**Ministerul Educaţiei și 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, insertionSort

**Elaborated:**
**st. gr. FAF-221** **Mîndrescu Dragomir**
**Verified:**
**asist. univ.** **Fiștic Cristofor**

**Chişinău - 2024**

## TABLE OF CONTENTS

# ALGORITHM ANALYSIS

**Objective**

Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, insertionSort(one of my 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:**

QuickSort:

Overview: QuickSort is a divide-and-conquer sorting algorithm that 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. The process is then applied recursively to the sub-arrays.

Key Features:

- Efficient in practice due to low overhead.
- In-place sorting, requiring only a constant amount of additional memory.
- Best and average case time complexity is O(n log n), but worst-case time complexity is O(n^2) in the presence of an already sorted array.

MergeSort:

Overview: MergeSort is another divide-and-conquer algorithm that divides the unsorted list into n sub-lists, each containing one element, and then repeatedly merges sub-lists to produce new sorted sub-lists until there is only one remaining.

Key Features:

- Stable sorting algorithm.
- Requires additional memory proportional to the size of the input, making it less space-efficient than QuickSort.
- Guaranteed worst-case time complexity of O(n log n), making it suitable for large datasets.

HeapSort:

Overview: HeapSort is based on the binary heap data structure. It builds a max heap or min heap (depending on sorting order) from the array and repeatedly extracts the maximum (or minimum) element to obtain a sorted array.

Key Features:

- In-place sorting algorithm with no additional memory requirements.

- Not stable, as it may change the relative order of equal elements.
- Time complexity is O(n log n) in all cases, making it consistent in performance.

InsertionSort:

Overview: InsertionSort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as QuickSort, MergeSort, or HeapSort.

Key Features:

- Adaptive algorithm, efficient for small data sets or mostly sorted data.
- In-place sorting with minimal memory requirements.
- Time complexity is O(n^2) in the worst case, making it less suitable for large datasets compared to other algorithms.

## Introduction:

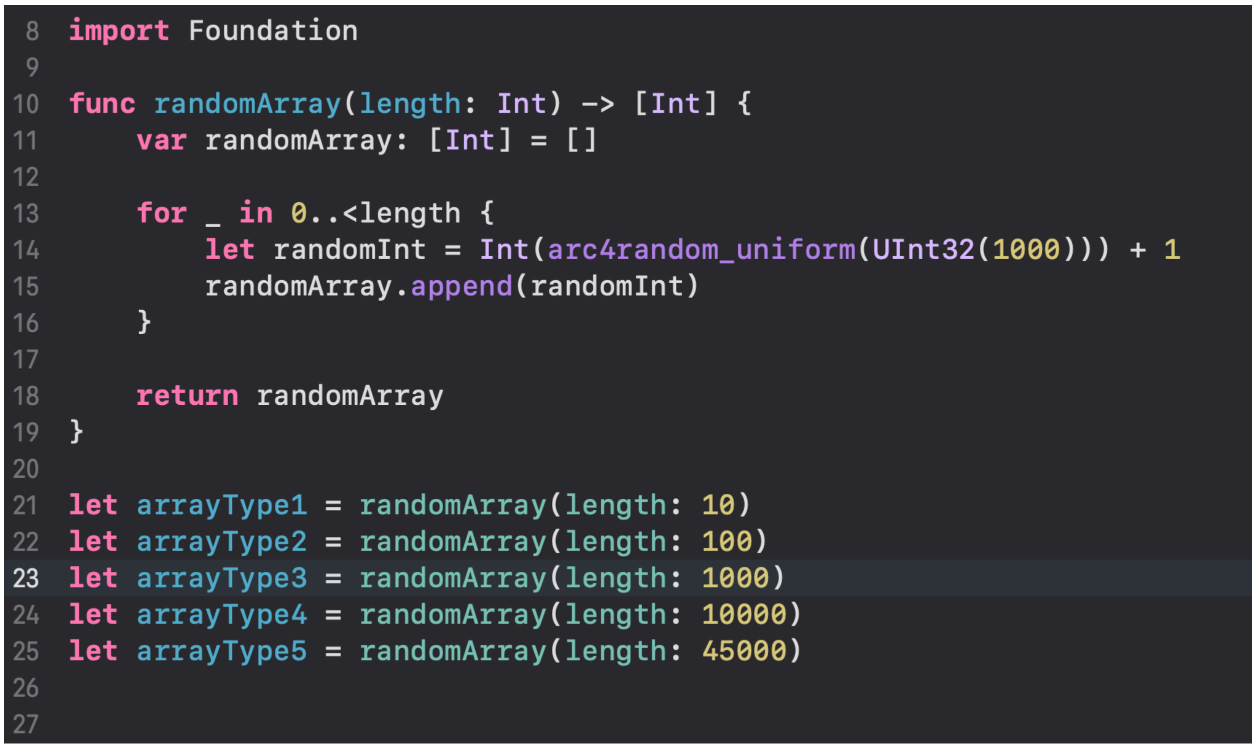
Sorting algorithms are essential processes in computer science, used to rearrange elements in a specific order within a collection or dataset. The goal is to organize the data in a way that makes it easier to search, access, and manage. Various sorting algorithms exist, each with its own advantages and disadvantages.

## Comparison Metric:

The primary metric for this analysis will be the execution time of each algorithm ($T(n)$).

## Input Format:

In assessing the sorting algorithms' efficiency, arrays of varying lengths, ranging from 10 to 45,000 elements, will be generated. These arrays represent different difficulty levels, from small (10 elements) to extremely large (45,000 elements). The goal is to evaluate how well the algorithms handle the diverse challenges posed by different array sizes. The focus will be on sorting arrays of integers to simulate real-world scenarios. This approach enables a comprehensive analysis of the algorithms' scalability and performance across a spectrum of input sizes, shedding light on their suitability for integer array sorting.



```swift
import Foundation

func randomArray(length: Int) -> [Int] {
    var randomArray: [Int] = []

    for _ in 0..<length {
        let randomInt = Int(arc4random_uniform(UInt32(1000))) + 1
        randomArray.append(randomInt)
    }

    return randomArray
}

let arrayType1 = randomArray(length: 10)
let arrayType2 = randomArray(length: 100)
let arrayType3 = randomArray(length: 1000)
let arrayType4 = randomArray(length: 10000)
let arrayType5 = randomArray(length: 45000)
```

*Figure 1 Array Generator in Swift*

## IMPLEMENTATION

### QuickSort Method:

QuickSort, a divide-and-conquer sorting algorithm developed by Tony Hoare, stands out for its efficiency and widespread use. It selects a pivot, partitions the array, and recursively applies the process, achieving an average and best-case time complexity of O(n log n). Notably, QuickSort excels in handling large datasets due to its in-place sorting and low overhead. Despite a worst-case time complexity of O(n^2) for already sorted arrays, its adaptability, simplicity, and overall performance make it a popular choice in practical applications.

