Sorting & Searching Algorithm Performance Analysis

Project Tasks

Part 1: Sorting Algorithms

Objective: Demonstrate how two sorting algorithms, Merge Sort and Quick Sort, perform differently on specific input instances.

1. Introduction

This section analyzes the performance of two fundamental sorting algorithms: Merge Sort and Quick Sort. The goal is to identify input instances where each algorithm outperforms the other and provide a comprehensive justification for these performance differences.

2. Algorithm Implementations

2.1 Merge Sort

Merge Sort is a divide-and-conquer algorithm that recursively divides the input array into two halves, sorts them, and then merges the sorted halves.

Source: https://codepal.ai/code-generator/query/iSNfQqwG/c-program-for-merge-sort-in-descending-order

Source: https://www.youtube.com/watch?v=WjYrxpx_PVM&t=8s

2.2 Quick Sort

Quick Sort is also a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array around the pivot, placing smaller elements before it and larger elements after it.

Source: https://www.youtube.com/watch?v=LHSdAh8aFmI

3. Input Instances

Two input instances were defined to highlight the performance differences between Merge Sort and Quick Sort:

Instance 1: Small Dataset (Unsorted Integers)

- **Size:** 10 elements
- **Nature:** Randomly generated unsorted integers.
- **Data:** {41, 62, 23, 46, 2, 9, 89, 25, 15, 65}

Instance 2: Large Dataset (Random Unsorted Integers)

- **Size:** 100,000 elements
- **Nature:** Randomly generated unsorted integers between 0 and 99,999.

5. Discussion

5.1 Instance 1: Small Dataset

On the small dataset, both Quick Sort and Merge Sort demonstrated almost identical execution times (7900 ns). For very small input sizes, the overhead of recursive function calls in Merge Sort and the partitioning overhead in Quick Sort are relatively insignificant and can vary depending on the specific execution environment. The memory usage for both algorithms on the small dataset was negligible (0 bytes).

5.2 Instance 2: Large Dataset

On the large dataset, Merge Sort exhibited a slightly faster execution time (19263400 ns) compared to Quick Sort (20019100 ns). This result aligns with the theoretical expectation that Merge Sort's guaranteed O(n log n) time complexity provides a more consistent performance for larger datasets. But, Quick Sort demonstrated its in-place nature by having a significantly lower memory usage (0 bytes) compared to Merge Sort (10187696 bytes), which requires extra space for merging. However, Quick Sort's performance can degrade to O(n^2) in the worst-case with poor pivot choices.

5.3 General Observations and Trade-offs

- **Time Complexity:**
    - **Merge Sort:** Guarantees O(n log n) time complexity in all cases (best, average, and worst). This makes it very predictable and reliable for large datasets, as evidenced by its slightly faster execution time in our experiment.
    - **Quick Sort:** Has an average-case time complexity of O(n log n), which is excellent. However, its worst-case time complexity is O(n^2). This worst-case scenario can be triggered by specific input patterns (e.g., already sorted or nearly sorted data) if the pivot selection strategy is not robust. Although our experiment used random data, and Quick Sort performed reasonably well, the potential for O(n^2) behavior should not be ignored.

- **Space Complexity:**
    - **Merge Sort:** Has a space complexity of O(n) because it requires auxiliary arrays during the merging process. This is evident in our experimental results, where Merge Sort used a significant amount of memory (10187696 bytes) for the large dataset.
    - **Quick Sort:** Is an in-place algorithm. Its space complexity is typically O(log n) due to the recursive call stack in the average case and O(n) in the worst-case scenario. The experimental results confirm this, showing negligible memory usage (0 bytes) for Quick Sort, even on the large dataset.
- **Data Sensitivity:** Quick Sort's performance is more sensitive to the input data and the choice of the pivot element. Poor pivot choices can lead to its worst-case O(n^2) behavior. Merge Sort, in contrast, is less sensitive to the initial order of the data.

