MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work 2:

Study and empirical analysis of sorting algorithms

Checked by:

Fiștic Cristof, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

TABLE OF CONTENTS:

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, (one of your choices).

Tasks:

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer. In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.

2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.

3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).

4. The algorithm is implemented in a programming language.

5. Generating multiple sets of input data.

6. Run the program for each input data set.

7. The obtained data are analyzed. The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

In computer science, a sorting algorithm is an algorithm that puts elements of a list into an order. The most frequently used orders are numerical order and lexicographical order, and either ascending or descending. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output.

Formally, the output of any sorting algorithm must satisfy two conditions:

1. The output is in monotonic order (each element is no smaller/larger than the previous element, according to the required order).
2. The output is a permutation (a reordering, yet retaining all of the original elements) of the input.

For optimum efficiency, the input data should be stored in a data structure which allows random access rather than one that allows only sequential access.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (O(n))

Input Format:

As input, each algorithm will receive different size unsorted arrays. We will have an initial array with the lengths ‘arrays_lengths = [100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 15000, 20000, 30000, 40000, 50000, 75000, 100000]’, and for each element of this array we will create a new array with the corresponding size, the elements of this array will be random. For each array created we will call the respective function and will sort the array 3 times and return the best sorting time.

IMPLEMENTATION

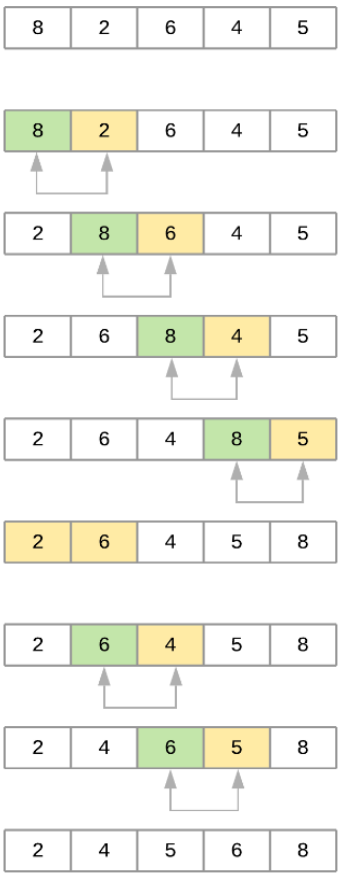
We will make an empirical analysis of bubble, insertion, merge, quick, heap and tim sorting algorithms in python. All these algorithms will be implemented in their naive form in python and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

Bubble Sort:

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed. These passes through the list are repeated until no swaps have to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a comparison sort, is named for the way the larger elements "bubble" up to the top of the list.

Algorithm Description:

To properly analyze how the algorithm works, consider a list with values [8, 2, 6, 4, 5]. Assume you're using bubble_sort. Here's a figure illustrating what the array looks like at each iteration of the algorithm:



Implementation:

```python
def bubble_sort(array):
    n = len(array)

    for i in range(n):
        already_sorted = True
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]
                already_sorted = False
        if already_sorted:
            break

    return array
```

Figure 1 Bubble sort python implementation

Results:

After running the function for each different size unsorted arrays and for each array repeating the sorting process for 3 times and selecting the best time result we got the following results:

Time results of bubble_sort algorithm

Array Length	Time
100	0.002543
1000	0.275203
2000	1.149572
3000	2.641792
4000	4.750357
5000	7.473605
6000	10.757229
7000	14.549099
8000	19.168438
9000	24.173812
10000	29.771275

Figure 2 Bubble sort results

Looking at the table we may observe that it takes more than 10 seconds to sort an array of 6000 elements.

