Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

REPORT

Laboratory work no.2
Sorting Algorithms

Elaborated:
st. gr. FAF-213 Botnari Ciprian

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

GITHUB

Sufferal/Algorithms: Analysis of algorithms (github.com)

ALGORITHM ANALYSIS

Objective

Study and analyze different algorithms for sorting.

Tasks

1. Implement at least 3 algorithms for determining Fibonacci n-th term;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical notes

An alternative approach to evaluating the complexity of an algorithm is through empirical analysis, which can provide insights on the complexity class of the algorithm, comparison of efficiency between algorithms solving the same problem, comparison of different implementations of the same algorithm, and performance on specific computers.

The process of empirical analysis typically involves:

1. Establishing the purpose of the analysis
2. Choosing the efficiency metric (number of operations or execution time)
3. Defining the properties of the input data
4. Implementing the algorithm in a programming language
5. Generating input data
6. Running the program with each set of data
7. Analyzing the results

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

A sorting algorithm in computer science is a method for organizing the items on a list. The most often used ordering systems are lexicographical and numerical, both in ascending and decreasing order. The effectiveness of other algorithms (like search and merge algorithms) that require input data to be in sorted lists must be maximized, hence efficient sorting is crucial. Moreover, canonicalizing data and creating output that is intelligible by humans also benefit from sorting.

Each sorting algorithm's output must formally meet the following two requirements:

- The output follows the specified order and is in monotonic order, meaning that no element is smaller or greater than the one before it.
- The result is a permutation of the input, which reorders the elements while keeping the original ones.

The sorting problem has drawn a lot of attention since the dawn of computing, perhaps as a result of how difficult it is to efficiently solve it despite its straightforward and well-known formulation. About 1951, Betty Holberton, who worked on the ENIAC and UNIVAC, was one of the authors of the earliest sorting algorithms. In 1956, bubble sort was studied. Asymptotically optimum algorithms have existed since the middle of the 20th century; nevertheless, new ones are constantly being developed. The widely used Timsort was created in 2002, while the library sort was introduced in 2006.

A fundamental criterion for comparison sorting algorithms is (n log n) comparisons (some input sequences will require a multiple of n log n comparisons, where n is the number of elements in the array to be sorted). Algorithms without a comparison foundation, like the counting sort, may perform better.

The prevalence of sorting algorithms in introductory computer science courses serves as a gentle introduction to a number of key algorithm concepts, including big O notation, divide-and-conquer algorithms, data structures like heaps and binary trees, randomized algorithms, best, worst, and average case analysis, time-space tradeoffs, and upper and lower bounds.

The best way to sort small arrays quickly (i.e., taking into account machine-specific characteristics) or ideally (in the fewest comparisons and swaps) is still an open research question, with solutions only being known for very small arrays (20 elements). Similarly optimal (by various definitions) sorting on a parallel machine is an open research topic.

Some examples of sorting algorithms are:

1. **Quick sort**
2. **Merge sort**
3. **Heap sort**
4. **Intro sort**

Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each sorting algorithm T(n) for each length of list.

Input Format

As input, each sorting algorithm will receive multiple lists with different number of elements, their length being:

- 10
- 100
- 1000
- 10000
- 25000
- 50000
- 100000
- 150000

IMPLEMENTATION

All four sorting algorithms will be put into practice in Python and objectively evaluated depending on how long it takes to complete each one. The particular efficiency in rapport with input will vary based on the memory of the device utilized, even though the overall trend of the results may be similar to other experimental data. As determined by experimental measurement, the error margin will be a few seconds.

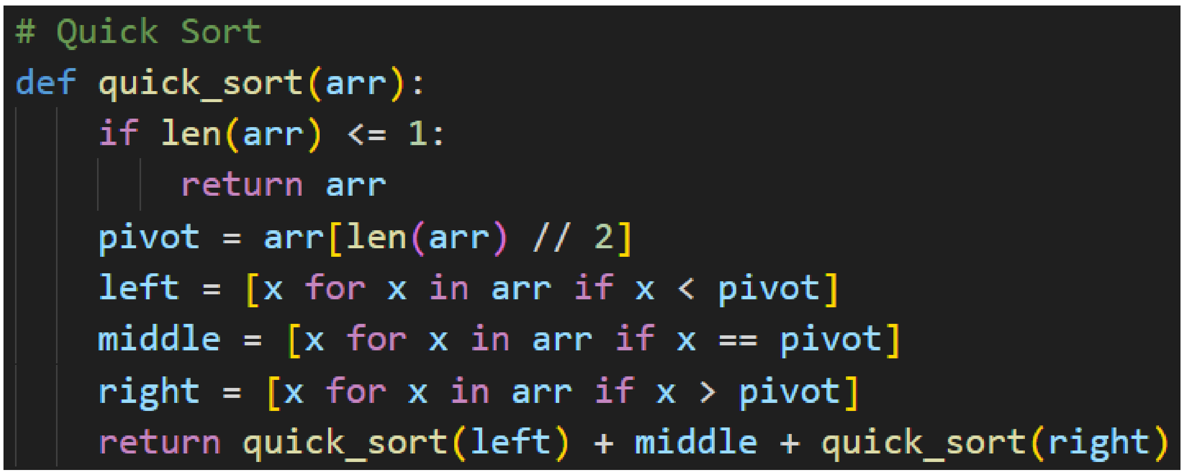
Quick sort

An effective, all-purpose sorting algorithm is quicksort. Tony Hoare, a British computer scientist, created the Quicksort algorithm in 1959, which was later published in 1961 and is still a widely used sorting algorithm today. With randomized data, it is generally somewhat quicker than merge sort and heapsort, especially for bigger distributions.

