Assignment - 1

Course: CSE373

Section:1

A Comprehensive Analysis of Merge Sort, Insertion Sort, Heap Sort, and Quick Sort in Real World Scenarios

Submitted by

Faisal Mahmud 2011523642

Submitted to

Dr. Sifat Momen

Department of Electrical and Computer Engineering

North South University

Dhaka, Bangladesh

METHODOLOGY

The reliability of algorithms has been checked using test_algorithms() function. After that, the process has been broken down into several key components.

Sorting Algorithms Implementation:

This part includes the implementation of sorting algorithms that include Insertion Sort, Merge Sort, Heapsort, and Quicksort. Each sorting algorithm is defined with specific functions for sorting arrays.

Dataset Generation

This part provides functions to generate different types of datasets for testing the sorting algorithms. Datasets include random, sorted, reverse sorted, Fibonacci, nearly sorted, few unique elements, all unique elements, small and large range of values, few large gaps, alternating high/low values, large and small datasets, prime number of elements, lexicographical strings, floating-point numbers, negative and positive numbers, sparse datasets, mixed data types, Fibonacci and geometric sequences, arithmetic sequences, prime numbers, Fibonacci mod-sequence, powers of two, Fourier series, and Perlin noise sequence.

Time and Space Measurement

Functions have been created to measure time and space. Time measurement is done by using the python time library. Space measurement is done using the memory_profiler module to accurately calculate memory usage during sorting.

Experimental Setup

run_time_experiment() and run_space_experiment() functions were created to run time and space experiments for different dataset sizes and types. It iterates over dataset sizes and types, applies sorting algorithms to generated datasets, and records the time taken and space used.

Results Visualization

This part includes functions to plot the results of time and space measurements for sorting algorithms across different dataset sizes and types. Plots show the performance of sorting algorithms in terms of time and space complexity.

Discussion and Analysis

The main() function executes the experiments, data generation, sorting algorithm application, and result plotting. The methodology involves running experiments, collecting data on algorithm performance, and visualizing the results to analyze the efficiency of sorting algorithms under various conditions.

DATASET SIZE CONSIDERATION

A dataset containing up to 20,000 entries has been tested. An observation made from that is that when using insertion sort on large datasets, the graph shows an exponential increase, causing the graphs of mergesort, heapsort, and quicksort to appear as a single line. To address this issue and ensure clarity as graph of both 20,000 entries dataset and 2000 entries dataset shows simmilar growth in the experiment, a maximum dataset size of 2000 has been chosen for convenience in this thesis study.

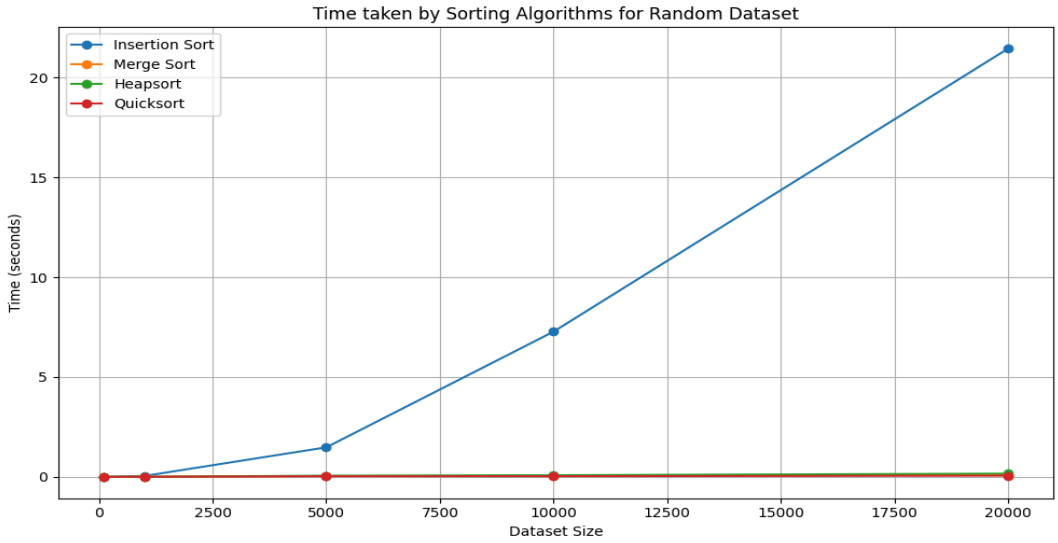


Fig 1: Dataset size of 20,000

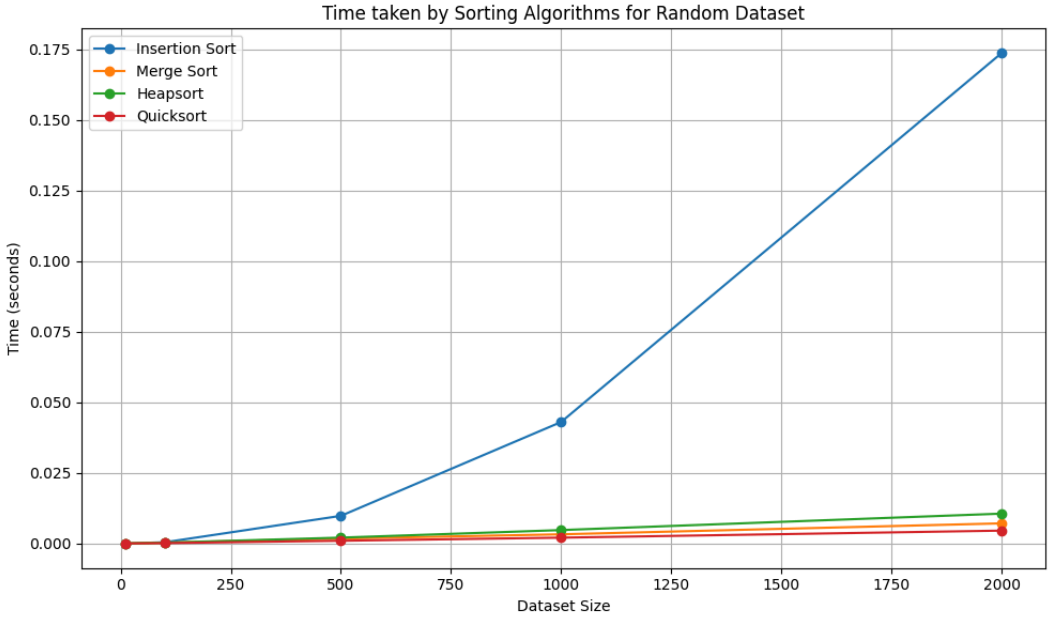


Fig 2: Dataset size of 2000

EXPERIMENTAL ANALYSIS

Random Value Dataset

This Dataset is generated with random values without any specific order.

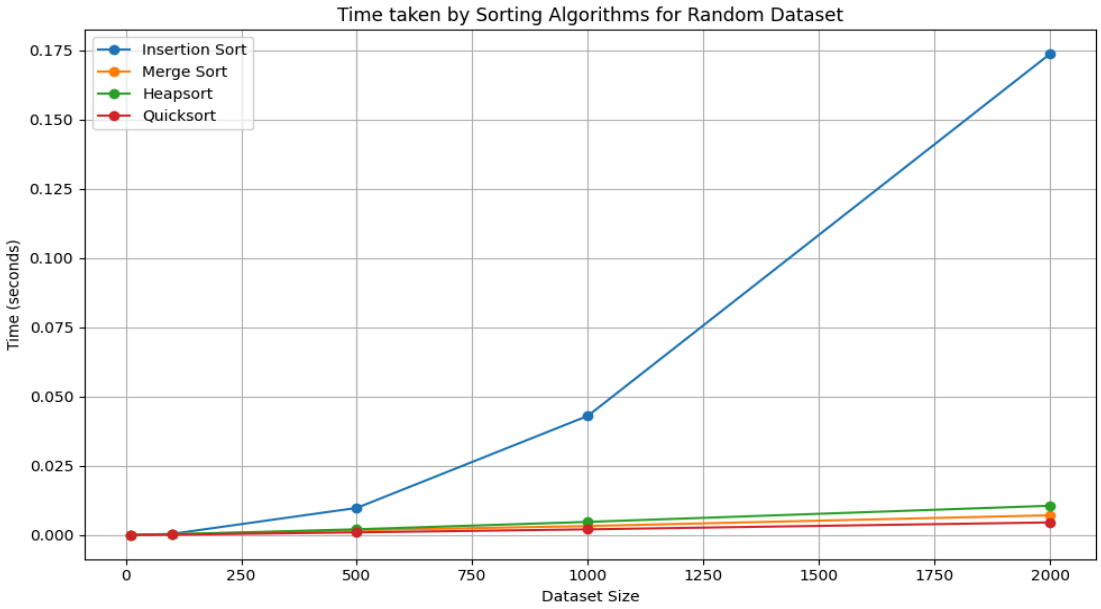


Fig 3: time taken by sorting algorithms for random value dataset

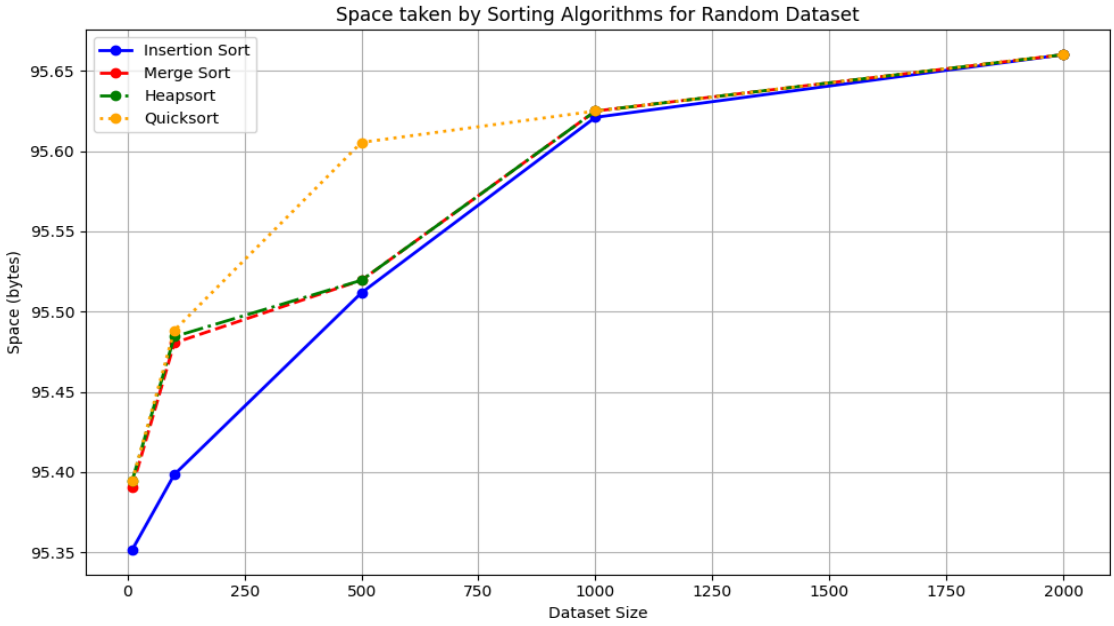


Fig 4: space taken by sorting algorithms for random value dataset

Sorted Dataset

In this dataset values are sorted in ascending order

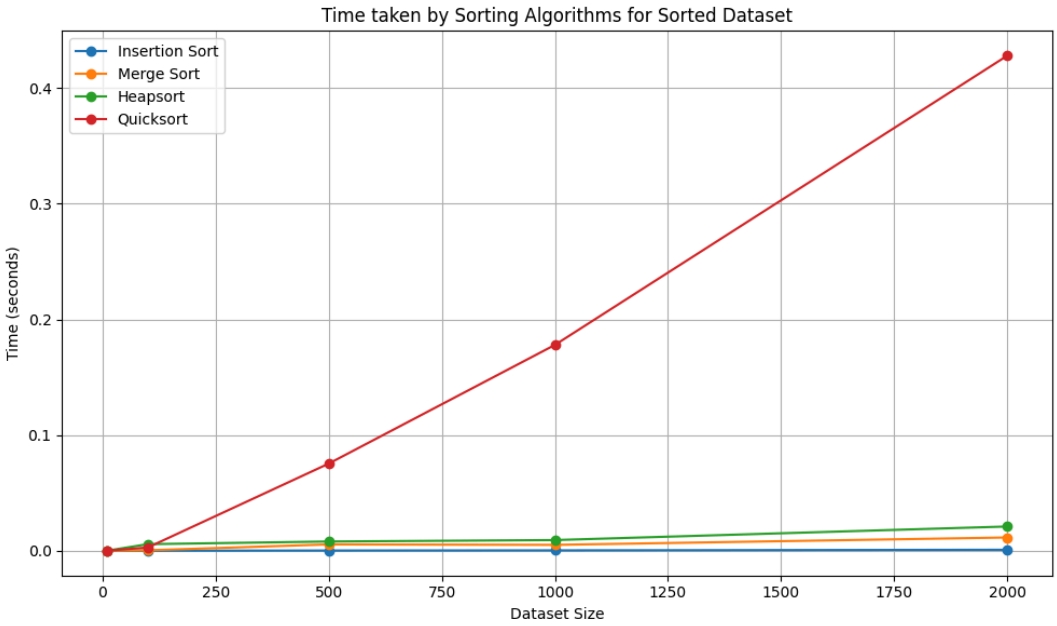


Fig 5: time taken by sorting algorithms for sorted dataset

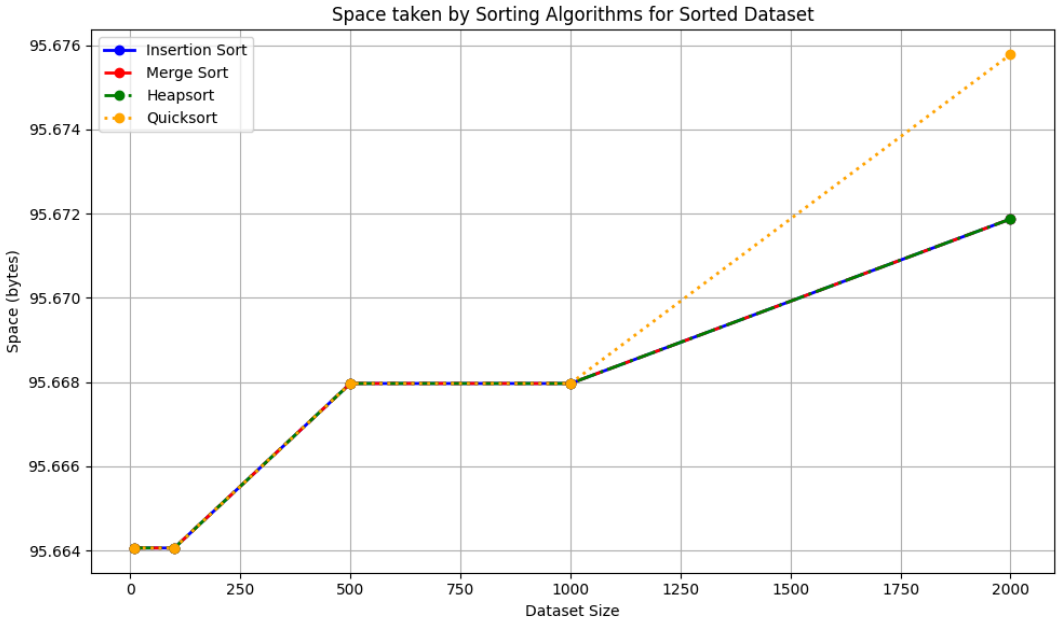


Fig 6: space taken by sorting algorithms for sorted dataset

Reverse Sorted Dataset

In this dataset values are arranged in descending order.

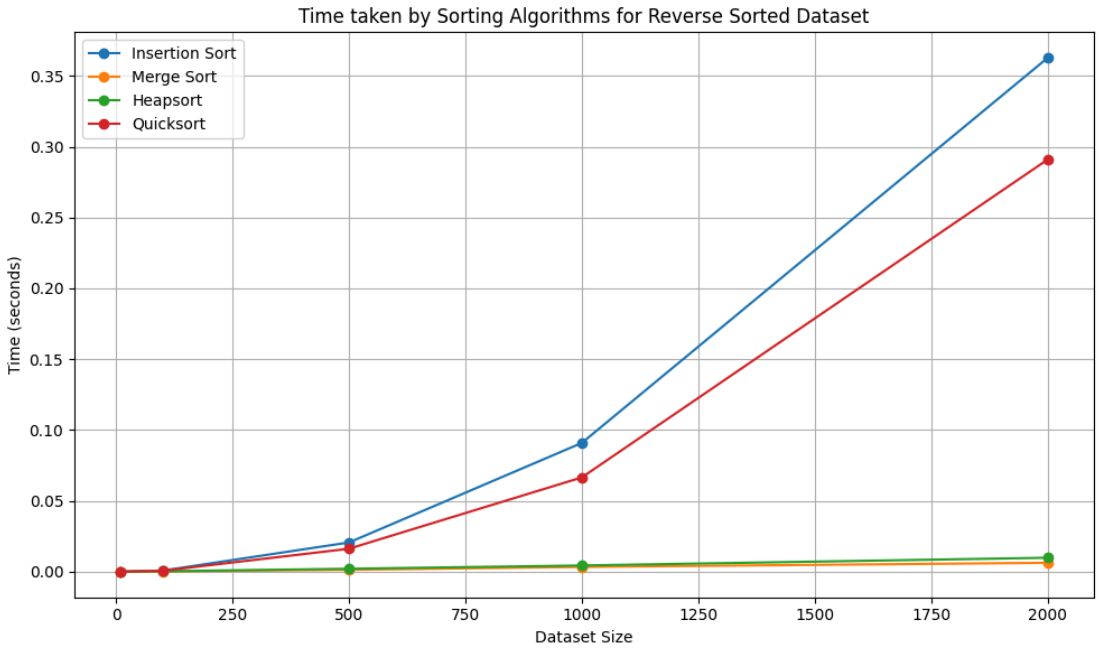


Fig 7: time taken by sorting algorithms for reverse sorted dataset

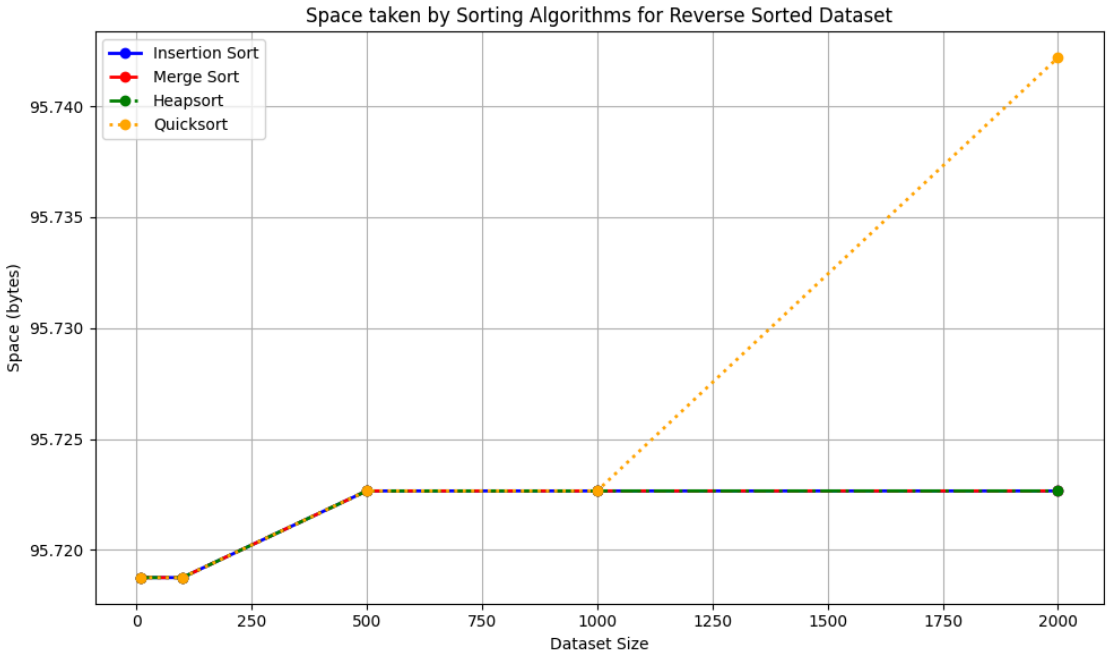


Fig 8: space taken by sorting algorithms for reverse sorted dataset

Fibonacci Dataset

This dataset consists of values that are the sum of the two preceding ones.

