CS2001 W10 Coursework

220019540

Tutor: Abudureyimu Halite

15/11/2023

Overview

In this practical, I was tasked with creating an application in which the user can compare the complexity and performance of 2 different sorting algorithms - Merge Sort and Selection Sort.

I will be doing asymptotic analysis on all of my algorithms. Asymptotic analysis involves evaluating the performance of an algorithm as the input size approaches infinity. Instead of focusing on exact running times, which can vary depending on the specific hardware and other factors, asymptotic analysis provides a broader understanding of how the algorithm scales with larger inputs. For this I will be looking at how the run time compares to the size of the data structure I provide to the sort algorithm.

Design

I wanted to create an application that could automate tests and output reliable data that could instantly be used in a graph. For this, I decided to split my program into - different parts:

- I created a properties file which will hold all the arrays and data that would be tested.
- I created 2 class for the merge sort and selection sort, of which their purpose was to run the algorithm and return a sorted array
- A test class which would run all the algorithms on the varying data sizes, display their statistics, and output the data to a csv file where it can be recorded.

This way, I created an application which made adding test data and getting the output in a clean form very straight-forward.

Test Data

One of the main priorities of my test was to ensure that my data was created equally and fairly, so that neither of the sorting algorithms had an advantage over the other one. This way, I can ensure that my data is reliable and is close to the true complexity of the algorithms in any scenario.

I wrote my configuration class so that it could read all the data from the file and store it in a map of lists. This way, I can create a clean test class which would need the key in order to find the appropriate data structure used to test the algorithm.

I decided to test the following sizes of lists: 10, 15 (for functionality testing), 25, 35, 50, 100, 250, 500, 1000, 10 000.

The reason for this is because I wanted to create a graph where the x axis increased logarithmically, allowing me to see a large variety of list sizes.

When Testing my program, I decided it was more accurate to measure in nanoseconds, then convert the times back to milliseconds when building the graph of my data. I took this approach as I wanted to look more closely at the time differences between various list sizes.

Graph Design

I used an external library (Jupyter Notebook) and used Python to create a graph of my test results. I used the data from my CSV file in the python script to generate the graph. Here is my code for generating the graph (also found in the graphdata directory in the project).

```
import matplotlib.pyplot as plt

# Data for the first algorithm (Merge Sort)
sizes_merge_sort = [10, 25, 35, 50, 100, 250, 500, 1000]
runtime_merge_sort_ns = [110330, 106385, 76205, 31085, 83150, 125622, 265235,
469120]
runtime_merge_sort_ms = [time_ns / 1000000 for time_ns in
runtime_merge_sort_ns]

# Data for the second algorithm (Selection Sort)
sizes_selection_sort = [10, 25, 35, 50, 100, 250, 500, 1000]
runtime_selection_sort_ns = [45149, 43992, 87875, 140571, 631220, 1220585,
1890210, 2198650]
runtime_selection_sort_ms = [time_ns / 1000000 for time_ns in
runtime_selection_sort_ns]

# Plotting the data with a logarithmic x-axis
plt.plot(sizes_merge_sort, runtime_merge_sort_ms, marker='o', label='Merge
Sort')
plt.plot(sizes_selection_sort, runtime_selection_sort_ms, marker='o',
label='Selection Sort')

# Adding labels and title
plt.xscale('log')  # Set x-axis to logarithmic scale
plt.xlabel('List Size (log scale)')
plt.ylabel('Runtime (ms)')
plt.title('Runtime Comparison of Merge Sort and Selection Sort')
plt.legend()

# Display the plot
```

```
plt.show()
```

(Please note that the code above does not include the result for the list size 10000 for reasons mentioned in the Testing section. The full code is found in the project directory graphdata.)

Implementation

To make sure that all of my tests ran accordingly and were fair, I wrote python scripts to generate sequences of numbers that I would include within my properties file. Below is an example of a python script written to generate a sequence of random numbers from 1 - 100 (also found in the graphdata directory of the project:

```
import random
# Number generator code to create sequences for testing
# By 220019540

# Generate a list of numbers from 1 to 100
numbers = list(range(1, 101))

# Shuffle the list to make it unsorted
random.shuffle(numbers)

# Convert the list to a string in the desired format
unsortedArray100 = ",".join(map(str, numbers))

# Print the result
print(f"unsortedArray100 = {unsortedArray100}")
```

Testing

Before I could start running runtime tests on my algorithms, I had to ensure that my sorting algorithms worked correctly. For this, I created 3 functionality tests to check that my algorithm can sort any sequence of number. I also added an exceptional test to ensure that my program would not throw fatal errors during runtime.

