Data Algorithms and Representation

Escuela Colombiana de Ingeniería Julio Garavito

Analysis of Sorting Algorithms

Sebastian Cardona Parra

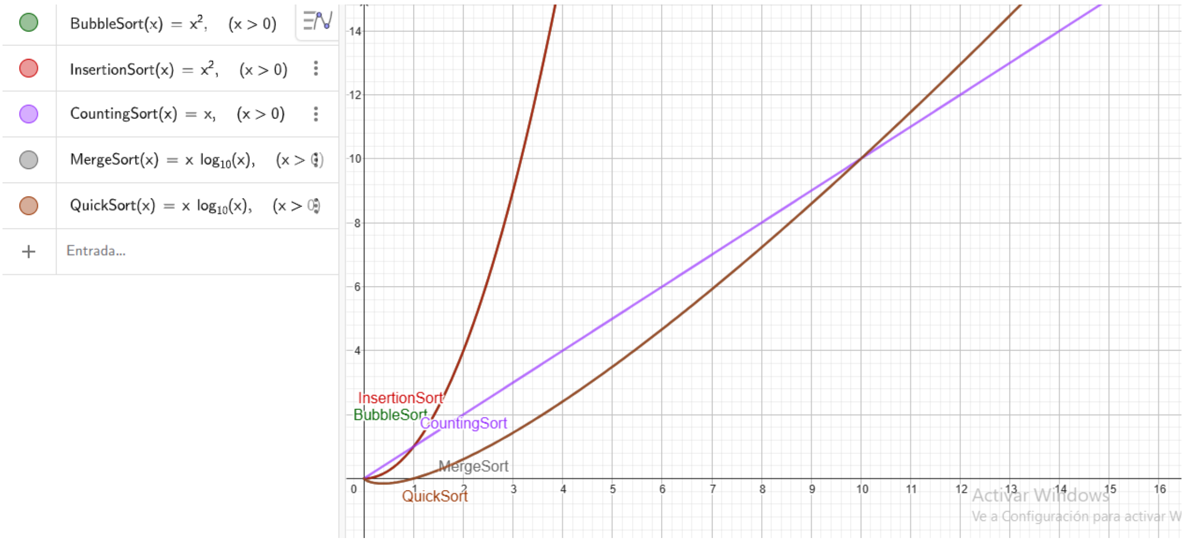
Professor: Rafael Alberto Niquefa Velasquez

2025-1

Table of Contents

Introduction

Sorting data is a fundamental task in computer science, and multiple algorithms have been developed to optimize this process. This report analyzes five sorting algorithms: Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, and Counting Sort, comparing their efficiency in terms of execution time and computational complexity.



Algorithms

Bubble Sort

Bubble Sort is a simple sorting algorithm that works by comparing adjacent elements in a list and swapping them into the correct order. This process is repeated until the entire list is sorted.

```

# n is the size of arr
def bubble_sort(arr):  2 usages  SebastianCardona-P
    n = len(arr)  # O(1)
    for i in range(n):  # O(n)
        for j in range(n - i - 1):  # O(n)
            if arr[j] > arr[j + 1]:  # O(1)
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # O(1)
    return arr  # O(1)


# O(bubble_sort) = O(1 + n * n + 1 + 1)
# = O(n^2 + 3) = O(n^2)
```

Algorithmic Complexity

Worst case (list in reverse order): $O(n^2)$

Best case (list already sorted): $O(n)$

Average case $O(n^2)$

Characteristics

This algorithm is easy to understand and implement but inefficient when used on large lists.