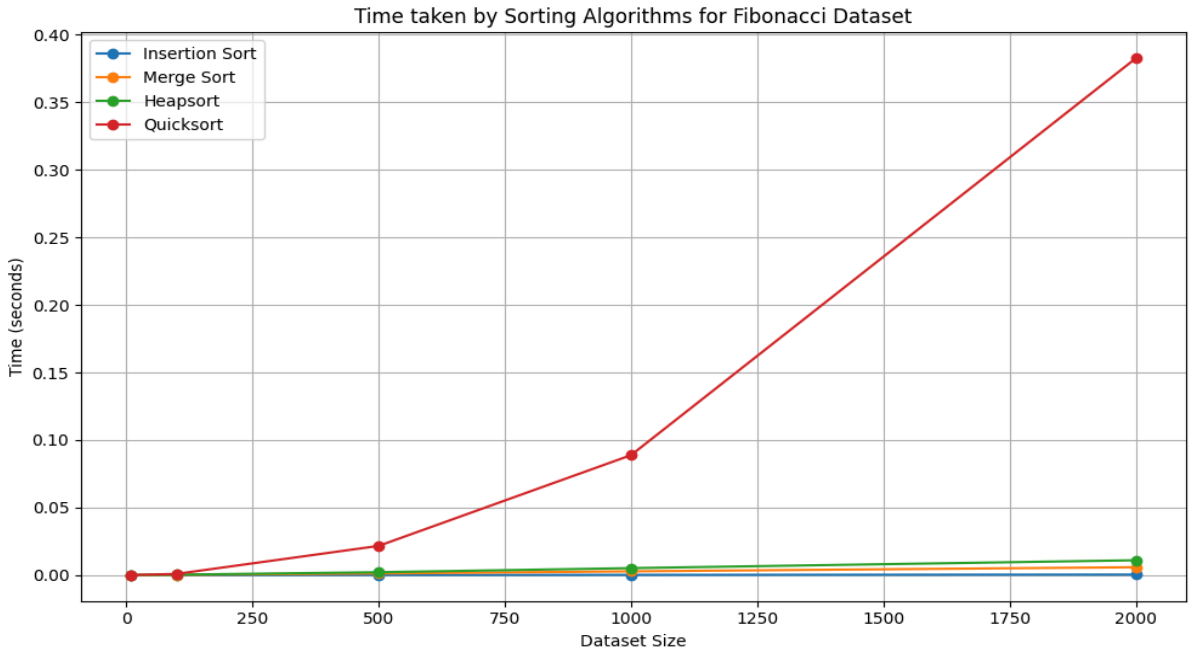


Fig 9: space taken by sorting algorithms for Fibonacci dataset

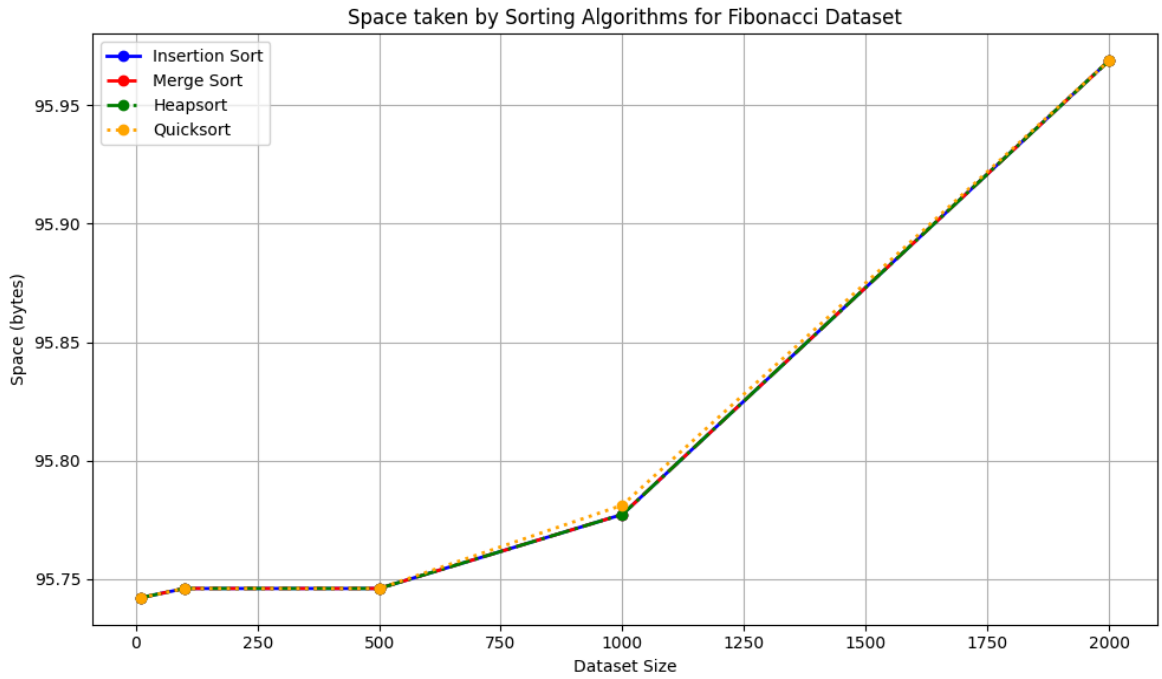


Fig 10: space taken by sorting algorithms for the Fibonacci dataset

Nearly Sorted Dataset

This dataset is mostly sorted with a few values intentionally misplaced.

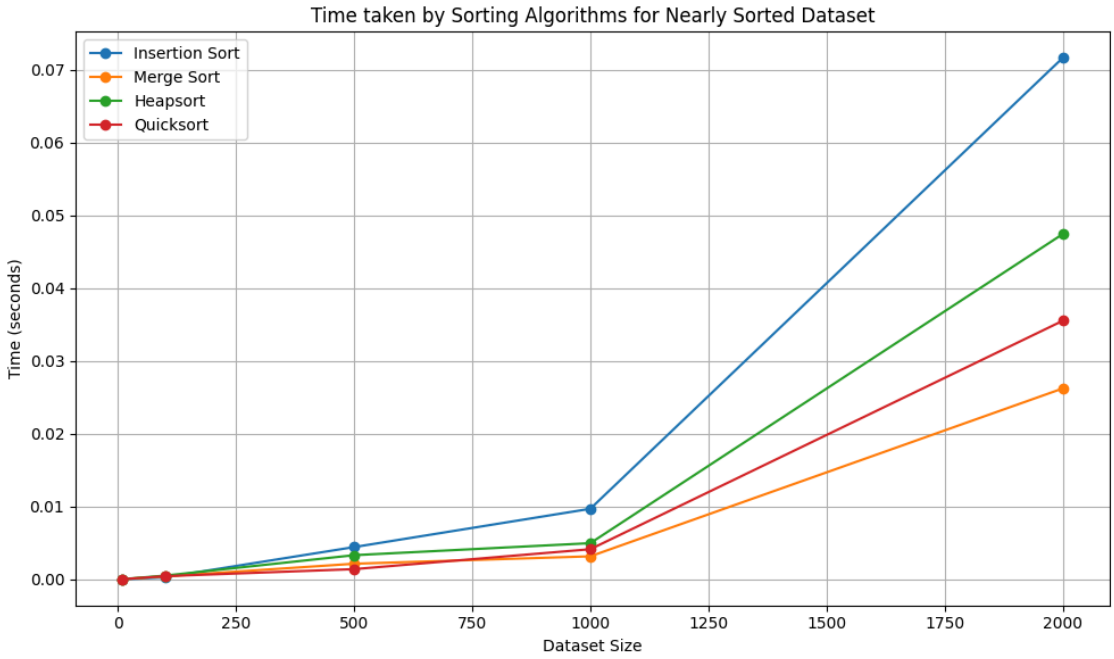


Fig 11: time taken by sorting algorithms for nearly sorted dataset

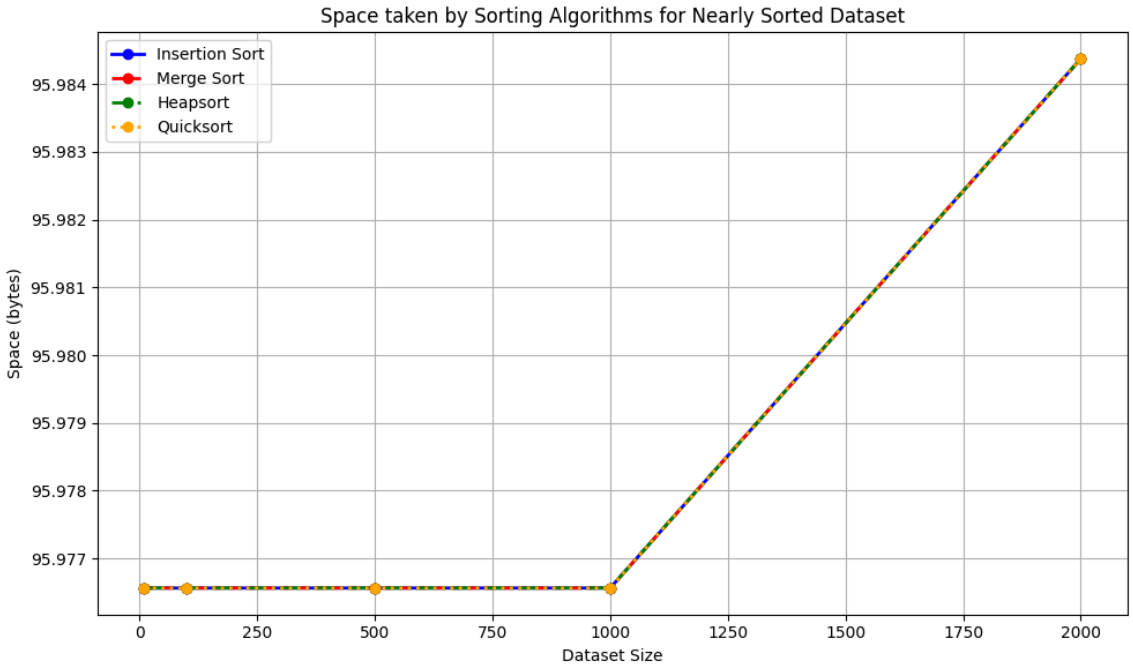


Fig 12: space taken by sorting algorithms for nearly sorted dataset

Few Unique Dataset

This dataset contains moslty duplicate values, with a few number of unique elements