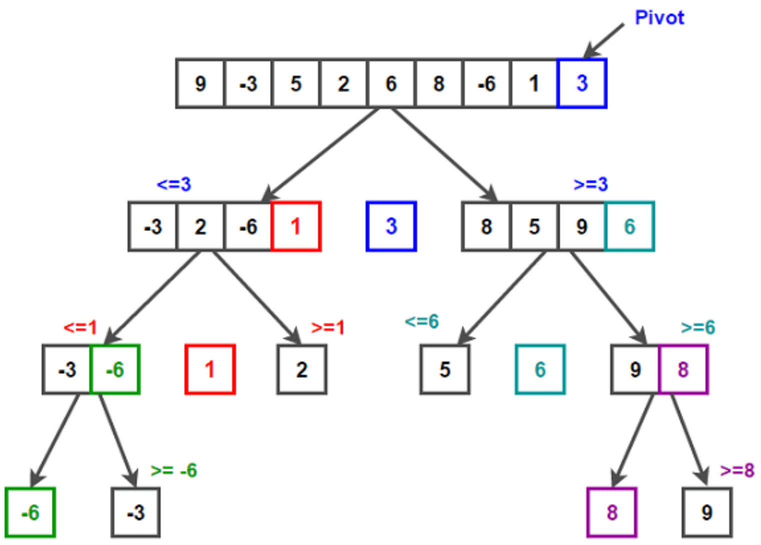


*Figure 2 QuickSort Algorithm*

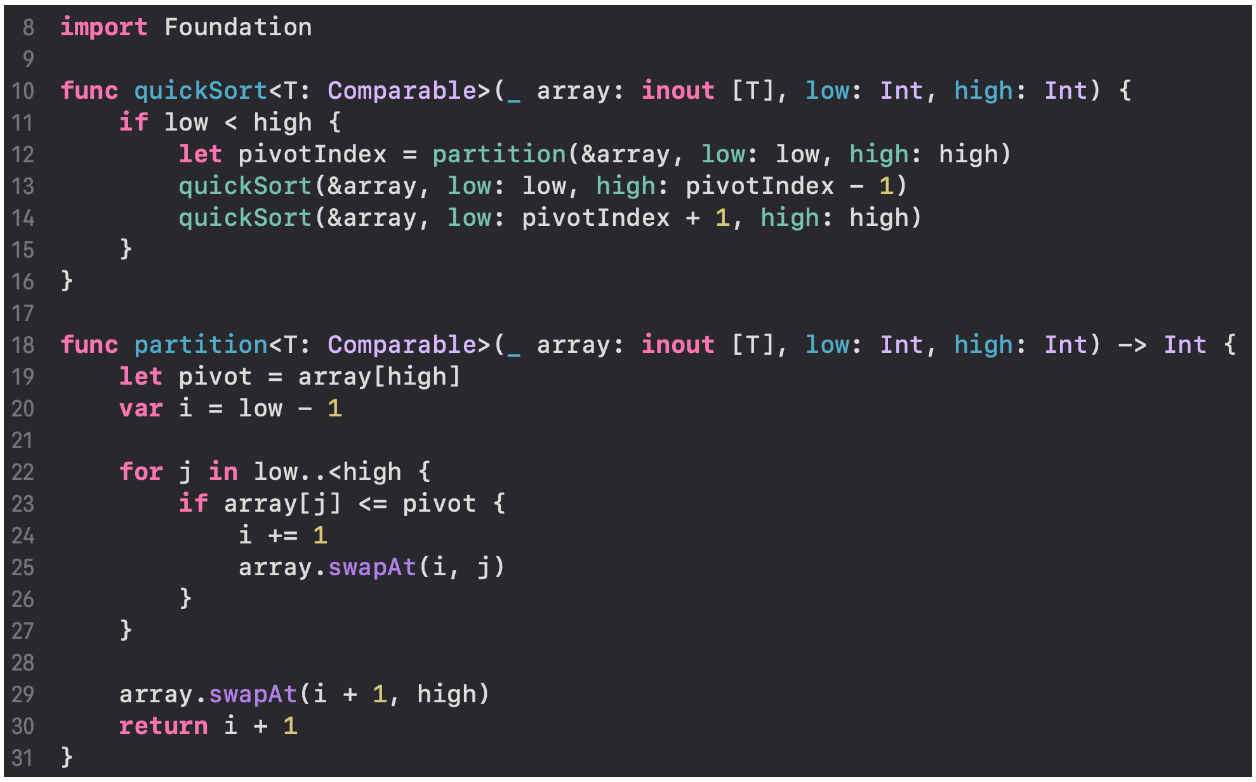
*Algorithm Description:*

Here's a simple pseudocode for the QuickSort algorithm:

```
quicksort(arr, low, high)
    if low < high
        pivotIndex = partition(arr, low, high)
        quicksort(arr, low, pivotIndex - 1)
        quicksort(arr, pivotIndex + 1, high)

partition(arr, low, high)
    pivot = arr[high]
    i = low - 1

    for j = low to high - 1
        if arr[j] <= pivot
            i = i + 1
            swap(arr[i], arr[j])

    swap(arr[i + 1], arr[high])
    return i + 1
```

*Implementation:*

The implementation in Swift utilizes the QuickSort algorithm to efficiently sort five arrays of different sizes. By measuring the time taken for each sorting operation, the code provides insights into the algorithm's performance across diverse scenarios. The modular Swift code employs functions for sorting and time measurement, ensuring clarity and readability. The results are presented in a tabular format, offering a concise summary of the QuickSort algorithm's effectiveness in handling arrays of varying lengths.

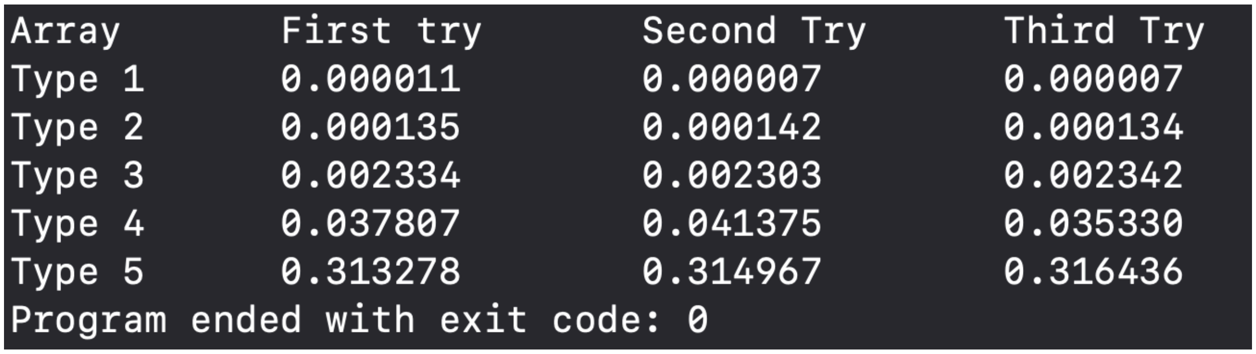


```swift
import Foundation

func quickSort<T: Comparable>(_ array: inout [T], low: Int, high: Int) {
    if low < high {
        let pivotIndex = partition(&array, low: low, high: high)
        quickSort(&array, low: low, high: pivotIndex - 1)
        quickSort(&array, low: pivotIndex + 1, high: high)
    }
}

func partition<T: Comparable>(_ array: inout [T], low: Int, high: Int) -> Int {
    let pivot = array[high]
    var i = low - 1

    for j in low..<high {
        if array[j] <= pivot {
            i += 1
            array.swapAt(i, j)
        }
    }

    array.swapAt(i + 1, high)
    return i + 1
}
```

*Figure 3 QuickSort Method in Swift*

*Results:*

The output of the Swift implementation provides a clear and organized presentation of the QuickSort algorithm's performance on five distinct arrays. Displayed in a tabular format, the results showcase the time taken for each sorting operation across three repetitions. The table structure effectively communicates how the algorithm handles different array types, offering a concise summary of its efficiency and consistency.



```
Array        First try       Second Try      Third Try
Type 1       0.000011        0.000007        0.000007
Type 2       0.000135        0.000142        0.000134
Type 3       0.002334        0.002303        0.002342
Type 4       0.037807        0.041375        0.035330
Type 5       0.313278        0.314967        0.316436
Program ended with exit code: 0
```

*Figure 4 Results for the QuickSort Algorithm performance*

This output serves as a valuable reference for understanding the algorithm's behavior under various input scenarios, aiding in the assessment of its practical applicability and performance characteristics.

