A blue dots on a black background

Description automatically generated

**Lab Algorithms and functional programming**

**Rakesh Meenakshi Sundaram**

**Project GitHub Repository:** [**https://github.com/Newt93/Algorithm-App.git**](https://github.com/Newt93/Algorithm-App.git)

**Autumn semester 2024**

**Course Coordinator: Abdulghafour Mohammad**

**Introduction**

This report covers the development of a console application to explore sorting, searching algorithms and C# filter, map, reduce for reading employee information through files and performing operations. The application allows users to input an array size, select specific implementation to run, and view the results, including the time taken by each algorithm and the output results.

The project includes sorting algorithms like Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort, as well as searching methods like Linear Search and Binary Search. Lambda expressions were also used for concise operations, and dictionaries were implemented for their efficiency during switch case handling during sorting algorithms

The goal of this assignment was to understand how different algorithms and data structures perform in various scenarios, helping to choose the right tools for solving problems in programming. This report explains the implementation, results, and key learnings from the project.

I have completed my assignment till A grade below I have a cumulative report for my assignment

The below screen short represents the sample output of the console application implemented or assignment

A black screen with a black background

Description automatically generated

**Report For D**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | N=100 | N=5000 | N=100,000 | N=1000,000 |
| Insertion Sort | 0.1334 | 37.0423 | 10396.1371 | 103,961,371.0 (28 hours) |
| Selection sort | 0.2572 | 59.6301 | 16251.6097 | 162,516,097.0 (45 hours) |
| Bubble Sort | 0.1498 | 214.6157 | 93293.7911 | 932,937,911.0 (259 hours) |
| Merge sort | 0.3131 | 2.1895 | 44.5418 | 6,235.85 |
| Quick sort | 0.2088 | 1.6546 | 40.1032 | 5,614.45 |
| Using Lambda | 0.3608 | 1.2616 | 23.9223 | 2062.9607 |

Units are measured in milliseconds

Since I couldn’t be able to run the last value(n=1000,0000) for insertion, selection and bubble sort, I used ChatGPT to generate me results based on the formula as below

A screenshot of a math problem

Description automatically generated

A math equations on a white background

Description automatically generated

**1, Which algorithm wins (the best one)?**

The best algorithm is determined based on its performance. From the above table, **Using Lambda** performs faster due to its strong and efficient internal sorting methods plus the comparison delegate which takes two elements from array and compares the value.

**2, Reflect on the results.**

My understanding about the sorting algorithms

**1. Bubble Sort**

* + Repeatedly compares adjacent elements in the array and swaps them if they are in the wrong order.
  + This process is repeated from the beginning until the array is fully sorted.

**Notes:**

* + Very slow compared to modern sorting algorithms.

**2. Selection Sort**

* + Goes through the entire array and finds out the smallest element and moves it to the first cell and finds the second smallest element and moves it to the second cell.
  + Repeats the process for the rest of the unsorted elements until everything is sorted.

**Notes:**

* + Inefficient for large datasets.

**3. Insertion Sort**

* + Mimics the way we sort playing cards.
  + Starts with the first element (treated as sorted).
  + Picks the next element and inserts it into its correct position in the sorted part.
  + Repeats this process for all elements.

**Notes:**

* + Works well for small or nearly sorted datasets.
  + Not suitable for large datasets.

**4. Merge Sort**

**Concept:**

* + Divides the array into halves recursively until each subarray has one element.
  + Merges the subarrays back together in sorted order.
  + Utilizes a "divide and conquer" approach.

**Notes:**

* + Stable and efficient for large datasets.
  + Requires additional memory for merging.

**5. Quick Sort**

**Concept:**

* + Selects a pivot element.
  + Partitions the array such that elements smaller than the pivot go to the left and elements larger go to the right.
  + Recursively applies the same logic to the left and right subarrays.

**Notes:**

* + Very efficient and widely used in practice.
  + Can be optimized using techniques like randomizing the pivot.

