Comparison of sorting algorithms

INF221 Small project, NMBU, Spring 2024

Hanna Lye Moum
hanna.lye.moum@nmbu.no

Hanna Steine
hanna.steine@nmbu.no

Sunniva Strumse
sunniva.annette.staalesen.strumse@nmbu.no

March 24, 2024

Abstract

This paper examines the performance of three sorting algorithms; insertion sort, quicksort, and merge sort. In advance, we anticipate that the insertion sort algorithm will outperform the others when the arrays are small, while quicksort and merge sort are expected to excel as the array size increases. However, we hypothesize that the merge sort and the quicksort algorithms will outperform each other under different circumstances, due to their respective strengths and weaknesses. The plots provided in this paper are generated from the code in the following Git repository; https://github.com/HannaMoum/INF221.

1 Introduction

According to many computer scientists, sorting is the most fundamental problem in the study of algorithms. Several engineering issues are most effectively addressed at the algorithmic level, rather than by making minor adjustments to the code [2]. In this paper, the sorting algorithms insertion sort, merge sort, and quicksort will be discussed. The discussion will cover the implementation and benchmarking of the algorithms. Ultimately, the paper seeks to evaluate how sorting algorithms perform in practice, compared to their anticipated theoretical outcomes.

2 Theory

While both quicksort and merge sort employ the divide-and-conquer method, insertion sort is an incremental algorithm [2][1]. In theory, an incremental algorithm is suitable for sorting a small number of elements, while the divide-and-conquer methods are more efficient for a larger number of elements [2].

The insertion sort algorithm sorts by placing an unsorted element at its suitable position in each iteration. The procedure of the algorithm can be compared to the way a set of cards is sorted: The first card is considered to be sorted, and another unsorted card is selected. The card is placed to the right if it's larger and to the left if it's smaller than the card in hand. This procedure is repeated for each subsequent unsorted card, positioning each in its correct place. This method mirrors the procedure of the insertion sort algorithm [3].

For each element being sorted, the insertion sort algorithm performs a series of comparisons and shifts. While the exact number of operations varies, it can potentially involve multiple operations per element, especially in less optimal arrangements [2]. The complexity of the insertion sort algorithm is $O(n^2)$ in the worst and the average case, and $O(n)$ in the best case [3].

The divide-and-conquer method consists of three characteristic steps: divide, conquer and combine. Firstly, the problem is divided into one or more subproblems that are smaller instances of the same problem (divide). Secondly, the subproblems are solved recursively (conquer), and lastly, the solutions of the subproblems are combined to form a solution to the original problem (combine) [2].

Hence, a divide-and-conquer algorithm breaks down a large problem into smaller subproblems. Theoretically, this makes divide-and-conquer algorithms suitable when the number of elements is large [2]. As mentioned, the divide-and-conquer algorithms that will be discussed in this paper are merge sort and quicksort. The time complexity of the quicksort algorithm is $O(nlog(n))$ in both the average and the best case, but $O(n^2)$ in the worst case [5]. The time complexity of merge sort, on the other hand, is $O(nlog(n))$ in the worst, the average and the best case [4].

3 Methods

The three sorting algortihms were implemented based on pseudo code provided by Cormen et al. [2]. No attempt for optimization has been done. Each algorithm were benchmarked on test data with sizes increased by a factor of 2. Hence, the range for the input sizes for insertion sort and merge sort are $(2^0, 2^1, ..., 2^{13})$, while for quicksort it is limited to $(2^0, 2^1, ..., 2^9)$ due to computational limits in recursion depth for this algorithm. In addition each input are benchmarked three times; for sorted, reversed sorted, and for randomly sorted arrays of the respective lengths. The implementation of test data generation can be found in 5.

The benchmarking results in figures 1, 2 and 3 are produced by running each algorithm for each variation of each input size, for N executions and R repetitions. R repetitions has been set to 5, while N is designed to be dependent on the input size. Larger input corresponds to lower N, limiting the total runtime of the benchmarking. Eventually the mean time of all executions and repetitions are saved as the benchmarked time.

