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

Laboratory work 2: Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, shellSort.

Elaborated:
st. gr. FAF-221 Lupan Lucian

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2024

Contents

ALGORITHM ANALYSIS

Study 4 sorting algorithms and empirically compare them in a language of your choice.

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:

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure. Sorting algorithms are essential components of computer science and play a crucial role in various applications, where organizing and retrieving data efficiently is essential. There are numerous sorting algorithms, each with its own advantages and disadvantages. Here are some quick theoretical notes about sorting algorithms used in this laboratory work, followed by the importance of empirically comparing them:

Sorting algorithms:

1. **Quick Sort:**
    - A divide-and-conquer algorithm with average-case O(n log n) time complexity.
    - Can perform poorly on nearly sorted datasets (due to choosing a bad pivot element).
2. **Merge Sort:**
    - Another divide-and-conquer algorithm with a guaranteed O(n log n) time complexity.
    - Requires additional space for merging.
3. **Heap Sort:**
    - Uses a binary heap data structure.
    - Guaranteed O(n log n) time complexity, in-place sorting.
4. **Shell Sort:**
    - A variation of insertion sort, with an average time complexity of O(n log n).
    - Employs diminishing increment sequences for efficient sorting.

Empirical Comparison:

Empirical comparison involves testing sorting algorithms on real-world datasets to observe their performance characteristics. Here's why it is crucial:

1. **Practical Efficiency:**
    - Theoretical analyses provide insights into the worst-case, average-case, and best-case scenarios. However, real-world data often exhibits unique patterns that may affect the actual performance of an algorithm.
2. **Adaptability:**
    - Different sorting algorithms may excel in specific scenarios. Empirical comparison helps identify which algorithm performs best under particular conditions, enabling the selection of the most suitable algorithm for a given task.
3. **Implementation Factors:**
    - The actual implementation of an algorithm can impact its performance. Empirical comparisons help identify subtle implementation details or optimizations that can significantly influence the practical efficiency of an algorithm.
4. **Hardware Dependencies:**
    - The efficiency of sorting algorithms can be influenced by the underlying hardware architecture. Empirical testing on different machines and environments provides a more realistic understanding of their performance.
5. **Scaling Behavior:**
    - The scalability of sorting algorithms is crucial, especially when dealing with large datasets. Empirical studies reveal how algorithms scale as the size of the input data increases.
6. **Benchmarking:**
    - Empirical comparison facilitates benchmarking, which is essential for assessing the relative merits of various algorithms in a specific context. Benchmark results guide the selection of the most efficient sorting algorithm for a particular application.

Introduction:

From the beginning of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Among the authors of early sorting algorithms around 1951 was Betty Holberton, who worked on ENIAC and UNIVAC. Bubble sort was analyzed as early as 1956. Asymptotically optimal algorithms have been known since the mid-20th century – new algorithms are still being invented, with the widely used Timsort dating to 2002, and the library sort being first published in 2006.

Comparison sorting algorithms have a fundamental requirement of $\Omega(n \log n)$ 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 not based on comparisons, such as counting sort, can have better performance.

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide-and-conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time–space tradeoffs, and upper and lower bounds.
Sorting small arrays optimally (in fewest comparisons and swaps) or fast (i.e. taking into account machine specific details) is still an open research problem, with solutions only known for very small arrays (<20 elements). Similarly optimal (by various definitions) sorting on a parallel machine is an open research topic.

Within this laboratory, we will be analyzing 4 sorting algorithms (quick sort, merge sort, heap sort and shell sort) empirically.

Comparison Metric:

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

Input Format:

As input, each sorting algorithm will receive the same array, composed of n random numbers (ranging from 0 to n), where n = size = 10000. All the algorithms will be run 25 times to avoid local errors in execution times, each time utilizing a new randomly generated array.

IMPLEMENTATION

All four algorithms are implemented in Python and analyzed empirically based on the time required for their completion, using the matplotlib and timeit libraries (as well as the random library to generate the array).

Quick sort:

Algorithm Description:

QuickSort is a sorting algorithm based on the divide and conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Generally, partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array. However, because our algorithm works with a large dataset, we will instead be utilizing the iterative quick sort approach, in order to avoid the maximum recursion limit (and the stack overflow error if we modify it).

