Ministry of Education and Research of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

Laboratory work #2

Study and Empirical Analysis of Sorting Algorithms

Author:
Mihaela Catan
std. gr. FAF-231

Verified:
Cristofor Fiștic

Chișinău 2025

Contents

1 Objective

Study and analyze QuickSort, MergeSort, HeapSort and an algorithm of your choice against different inputs.

2 The Task

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.

3 Theorethical Notes

Empirical Analysis is an alternative to mathematical analysis of complexity, which helps to obtain information on the efficiency of an algorithm based on experimental observations. It is useful for:

1. Preliminary insights into the complexity class of an algorithm.
2. Comparing the performance of multiple algorithms or implementations.
3. Understanding how an algorithm behaves on a specific machine or system.

Steps in Empirical Analysis:

1. Establish the purpose of the analysis.
2. Decide the efficiency metric.
3. Establish the properties of the input data in relation to which the analysis is performed.
4. Implement the algorithms in a programming language.
5. Generate multiple sets of inputs.
6. Analyze the results.

4 Introduction

Sorting algorithms are fundamental procedures in computer science that arrange data in a specific order, typically numerical or lexicographical. They play a vital role in optimizing data searching, organizing information, and enabling efficient data processing.

5 Types of Sorting Algorithms

Sorting algorithms can be broadly classified into the following categories:

- **Comparison-based Sorting:** Algorithms that sort data by comparing pairs of elements. Examples include:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
- **Non-Comparison-based Sorting:** Algorithms that sort without direct comparisons. Examples include:
  - Counting Sort
  - Radix Sort
  - Bucket Sort
- **Hybrid Algorithms:** Algorithms that combine two or more sorting techniques. Examples include:
  - Timsort
  - IntroSort

6 Complexity Analysis

Sorting algorithms are often evaluated based on their time complexity and space complexity.

- **Time Complexity:** The number of operations performed as a function of the input size n.

  - Best Case: Optimal performance for a specific type of input.
  - Average Case: Expected performance for random input.
  - Worst Case: Maximum time the algorithm may take.

- **Space Complexity:** The amount of additional memory required by the algorithm.

7 Technical Implementation

This analysis implements four algorithms and evaluates their time and space complexities. The experiments are conducted on a backend hosted on Google Compute Engine, utilizing Python 3 and approximately 12 GB of RAM. While this environment offers a suitable platform for algorithmic analysis, several limitations and potential sources of error must be acknowledged. These include the possibility that certain algorithms may exhibit suboptimal performance on larger input sizes, as well as inherent inaccuracies in time and memory measurements due to factors such as Python's interpreter overhead and the effects of garbage collection.

7.1 Test Cases

For the analysis of sorting algorithms, the following test cases will be used:

- **Random array:** {64, 34, 25, 12, 22, 11, 90, 5, 88, 43, 55, 67, 72, 99, 31, 15, 58, 50, 33, 75, 20, 81, 13, 27, 98, 41, 37, 56, 44, 66, 39, 46, 51, 74, 59, 60, 65, 76, 23, 29}
- **Increasing-order sorted array:** {5, 11, 12, 13, 15, 20, 22, 23, 25, 27, 29, 31, 33, 34, 37, 39, 41, 43, 44, 46, 50, 51, 55, 56, 58, 59, 60, 64, 65, 66, 67, 72, 74, 75, 76, 81, 88, 90, 98, 99}
- **Decreasing-order sorted array:** {99, 98, 90, 88, 81, 76, 75, 74, 72, 67, 66, 65, 64, 60, 59, 58, 56, 55, 51, 50, 46, 44, 43, 41, 39, 37, 34, 33, 31, 29, 27, 25, 23, 22, 20, 15, 13, 12, 11, 5}
- **Array with duplicate elements:** {64, 34, 25, 12, 22, 11, 34, 5, 64, 5, 55, 67, 72, 34, 31, 15, 58, 64, 5, 75, 20, 81, 13, 27, 98, 5, 37, 64, 44, 66, 39, 5, 51, 74, 59, 60, 65, 76, 5, 29}

7.2 Quick Sort

As the name suggests, Quicksort is one of the fastest sorting algorithms.

The Quicksort algorithm takes an array of values, chooses one of the values as the *pivot* element, and moves the other values so that lower values are on the left of the pivot element, and higher values are on the right of it.

7.2.1 How it works:

1. Choose a value in the array to be the pivot element.
2. Order the rest of the array so that lower or equal values to the pivot element are on the left, and higher values are on the right.
3. Swap the pivot element with the first element of the higher values so that the pivot element lands in between the lower and higher values.
4. Repeat the same operations (recursively) for the sub-arrays on the left and right side of the pivot element.