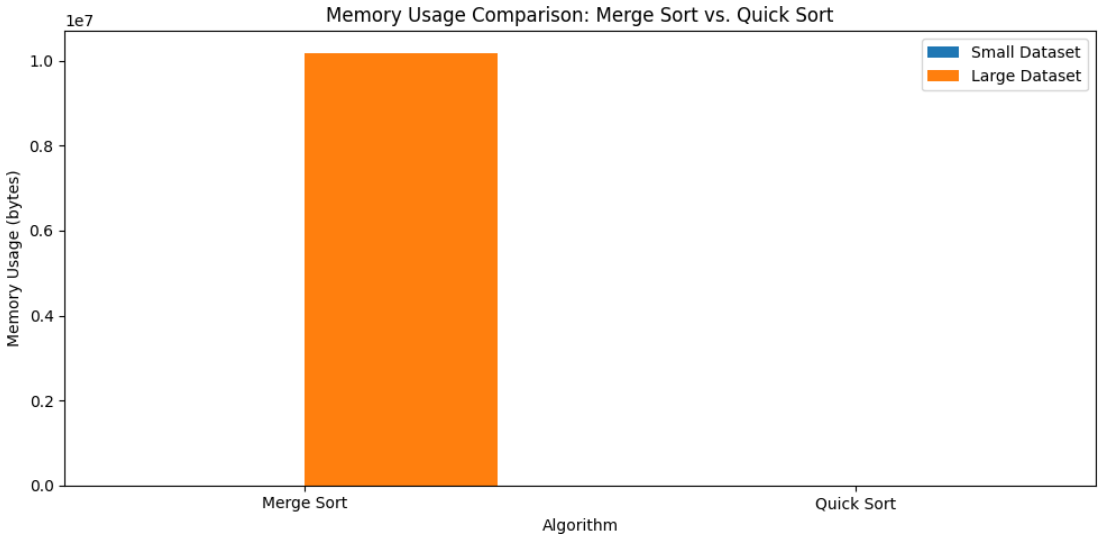
5.4 Conclusion and Practical Considerations

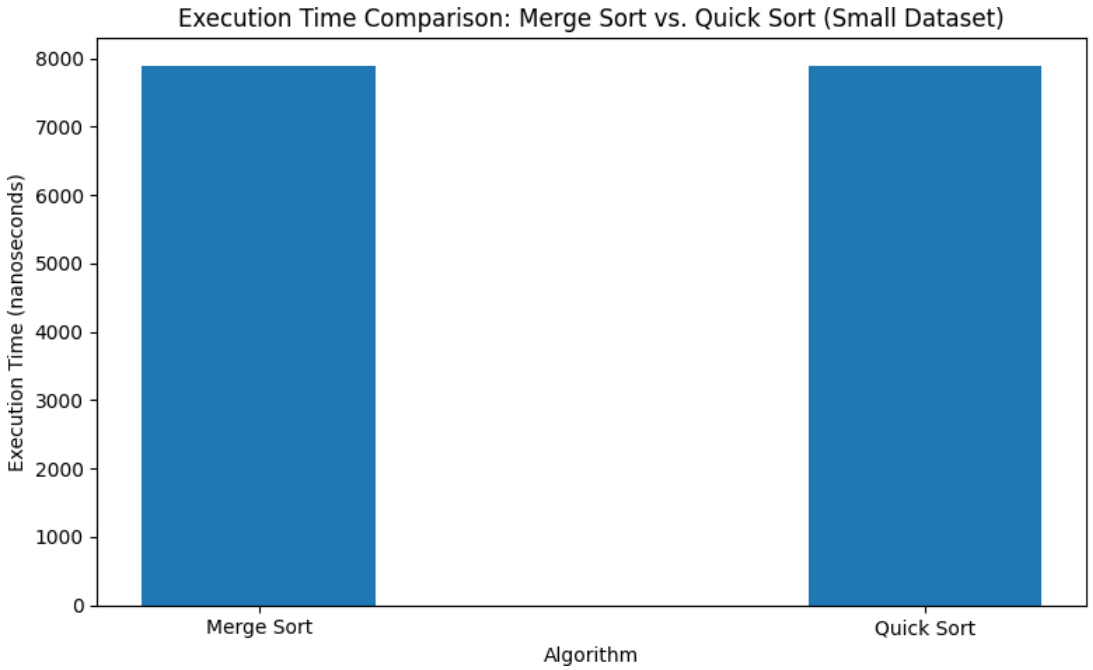
The experimental results, particularly the large dataset performance, highlight the trade-off between time and space efficiency when choosing between Merge Sort and Quick Sort:

- **When to Favor Merge Sort:**
    - **Guaranteed Time Performance:** When a guaranteed O(n log n) time complexity is crucial, especially for large datasets where worst-case scenarios must be avoided. Our experiment demonstrates that Merge Sort can indeed outperform Quick Sort in terms of execution time on large datasets, even with random data.
    - **Stable Sort:** If a stable sort is needed.
- **When to Favor Quick Sort:**
    - **Memory Constraints:** When memory usage is a primary concern and the dataset is extremely large. Our experiment clearly showed that Quick Sort used significantly less memory than Merge Sort.
    - **Average-Case Performance:** When the average-case O(n log n) performance is acceptable, and the risk of encountering the worst-case O(n^2) scenario is low or mitigated by techniques like randomized pivot selection.