The benchmarking of running time variation, shown i figure 4, is produced by running a set number of executions (N_{set}), on the randomly sorted arrays. Each execution is timed and saved. N_{set} differs between the algorithms due to their difference in efficiency, and is set to a number limiting the total runtime. For insertion sort $N_{set} = 30$, for merge sort $N_{set} = 300$, and for quicksort $N_{set} = 10000$.

The benchmarking has been performed on macOS Somona 14.1.1 with Python V 3.11.5 and conda 23.7.4. The benchmarking implementation can be found in appendix D and the environment requirements in appendix H.

4 Results

To visualize the performance of the three sorting algorithms, a section of the results are presented in tables 1, 2 and 3 where the running time of the specific sorting algorithm are presented for different input sizes.

The results are also visualized using line charts in figure 1, 2 and 3. The x-axis represents the *input size*, indicating the size of the dataset being sorted. The y-axis represents the *running time*, measuring the time taken by each sorting algorithm to sort datasets of specific sizes. Each sorting algorithm's performance data is depicted as lines on the plot, where different colors distinguish the original type of array. The blue line represents a sorted array, the orange represents a reversed sorted array, and the green represents a randomly mixed array. The colored lines are compared to the theoretical time complexity (best, worst and average), represented by the black lines. The theoretical time complexity is scaled to be comparable with the actual benchmark results.

To indicate variations in each sorting algorithm, the mean execution time for different array sizes is illustrated in a line plot along its confidence intervals in figure 4. The line represents the *mean execution time* while the shaded area around the mean indicates the variance within the *confidence intervals*. In this plot, the orange line represents insertion sort, the blue line represents merge sort, and the green line represents quicksort.

4.1 Insertion sort

The benchmark summary in table 1 shows that has the highest running times, especially for large datasets. For example, with a size of 8192, the time for a random array is 6.5475 seconds. A sorted array is the best case scenario for the insertion sort algorithm, where execution time stays fast, e.g., 0.004 seconds for 8192 elements. The worst case scenario is a reversed arrays, where the execution time increases significantly as the array size increases as illustrated in figure 1.

Table 1: Benchmark results for the insertion sort algorithm.

size	random	sorted	reversed
1.0	0.0	0.0	0.0
2.0	0.0	0.0	0.0
4.0	0.0	0.0	0.0
64.0	0.0003	0.0001	0.0015
128.0	0.0015	0.0002	0.0033
256.0	0.0059	0.0003	0.0128
2048.0	0.3924	0.0012	0.9178
4096.0	1.6081	0.0023	3.3036
8192.0	6.5475	0.004	13.3908

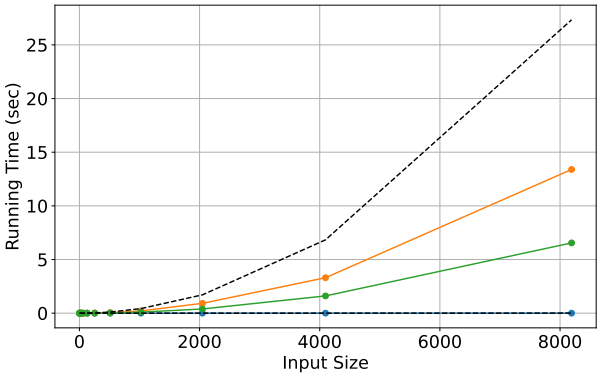

Figure 1: Benchmark results for the insertion sort algorithm in seconds.

4.2 Merge sort

The benchmark summary in table 2 show a consistently low execution time across different dataset types and sizes. For example, with a size of 54, the time for a random array is 0.0002 seconds, while with a size of 8192, the time is 0.0558 seconds. As figure 2 illustrates there is a similar performance on sorted, reversed, and random arrays.