Algorithm Description:

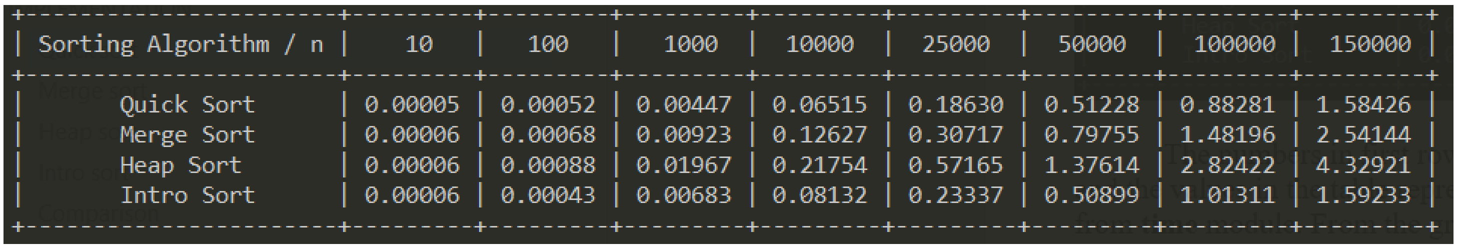
Quicksort is a divide-and-conquer algorithm. It operates by choosing one element from the array to serve as the "pivot," and then dividing the remaining elements into two sub-arrays based on whether they are less than or greater than the pivot. It is referred to as partition-exchange sort for this reason. After that, the sub-arrays are recursively sorted. The sorting can be done in-place and only needs a small amount of extra memory.

Implementation



```python
# Quick Sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

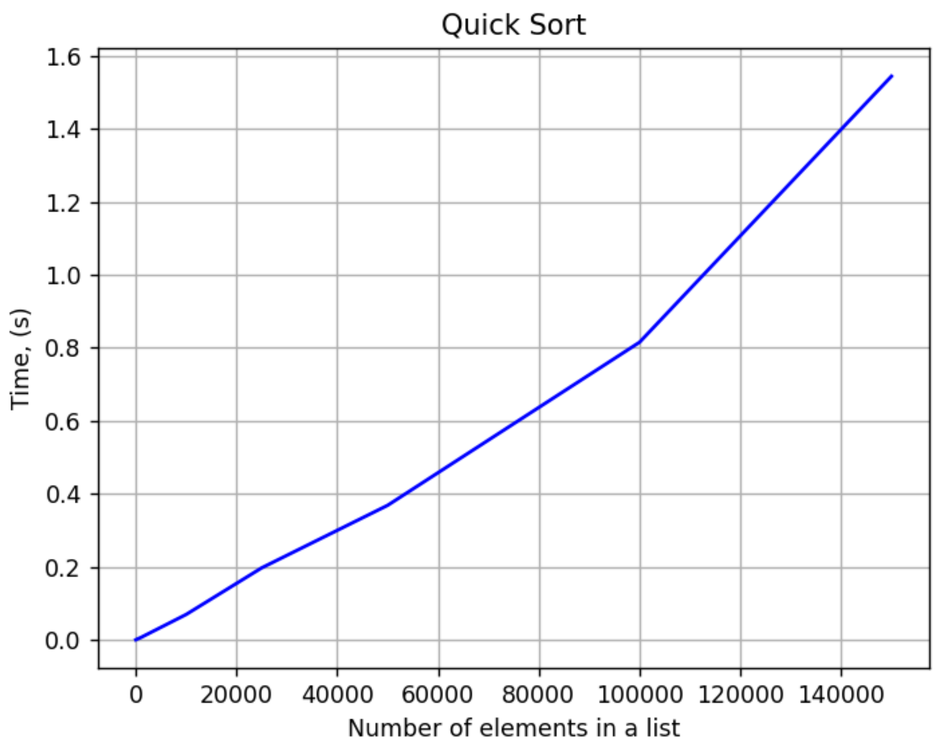
Results



```
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
| Sorting Algorithm / n |    10   |   100   |  1000   |  10000  |  25000  |  50000  |  100000 |  150000 |
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
|       Quick Sort        | 0.00005 | 0.00052 | 0.00447 | 0.06515 | 0.18630 | 0.51228 | 0.88281 | 1.58426 |
|       Merge Sort        | 0.00006 | 0.00068 | 0.00923 | 0.12627 | 0.30717 | 0.79755 | 1.48196 | 2.54144 |
|       Heap Sort         | 0.00006 | 0.00088 | 0.01967 | 0.21754 | 0.57165 | 1.37614 | 2.82422 | 4.32921 |
|       Intro Sort        | 0.00006 | 0.00043 | 0.00683 | 0.08132 | 0.23337 | 0.50899 | 1.01311 | 1.59233 |
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. From the graph below we could deduce that the average time complexity is T(n log n), which is faster than usual T(n^2), but it's worth noting that the worst case theoretically is actually T(n^2).

Graph



Merge sort

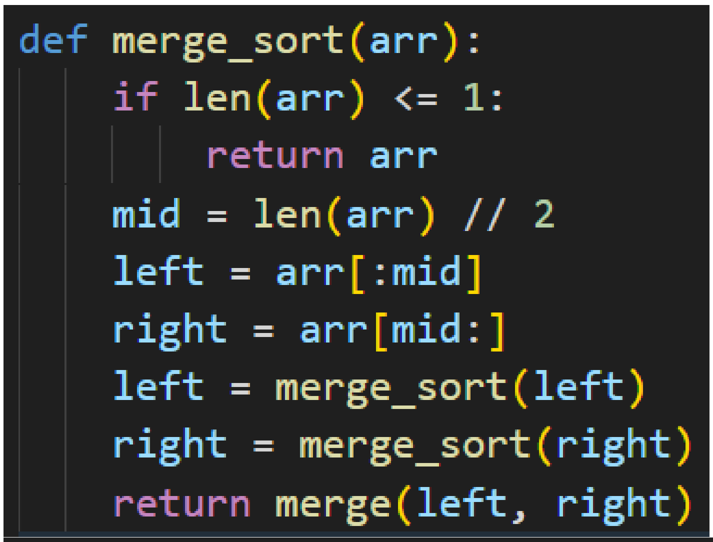
Merge sort, which is also frequently spelled mergesort, is a general-purpose, effective sorting algorithm used in computer science. The order of equal elements is the same in the input and output for the majority of implementations, which results in a stable sort. John von Neumann developed the divide-and-conquer algorithm known as merge sort in 1945. In a report published in 1948 by Goldstine and von Neumann, bottom-up merge sort was thoroughly described and analyzed.