Merge Sort

Merge Sort is a recursive sorting algorithm that follows the "divide and conquer" paradigm, dividing the list into two parts, sorting them independently, and then merging them in order.

```python
# n is the size of arr
def merge_sort(arr):  4 usages  SebastianCardona-P
    if len(arr) > 1:  # O(1)
        mid = len(arr) // 2  # O(1)
        left = arr[:mid]  # O(n)
        right = arr[mid:]  # O(n)

        merge_sort(left)  # O(T(n/2))
        merge_sort(right)  # O(T(n/2))

        i = j = k = 0  # O(1)
        # merge arrays
        while i < len(left) and j < len(right):  # O(n)
            if left[i] < right[j]:  # O(1)
                arr[k] = left[i]  # O(1)
                i += 1  # O(1)
            else:
                arr[k] = right[j]  # O(1)
                j += 1  # O(1)
            k += 1  # O(1)

        # Put the remainder of the left array
        while i < len(left):  # O(n)
            arr[k] = left[i]  # O(1)
            i += 1  # O(1)
            k += 1  # O(1)
```

```python
        # Put the remainder of the right array
        while j < len(right):  # O(n)
            arr[k] = right[j]  # O(1)
            j += 1  # O(1)
            k += 1  # O(1)

    return arr  # O(1)


# O(merge_sort) = Master theorem
# T(n) = 2 * T(n/2) + O(n)
# A = 2, B = 2, C = 1
# log_B(A) = log_2(2) = 1
# O(merge_sort) = O(n * log(n))
```

Algorithmic Complexity

This is a recursive algorithm and follows the divide and conquer paradigms, So the master theorem was used.

Worst case $O(n \log n)$

Best case $O(n \log n)$

Average case O(n log n)

Characteristics

More efficient for large datasets and works well with linked data structures but requires additional memory for sub lists.

Insertion Sort

Insertion Sort is an efficient algorithm for small lists. It assumes the first element is already sorted, then takes an element from the unsorted part and inserts it correctly into the sorted part, repeating this process.

```python
# n is the size of arr
def insertion_sort(arr):
    for i in range(1, len(arr)):  # O(n)
        key = arr[i]  # O(1
        j = i - 1  # O(1)

        while j >= 0 and key < arr[j]:  # O(n)
            arr[j + 1] = arr[j]  # O(1)
            j -= 1  # O(1)

        arr[j + 1] = key  # O(1)

    return arr  # O(1)
```

Algorithmic Complexity

Worst case (list in reverse order) O(n^2)

Best case (list already sorted) O(n)

Average case O(n^2)

Characteristics

It is a simple and efficient algorithm for small and nearly sorted list but inefficient for larger list.

Quick Sort

Quick Sort is one of the most efficient sorting algorithms. Like Merge Sort, it follows the divide-and-conquer paradigm but uses memory more efficiently. A pivot is chosen, elements smaller than the pivot are placed behind it, and elements larger than the pivot are placed ahead. Quick Sort is then applied recursively to both sides.

```python
# n is the size of arr
def quick_sort(arr):  4 usages  SebastianCardona-P
    if len(arr) <= 1:  # O(1)
        return arr  # O(1)

    pivot = arr[len(arr) // 2]  # O(1)
    left = [x for x in arr if x < pivot]  # O(n)
    middle = [x for x in arr if x == pivot]  # O(n)
    right = [x for x in arr if x > pivot]  # O(n)
    return quick_sort(left) + middle + quick_sort(right)  # O(T(n/2))


# O(quick_sort) = Master theorem
# T(n) = 2 * T(n/2) + O(n)
# A = 2, B = 2, C = 1
# log_B(A) = log_2(2) = 1
# O(quick_sort) = O(n * log(n))
```

Algorithmic Complexity

The Master theorem was used, because Quick sort is a recursive and divide and conquer algorithm.

Worst case (Choosing the worst pivot, largest or smaller element in the list) $O(n^2)$

Best case O(n log n)

Average case O(n log n)

Characteristics

Efficient for large datasets and does not require significant extra memory, but care must be taken in pivot selection to avoid increased complexity.