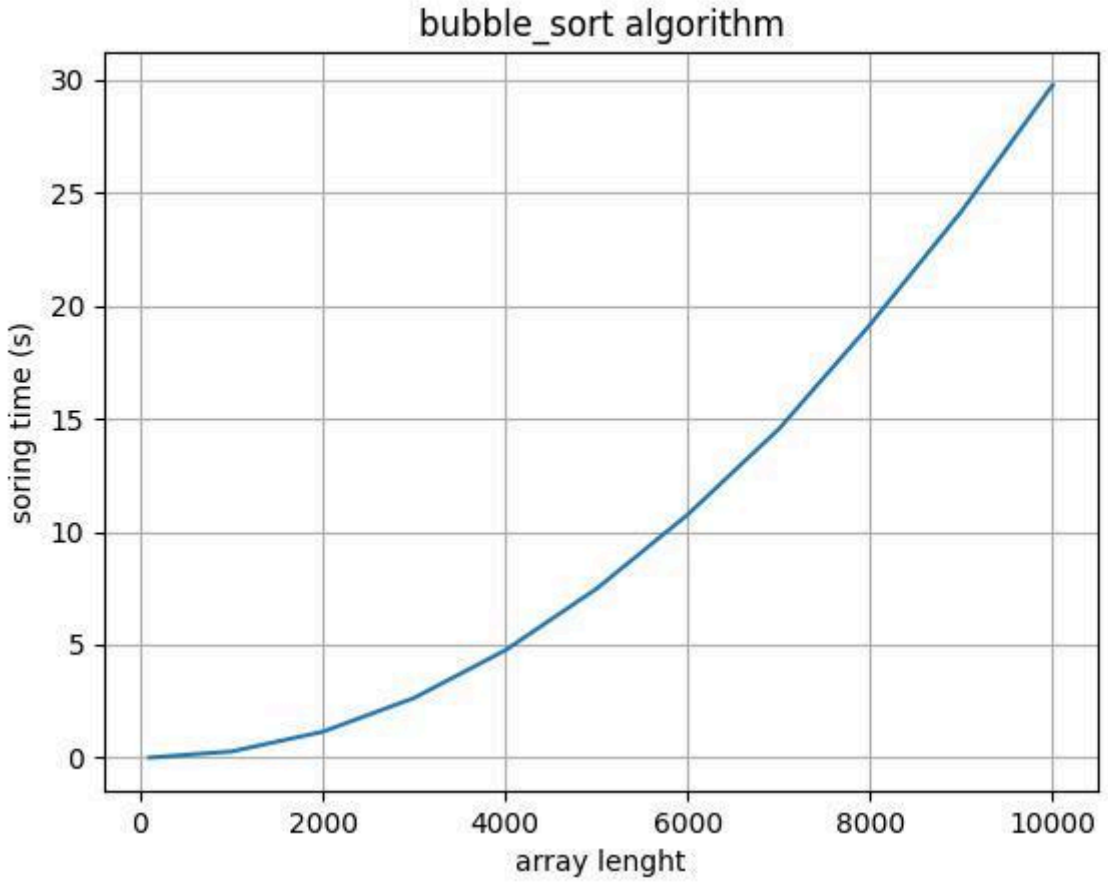


Figure 3 Graph of Bubble sort algorithm

The main advantage of the bubble sort algorithm lies in its simplicity. It is straightforward for both implementation and comprehension, which is likely the primary reason why most computer science courses introduce the topic of sorting using bubble sort.

As observed previously, the disadvantage of bubble sort is its slow performance, characterized by a runtime complexity of O(n^2). Regrettably, this precludes its practical suitability for sorting large arrays.

Insertion sort:

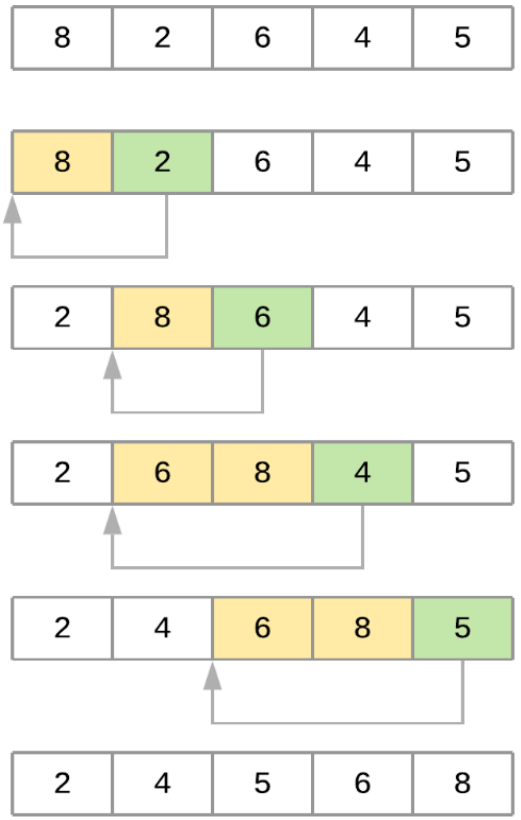
Like bubble sort, the insertion sort algorithm is straightforward to implement and understand. But unlike bubble sort, it builds the sorted list one element at a time by comparing each item with the rest of the list and inserting it into its correct position. This “insertion” procedure gives the algorithm its name.

An excellent analogy to explain insertion sort is the way you would sort a deck of cards. Imagine that you’re holding a group of cards in your hands, and you want to arrange them in order. You’d start by comparing a single card step by step with the rest of the cards until you find its correct position. At that

point, you'd insert the card in the correct location and start over with a new card, repeating until all the cards in your hand were sorted.

Algorithm Description:

Here's a figure illustrating the different iterations of the algorithm when sorting the array [8, 2, 6, 4, 5]:



Implementation:

```
def insertion_sort(array):
    for i in range(1, len(array)):
        key_item = array[i]
        j = i - 1
        while j >= 0 and array[j] > key_item:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = key_item

    return array
```

Figure 4 Insertion sort python implementation

Results:

After running the function for each different size unsorted arrays and for each array repeating the sorting process for 3 times and selecting the best time result we got the following results:

Time results of insertion_sort algorithm

Array Length	Time
100	0.001842
1000	0.114144
2000	0.44383
3000	1.027821
4000	1.853004
5000	2.857569
6000	4.109781
7000	5.470225
8000	7.309627
9000	9.348465
10000	11.544728

Figure 5 Insertion sort results

Noticeably, the insertion sort implementation took around 17 fewer seconds than the bubble sort implementation to sort the same array. Despite both being O(n^2) algorithms, insertion sort proves to be more efficient.