Here is the code for my functionality tests. I added a functionalityTesting flag so I could see the results of my testing in the terminal:

```
//set the functionality flag to true to avoid writing to the csv file
functionalityTest = true;
test(m, s, "functionalityArrayRandomised");
test(m, s, "functionalityArrayReversed");
test(m, s, "functionalityArrayOrdered");
functionalityTest = false;
```

Here are the results of the functionality testing:

```
Array of size 15
Initial Array: [13, 11, 10, 8, 12, 6, 9, 7, 3, 14, 5, 4, 1, 2, 15]
Merge Sort time elapsed: 2549 nanoseconds (on average).
Selection Sort time elapsed: 833 nanoseconds (on average).
Merge-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Selection-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
**********
Array of size 15
Initial Array: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Merge Sort time elapsed: 1717 nanoseconds (on average).
Selection Sort time elapsed: 1872 nanoseconds (on average).
Merge-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Selection-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
**********
Array of size 15
Initial Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Merge Sort time elapsed: 1764 nanoseconds (on average).
Selection Sort time elapsed: 750 nanoseconds (on average).
Merge-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Selection-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
-----
```

As shown, the sorting algorithms work as they are supposed to.

Here is the test for when the program receives an empty array:

```
-----
Exception case testing
**********
Array of size 0
Merge Sort time elapsed: 14477 nanoseconds (on average).
Selection Sort time elapsed: 7983 nanoseconds (on average).
Skipping CSV write for exceptional case.
```

After I was sure that my algorithms and code worked correctly, I ran my final test to gather data for the graph.

To ensure that my test results were reliable and fair, I ran each runtime test 5 times and took the average value of each to be used as a final test result. This way I can avoid large skews in my test results due to variables out of my control.

Here is the final output of all my tests:

```
aw356@lyrane:~/.../cs2001/Coursework/W10/src $ java TestAlgorithms
CSV file cleared and headers written successfully!
```