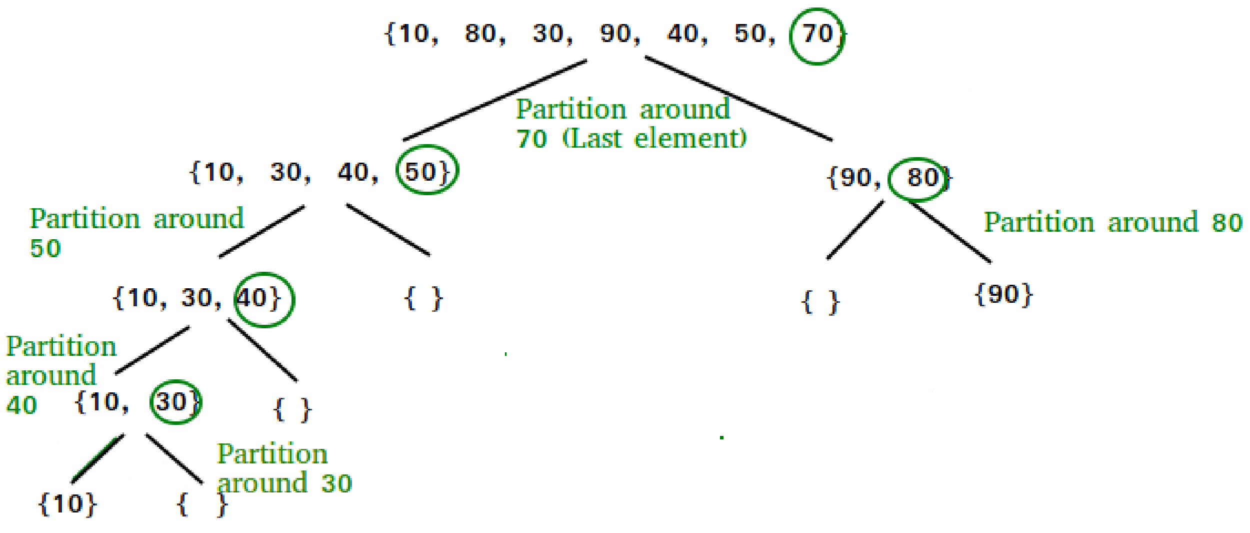
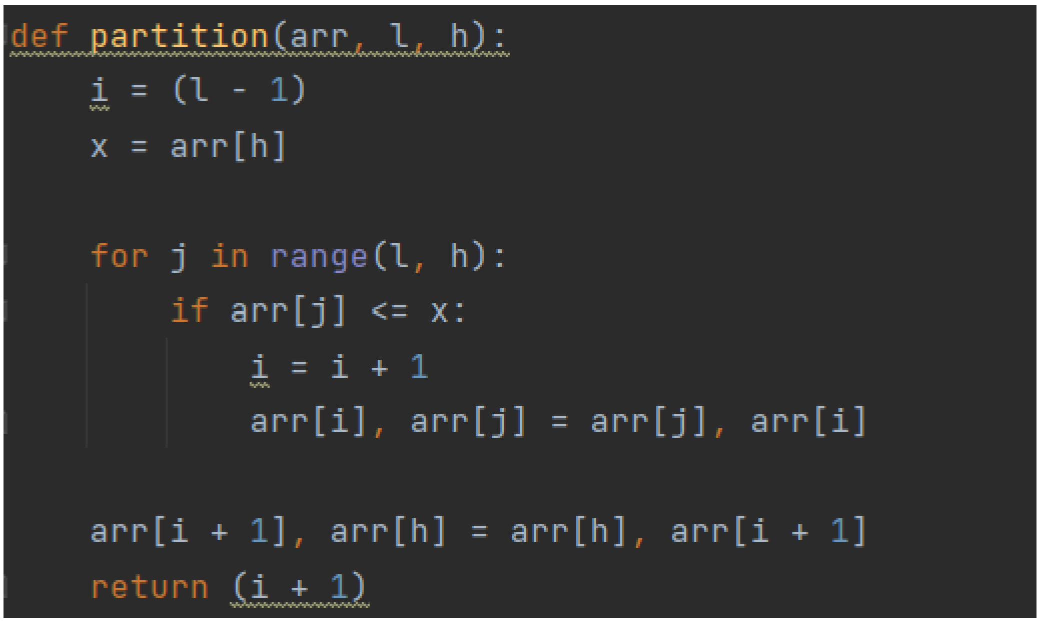


Figure 1. Quick sort execution.

Implementation:

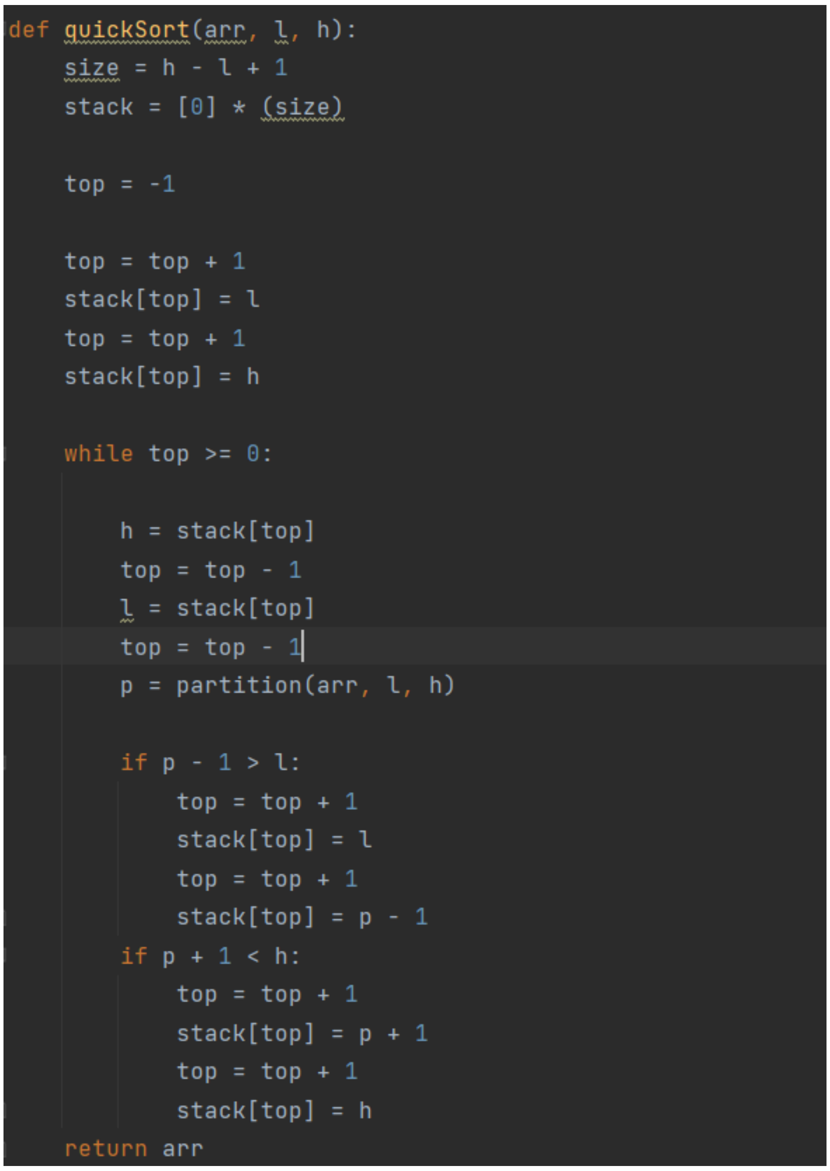


```python
def partition(arr, l, h):
    i = (l - 1)
    x = arr[h]

    for j in range(l, h):
        if arr[j] <= x:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[h] = arr[h], arr[i + 1]
    return (i + 1)
```

Figure 2. Quick sort partition.



```python
def quickSort(arr, l, h):
    size = h - l + 1
    stack = [0] * (size)

    top = -1

    top = top + 1
    stack[top] = l
    top = top + 1
    stack[top] = h

    while top >= 0:

        h = stack[top]
        top = top - 1
        l = stack[top]
        top = top - 1
        p = partition(arr, l, h)

        if p - 1 > l:
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = p - 1
        if p + 1 < h:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = h
    return arr
```

Figure 3. Quick sort code.