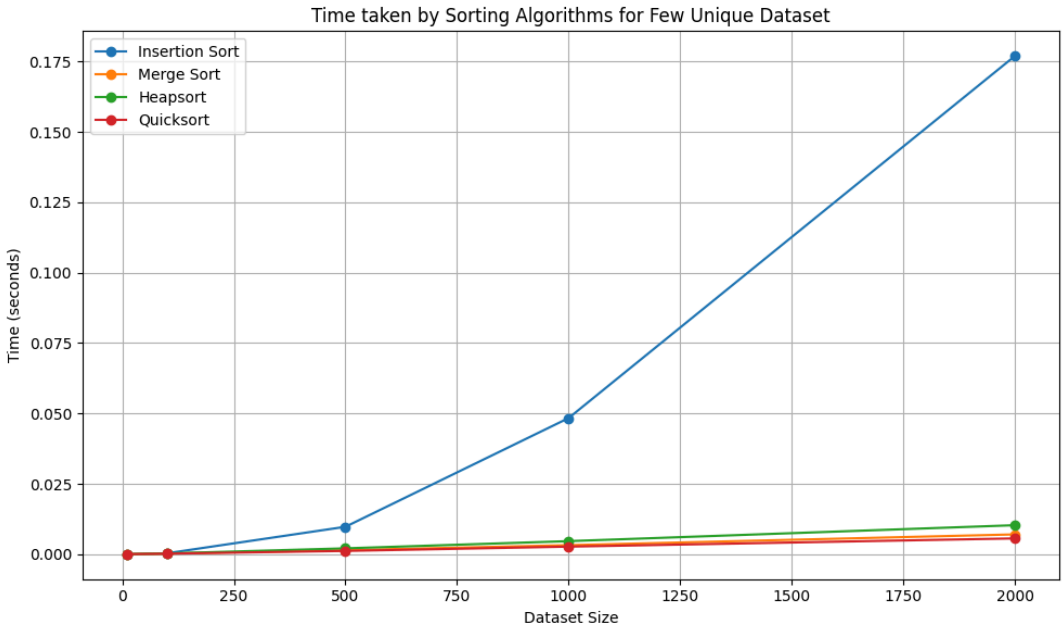


Fig 13: time taken by sorting algorithms for few unique dataset

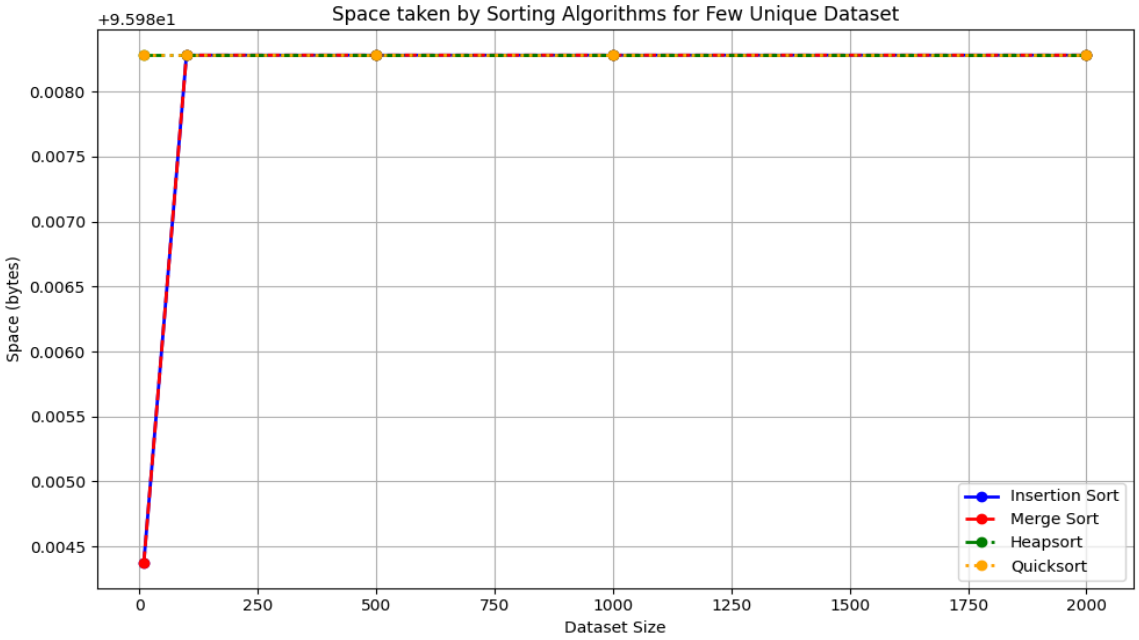


Fig 14: time taken by sorting algorithms for few unique dataset

All Unique Dataset

Each and every elemnet is unique in this dataset

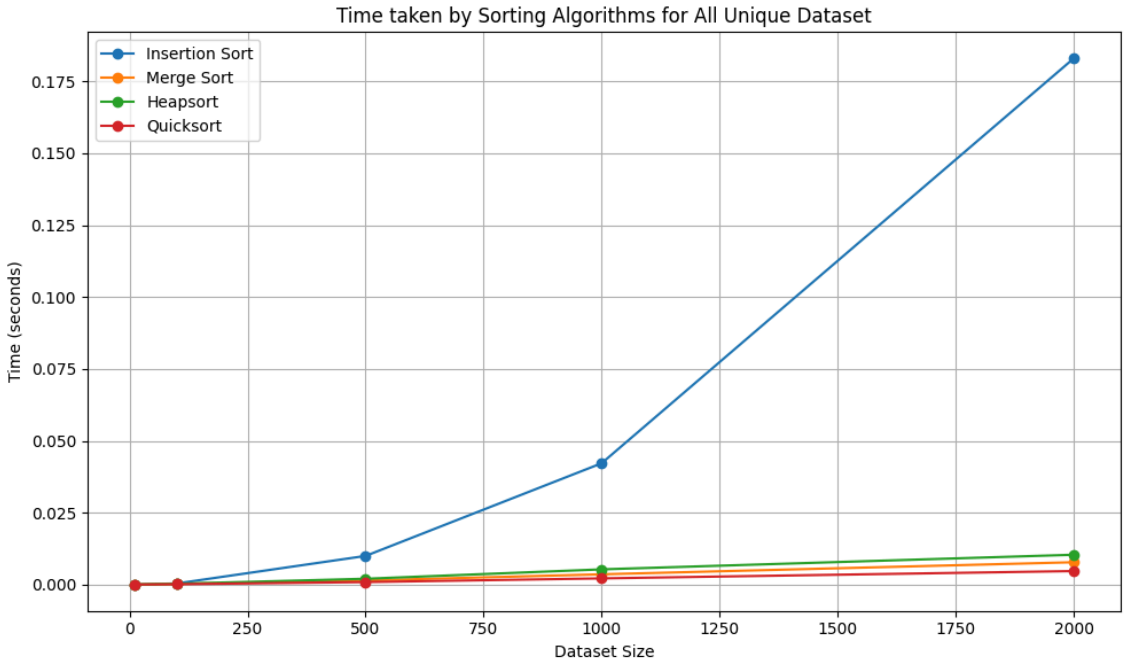


Fig 15: time taken by sorting algorithms for all unique dataset

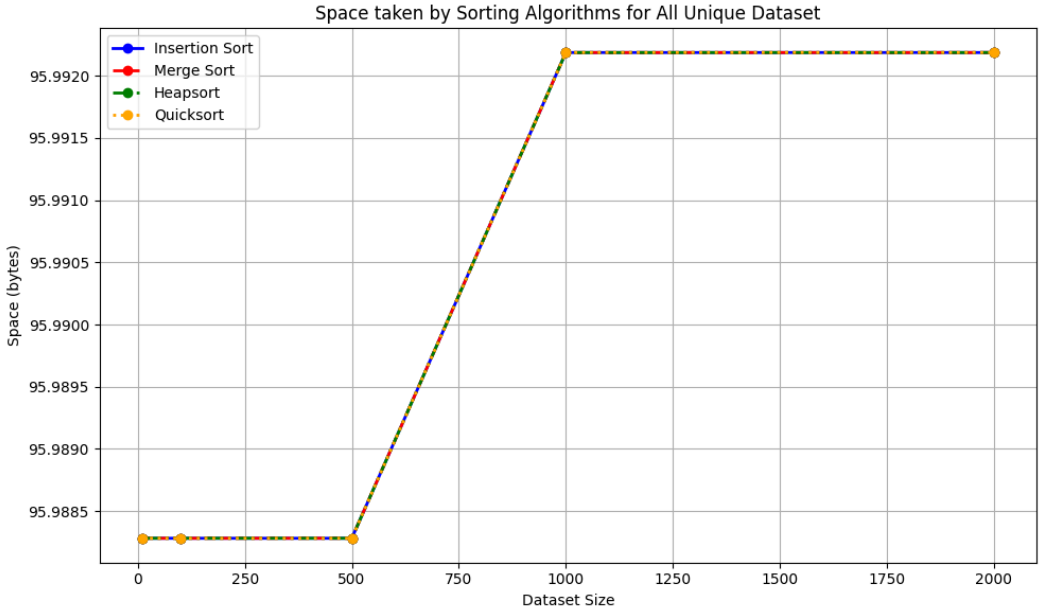


Fig 16: space taken by sorting algorithms for all unique dataset

Small Range Dataset

This dataset consists of very close values and has very small numerical range

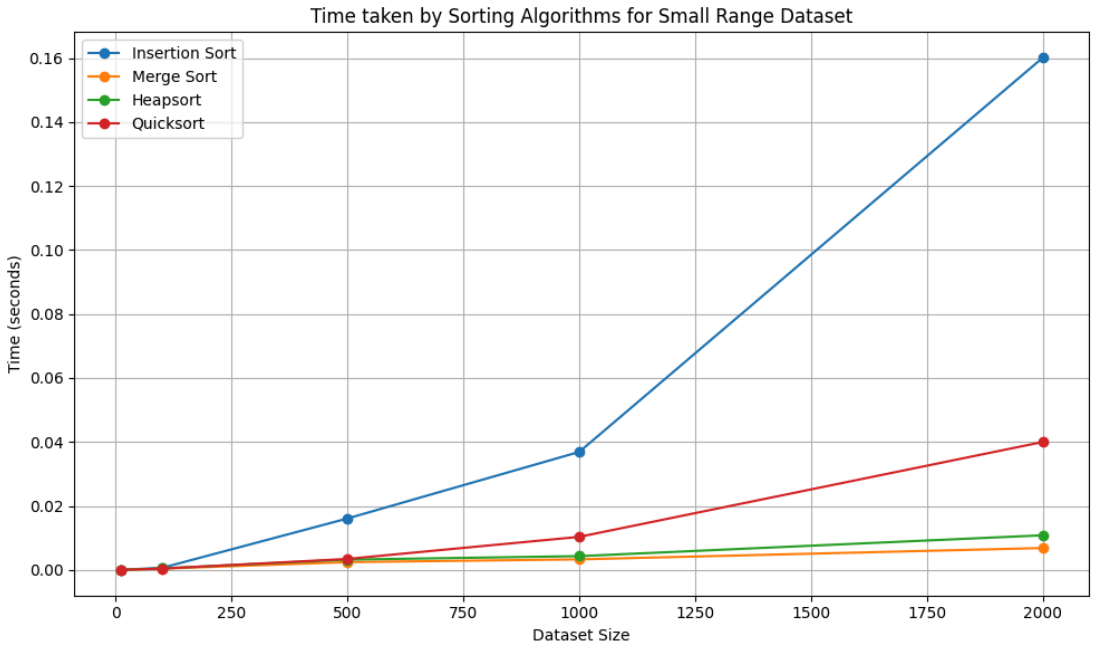


Fig 17: time taken by sorting algorithms for small range dataset

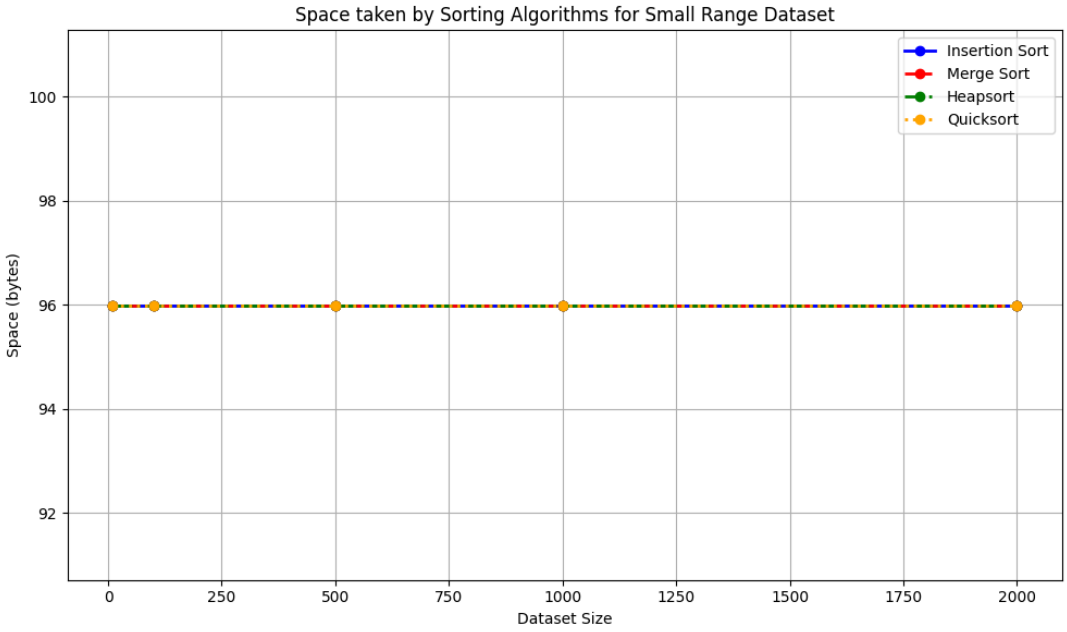


Fig 18: space taken by sorting algorithms for small range dataset

Large Range Dataset

This dataset consists of values that are very wide in numerical range

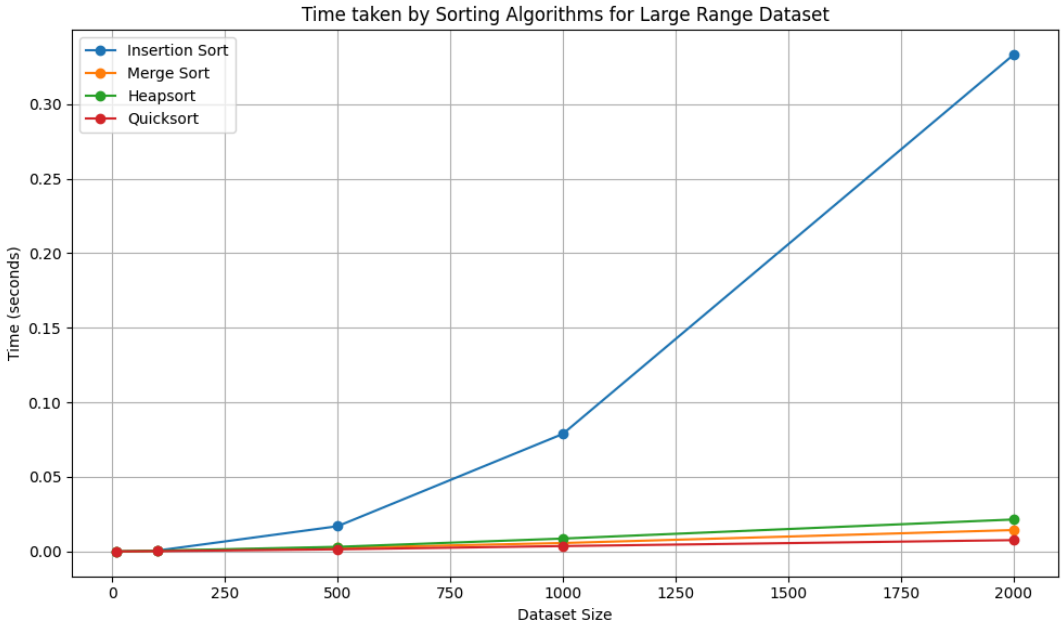


Fig 19: time taken by sorting algorithms for large range dataset

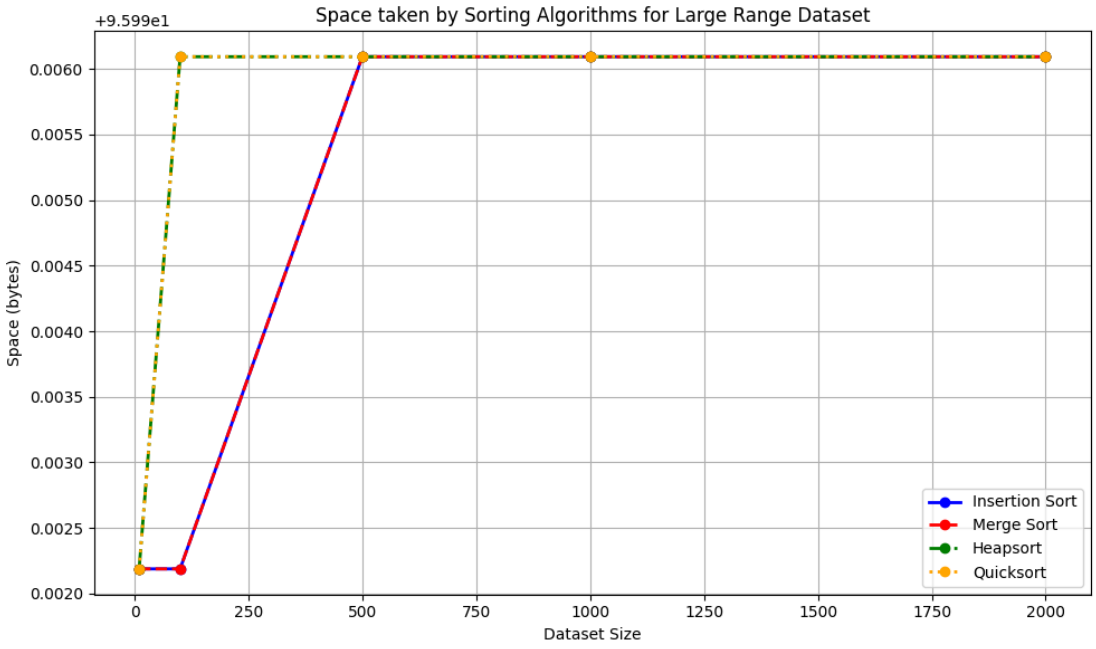


Fig 20: space taken by sorting algorithms for large range dataset

Few large Gaps Dataset

This dataset consists of data where some of the gap between next value and current value is very large

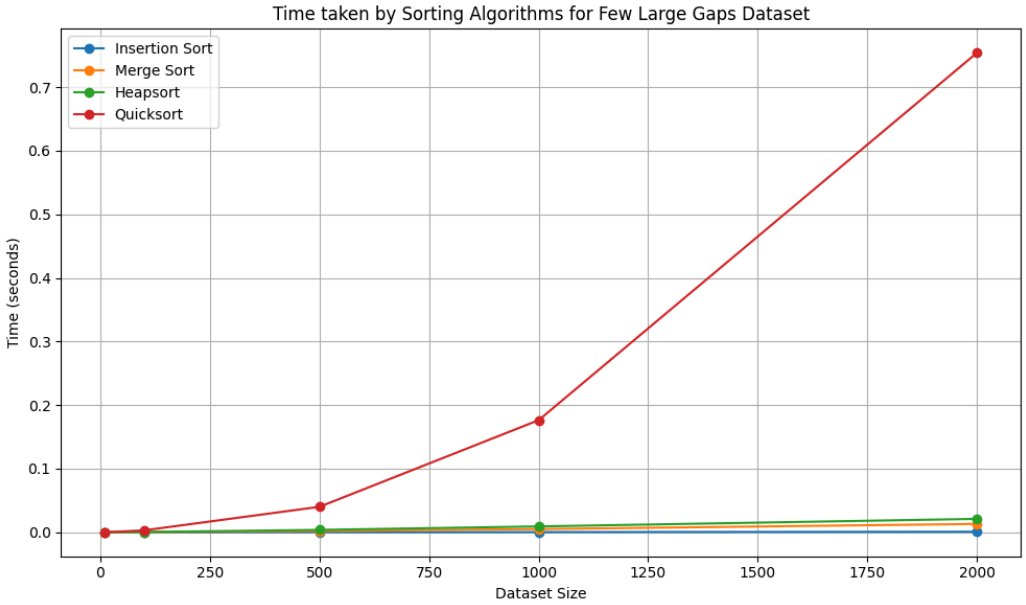


Fig 21: time taken by sorting algorithms for few large gap dataset

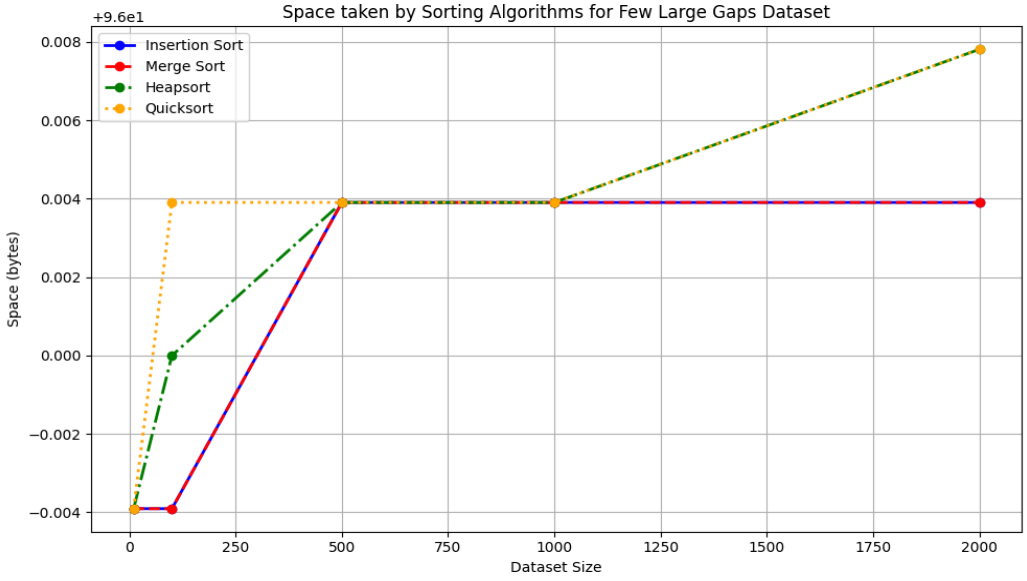


Fig 22: space taken by sorting algorithms for a few large gap dataset

Alternating High/Low Dataset

This dataset consists of values that alternate between high and low. For example, if the current value is small then the next value will be very large. But the value after that will be small again.

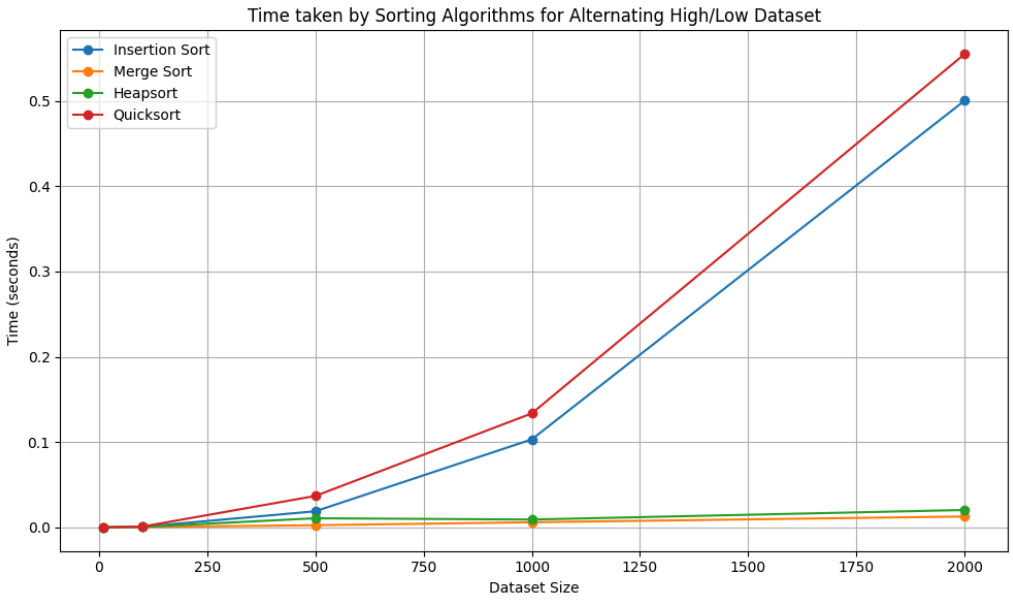


Fig 23: time taken by sorting algorithms for alternating high/low dataset

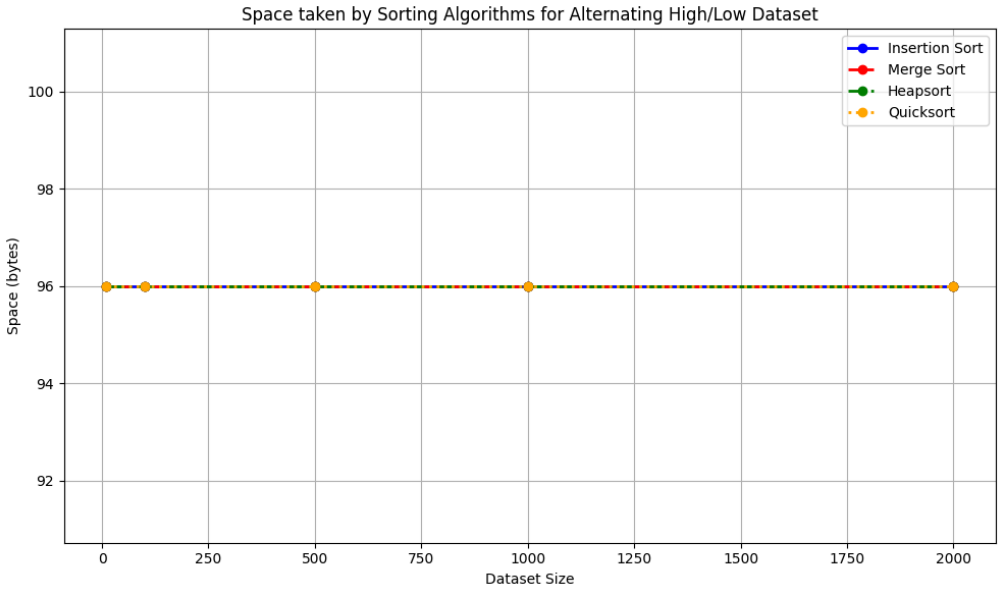


Fig 24: space taken by sorting algorithms for alternating high/low dataset

Prime Number Dataset

This dataset consists of prime number elements

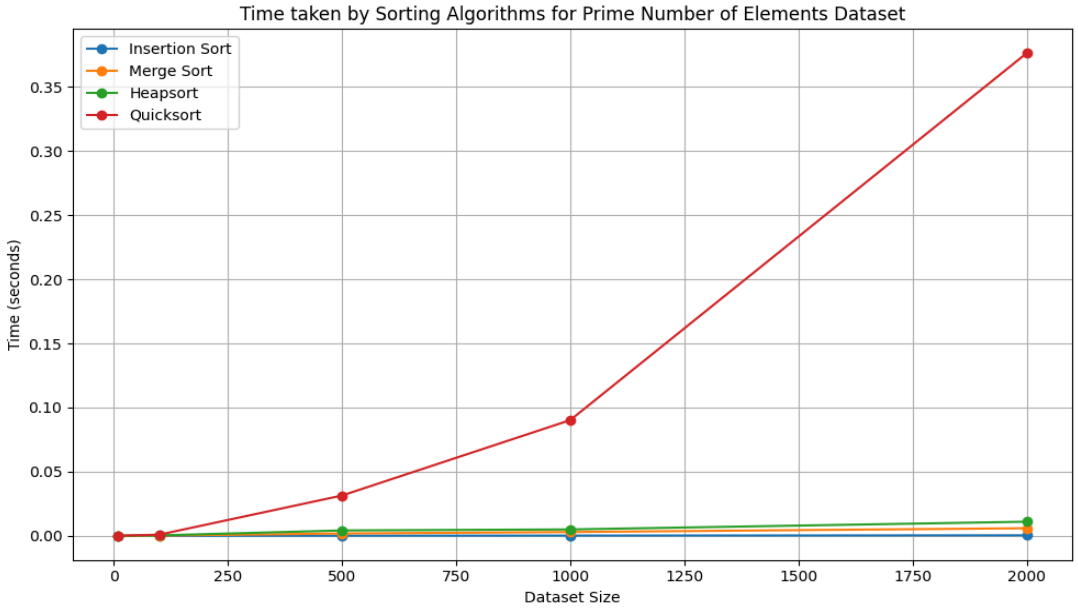


Fig 25: time taken by sorting algorithms for prime number dataset

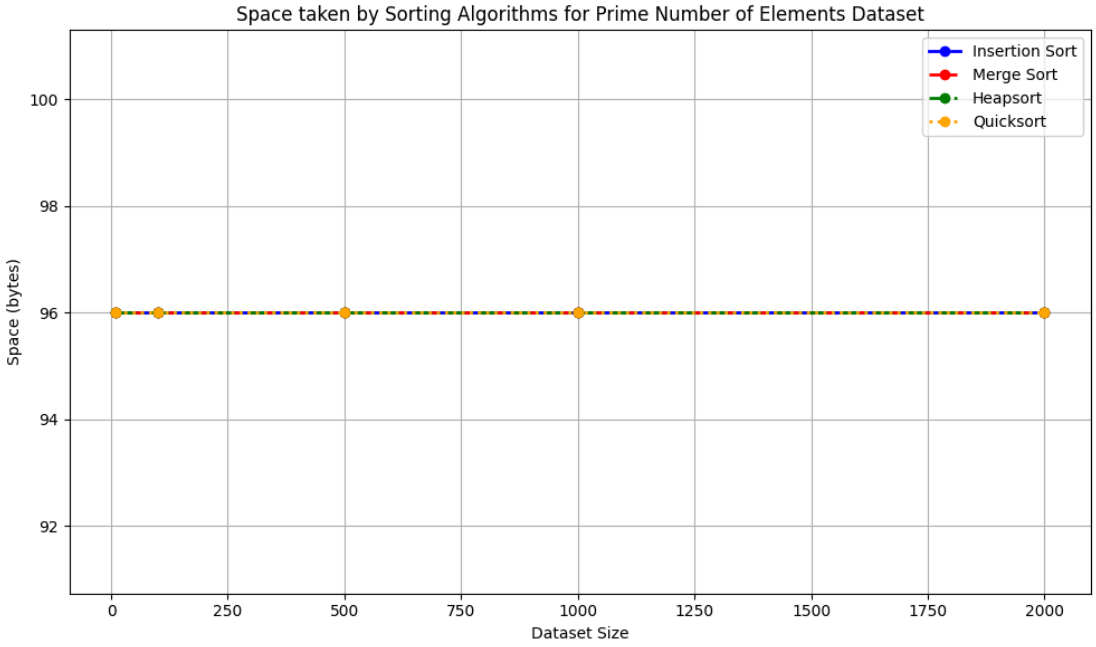


Fig 26: space taken by sorting algorithms for prime number dataset

Strings Lexicographical Dataset

This dataset is create using ASCII value of strings stored in lexicographical order