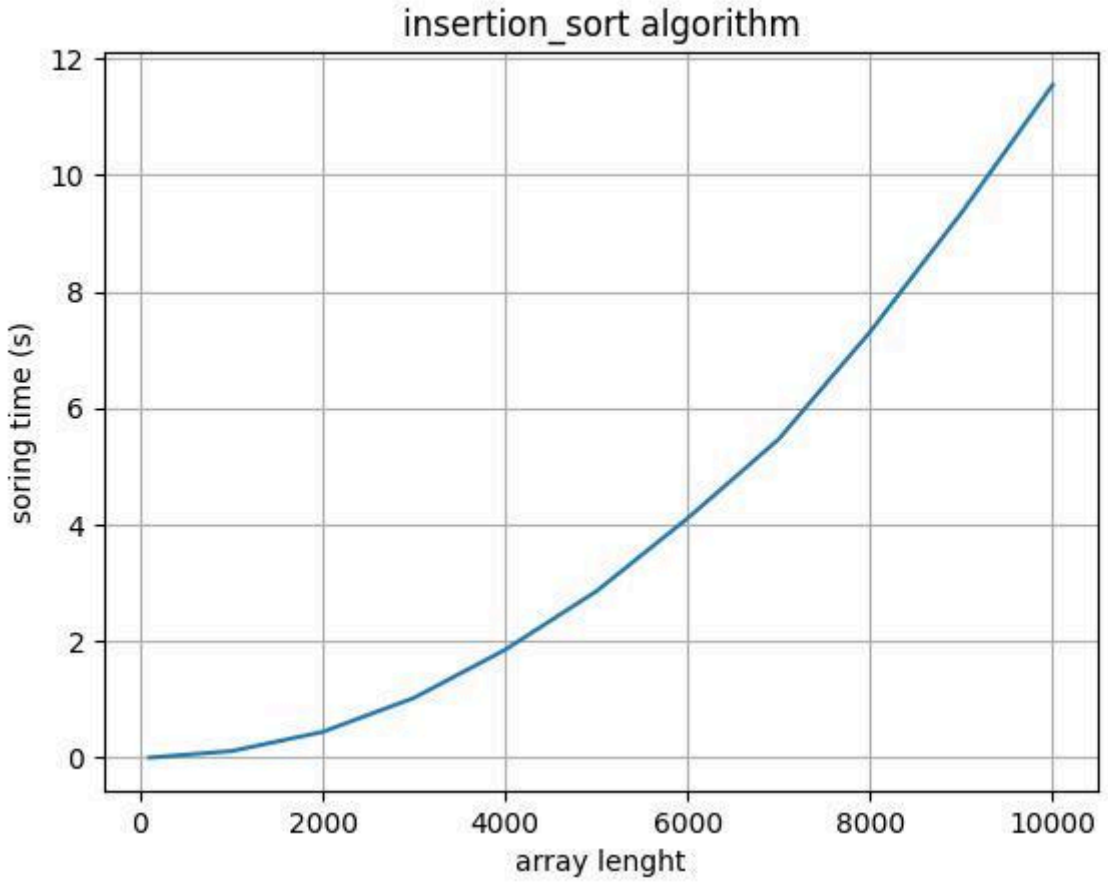


Figure 6 Graph of Insertion sort algorithm

Similar to bubble sort, the insertion sort algorithm is highly uncomplicated to implement. Despite being an O(n^2) algorithm, insertion sort demonstrates greater efficiency in practical applications compared to other quadratic implementations like bubble sort. However, it is noteworthy that insertion sort becomes impractical for large arrays, prompting consideration of alternative algorithms that can scale more efficiently.

Merge sort:

Merge sort stands out as a highly efficient sorting algorithm, leveraging the divide-and-conquer approach—an influential algorithmic technique for addressing intricate problems. A comprehensive grasp of divide and conquer requires prior familiarity with the concept of recursion. Recursion involves breaking down a problem into smaller, more manageable subproblems, often expressed in programming through a function calling itself.

Divide-and-conquer algorithms typically adhere to a structured process:

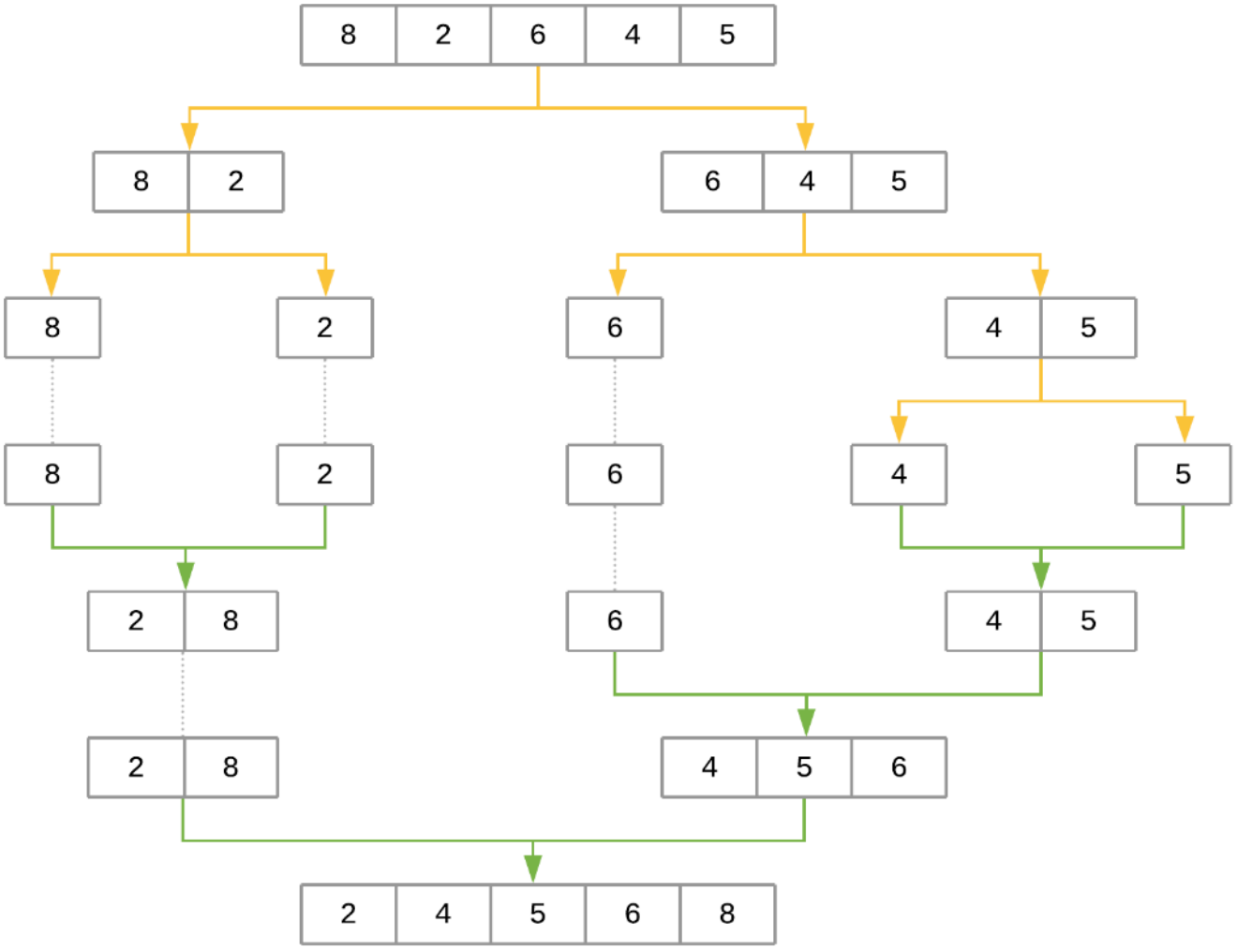
- The original input is subdivided into multiple parts, each representing a subproblem akin to the original but simpler.

- Each subproblem is addressed recursively.
- The solutions to all subproblems are amalgamated into a singular comprehensive solution.