7.2.2 Manual Run Through:

- Step 1: Start with an unsorted array: {11, 9, 12, 7, 3}
- Step 2: Choose the last value 3 as the pivot element.
- Step 3: Swap 3 with 11, resulting in: {3, 9, 12, 7, 11}
- Step 4: Sort the right sub-array with pivot 11.
- Step 5: Rearrange elements, then swap 11 with 12: {3, 9, 7, 11, 12}
- Step 6: Sort the left sub-array {9, 7} with pivot 7.
- Step 7: Swap 9 with 7: {3, 7, 9, 11, 12}
- The array is now sorted.

7.2.3 Python Implementation:

```
# Partition function
def partition(array, low, high):
    pivot = array[high]
    i = low
    j = high - 1

    while i < j:
      while i < high and array[i] <= pivot:
        i += 1
      while j > low and array[j] > pivot:
        j -= 1
      if i < j:
```

```
        array[i], array[j] = array[j], array[i]
    if array[i] > pivot:
      array[i], array[high] = array[high], array[i]
    return i

# Quick sort function
def quicksort(array, low=0, high=None):
    if high is None:
        high = len(array) - 1

    if low < high:
        pivot_index = partition(array, low, high)
        quicksort(array, low, pivot_index-1)
        quicksort(array, pivot_index+1, high)
```

7.2.4 Performance Results:

- Random array: 6.0558×10^{-5} seconds
- Increasing-order sorted array: 8.7499×10^{-5} seconds
- Decreasing-order sorted array: 8.7499×10^{-5} seconds
- Array with duplicate elements: 4.4345×10^{-5} seconds

The worst-case scenario for Quicksort is $O(n^2)$. This occurs when the pivot element is either the highest or lowest value in every sub-array, which results in excessive recursive calls. This typically happens when the array is already sorted.

On average, the time complexity is $O(n \log n)$, making Quicksort highly efficient in practice. The space complexity is $O(\log n)$ due to the recursive call stack.

7.3 Optimized Quick Sort

As an improvement over the standard Quicksort algorithm, tail recursion optimization can be applied to reduce the recursion depth and improve performance on large datasets.

The smaller partition is always sorted first using recursion, while the larger partition is sorted using a while loop instead of recursion. This reduces the maximum recursion depth from $O(n)$ to $O(\log n)$, preventing stack overflow.

7.3.1 Python Implementation:

```
# Quick sort with Tail Recursion Optimization
def quicksort_tail(array, low=0, high=None):
    if high is None:
        high = len(array) - 1

    while low < high:
        pivot_index = partition(array, low, high)

        # Sort the smaller partition recursively
        if pivot_index - low < high - pivot_index:
            quicksort_tail(array, low, pivot_index - 1)
            low = pivot_index + 1  # Tail recursion (
               converted to loop)
        else:
            quicksort_tail(array, pivot_index + 1, high)
            high = pivot_index - 1  # Tail recursion (
               converted to loop)
```

7.3.2 Performance Results:

The results of the optimized Quick Sort algorithm on the test cases are presented below:

- Random array: 4.6015×10^{-5} seconds
- Increasing-order sorted array: 7.7486×10^{-5} seconds
- Decreasing-order sorted array: 8.0109×10^{-5} seconds
- Array with duplicate elements: 3.8862×10^{-5} seconds

7.4 Merge Sort

The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.

Divide: The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

Conquer: The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.

The breaking down and building up of the array to sort the array is done recursively.

7.4.1 How it works:

1. Divide the unsorted array into two sub-arrays, half the size of the original.
2. Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
3. Merge two sub-arrays together by always putting the lowest value first. Keep merging until there are no sub-arrays left.

7.4.2 Manual Run Through:

- Step 1: Start with an unsorted array: {12, 8, 9, 3, 11, 5, 4}
- Step 2: Split the array into two halves: {12, 8, 9} and {3, 11, 5, 4}
- Step 3: Recursively split the arrays further: {12}, {8, 9}, {3, 11}, {5, 4}
- Step 4: Merge {8} and {9} into {8, 9}, then merge {12} with {8, 9} into {8, 9, 12}
- Step 5: Split {5, 4} and merge into {4, 5}
- Step 6: Merge {3, 11} into {3, 11} and then merge {3, 11} with {4, 5} into {3, 4, 5, 11}
- Step 7: Merge the final two arrays {8, 9, 12} and {3, 4, 5, 11} into {3, 4, 5, 8, 9, 11, 12}