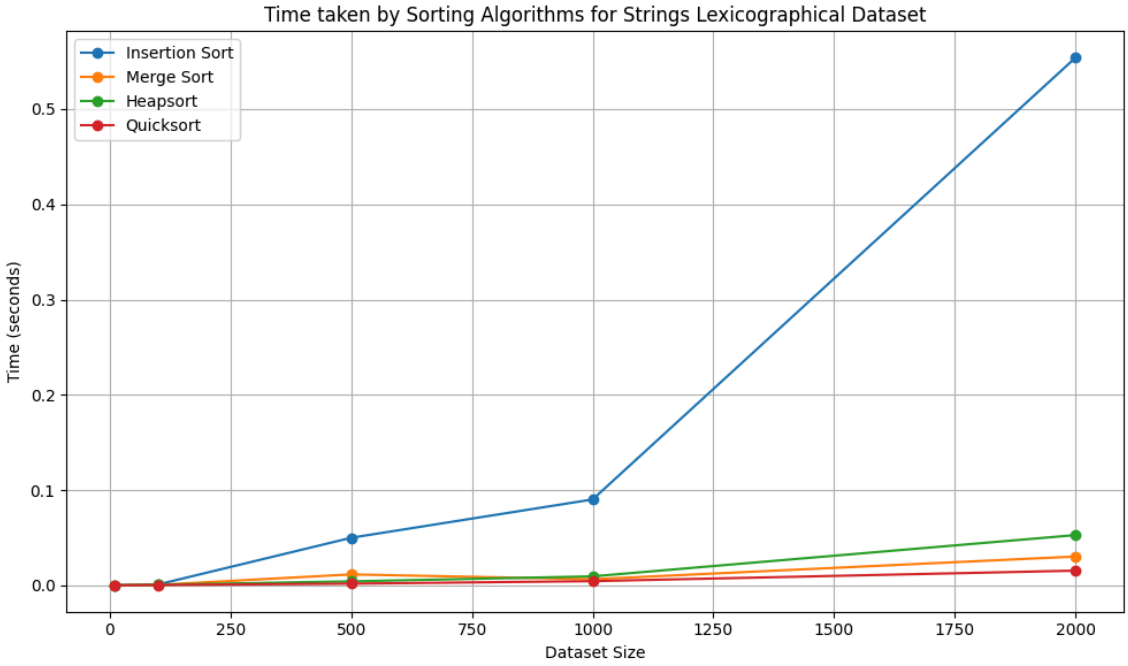


Fig 27: time taken by sorting algorithms for strings lexicographical dataset

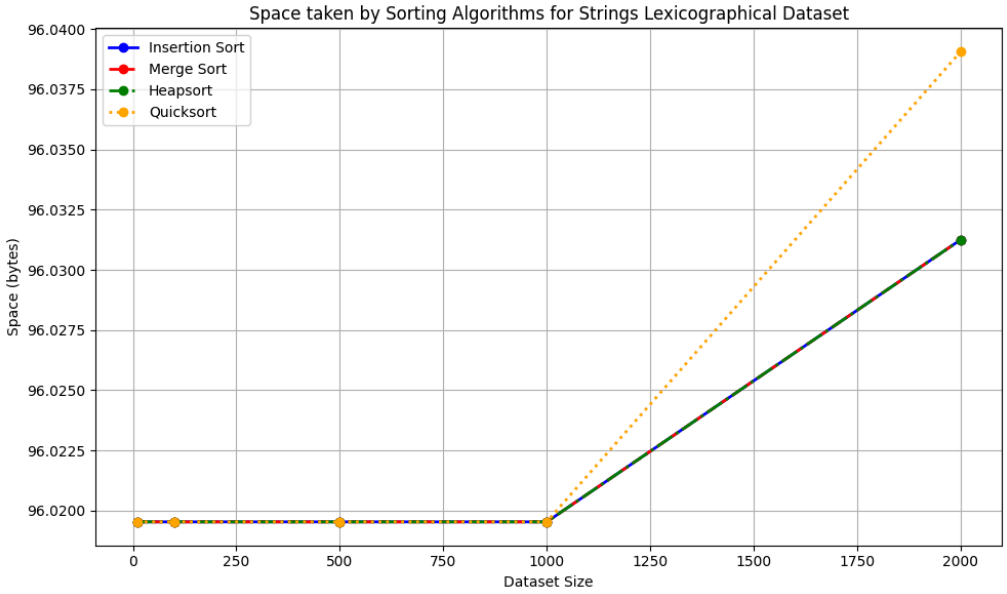


Fig 28: space taken by sorting algorithms for strings lexicographical dataset

String Reverse Lexicographical Dataset

This dataset is create using ASCII value of strings stored in reverse lexicographical order

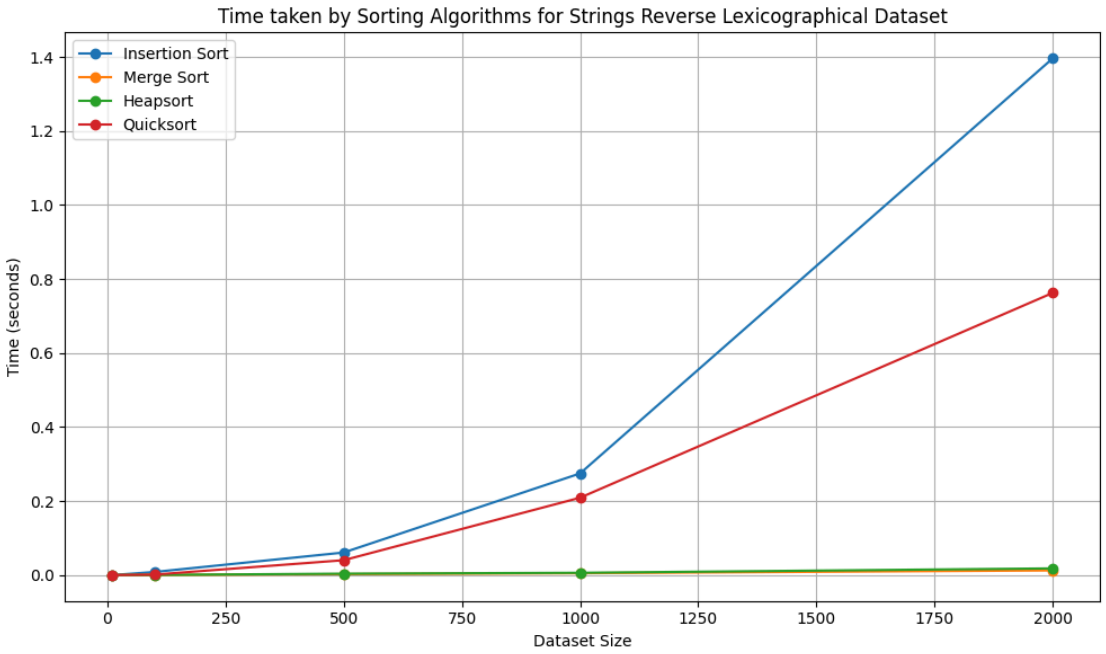


Fig 29: time taken by sorting algorithms for strings reverse lexicographical dataset

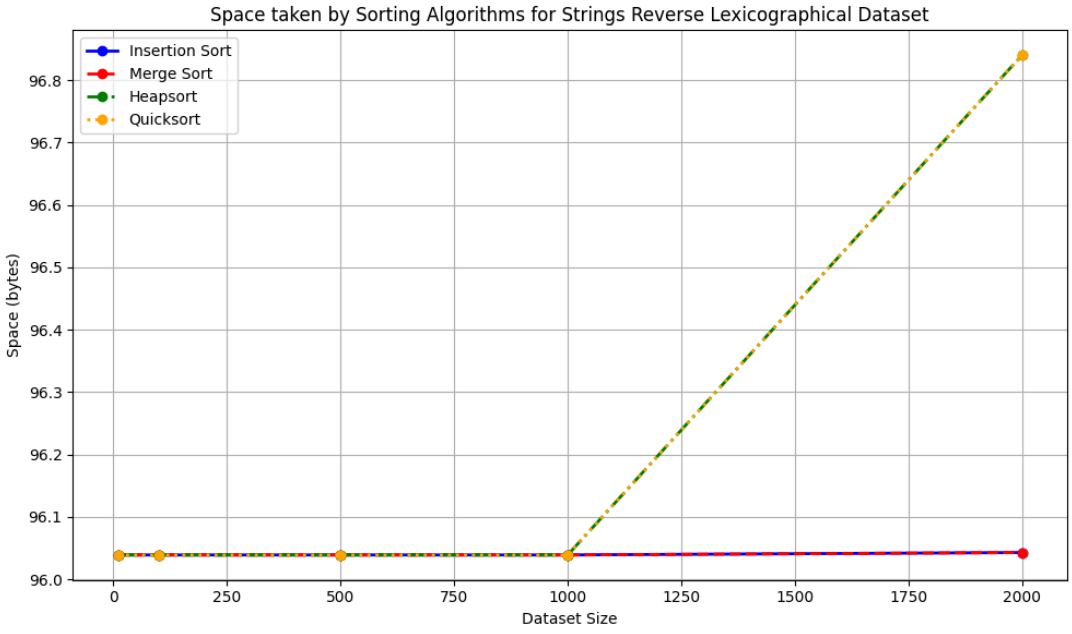


Fig 30: space taken by sorting algorithms for strings reverse lexicographical dataset

Floating Point Dataset

All the value in this dataset are floating point numbers

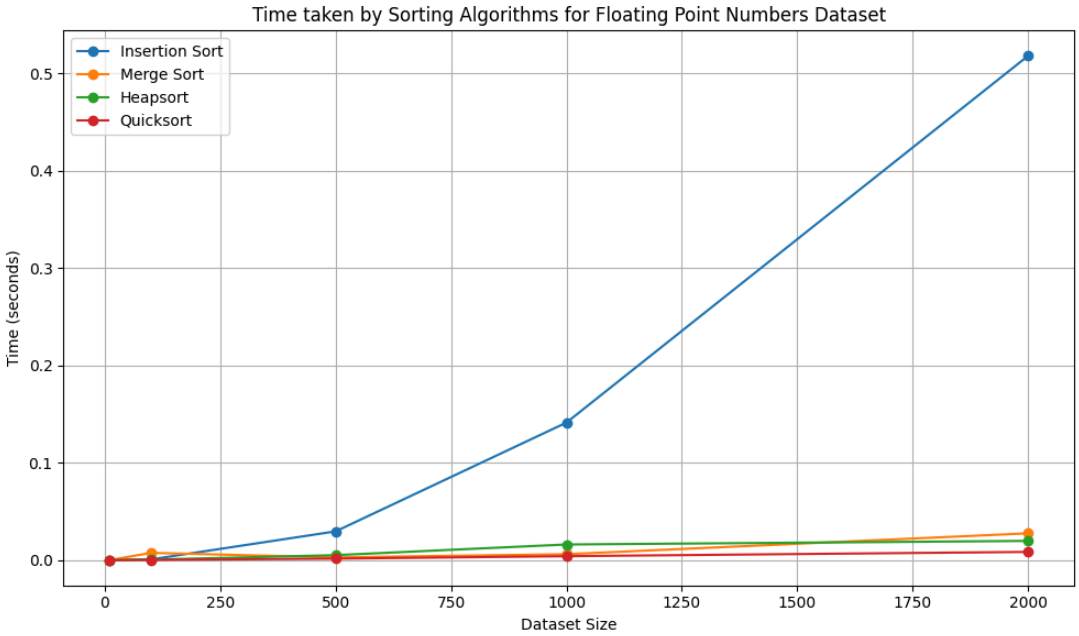


Fig 31: time taken by sorting algorithms for floating point dataset

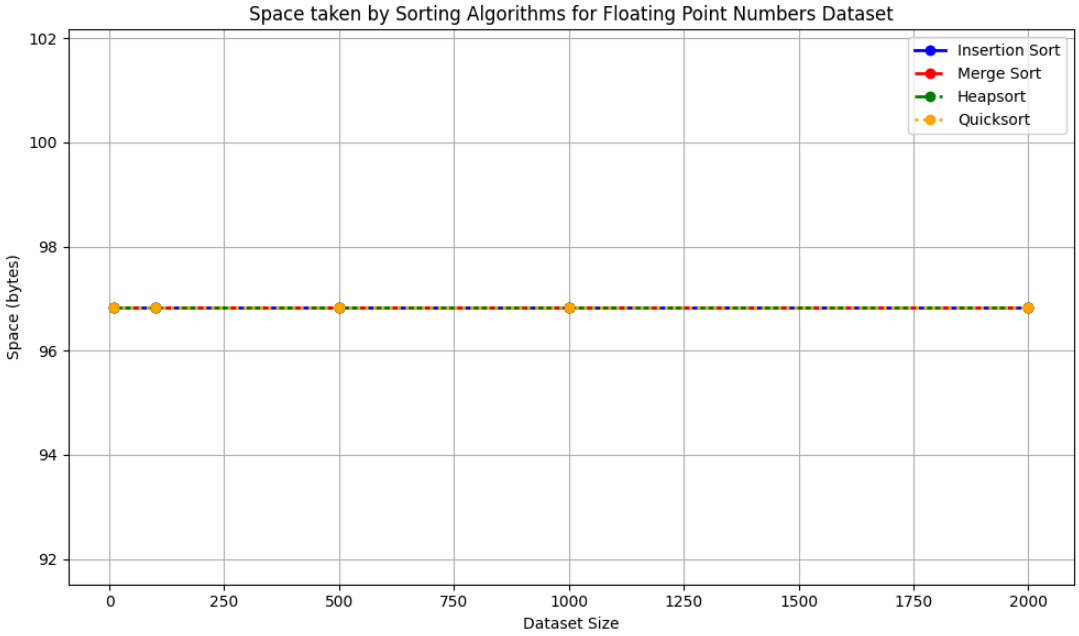


Fig 32: space taken by sorting algorithms for floating point dataset

Negative and Positive Number Dataset

This dataset consists both positive and Negative Numbers

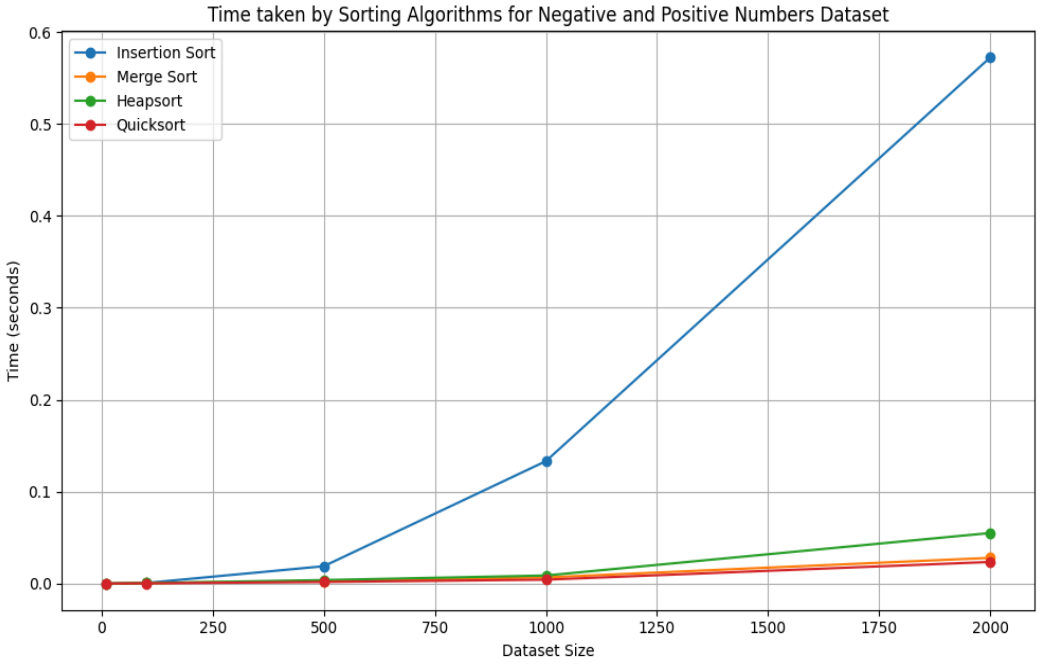


Fig 33: time taken by sorting algorithms for negative and positive number dataset

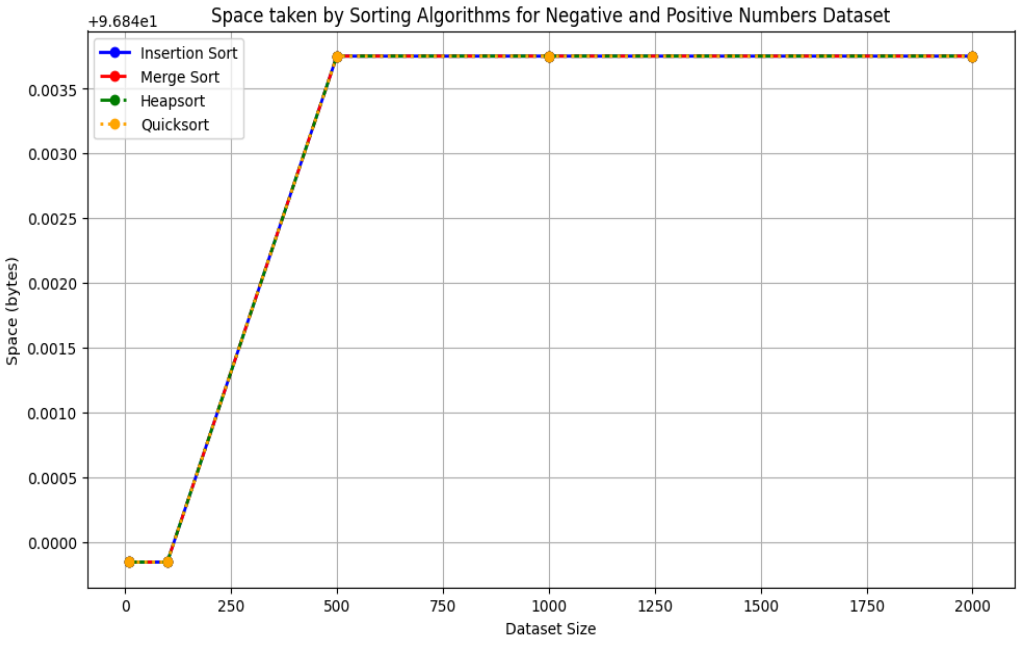


Fig 34: space taken by sorting algorithms for negative and positive number dataset

Sparse Dataset

This dataset consists of sparse data with zero and one values.