Counting Sort

Counting Sort is an efficient sorting algorithm when working with numbers within a known and non-negative range. It counts the frequency of each element and uses that information to reconstruct the sorted list. However, this algorithm requires creating a new list with a size equal to the highest value in the given array, making it inefficient for large maximum values.

```python
# n is the size of arr
def counting_sort(arr):

    if len(arr) == 0:
        return arr  # O(1)

    # Extract the maximum value from the list
    max_value = max(arr)  # O(n)

    # Create a list to store the count of each element
    count = [0] * (max_value + 1)  # O(k)

    # Count the number of times each element appears
    for number in arr:  # O(n)
        count[number] += 1  # O(1)

    new_arr = []  # O(1)

    for i in range(max_value + 1):  # O(k)
        new_arr.extend([i] * count[i])  # O(1)

    return new_arr  # O(1)


# O(counting_sort) =
# O(n + k + n + k + 1) = O(2n + 2k + 1) = O(n + k)
```

Algorithmic Complexity

N is the array size; k is the maximum number in the array

Worst case O(n + k)

Best case O(n + k)

Average case O(n + k)

Characteristics

More efficient than Quick Sort if the value range of k is small compared to n, also is useful for sorting large volumes of data within a limited range of k, but inefficient if k has a very high range.

Analysis and Graphs

Analysis methodology

A series of Python modules were implemented for testing. First, a data generation module was created to generate a list of random values given a size and an upper limit (MAX_VALUE).

```python
import random
from Sort import constants


def get_random_list(size, limit=constants.MAX_VALUE):  5 usages  SebastianCardona-P
    return [random.randint( a: 0, limit) for _ in range(size)]

```

Another module was used to measure the execution time of each algorithm for a series of randomly generated datasets of different sizes.

```python
import time


from Sort import algorithms
from Sort import constants
from Sort import data_generator


def take_execution_time(minimum_size, maximum_size, step, samples_by_size):
    return_table = []

    for size in range(minimum_size, maximum_size + 1, step):
        table_row = [size]
        times = take_times(size, samples_by_size)
        return_table.append(table_row + times)

    return return_table

```

```python
"""
    It will return five values, one per algorithm: The execution time
"""


def take_times(size, samples_by_size):  1 usage  SebastianCardona-P
    samples = [
        data_generator.get_random_list(size) for _ in range(samples_by_size)
    ]

    return [
        take_time_for_algorithm(samples, algorithms.bubble_sort),
        take_time_for_algorithm(samples, algorithms.insertion_sort),
        take_time_for_algorithm(samples, algorithms.merge_sort),
        take_time_for_algorithm(samples, algorithms.quick_sort),
        take_time_for_algorithm(samples, algorithms.counting_sort),
    ]

```

```python
    """
    Returns the median of the execution time
    """


def take_time_for_algorithm(samples, algorithm):  5 usages  SebastianCardona-P
    times = []

    for sample in samples:
        start = time.time()
        algorithm(sample.copy())
        end = time.time()
        times.append(constants.TIME_MULTIPLIER * (end - start))

    times.sort()
    return times[len(times) // 2]
```

Finally, from the main function "app.py," all parameters were set, and data was collected.

```python
from Sort import execution_time_gathering


if __name__ == "__main__":
    minimum_size = 100
    maximum_size = 500
    step = 50
    samples_by_size = 10

    table = execution_time_gathering.take_execution_time(
        minimum_size, maximum_size, step, samples_by_size
    )

    print("Size | BubbleSort | InsertionSort | MergeSort | QuickSort | CountingSort")
    for row in table:
        print(row)
```