Table 2: Benchmark results for the merge sort algorithm in seconds.

size	random	sorted	reversed
1.0	0.0	0.0	0.0
2.0	0.0	0.0	0.0
4.0	0.0	0.0	0.0
64.0	0.0002	0.0002	0.0002
128.0	0.0005	0.0005	0.0008
256.0	0.0011	0.0011	0.0012
2048.0	0.0121	0.0117	0.0236
4096.0	0.0263	0.025	0.0379
8192.0	0.0558	0.0531	0.057

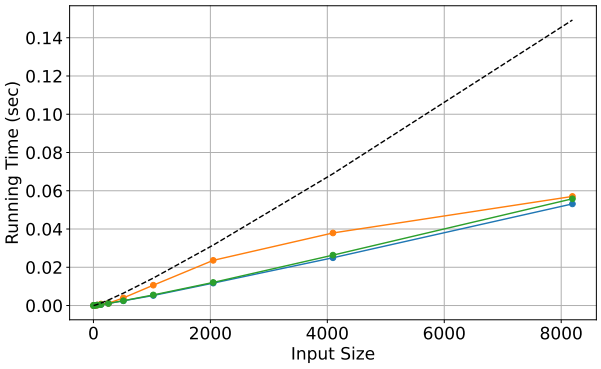

Figure 2: Benchmark results for the merge sort algorithm.

4.3 Quicksort

The benchmark summary in table 3 shows how the execution time increases with the size of the input array. A randomly sorted array is the best case scenario for the quicksort algorithm, as it only has a small increase in running time, e.g. 0.0019 seconds for 512 elements. The worst case scenario appears for the sorted array as illustrated in figure 3.

Table 3: Benchmark results for the quicksort algorithm in seconds.

size	random	sorted	reversed
1.0	0.0	0.0	0.0
2.0	0.0	0.0	0.0
4.0	0.0	0.0	0.0
16.0	0.0	0.0001	0.0001
32.0	0.0001	0.0003	0.0002
64.0	0.0001	0.0011	0.0007
128.0	0.0004	0.0043	0.0028
256.0	0.0009	0.0173	0.0108
512.0	0.0019	0.0686	0.0439

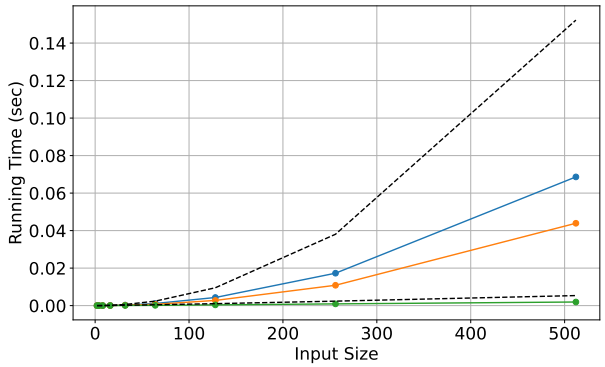

Figure 3: Benchmark results for the quicksort algorithm.

4.4 Variation in results

The plot in figure 4 illustrates the mean running time with its confidence interval. The insertion sort algorithm has a mean running time of 0.0183 seconds with a confidence interval spanning from 0.0176 seconds to 0.0189 seconds, at 512 elements. In comparison merge sort has a mean of 0.0021 seconds and a confidence interval spanning from 0.002 seconds to 0.0021 seconds, and quicksort has a mean of 0.0015 seconds and a confidence interval spanning from 0.0015 seconds to 0.0015 seconds.

At 8012 elements, insertion sort uses 5.6033 s mean running time with a confidence interval spanning from 5.3392 seconds to 5.8673 seconds, while merge sort uses 0.0507 seconds mean running time with a confidence interval spanning from 0.0485 seconds to 0.053 seconds. There are no benchmark results for quicksort at this array size.

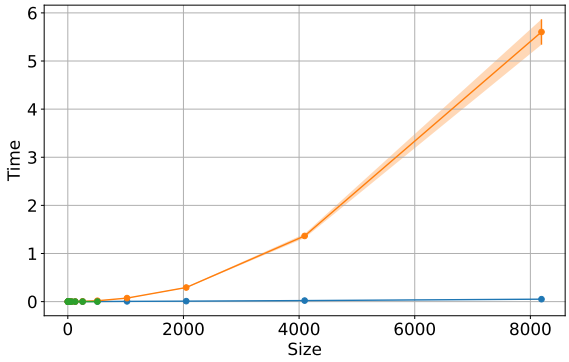

Figure 4: Variance in running time for the sorting algorithms. The running time is measured in seconds.