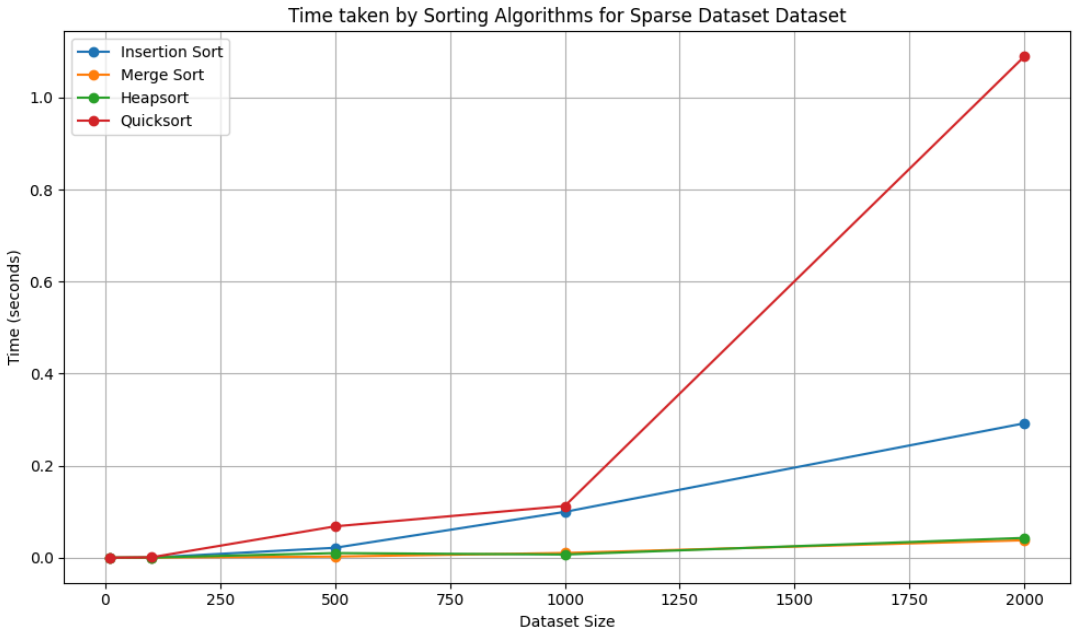


Fig 35: time taken by sorting algorithms for sparse dataset

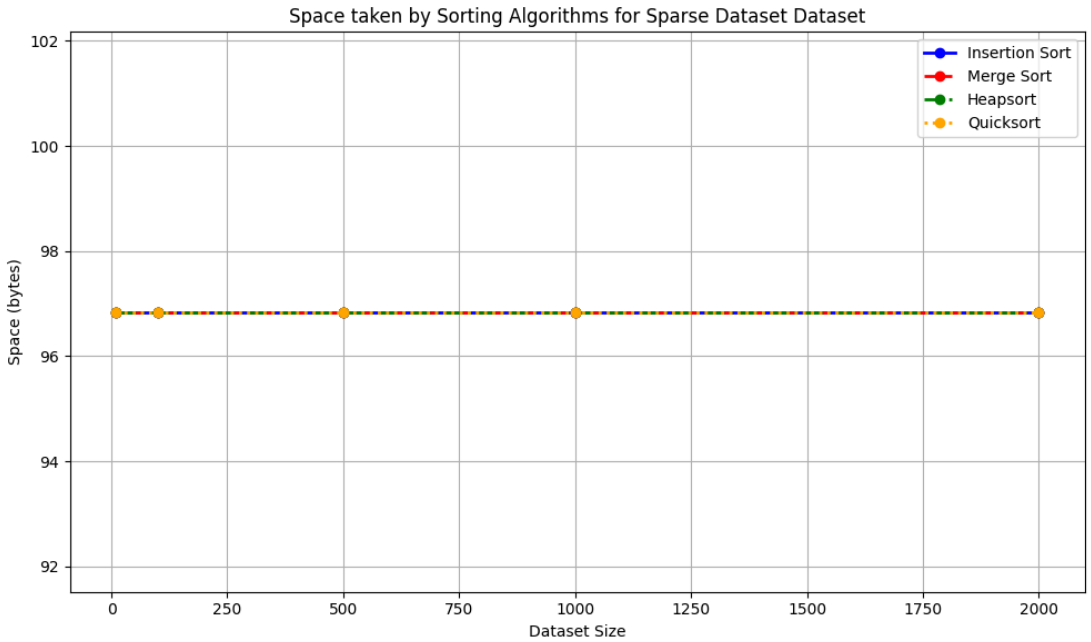


Fig 36: space taken by sorting algorithms for sparse dataset

Mixed Data Types Dataset

This datatype consists of mixed values like integer, float

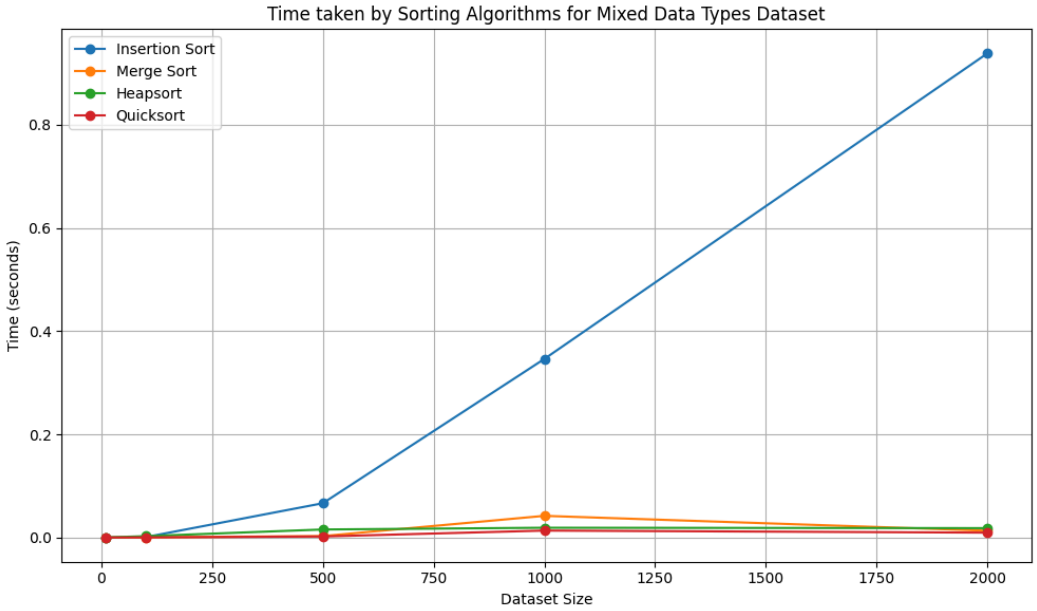


Fig 37: time taken by sorting algorithms for mixed data types dataset

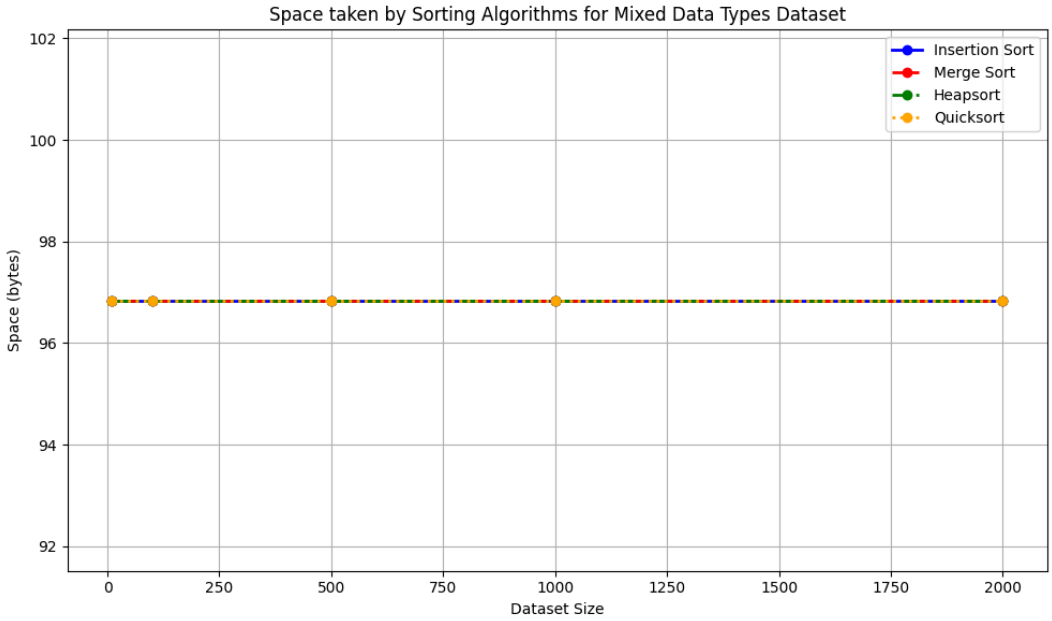


Fig 38: space is taken by sorting algorithms for mixed data types dataset

Randomized Fibonacci Sequence Dataset

Dataset consisits values from the Fibonacci sequence in random order.

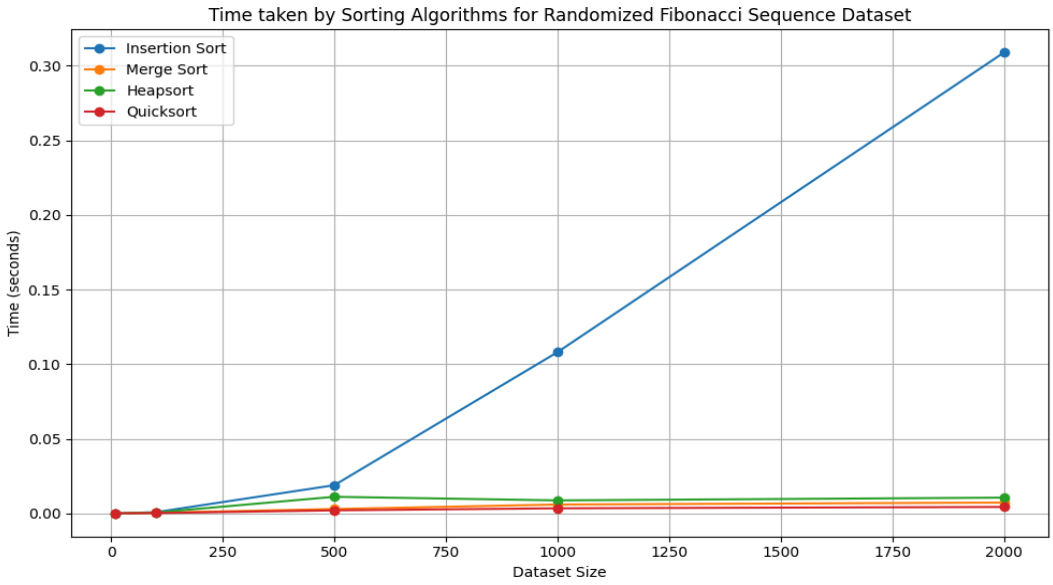


Fig 39: time taken by sorting algorithms for randomized fibocanni sequence dataset

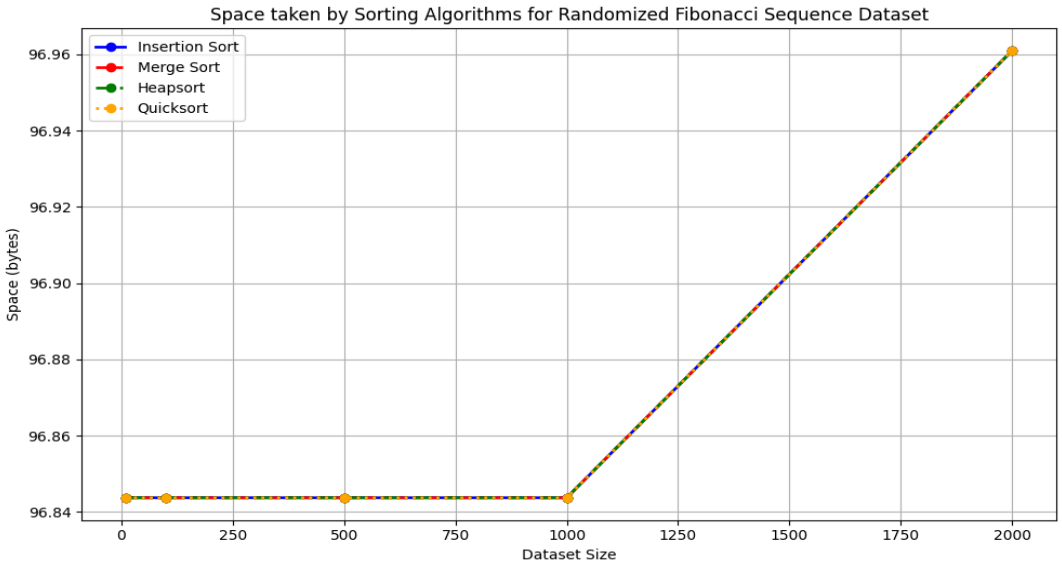


Fig 40: space taken by sorting algorithms for randomized fibocanni sequence dataset

Randomized Geometric Sequence Dataset

Dataset consists of values from a geometric sequence in random order.

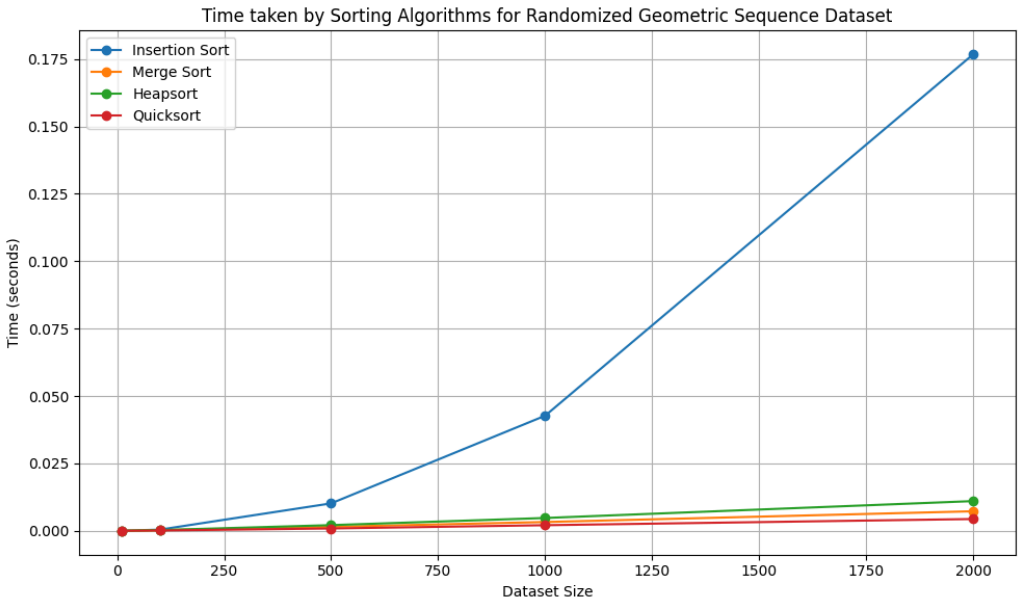


Fig 41: time taken by sorting algorithms for randomized geometric sequence dataset

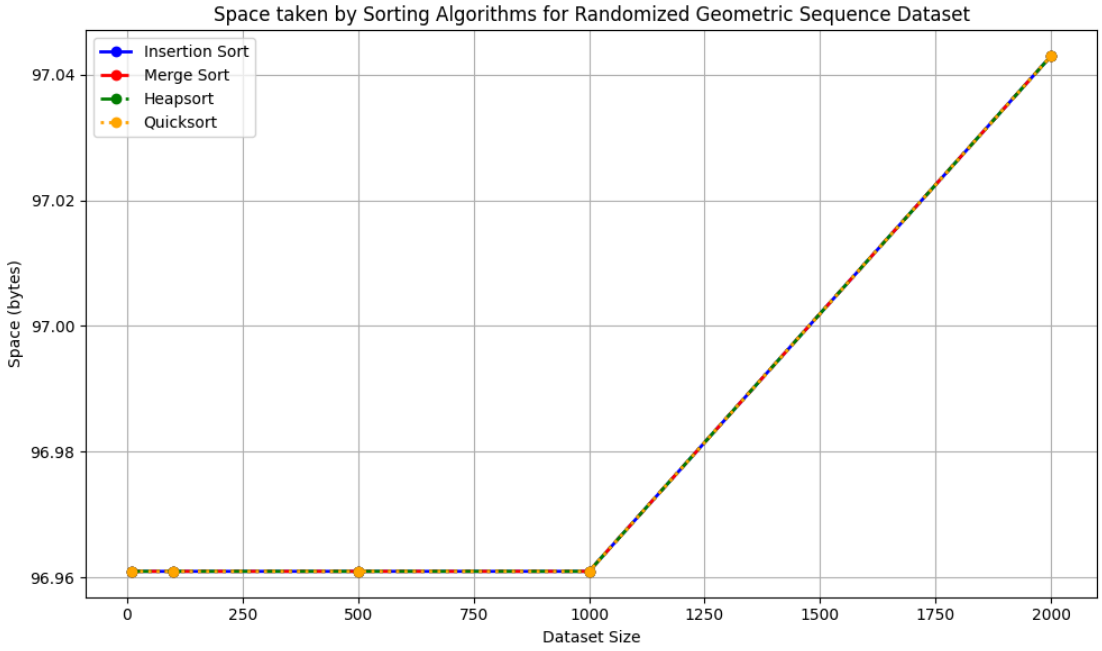


Fig 42: space taken by sorting algorithms for randomized geometric sequence dataset

Randomized Arithmetic Sequence Dataset

Dataset with values from an arithmetic sequence in random order.

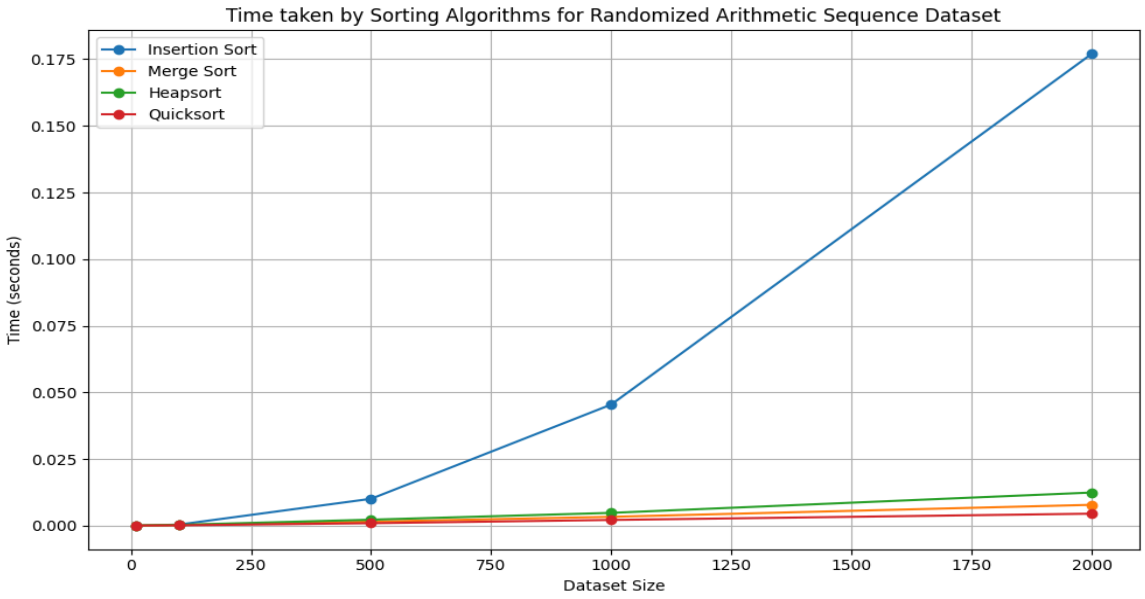


Fig 43: time taken by sorting algorithms for randomized arithmetic sequence dataset

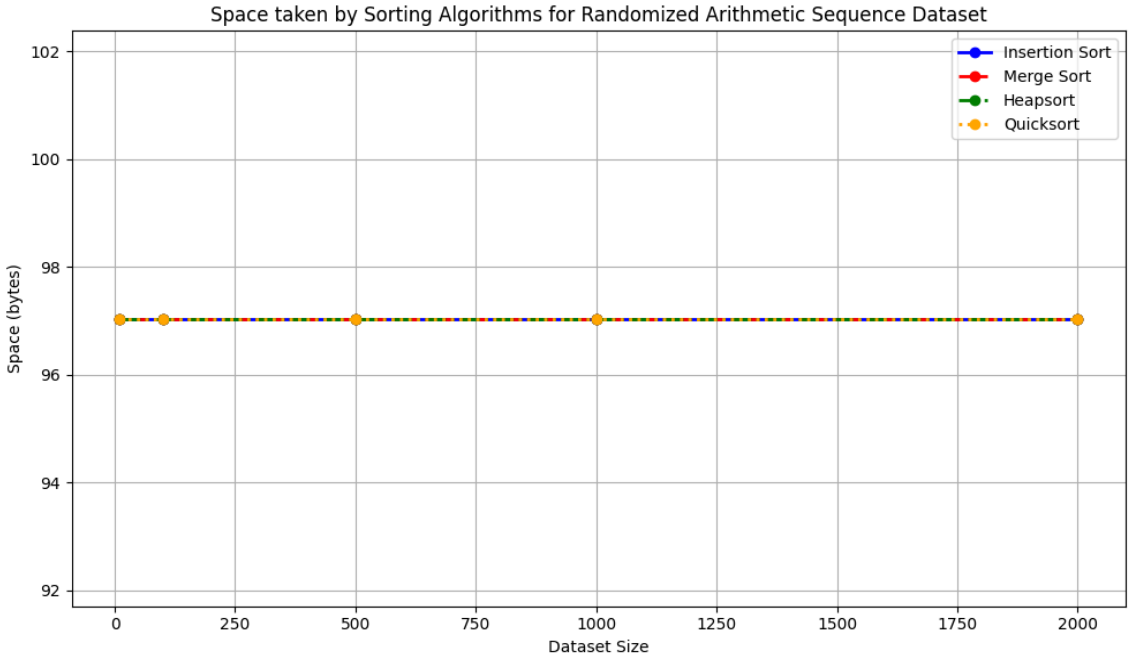


Fig 44: space taken by sorting algorithms for randomized arithmetic sequence dataset

Randomized Prime Number Dataset

The dataset consists of prime numbers in random order.