Algorithm Description:

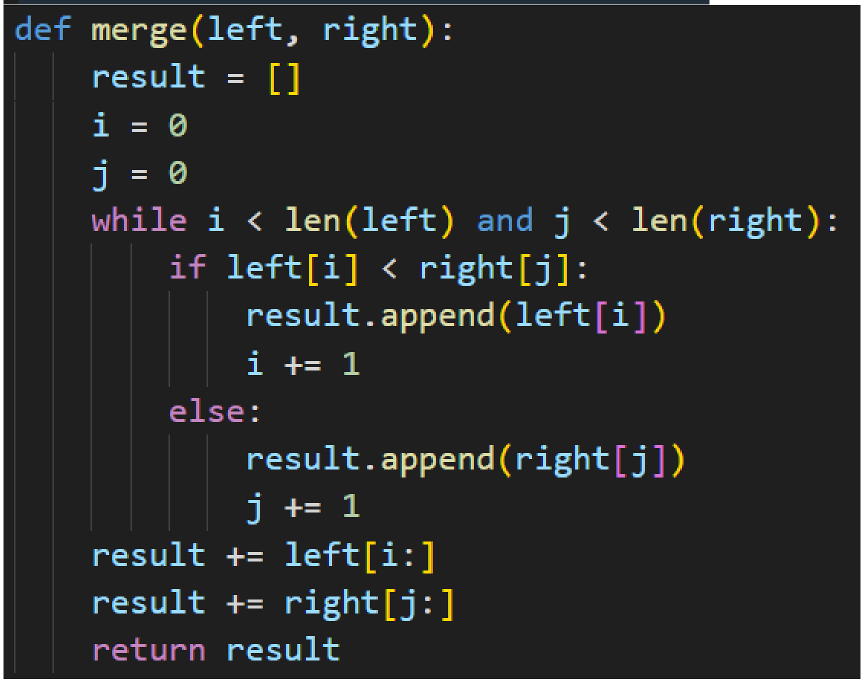
Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

Implementation

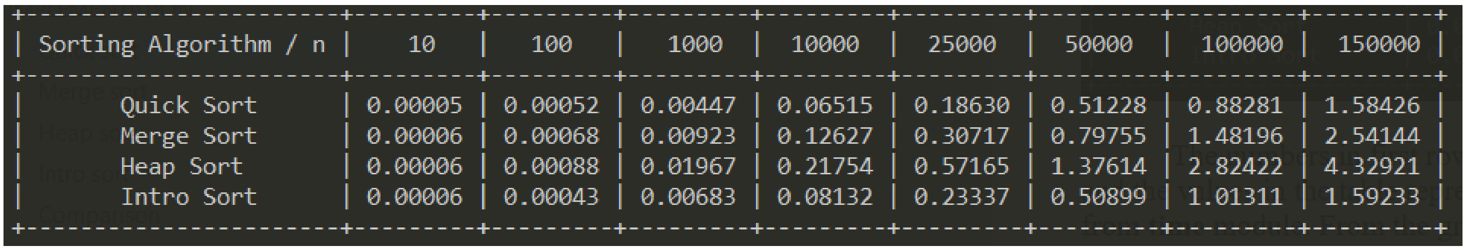


```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)
```



```python
def merge(left, right):
    result = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result
```