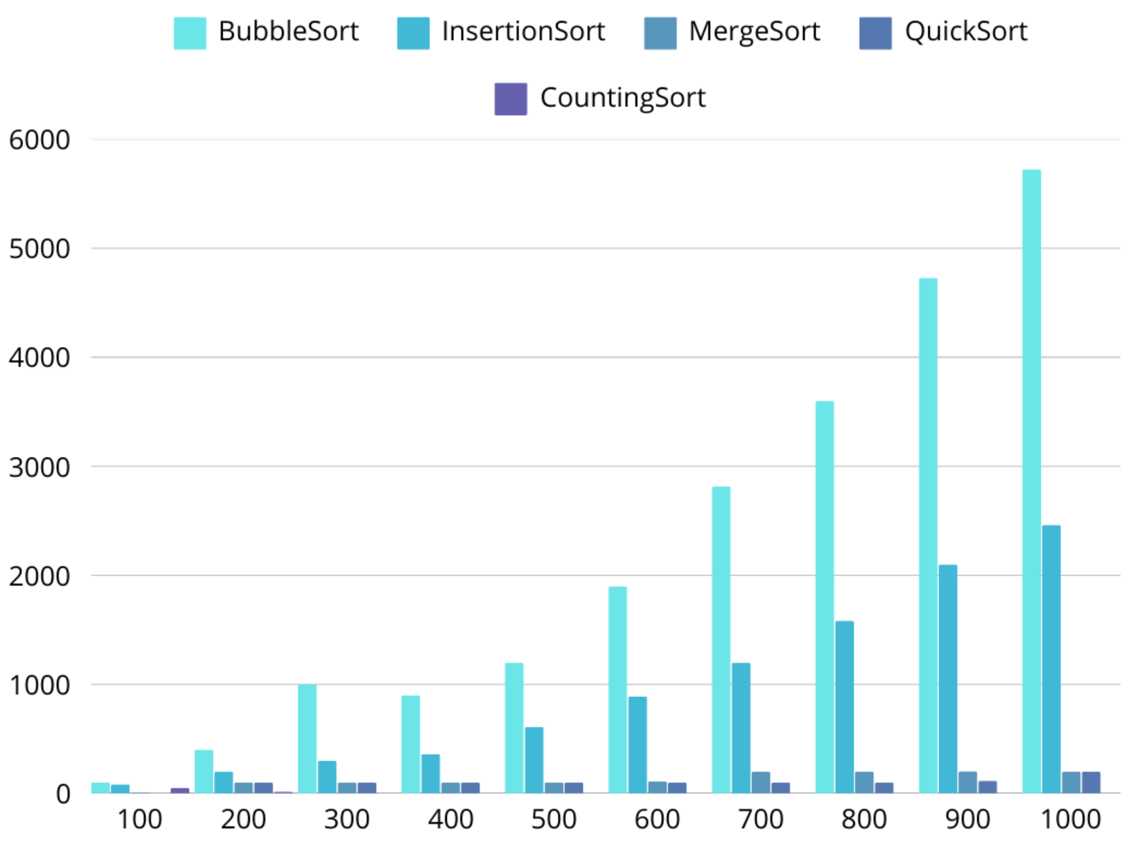
Data collection:

Times are given in hundredths of milliseconds.

1. For lists of minimum size 100 and maximum size 1000, with 10 test cases per size and MAX_VALUE = 1000:

```
"C:\Users\Sebastian\Desktop\Universidad\Octavo Semestre\ALDA_M\Tarea1\SortAlgorithms\.venv\Scripts\python.
Size | BubbleSort | InsertionSort | MergeSort | QuickSort | CountingSort
[100, 99.945068359375, 83.160400390625, 12.159347534179688, 0.0, 49.85332489013672]
[200, 399.8756408691406, 199.7232437133789, 99.7781753540039, 100.01659393310547, 16.164779663085938]
[300, 999.8321533203125, 300.1213073730469, 99.49207305908203, 99.87354278564453, 0.0]
[400, 899.9347686767578, 361.5856170654297, 99.96891021728516, 99.99275207519531, 0.0]
[500, 1200.3660202026367, 608.7779998779297, 99.99275207519531, 99.99275207519531, 0.0]
[600, 1900.0530242919922, 889.2297744750977, 110.81695556640625, 100.08811950683594, 0.0]
[700, 2815.580368041992, 1199.3408203125, 199.91397857666016, 100.08811950683594, 0.0]
[800, 3599.9536514282227, 1583.1708908081055, 199.9378204345703, 100.1119613647461, 0.0]
[900, 4727.911949157715, 2099.895477294922, 202.48889923095703, 115.27538299560547, 0.0]
[1000, 5724.740028381348, 2462.2201919555664, 200.20008087158203, 199.60403442382812, 0.0]

Process finished with exit code 0
```



The worst-performing functions in terms of time and space are those with a complexity of O(n^2), namely Bubble Sort and Insertion Sort. The larger the list, the longer it takes to sort.