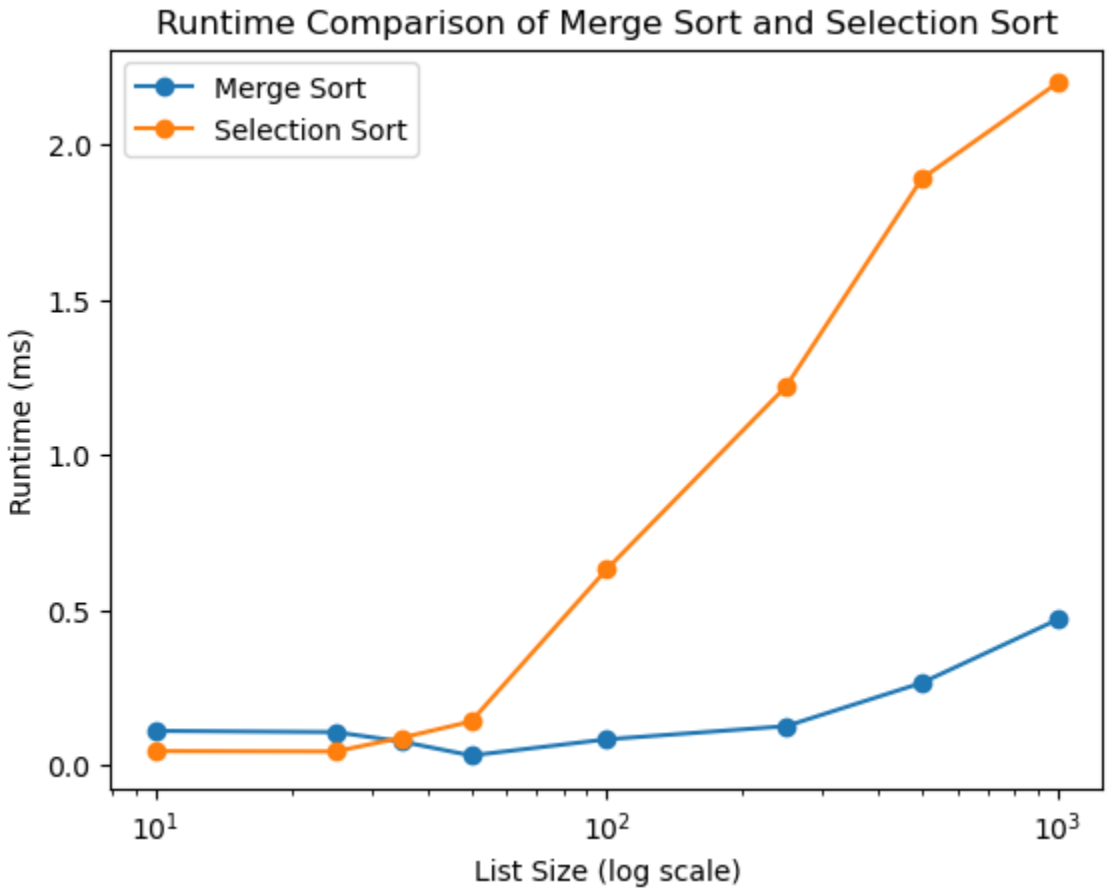
```
**********
Array of size 10
Merge Sort time elapsed: 110330 nanoseconds (on average).
Selection Sort time elapsed: 45149 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 25
Merge Sort time elapsed: 106385 nanoseconds (on average).
Selection Sort time elapsed: 43992 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 35
Merge Sort time elapsed: 76205 nanoseconds (on average).
Selection Sort time elapsed: 87875 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 50
Merge Sort time elapsed: 31085 nanoseconds (on average).
Selection Sort time elapsed: 140571 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 100
Merge Sort time elapsed: 83150 nanoseconds (on average).
Selection Sort time elapsed: 631220 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 250
Merge Sort time elapsed: 125622 nanoseconds (on average).
Selection Sort time elapsed: 1220585 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 500
Merge Sort time elapsed: 265235 nanoseconds (on average).
Selection Sort time elapsed: 1890210 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 1000
Merge Sort time elapsed: 469120 nanoseconds (on average).
Selection Sort time elapsed: 2198650 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 10000
Merge Sort time elapsed: 2659841 nanoseconds (on average).
Selection Sort time elapsed: 66807764 nanoseconds (on average).
Data appended to CSV successfully!
**********
Array of size 15
Initial Array: [13, 11, 10, 8, 12, 6, 9, 7, 3, 14, 5, 4, 1, 2, 15]
```

```
Merge Sort time elapsed: 2549 nanoseconds (on average).
Selection Sort time elapsed: 833 nanoseconds (on average).
Merge-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Selection-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
**********
Array of size 15
Initial Array: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Merge Sort time elapsed: 1717 nanoseconds (on average).
Selection Sort time elapsed: 1872 nanoseconds (on average).
Merge-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Selection-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
**********
Array of size 15
Initial Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Merge Sort time elapsed: 1764 nanoseconds (on average).
Selection Sort time elapsed: 750 nanoseconds (on average).
Merge-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Selection-Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
-----
Exception case testing
**********
Array of size 0
Merge Sort time elapsed: 14477 nanoseconds (on average).
Selection Sort time elapsed: 7983 nanoseconds (on average).
Skipping CSV write for exceptional case.
```

Here is the contents of the CSV file which stores the test result data:

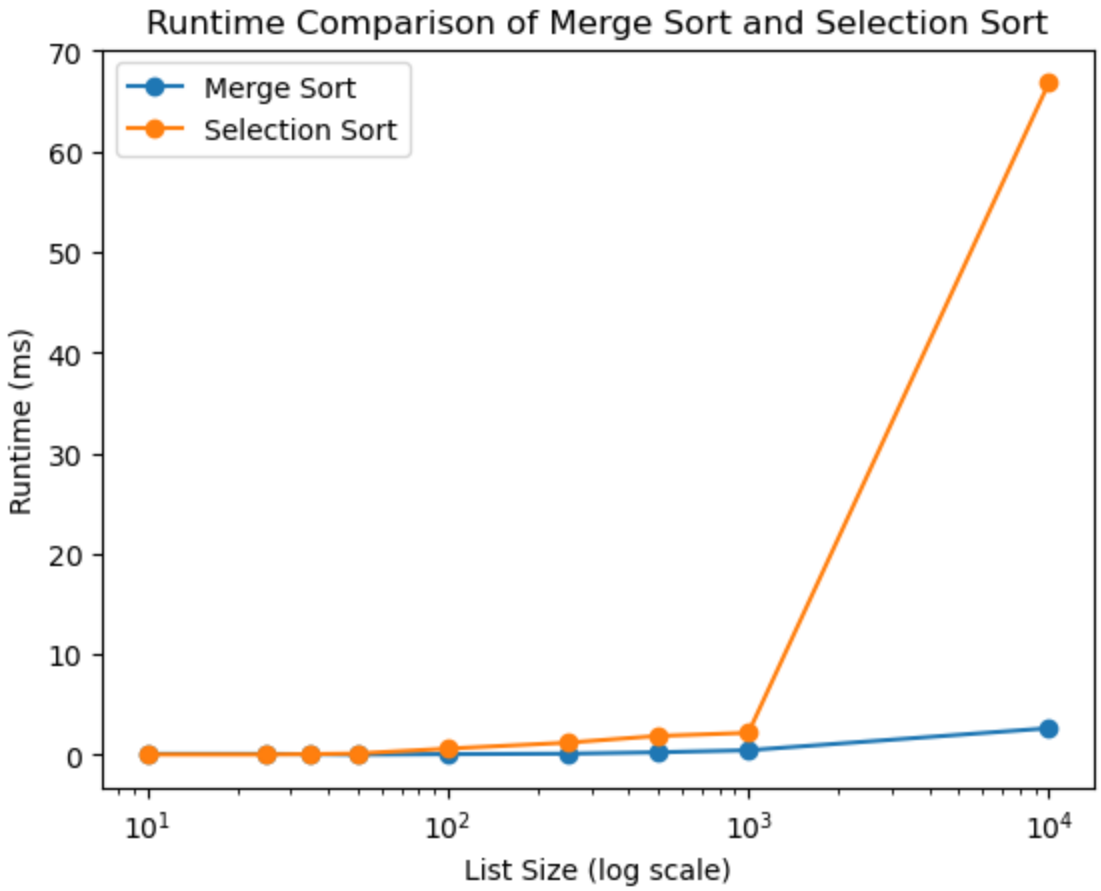
```
List Size, Merge Sort Runtime, Selection Sort Runtime
10,110330,45149
25,106385,43992
35,76205,87875
50,31085,140571
100,83150,631220
250,125622,1220585
500,265235,1890210
1000,469120,2198650
10000,2659841,66807764
```

Here is the final graph of my test results:



After testing my program with different list sizes, I found that both of the algorithms ‘cross-over’ in terms of complexity when sorting a list of size 35.

For the sake of being able to see both algorithms clearly, I decided to omit the size 100000 list test in the graph above. Here is a graph which also includes the results for the list of size 100000:



Because the disparity between the 2 algorithms is so high at the extreme list sizes, I decided to use my earlier graph to look closely at the results. The python script for this graph can be found in the graphdata directory in my project.

Analysis - Complexity

What does this say about the two algorithms?

Initially, for smaller datasets like those with 10 elements, the runtimes for both algorithms appear relatively close, with selection sorts seeming more efficient. However, as the list sizes expand, the differences in the algorithms become more pronounced. Merge sort, known for its efficient divide-and-conquer approach, maintains a consistent O(n log n) time complexity, resulting in a more gradual increase in runtime as the list size grows. On the other hand, selection sort, characterized by its quadratic O(n^2) time complexity, experiences a more rapid and exponential rise in runtime, leading to a large disparity in efficiency.

The crossover point at around 35 elements represents the point where the superior time complexity of merge sort begins to outweigh selection sort. Beyond this point, the difference in performance becomes more evident, with merge sort showcasing how it scales better with larger lists. The algorithm achieves this by recursively dividing the input list into smaller sublists, sorting them individually, and then merging them in a sorted manner. This results in a consistent and efficient sorting process across various list sizes.

In contrast, selection sort, a simple and intuitive sorting algorithm, repeatedly selects the minimum element from the unsorted portion of the list and swaps it with the first unsorted element. While effective for small datasets, its time complexity becomes a limiting factor as the input size increases. The quadratic nature of selection sort makes it less suitable for handling larger datasets, as reflected in the considerable runtime differences observed in the dataset, particularly notable in the case of a list with 10,000 elements.

Evaluation

I believe that my program has managed to achieve a good analysis of the two sorting algorithms. I have created a straight-forward application, with features that allow potential users to add their own pieces of test data and run customised tests on the algorithm by themselves. I represented my test results in a clean graph and I was able to highlight the key differences between both algorithms based on my test results. My program includes repeatability of all the tests, and creates reliable results which are saved to an external file for recording everything.

Conclusion