5 Discussion

When comparing the in-practice performance of insertion sort, merge sort and quicksort across different datasets, we find that the results align with the theoretical expectations of each algorithm.

Insertion sort has a high execution time compared with the other two algorithms, especially for large datasets as it uses 13.3908 seconds for a reversed sorted array of 8192 elements. This plots with a similar trend as its worst/average time complexity of $O(n^2)$, highlighting its inefficiency for larger, unsorted datasets. However, its performance is improved for a sorted array, taking only 0.004 seconds for an array of the same size which aligns with its best time complexity of $O(n)$.

Merge sort demonstrated consistently low execution times across all datasets, including randomly, reversed and sorted arrays. Its' even performance is consistent with its $O(nlog(n))$ complexity, and demonstrates merge sort's scalability and efficiency for large-scale sorting tasks.

Quicksort showed variability in performance. For instance, its best performance was observed with random datasets, while its efficiency decreased with sorted or nearly sorted arrays, reflecting its $O(nlog(n))$ average-case and $O(n^2)$ worst-case complexities.

We find that insertion and quicksort are sensitive to the order of the input data, while merge sort maintains a more consistent performance. For our results its is clear that merge sort out-performs insertion sort in scalability. However, we can't be sure of how quicksort would perform in comparison.

For practical applications we conclude that insertion sort is optimal for small or nearly sorted datasets, while merge sort is the most reliable for larger and diverse datasets.

References

[1] I. Adelekan. Algorithms deconstructed: Loop invariants and incremental algorithms. `https://medium.com/@iyanuadelekan/algorithms-deconstructed-loop-invariants-and-incremental-algorithms-9ffa6f5a716d`, 2017. Last accessed 19 March 2024.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Fourth edition.* The MIT

Press, Cambridge, MA, 4th edition, 2022.

[3] Programiz. Insertion sort algorithm. https://www.programiz.com/dsa/insertion-sort, n.d. Last accessed 19 March 2024.

[4] Programiz. Merge sort algorithm. https://www.programiz.com/dsa/merge-sort, n.d. Last accessed 19 March 2024.

[5] Programiz. Quicksort algorithm. https://www.programiz.com/dsa/quick-sort, n.d. Last accessed 19 March 2024.

Appendices

A Insertion sort implementation

```
""" The insertion sort algorithm based on the pseudocode from "Introduction to algorithms", 4.ed, by Cormen, Leierson, Riverst and Stein.
    (page 19)
"""
def insertionSort(A):
    # Loop over the array from the second element
    for i in range(1, len(A)):
        key = A[i]
        # Inserting A[i] into the sorted part of the array, i.e. A[0:i-1]
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key

if __name__ == "__main__":
    # Test the insertion sort function
    array = [15, 3, 9, 7, 5, 13, 20]
    print("The original array is:", array)

    insertionSort(array)
    print("The sorted array is:", array)
```

Listing 1: insertion sort algorithm

B Merge sort implementation

```
"""
Python program for implementation of MergeSort

Merges two subarrays of A[].
First subarray is A[p...q]
Second subarray is A[q+1...r]

Inspired by code contributed by Mohit Kumra, at https://www.geeksforgeeks.org/python-program-for-merge-sort/

"""
import numpy as np

def merge(A, p, q, r):
    # Calculate the sizes of the left and right halves
    n1 = q - p + 1
    n2 = r - q

    # Create temporary arrays `L` and `R` to hold the left and right halves
    L = [0] * n1
    R = [0] * n2

    # Copy the elements of the left half into L
    for i in range(n1):
        L[i] = A[p + i]

    # Copy the elements of the right half into R
    for j in range(n2):
        R[j] = A[q + j + 1]

    # Merge the sorted L and R arrays back into A
    i, j, k = 0, 0, p
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
        k += 1

    # Copy any remaining elements from `L` into `A`
    while i < n1:
        A[k] = L[i]
        i += 1
        k += 1

    # Copy any remaining elements from `R` into `A`
    while j < n2:
        A[k] = R[j]
        j += 1
        k += 1

def mergeSort(A, p, r):
    if p < r:
        q = p + (r - p) // 2

        mergeSort(A, p, q)
        mergeSort(A, q + 1, r)
        merge(A, p, q, r)


if __name__ == "__main__":
    # Test the merge sort function
    array = [15, 3, 9, 7, 5, 13, 20]
```

```
    print("The original array is:", array)

    # Pass correct indices to the mergeSort function
    mergeSort(array, 0, len(array) - 1)

    print("The sorted array is:", array)
```