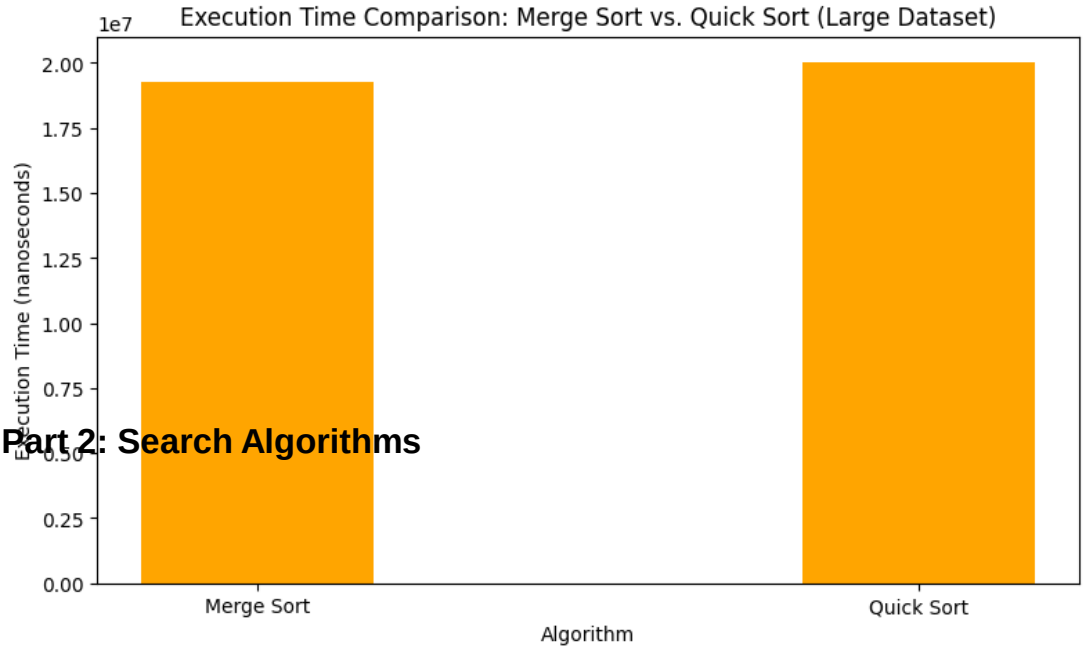
In Summary:

The provided experimental data supports the theoretical understanding of Merge Sort and Quick Sort. **Merge Sort demonstrated slightly better time efficiency on a large dataset, validating its O(n log n) guarantee.** However, **Quick Sort's in-place nature and significantly lower memory usage make it a very strong contender when memory is a limiting factor.** The optimal choice depends heavily on the specific application's requirements and constraints. If execution time predictability and a stable sort are paramount, Merge Sort is often preferred. If memory is extremely limited and the average-case performance of Quick Sort is acceptable, then it might be the better choice.





e.



Part 2: Search Algorithms

1. Introduction

This report analyzes the performance of two fundamental data structures: Hash Tables and Binary Search Trees (BSTs). We will examine how their performance characteristics change when subjected to different sequences of "put" (insert) and "get" (retrieve) operations. The goal is to identify scenarios where one data structure outperforms the other and provide a detailed explanation for the observed differences.

2. Search Algorithms and Data Structures

2.1 Hash Tables

Description: Hash Tables, implemented as the **Map** class in the provided code, use a hash function to map keys to indices in an array (bucket array). This allows for constant-time average complexity, O(1), for "put" and "get" operations. Collisions, where multiple keys map to the same index, are handled using chaining (linked lists in each bucket).