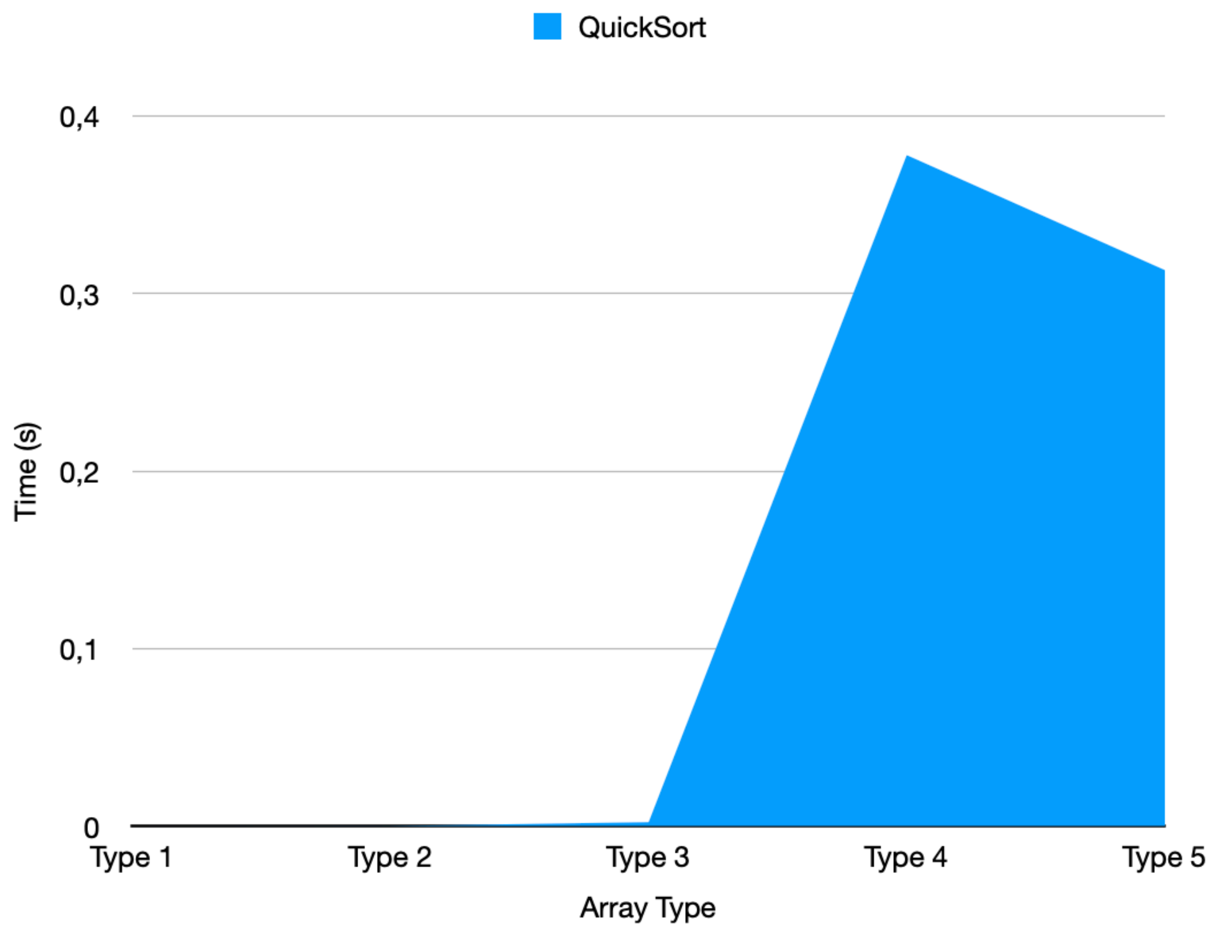


*Figure 5 Graph for the QuickSort Algorithm performance*

The line chart visually represents the relationship between the type of array and the corresponding time spent during the QuickSort algorithm's execution. Each line on the chart signifies a specific array type, providing a clear comparison of how the algorithm's performance varies across different scenarios. The x-axis denotes the array types, while the y-axis captures the time spent on sorting in seconds. The chart's ascending or descending trends offer insights into the algorithm's scalability and efficiency, with potential patterns indicating its responsiveness to varying array lengths. This graphical representation enhances the understanding of QuickSort's behavior, aiding in the identification of optimal use cases and potential trade-offs based on the input data characteristics.

**MergeSort Method:**

Merge Sort is a widely-used and efficient sorting algorithm known for its stability and consistent performance. It follows a divide-and-conquer strategy to sort an array or list of elements. The algorithm recursively divides the input array into smaller halves until each sub-array contains only one element, a trivially sorted state. Subsequently, the sorted sub-arrays are merged back together in a manner that preserves their order, resulting in a fully sorted array. Merge Sort boasts a guaranteed worst-case time complexity of O(n log n), making it suitable for handling large datasets. Its stability, meaning it preserves the relative order of equal elements, makes Merge Sort particularly advantageous in scenarios where maintaining the initial order is crucial. Despite its additional memory requirements due to the creation of temporary arrays during merging, the algorithm's reliability and efficiency contribute to its widespread adoption in various applications. Merge Sort's clear and straightforward implementation, coupled with its consistent performance, positions it as a reliable choice for sorting tasks in diverse computational contexts.

