**DESIGN AND ANALYSIS OF ALGORITHMS**

**PROJECT 1B**

**CS - 5120**

**Project Report**

**Name of Student: Sakshi Shastri**

**Professor Name: Dr. Sankardas Roy**

**Fall 2023**

**Department of Computer Science**

**Bowling Green State University, Ohio**

**TABLE OF CONTENTS**

1. THEORITICAL ANALYSIS

* INSERTION SORT
* MERGE SORT
* HEAP SORT
* 4-WAY MERGE SORT

2. COMPARISION

* GRAPHS
* TABLES

3. ANALYSIS AND DISCUSSION

**Theoretical Analysis**

**Insertion Sort**

Insertion sort is a comparison-based algorithm. The array is divided into two subarrays, sorted and unsorted arrays, and the items from the unsorted part are assigned to the appropriate location in the sorted part. It's like the way we arrange the playing cards in our hands. It is also called in- place Sorting algorithm.

It works efficiently for the smaller data sets and sorted arrays, however for large data sets and random numbers it is not efficient in comparison to other sorting algorithms.

A white background with black text

Description automatically generated

The graph above is the insertion sort analysis for different cases: random number, constant numbers, reverse sorted and sorted numbers. As we can see in the above graph, insertion sort is more efficient for sorted and constant numbers and less efficient for reverse sorted and random numbers because for the sorted array no element is moved, only comparisons are being made. This is the best case of insertion sort, and the time complexity is O(n) and for the case of reverse sorted numbers, the insertion sort have two loops and the second loop runs multiple time making it worst case of insertion sort and the time complexity is O(n^2)

**Merge Sort**

Merge sort uses the divide and conquer approach to sort an array. This algorithm divides an array into subarrays, sorting those subarrays and merging the sorted subarrays together to get a final sorted array.

Merge sort is efficient for large data sets.

A white background with black text

Description automatically generated

As we can see in the above graph, even if the array is sorted, it still must divide it into subarrays, for all type of cases we have to first divide the array into subarrays therefore merge sort takes extra space to sort the array.

So, the best case, average case and the worst case for merge sort are same O(nlogn) because it always divides the array into subarrays and then merge them in linear time.

**Heap Sort**

Heap sort is a comparison-based sorting technique in which first we build a heap using the unsorted array and then we use the same heap to sort the array by removing the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Algorithm of heap sort.

A math equations on a white background

Description automatically generated

Here in heap sort if the array is sorted in ascending order or reversely sorted then also it will take O(nlogn) amount of time.

**4-way merge sort**

This sort is basically similar to merge-sort algorithm, the only difference is that in this the array breaks down into the subarrays of size one-fourth.

Algorithm of 4-way merge sort

MergeSort4(A,p,r){  
1. if (r-p <=3) {  
2. sort A[p,r] and then return  
}  
3. q1 = p +(r - p)/4  
4. q2 = p +2\*(r - p)/4  
5. q3 = p +3\*(r - p)/4  
6. MergeSort4(A, p, q1)  
7. MergeSort4(A, q1+1, q2)  
8. MergeSort4(A, q2+1, q3)  
9. MergeSort4(A, q3+1, r)  
10.Merge(A,p,q1,q2,q3,r)

}

When we look at the Time Complexity of 4-way merge sort then it is same as merge sort but only the base of log will change because at each iteration, we are dividing the array into one-fourth, so the time complexity for 4 way merge sort will be {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><msub><mi>log</mi><mn>4</mn></msub><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"}.

**COMPARISON:**

**Graphs**

1. **Comparison in regard to execution time.**

Fig1. Comparison of all 4 algorithms for constant numbers.

Fig2. Comparison of all 4 algorithms for Sorted numbers.

Fig3. Comparison of all 4 algorithms for random numbers

Fig4. Comparison of all 4 algorithms for reverse sorted numbers

1. **Comparison in regard to number of comparisons.**

Fig1. Comparison of all 4 algorithms for reverse sorted numbers

Fig2. Comparison of all 4 algorithms for constant numbers.

Fig3. Comparison of all 4 algorithms for random numbers.

Fig4. Comparison of all 4 algorithms for sorted numbers.

**Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TIME COMPLEXITY | INSERTION SORT | MERGE SORT | HEAP SORT | 4 WAY MERGE SORT |
| BEST CASE | {"mathml":"<math xmlns=\"http://www.w3.org/1998/Math/MathML\" style=\"font-family:stix;font-size:16px;\"><mi>O</mi><mfenced><mi>n</mi></mfenced><mspace linebreak=\"newline\"/></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><mi>log</mi><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><mi>log</mi><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><msub><mi>log</mi><mn>4</mn></msub><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} |
| AVERAGE CASE | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><msup><mi>n</mi><mn>2</mn></msup></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math xmlns=\"http://www.w3.org/1998/Math/MathML\" style=\"font-family:stix;font-size:16px;\"><mi>O</mi><mfenced><mrow><mi>n</mi><mi>log</mi><mi>n</mi></mrow></mfenced></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><mi>log</mi><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><msub><mi>log</mi><mn>4</mn></msub><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} |
| WORST CASE | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><msup><mi>n</mi><mn>2</mn></msup></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math xmlns=\"http://www.w3.org/1998/Math/MathML\" style=\"font-family:stix;font-size:16px;\"><mi>O</mi><mfenced><mrow><mi>n</mi><mi>log</mi><mi>n</mi></mrow></mfenced></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><mi>log</mi><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} | {"mathml":"<math style=\"font-family:stix;font-size:16px;\" xmlns=\"http://www.w3.org/1998/Math/MathML\"><mstyle mathsize=\"16px\"><mi>O</mi><mfenced><mrow><mi>n</mi><msub><mi>log</mi><mn>4</mn></msub><mi>n</mi></mrow></mfenced></mstyle></math>","origin":"MathType for Microsoft Add-in"} |

**Analysis and Discussion**

We have performed 4 sorting algorithms.

1.Insertion sort

2.Merge sort

3.Heap sort

4. 4-way merge sort

After performing all the algorithms on different data sets such as random numbers, sorted number, reverse sorted numbers and constant numbers I analyzed that,

Insertion sort is easy to implement and performs efficiently on sorted numbers and constant numbers with minimum number of comparisons and swaps while on the other hand merge sort is also efficient, but it doesn’t utilize the fact that the input is already sorted and goes through the entire process. Insertion sort is efficient for small data sets and takes a lot of time for large data sets due to its quadratic time complexity.

For random and reverse sorted numbers, insertion sort is simple but is inefficient and merge sort, 4-way merge sort and heap sort are highly efficient.

Heap sort is a very efficient algorithm, but it takes a lot of time to compare and execute. It is also known as an in-place algorithm as it also does not require extra space.

To summarize, insertion sort works well in scenarios where data is sorted or have constant numbers. Merge sort, heap sort, and 4-way merge sort are consistent performers across different input scenarios. However, merge sort and 4-way merge sort require additional space for the merging step because it follows divide and conquer, making heap sort the most space-efficient among them but it has additional overhead due to heapify operation.