Listing 2: merge sort algorithm

C Quick sort implementation

```
""" Quicksort algorithm based on pseudo-code from "Introduction to algorithms", 4.ed, by Cormen, Leierson, Riverst and Stein.
    (page 183)
"""
def quicksort(A, p, r):
    if p < r:
        # Partition subarray around pivot. Pivot ends up in position A[q]
        q = partition(A, p, r)
        quicksort(A, p, q - 1) # Recursively sort the lower-than-the-pivot side
        quicksort(A, q + 1, r) # Recursively sort the higher-than-the-pivot side


def partition(A, p, r):
    x = A[r]                        # pivot
    i = p - 1                       # highest index on the low side
    for j in range(p, r):           # process all elements, expect the pivot
        if A[j] <= x:               # move element to the low side if it belongs there, and update the highest index i
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[r] = A[r], A[i + 1] # move pivot to it's correct position, just to the left of the low side
    return i + 1



if __name__ == "__main__":
    import numpy as np
    array=[2,3,55,6,7,8, 239, 10]
    array = np.arange(998) # i can't run over 999...
    arr_copy = array.copy()  # Make a copy of the array to sort
    quicksort(arr_copy, 0, len(arr_copy) - 1)  # Sort the array
    print(arr_copy)
```

Listing 3: quicksort algorithm

D Benchmarking implementation

```python
import numpy as np
import timeit
from copy import copy
from sorting_algorithms.quicksort import quicksort
from sorting_algorithms.merge_sort import mergeSort
from sorting_algorithms.insertion_sort import insertionSort
from test_data import generate_test_data
import pandas as pd

def benchmark(sorting_algorithm):
    """
    Benchmark the sorting_algorithm by running for various array sizes,
    determined in the "generate_test_data" function.
    Also, benchmark average, best- and worst case.
    The number of executions is limited depedendent on the array size to
    prevent benchmarking from taking an unneccessary long time.
    """
    data_lengths, test_data = generate_test_data(sorting_algorithm)

    repeat = 5 # number of repetitions we will perform
    df = pd.DataFrame({"size": data_lengths}) # will store all benchmark times in this dataframe
    n = len(data_lengths)

    if sorting_algorithm.__name__ == "insertionSort":
        stmt = "sort_func(copy(data))"
    else:
        stmt = "sort_func(copy(data), 0, len(data)-1)"

    for name, data in test_data.items(): # benchmark for all random, sorted and reversed arrays
        run_times = np.empty(n) # will store running times for each iteration
        print(name)

        # perform sorting for all array sizes
        for i in range(n):
            print("Running array size", len(data[i]))

            # Set up timer with algorithm and data
            clock = timeit.Timer(stmt=stmt,
                                 globals={"sort_func": sorting_algorithm, # function to time
                                          "data": data[i], # input array to sort
                                          "copy": copy # copy function, needed to make fresh copy of array every time
                                 }
            )

            # calculate number of executions dependent on input size
            n_executions, t = clock.autorange()
            n_executions *= 3 # closer to 1 second than 0.2
            # perform timing
            time = clock.repeat(repeat=repeat,
                                number=n_executions
                                ) # number of executions, repeated 5 times

            # average value for each execution considering all the repetitions
            run_times[i] = sum(time)/(n_executions*repeat)
        df[name] = run_times # add columns for benchmarking random, sorted and reversed arrays

    df.to_pickle("results/" + sorting_algorithm.__name__ + ".pkl") # upload benchmarking to pickle file


def benchmark_with_variance(sorting_algorithm):
    """ Benchmark the sorting_algorithm by running for various array sizes,
    determined in the "generate_test_data" function. Save all times for each execution
    for each array size to get an idea of the variance in running times.
    Only the randomly ordered arrays are considered here.
```

```
    The number of executions is limited differently for the different sorting algorithms due to
    the difference in efficiency for the algorithms.
    """
    data_lengths, test_data = generate_test_data(sorting_algorithm)
    df = pd.DataFrame(columns = data_lengths) # will store all benchmark times in this dataframe
            # each column represents all times measured for one array size marked by the columnname

    if sorting_algorithm.__name__ == "insertionSort":
        stmt = "sort_func(copy(data))"
        n_executions = 30
    elif sorting_algorithm.__name__ == "mergeSort":
        stmt = "sort_func(copy(data), 0, len(data)-1)"
        n_executions = 300
    elif sorting_algorithm.__name__ == "quicksort":
        stmt = "sort_func(copy(data), 0, len(data)-1)"
        n_executions = 10000
    else:
        raise AttributeError ("The provided sorting algorithm is not defined")

    for data in test_data["random"]: # benchmark for all random, sorted and reversed arrays
        print('running array size: ', len(data))
        # Set up timer with algorithm and data
        clock = timeit.Timer(stmt=stmt,
                             globals={"sort_func": sorting_algorithm, # function to time
                                      "data": data, # input array to sort
                                      "copy": copy # copy function, needed to make fresh copy of array every time
                             }
        )

        run_times = np.empty(n_executions) # will store running times for each iteration
        # perform timing
        for i in range(n_executions):
            run_times[i] = clock.timeit(1)

        df[len(data)] = run_times # update dataframe

    df.to_pickle("results/variance_" + sorting_algorithm.__name__ + ".pkl") # upload benchmarking to pickle file


if __name__ == "__main__":
    # Decide which algorithms to benchmark
    #benchmark(insertionSort)
    #benchmark(mergeSort)
    #benchmark(quicksort)
    #benchmark_with_variance(insertionSort)
    benchmark_with_variance(mergeSort)
    #benchmark_with_variance(quicksort)