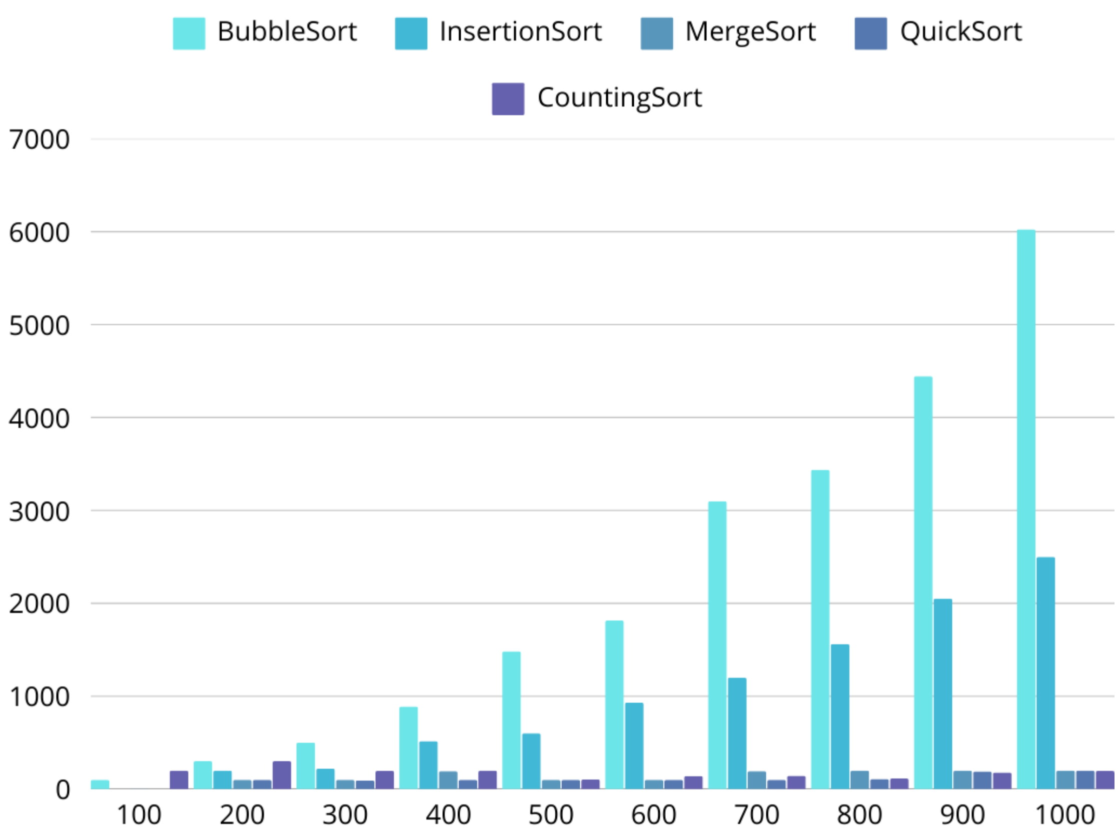
Regarding recursive algorithms, they significantly outperform the previous two when the list size is large, despite increasing execution time with list size, this increase is not proportional.

Regarding Counting Sort when the largest number in the list was 1000, it was initially observed that with a list size of 100 or 200, the execution time was longer than for sizes above 500. This is because Counting Sort is more efficient when the range of numbers in the list closely matches its size.

