

Alexandria University

Faculty of Engineering

CSED 2025

Data Structures & Algorithms

Assignment 4

Binary Heap and Sorting Techniques

**Submitted By:**

Sama Tarek Anwar Zayed 20010698

Marshelino Maged Gaber 20011136

Mariam Osama Ahmed 20011878

Mariam Gerges Zaki 20011880

Patrick Georges Kromil 20010383

**Submitted To:**

Dr. Ayman Khalafallah

Eng. Mohamed Tarek El Habiby

Eng. Ismail El Yamani

Sorting Algorithms Implemented :

1- Bubble Sort Algorithm :

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

**TIme Complexity of Bubble Sort is O(n²)** **:**

But why ?

Here are the steps needed :

* traverse from left and compare adjacent elements and the higher one is placed at right side.
* In this way, the largest element is moved to the rightmost end at first.
* This process is then continued to find the second largest and place it and so on until the data is sorted.

The algorithm requires traversing the array and for each element, we compare it with the next one if it is larger than the next we then swap it until the max element is bubbled out at the end of the array. So for n elements w perform nearly n comparisons which makes this algorithm takes nearly n² comparisons so its Time Complexity is O(n²).

**Space Complexity of Bubble Sort is O(1) :**

One of the advantage of this algorithm is that it does not require any auxiliary space just a temporary variable while swapping.So it performs the sorting operation in place.

2- Merge Sort Algorithm :

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

**Time Complexity of Merge Sort is O(n.log₂(n)) :**

But why ?

* Divide - It is the initial stage where the midpoint of the array is found using

mid=start+(end−start)/2

* Conquer - In this step, the array is divided into subarrays, using the midpoint calculated. The process repeats itself recursively until all elements become single array elements.
* Combine - In this step, the subarrays formed are combined in sorted order.

Here are the steps needed :

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

So by the recurrence relation : T(n) = 2\*T(n/2) + O(n)

This recurrence will run in O(n.log₂(n)), which makes the merge sort a favorable sorting algorithm

**Space Complexity of Merge Sort is O(n) :**

In merge sort, all elements are copied into an auxiliary array of size N, where N is the number of elements present in the unsorted array. Hence, the space complexity for Merge Sort is O(n).

3- Radix Sort Algorithm :

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

**Time Complexity of Radix Sort is O(n)**

But why ?

What is the running time of Radix Sort?

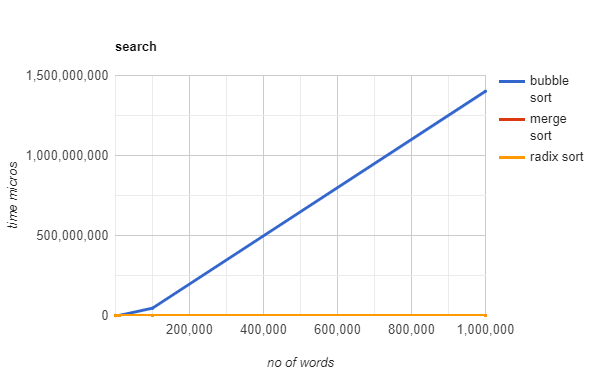
Let there be **d** digits in input integers. Radix Sort takes O(d\*(n+b)) time where **b** is the base for representing numbers, for example, for the decimal system, **b** is 10. What is the value of **d**? If **k** is the maximum possible value, then d would be O(logb(k)). So overall time complexity is O((n+b) \* log**b**(k)). Which looks more than the time complexity of comparison-based sorting algorithms for a large k. Let us first limit k. Let k <= nc where c is a constant. In that case, the complexity becomes O(n\*Logb(n)). But it still doesn’t beat comparison-based sorting algorithms.