In the context of merge sort, the divide-and-conquer strategy involves splitting the set of input values into two equal-sized parts, sorting each half recursively, and eventually merging these two sorted parts into a unified and sorted list. This method contributes to the efficiency and effectiveness of merge sort as a sorting algorithm.

Algorithm Description:

Take a look at a representation of the steps that merge sort will take to sort the array [8, 2, 6, 4, 5]:



Implementation:

```python
def merge(left, right):
    if len(left) == 0:
        return right
    if len(right) == 0:
        return left

    result = []
    index_left = index_right = 0

    while len(result) < len(left) + len(right):
        if left[index_left] <= right[index_right]:
            result.append(left[index_left])
            index_left += 1
        else:
            result.append(right[index_right])
            index_right += 1

        if index_right == len(right):
            result += left[index_left:]
            break

        if index_left == len(left):
            result += right[index_right:]
            break

    return result

def merge_sort(array):
    if len(array) < 2:
        return array

    midpoint = len(array) // 2
    return merge(
        left=merge_sort(array[:midpoint]),
        right=merge_sort(array[midpoint:]))
```

Figure 7 Merge sort python implementation

Results:

After running the function for each different size unsorted arrays and for each array repeating the sorting process for 3 times and selecting the best time result we got the following results:

Time results of merge_sort algorithm

Array Length	Time
100	0.000891
1000	0.013099
2000	0.029738
3000	0.046269
4000	0.065632
5000	0.081914
6000	0.104187
7000	0.119572
8000	0.13897
9000	0.160496
10000	0.183175
15000	0.285775
20000	0.391403
30000	0.608206
40000	0.83725
50000	1.067593
75000	1.67619
100000	2.266633

Figure 8 Results for Merge sort

Compared to bubble sort and insertion sort, the merge sort implementation is extremely fast, sorting the ten-thousand-element array in less than a second!

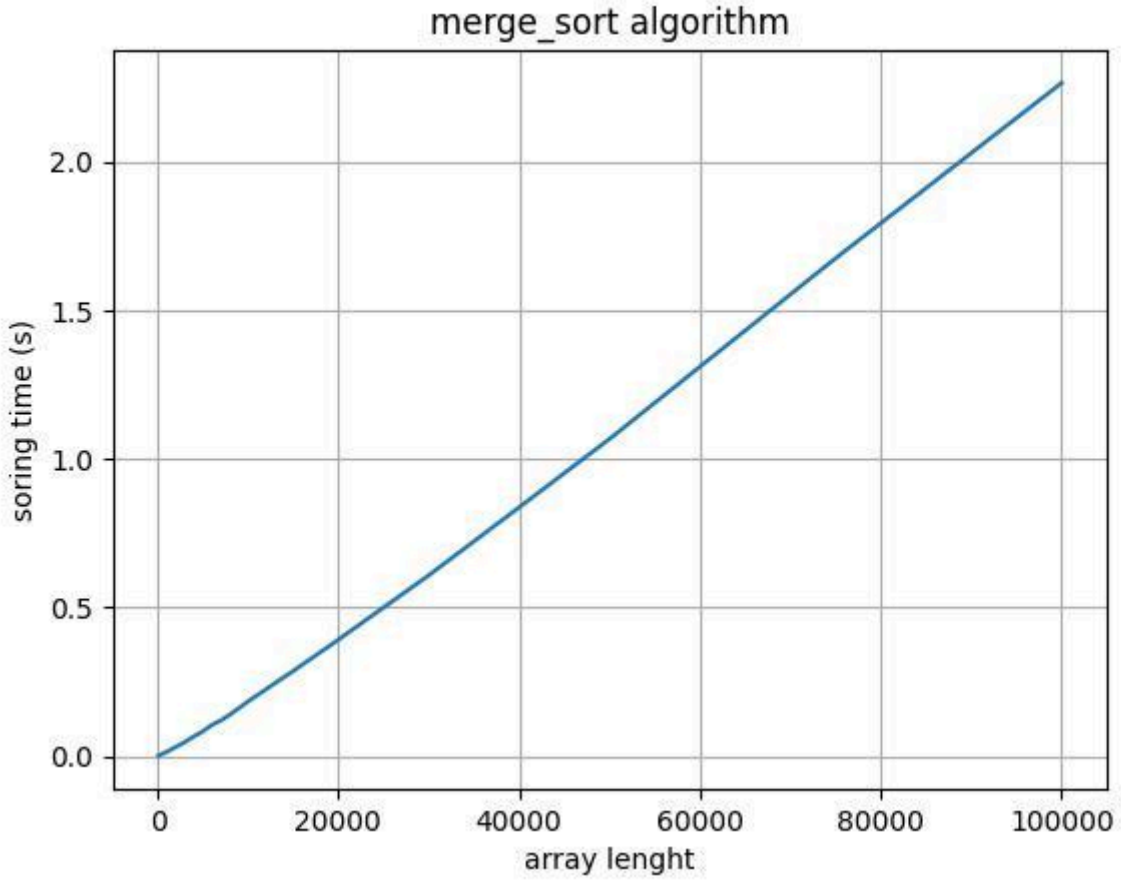


Figure 9 Graph of Merge sort algorithm

With a commendable runtime complexity of O(n log2n), merge sort stands out as a highly efficient algorithm, showcasing scalability as the size of the input array increases. Notably, its design facilitates parallelization, as it breaks the input array into chunks that can be distributed and processed concurrently if needed. However, a notable drawback of merge sort is its tendency to generate copies of the array during recursive calls. Additionally, the creation of a new list inside the merge() function, intended to sort and return both input halves, contributes to increased memory usage. This contrasts with bubble sort and insertion sort, which have the capability to sort the list in place, resulting in a more frugal use of memory. Given this memory-intensive nature, it might be prudent to reconsider using merge sort for sorting large lists, particularly in scenarios with memory-constrained hardware.

