HOMEWORK 1: INSERTION SORT

Introduction:

Reorganizing an array or list of elements based on a comparison operator on the elements is known as sorting. The comparison operator is used to find the definite order of the elements, so that search becomes easier. In computer science, sorting is essential to effective searching and data organization. It speeds up algorithms and promotes better decision-making by allowing well-organized data structures.

There are different methods to sort the elements, based on the comparison function, we have:

1. Bubble Sort: Compares neighboring elements repeatedly, switching them if they are out of order.
2. Selection Sort: Divides the input list into a sorted array and an unsorted array, selecting the smallest element from the latter and placing it in the prior.
3. Insertion Sort: Builds the final sorted array one item at a time by shifting elements as needed.
4. Merge Sort: Divides the input array into two halves, recursively sorts each half, and then merges the sorted halves.
5. Quick Sort: Selects a pivot element, partitions the array around the pivot, and recursively sorts the subarrays.
6. Heap Sort: Builds a heap from the input array and repeatedly extracts the maximum/minimum element from it to produce a sorted array.

In the context of this assignment, the majority of my scope of focus is on the study and assessment of the insertion sort algorithm and merge sort algorithm, focusing on its workings, performance features, and efficiency in the given problem.

Theory Overview:

a. Insertion sort:
A simple algorithm for sorting that uses comparisons to create the final array of sorted items by comparing a single item at a time. It gets the final sorted array by comparing one item at a time. This is a description of the way it operates:

1. Beginning with the second element: The second element in the array (or list) is thought to be the key at the beginning of the algorithm.
2. Comparing with elements to the left: Next, it proceeds from left to right, comparing this key with the elements to its left, until it determines the proper location for the key.
3. Shifting elements: It moves an element one position to the right if it discovers an element that is greater than the key while comparing it to elements to its left.
4. Inserting the key: The key is inserted into the designated spot after it has determined where it should go.
5. Proceeding to the next element: The algorithm builds a sorted sub-array progressively from left to right by repeating this procedure for each element in the array.
6. Iterative nature: The method by which insertion sort operates is by keeping a sorted subarray and adding elements to it one at a time.

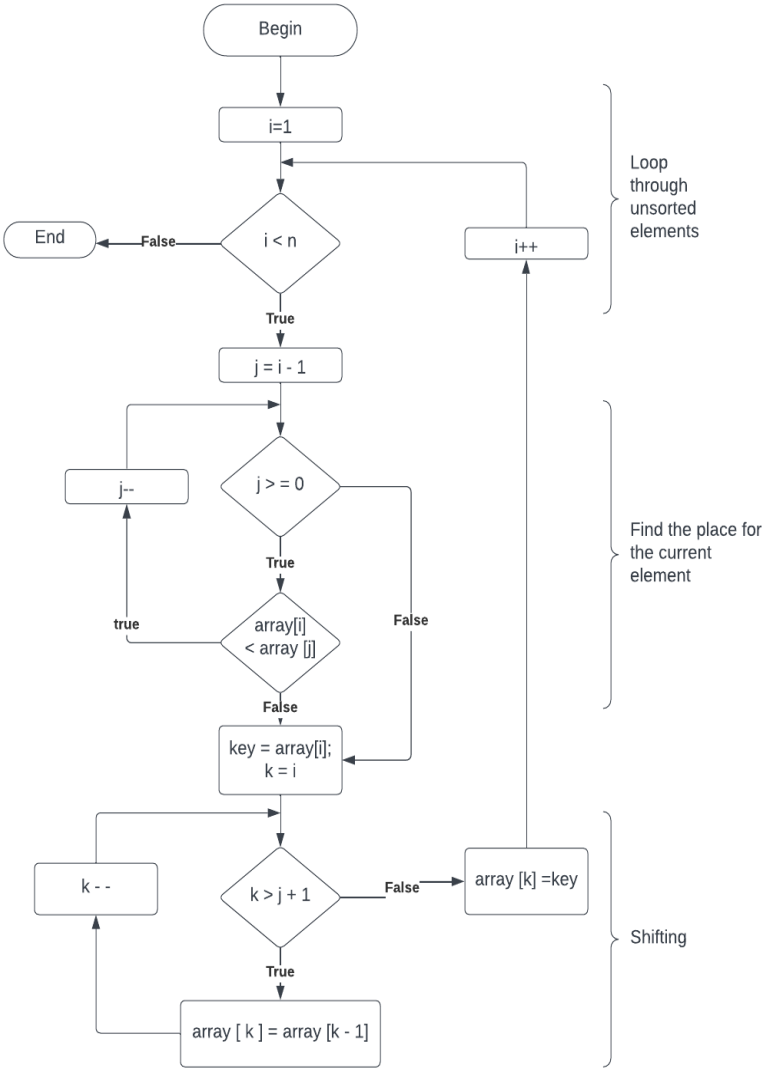


Fig 1. Flow chart of insertion sort (standard).

Time Complexity:

We consider 3 scenario cases to analyse the time complexity:

a. Random case scenario: Insertion sort typically performs worse than its best-case scenario but better than its worst-case scenario. Insertion sort's time complexity is typically $O(n^2)$ when the input array is randomly shuffled. This is because, to determine its proper position, each element must, on average, be compared with about half of the elements to its left.
b. Best-case scenario: When the input array is already sorted, insertion sort performs at its best. In this instance, the algorithm has to determine the proper position to compare each element with the elements on its left. Here, the number of elements in the array is n, and the time complexity is O(n). This is due to the possibility that to determine its proper location, each element may need to be compared to every element to its left.
c. Worst-case scenario: When the input array is organized in reverse order, it results in the worst-case scenario for insertion sort. In this case, every element must be compared with every element that is to its left up until the array's start. Here, the number of elements is n,

and the time complexity is $O(n^2)$. This is due to the possibility that the algorithm will need to make up to n comparisons for each element.

Assignment Statement and Implementation:

Implementation of question 1,

1. To develop an improved implementation of insertion sort for integer vector (insertion_sort_im) that precomputes the length of each vector before the sorting. By making sure, that the vectors are sorted according to their length (see ivector_length function) and checking the correctness using the check_sorted function.