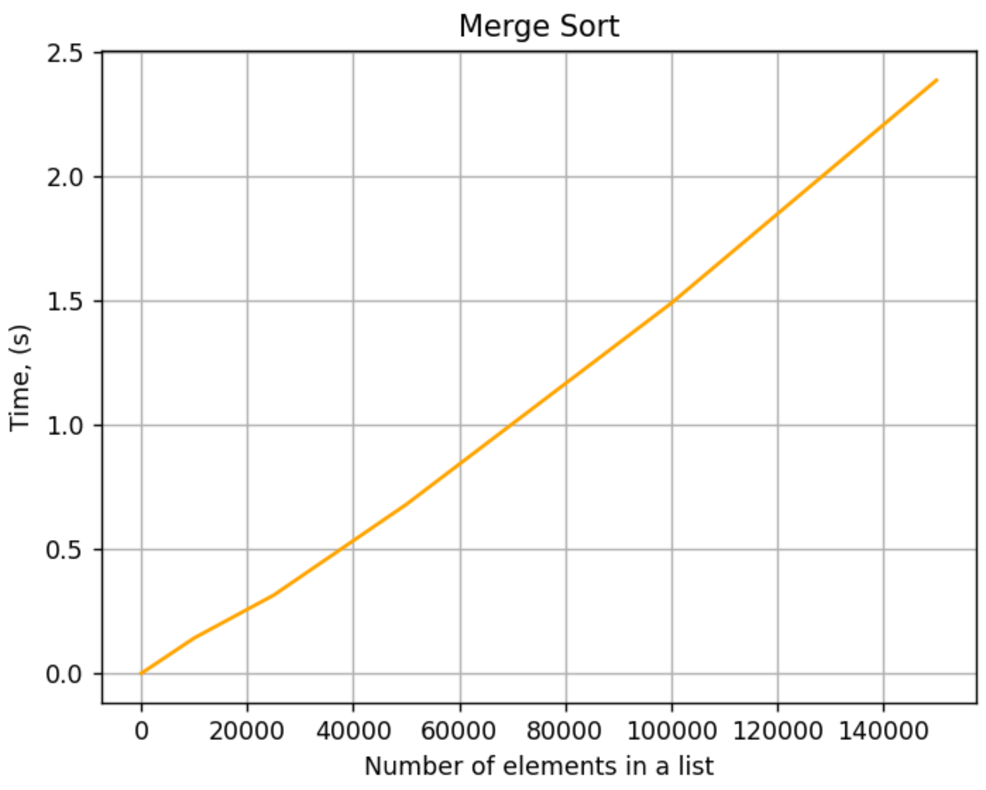
Results



```
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
| Sorting Algorithm / n |   10    |   100   |  1000   |  10000  |  25000  |  50000  | 100000  | 150000  |
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
|      Quick Sort       | 0.00005 | 0.00052 | 0.00447 | 0.06515 | 0.18630 | 0.51228 | 0.88281 | 1.58426 |
|      Merge Sort       | 0.00006 | 0.00068 | 0.00923 | 0.12627 | 0.30717 | 0.79755 | 1.48196 | 2.54144 |
|      Heap Sort        | 0.00006 | 0.00088 | 0.01967 | 0.21754 | 0.57165 | 1.37614 | 2.82422 | 4.32921 |
|      Intro Sort       | 0.00006 | 0.00043 | 0.00683 | 0.08132 | 0.23337 | 0.50899 | 1.01311 | 1.59233 |
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. From the graph below we could deduce that the time complexity is T(n), which is slower than quick sort, but it’s worth noting that the worst case theoretically is better than quick sort as it is T(n log n).

Graph



Heap sort

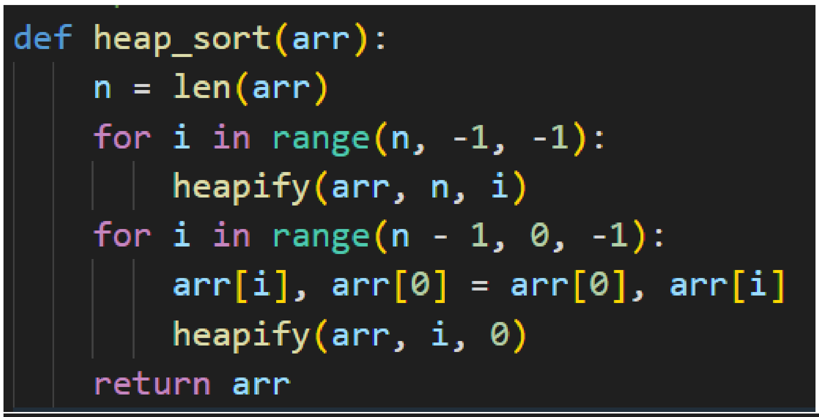
Heapsort is a comparison-based sorting algorithm used in computer science. Heapsort can be compared to a better version of selection sort in that it iteratively reduces the size of the unsorted region by extracting the largest element from it and inserting it into the sorted region. Like selection sort, heapsort divides its input into sorted and unsorted regions. In contrast to selection sort, heapsort keeps the unsorted region in a heap data structure to more quickly find the largest element at each step rather than wasting time with a linear-time scan of the unsorted region.

Algorithm Description:

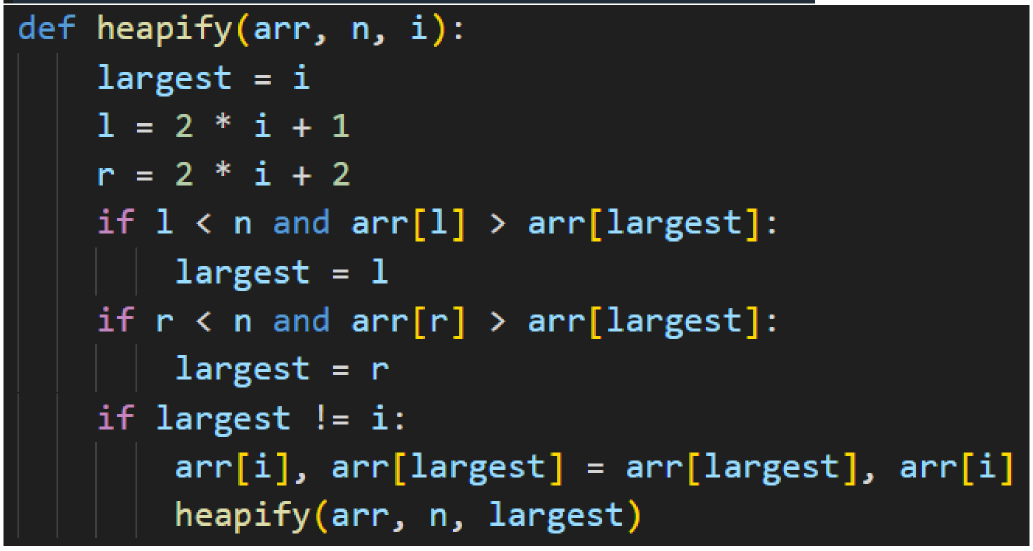
Here's a step-by-step algorithm description for Heap Sort:

1. Build a heap from the input array. To do this, start from the middle of the array and work your way up to the beginning, calling the heapify() function on each element. heapify() ensures that the max-heap property is maintained by swapping the element with its children as needed.
2. Extract the maximum element from the heap and place it at the end of the sorted portion of the array. To do this, swap the first element (which is the maximum element in a max-heap) with the last element of the heap, decrement the heap size, and call heapify() on the first element to restore the max-heap property.
3. Repeat step 2 until the heap is empty. At each iteration, the maximum element is placed at the end of the sorted portion of the array.

Implementation

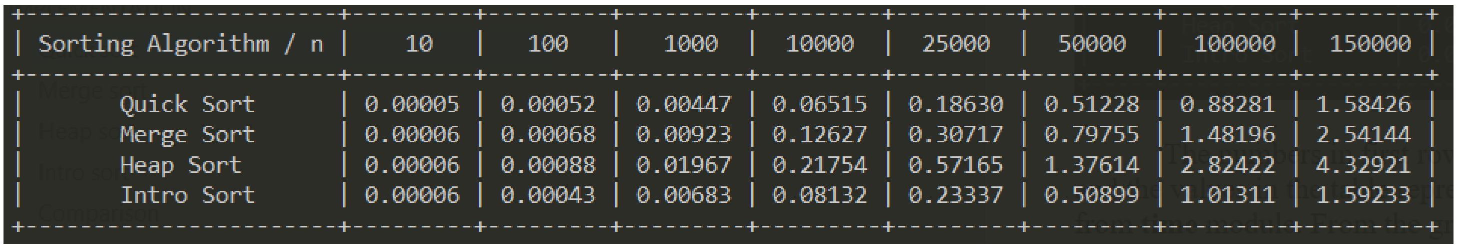