7.4.3 Python Implementation:

```
def merge_sort(arr):
  if len(arr) > 1:
    left_arr = arr[:len(arr)//2]
    right_arr = arr[len(arr)//2:]

    # Recursion
    merge_sort(left_arr)
    merge_sort(right_arr)

    # Merging
    i = 0
    j = 0
    k = 0
```

```
    while i < len(left_arr) and j < len(right_arr):
      if left_arr[i] < right_arr[j]:
        arr[k] = left_arr[i]
        i += 1
      else:
        arr[k] = right_arr[j]
        j += 1
      k += 1

    while i < len(left_arr):
      arr[k] = left_arr[i]
      i += 1
      k += 1

    while j < len(right_arr):
      arr[k] = right_arr[j]
      j += 1
      k += 1
```

7.4.4 Performance Results:

- Random array: 8.1539×10^{-5} seconds
- Increasing-order sorted array: 5.2929×10^{-5} seconds
- Decreasing-order sorted array: 6.1750×10^{-5} seconds
- Array with duplicate elements: 8.0347×10^{-5} seconds

The time complexity for Merge Sort is $O(n \log n)$ and remains consistent across different input types. The space complexity is $O(n)$ due to the temporary arrays used during merging.

7.5 Optimized Merge Sort

As an optimization, a hybrid approach can be applied where Insertion Sort is used for small arrays, reducing the overhead of recursion.

7.5.1 Python Implementation:

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


def merge_sort_optimized(arr):
  if len(arr) <= 20:
    insertion_sort(arr)
    return

  if len(arr) > 1:
    mid = len(arr) // 2
    left_arr = arr[:mid]
    right_arr = arr[mid:]

    merge_sort_optimized(left_arr)
    merge_sort_optimized(right_arr)

    i = j = k = 0
    while i < len(left_arr) and j < len(right_arr):
      if left_arr[i] <= right_arr[j]:
        arr[k] = left_arr[i]
        i += 1
      else:
        arr[k] = right_arr[j]
        j += 1
      k += 1

    arr[k:] = left_arr[i:] + right_arr[j:]
```

7.5.2 Performance Results:

- Random array: 7.0810×10^{-5} seconds

- Increasing-order sorted array: 2.2888×10^{-5} seconds
- Decreasing-order sorted array: 6.4373×10^{-5} seconds
- Array with duplicate elements: 4.2438×10^{-5} seconds

The optimized version improves performance on small arrays by applying Insertion Sort, reducing the recursion depth and speeding up the sorting process.

7.6 Heap Sort

Heap Sort is an efficient sorting technique based on the heap data structure.

The heap is a nearly-complete binary tree where the parent node could either be minimum or maximum. The heap with the minimum root node is called a min-heap, and the root node with the maximum root node is called a max-heap. The elements in the input data of the heap sort algorithm are processed using these two methods.

7.6.1 How it works:

1. Build a max heap from the input array.
2. Swap the root element (maximum value) with the last element of the heap.
3. Reduce the heap size by one and heapify the root element to restore the heap property.
4. Repeat the process until the heap size becomes one.

7.6.2 Manual Run Through:

- Step 1: Start with an unsorted array: {11, 9, 12, 7, 3}
- Step 2: Build the max heap: {12, 9, 11, 7, 3}
- Step 3: Swap 12 with 3: {3, 9, 11, 7, 12}
- Step 4: Restore heap property: {11, 9, 3, 7, 12}
- Step 5: Swap 11 with 3: {3, 9, 7, 11, 12}
- Step 6: Restore heap property: {9, 7, 3, 11, 12}
- Step 7: Swap 9 with 3: {3, 7, 9, 11, 12}
- The array is now sorted.

7.6.3 Python Implementation:

```
def sift_down(arr, i, upper):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < upper and arr[left] > arr[largest]:
        largest = left

    if right < upper and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        sift_down(arr, largest, upper)


def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        sift_down(arr, i, n)

    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        sift_down(arr, 0, i)
```

7.6.4 Performance Results:

- Random array: 1.0753×10^{-4} seconds
- Increasing-order sorted array: 8.5354×10^{-5} seconds
- Decreasing-order sorted array: 6.8903×10^{-5} seconds
- Array with duplicate elements: 6.9857×10^{-5} seconds

Heap Sort has a time complexity of $O(n \log n)$ in all cases. The log n factor comes from the height of the binary heap, ensuring good performance for large datasets. The space complexity is $O(\log n)$ due to the recursive call stack, but it can be optimized to $O(1)$ with an iterative implementation.

7.7 Optimized Heap Sort

An optimized version of Heap Sort can be implemented to reduce the number of swaps, which improves the performance by storing the root element temporarily and moving larger children up directly.

7.7.1 Python Implementation:

```
def sift_down_optimized(arr, i, upper):
    largest = i
    temp = arr[i]

    while True:
        left = 2 * i + 1
        right = 2 * i + 2
        if left < upper and arr[left] > temp:
            largest = left
        if right < upper and arr[right] >= arr[largest]:
            largest = right

        if largest == i:
            break

        arr[i] = arr[largest]
        i = largest

    arr[i] = temp


def heap_sort_optimized(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        sift_down_optimized(arr, i, n)

    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        sift_down_optimized(arr, 0, i)
```

7.7.2 Performance Results:

- Random array: 4.8161×10^{-5} seconds
- Increasing-order sorted array: 3.5763×10^{-5} seconds
- Decreasing-order sorted array: 2.9564×10^{-5} seconds
- Array with duplicate elements: 3.4094×10^{-5} seconds

The optimized Heap Sort reduces the number of swaps, making it faster for larger arrays. The time complexity remains $O(n \log n)$, but the practical performance shows significant improvement.

7.8 Shell Sort

Shell Sort is a variation of the Insertion Sort algorithm. In Insertion Sort, elements are moved only one position ahead at a time, which can be inefficient when elements need to be moved far ahead. Shell Sort improves this by allowing the exchange of distant elements.

The algorithm works by sorting elements at a specific gap interval and gradually reducing the gap until it becomes 1. An array is said to be *h-sorted* if all sublists of every h'th element are sorted.

7.8.1 How it works:

1. Start with a large gap value (usually half the array length).
2. Sort elements at each gap interval.
3. Reduce the gap and repeat until the gap becomes 1.
4. Perform the final iteration with gap 1, which acts as a standard insertion sort.

7.8.2 Manual Run Through:

- Step 1: Start with an unsorted array: {12, 34, 54, 2, 3}
- Step 2: Initial gap = 2, sort sublists {12, 54, 3} and {34, 2}
- Step 3: Rearrange to {12, 2, 3, 34, 54}
- Step 4: Reduce gap to 1 and perform insertion sort: {2, 3, 12, 34, 54}
- The array is now sorted.

7.8.3 Python Implementation:

```
def shell_sort(arr):
    n = len(arr)
    gap = n // 2

    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
```

7.8.4 Performance Results:

- Random array: 4.6015×10^{-5} seconds
- Increasing-order sorted array: 2.2888×10^{-5} seconds
- Decreasing-order sorted array: 3.3140×10^{-5} seconds
- Array with duplicate elements: 3.5763×10^{-5} seconds

The time complexity of Shell Sort is $O(n^2)$ in its basic form, but it can be optimized with different gap sequences. The best-case time complexity is $O(n \log n)$, while the average case ranges between $O(n \log n)$ and $O(n^{1.25})$. The space complexity is $O(1)$ since the algorithm sorts in place.

7.9 Optimized Shell Sort

To optimize Shell Sort, the Knuth sequence is commonly used, where the gap is calculated as $3k + 1$. This method reduces the number of comparisons and improves overall performance.

7.9.1 Python Implementation:

```
def shell_sort_optimized(arr):
    n = len(arr)
    gap = 1
```

```
    while gap < n // 3:
        gap = 3 * gap + 1

    while gap >= 1:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 3
```

7.9.2 Performance Results:

- Random array: 3.7670×10^{-5} seconds
- Increasing-order sorted array: 1.3828×10^{-5} seconds
- Decreasing-order sorted array: 1.9550×10^{-5} seconds
- Array with duplicate elements: 2.0504×10^{-5} seconds

The optimized Shell Sort algorithm reduces the maximum number of comparisons, making it faster in practice. The use of the Knuth sequence enhances performance on large datasets while maintaining the same space complexity of $O(1)$.

Analysis of Sorting Algorithms Performance

The obtained results of the sorting algorithms' performance, as depicted in the scatter plot, reveal significant differences in execution time depending on both the algorithm type and the input data pattern. Here's an analysis of the results:

General Observations

- Optimized variants consistently outperform their standard counterparts across all test cases, confirming the effectiveness of the applied optimizations.
- The performance order from fastest to slowest is typically:

  - Optimized Shell Sort
  - Optimized Heap Sort
  - Optimized Quick Sort
  - Optimized Merge Sort

- Arrays with duplicate elements are sorted fastest by most algorithms due to their reduced complexity.

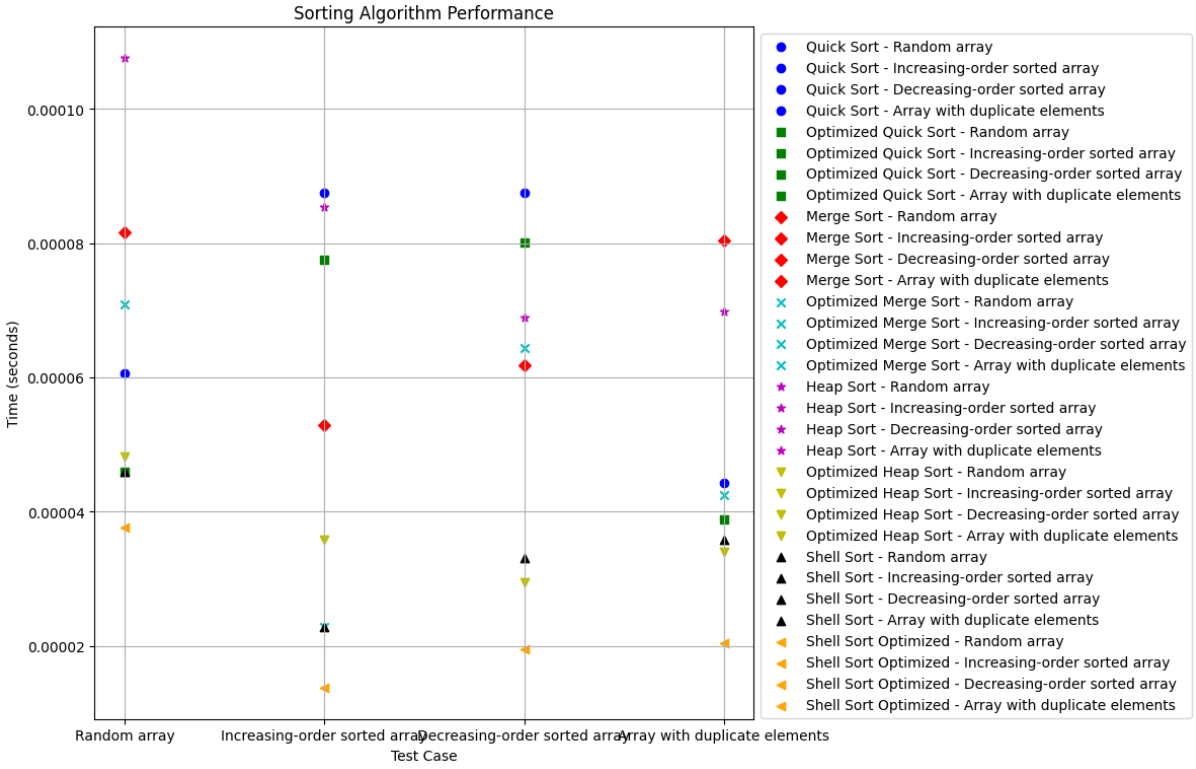

Figure 1: Algorithm Comparison

Case-by-Case Analysis

Random Array

- Optimized Shell Sort shows the best performance.
- Optimized Merge Sort and Quick Sort are slower but still competitive.
- Heap Sort shows slightly higher execution times compared to others.

Increasing-order Sorted Array

- Optimized Shell Sort and Optimized Heap Sort perform exceptionally well, indicating their adaptability to nearly sorted data.

- Quick Sort, Merge Sort, and Heap Sort show slower performance on sorted arrays.

Decreasing-order Sorted Array

- Optimized Shell Sort outperforms other algorithms.
- Heap Sort remains consistent across different data patterns.
- Quick Sort shows one of its worst performances due to its vulnerability to already sorted arrays without random pivots.

Array with Duplicate Elements

- Optimized Shell Sort and Optimized Heap Sort perform best.
- Quick Sort maintains stable performance.
- Merge Sort, without optimizations, shows slower results.

Conclusion

Shell Sort with gap sequence optimizations emerges as the fastest algorithm overall, especially for nearly sorted arrays and arrays with duplicate elements. However, Merge Sort remains the most consistent across all test cases, while Quick Sort benefits greatly from optimizations in random data scenarios.

8 Bibliography

- The code is available at: Google Colab Notebook
- GitHub Repository: GitHub