```

Listing 4: Implementation of benchmarking

E Test data generation

```python
import numpy as np

def generate_test_data(sorting_algorithm):
    rng = np.random.seed(72) #use for creating test data with a seed
    increase_factor = 2

    if sorting_algorithm.__name__ == "quicksort":
        num_lengths = 10 # up to length of 500
    else:
        num_lengths = 14 # up to length of 8000

    # initialize an array contaning the different array sizes that will be benchmarked
    data_lengths = np.zeros(num_lengths).astype("int64")
    data_lengths[0] = 1
    for i in range(1, num_lengths):
        data_lengths[i] = data_lengths[i-1]*increase_factor

    # Initialize empty arrays
    random_arrs   = np.empty((num_lengths, ), dtype=object)
    sorted_arrs   = np.empty((num_lengths, ), dtype=object)
    reversed_arrs = np.empty((num_lengths, ), dtype=object)

    # Fill arrays with random, sorted and reverse sorted values
    for idx, length in enumerate(data_lengths):
        random_arrs[idx]   = np.random.uniform(size=length)
        sorted_arrs[idx]   = np.arange(length)
        reversed_arrs[idx] = sorted_arrs[idx][::-1]

    test_data = {"random": random_arrs, "sorted": sorted_arrs, "reversed": reversed_arrs}
    return data_lengths, test_data
```

Listing 5: Generation of test data

F Creation of benchmark plots

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sorting_algorithms.quicksort import quicksort
from sorting_algorithms.merge_sort import mergeSort
from sorting_algorithms.insertion_sort import insertionSort
from matplotlib.backends.backend_pdf import PdfPages


# Read the pickled benchmark results, and return a dataframe
def read_benchmark_results(algorithm_name):
    file_path = f'results/{algorithm_name}.pkl'
    df = pd.read_pickle(file_path)
    # Ensure there are no non-positive sizes that could result in log(0)
    df = df[df['size'] > 0]
    return df

# Generate plot function
def generate_plot(df, algorithm, xlabel, ylabel, file_name):
    plt.rcParams.update({'font.size': 18})
    fig, ax = plt.subplots(figsize=(10, 6))