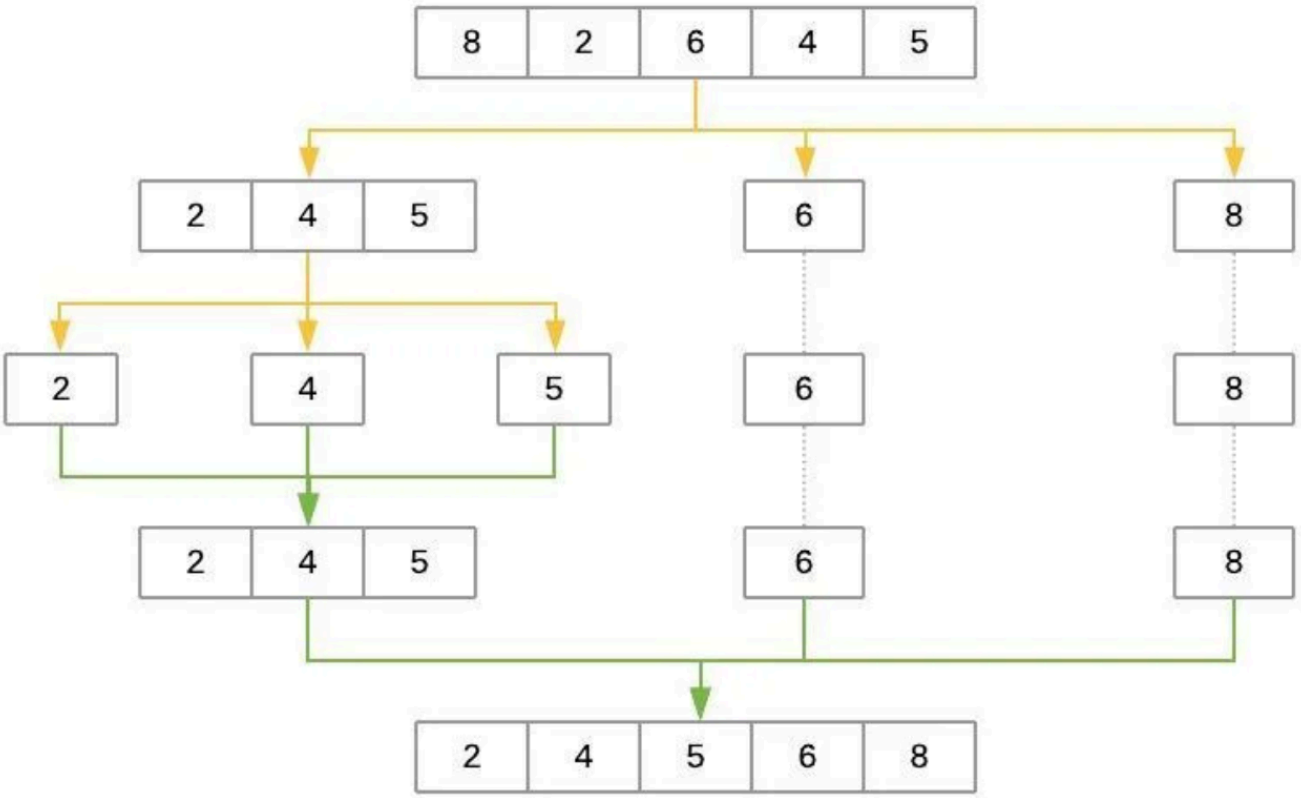
Quick sort:

Similar to merge sort, the Quicksort algorithm embraces the divide-and-conquer principle to segment the input array into two lists: one containing small items and the other containing large items. The algorithm systematically sorts both lists through recursive operations until the final list is entirely sorted. Central to Quicksort is the process of partitioning the input list. The algorithm begins by selecting

a pivot element and organizes the list around this pivot, placing smaller elements into a low array and larger elements into a high array. By arranging every element from the low list to the left of the pivot and every element from the high list to the right, the pivot is precisely positioned in the final sorted list. Consequently, the function can iteratively apply the same procedure to the low and high arrays until the entire list achieves a sorted state. This distinctive approach makes Quicksort an effective sorting algorithm, leveraging partitioning and recursion to attain efficient and rapid sorting.

Algorithm Description:

Here's an illustration of the steps that Quicksort takes to sort the array [8, 2, 6, 4, 5]:



Implementation:

```
def quick_sort(array):
    if len(array) < 2:
        return array

    low, same, high = [], [], []
    pivot = array[randint(0, len(array) - 1)]

    for item in array:
        if item < pivot:
            low.append(item)
        elif item == pivot:
            same.append(item)
        elif item > pivot:
            high.append(item)
    return quick_sort(low) + same + quick_sort(high)
```

Figure 10 Quicksort python implementation

Results:

After running the function for each different size unsorted arrays and for each array repeating the sorting process for 3 times and selecting the best time result we got the following results:

Time results of quick_sort algorithm

Array Length	Time
100	0.001209
1000	0.007903
2000	0.014927
3000	0.02255
4000	0.031151
5000	0.037916
6000	0.042836
7000	0.045688
8000	0.052651
9000	0.05791
10000	0.063932
15000	0.089822
20000	0.115147
30000	0.166223
40000	0.219578
50000	0.284961
75000	0.426808
100000	0.562572

Figure 11 Results for Quicksort

Comparing with the result of merge sort we easily can see that quick sort is much faster, especially on big length arrays