Looking at the standard insertion sort,

Code:

Given, the simple code of insertion sort to be:

main.cpp | random_generator.... | Makefile | random_generator.h | sort.cpp | sort.h | timer.cpp | timer.h

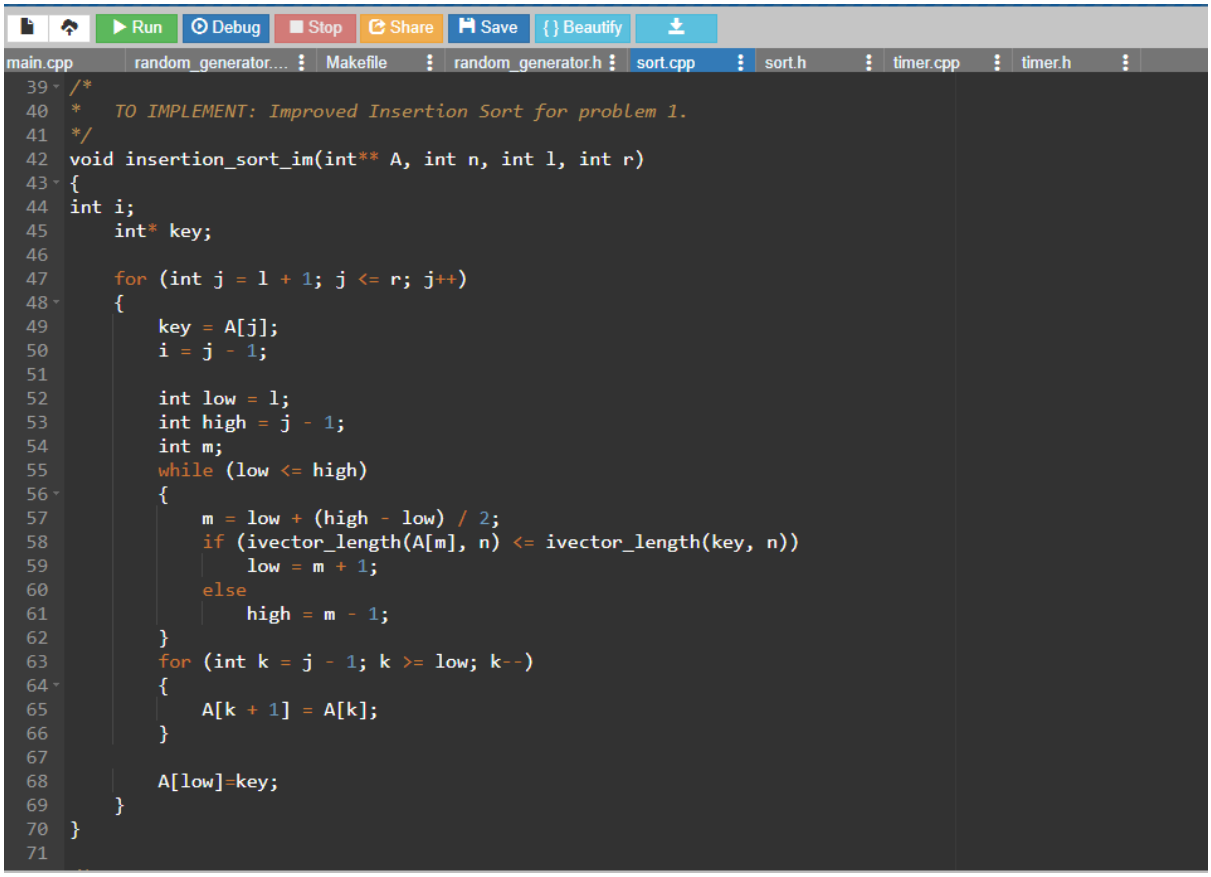
```
}

/*
 * insertion sort
 */
void insertion_sort(int** A, int n, int l, int r)
{
  int i;
  int* key;

  for (int j = l+1; j <= r; j++)
    {
      key = A[j];
      i = j - 1;

      while ((i >= l) && (ivector_length(A[i], n) > ivector_length(key, n)))
        {
      A[i+1] = A[i];
      i = i - 1;
    }

      A[i+1] = key;
    }
}

```

Explanation: The insertion sort function has 4 arguments (a pointer that points to a pointer to an integer, array size, left index, and right index). The *for loop* begins with the second element (l+1) and goes all the way up to and including the last element (r), this loop iterates over the elements in the range [l+1, r]. The while loop continues until the length of the vector (or array) at index i is greater than the length of the key vector, and i is greater than or equal to l (making sure it stays inside the left boundary). The function ivector_length likely determines a vector's (array's) length in a certain manner as declared. Later, it essentially creates space for the key element to be inserted by moving the element at index i one position to the right. Following that, it decreases i as it moves to the sorted subarray preceding element.

Now, improving the insertion sort:
We can improve the insertion sort to improve the time complexity of the algorithm. The algorithm takes a quite more amount of time for execution where there are large datasets. This slowness is because of a few factors while working with large datasets or while the key needs to be inserted close to the beginning:

- The *linear search* can be time-consuming.
- The *shifting operation* can be inefficient.
- The repeated *comparisons* can add up.

Taking the first point into consideration, we can implement binary search instead of linear search and improvise the time complexity. Let's look at the improvised version of the insertion sort (implemented using binary search instead of linear search),




```
/*
*   TO IMPLEMENT: Improved Insertion Sort for problem 1.
*/
void insertion_sort_im(int** A, int n, int l, int r)
{
int i;
    int* key;

    for (int j = l + 1; j <= r; j++)
    {
        key = A[j];
        i = j - 1;

        int low = l;
        int high = j - 1;
        int m;
        while (low <= high)
        {
            m = low + (high - low) / 2;
            if (ivector_length(A[m], n) <= ivector_length(key, n))
                low = m + 1;
            else
                high = m - 1;
        }
        for (int k = j - 1; k >= low; k--)
        {
            A[k + 1] = A[k];
        }

        A[low]=key;
    }
}

```

Explanation:
The improved insertion sort algorithm works on a range {[l, r]} of a two-dimensional integer array {A} with vector length {n}. Then, declare the variables key as a pointer to an integer and variables i as an integer. Begin a loop that iterates through the elements in the range [l+1, r] from l + 1 to r. Assign key the value of array A's jth element. Assign i to j - 1. Declare the variables m, low, and high. To determine the exact position of the key's insertion, start a binary search loop, wherein we locate the middle index "m" to split the sorted array (the elements left to the key that are already sorted) in half. Compare the key and the search proper index for the middle element. It performs binary search logic. Until the key is located or all of the elements of the array are used up, this process is repeated. Update low to m + 1 if the vector's length at index m in array A is less than or equal to the key's length; update high to m - 1. To create space for the key to be inserted at the index low, shift elements to the right. Place the key in array A at the proper low position.