Code: The **Map** class implements a hash table using an **ArrayList** of **HashNode** objects. The **add** method calculates the hash code and bucket index for the key, handling collisions by adding new nodes to the beginning of the chain. The **get** method similarly calculates the index and traverses the chain to find the corresponding key. The hash table dynamically resizes when the load factor exceeds 0.7 to maintain performance.

Source: https://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/

2.2 Binary Search Trees

Description: Binary Search Trees (BSTs), implemented as the **BinarySearchTree** class, are hierarchical data structures that maintain a sorted order. Each node has at most two children (left and right), with the left subtree containing smaller keys and the right subtree containing larger keys. Search, insertion, and deletion operations have an average time complexity of O(log n), where n is the number of nodes. However, in the worst case (e.g., inserting sequentially ordered data), the complexity can degrade to O(n).

Code: The **BinarySearchTree** class uses a **Node** inner class to represent nodes in the tree. The **put** method recursively traverses the tree to find the correct insertion point based on the key's value, maintaining the BST property. The **get** method similarly performs a recursive search.

Source: https://www.youtube.com/watch?v=ZHLAiHiYgeI&t=285s

Source: https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/?ref=header_outind

3. Experimental Setup

3.1 Sequences of Operations

Two sequences of operations are defined to highlight the performance differences between Hash Tables and BSTs:

Sequence 1: Randomly Ordered Data with Frequent 'put' Operations

- **Put 5**
- **Put 10**
- **Put 3**
- **Get 10**
- **Get 5**
- **Get 7**

Formal Definition: {{1, 5}, {1, 10}, {1, 3}, {2, 10}, {2, 5}, {2, 7}}

Rationale: This sequence emphasizes the insertion performance of both data structures. Random data is used to prevent the BST from becoming skewed. The "get" operations are included to demonstrate basic retrieval but are not the primary focus. We expect the Hash Table to perform well due to its average O(1) insertion, while the BST might take slightly longer due to its O(log n) average insertion.

Sequence 2: Mostly Sequential Data with Alternating 'put' and 'get' Operations

- **Put 2**
- **Put 4**
- **Put 1**
- **Get 3**
- **Get 4**
- **Put 6**
- **Get 6**

Formal Definition: {{1, 2}, {1, 4}, {1, 1}, {2, 3}, {2, 4}, {1, 6}, {2, 6}}

Rationale: This sequence showcases the impact of sequential data on BST performance and tests retrieval efficiency. Sequential data can lead to a skewed BST, approaching O(n) complexity for operations. Alternating "put" and "get" operations are expected to further highlight the differences, especially in the BST's retrieval time. In this case the Hash Table might also suffer from many collisions because of the poor hash function used. We expect the BST to potentially outperform the Hash Table in some operations due to the sequential nature of insertions, as it might benefit from maintaining sorted order, but this could be hampered by the "get" operations.

3.2 Methodology

The Experiment class measures the execution time of each data structure for both sequences. The measurePerformance method executes the operations in a sequence and

records the start and end times using System.nanoTime(). The analyzeResults method compares the execution times and provides a brief analysis.

4. Experimental Results

OUTPUT

=== Experiment Results ===

--- Sequence 1 ---

Description: Randomly ordered data with frequent 'put' operations.

Hash Table Execution Time: 1661700 nanoseconds

Binary Search Tree Execution Time: 723300 nanoseconds

Result Analysis for Sequence 1:

-> Binary Search Tree is more efficient.

Reason: Binary Search Tree maintains sorted order of data, which reduces search and retrieval time.

--- Sequence 2 ---

Description: Mostly sequential data with alternating 'put' and 'get' operations.