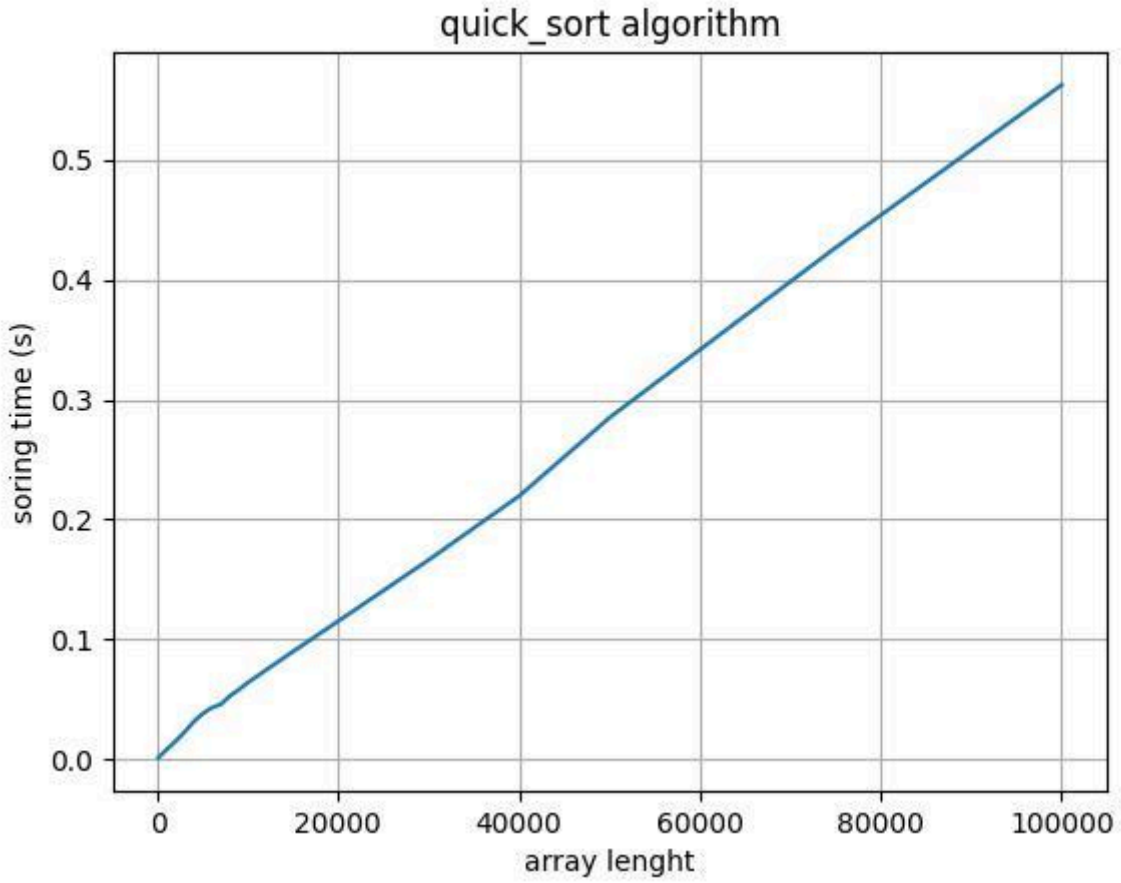


Figure 12 Graph of Quicksort algorithm

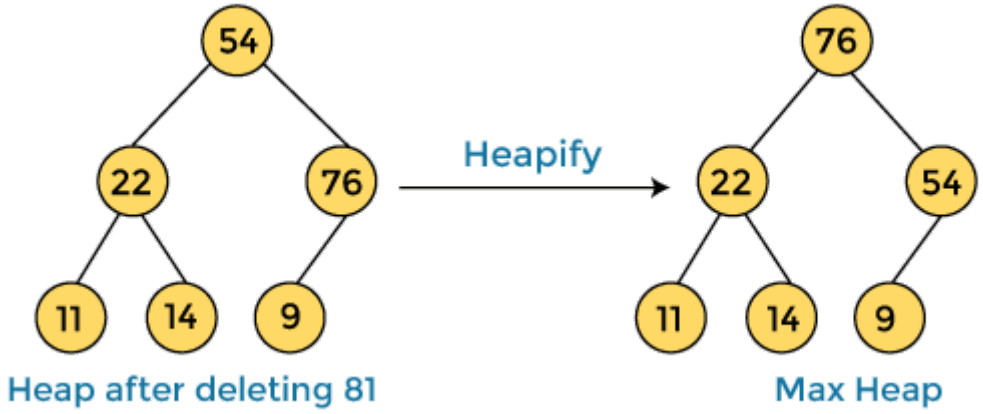
True to its name, Quicksort lives up to its reputation for speed. Despite its theoretical worst-case scenario being O(n^2), well-implemented Quicksort consistently outperforms many other sorting algorithms in practice. Additionally, akin to merge sort, Quicksort boasts simplicity in parallelization. However, a notable drawback of Quicksort is the absence of a guaranteed average runtime complexity. While worst-case scenarios are infrequent, certain applications cannot afford to risk potential poor performance, prompting a preference for algorithms that consistently maintain O(n log2n) efficiency regardless of the input. Similar to merge sort, Quicksort also engages in a trade-off, sacrificing memory space for speed. This trade-off may pose limitations when sorting large lists, making it a factor to consider in scenarios where memory constraints are a significant concern.

Heap sort:

Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantages of very simple implementation and a more favorable worst-case O(n log n) runtime.

Most real-world quicksort variants include an implementation of heapsort as a fallback should they detect that quicksort is becoming degenerate. Heapsort is an in-place algorithm, but it is not a stable sort.

Algorithm Description:



Implementation:

```
def heap_sort(array):
    def heapify(array, N, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2

        if l < N and array[largest] < array[l]:
            largest = l
        if r < N and array[largest] < array[r]:
            largest = r
        if largest != i:
            array[i], array[largest] = array[largest], array[i]
            heapify(array, N, largest)

    N = len(array)
    for i in range(N // 2 - 1, -1, -1):
        heapify(array, N, i)
    for i in range(N - 1, 0, -1):
        array[i], array[0] = array[0], array[i]
        heapify(array, i, 0)
    return array
```

Figure 13 Heap sort python implementation

Results:

After running the function for each different size unsorted arrays and for each array repeating the sorting process for 3 times and selecting the best time result we got the following results:

Time results of heap_sort algorithm

Array Length	Time
100	0.001577
1000	0.015977
2000	0.034074
3000	0.05238
4000	0.072966
5000	0.0972
6000	0.114356
7000	0.138958
8000	0.158746
9000	0.183545
10000	0.204559
15000	0.337519
20000	0.444398
30000	0.698253
40000	0.953959
50000	1.21879
75000	1.901548
100000	2.617234

Figure 14 Results for Heap sort

We may observe excellent results but with some seconds behind the quick sort benchmark.