Merge sort:

Algorithm Description:

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.

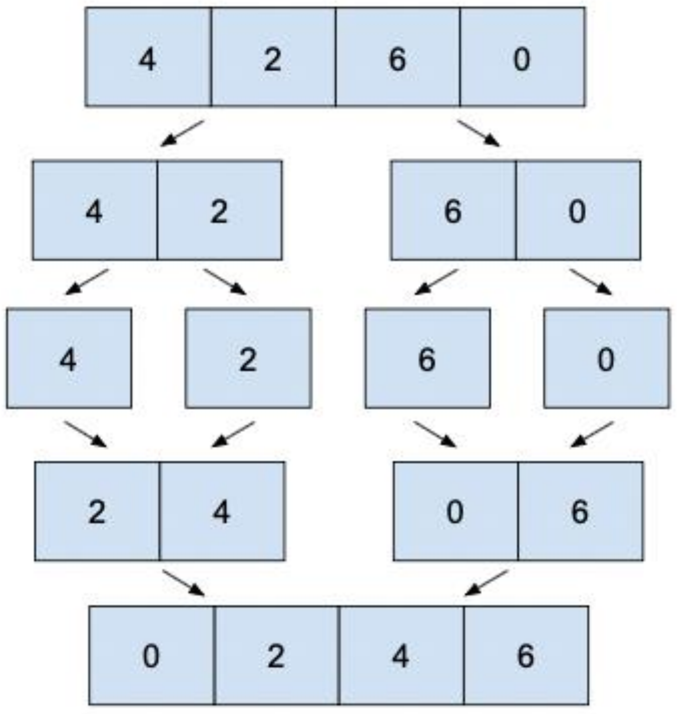
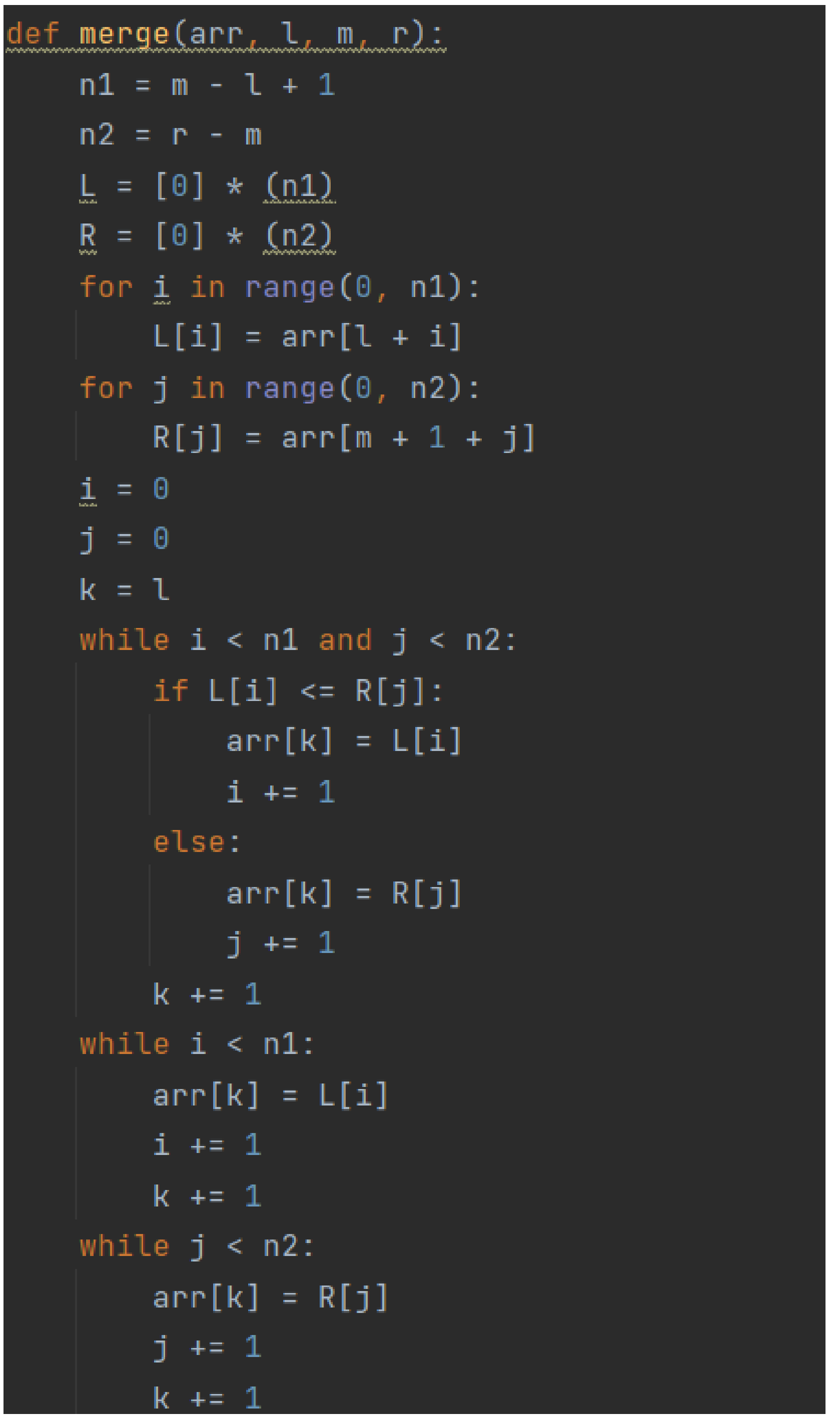


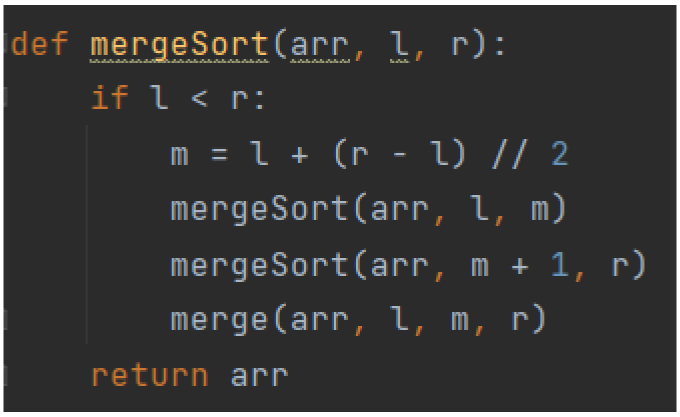
Figure 4. Merge sort execution.