2. For lists of minimum size 100 and maximum size 1000, with 10 test cases per size and MAX_VALUE = 10000:

```
"C:\Users\Sebastian\Desktop\Universidad\Octavo Semestre\ALDA_M\Tarea1\SortAlgorithms\.venv\Scripts\python.exe" "
Size | BubbleSort | InsertionSort | MergeSort | QuickSort | CountingSort
[100, 99.89738464355469, 0.0, 5.6743621826171875, 0.0, 200.17623901367188]
[200, 300.0020980834961, 200.00934600830078, 99.99275207519531, 99.82585906982422, 299.9544143676758]
[300, 500.0114440917969, 219.96498107910156, 100.01659393310547, 92.6971435546875, 196.4569091796875]
[400, 889.8019790649414, 516.510009765625, 194.14424896240234, 99.945068359375, 199.9378204345703]
[500, 1483.0350875854492, 600.0280380249023, 99.945068359375, 100.04043579101562, 103.5928726196289]
[600, 1814.2938613891602, 930.6192398071289, 101.01795196533203, 100.04043579101562, 141.1914825439453]
[700, 3100.0375747680664, 1199.9845504760742, 194.57340240478516, 100.4934310913086, 143.40877532958984]
[800, 3437.685966491699, 1561.0456466674805, 200.00934600830078, 107.62214660644531, 114.58396911621094]
[900, 4444.83757019043, 2051.9256591796875, 200.00934600830078, 190.32955169677734, 180.12523651123047]
[1000, 6025.838851928711, 2499.842643737793, 200.12855529785156, 200.00934600830078, 195.4793930053711]

Process finished with exit code 0
```

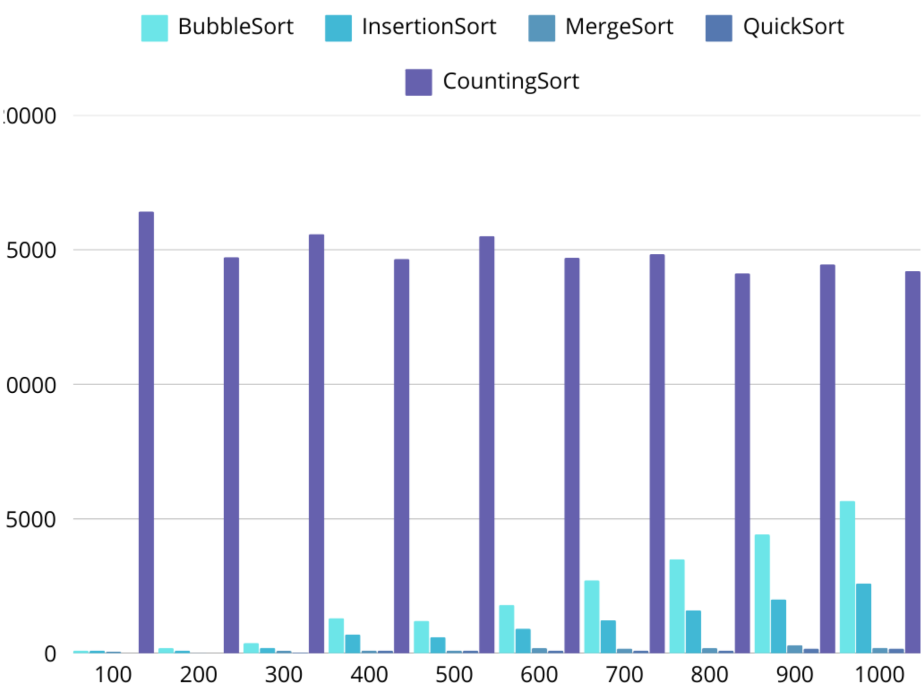


Changing the maximum number to ten thousand, the execution time of all algorithms remained the same except for Counting Sort.

Since Counting Sort depends on the maximum range, increasing it made the algorithm less effective.