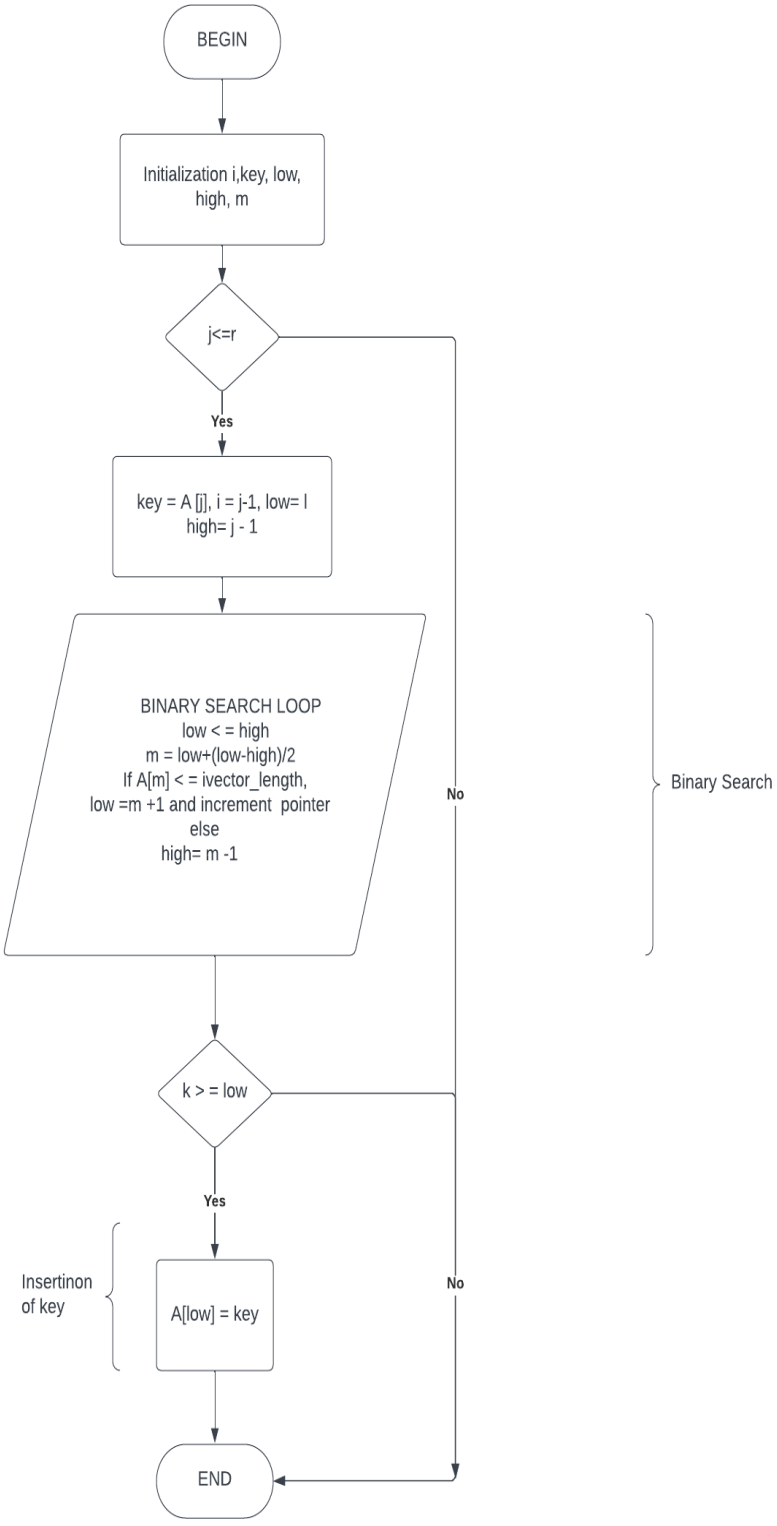


Fig 2. Flowchart of improved insertion sort

Time Complexity:

The time complexity of the insertion sort improves to *O (log n)* for each insertion when a binary search is used to determine the proper position for insertion. This causes the time to be *O (n log n)* for the sorting process. Particularly for larger datasets, this can be a significant improvement compared to the standard $O(n^2)$ time complexity of insertion sort.

Implementation of question 2,

2. To implement a merge sort for an array of integer vectors. Similar to the insertion sort, precomputing the length of the vectors before the sorting, and the sorting is done according to the vector lengths. checking the correctness using the check_sorted function.

Merge Sort:
An array is divided into smaller subarrays (having 1 element at the end), each is sorted, and the sorted subarrays are then merged back to form the final sorted array. This sorting algorithm is known as merge sort.
The merge search works on the principle of divide and conquer, as follows:

- Initial Division: Until each subarray has just one element after reduction, the input array is continuously divided into smaller subarrays.
  In general, the array is divided into parts recursively until each element is isolated.
- Merge Phase: The smaller sorted subarrays are put back together to form a single sorted array by the merge operation, which follows the divide phase.
  Comparing elements from two sorted subarrays and combining them into one sorted array is known as the merge operation.
- Recursion: Before merging the array back together, merge sort breaks the array into smaller chunk of subarrays and sorts each one separately.
  The process of recursion continues until every subarray consists of a single element, which can be easily sorted.