```python
def heap_sort(arr):
    n = len(arr)
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
```



```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

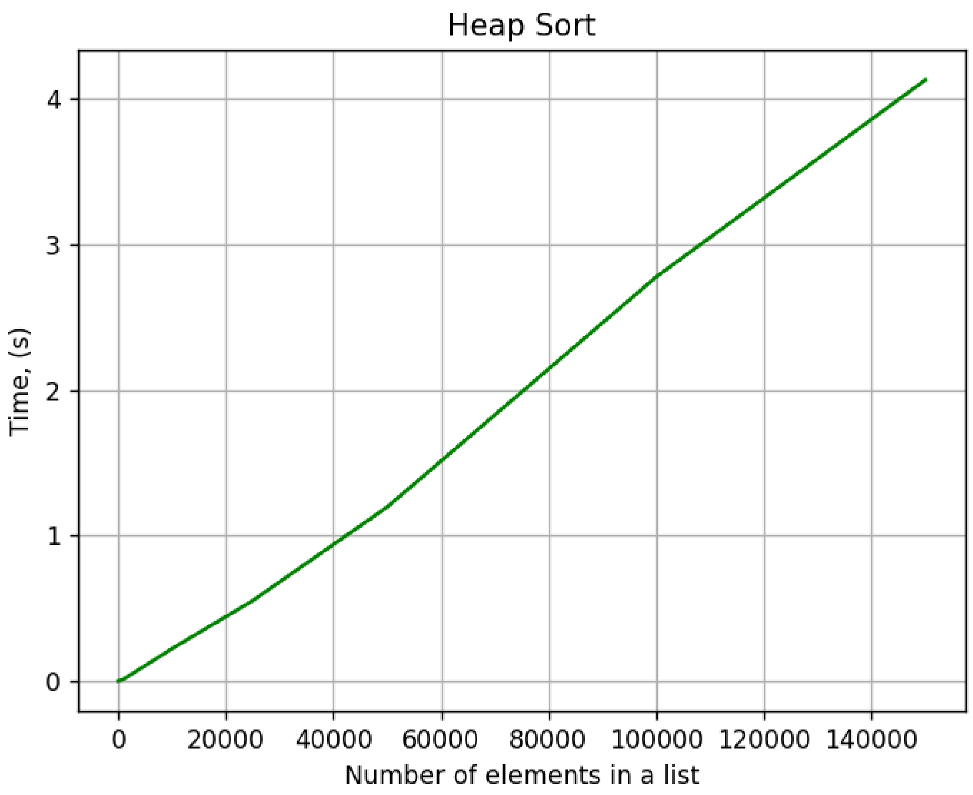
Results



```
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
| Sorting Algorithm / n |    10   |   100   |  1000   |  10000  |  25000  |  50000  | 100000  | 150000  |
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
|       Quick Sort        | 0.00005 | 0.00052 | 0.00447 | 0.06515 | 0.18630 | 0.51228 | 0.88281 | 1.58426 |
|       Merge Sort        | 0.00006 | 0.00068 | 0.00923 | 0.12627 | 0.30717 | 0.79755 | 1.48196 | 2.54144 |
|       Heap Sort         | 0.00006 | 0.00088 | 0.01967 | 0.21754 | 0.57165 | 1.37614 | 2.82422 | 4.32921 |
|       Intro Sort        | 0.00006 | 0.00043 | 0.00683 | 0.08132 | 0.23337 | 0.50899 | 1.01311 | 1.59233 |
+-------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. From the graph below we could deduce that the time complexity is T(n), which is similar to merge sort but still slower judging by the time obtained, but it's worth noting that the worst case theoretically is better than quick sort as it is T(n log n).

Graph



Intro sort

Introsort was invented by David Musser in Musser (1997), in which he also introduced introselect, a hybrid selection algorithm based on quickselect (a variant of quicksort), which falls back to median of medians and thus provides worst-case linear complexity, which is optimal. Both algorithms were introduced with the purpose of providing generic algorithms for the C++ Standard Library which had both fast average performance and optimal worst-case performance, thus allowing the performance requirements to be tightened. Introsort is in place and not stable.

Algorithm Description:

Here's a step-by-step algorithm description for Intro Sort:

1. The algorithm begins by first checking if the input list is empty or has only one element. If either of these conditions is true, then the list is already sorted, and the algorithm returns the list.

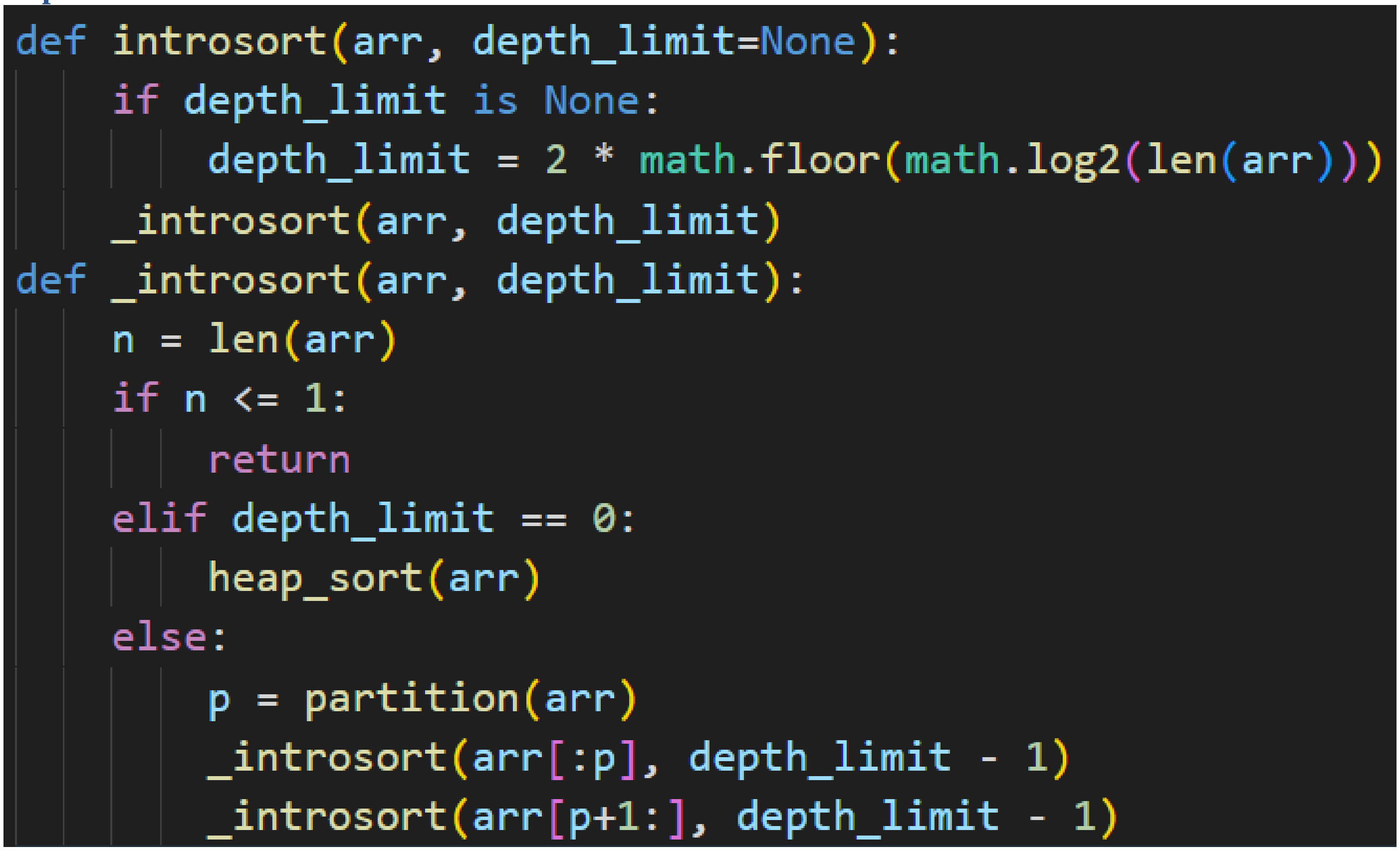
2. If the list has more than one element, the algorithm proceeds to partition the list using Quick Sort. The partitioning selects a pivot element from the list and arranges all the elements in the list such that all the elements smaller than the pivot are on the left of the pivot, and all the elements larger than the pivot are on the right of the pivot.

3. After partitioning the list, the algorithm checks if the recursion depth is more than a threshold value. If so, the algorithm switches to Heap Sort. Heap Sort is an efficient sorting algorithm for large input sizes with a worst-case time complexity of O(n log n).

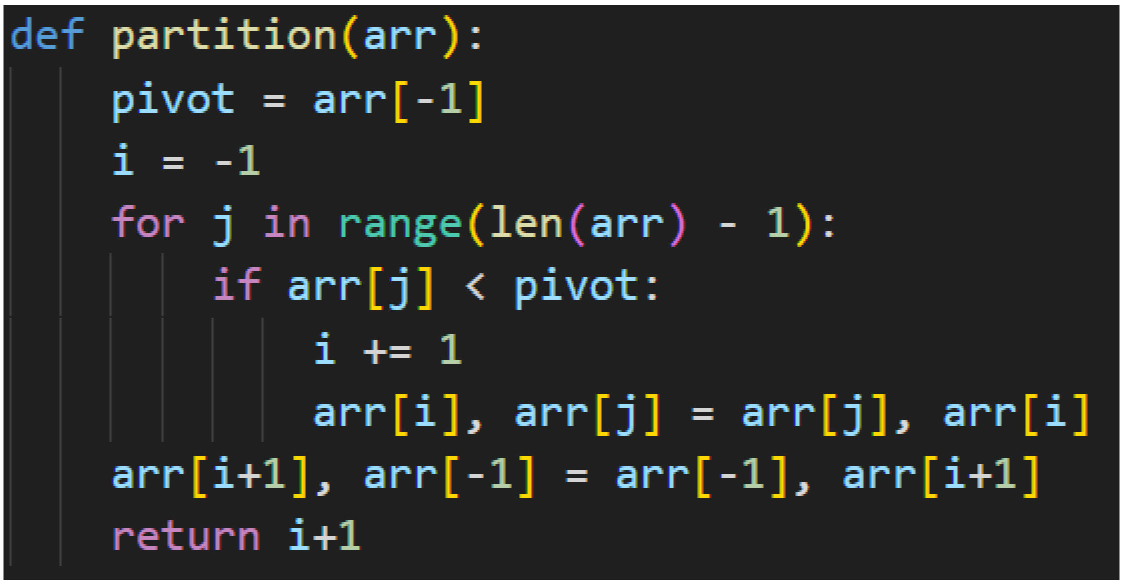
4. If the recursion depth is less than the threshold value, the algorithm continues with Quick Sort on the two partitions generated in step 2 recursively until the list is sorted.

5. Finally, if the input list is small, the algorithm switches to Insertion Sort. Insertion Sort is an efficient algorithm for small input sizes with a worst-case time complexity of O(n^2).

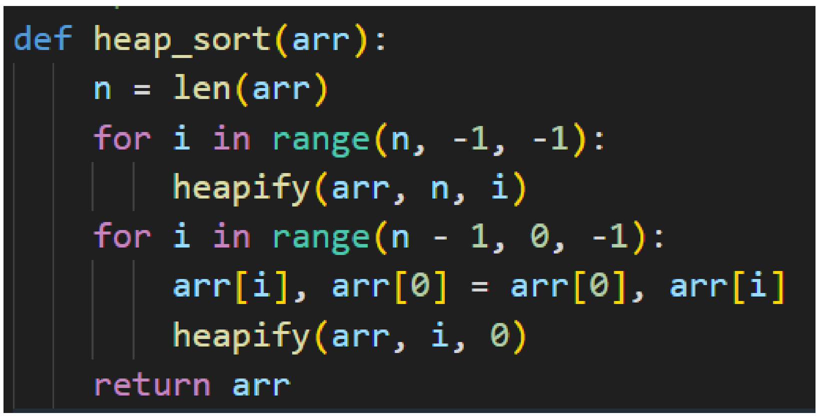
Implementation



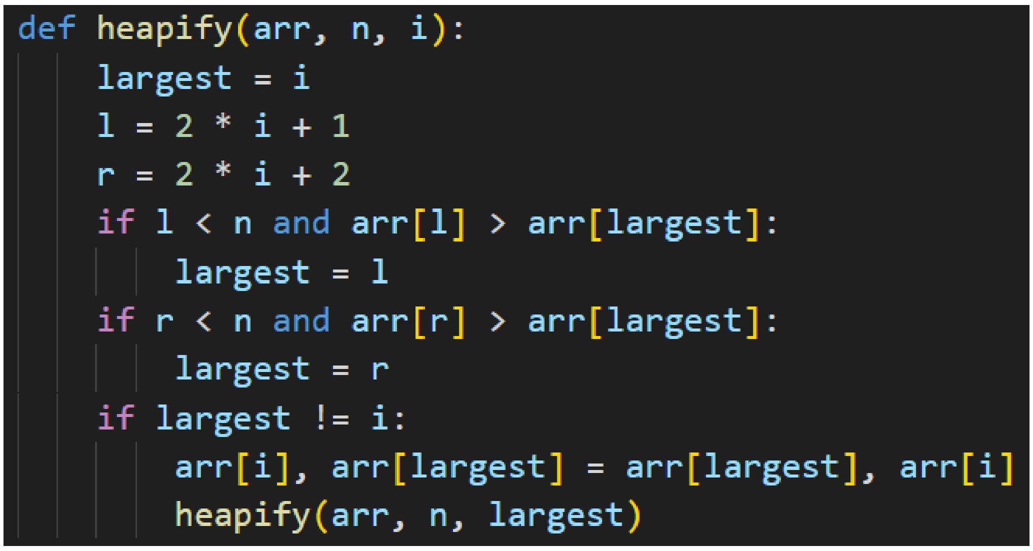
```python
def introsort(arr, depth_limit=None):
    if depth_limit is None:
        depth_limit = 2 * math.floor(math.log2(len(arr)))
    _introsort(arr, depth_limit)
def _introsort(arr, depth_limit):
    n = len(arr)
    if n <= 1:
        return
    elif depth_limit == 0:
        heap_sort(arr)
    else:
        p = partition(arr)
        _introsort(arr[:p], depth_limit - 1)
        _introsort(arr[p+1:], depth_limit - 1)
```