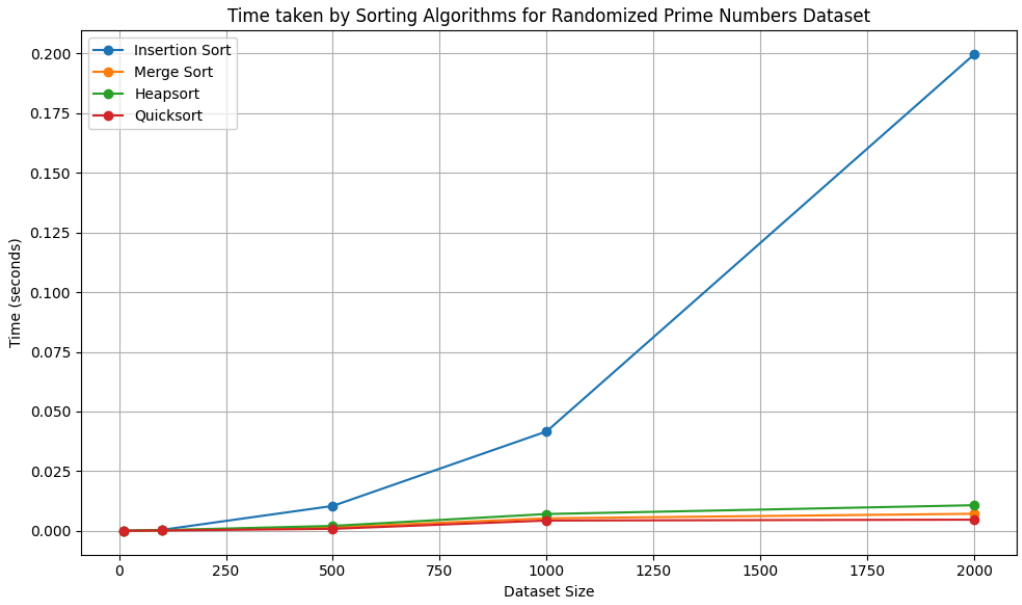


Fig 45: time taken by sorting algorithms for randomized prime nmber dataset

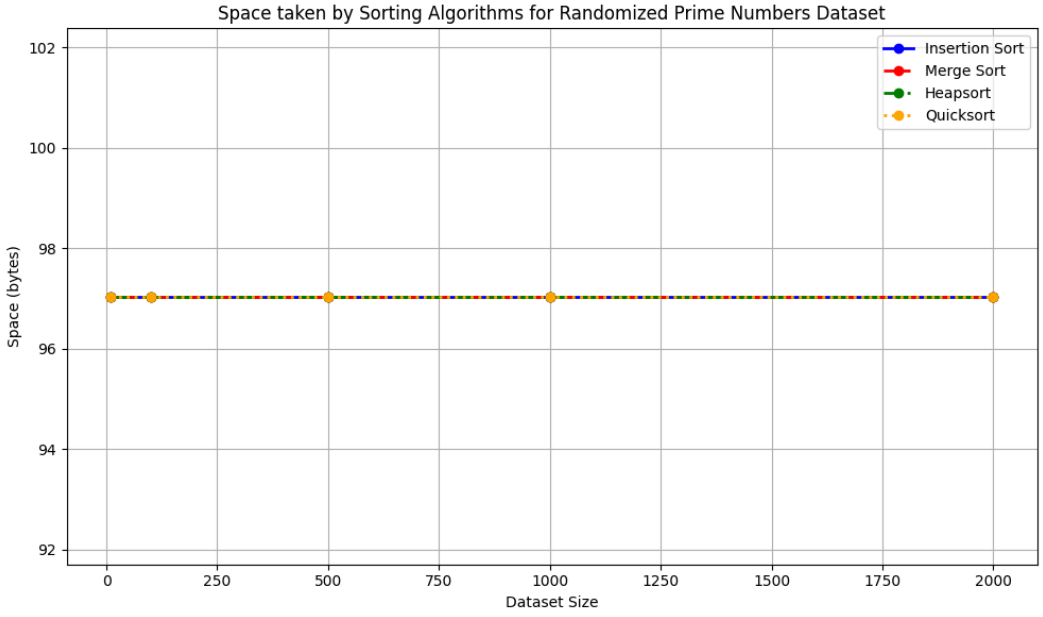


Fig 46: space taken by sorting algorithms for randomized prime number dataset

Randomized Fibonacci Mod Sequence Dataset

a dataset with values from a Fibonacci sequence modulo 10, in random order

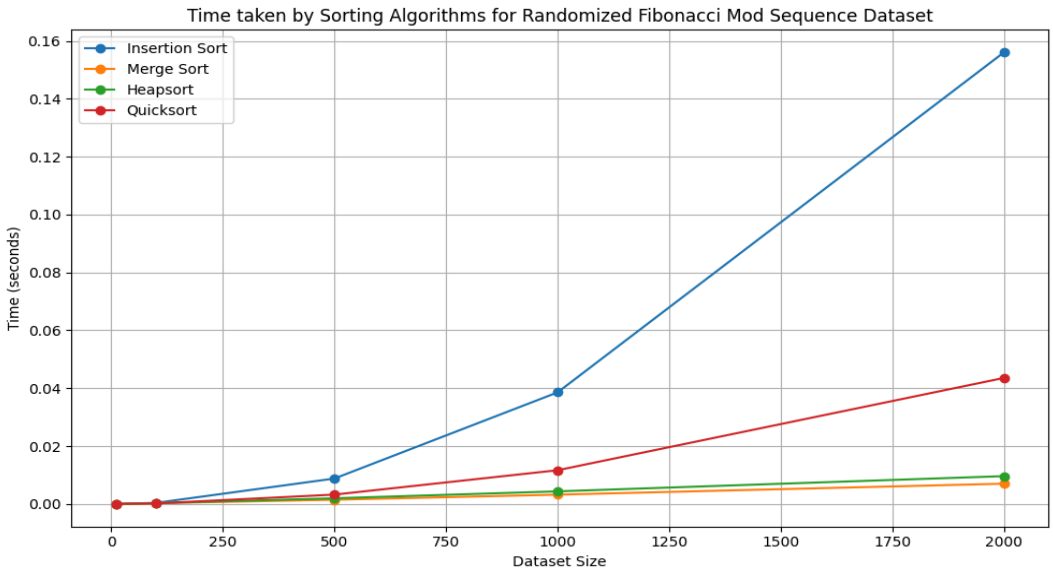


Fig 47: time taken by sorting algorithms for randomized Fibonacci mod sequence dataset

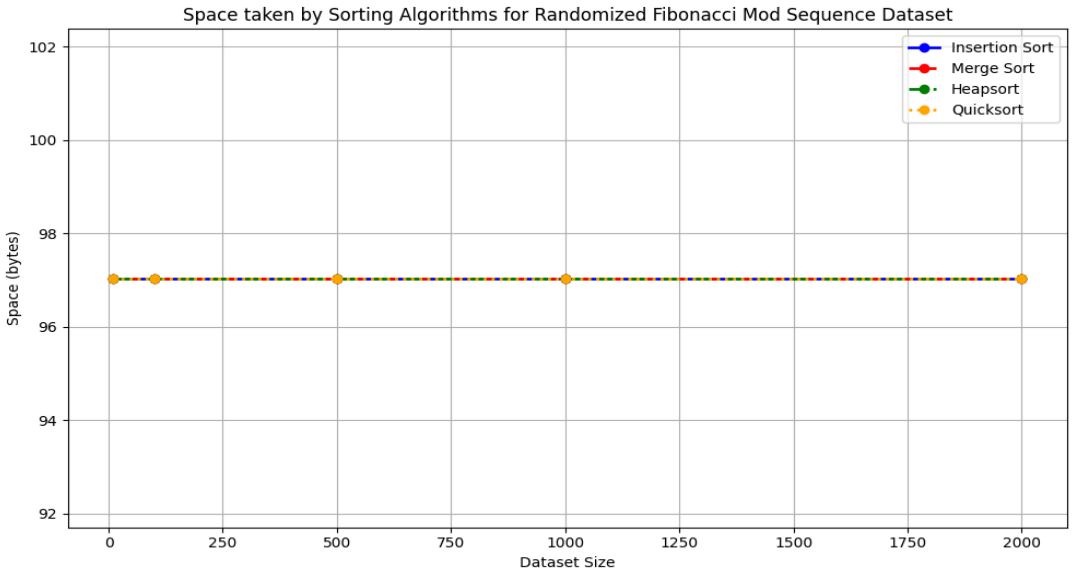


Fig 48: space taken by sorting algorithms for randomized fibonacci mod sequence dataset

Randomized Power of Two Dataset

A dataset with powers of two in random order.

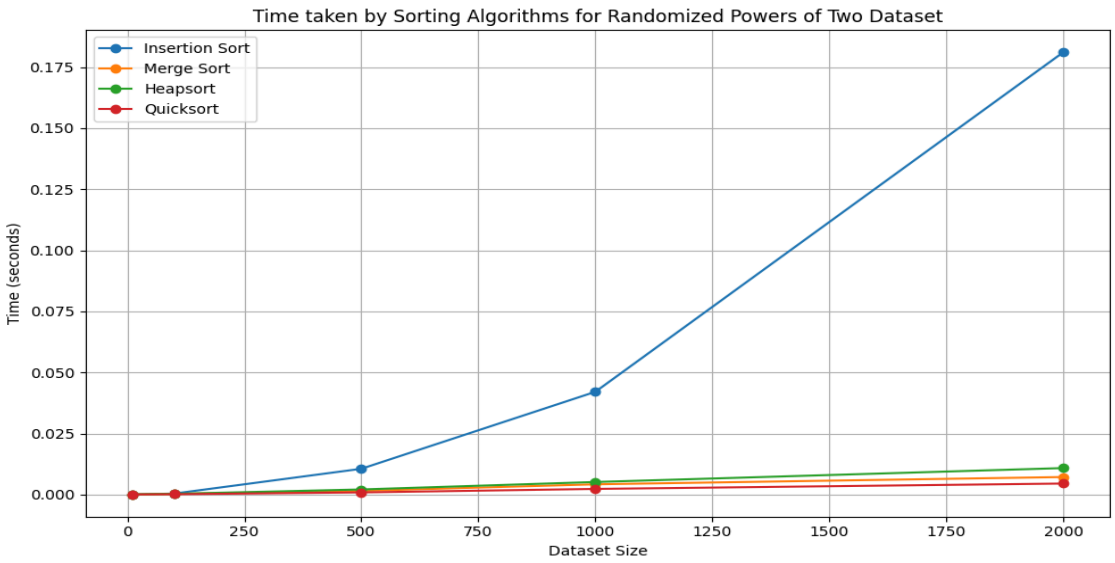


Fig 49: time taken by sorting algorithms for randomized power of two dataset

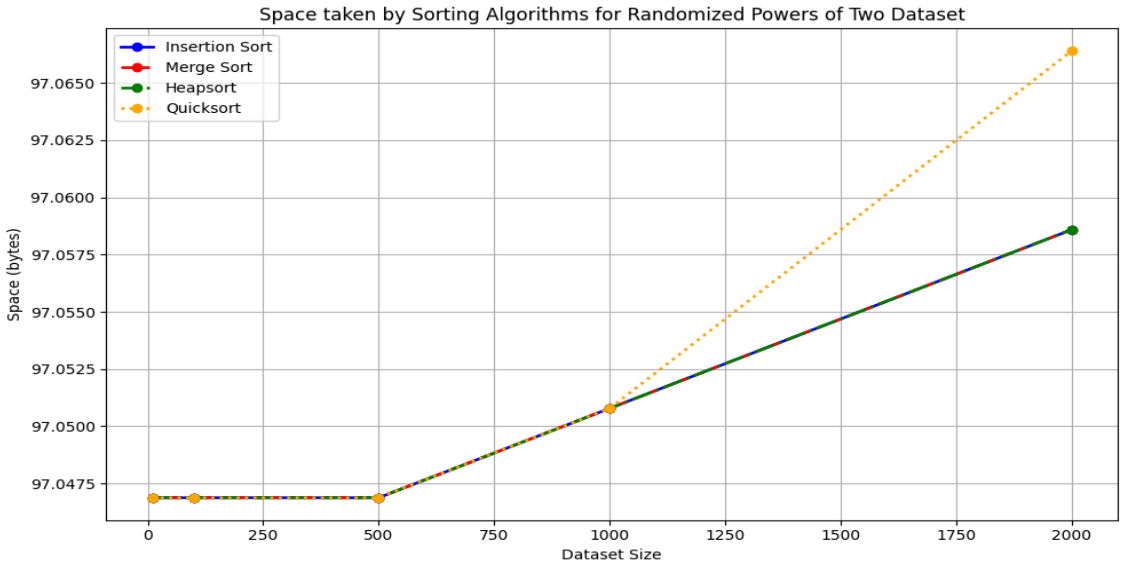


Fig 50: space taken by sorting algorithms for randomized power of two dataset

Randomized Fourier Series Dataset

A dataset with values from a Fourier series in random order

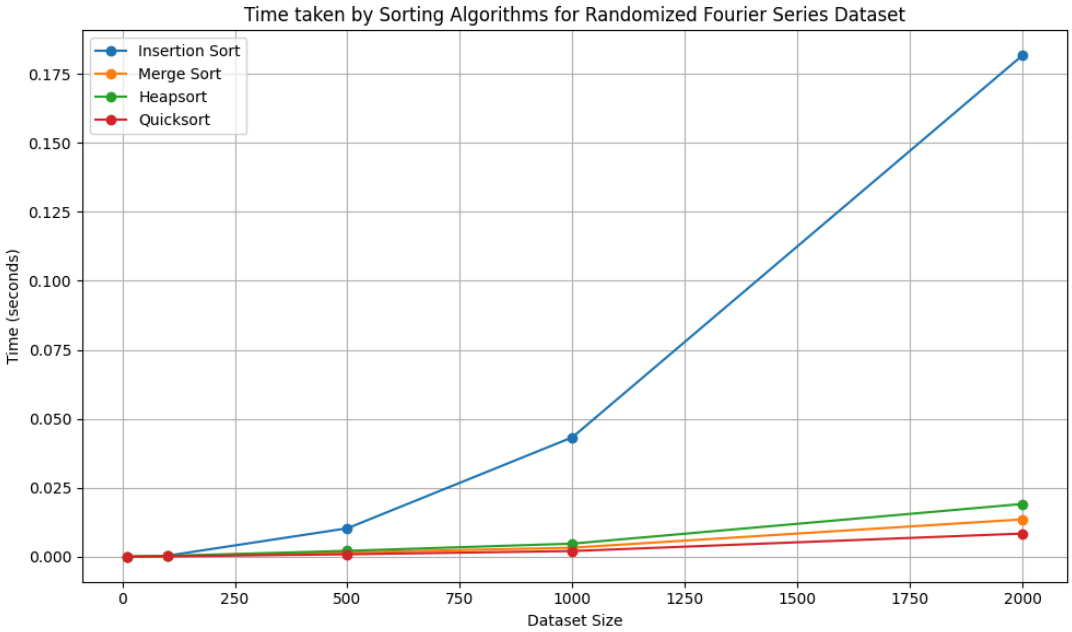


Fig 51: time taken by sorting algorithms for randomized fourier dataset

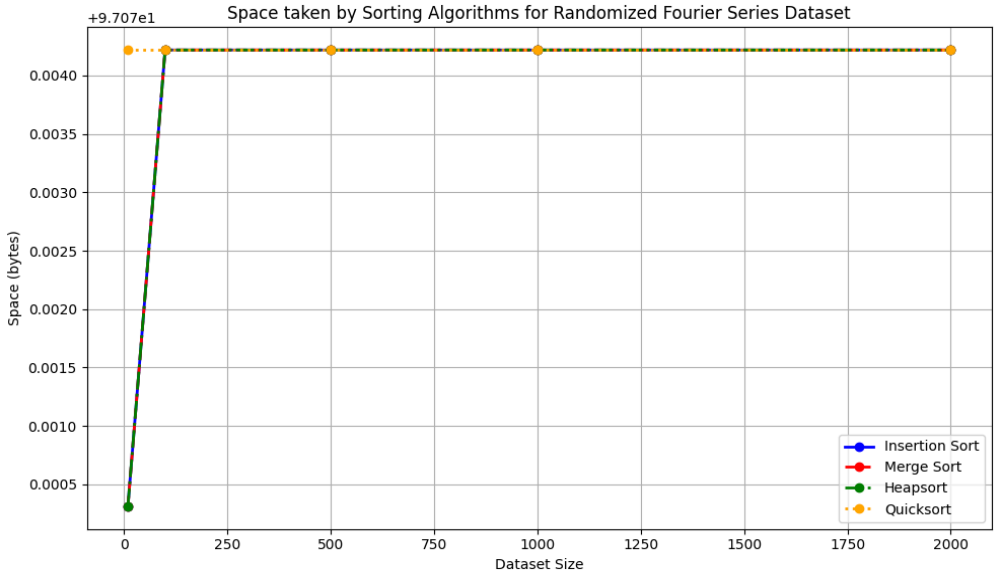


Fig 52: space taken by sorting algorithms for randomized Fourier series dataset

Perlin Noise Sequence Dataset

A dataset based on Perlin noise, Perlin noise sequence generate natural looking texures and patterns.

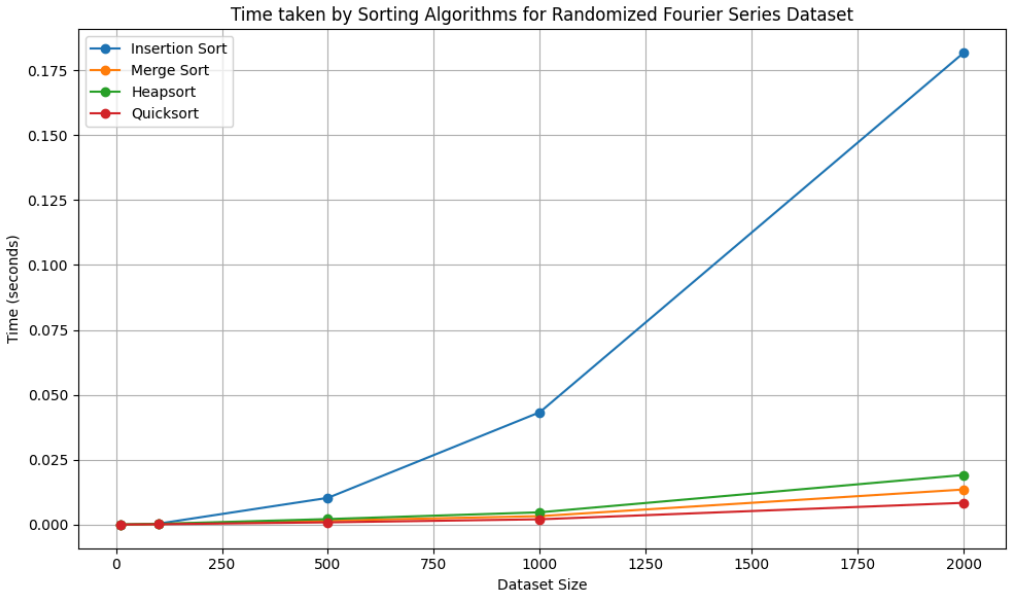


Fig 53: time taken by sorting algorithms for randomized Perlin noise sequence dataset

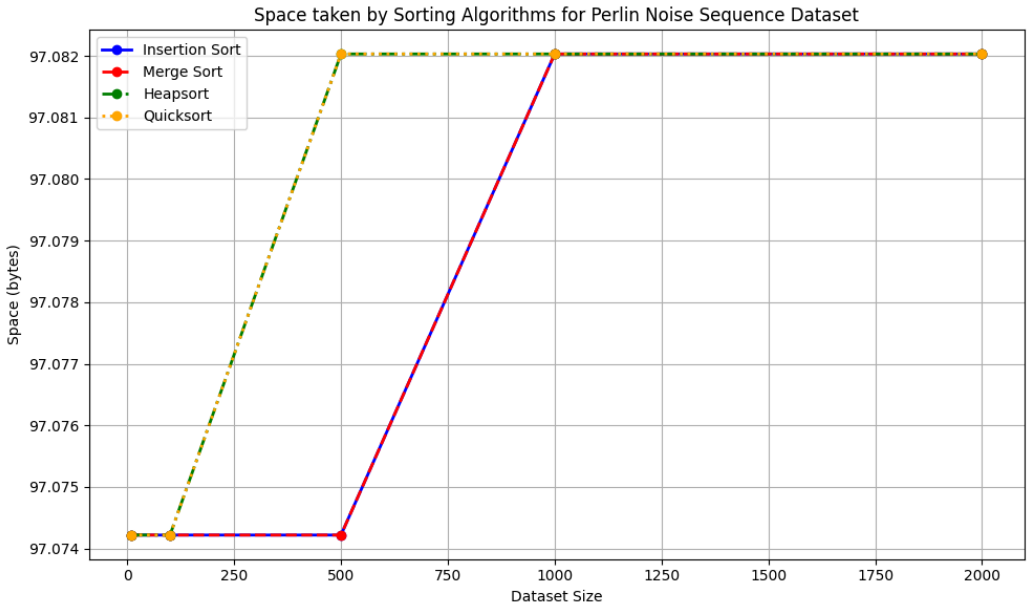


Fig 54: space taken by sorting algorithms for randomized Perlin noise sequence dataset

DISCUSSION

Performance comparison of sorting algorithms

The experimental analysis of algorithms on diverse datasets highlights the core weaknesses and strengths of an algorithm. in this experimental analysis, we have done experiments on four different sorting algorithms. The four algorithms are Insertion sort, Merge sort, Heap sort, and Quick sort. The purpose of this experiment was to do a comprehensive analysis of these algorithms and find out in which situations they perform well and in which situations they struggle to keep up. And also to compare these four algorithms to find which algorithm fits perfectly in certain situations.

From the result of the experiment, we can see that when a dataset consists of random values with no patterns, in terms of time insertion sort performs worse and the graph increases exponentially, whereas Heapsort, Mergesort, and Quicksort perform well. quicksort takes less amount to time in this case to sort values in a random value dataset. in terms of space insertion sort performs better than others till 1000 elements in data which makes it the least space-hungry.

When data becomes sorted or when we use Fibonacci we see a huge time raise to sort data using quicksort. whereas the other three algorithms perform with a time complexity of O(nlogn). in terms of space we see a spike in quicksort space usage.