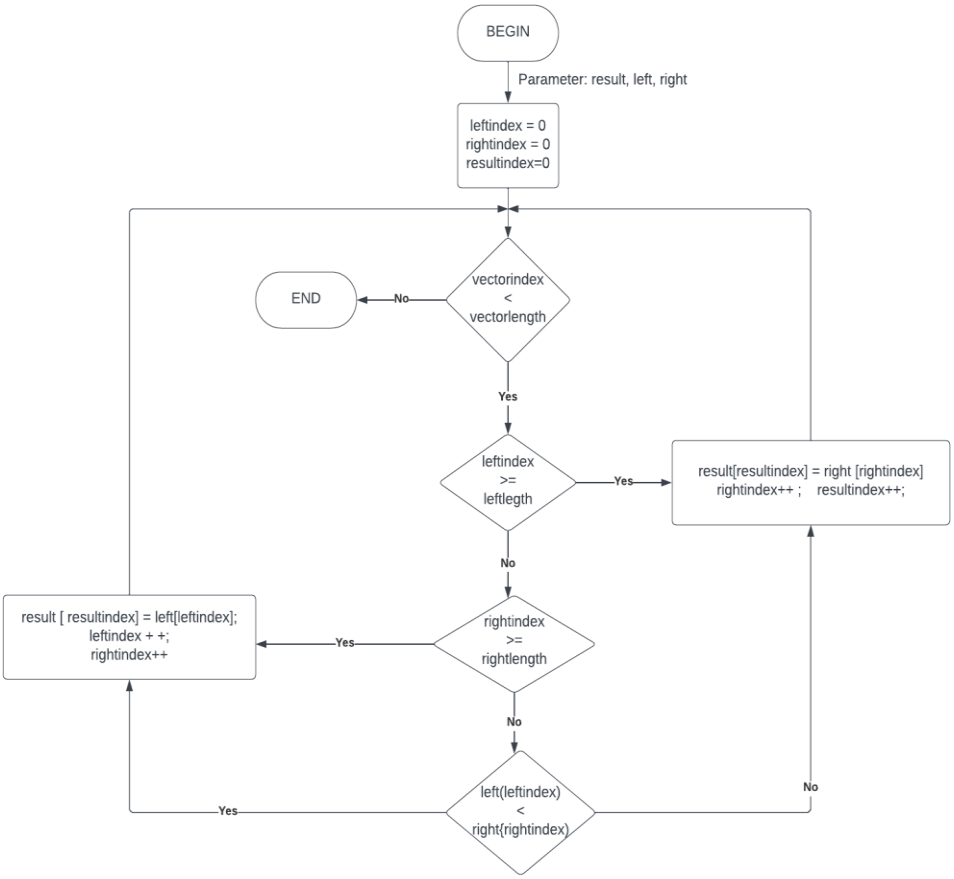
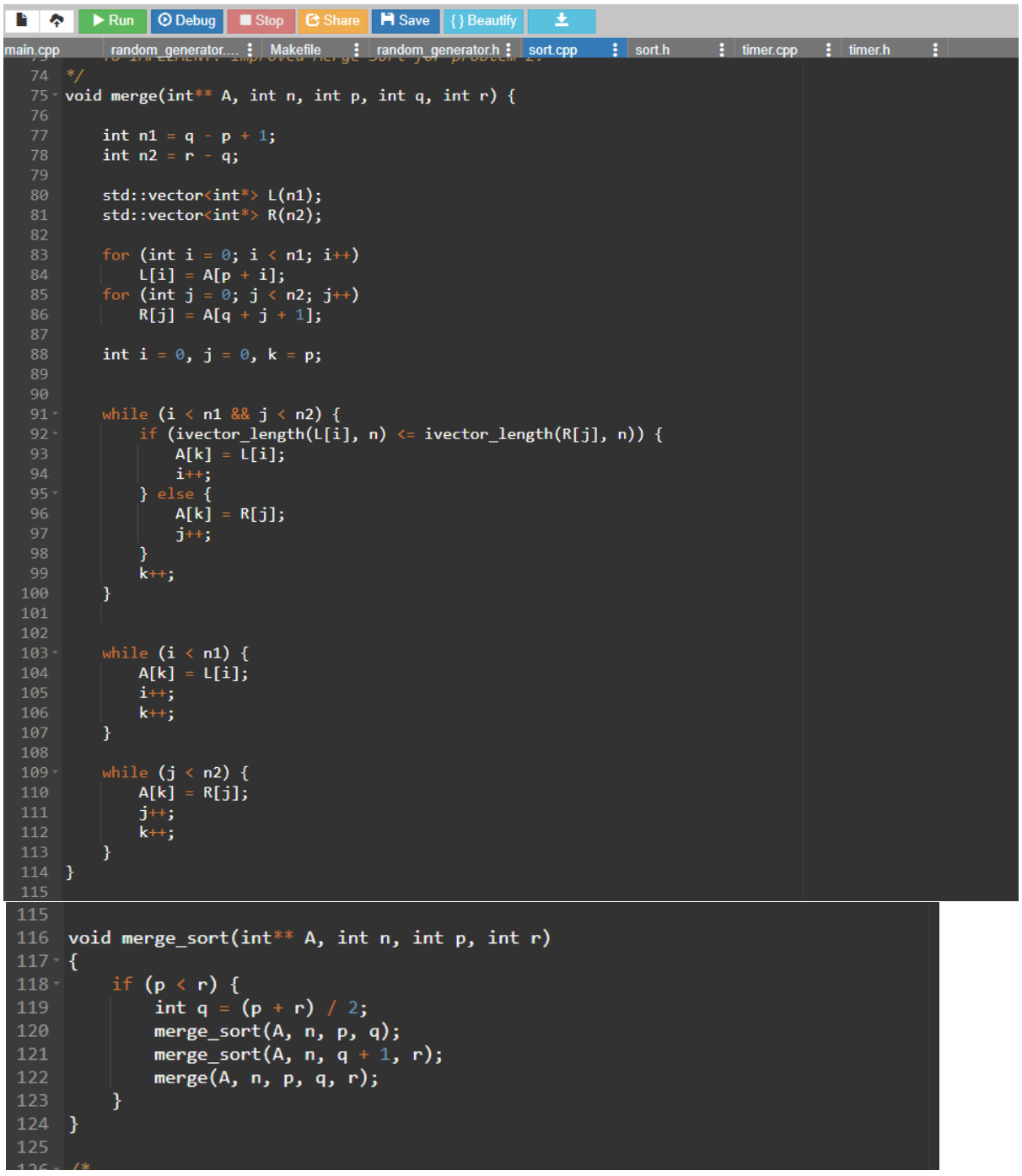


Fig 3. A generalized flow chart of merge sort

Code and Implementation:



```
*/
void merge(int** A, int n, int p, int q, int r) {

    int n1 = q - p + 1;
    int n2 = r - q;

    std::vector<int*> L(n1);
    std::vector<int*> R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = A[p + i];
    for (int j = 0; j < n2; j++)
        R[j] = A[q + j + 1];

    int i = 0, j = 0, k = p;


    while (i < n1 && j < n2) {
        if (ivector_length(L[i], n) <= ivector_length(R[j], n)) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
        k++;
    }


    while (i < n1) {
        A[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        A[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int** A, int n, int p, int r)
{
    if (p < r) {
        int q = (p + r) / 2;
        merge_sort(A, n, p, q);
        merge_sort(A, n, q + 1, r);
        merge(A, n, p, q, r);
    }
}
```

Explanation:

- Merge_function:
  The array A, its size n, and the indices p, q, and r, which indicate the subarrays to be merged, are the five parameters called for by this function. The two subarray sizes, n1 for the left subarray and n2 for the right subarray are computed. The elements of the left and right subarrays are stored in two temporary vectors, L and R, respectively. The left and right subarray elements are duplicated into the corresponding vectors. Next, using the comparison of vector lengths, the function returns the elements from vectors L and R to the original array A in sorted order. After handling any elements that are still in the left and right subarrays, they are copied back into the original array.
- Merge sort:

The array A, its size n, and the indices p and r, which indicate the range of the array to be sorted, are the four parameters required by this function. The array is split in half recursively until the base case when p is not less than r is reached. On both the left and right sides of the array, merge_sort is called. Ultimately, the sorted halves are merged back together by calling the merge function.