Implementation:



```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0, n1):
        L[i] = arr[l + i]
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]
    i = 0
    j = 0
    k = l
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
```

Figure 5. Merge sort merging function.



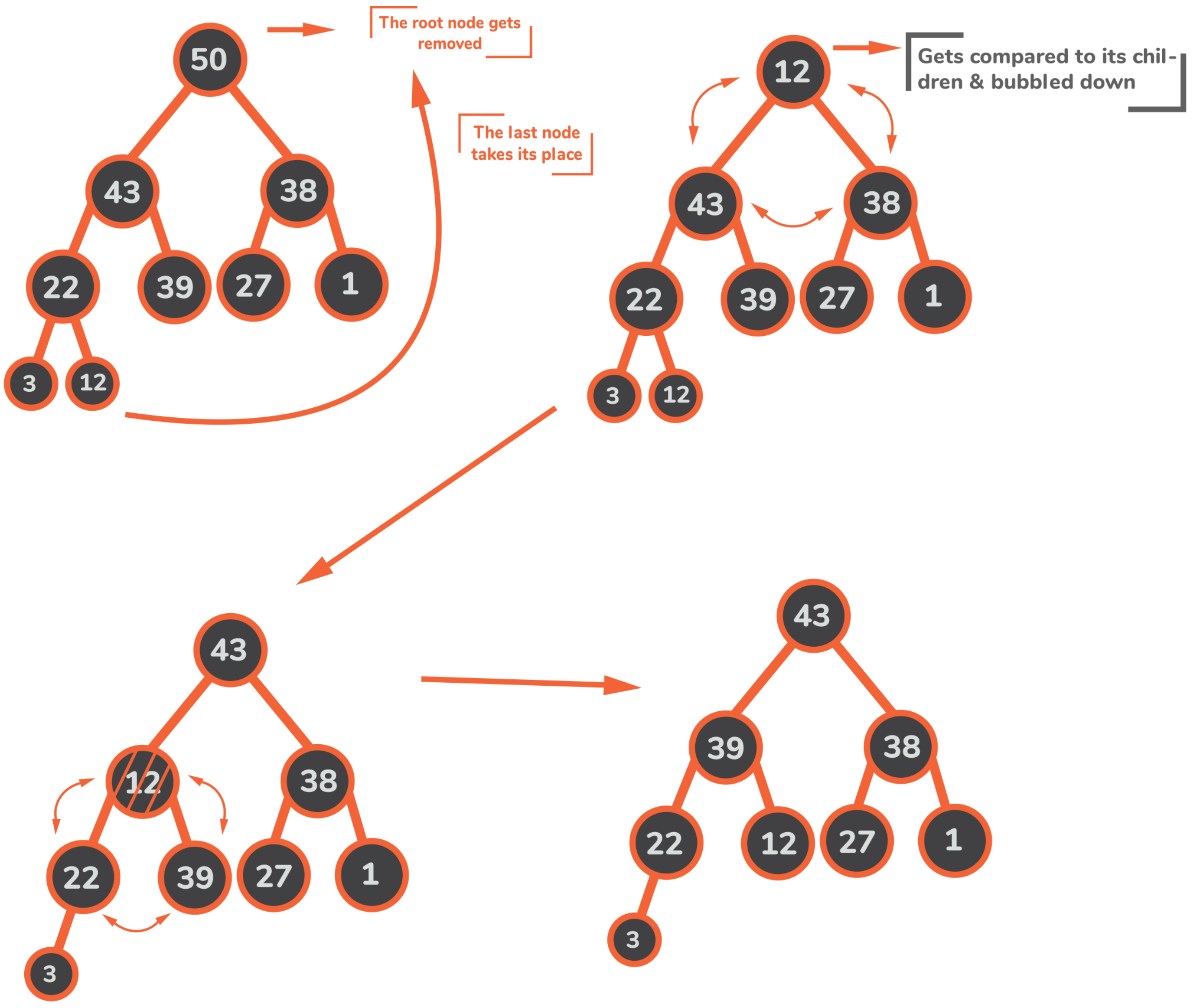
```
def mergeSort(arr, l, r):
    if l < r:
        m = l + (r - l) // 2
        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)
    return arr
```

Figure 6. Merge sort code.

Heap sort:

Algorithm Description:

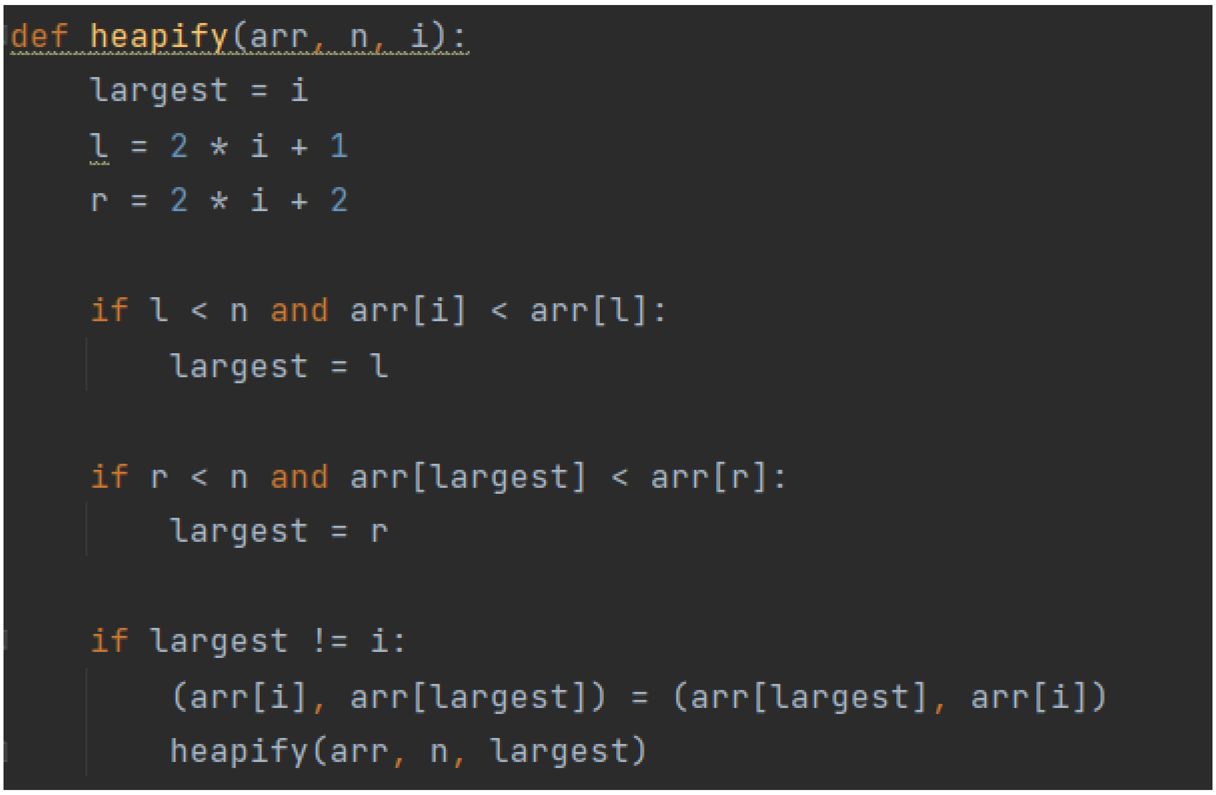
Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.



The root node gets removed
The last node takes its place
Gets compared to its children & bubbled down
50
43
38
22
39
27
1
3
12

Figure 7. Heap sort execution

Implementation:



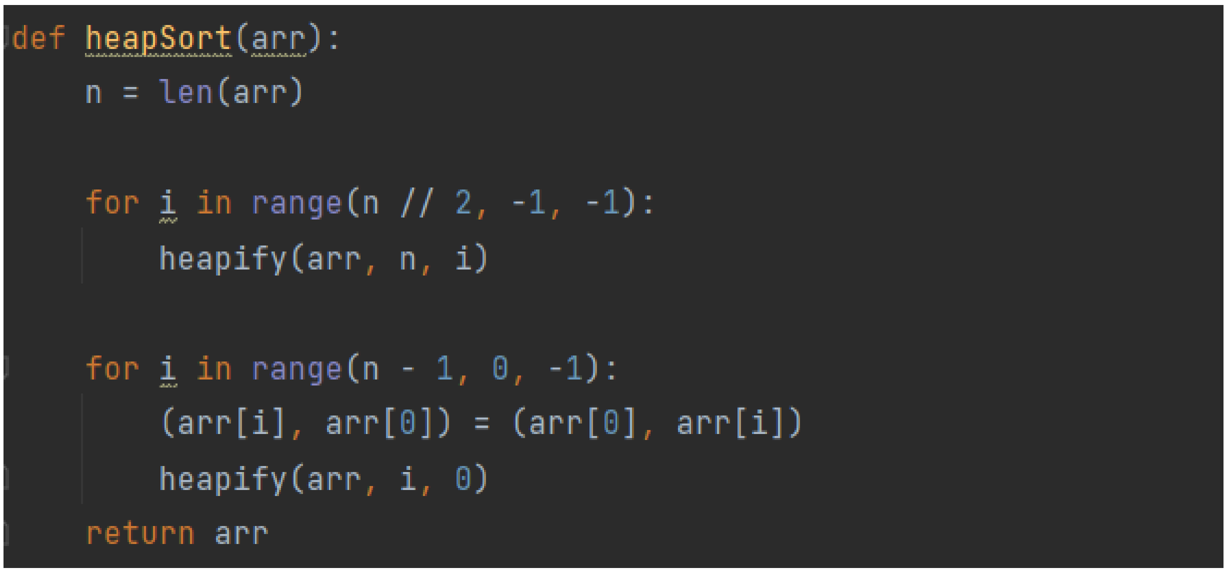
```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i])
        heapify(arr, n, largest)
```

Figure 8. Heapify function code.



```python
def heapSort(arr):
    n = len(arr)

    for i in range(n // 2, -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
```

Figure 9. Heap sort code.

Shell sort:

Algorithm Description:

Shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element are sorted.