    # Filtering out rows where size is less than 2 to avoid log(1) which is 0
    df = df[df['size'] > 1]

    # Plot data
    df.plot(x='size', y=['sorted', 'reversed', 'random'], ax=ax, marker='o', linestyle='-', legend=False)

    # Theoretical complexities start at a size greater than 1
    sizes = df['size']
    first_size = sizes.iloc[0]
    first_random_time = df.iloc[0]['random']

    # Calculating normalization factors based on the first data point larger than 1
    norm_factor_n_log_n = first_random_time / (first_size * np.log2(first_size))
    norm_factor_n_square = first_random_time / (first_size ** 2)

    # Plot the theoretical complexities
    if algorithm in ['mergeSort', 'quicksort']:
        ax.plot(sizes, norm_factor_n_log_n * sizes * np.log2(sizes), label='O(n log n)', color='black',
        ↪  linestyle='--')
        if algorithm == 'quicksort':
            ax.plot(sizes, norm_factor_n_square * sizes ** 2, label='O(n^2) [worst]', color='black', linestyle='--')
    elif algorithm == 'insertionSort':
        ax.plot(sizes, norm_factor_n_square * sizes ** 2, label='O(n^2)', color='black', linestyle='--')
        ax.plot(sizes, first_random_time * sizes / first_size, label='O(n) [best]', color='black', linestyle='--')

    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    plt.grid(True)
    plt.savefig('figures/' + file_name, bbox_inches='tight')
    plt.show()

# Create a table in pdf
def create_pdf_table(df, algorithm_name, decimal_places=4):
    # Rounding the numerical values
    df = df.round(decimal_places)

    # Selecting some start, middle and end-rows
    start_rows = df.head(3)
    middle_rows = df.iloc[len(df)//2 - 1 : len(df)//2 + 2]
    end_rows = df.tail(3)

    # Combining these into a single dataframe
```

```
    summary_df = pd.concat([start_rows, middle_rows, end_rows])

    # Estimate figure size and creating the table
    table_height = 0.05 * len(summary_df) + 0.1
    fig, ax = plt.subplots(figsize=(8, table_height))
    ax.axis('off')
    table = ax.table(cellText=summary_df.values, colLabels=summary_df.columns, loc='center', cellLoc='center')
    table.auto_set_font_size(False)
    table.set_fontsize(10)
    table.scale(1, 1.2)

    # Styling the table
    for k, cell in table._cells.items():
        if k[0] == 0:  # Header row
            cell.set_text_props(weight='bold', color='w')
            cell.set_facecolor('#40466e')
        else:
            cell.set_facecolor('white')

    # Adjust layout and save to PDF
    pdf_file = f'figures/{algorithm_name}_benchmark_summary.pdf'
    with PdfPages(pdf_file) as pdf:
        pdf.savefig(fig, bbox_inches='tight', pad_inches=0)

    plt.close(fig)


# Main function
if __name__ == "__main__":
    # Combining the benchmark results from different algorithms
    merge_sort_df = read_benchmark_results('mergeSort')
    quicksort_df = read_benchmark_results('quicksort')
    insertion_sort_df = read_benchmark_results('insertionSort')

    # Generating and save line plots
    generate_plot(merge_sort_df, 'mergeSort', 'Input Size', 'Running Time (sec)', 'mergesort_line_plot.pdf')
    generate_plot(quicksort_df, 'quicksort', 'Input Size', 'Running Time (sec)', 'quicksort_line_plot.pdf')
    generate_plot(insertion_sort_df, 'insertionSort', 'Input Size', 'Running Time (sec)',
    ↪  'insertionsort_line_plot.pdf')