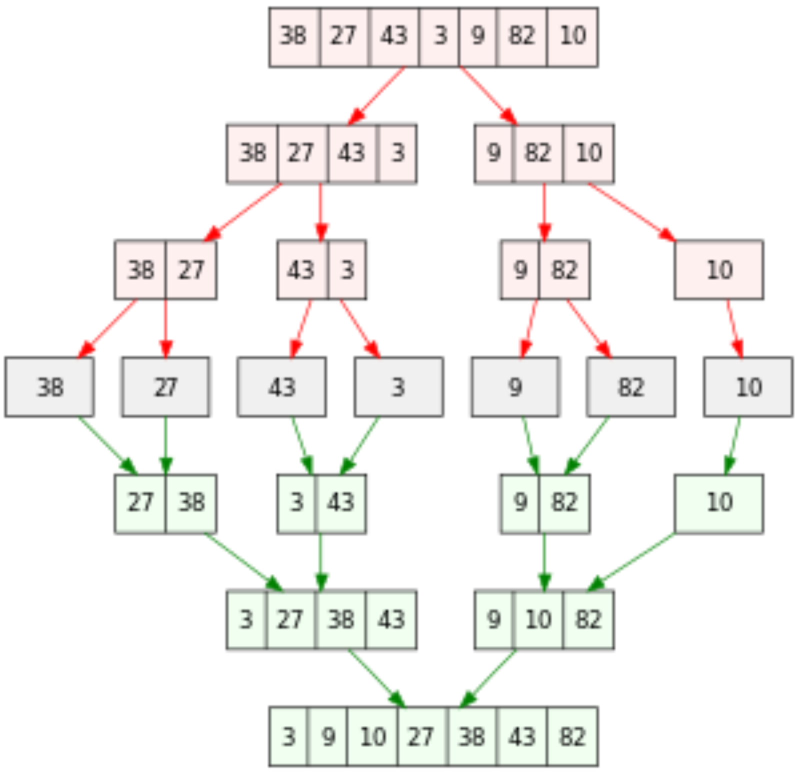


*Figure 6 MergeSort Algorithm*

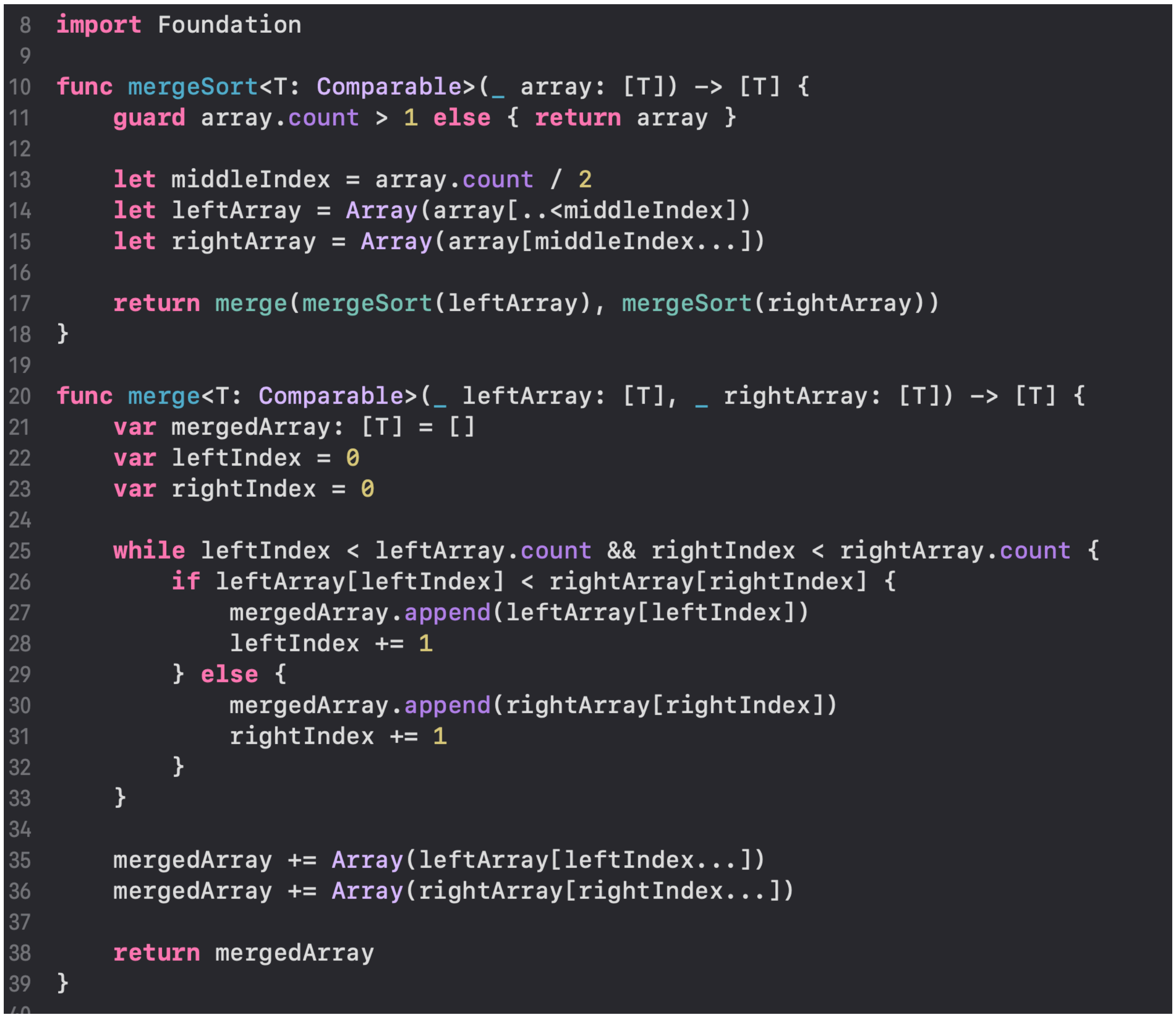
*Algorithm Description:*

Here's a simple pseudocode for the Merge Sort algorithm:

```
    mergesort(arr)
        if length of arr <= 1
           return arr

        middle = length of arr / 2
        left = mergesort(first half of arr)
        right = mergesort(second half of arr)

        return merge(left, right)

    merge(left, right)
        result = []
        leftIndex = 0
        rightIndex = 0

        while leftIndex < length of left and rightIndex < length
of right
            if left[leftIndex] < right[rightIndex]
                add left[leftIndex] to result
                leftIndex = leftIndex + 1
            else
                add right[rightIndex] to result
                rightIndex = rightIndex + 1

        append remaining elements of left to result
        append remaining elements of right to result

        return result
```

*Implementation:*

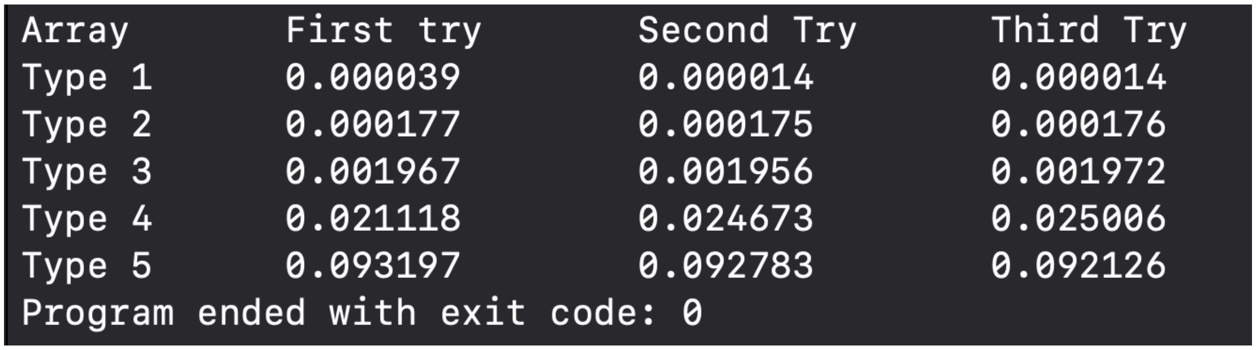
Implementing Merge Sort in Swift involves creating two main functions: `mergesort` and `merge`. The `mergesort` function recursively divides the input array into halves until reaching sub-arrays of size 1. It then utilizes the `merge` function to combine these sorted sub-arrays into larger sorted arrays. The `merge` function compares elements from the left and right sub-arrays and builds a merged array in ascending order. The Swift implementation leverages the language's syntax to make the code clear and readable. Notably, the use of generic types allows for sorting arrays of any comparable elements. The resulting Swift code is concise, modular, and efficient, providing a reliable solution for sorting arrays with the Merge Sort algorithm in a Swift environment.



```swift
import Foundation

func mergeSort<T: Comparable>(_ array: [T]) -> [T] {
    guard array.count > 1 else { return array }

    let middleIndex = array.count / 2
    let leftArray = Array(array[..<middleIndex])
    let rightArray = Array(array[middleIndex...])

    return merge(mergeSort(leftArray), mergeSort(rightArray))
}

func merge<T: Comparable>(_ leftArray: [T], _ rightArray: [T]) -> [T] {
    var mergedArray: [T] = []
    var leftIndex = 0
    var rightIndex = 0

    while leftIndex < leftArray.count && rightIndex < rightArray.count {
        if leftArray[leftIndex] < rightArray[rightIndex] {
            mergedArray.append(leftArray[leftIndex])
            leftIndex += 1
        } else {
            mergedArray.append(rightArray[rightIndex])
            rightIndex += 1
        }
    }

    mergedArray += Array(leftArray[leftIndex...])
    mergedArray += Array(rightArray[rightIndex...])

    return mergedArray
}
```

*Figure 7 MergeSort Algorithm in Swift*

Results:



```
Array         First try        Second Try       Third Try
Type 1        0.000039         0.000014         0.000014
Type 2        0.000177         0.000175         0.000176
Type 3        0.001967         0.001956         0.001972
Type 4        0.021118         0.024673         0.025006
Type 5        0.093197         0.092783         0.092126
Program ended with exit code: 0
```