3. For lists of minimum size 100 and maximum size 1000, with 10 test cases per size and MAX_VALUE = 1000000:

```
"C:\Users\Sebastian\Desktop\Universidad\Octavo Semestre\ALDA_M\Tarea1\SortAlgorithms\.venv\Scripts\python.exe" '
Size | BubbleSort | InsertionSort | MergeSort | QuickSort | CountingSort
[100, 105.8816909790039, 99.7304916381836, 62.537193298339844, 0.0, 16420.12596130371]
[200, 192.1892166137695З, 99.96891021728516, 8.893013000488281, 0.0, 14727.044105529785]
[300, 387.5017166137695З, 199.98550415039062, 93.22166442871094, 25.62999725341797, 15585.27946472168]
[400, 1310.4915618896484, 700.3545761108398, 101.6378402709961, 100.13580322265625, 14660.811424255371]
[500, 1205.9688568115234, 599.980354309082, 100.01659393310547, 99.945068359375, 15511.894226074219]
[600, 1800.1794815063477, 917.6731109619141, 194.1204071044922, 100.06427764892578, 14701.17568969726б]
[700, 2712.6073837280273, 1229.9537658691406, 178.24172973632812, 100.06427764892578, 14834.332466125488]
[800, 3499.889373779297, 1599.9555587768555, 191.73622131347656, 100.01659393310547, 14128.375053405762]
[900, 4416.775703430176, 2008.199691772461, 303.7452697753906, 181.38885498046875, 14458.417892456055]
[1000, 5672.192573547363, 2595.7345962524414, 203.2756805419922, 181.29348754882812, 14204.120635986328]
```

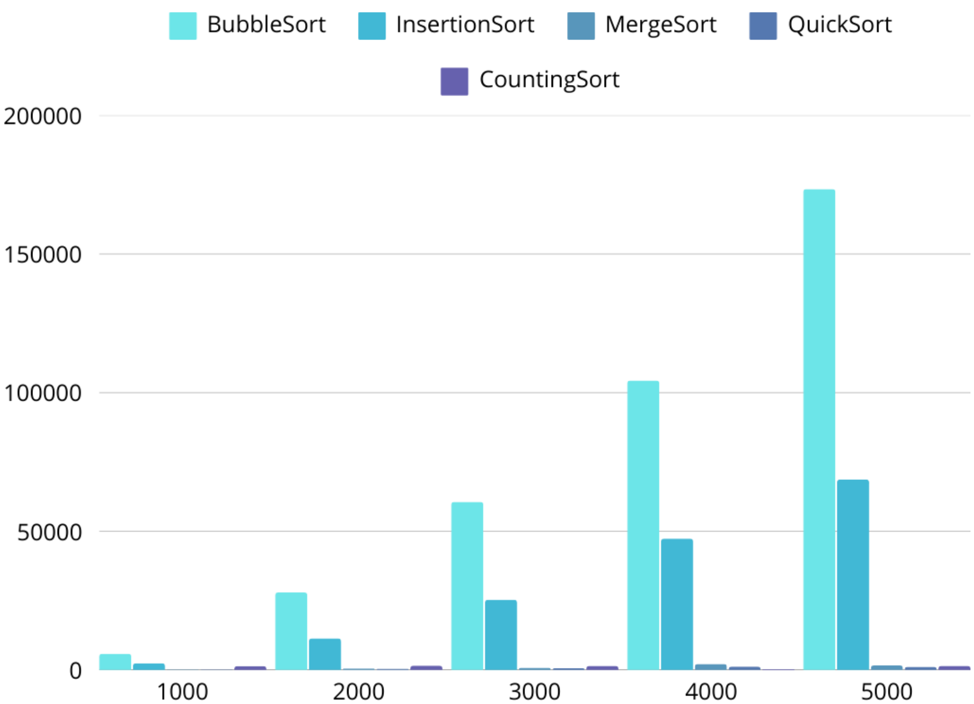


Counting Sort became significantly less efficient, even worse than Bubble Sort, this confirms that Counting Sort is only effective when the value range closely matches the list size.

4. For lists of minimum size 1000 and maximum size 5000, with 10 test cases per size and MAX_VALUE = 100000:

```
"C:\Users\Sebastian\Desktop\Universidad\Octavo Semestre\ALDA_M\Tarea1\SortAlgorithms\.venv\Scripts\python.exe"
Size | BubbleSort | InsertionSort | MergeSort | QuickSort | CountingSort
[1000, 5807.590484619141, 2420.353889465332, 200.0570297241211, 199.74708557128906, 1377.3441314697266]
[2000, 27952.50415802002, 11360.621452331543, 499.9399185180664, 391.3402557373047, 1554.3937683105469]
[3000, 60639.166831970215, 25332.021713256836, 775.456428527832, 599.8849868774414, 1413.3930206298828]
[4000, 104345.05939483643, 47399.97386932373, 2100.3007888793945, 1212.5015258789062, 2999.8779296875]
[5000, 173460.91270446777, 68665.45677185059, 1600.0986099243164, 1100.2302169799805, 1493.1201934814453]

Process finished with exit code 0
```