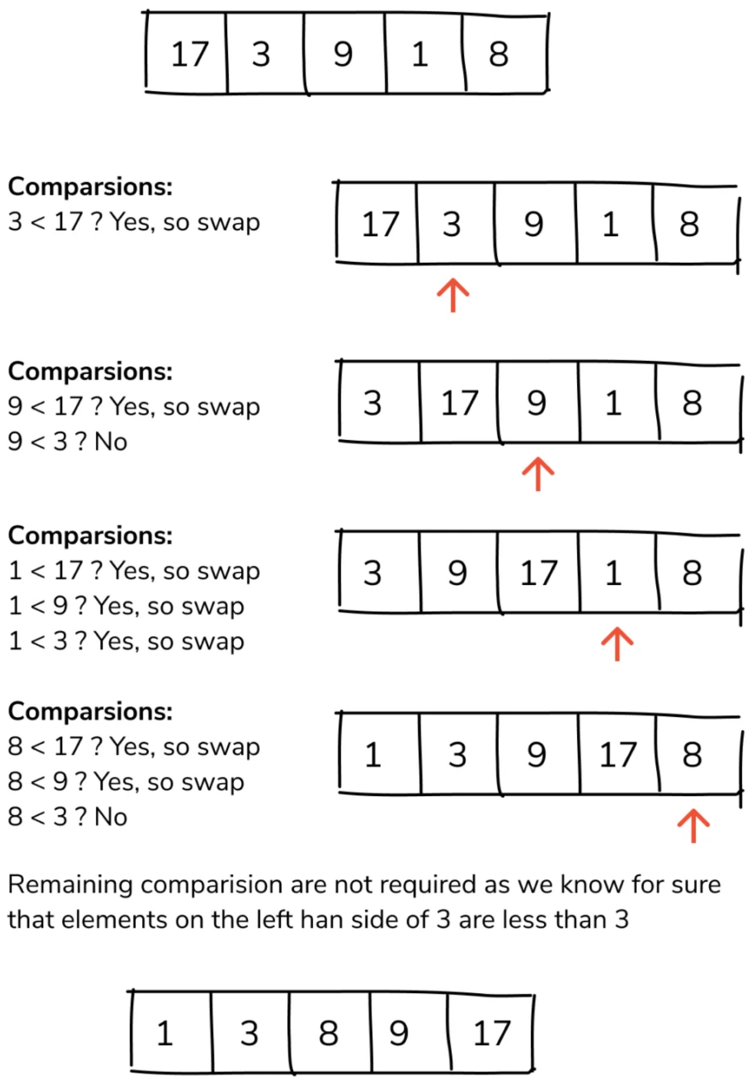
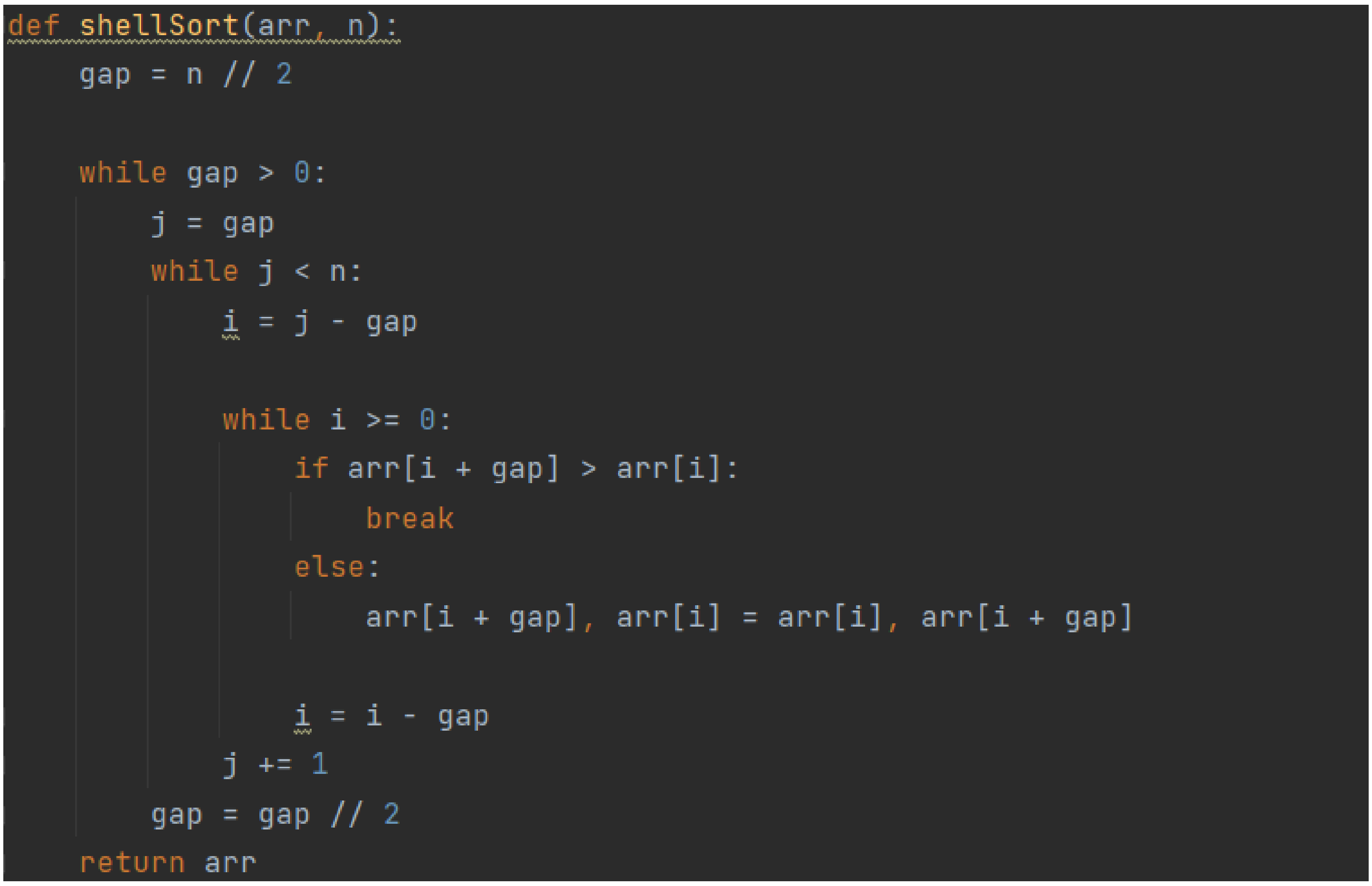


Figure 10. Shell sort execution

Implementation:



```python
def shellSort(arr, n):
    gap = n // 2

    while gap > 0:
        j = gap
        while j < n:
            i = j - gap

            while i >= 0:
                if arr[i + gap] > arr[i]:
                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]

                i = i - gap
            j += 1
        gap = gap // 2
    return arr
```

Figure 11. Shell sort code

Algorithm comparison results:

After performing the results on randomized datasets, saving the execution times, and plotting them on a graph, the results are as follows:

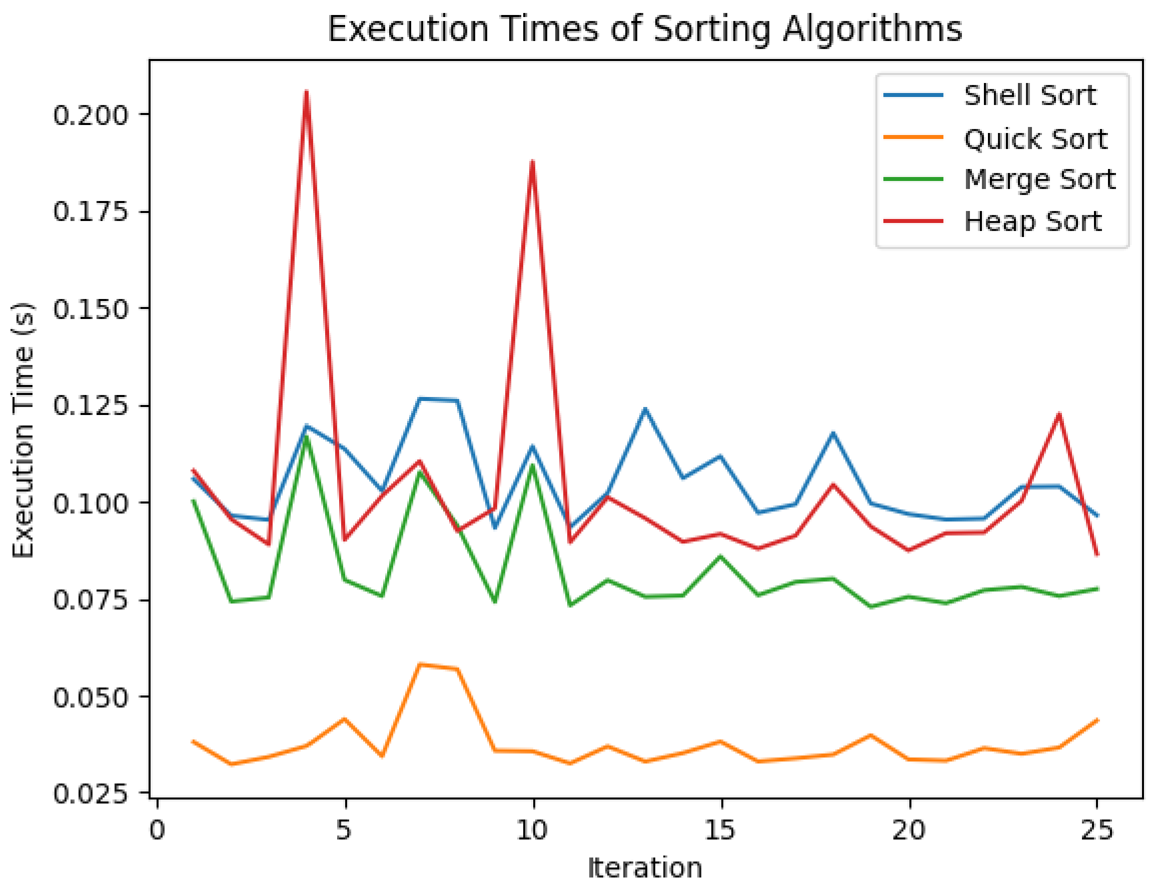


Figure 12. Algorithm comparison, first execution.

In figure 12, we can notice that quick sort is the clear winner, being, on average, 3 times faster than the second-place algorithm – merge sort. Apart from 2 spikes, heap sort is generally in third place, with shell sort following closely behind in terms of execution times, however, it still lands in last place.

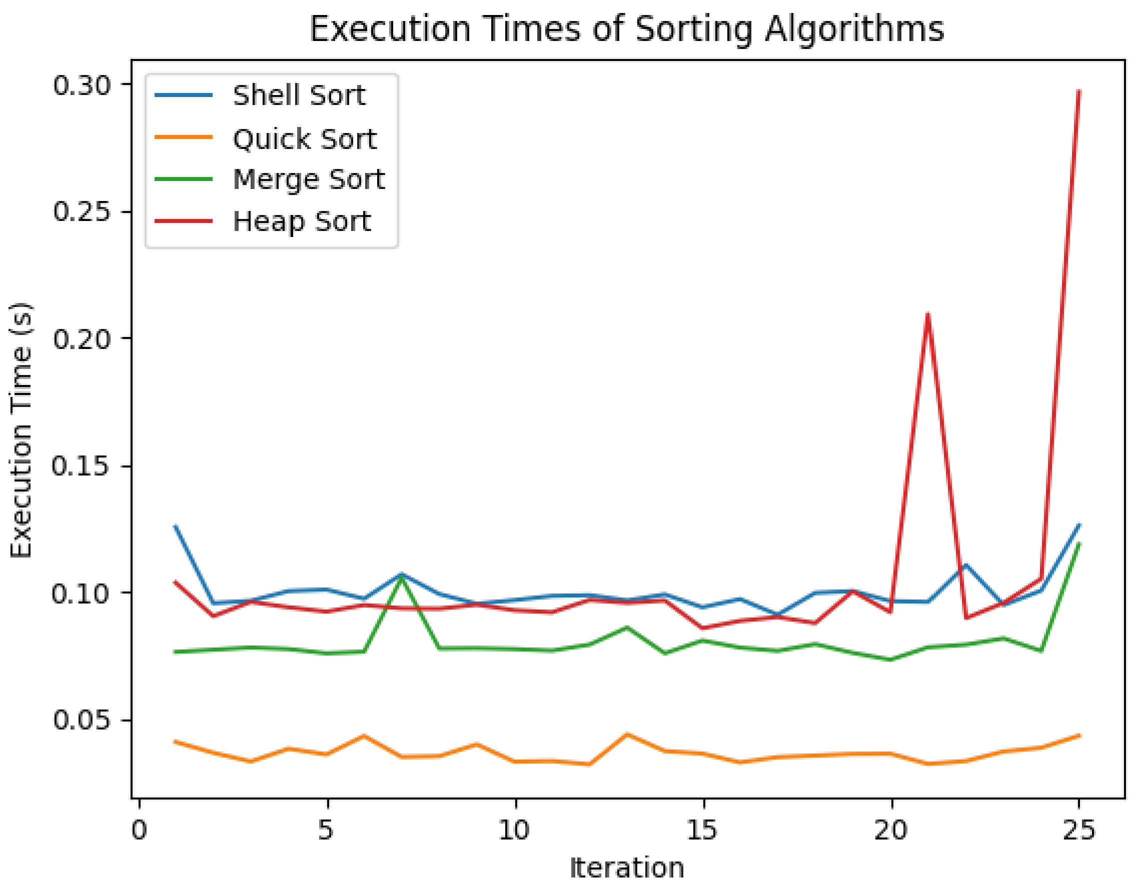


Figure 13. Algorithm comparison, second execution.