*Figure 8 Results of the MergeSort performance*

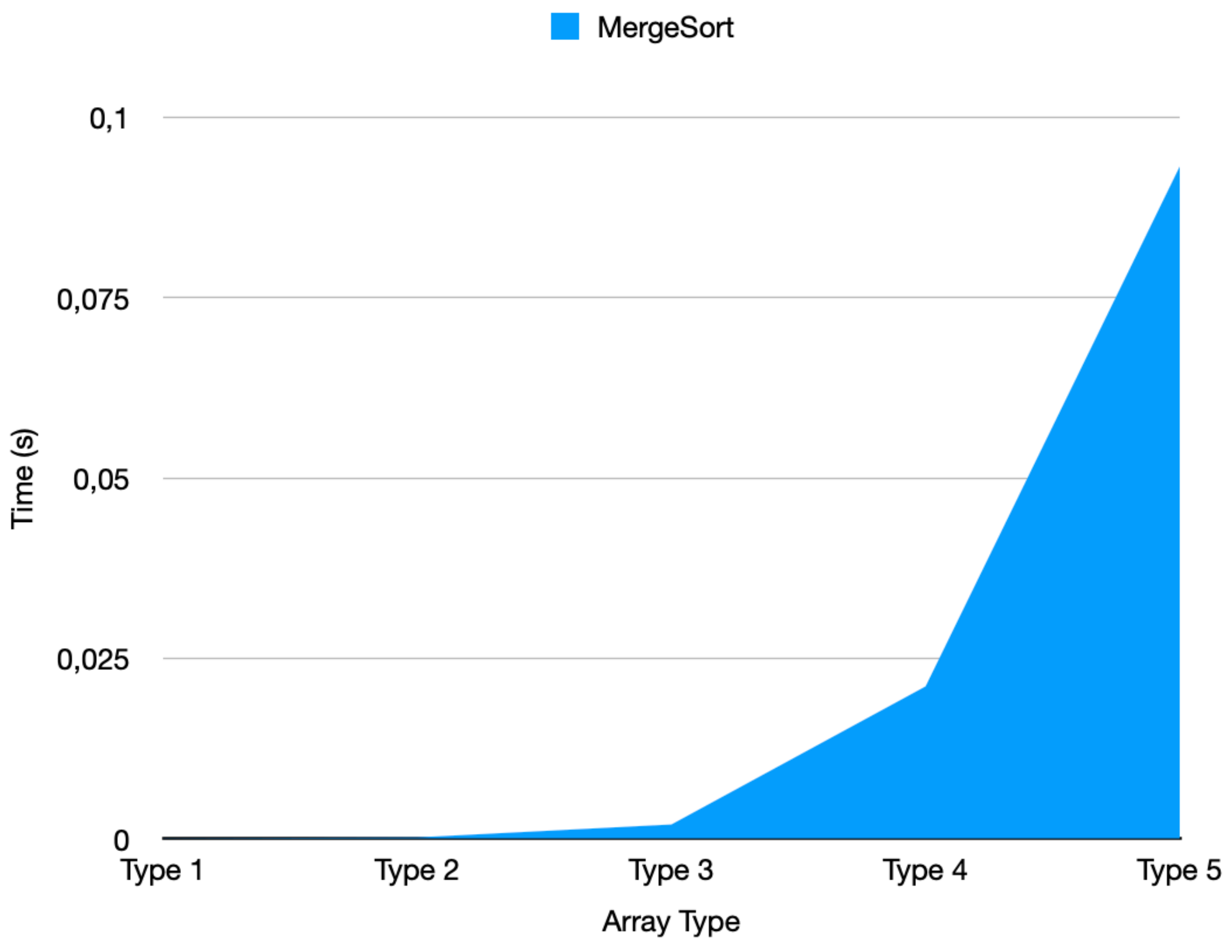


*Figure 9 Graph for the MergeSort Algorithm Performance*

The Merge Sort performance chart visually represents the algorithm's efficiency across different array types. The x-axis of the chart denotes the types of arrays, likely ranging from small to large or exhibiting varying degrees of disorder. On the y-axis, the chart illustrates the corresponding time spent by the Merge Sort algorithm to sort each array type. Typically, the chart will reveal a consistent and predictable pattern, showcasing Merge Sort's reliable O(n log n) time complexity across various scenarios. As array size or disorder increases, the chart may depict a proportional increase in the time required for sorting. Importantly, Merge Sort's stability ensures that its performance remains steady and predictable, making it a suitable choice for scenarios where maintaining the relative order of equal elements is crucial. Overall, the performance chart serves as a valuable visual tool for understanding the scalability and efficiency of the Merge Sort algorithm under different input conditions.

**HeapSort Method:**

HeapSort, a highly efficient in-place sorting algorithm, utilizes the concept of a binary heap to achieve consistent O(n log n) time complexity. The algorithm transforms the input array into a max-heap, systematically extracting the maximum element and placing it at the end of the array. This process is iteratively applied until the entire array is sorted.

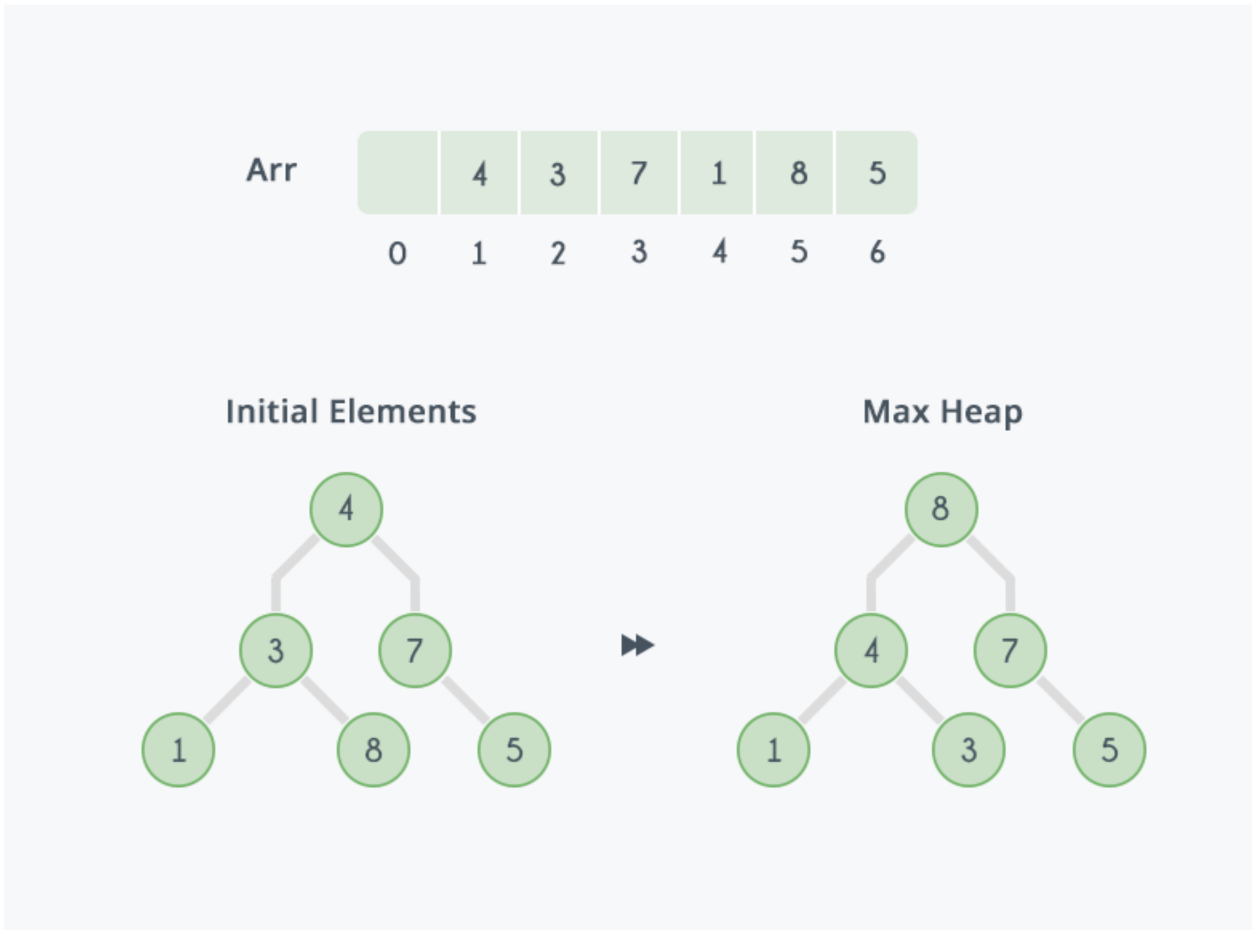


*Figure 10 Heap Sort Algorithm*

HeapSort's simplicity and stability, coupled with its guaranteed performance, make it an advantageous choice for sorting large datasets without the need for additional memory.

