted array will only compare each element to the element before, needing m steps to sort the already sorted array of m elements.
* Worst Case: O(m2)
  + Doing insertion sort on a reverse-sorted array, it will insert each element at the beginning of the sorted subarray, making it the worst time complexity of insertion sort causing a total of m insertions done m times
* Average Case: θ (m2)
  + When the array elements are in random order, the average running time is θ (m2)

## For Optimized:

* Since we are using binary search to find the best index to place the element at instead of linear search, we reduce the number of comparisons for inserting one element from O(m) to O (log m).
* Worst Case: O(m2)
  + For inserting the i-th element in its correct position in the sorted array, finding the position(pos) will take O (log i) steps using binary search.
  + To insert the element, we need to shift all the elements from pos to i-1. This will take i steps in the worst case that is when the element needs to be at the start of the array when pos=0.
  + We make a total of m insertions, so the worst-case time complexity of binary insertion sort is

O(logi) + O(m\*i) or generalized as 0(m2) as the m2 is the dominating term

* Best Case: Ω (m log m)
  + The best case will be when the array is sorted and every element is in its place, which means we don’t have to shift any of the elements hence has runtime of O (1) for inserting the element.
  + But we are using binary search to find the position where we need to insert the element and if the element is already in its final place, binary search will take (log i) steps to complete. Thus, for the i-th element, we do (log i) operations.
  + so, for an m sized array it will be Ω (m log i) or Ω (m log m).
* Average Case: θ(m2)
  + When the array elements are in random order, we will need O (i /2) steps for inserting the i-th element, so the average time complexity of binary insertion sort is θ(m\*i/2) or as θ(m2)

Here n is the vector dimension of each element of the array of size m.

# Results (Tabulated Averages from Each Run)

## Default Algorithm:



## Optimized Algorithm:



Blank entries indicate the algorithm timed-out after 5 minutes of runtime.

# Graphical Representation of Default Vs Optimized Algorithms Runtimes

## Default

Observations:

* The default algorithm is following the predicted time complexity as predicted on (page 3) for all cases

## Optimized

Observations:

* The optimized algorithm is following the predicted time complexity as predicted on (page 3) for all cases

## Random Vector

Observations:

* The optimised algorithm is clearly better in terms of performance for the average case of randomized arrays with any dimension
* For all dimensions and input sizes till 1,000,000 the optimized algorithm worked efficiently with a similar trend across all inputs
* Whereas the default algorithm timed-out for different input sizes and dimension sizes showing its limitations in handling a larger sample set.
* Average Case:   
  The default algorithm is following the predicted time complexity curve of O(m2) and the optimized algorithm is following the predicted time complexity curve θ(m\*i/2)  
  (page 3)

## Inverted Vector

Observations:

* The optimised algorithm outperforms in terms of performance for the worst case of inverted arrays with any dimension with a clear distinction of the graphs following a
* For all dimensions and input sizes till 500,000 the optimized algorithm worked efficiently with a similar trend across all inputs
* Whereas the default algorithm timed-out for input sizes above 50,00 and dimension sizes showing its limitations in handling a larger sample set.
* The default algorithm is following the predicted time complexity curve of O(m2) and the optimized algorithm is following the predicted time complexity curve O(m\*i)   
  (page 3)

## Sorted Vector

Observations :

* The optimized algorithm has a fix trend for any input size and dimension size of sorted vectors whereas the default algorithm varies a lot depending on the input sample size.
* The optimized algorithm performs less efficiently on smaller input sizes due to the added operations of binary search even if the array is sorted.
* It is also seen that as the dimension size grows the optimized algorithm performs better than the default algorithm as shown for cases n=100 and n=1000. This is more likely due to the binary search reducing the number of comparisons for a larger set as compared to linear search used by the default algorithm.
* Thus, making the optimized insertion sort algorithm to work better on a much larger sample set which is more realistic in a practical approach as very large sample set of data is usually used for analysis.
* The optimized algorithm is following the complexity of C\*Ω (m log m) whereas the default algorithm is following the complexity of C\*Ω (m) where C is some constant dependent on the size of m as the curve slope is increasing as m increases

Merge Sort:

# Algorithm Implemented:

*Text

Description automatically generated*

Merge(A, p, q, r)

*Text

Description automatically generated*

Time Complexity Analysis:

* Best Case: O(C\*log m)
* Worst Case: O(C1\*log m)
* Average Case: O(C2\*log m)

Here n is the vector dimension of each element of the array of size m, C,C1 and C2 are arbitrary constants

# Results (Tabulated Averages from Each Run)



and plotting data for c1 m log m and c2 m log m

|  |  |
| --- | --- |
| C1=0.000023 | C2=0.0000075 |
| **c1 m log m2** | **c2 m log m** |
| 3.056173847 | 0.996578428 |
| 7.640434618 | 2.491446071 |
| 15.28086924 | 4.982892142 |
| 30.56173847 | 9.965784285 |
| 76.40434618 | 24.91446071 |
| 152.8086924 | 49.82892142 |
| 305.6173847 | 99.65784285 |
| 764.0434618 | 249.1446071 |

## Results Graph

Observations:

* for all inputs the implemented merge sort is within the theoretical bounds of c1 m log m and c2 m log m
* c1 and c2 are arbitrary constants which we found as C1=0.000023 C2=0.0000075
* also the behavior of the merge sort is consistent for all types of vector inputs at a sufficiently large value of n ( here n =50 ) even for a large sample set of data making it a reliable and consistent sorting algorithm for a sufficiently large input array.

|  |  |
| --- | --- |
|  |  |

References:

1. Increasing Time Efficiency of Insertion Sort for the Worst Case Scenario <https://research.ijcaonline.org/icict/number7/icict1476.pdf>
2. An Approach to Improve the Performance of Insertion Sort Algorithm  
   <http://ijcset.com/docs/IJCSET13-04-05-068.pdf>
3. Enhanced Insertion Sort by Threshold Swapping  
   <https://thesai.org/Downloads/Volume11No6/Paper_59-Enhanced_Insertion_Sort_by_Threshold_Swapping.pdf>