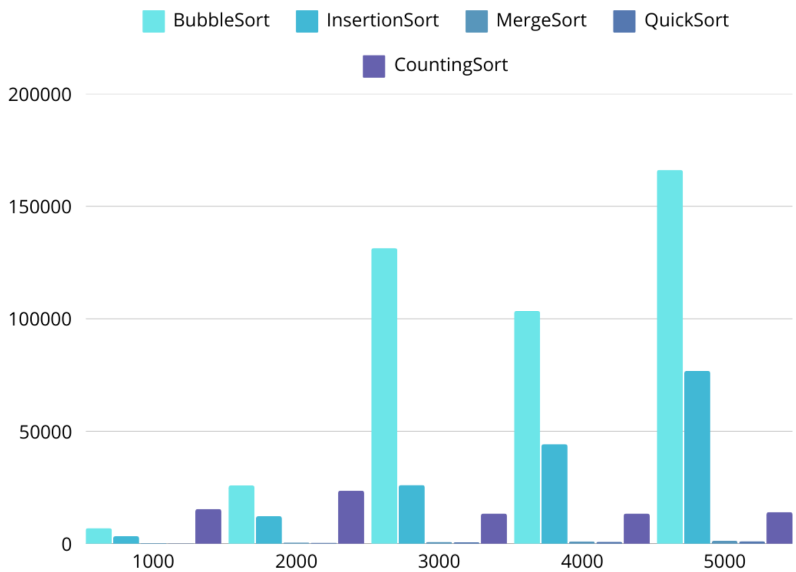
As list size increased, O(n^2) algorithms became increasingly inefficient.

Other algorithms maintained their effectiveness regardless of list size.

Quick Sort was slightly better than Merge Sort.

5. For lists of minimum size 1000 and maximum size 5000, with 10 test cases per size and MAX_VALUE = 1000000:

```
"C:\Users\Sebastian\Desktop\Universidad\Octavo Semestre\ALDA_M\Tarea1\SortAlgorithms\.venv\Scripts\python.exe" "C
Size | BubbleSort | InsertionSort | MergeSort | QuickSort | CountingSort
[1000, 6958.150863647461, 3325.986862182617, 299.9305725097656, 200.4861831665039, 15352.368354797363]
[2000, 25928.139686584473, 12195.920944213867, 499.74918365478516, 399.8279571533203, 23607.182502746582]
[3000, 131514.1201019287, 26097.20230102539, 792.0026779174805, 601.0293960571289, 13455.367088317871]
[4000, 103511.6195678711, 44242.238998413086, 1010.7517242431641, 893.5213088989258, 13411.164283752441]
[5000, 166140.5324935913, 76881.07490539551, 1299.9773025512695, 1099.9679565429688, 14016.366004943848]
```



As before, changing the maximum number to one million affected only Counting Sort, as the list sizes were small compared to the possible maximum range.

Conclusions

The analysis of sorting algorithms allowed for a comparison of their efficiency in different scenarios and demonstrated the importance of selecting the appropriate algorithm based on data size and distribution.

$O(n^2)$ algorithms like Bubble Sort and Insertion Sort are inefficient for large data volumes, showing significantly higher execution times compared to advanced algorithms.

Quick Sort and Merge Sort proved to be efficient options for large lists, with complexities of $O(n \log n)$. However, Quick Sort heavily depends on pivot selection to avoid unfavorable $O(n^2)$ cases.

Counting Sort was highly efficient when the value range of the list was close to its size. However, increasing the maximum allowed value drastically reduced its performance, making it even less efficient than Bubble Sort.

As list size increased, inefficient algorithms scaled exponentially in execution time, while $O(n \log n)$ algorithms remained stable.