When the data get reversly sorted quicksort as well as insertion sort takes exponential time to complete sorting. but mergesort and heapsort take (nlogn) amount of time which keeps them consistent with previous findings.

we can see a similar trend of insertion sort performing worse when dataset contains few unique elements, all unique elements, small numerical range between numbers, large numerical range between numbers, lexicographical, reverse lexicographical, floating point dataset, negative and positive number, mixed data types, randomized Fibonacci sequence, randomized geometric sequence, randomize arithmetic sequence, randomized prime number sequence, randomized Fibonacci mod-sequence, randomized power of two sequences, randomized Fourier series, and Perlin Noise sequence. On the other hand, insertion sort performs well when a data set consists of sparse data, prime number sequence, few large gaps between values, and sorted Fibonacci series. in terms of space, it occupies less or similar space than the other three algorithms.

Quick sort performs badly when the dataset contains data in sorted order, reverse sorted order, sorted Fibonacci sequence, few larger gaps between values in the dataset, data alternate between high and low, sorted prime number, spars data or the dataset contains a lot of zero and one values. other than these cases quick sort performs in O(nlogn) times.

Merge sort and Heap sort are shown to perform in around O(nlogn) time in all the cases which them extremely stable in the average case scenarios. Even Though merge sort occupied more space while sorting in the experiment.

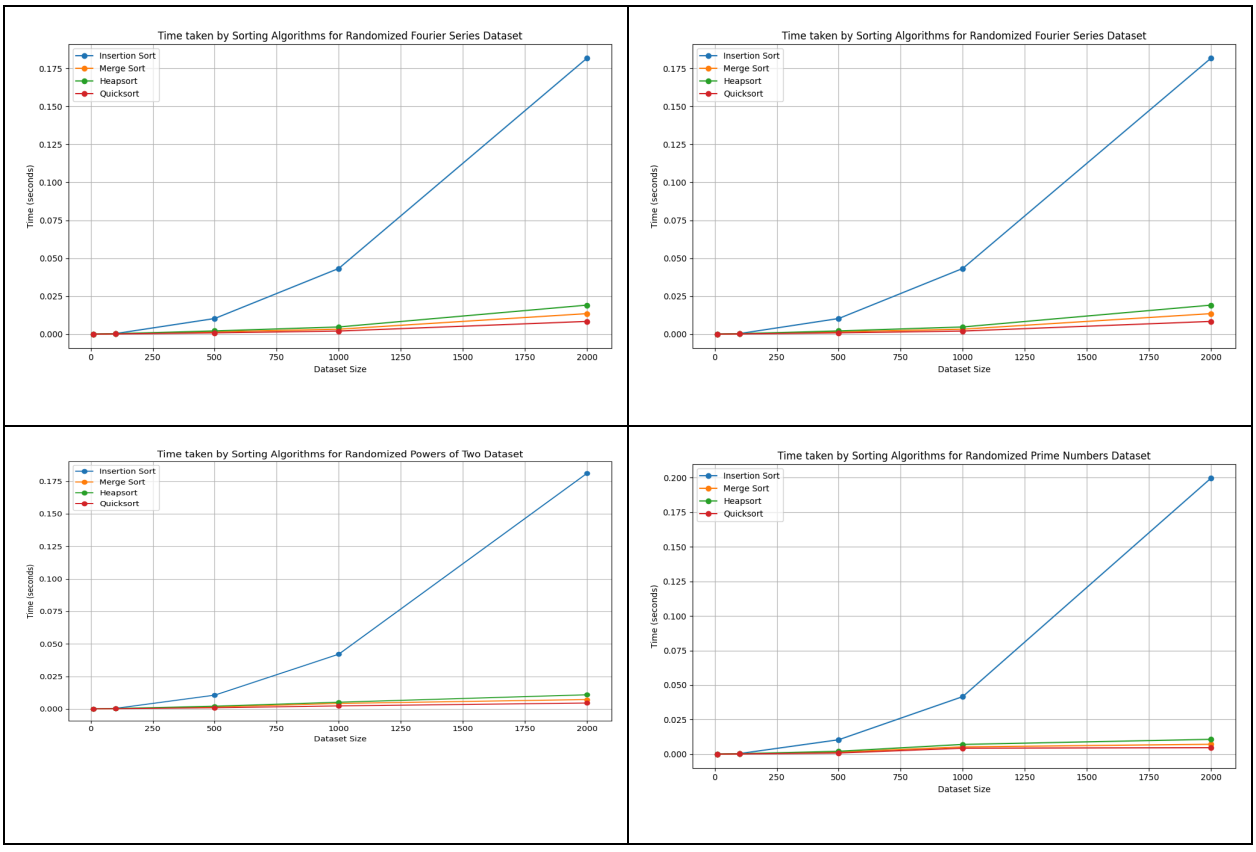
Most suitable algorithms for sorting packages in logistic scenarios.

To handle packages in the logistic company scenario, there are three algorithms that can be suitable. Among them Quick Sort is the most suitable one. the other two algorithms are Merge sort and Heap sort. Because the logistics company receives thousands of packages daily with different weights, sizes, and delivery destinations. which means all data are randomized and in randomized situations, Quick sort outperforms any other sorting algorithms.

Quick sort is more suitable because when we compare it to merge sort we will see that in average case it takes around a similar or less amount of time but the main difference is made in space as merge sort requires more space to perform tasks.

Quick sort is more suitable than heap sort in this scenario because of adaptability. quicksort's partitioning-based approach allows it to adapt quickly to changes in the data set. whereas heap sort relies on maintaining binary heap data. any changes in the dataset require it to rebuild the entire heap which can be less efficient than quicksort. also, heap sort's design involves elements to maintain specific order which can also hamper the process as the company receives thousands of packages with different weights, sizes, and delivery destinations

Experimental data to support Quicksort performs better than others in randomized situations,



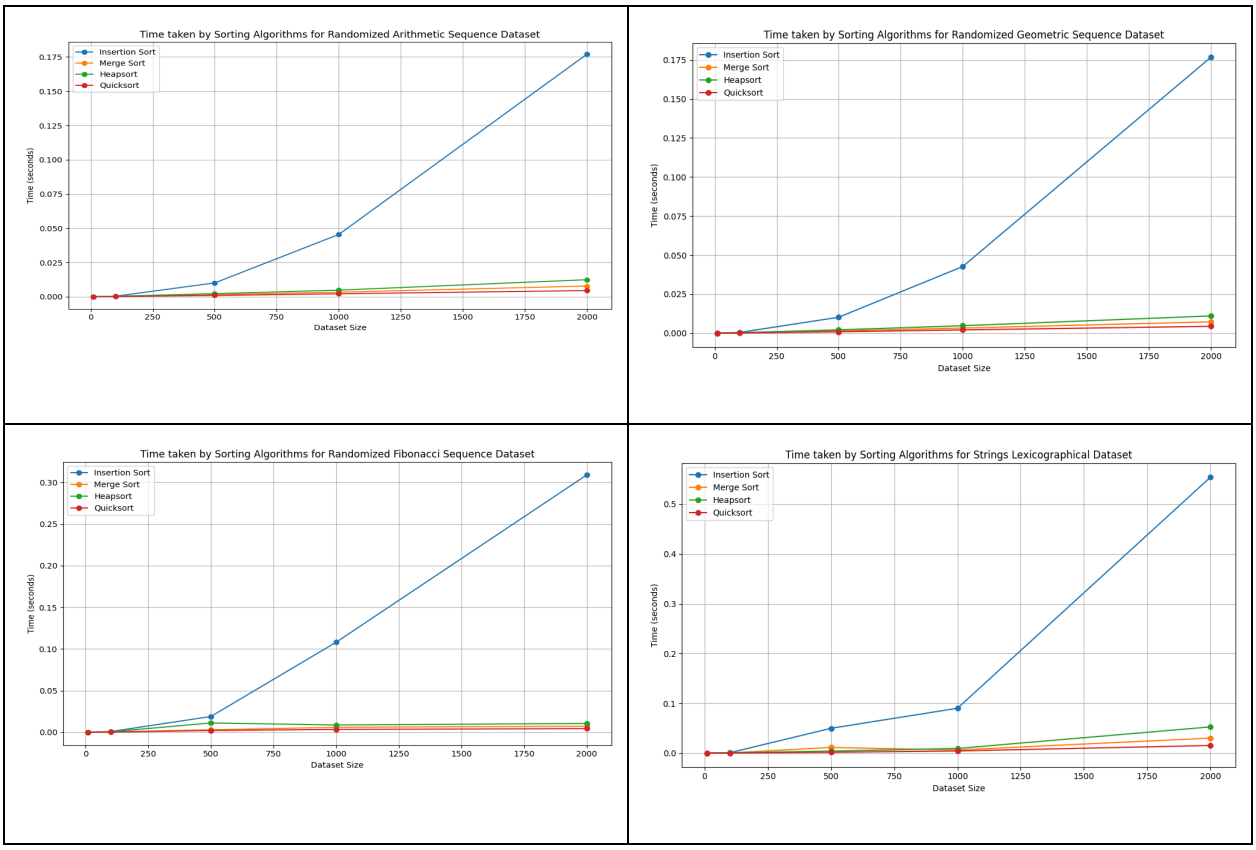


Fig 55: Quick sort time measurement

From the above data we can state that when data get randomized, Quicksort outperforms every other algorithm in sorting.

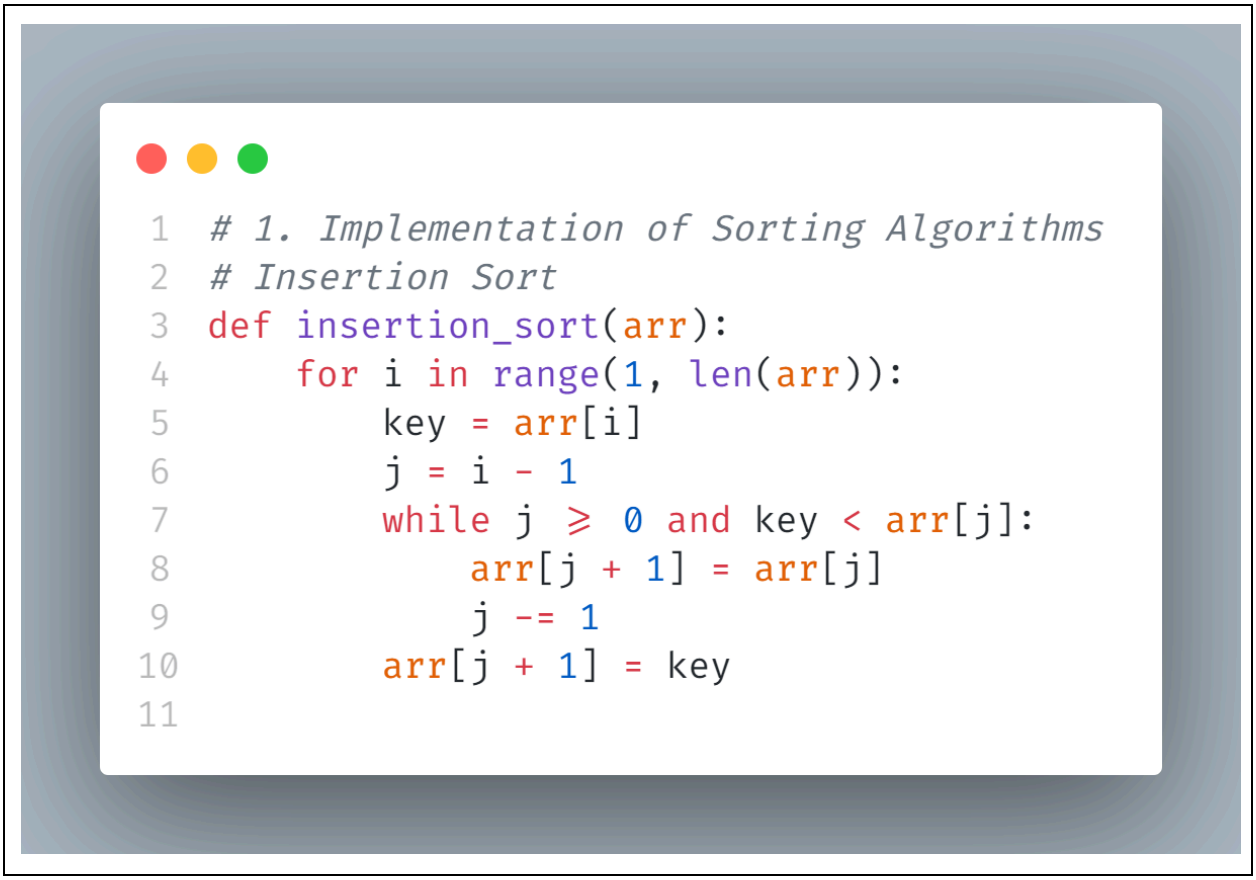
==========This Section Contains the Code of the Expreriment==========

CODE

Google Colab link:

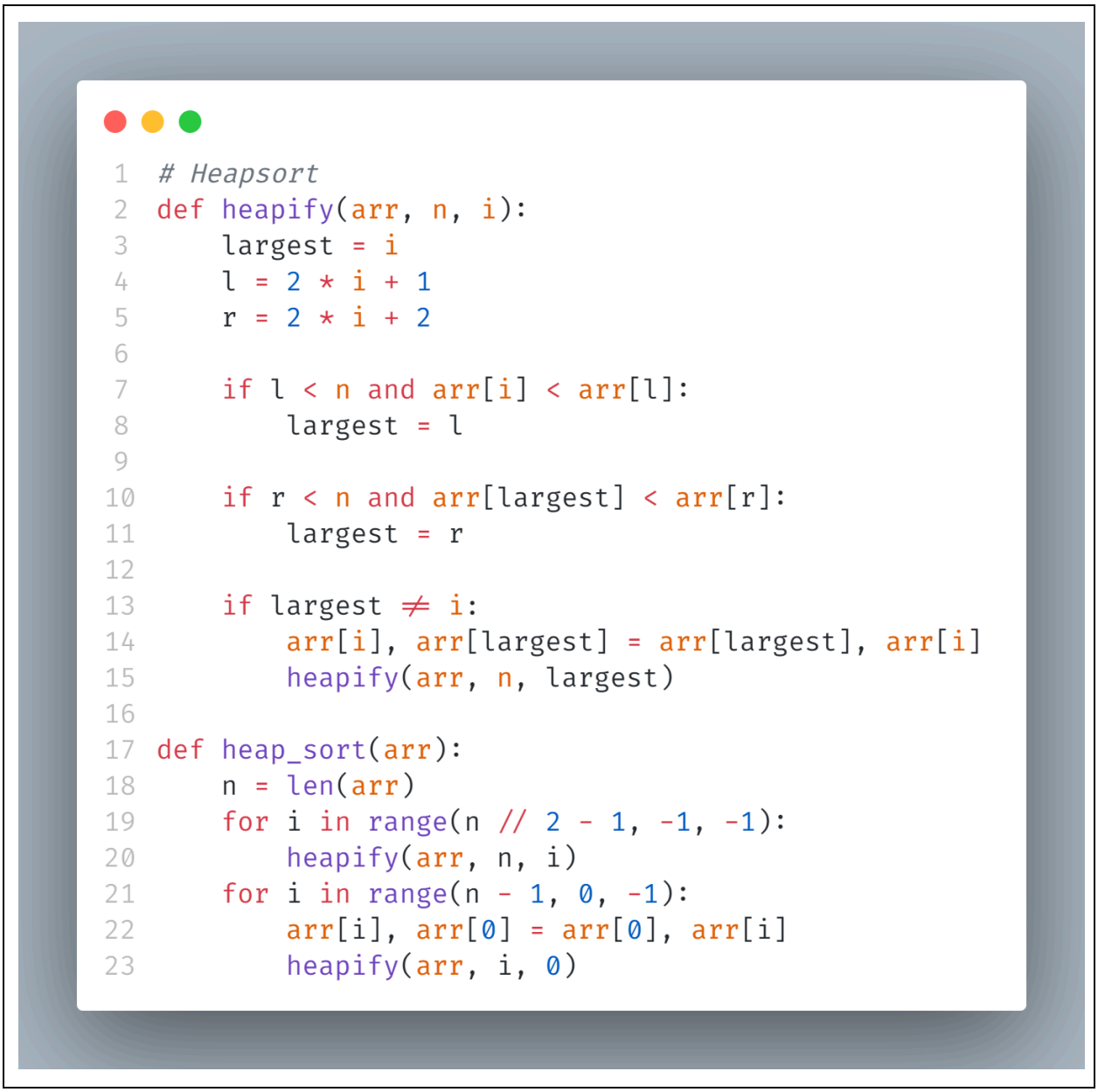
https://colab.research.google.com/drive/1IRPZ66Mhkh4mwCRTBcVem7wQ-1GgrQz5?usp=sharing

```
1 import time
2 import random
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from memory_profiler import memory_usage
6 from memory_profiler import profile
7 import string
8 import math
```



```python
# 1. Implementation of Sorting Algorithms
# Insertion Sort
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

```

```python
# Merge Sort
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

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

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



```python
# Heapsort
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)

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

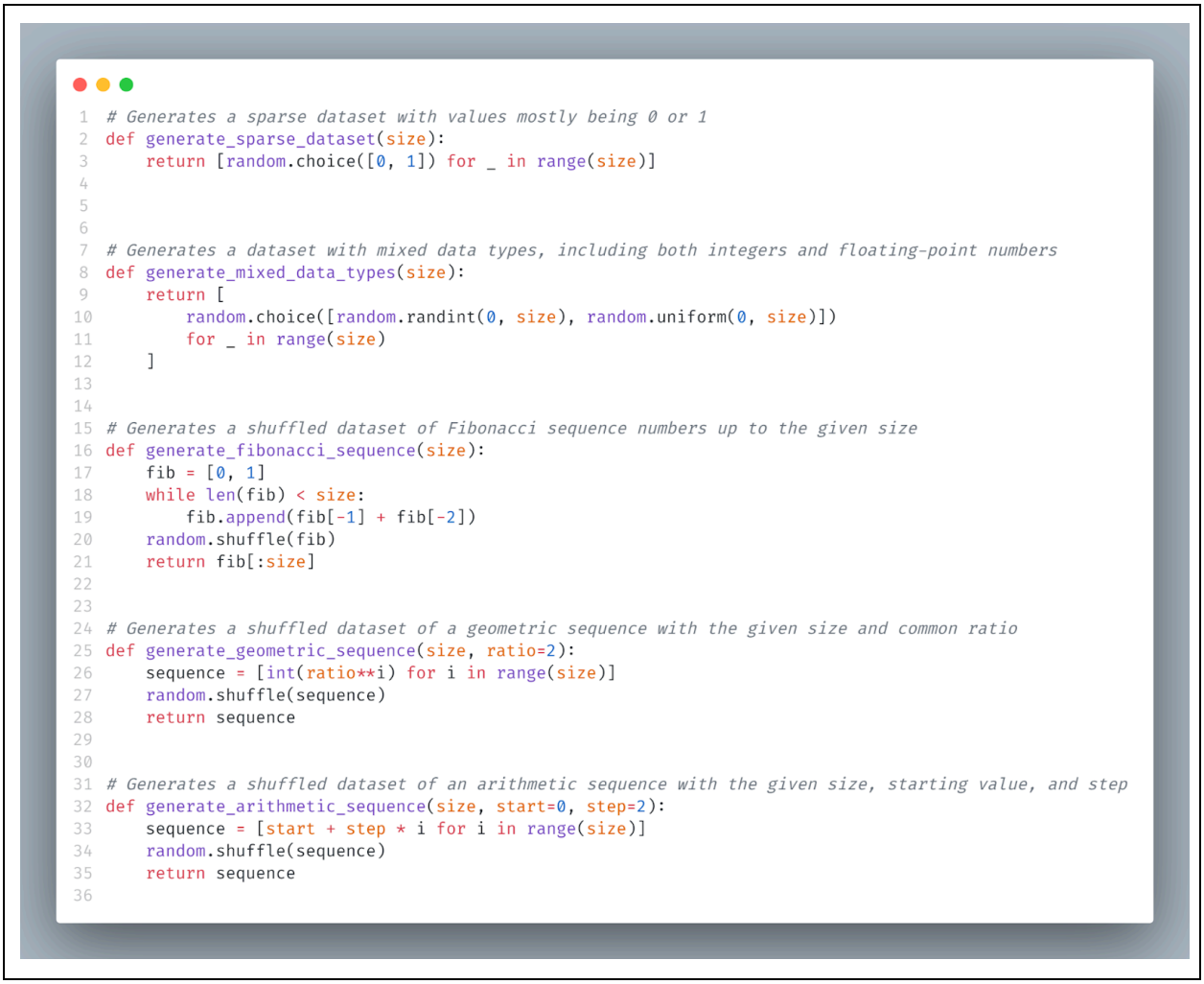
```python
# Quicksort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    stack = [(0, len(arr) - 1)]
    while stack:
        low, high = stack.pop()
        if low < high:
            pivot_index = partition(arr, low, high)
            stack.append((low, pivot_index - 1))
            stack.append((pivot_index + 1, high))

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

```python
# Experimental Analysis

# Generates a dataset of random integers ranging from 1 to 10000
def generate_random_dataset(size):
    return [random.randint(1, 10000) for _ in range(size)]