In figure 13, much like the first execution, quick sort takes the first place, with no significant performance spikes, followed by the merge sort in second place. Due to the massive spikes in heap sort's execution of the 21st and 25th sorting iteration, shell sort takes 3rd place, bumping heap sort up to last.

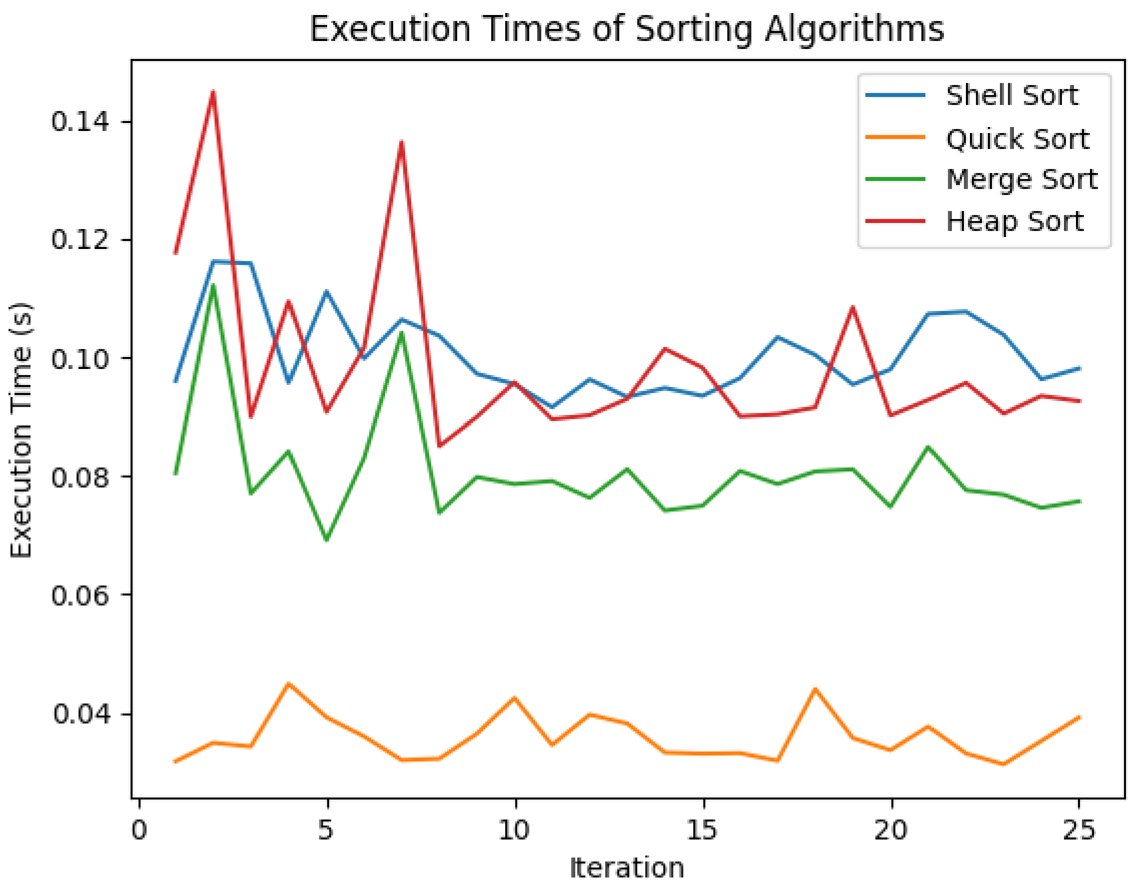


Figure 14. Algorithm comparison, third execution.

The speed rankings, like the first two executions, stay the same. A notable difference, from all three executions, is that heap sort tends to spike more often than the other algorithms, causing it to take the last place overall, making it the slowest algorithm in this comparison on average.

CONCLUSION

After finishing this laboratory work, I learnt more in-depth information about the 4 studied sorting algorithms – quick sort, merge sort, heap sort and shell sort. I also gained some knowledge about programming in python, improved my problem-solving skills and gained a broader perspective on how important efficiency is when taking into account large datasets.

Even though all the algorithms that were used in this laboratory work had a time complexity of just O(n log n), paired with a very fast execution time, some shined brighter than others. More specifically, quick sort was much faster than all of the other 3 algorithms, being, on average, 2 times faster than the second faster sorting algorithm – merge sort. This is probably due to the fully randomized dataset, as using a semi-filtered dataset can increase its time complexity to O(n^2), taking seconds, as opposed to milliseconds to sort through the entire dataset.

Another interesting observation is the fact that heap sort and shell sort were mostly neck-to-neck when it came to sorting the randomized dataset. However, heap sort sometimes experienced large spikes in execution times, causing it to fall behind shell sort, even though it generally had an execution time faster than that of shell sort by a few milliseconds.

Although I learnt about these algorithms in the previously taught DSA classes, I learnt more about implementing them in a "real world" use case scenario, on larger datasets and adapting them to restricting conditions - by the programming language, Python having a set recursion limit to avoid stack overflows, and, like the previous laboratory work, I learnt how to work with the timeit and matplotlib libraries to empirically measure algorithms' efficiency.

GitHub repository link - https://github.com/Lucian-Lu/AA-Labs