Time Complexity:
An array of integer pointers is successfully sorted according to the length of the vectors they point to by the provided merge sort algorithm. By continuously splitting the array in half and merging them back together in sorted order, it achieves an *O (n log n)* time complexity.

Result and analysis:

Implementation of question 3,

3. To measure the runtime performance of insertion sort (naive and improved) and merge sort for random, sorted, and inverse sorted inputs of size m = 10000; 25000; 50000; 100000; 250000; 500000; 1000000; 2500000 and vector dimension n = 10; 25; 50.

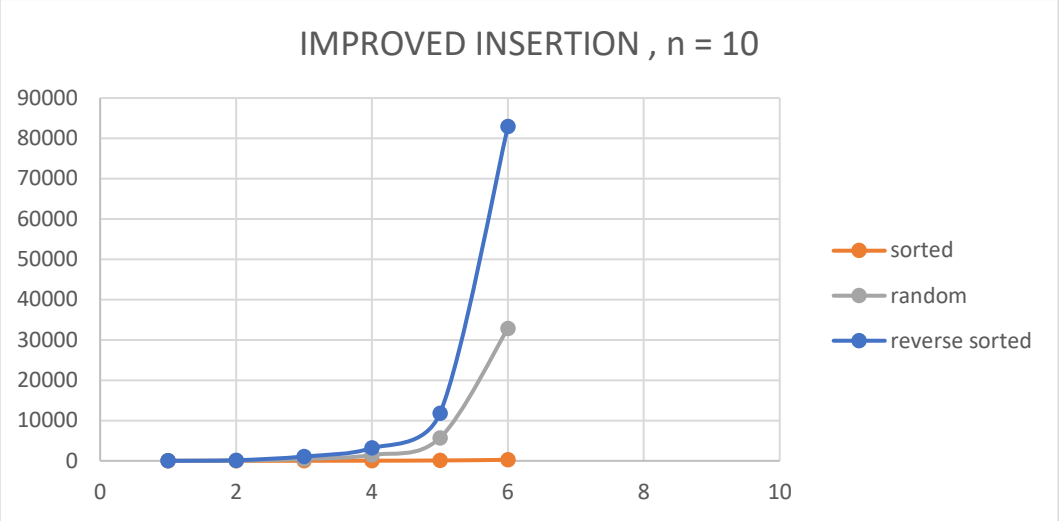
A. Improved Insertion sort:

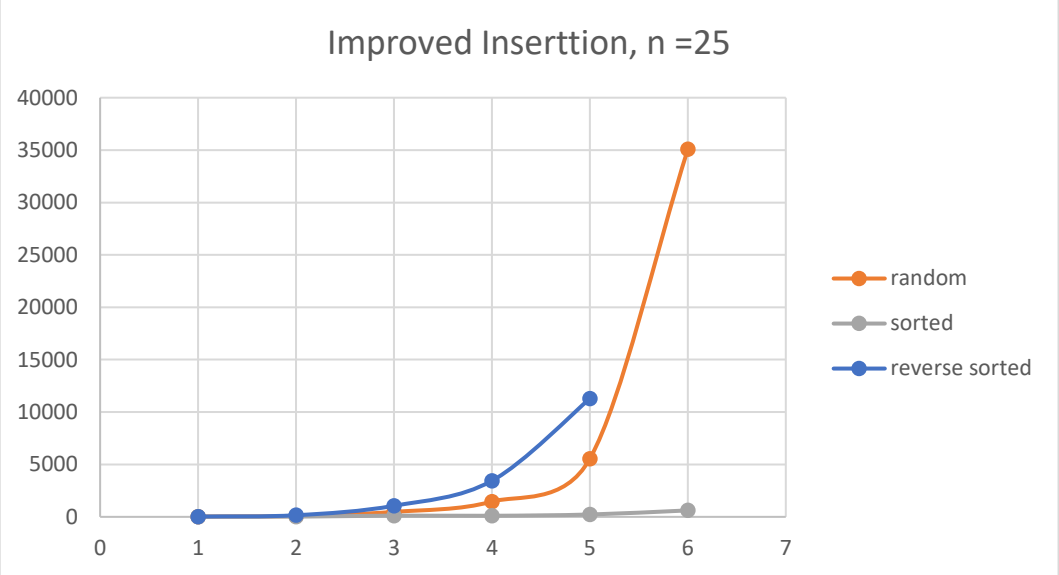
		Random	
M\N	**10**	**25**	**50**
10,000	72.3	88.4	107.9
25,000	468.2	473.3	521.9
50000	1457.3	1457.5	1677.2
100000	5697.3	5548.8	5,743
250000	32839.9	35079.3	34334.4
500000	-	-	-
1000000	-	-	-
2500000	-	-	-