```python
def partition(arr):
    pivot = arr[-1]
    i = -1
    for j in range(len(arr) - 1):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[-1] = arr[-1], arr[i+1]
    return i+1
```

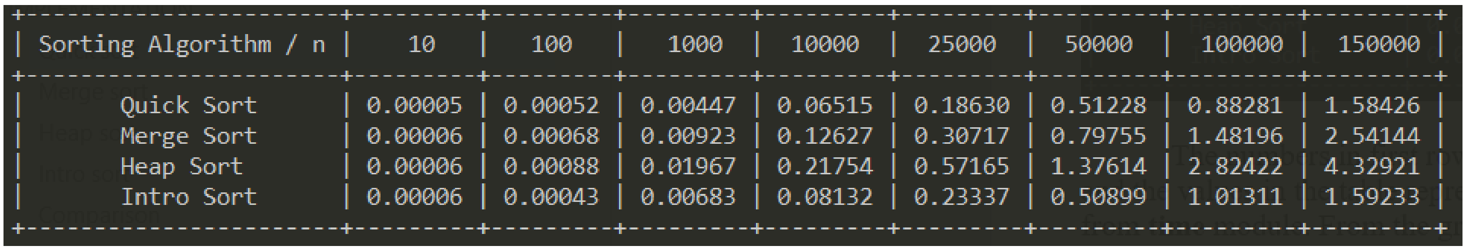


```python
def heap_sort(arr):
    n = len(arr)
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
```