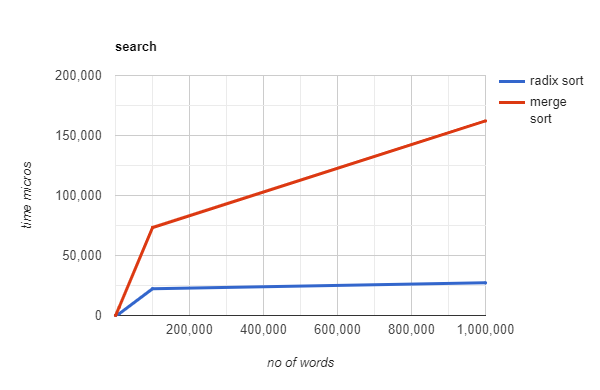
What if we make the value of b larger? What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array of integers with a range from 1 to nc if the numbers are represented in base n (or every digit takes log2(n) bits).

**Space Complexity of Radix Sort is O(n+b) where b is the base of the numbers represented.**

Performance Measures Comparison:

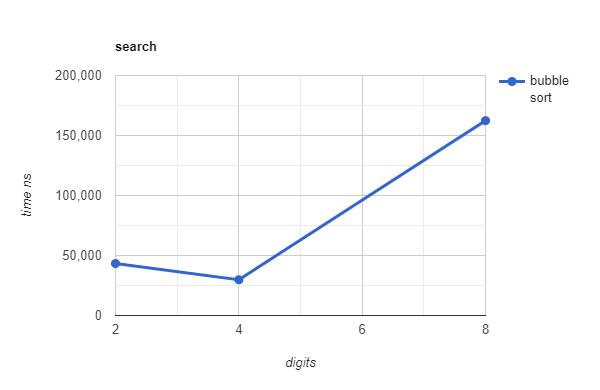
| Array Size | Bubble Sort | Merge Sort | Radix Sort | Notes |
| --- | --- | --- | --- | --- |
| 10 | 21,600 nanosecond | 59,000 nanosecond | 140,000 nanosecond | Array has negative numbers |
| 1000 | 6,659,100 nanosecond | 541,300 nanosecond | 188,000 nanosecond | All positive |
| 10,000 | 205,675,400 nanosecond | 6,876,900 nanosecond | 1,464,600 nanosecond | All positive |
| 100k | 45,732,031,300 nanosecond | 73,375,000 nanosecond | 22,293,900 nanosecond | All positive |
| 1M | 1,401,590,668,700 nanosecond | 162,132,200 nanosecond | 27,192,900 nanosecond | All positive |





Study the effect of number of digits on the performance of radix sort

| Number of digits | Radix Sort Time |
| --- | --- |
| 2 digits | 43,300 nanosecond |
| 4 digits | 29,800 nanosecond |
| 8 digits | 162400 nanosecond |



Study of best, average, and worst case for Bubble Sort

Array Size = 100 elements

| Case | Bubble Sort Time |
| --- | --- |
| Worst Case (array sorted descendingly) | 206,100 nanoseconds |
| Average Case (Scattered values) | 169,000 nanoseconds |
| Best Case (Array is already sorted) | 77,800 nanoseconds |

Comparison between the performance of the 3 algorithms using arrays of positive and negative numbers\*.

| Array Size | Bubble Sort | Merge Sort | Radix Sort |
| --- | --- | --- | --- |
| 100 | 172,200 nanoseconds | 77,000 nanoseconds | 51,300 nanosecond |
| 1000 | 211,900 nanosecond | 42,600 nanosecond | 47,100 nanosecond |
| 10,000 | 172,200 nanosecond | 42,400 nanosecond | 81800 nanosecond |
| 100,000 | 12,656,587,700 nanosecond | 19,590,700 nanosecond | 5,663,600 nanosecond |
| 1,000,000 | 1,389,987,204,400 nanosecond | 151,984,200 nanosecond | 48,371,300 nanosecond |

\*Negative numbers causes an overhead on the radix sorting algorithm due to the need to get the minimum number (the largest in magnitude with negative sign) and map that number to zero before starting, then after sorting we need to return the numbers to their initial values which needs to loop again over the array contributing to a large overhead which makes the algorithm performs worse than the comparison and divide and conquer based algorithms. Note that this case only happens when the range between the min and max numbers is large.