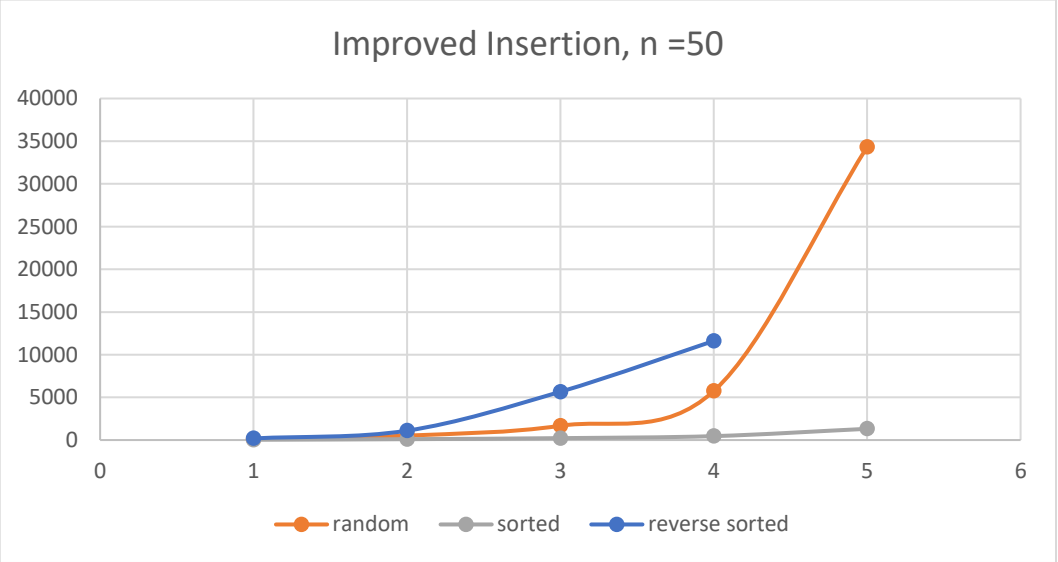
	Sorted	
10	**25**	**50**
6.778	18.714	36
18.7	117.9	106.2
43.6	103.4	226.2
93.909	224	461.2
258.1	614.222	1317.7
-	-	-
-	-	-
-	-	-

	Reverse	
10	**25**	**50**
159.6	153.8889	192.4
1076.75	1055.111	1097.818
3186.44	3440.5	5656.909
11759.2	11290	11615.2
-	-	-
-	-	-
-	-	-

Table 1. Tabulation for the time taken by the improved insertion sort for the given datasets.


IMPROVED INSERTION , n = 10
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
2
4
6
8
10
sorted
random
reverse sorted


Improved Inserttion, n =25
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
7
random
sorted
reverse sorted

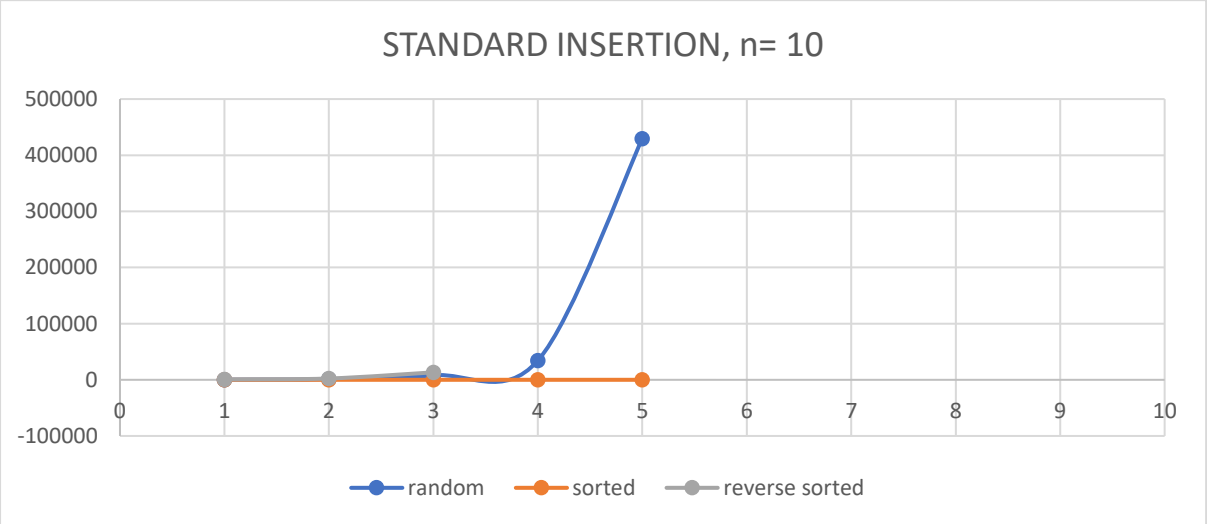

Improved Insertion, n =50
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
random
sorted
reverse sorted

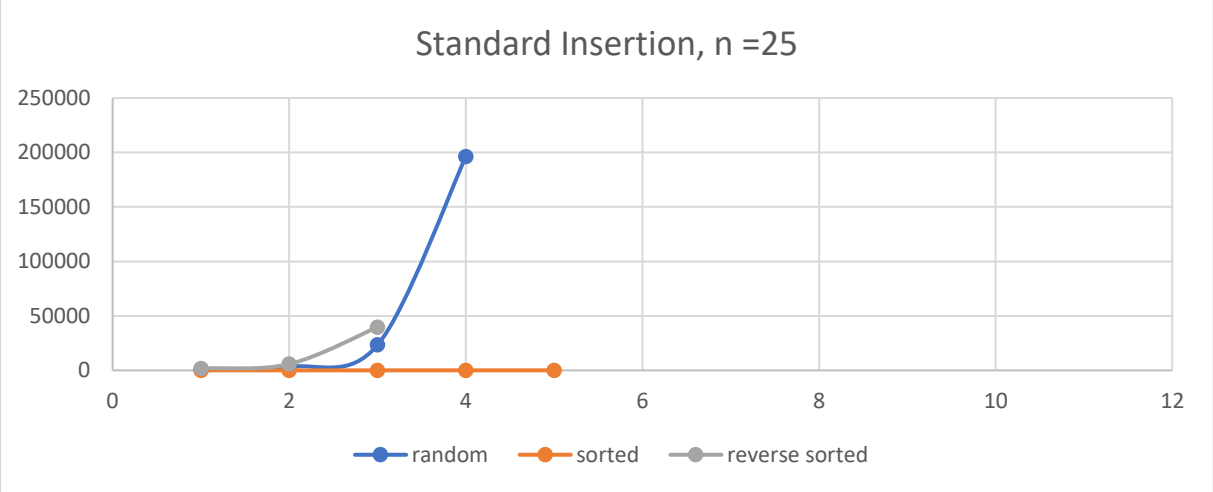
Inference from the above graphs:

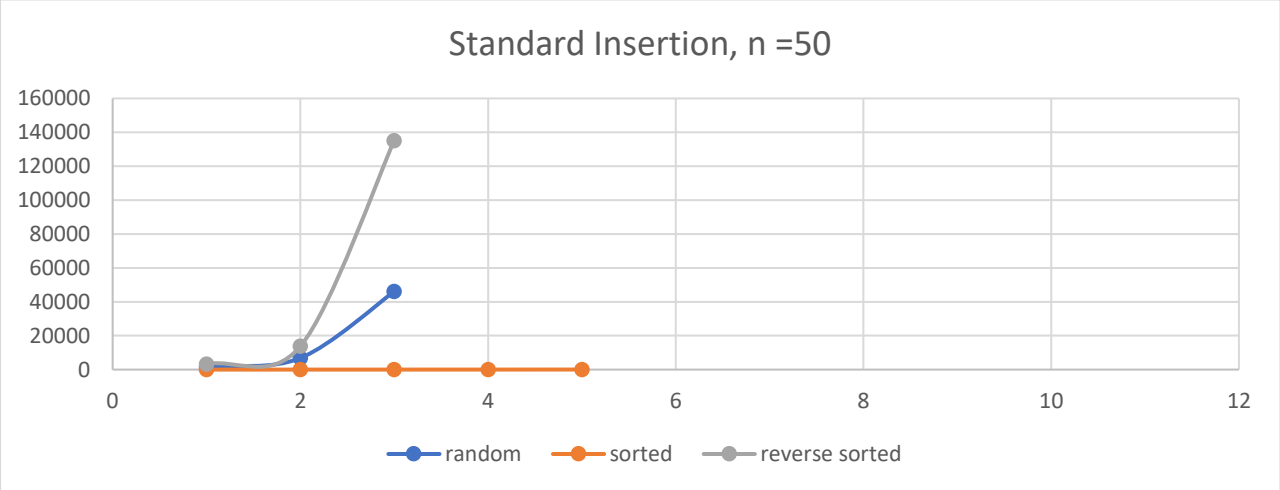
- Best Case: Time Complexity: *O(n2),* binary search finds correct insertion positions efficiently The effective search operation of binary search significantly increases the time complexity in the best-case scenario, leading to a significant reduction in performance over standard insertion sort.
- Worst Case: - Time Complexity: *O(nlog n)*, comparable to traditional insertion sort. Performance is improved over standard insertion sort.
- Random Case: - Performance varies depending on data distribution. *Binary search offers improvement*.

B. Standard insertion sort:

	Random			Sorted			Reverse sorted		
m/n	10	25	50	10	25	50	10	25	50
5000	313.8	878	1659.7	0	0.7	1.5	589.8	1582.9	3219.1
10,000	1219.7	3206.9	6844.5	0.4	1.5	2.8	2331.2	5770.3	13764.7
25,000	8106.4	23334.2	46089	1.5	3.4	8.4	13292	39738.1	135158.333
50000	34293.1	196262	-	3.428	7.545	15.3	-	-	-
100000	429562.14	-	-	6.2	13.9	28.7	-	-	-
250000	-	-	-	-	-	-	-	-	-
500000	-	-	-	-	-	-	-	-	-
1000000	-	-	-	-	-	-	-	-	-







Inference from the above graphs:

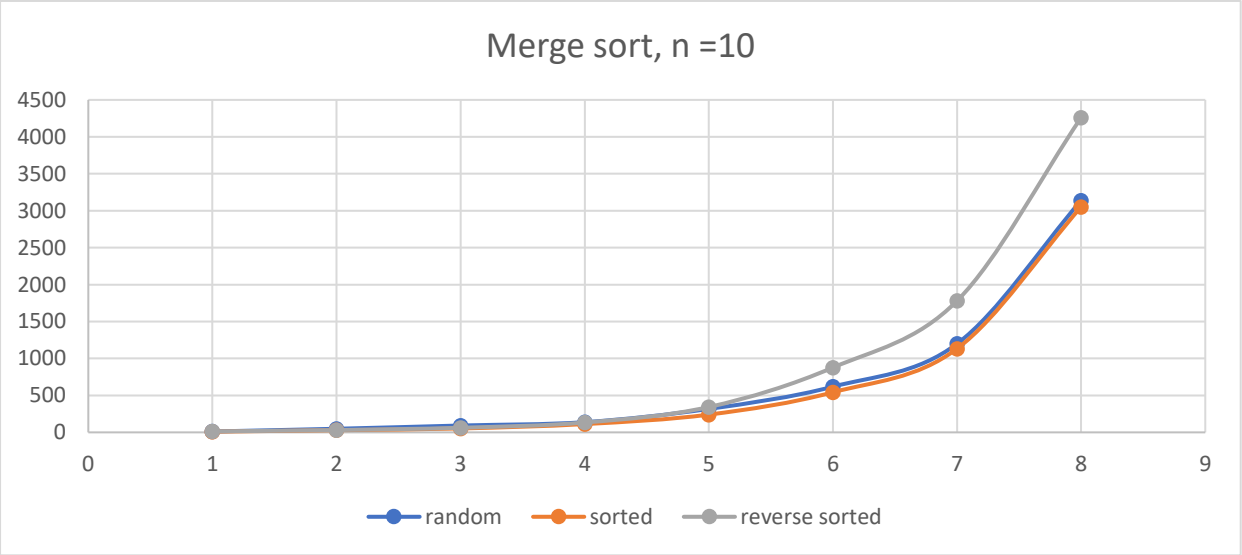
- Random Case: $O(n^2)$ is the average time complexity. The graph exhibits *a quadratic curve* with an increase in input size.
- Best Case: The time complexity is O(n). Performance that is noticeably superior to other scenarios. As input size increases, *the graph displays a linear trend.*
- Worst Case: The time complexity is $O(n^2)$. Most comparisons and swaps are necessary. As input size increases, the graph displays a *quadratic curve* that resembles the random case.