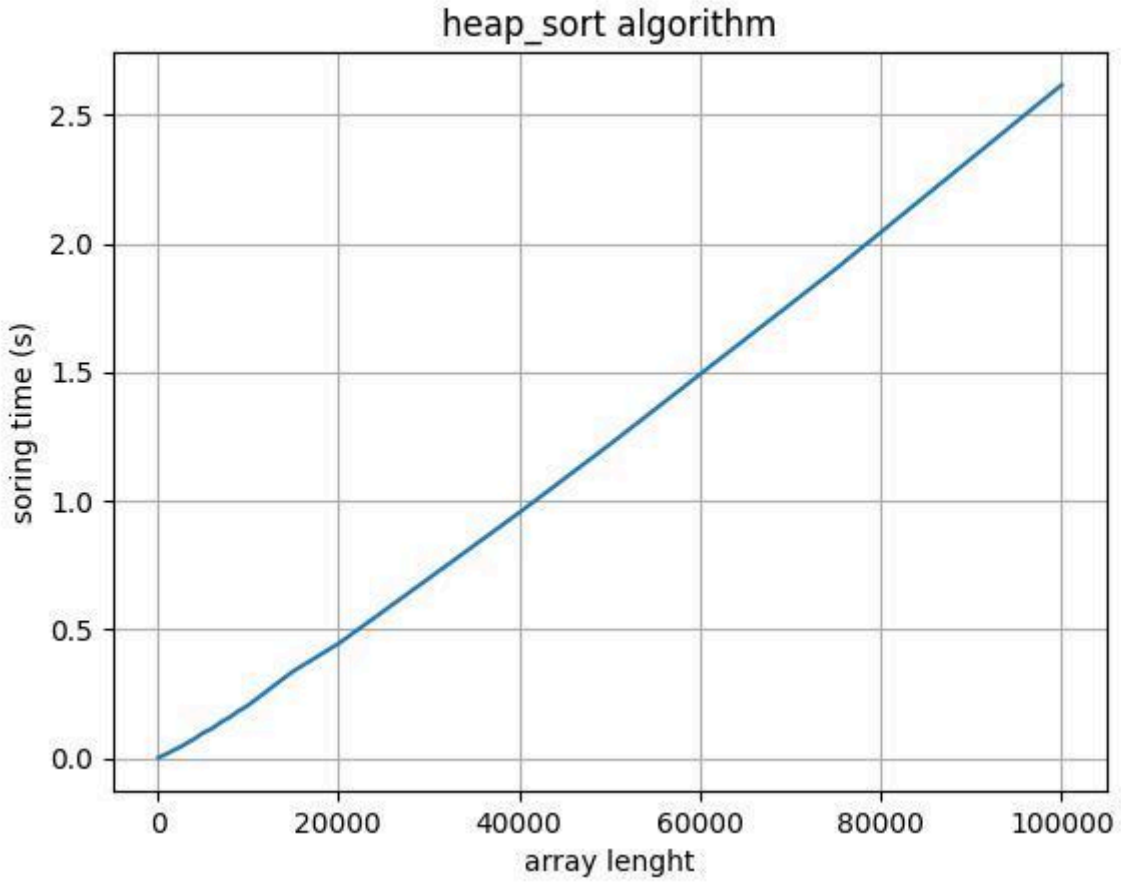


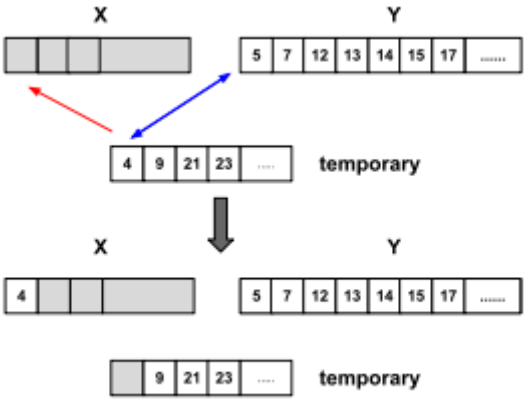
Figure 15 Graph of Heap sort algorithm

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, and of the theoretical lower bound on comparison sorts. While it cannot do better than O(n log n) for pre-sorted inputs, it does not suffer from quicksort's O(n2) worst case, either.

Timsort:

The Timsort algorithm holds a distinct position as a hybrid sorting algorithm, integrating the strengths of both insertion sort and merge sort. This algorithm is particularly cherished within the Python community as it was introduced by Tim Peters in 2002 to serve as the standard sorting algorithm for the Python language. Timsort distinguishes itself by capitalizing on the presence of already-sorted elements, commonly found in real-world datasets and referred to as natural runs. The algorithm systematically traverses the list, gathering elements into runs, and subsequently merges these runs to construct a singular, sorted list. This approach allows Timsort to harness the benefits of pre-existing order in the data, contributing to its efficiency and performance in various applications.

Algorithm Description:



Implementation:

```python
def tim_sort(array):
    def insertion_sort_modified(array, left=0, right=None):
        if right is None:
            right = len(array) - 1
        for i in range(left + 1, right + 1):
            key_item = array[i]
            j = i - 1
            while j >= left and array[j] > key_item:
                array[j + 1] = array[j]
                j -= 1
            array[j + 1] = key_item
        return array
    min_run = 32
    n = len(array)

    for i in range(0, n, min_run):
        insertion_sort_modified(array, i, min((i + min_run - 1), n - 1))

    size = min_run
    while size < n:
        for start in range(0, n, size * 2):
            midpoint = start + size - 1
            end = min((start + size * 2 - 1), (n-1))

            merged_array = merge(
                left=array[start:midpoint + 1],
                right=array[midpoint + 1:end + 1])

            array[start:start + len(merged_array)] = merged_array

        size *= 2
    return array
```

Figure 16 Timsort python implementation

Results:

After running the function for each different size unsorted arrays and for each array repeating the sorting process for 3 times and selecting the best time result we got the following results:

Time results of tim_sort algorithm

Array Length	Time
100	0.00124
1000	0.010767
2000	0.024382
3000	0.03905
4000	0.052332
5000	0.073925
6000	0.090614
7000	0.103323
8000	0.121511
9000	0.145446
10000	0.154277
15000	0.244941
20000	0.341407
30000	0.536275
40000	0.738969
50000	0.942192
75000	1.51311
100000	2.020925

Figure 17 Results for Timsort

Noticeably, Timsort exhibits a notable advantage by amalgamating two algorithms, namely insertion sort and merge sort, which may be comparatively slower when employed individually. The brilliance of Timsort lies in the strategic combination of these algorithms, leveraging their respective strengths to produce impressive and efficient results. This amalgamation allows Timsort to harness the efficiency of insertion sort in dealing with small, partially sorted sequences, while capitalizing on the superior performance of merge sort in handling larger datasets. The synergistic integration of these algorithms contributes to Timsort's effectiveness and versatility in sorting a wide range of datasets.