# Generates a dataset of integers in ascending order
def generate_sorted_dataset(size):
    return list(range(size))

# Generates a dataset of integers in descending order
def generate_reverse_sorted_dataset(size):
    return list(range(size, 0, -1))

# Generates a dataset of Fibonacci numbers up to 'size'
def generate_fibonacci_dataset(size):
    fib = [0, 1]
    while len(fib) < size:
        fib.append(fib[-1] + fib[-2])
    return fib[:size]

# Generates a dataset of integers nearly sorted with 10% of elements randomly swapped
def generate_nearly_sorted_order(size):
    arr = list(range(size))
    for _ in range(size // 10):  # Swap 10% of the elements
        idx1, idx2 = random.sample(range(size), 2)
        arr[idx1], arr[idx2] = arr[idx2], arr[idx1]
    return arr
```

```python
# Generates a dataset with a few unique elements repeated multiple times
def generate_few_unique_elements(size):
    unique_elements = random.sample(range(1, size // 10 + 1), size // 10)
    return [random.choice(unique_elements) for _ in range(size)]


# Generates a dataset with all unique elements
def generate_all_unique_elements(size):
    return random.sample(range(size * 2), size)


# Generates a dataset with values within a small range (0 to 10)
def generate_small_range_of_values(size):
    return [random.randint(0, 10) for _ in range(size)]


# Generates a dataset with values within a large range (0 to 10000)
def generate_large_range_of_values(size):
    return [random.randint(0, 10000) for _ in range(size)]


# Generates a dataset with values having large gaps between them
def generate_few_large_gaps(size):
    base = 10
    return [i * base + random.randint(0, base) for i in range(size)]
```

```python
# Generates a dataset with alternating high and low values
def generate_alternating_high_low(size):
    return [i if i % 2 == 0 else size - i for i in range(size)]


# Generates a large dataset with values up to one-fifth of the size
def generate_large_dataset(size):
    return [random.randint(0, size // 5) for _ in range(size)]


# Generates a small dataset with values starting from the size up to five times the size
def generate_small_dataset(size):
    return [random.randint(size, size * 5) for _ in range(size)]
```

```python
# Generates a dataset with a length that is a prime number greater than or equal to the given size
def generate_prime_number_of_elements(size):
    def is_prime(n):
        if n <= 1:
            return False
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                return False
        return True

    prime = size
    while not is_prime(prime):
        prime += 1

    return list(range(prime))



# Generates a dataset of random strings sorted lexicographically
def generate_strings_lexicographical(size):
    return ["".join(random.choices(string.ascii_lowercase, k=5)) for _ in range(size)]


# Generates a dataset of random strings sorted in reverse lexicographical order
def generate_strings_reverse_lexicographical(size):
    return sorted(
        ["".join(random.choices(string.ascii_lowercase, k=5)) for _ in range(size)],
        reverse=True,
    )


# Generates a dataset of random floating-point numbers
def generate_floating_point_numbers(size):
    return [random.uniform(0, size) for _ in range(size)]


# Generates a dataset of random integers including both negative and positive values
def generate_negative_and_positive_numbers(size):
    return [random.randint(-size, size) for _ in range(size)]
```

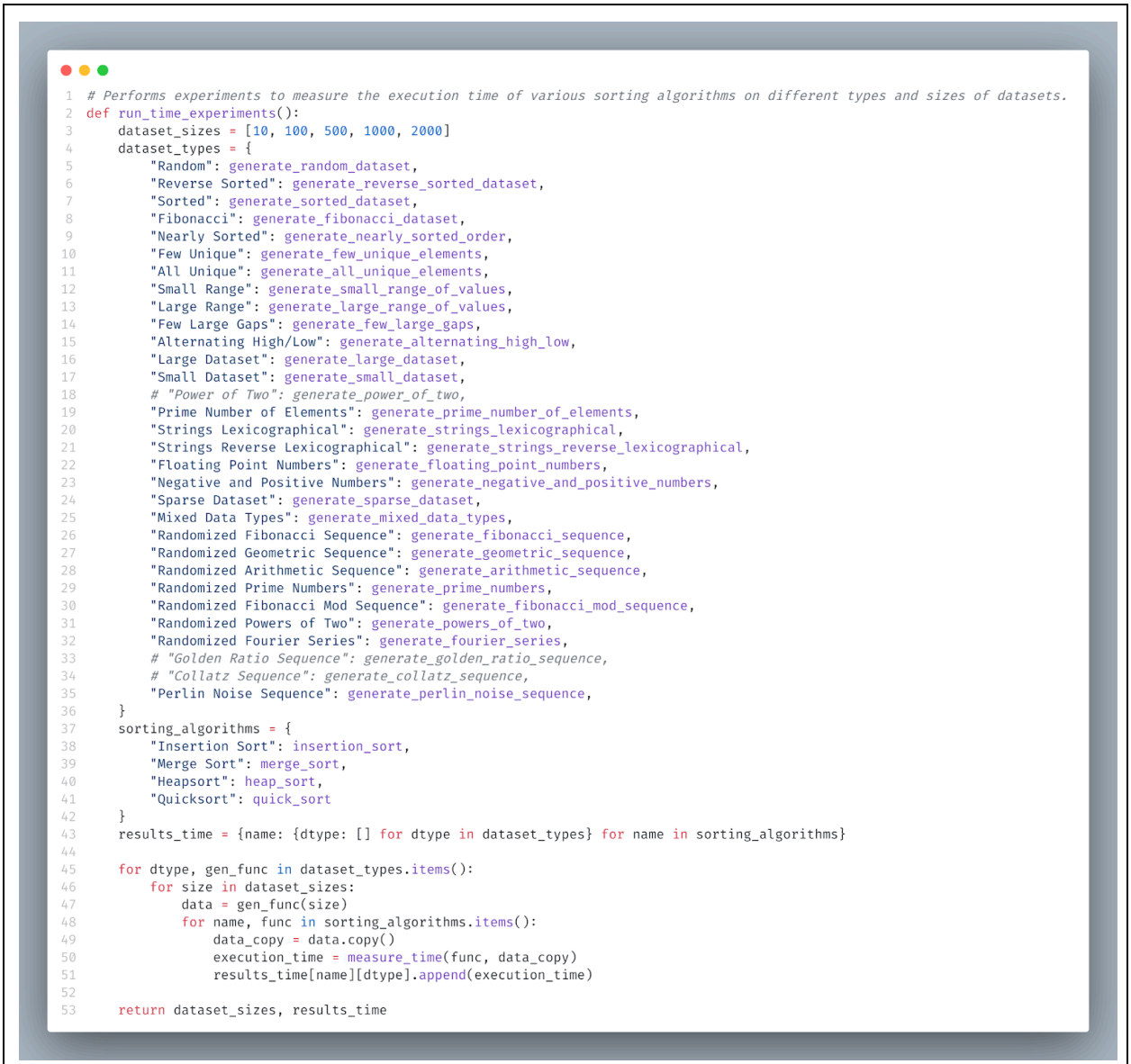


```python
# Generates a sparse dataset with values mostly being 0 or 1
def generate_sparse_dataset(size):
    return [random.choice([0, 1]) for _ in range(size)]



# Generates a dataset with mixed data types, including both integers and floating-point numbers
def generate_mixed_data_types(size):
    return [
        random.choice([random.randint(0, size), random.uniform(0, size)])
        for _ in range(size)
    ]


# Generates a shuffled dataset of Fibonacci sequence numbers up to the given size
def generate_fibonacci_sequence(size):
    fib = [0, 1]
    while len(fib) < size:
        fib.append(fib[-1] + fib[-2])
    random.shuffle(fib)
    return fib[:size]


# Generates a shuffled dataset of a geometric sequence with the given size and common ratio
def generate_geometric_sequence(size, ratio=2):
    sequence = [int(ratio**i) for i in range(size)]
    random.shuffle(sequence)
    return sequence


# Generates a shuffled dataset of an arithmetic sequence with the given size, starting value, and step
def generate_arithmetic_sequence(size, start=0, step=2):
    sequence = [start + step * i for i in range(size)]
    random.shuffle(sequence)
    return sequence

```

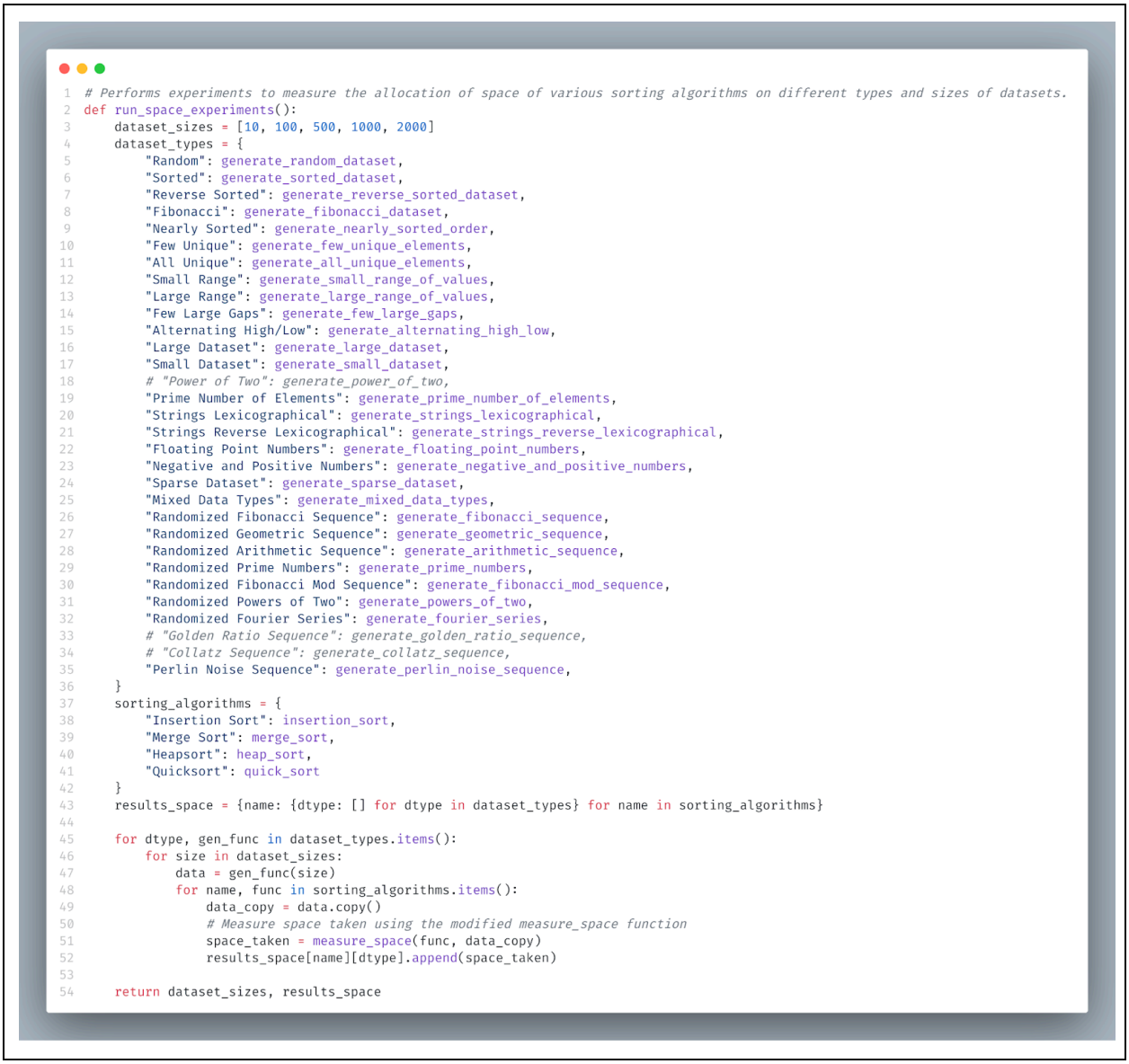
```python
# Generates a shuffled dataset of prime numbers up to the given size
def generate_prime_numbers(size):
    primes = []
    num = 2
    while len(primes) < size:
        if all(num % p != 0 for p in primes):
            primes.append(num)
        num += 1
    random.shuffle(primes)
    return primes


# Generates a shuffled dataset of Fibonacci sequence numbers modulo a given value
def generate_fibonacci_mod_sequence(size, mod=10):
    fib = [0, 1]
    while len(fib) < size:
        fib.append((fib[-1] + fib[-2]) % mod)
    random.shuffle(fib)
    return fib[:size]


# Generates a shuffled dataset of powers of two up to the given size
def generate_powers_of_two(size):
    powers = [2**i for i in range(size)]
    random.shuffle(powers)
    return powers


# Generates a shuffled dataset based on a Fourier series with a specified number of terms
def generate_fourier_series(size, terms=5):
    import math

    def fourier_series(x, n):
        return sum(math.sin((2 * k - 1) * x) / (2 * k - 1) for k in range(1, n + 1))

    series = [fourier_series(i, terms) for i in range(size)]
    random.shuffle(series)
    return series

```

```python
# Generates a dataset based on Perlin noise
def generate_perlin_noise_sequence(size):
    def fade(t):
        return t * t * t * (t * (t * 6 - 15) + 10)

    def lerp(t, a, b):
        return a + t * (b - a)

    def grad(hash, x):
        h = hash & 15
        grad = 1 + (h & 7)
        if h & 8:
            grad = -grad
        return grad * x

    def perlin(x):
        xi = int(x) & 255
        xf = x - int(x)
        u = fade(xf)
        return lerp(u, grad(p[xi], xf), grad(p[xi + 1], xf - 1)) * 2

    p = list(range(256))
    random.shuffle(p)
    p += p
    return [perlin(i / 10.0) for i in range(size)]
```

```python
# Measures the time taken to sort the data using the provided sorting function
def measure_time(sort_func, data):
    start_time = time.time()
    sort_func(data)
    return time.time() - start_time


# Measures the peak memory usage during the sorting process using the provided sorting function
def measure_space(sort_func, data):
    # Wrapper function to use memory_usage for measuring peak memory usage
    def wrapper():
        sort_func(data)
    mem_usage = memory_usage(wrapper, max_usage=True)
    return np.mean(mem_usage)
```



```python
# Performs experiments to measure the execution time of various sorting algorithms on different types and sizes of datasets.
def run_time_experiments():
    dataset_sizes = [10, 100, 500, 1000, 2000]
    dataset_types = {
        "Random": generate_random_dataset,
        "Reverse Sorted": generate_reverse_sorted_dataset,
        "Sorted": generate_sorted_dataset,
        "Fibonacci": generate_fibonacci_dataset,
        "Nearly Sorted": generate_nearly_sorted_order,
        "Few Unique": generate_few_unique_elements,
        "All Unique": generate_all_unique_elements,
        "Small Range": generate_small_range_of_values,
        "Large Range": generate_large_range_of_values,
        "Few Large Gaps": generate_few_large_gaps,
        "Alternating High/Low": generate_alternating_high_low,
        "Large Dataset": generate_large_dataset,
        "Small Dataset": generate_small_dataset,
        # "Power of Two": generate_power_of_two,
        "Prime Number of Elements": generate_prime_number_of_elements,
        "Strings Lexicographical": generate_strings_lexicographical,
        "Strings Reverse Lexicographical": generate_strings_reverse_lexicographical,
        "Floating Point Numbers": generate_floating_point_numbers,
        "Negative and Positive Numbers": generate_negative_and_positive_numbers,
        "Sparse Dataset": generate_sparse_dataset,
        "Mixed Data Types": generate_mixed_data_types,
        "Randomized Fibonacci Sequence": generate_fibonacci_sequence,
        "Randomized Geometric Sequence": generate_geometric_sequence,
        "Randomized Arithmetic Sequence": generate_arithmetic_sequence,
        "Randomized Prime Numbers": generate_prime_numbers,
        "Randomized Fibonacci Mod Sequence": generate_fibonacci_mod_sequence,
        "Randomized Powers of Two": generate_powers_of_two,
        "Randomized Fourier Series": generate_fourier_series,
        # "Golden Ratio Sequence": generate_golden_ratio_sequence,
        # "Collatz Sequence": generate_collatz_sequence,
        "Perlin Noise Sequence": generate_perlin_noise_sequence,
    }
    sorting_algorithms = {
        "Insertion Sort": insertion_sort,
        "Merge Sort": merge_sort,
        "Heapsort": heap_sort,
        "Quicksort": quick_sort
    }
    results_time = {name: {dtype: [] for dtype in dataset_types} for name in sorting_algorithms}

    for dtype, gen_func in dataset_types.items():
        for size in dataset_sizes:
            data = gen_func(size)
            for name, func in sorting_algorithms.items():
                data_copy = data.copy()
                execution_time = measure_time(func, data_copy)
                results_time[name][dtype].append(execution_time)

    return dataset_sizes, results_time
```



```python
# Performs experiments to measure the allocation of space of various sorting algorithms on different types and sizes of datasets.
def run_space_experiments():
    dataset_sizes = [10, 100, 500, 1000, 2000]
    dataset_types = {
        "Random": generate_random_dataset,
        "Sorted": generate_sorted_dataset,
        "Reverse Sorted": generate_reverse_sorted_dataset,
        "Fibonacci": generate_fibonacci_dataset,
        "Nearly Sorted": generate_nearly_sorted_order,
        "Few Unique": generate_few_unique_elements,
        "All Unique": generate_all_unique_elements,
        "Small Range": generate_small_range_of_values,
        "Large Range": generate_large_range_of_values,
        "Few Large Gaps": generate_few_large_gaps,
        "Alternating High/Low": generate_alternating_high_low,
        "Large Dataset": generate_large_dataset,
        "Small Dataset": generate_small_dataset,
        # "Power of Two": generate_power_of_two,
        "Prime Number of Elements": generate_prime_number_of_elements,
        "Strings Lexicographical": generate_strings_lexicographical,
        "Strings Reverse Lexicographical": generate_strings_reverse_lexicographical,
        "Floating Point Numbers": generate_floating_point_numbers,
        "Negative and Positive Numbers": generate_negative_and_positive_numbers,
        "Sparse Dataset": generate_sparse_dataset,
        "Mixed Data Types": generate_mixed_data_types,
        "Randomized Fibonacci Sequence": generate_fibonacci_sequence,
        "Randomized Geometric Sequence": generate_geometric_sequence,
        "Randomized Arithmetic Sequence": generate_arithmetic_sequence,
        "Randomized Prime Numbers": generate_prime_numbers,
        "Randomized Fibonacci Mod Sequence": generate_fibonacci_mod_sequence,
        "Randomized Powers of Two": generate_powers_of_two,
        "Randomized Fourier Series": generate_fourier_series,
        # "Golden Ratio Sequence": generate_golden_ratio_sequence,
        # "Collatz Sequence": generate_collatz_sequence,
        "Perlin Noise Sequence": generate_perlin_noise_sequence,
    }
    sorting_algorithms = {
        "Insertion Sort": insertion_sort,
        "Merge Sort": merge_sort,
        "Heapsort": heap_sort,
        "Quicksort": quick_sort
    }
    results_space = {name: {dtype: [] for dtype in dataset_types} for name in sorting_algorithms}

    for dtype, gen_func in dataset_types.items():
        for size in dataset_sizes:
            data = gen_func(size)
            for name, func in sorting_algorithms.items():
                data_copy = data.copy()
                # Measure space taken using the modified measure_space function
                space_taken = measure_space(func, data_copy)
                results_space[name][dtype].append(space_taken)

    return dataset_sizes, results_space
```

```python
#plot time vs Dataset size graph
def plot_results_time(dataset_sizes, results_time):
    dataset_types = list(results_time.values())[0].keys()
    sorting_algorithms = list(results_time.keys())

    for dtype in dataset_types:
        plt.figure(figsize=(10, 6))
        plt.title(f"Time taken by Sorting Algorithms for {dtype} Dataset")
        plt.xlabel("Dataset Size")
        plt.ylabel("Time (seconds)")

        for name in sorting_algorithms:
            plt.plot(dataset_sizes, results_time[name][dtype], marker='o', label=name)

        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()


#plots space vs Dataset size graph
def plot_results_space(dataset_sizes, results_space):
    dataset_types = list(results_space.values())[0].keys()
    sorting_algorithms = list(results_space.keys())

    # Define line styles and colors
    line_styles = ['-', '--', '-.', ':']
    line_colors = ['blue', 'red', 'green', 'orange']

    for dtype in dataset_types:
        plt.figure(figsize=(10, 6))
        plt.title(f"Space taken by Sorting Algorithms for {dtype} Dataset")
        plt.xlabel("Dataset Size")
        plt.ylabel("Space (bytes)")

        for i, name in enumerate(sorting_algorithms):
            # Use different line style and color for each algorithm
            plt.plot(dataset_sizes, results_space[name][dtype], marker='o', linestyle=line_styles[i % len(line_styles)],
                     color=line_colors[i % len(line_colors)], label=name, linewidth=2)

        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()
```

```python
#main function
def main():
    dataset_sizes, results_time = run_time_experiments()  # Corrected order of return values
    plot_results_time(dataset_sizes, results_time)

    dataset_sizes, results_space = run_space_experiments()
    plot_results_space(dataset_sizes, results_space)

if __name__ == "__main__":
    main()
```