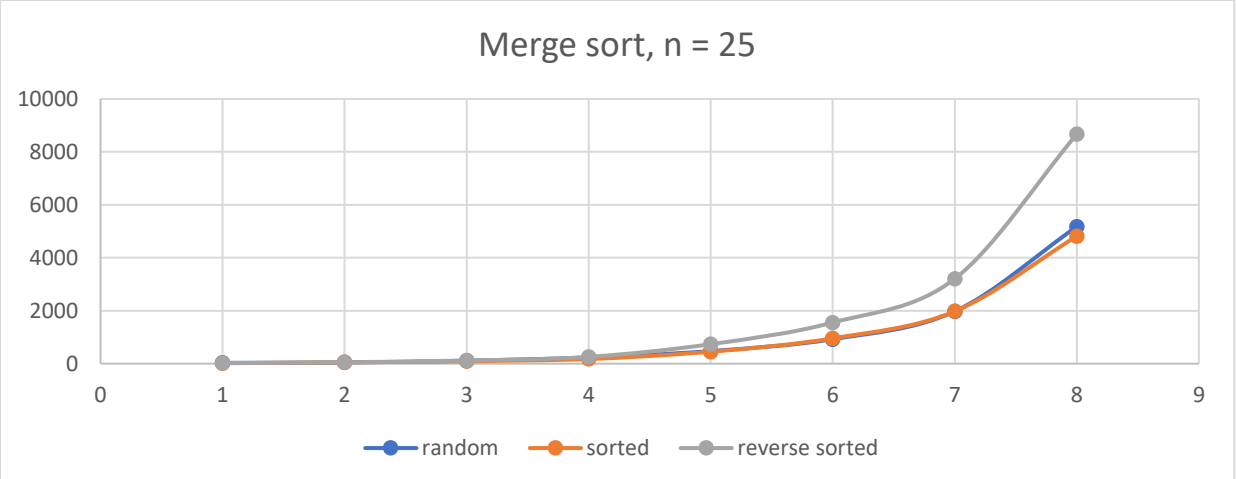
  Plotting the input size against the sorting time will allow you to infer these graphs; in the best case, you should see linear growth, and in the worst and random cases, quadratic growth.

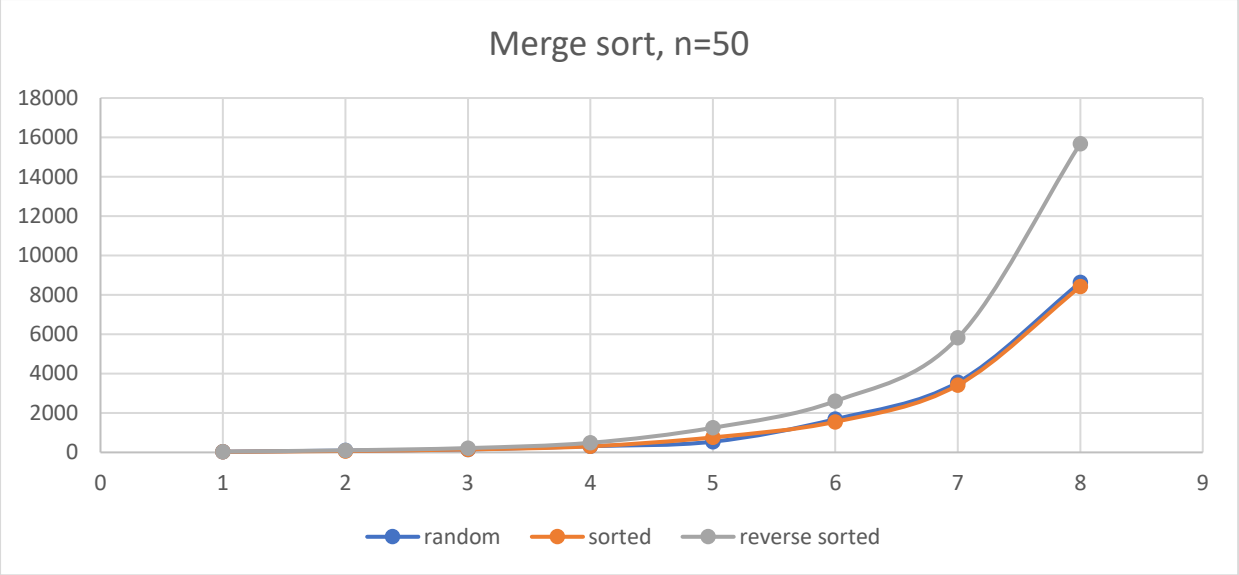
C. Merge sort:

	Random			Sorted			Reverse sorted		
m/n	10	25	50	10	25	50	10	25	50
10,000	12.1	31	33.692	9.8	17.8	27.9	13.1	21.4	37.1
25,000	49	47.5	90.9	29.6363	49.1	72.3	31.5	55.1	105.5
50,000	92.909	112	151.4	53.3	94.6	150.2	62.5	118.5	218.8
100000	138.4	235.4	315.2	112.4	176.2	308.3	133.4	257.8	487
250000	317.7	476.1	544.5	240.5	452.7	760.818	343.5	736.3	1250.1
500000	618.55	925.2	1687	543	958	1553.3	877.6	1553.8	2605
1000000	1199.5	1976.6	3558.4	1133.3	1978.4	3415.9	1780.6	3208.8	5815.4
2500000	3138.3	5177.6	8630.9	3052.8	4818.1	8419.62	4260.3	8669.4	15683.9

Table 3. Tabulation for the time taken by the merge sort for the given datasets.







Inference from the above graphs:

- Random Case Since merge sort splits the array recursively into smaller subarrays before merging them back together, it usually performs consistently with random data. While there may be occasional fluctuations, the random case graph mainly displays an even trend.
- Best Case: Because it equally divides and merges the subarrays that have already been sorted, merge sort is renowned for its reliable and effective performance even with sorted data. Similar performance traits may be shown by a close alignment of the sorted case's graph with the random
- Worst Case: The time complexity is O (n log n) in the worst case, best case. This situation is similar to insertion sort's worst-case results. The graph has a quadratic curve that resembles the other cases as the size of n increases.

Conclusion:

To put it all up, the merge sort algorithm is an efficient way to sort arrays of integer pointers. It can sort large sets of data because of its time complexity, which is obtained by using a technique known as divide-and-conquer. The system's dependability and consistent performance make it the preferred choice for a variety of applications where sorting efficiency and accuracy are essential. All things considered, merge sort is a reliable sorting method that strikes a mix between simplicity, speed, and efficacy.

The challenge is to improve the insertion sort algorithm more effectively in a variety of situations. It optimized the insertion operation and reduced it to O(log n) by including a binary search. This improvement is especially helpful for large sets of data that are difficult in standard insertion sort to manage. Implementing a threshold-based algorithm switching mechanism is something to think about in the future. It further aims to improve performance by actively switching, depending on the size of the input dataset, between more efficient algorithms like merge sort and insertion sort. With this flexible approach, sorting effectiveness would be

maximized in a variety of input situations.

In conclusion, the aim is to make the insertion sort algorithm more flexible and effective by including binary search and possibly implementing threshold-based algorithm switching in the future.

Reference list:

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Sedgewick, R., & Wayne, K. (2011). *Algorithms.* (4th ed.). Addison-Wesley