**Reflections based on the results**:

* **Insertion Sort, Selection Sort, Bubble Sort** are **O(N²)** algorithms, making them not suitable for large arrays. Their running times grows exponentially as the input size increases.
* **Merge Sort and Quick Sort** are **O(N log N)** algorithms, which are much more efficient for larger datasets.
* **Using Lambda** leverages the efficient internal sort implementation (Array.Sort or LINQ OrderBy), which also has **O(N log N)** performance.

**Scalability**:  
For small input sizes (N=100), all algorithms perform reasonably well, with small number of differences. However, for large datasets (N=5000 and above), the limitations of simpler algorithms like Bubble Sort and Selection Sort become evident as their running times grow significantly.

**Practical Implications**:  
While algorithms like Quick Sort and Merge Sort are complex to implement, they are better suited for real-world scenarios with large data sets. Bubble Sort and Selection Sort, on the other hand, are mostly used for teaching or cases where simplicity is prioritized over efficiency.

**3, Reflect on the benefits of avoiding mutation and using the delegate (Functions as first-class values ).**

**Avoiding Mutation**

* Sorting without mutating the original array ensures data integrity and reduces side effects. This important in functional programming paradigms where shared mutable state can cause strange behavior.
* By avoiding mutation, the same array can be used repeatedly across multiple operations, eliminating the need to create copies manually or reinitialize the data.
* Non-mutative operations reduce the risk of un planned changes, making the program easier to debug and maintain.

**Using Delegates (Functions as First-Class Values)**

A computer screen with text

Description automatically generated

* Delegates allow the sorting and searching algorithm to be passed as a parameter. This makes it easy to switch between different sorting and search strategies without changing the core code.
* The delegate approach decouples the sorting and searching logic from the execution framework, enabling the same DisplayRunningTime function to evaluate any algorithm dynamically.
* Delegates promote modular design by isolating the sorting logic from the performance evaluation mechanism, improving code readability and maintainability.
* Delegates enable higher-order functions, fostering a functional programming style within a predominantly object-oriented language like C#.

**Report For C**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best case  (target is the first item in the array) | Average case  (target is the middle item in the array) | Worst case  (target is the last item in Array) |
| Linear Search | 0.1445 | 0.1247 | 0.5624 |
| Binary Search | 0.2104 | 0.0016 | 0.0017 |
| Using Lambda | 2.1502 | 0.4846 | 0.6965 |

**1, Which algorithm wins**

Binary search wins based on the above table

**2, Reflect on the results.**

* **Binary Search** takes less time for large and sorted datasets because of its logarithmic approach.
* **Linear Search** performs faster but not as fast as binary search and is more versatile, as it doesn't need sorted data.
* **Lambda-based Search** provides clear and flexible code but is functionally similar to linear search unless used with specialized data structures.

**3, Describe one of these algorithms and how it works.**

**Linear Search Explanation**

Linear search is the simplest searching algorithm, where you sequentially check each element of the array or list until the target element is found or the end of the collection is reached.

**How It Works:**

1. Start at the first element of the array.
2. Compare the current element with the target:
   * If they match, the target is found.
   * If not, move to the next element.
3. Repeat step 2 until:
   * The target is found (return its index).
   * The end of the array is reached (indicating the target is not in the array).

**Time Complexity:**

1. **Best Case:**
   * The target is the first element in the array, so only one comparison is needed.
2. **Average Case:**
   * On average, the target is somewhere in the middle of the array.
3. **Worst Case**
   * The target is the last element in the array or is not in the array at all. Every element must be checked.

**Advantages:**

* + Very easy to understand and implement.
  + Works on both sorted and unsorted arrays.
  + Can be used on any data structure that supports sequential access (e.g., arrays, lists, linked lists).

**Disadvantages:**

* + Requires n comparisons in the worst case, making it slow for large datasets.
  + Other algorithms like binary search perform much faster on sorted datasets.