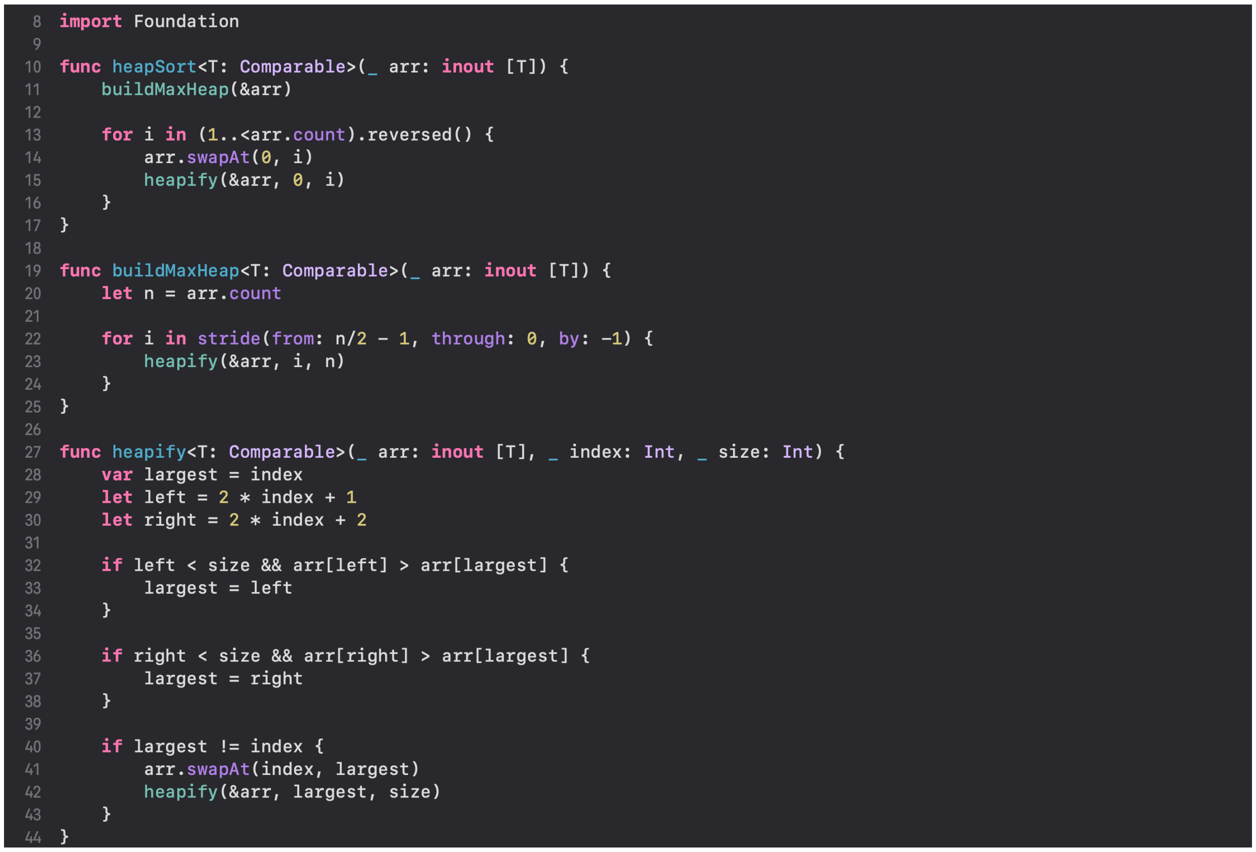
*Algorithm Description:*

Here's a pseudocode for the Heap Sort algorithm:

```
heapSort(arr)
    buildMaxHeap(arr)

    for i from length of arr - 1 down to 1
        swap arr[0] with arr[i]
        heapify(arr, 0, i)

buildMaxHeap(arr)
```

```
    for i from length of arr / 2 - 1 down to 0
        heapify(arr, i, length of arr)

heapify(arr, index, size)
    largest = index
    left = 2 * index + 1
    right = 2 * index + 2

    if left < size and arr[left] > arr[largest]
        largest = left

    if right < size and arr[right] > arr[largest]
        largest = right

    if largest != index
        swap arr[index] with arr[largest]
        heapify(arr, largest, size)
```

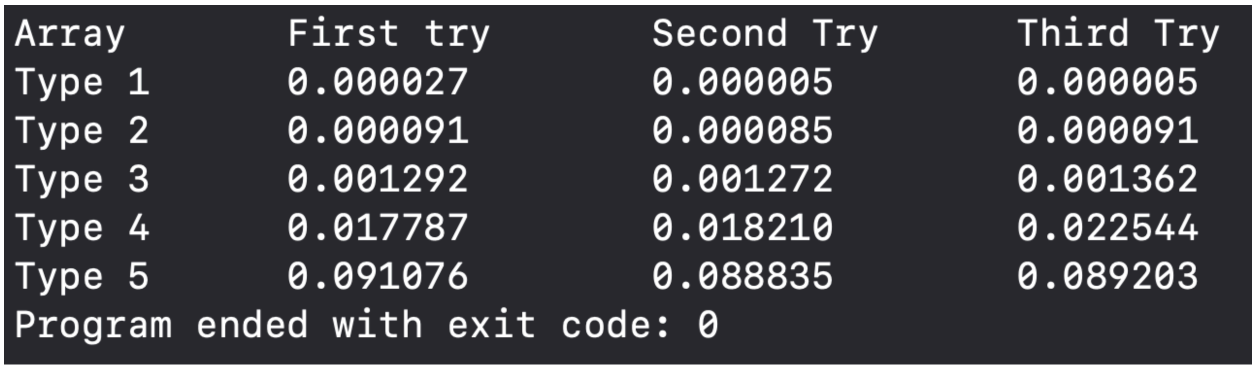
*Implementation:*



```swift
import Foundation

func heapSort<T: Comparable>(_ arr: inout [T]) {
    buildMaxHeap(&arr)

    for i in (1..<arr.count).reversed() {
        arr.swapAt(0, i)
        heapify(&arr, 0, i)
    }
}

func buildMaxHeap<T: Comparable>(_ arr: inout [T]) {
    let n = arr.count

    for i in stride(from: n/2 - 1, through: 0, by: -1) {
        heapify(&arr, i, n)
    }
}

func heapify<T: Comparable>(_ arr: inout [T], _ index: Int, _ size: Int) {
    var largest = index
    let left = 2 * index + 1
    let right = 2 * index + 2

    if left < size && arr[left] > arr[largest] {
        largest = left
    }

    if right < size && arr[right] > arr[largest] {
        largest = right
    }

    if largest != index {
        arr.swapAt(index, largest)
        heapify(&arr, largest, size)
    }
}
```

*Figure 11 HeapSort Method in Swift*

*Results:*



```
Array         First try         Second Try        Third Try
Type 1        0.000027          0.000005          0.000005
Type 2        0.000091          0.000085          0.000091
Type 3        0.001292          0.001272          0.001362
Type 4        0.017787          0.018210          0.022544
Type 5        0.091076          0.088835          0.089203
Program ended with exit code: 0
```