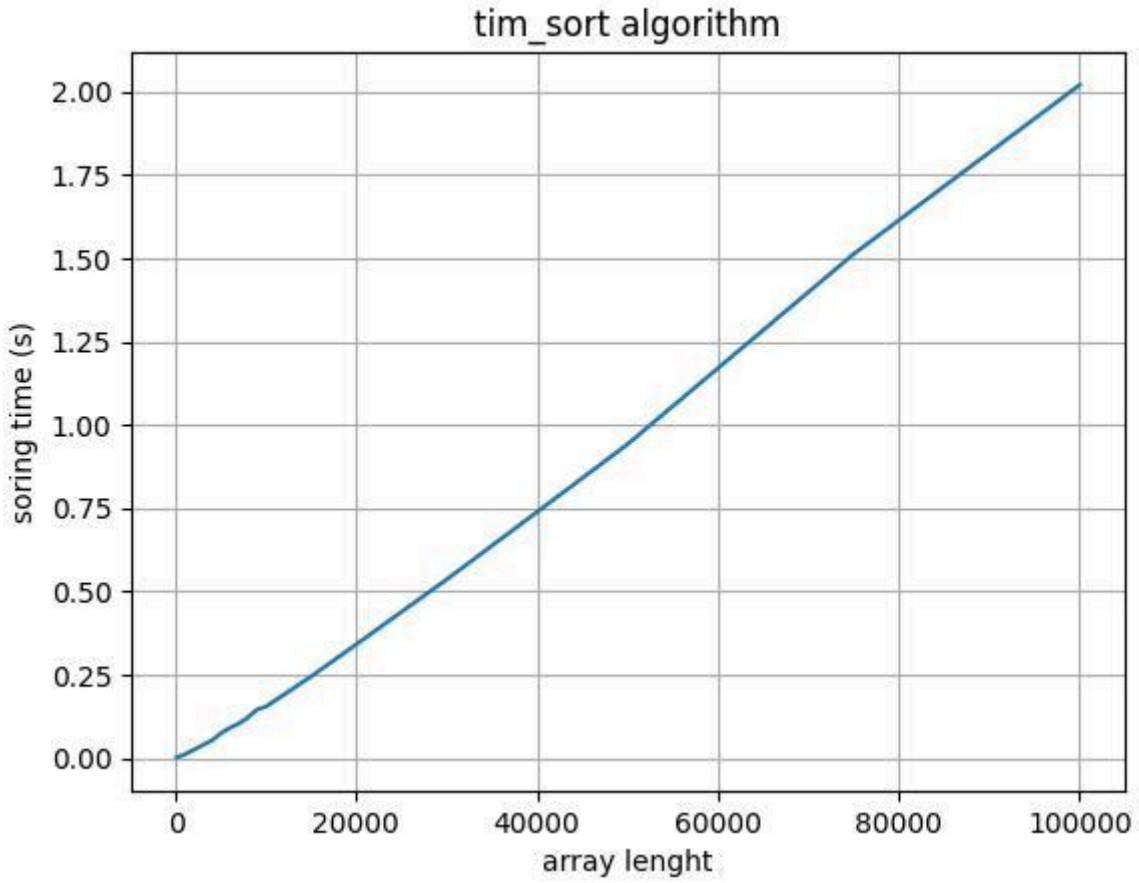


Figure 18 Graph of Timsort algorithm

The primary drawback of Timsort lies in its complexity. Even with the implementation of a simplified version of the original algorithm, it necessitates a more extensive codebase due to its reliance on both insertion_sort() and merge() functions. On the positive side, Timsort boasts the significant advantage of predictable performance, consistently achieving O(n log2n) regardless of the input array's structure. This stands in contrast to Quicksort, which can potentially degrade to O(n^2). Timsort further excels in handling small arrays, seamlessly transitioning into a single insertion sort for optimal efficiency. In practical, real-world scenarios where datasets often exhibit some preexisting order, Timsort emerges as an excellent choice. Its adaptability and ability to efficiently handle arrays of varying lengths make it a compelling option for a wide range of sorting applications.

CONCLUSION:

In conclusion, our empirical analysis of bubble, insertion, quick, merge, heap, and timsort algorithms has provided valuable insights into their performance characteristics. Each algorithm exhibits its own set of strengths and weaknesses, shedding light on the trade-offs inherent in algorithm design.

Bubble and insertion sort algorithms, while simple to implement, demonstrate poor performance for large datasets due to their quadratic time complexity. Quick sort, on the other hand, stands out for its efficiency in average and best-case scenarios with linearithmic time complexity. However, it can degrade to quadratic time complexity in the worst-case scenario, emphasizing the importance of understanding the nature of the input data.

Merge sort exhibits consistent performance with linearithmic time complexity, making it a reliable choice for various scenarios. Heap sort, although less intuitive, showcases efficient sorting with linearithmic time complexity as well and has the added advantage of being an in-place algorithm.

Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, demonstrated superior performance in our analysis. Its adaptability to varying input sizes and data distributions makes it a robust choice for practical applications.

In summary, while no single algorithm is universally superior, our comparative analysis has highlighted the importance of considering the characteristics of the dataset and the specific requirements of the application when selecting a sorting algorithm. The choice between simplicity and efficiency depends on the context, and understanding the trade-offs between time and space complexity is crucial for making informed decisions in algorithm design and implementation.