Anusha Komarlu Pradeep Kumar

NET ID: 1002082930

Problem Statement

Implement and compare the following sorting algorithm:

- Insertion sort
- Mergesort
- Heapsort
- Quicksort
- Radix sort

How do their running times change with respect to data size? How does their speed compare to each other in cases with different data sizes? Can you improve their running time with your discovery? Which one is better in terms of what conditions?

Insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons.

	Time complexity Best Case	**Time complexity Average Case**	**Time complexity Worst Case**	**Space complexity**
Insertion sort	O(n)	O(n^2)	O(n^2)	O(1)

When to use insertion sort?

- When data is already sorted (or nearly sorted).
- When space efficiency is preferred over time.
- When stable sorting is required (relative order is important).
- When the dataset to be operated is of small size.

- When in-place algorithm is needed.

Reason: Having a time complexity of O(n^2), Insertion sort is the main choice when the data is nearly sorted (because it is more adaptive) or when the data/array size is small (because it has low overhead). For these reasons, and because moreover, it is also stable, it is mostly used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

Mergesort

- The Merge Sort algorithm is a sorting algorithm that is based on the Divide and Conquer paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

	Time complexity **Best Case**	**Time complexity** **Average Case**	**Time complexity** **Worst Case**	**Space complexity**
Mergesort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)

- When to use merge sort?
    - The dataset to be sorted is large (also useful in Linked List sorting).
    - We need a faster algorithm (time is a major constraint).
    - Memory overhead is not a big issue.
    - Stability of data is needed with faster implementation.

- External Sorting method with the poor locality of reference is not an issue.

Reason: It has a time complexity of O(n log n) making it much faster. It uses a divide and conquer strategy and hence, therefore, requires extra space to be implemented. It also produces a stable algorithm. Due to the divide and conquer strategy it is an optimum algorithm for sorting Linked List data structure. Due to its lower time complexity and simple divide-and-conquer algorithm, it’s mostly applicable to all types of datasets.

Quicksort

- Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

	Time complexity Best Case	**Time complexity Average Case**	**Time complexity Worst Case**	**Space complexity**
Quick sort	**$\Omega(n\log n)$**	**$\theta(n\log n)$**	**$O(n^2)$**	**$O(\log n)$**

- Better to use when :
    - We need a faster sorting algorithm with a good locality of reference.
    - Internal sorting method is preferred.
    - Extra memory is a constraint.
    - The dataset size is not very big.

- Stability is not an issue (relative order is not important).

Reason : It is similar algorithm to Merge Sort having divide and conquer approach. When implemented well it can be even faster than Merge sort and a lot faster than Heap sort. But in worst cast its time complexity becomes quadratic and it can perform worse than Merge Sort. That's why its not much frequently used. As it is in-place sorting algorithm like insertion sort but also with less time like Merge sort hence it is used along with merge sort in implementing Library Sorting functions of various languages. It has better locality of reference than its neighbors Merge sort and Heap sort.

Heapsort

- Heapsort is the in-place sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

	Time complexity **Best Case**	**Time complexity** **Average Case**	**Time complexity** **Worst Case**	**Space complexity**
Heap sort	Ω(nlogn)	θ(nlogn)	O(nlogn)	O(1)

- Better to use when :
    - We need extreme elements very fast.
    - We need a partially sorted array (if execution stopped abruptly).
    - We need to avoid quicksort worst case.
    - Stability is not an issue (relative order is not important).

- Memory complexity (extra space) is an issue.

Reason : Its main competitor is the quicksort algorithm .Heapsort's primary advantages are its simple, non-recursive code, minimal auxiliary storage requirement, and reliably good performance: its best and worst cases are within a small constant factor of each other. This algorithm cannot do better than $O(n \log n)$ for pre-sorted inputs, but it does not suffer from quicksort's $O(n2)$ worst case which makes its implementation a little tricky. Due its poor locality of reference not much famous. As it use the property of heaps to sort elements so getting extreme elements is very easy.

Radixsort

- The idea of Radix Sort is to do digit-by-digit sort starting from the least significant digit to the most significant digit.

	Time complexity **Best Case**	**Time complexity** **Average Case**	**Time complexity** **Worst Case**	**Space complexity**
Radix sort	Ω(N k)	θ(N k)	O(N k)	O(N +k)

- Better to use when :
    - The biggest integer is shorter than array size.
    - We have fixed range of integers.
    - Numbers are not much repeated but their length have same range.
    - Stability of data is needed with faster implementation.

Reason : Radix sort is one of the unique sorting algorithm as here the time complexity depends upon the number of digits also along with number of elements in the list. Generalization of this algorithm is tough then the other sorting algorithms there as it requires fixed size keys, and some very specific way of breaking the keys into pieces.

Running time with different data sizes

Algorithm	100	1K	10K
Insertion sort	0.0273	0.4121	12.8598
Merge sort	0.00134	0.0180	0.21675
Quick sort	0.00129	0.0082	0.0632
Heap sort	0.00268	0.221286	0.4457
Radix sort	0.00214	0.0080	0.0952