*Figure 12 HeapSort Method Performance*

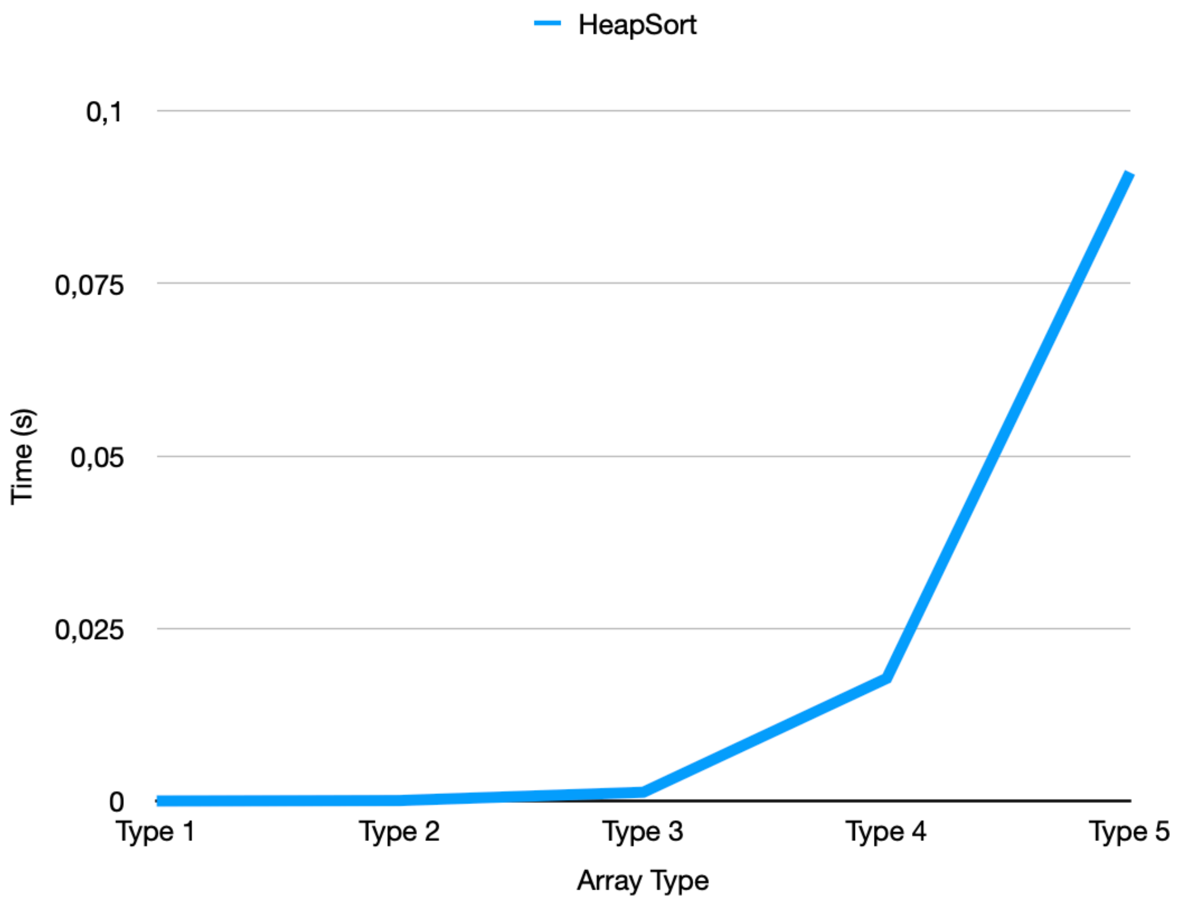


*Figure 13 Graph for HeapSort Method Performance*

As you can see, this method has an identical performance as MergeSort.

**Insertion Method:**

The Insertion Sort algorithm is a simple and intuitive sorting technique that builds the final sorted array one element at a time. It iterates through the input array, considering each element in turn and placing it in its correct position within the already sorted part of the array. This process continues until the entire array is sorted. Insertion Sort's main advantage lies in its simplicity and adaptability to small datasets or partially sorted arrays, making it efficient for those scenarios. However, its time complexity of O(n^2) in the worst case can make it less suitable for large or largely unsorted datasets when compared to more advanced algorithms.

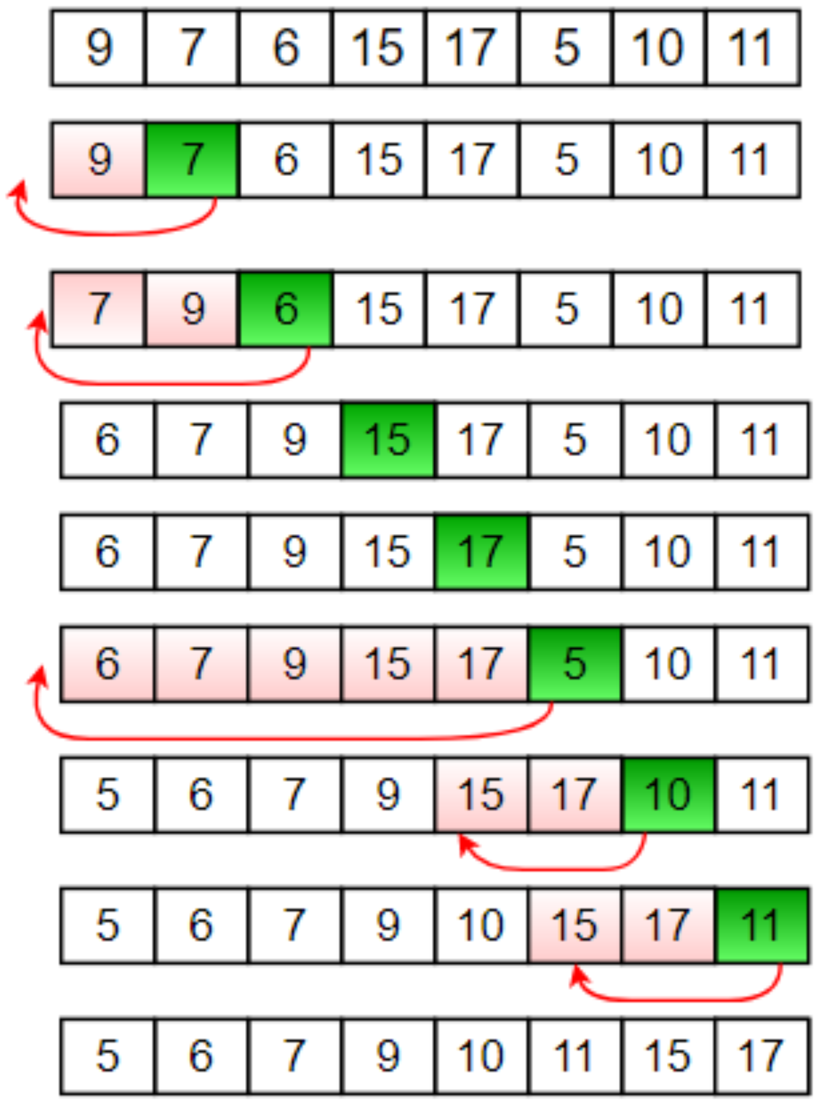


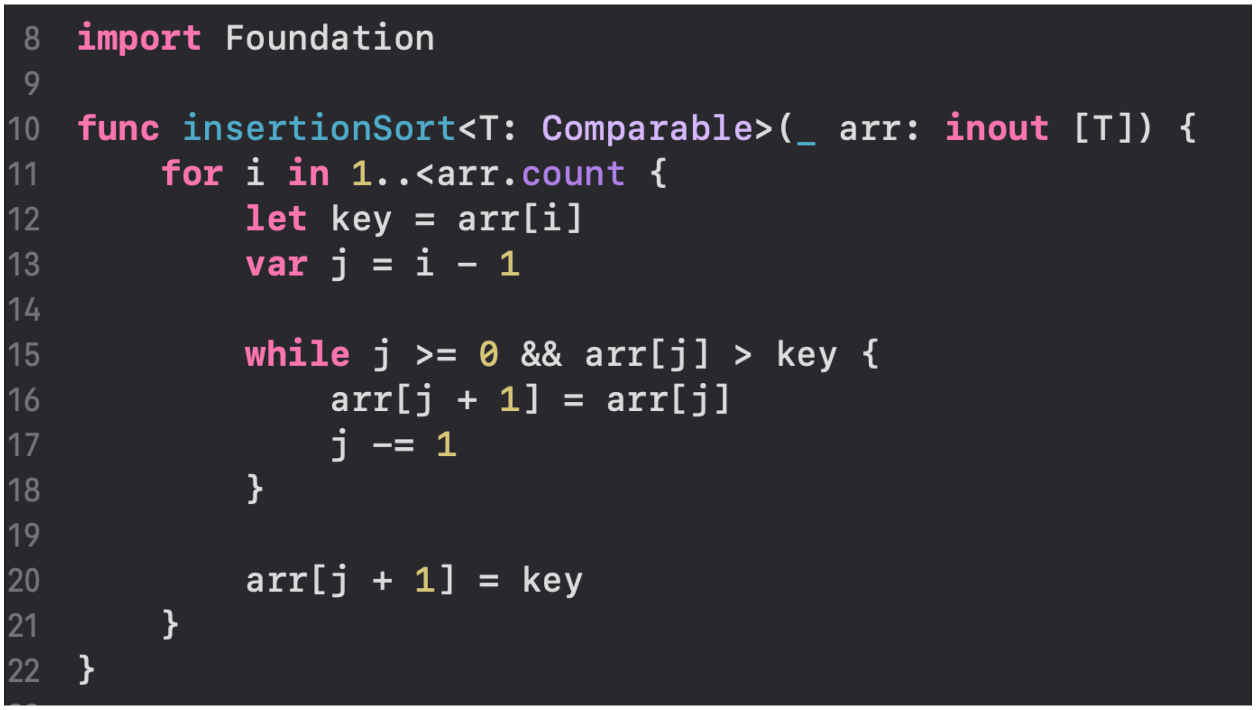
*Figure 14 Insertion Method*

*Algorithm Description:*

Here's a pseudocode for the Insertion Sort algorithm:

```
 insertionSort(arr)
for i from 1 to length of arr - 1
    key = arr[i]
    j = i - 1

    while j >= 0 and arr[j] > key
        arr[j + 1] = arr[j]
        j = j - 1

    arr[j + 1] = key
```