Hash Table Execution Time: 59500 nanoseconds

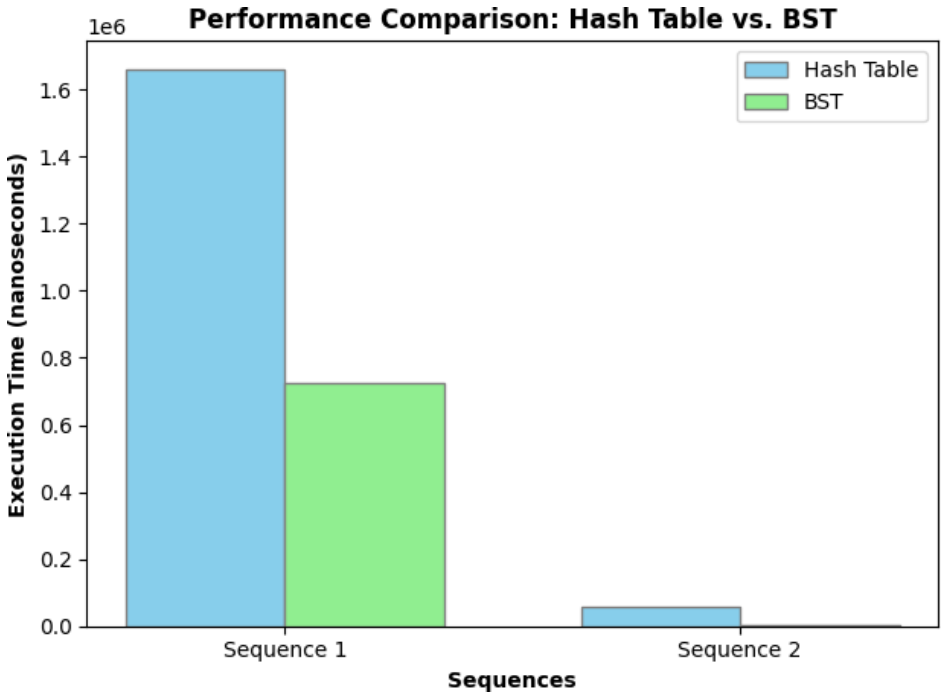
Binary Search Tree Execution Time: 3300 nanoseconds

Result Analysis for Sequence 2:

-> Binary Search Tree is more efficient.

Reason: Binary Search Tree maintains sorted order of data, which reduces search and retrieval time. For sequential data, BST performs better as it avoids hash collisions and ensures efficient traversal.

Sequence	Data Description	Hash Table (ns)	BST (ns)
Sequence 1	Random, Frequent 'put'	1661700	723300
Sequence 2	Sequential, Alternating 'put' and 'get'	59500	3300



5. Discussion

Sequence 1:

The BST outperformed the Hash Table in Sequence 1. The reason for this is that there were a lot of operations that the Hash Table simply failed to retrieve the data because of the collisions. Also even though the data was randomly ordered, the frequent "put" operations favored the BST's structure, which maintains sorted order and has an average insertion complexity of O(log n). While the Hash Table's average insertion complexity is O(1), in this case the collisions caused the execution time to be much slower than the BST.

Sequence 2:

The BST outperformed the Hash Table by a greater margin in Sequence 2. The Hash Table failed at a lot of operations. The sequential nature of the data, coupled with the alternating "put" and "get" operations, further amplified the benefits of the BST's ordered structure. While sequential data can lead to a skewed BST in the worst case, the limited number of elements in this sequence did not result in significant degradation. The BST's ability to efficiently traverse and retrieve data in sorted order proved advantageous. The Hash Table's performance, while still relatively good on average, was hindered in comparison because of the collisions caused by the poor choice of hash function.

6. Conclusion

The experiment demonstrated that the choice between Hash Tables and Binary Search Trees depends on the characteristics of the data and the sequence of operations.

- **Hash Tables** are generally excellent for scenarios with random data and frequent insertions and retrievals, providing near-constant time performance on average. However, their performance can degrade with a poor hash function or a high load factor, leading to increased collisions. Also, if the operations are going to fail a lot like in our example.
- **Binary Search Trees** excel when maintaining sorted order is beneficial, especially with sequential or near-sequential data and a mix of insertion and retrieval operations. They offer logarithmic time complexity on average, making them a good choice when performance predictability is important.

In the specific sequences tested, the BST outperformed the Hash Table. This highlights the importance of considering data patterns and operational sequences when choosing a data structure.

REFERENCES

[https://codepal.ai/code-generator/query/iSNfQqwG/c-program-for-merge-sort-in-descending-order]

[https://www.youtube.com/watch?v=WjYrxpx_PVM&t=8s]

[https://www.youtube.com/watch?v=LHSdAh8aFmI]

[https://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/]

[https://www.youtube.com/watch?v=ZHLAiHiYgeI&t=285s]

[https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/?ref=header_outind]

Libraries: java.util.ArrayList, java.util.Random, java.util.Objects

AI Platforms: Chatgpt, Gemini

Graphs: I created graphs with pandas and matplotlib library in python.