```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

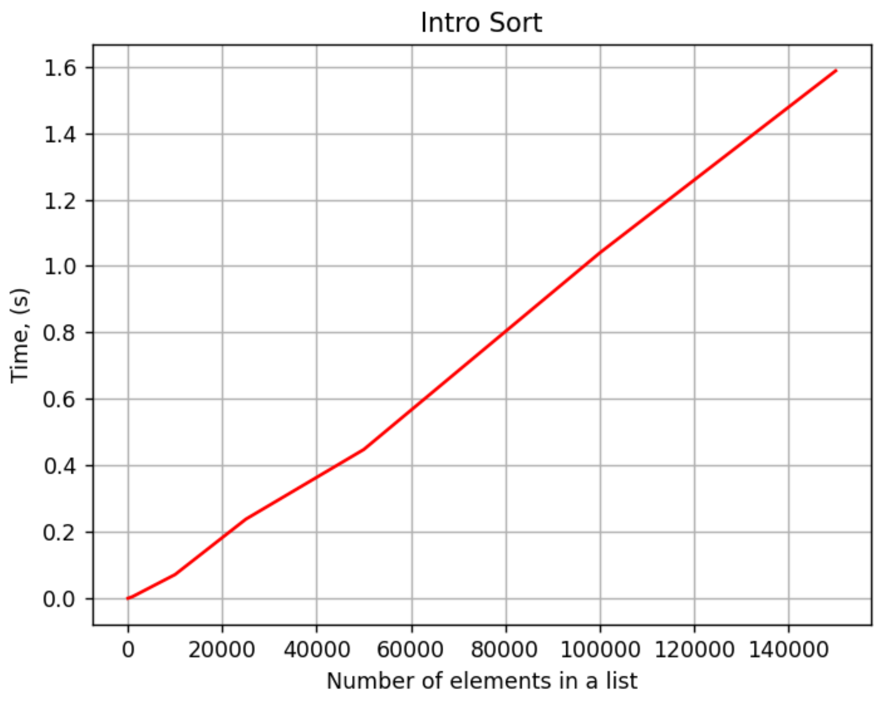
Results



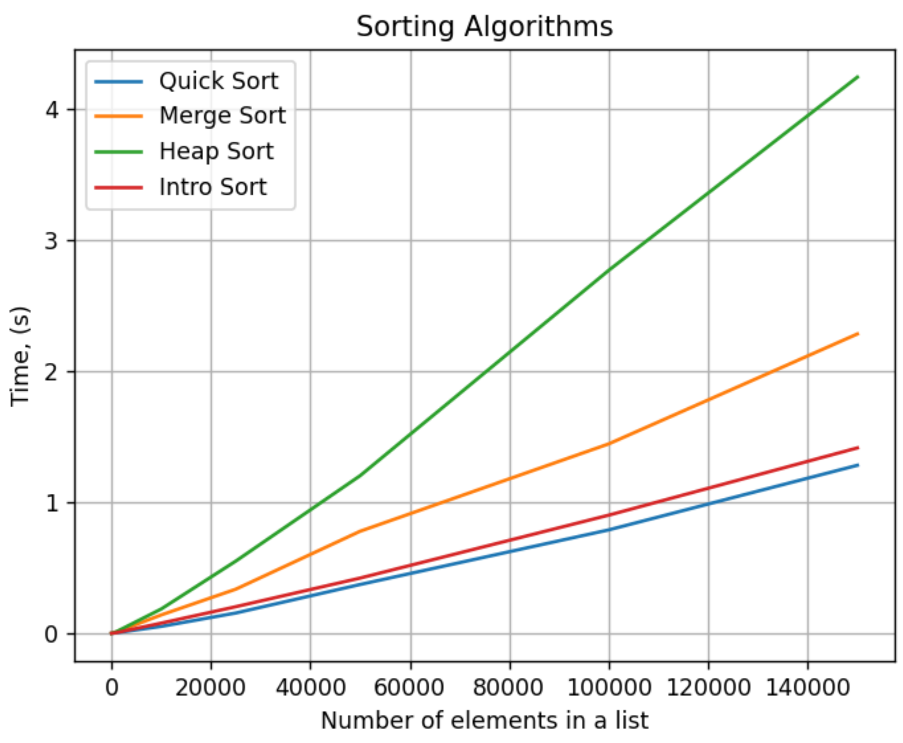
```
+------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
| Sorting Algorithm / n |    10   |   100   |   1000  |  10000  |  25000  |  50000  |  100000 |  150000 |
+------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
|       Quick Sort       | 0.00005 | 0.00052 | 0.00447 | 0.06515 | 0.18630 | 0.51228 | 0.88281 | 1.58426 |
|       Merge Sort       | 0.00006 | 0.00068 | 0.00923 | 0.12627 | 0.30717 | 0.79755 | 1.48196 | 2.54144 |
|       Heap Sort        | 0.00006 | 0.00088 | 0.01967 | 0.21754 | 0.57165 | 1.37614 | 2.82422 | 4.32921 |
|       Intro Sort       | 0.00006 | 0.00043 | 0.00683 | 0.08132 | 0.23337 | 0.50899 | 1.01311 | 1.59233 |
+------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

The numbers in first row that each are themselves columns contain how many elements are in the list and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. From the graph below we could deduce that the time complexity is T(n log n), which is similar to quick sort but still slower a bit slower judging by the time obtained, but it's worth noting that the worst case theoretically is better than quick sort as it is T(n log n).

Graph



Comparison



Rankings of sorting algorithms based on the graph are as follows:

1. Quick sort
2. Intro sort
3. Merge sort
4. Heap sort

CONCLUSION

From the conducted empirical analysis, it can be concluded that Quick sort is the fastest sorting algorithm for the given input sizes. This is followed closely by Intro sort which also performed well in terms of execution time. Merge sort was slightly slower than the top two, but still faster than Heap sort which was the slowest of the four algorithms.

The reason for Quick sort's superior performance can be attributed to its efficient partitioning of the input list and its use of in-place sorting. Meanwhile, Intro sort also employs efficient partitioning and combines it with the use of Heap sort when the recursion depth exceeds a certain threshold, making it a viable alternative to Quick sort.

Merge sort, although slower, is still a reliable algorithm especially for large input sizes due to its stable nature and ability to handle more complex data types. Heap sort, on the other hand, had the slowest execution times which can be attributed to its reliance on heap data structure and its inherent lack of stability.

Overall, the choice of sorting algorithm would depend on the specific requirements of the application, such as the size and complexity of the data, the desired stability of the output, and the available memory for sorting.