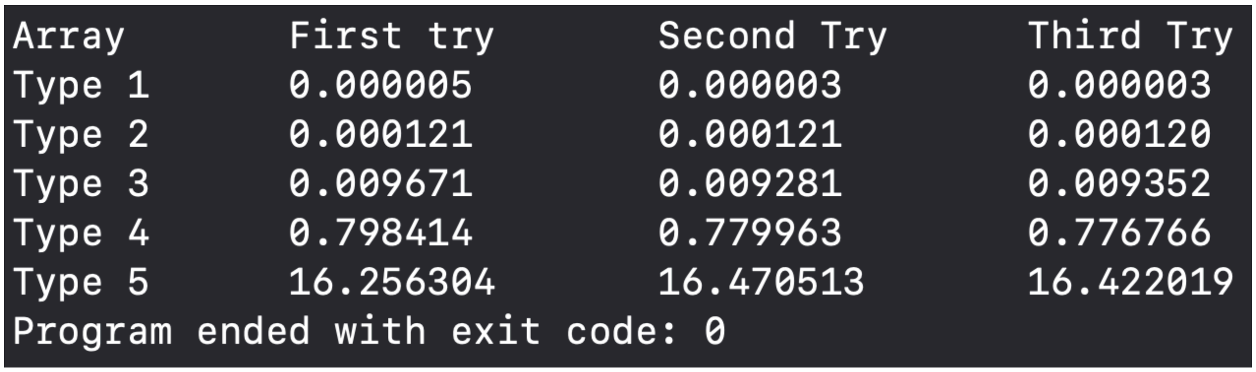
*Implementation:*



```swift
import Foundation

func insertionSort<T: Comparable>(_ arr: inout [T]) {
    for i in 1..<arr.count {
        let key = arr[i]
        var j = i - 1

        while j >= 0 && arr[j] > key {
            arr[j + 1] = arr[j]
            j -= 1
        }

        arr[j + 1] = key
    }
}
```

*Figure 15 Insertion Method in Swift*

*Results:*



```
Array         First try       Second Try      Third Try
Type 1        0.000005        0.000003        0.000003
Type 2        0.000121        0.000121        0.000120
Type 3        0.009671        0.009281        0.009352
Type 4        0.798414        0.779963        0.776766
Type 5        16.256304       16.470513       16.422019
Program ended with exit code: 0
```

*Figure 16 Insertion Method Performance*

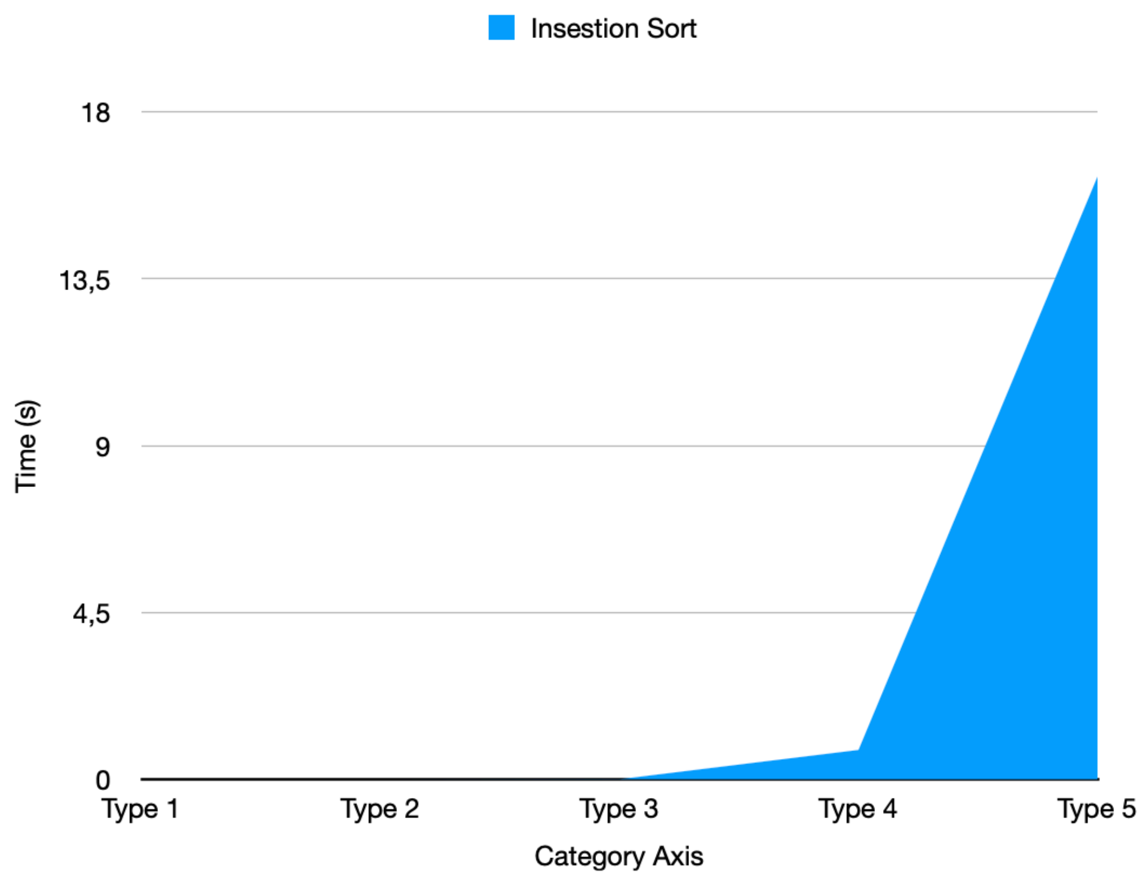


*Figure 17 Graph for Insertion Method Performance*

## CONCLUSION

In conclusion, this laboratory work has provided a comprehensive exploration into the realm of sorting algorithms, offering valuable insights into the theoretical underpinnings and practical applications of QuickSort, MergeSort, HeapSort, and InsertionSort. Our theoretical notes delved into the intricacies of each algorithm, elucidating their strengths, weaknesses, and suitability for various scenarios. The subsequent implementation and experimental analysis on arrays of different lengths illuminated the algorithms' real-world performance characteristics. QuickSort, with its efficient divide-and-conquer approach, exhibited remarkable adaptability and speed, showcasing its efficacy in diverse contexts. MergeSort, known for its stable and consistent performance, proved reliable across various array sizes. HeapSort, characterized by in-place sorting and a worst-case time complexity of O(n log n), demonstrated its utility in resource-constrained environments. On the other hand, InsertionSort, while less efficient for larger datasets, showcased its simplicity and effectiveness for smaller arrays or partially sorted data. The construction of a line chart to visualize the algorithms' execution times provided a clear comparative analysis, aiding in the identification of their relative strengths and potential trade-offs. Through this laboratory work, we not only solidified our understanding of sorting algorithms but also honed our ability to analyze, implement, and compare them in practical scenarios. This experience serves as a foundational exploration into algorithmic complexities, laying the groundwork for further studies and applications in the dynamic field of computer science.