    # Creating PDF table for each algorithm
    create_pdf_table(merge_sort_df, 'Merge Sort')
    create_pdf_table(quicksort_df, 'Quick Sort')
    create_pdf_table(insertion_sort_df, 'Insertion Sort')
```

Listing 6: Generation of benchmark plots

G Creation of variation plots

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t
from matplotlib.backends.backend_pdf import PdfPages


# Read the results from pkl file and preprocess
def read_preprocess_results(algorithm_name):
    file_path = f'results/variance_{algorithm_name}.pkl'
    df = pd.read_pickle(file_path)

    # Calculating the mean execution time for each array size
    mean_times = df.mean(axis=0)

    # Calculating the standard error of the mean for each array size
    std_err = df.std(axis=0) / np.sqrt(len(df))

    # Calculating the degrees of freedom
    degrees_of_freedom = len(df) - 1

    # Calculating the confidence intervals using t-distribution
    confidence = 0.95
    confidence_intervals = std_err * t.interval(confidence, degrees_of_freedom)[1]

    # Creating a new dataframe with columns for array size, mean time, and confidence intervals
    data = pd.DataFrame({'Size': mean_times.index, 'Mean Time (sec)': mean_times.values, 'CI': confidence_intervals})

    return data

# Plot mean with the variance
def plot_variance(preprocessed_dfs, algorithm_names):
    plt.figure(figsize=(10, 6))
    plt.rcParams.update({'font.size': 18})

    # Plotting each algorithm's variance
    for preprocessed_df, algorithm_name in zip(preprocessed_dfs, algorithm_names):
        plt.errorbar(preprocessed_df['Size'], preprocessed_df['Mean Time (sec)'], yerr=preprocessed_df['CI'],
                     fmt='-o', label=algorithm_name)
        plt.fill_between(preprocessed_df['Size'], preprocessed_df['Mean Time (sec)'] - preprocessed_df['CI'],
                         preprocessed_df['Mean Time (sec)'] + preprocessed_df['CI'], alpha=0.3)

    plt.xlabel('Size')
    plt.ylabel('Time')
    plt.rcParams.update({'font.size': 18})
    plt.grid(True)
    plt.savefig('figures/variance_plot.pdf')
    plt.show()

def add_ci_columns(df):
    df['CI Lower'] = df['Mean Time (sec)'] - df['CI']
    df['CI Higher'] = df['Mean Time (sec)'] + df['CI']
    return df

def create_pdf_table_per_algorithm(df, algorithm_name, decimal_places=4):
    # Rounding the numerical values
    df_rounded = df.round(decimal_places)

    # Create and style the table
    fig, ax = plt.subplots(figsize=(8, 0.5 * len(df_rounded)))
    ax.axis('off')
    table = ax.table(cellText=df_rounded.values, colLabels=df_rounded.columns, loc='center', cellLoc='center')
    table.auto_set_font_size(False)
    table.set_fontsize(10)
```

```
    table.scale(1, 1.5)

    # Styling the table
    for k, cell in table._cells.items():
        cell.set_edgecolor('black')
        if k[0] == 0 or k[1] < 0:
            cell.set_text_props(weight='bold', color='white')
            cell.set_facecolor('#40466e')
        else:
            cell.set_facecolor('white')

    # Save the table as a PDF
    pdf_file = f'figures/{algorithm_name}_performance_summary.pdf'
    plt.savefig(pdf_file, bbox_inches='tight', pad_inches=0.1)
    plt.close()


if __name__ == "__main__":
    # Combining the benchmark results from different algorithms
    merge_sort_df = read_preprocess_results('mergeSort')
    quicksort_df = read_preprocess_results('quicksort')
    insertion_sort_df = read_preprocess_results('insertionSort')

    # Plot variance for each algorithm
    plot_variance([merge_sort_df, insertion_sort_df, quicksort_df], ['Merge Sort', 'Insertion Sort', 'Quick Sort'])

    for algorithm in ['mergeSort', 'quicksort', 'insertionSort']:
        df = read_preprocess_results(algorithm)
        df_with_ci = add_ci_columns(df)
        create_pdf_table_per_algorithm(df_with_ci, algorithm)

```

Listing 7: Generation of variation plot

H Requirements

```
contourpy==1.2.0
cycler==0.12.1
fonttools==4.50.0
kiwisolver==1.4.5
matplotlib==3.8.3
numpy==1.26.4
packaging==24.0
pandas==2.2.1
pillow==10.2.0
pyparsing==3.1.2
python-dateutil==2.9.0.post0
pytz==2024.1
scipy==1.12.0
setuptools==68.2.2
six==1.16.0
tzdata==2024.1
wheel==0.41.2
```

Listing 8: Requirements