**4, Instead of calling the searching algorithms, search using Lambda expression and then fill in the table with running time.**

Instead of explicitly implementing search algorithms like linear search or binary search, we can use a lambda expression to search for an element in a collection. In C#, the Array.Find or Array.IndexOf methods can be used with a lambda expression to find a specific target efficiently, I used Array.IndexOf with FirstOrDefault method which identifies the first element that matches the condition.

**Report For B**

**1. Data Structures Comparison Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Structure | Addition | Search | Deletion | Access by Index |
| Array (T[]) |  |  |  | O(1) |
| Dynamic Array (List) |  |  |  | O(1) |
| Stack |  |  |  | O(n) (no direct access) |
| Queue |  |  |  | O(n) (no direct access) |
| Dictionary |  |  |  | Not applicable |
| SortedDictionary |  |  |  | Not applicable |
| HashSet |  |  |  | Not applicable |

**2. Reflection on the Results**

* **Arrays:** Efficient for random access, but expensive for dynamic operations like addition, deletion, and search due to lack of flexibility in resizing or data organization.
* **Dynamic Arrays (Lists):** Excellent for addition at the end due to gradual resizing, but search and deletion remain costly and complex as the size grows.
* **Stacks and Queues:** Specialized structures optimized for LIFO (Last In, First Out) and FIFO (First In, First Out) operations, respectively. They are simple and performance well for their intended use cases but inefficient for random access or search.
* **Dictionary:** Provides fast addition, search, and deletion on average, making it ideal for scenarios requiring fast lookups and updates. It lacks indexed access but excels in key-value pair storage and retrieval.
* **SortedDictionary:** Useful for maintaining ordered key-value pairs with efficient operations, but slower than Dictionary for average cases due to additional ordering overhead.
* **HashSet:** Like a Dictionary but stores only keys. Excellent for uniqueness checks and fast lookups, particularly when order is not important.

**3. Benefits of Dictionary in Functional Programming**

**Key Advantages:**

1. **Immutability:** Functional programming often emphasizes immutability. Dictionaries can be easily adapted to functional paradigms where modified instances are returned rather than in-place updates.
2. **Efficient Key-Based Access:** Functional programming relies heavily on mappings, such as translating one set of values to another. Dictionaries provide constant-time (O(1)O(1)O(1)) access to values by keys, streamlining these operations.
3. **Declarative Syntax Compatibility:** Dictionaries align with the declarative nature of functional programming. They allow developers to focus on *what* data they need (e.g., via keys) rather than *how* to access it (e.g., looping through arrays).
4. **Ease of Composition:** Dictionaries integrate seamlessly with higher-order functions like Map, Filter, and Reduce. For example:
5. **Support for Referential Transparency:** Since functional programming emphasizes no side effects, Dictionaries work well with snapshot-based or versioned updates that preserve the original state while creating a modified copy.

**Summary**

Dictionaries are a powerful data structure in functional paradigms. Their combination of efficiency, flexibility, and compatibility with functional concepts like immutability and higher-order functions makes them strong for key-value-based computations.

**Conclusion**

This project provided valuable insights into the performance and application of various sorting and searching algorithms in programming. Through the exploration of algorithms such as Bubble Sort, Selection Sort, Merge Sort, Insertion Sort and Quick Sort, alongside dynamic implementations using lambda expressions, the exercise underscored the importance of selecting the right algorithm based on the dataset size and problem requirements.

The results revealed the inefficiency of O(N²) algorithms like Bubble Sort and Selection Sort for large datasets, contrasting with the superior scalability of O(N log N) algorithms such as Merge Sort and Quick Sort. The use of lambda expressions and delegates further demonstrated the power of functional programming paradigms in creating modular, maintainable, and efficient code.

The project also highlighted the utility of data structures like Dictionaries for efficient key-value operations and their compatibility with functional programming principles. This reinforces the significance of understanding both theoretical and practical aspects of algorithm and data structure selection in solving real-world programming challenges.

Overall, this exercise enhanced technical problem-solving skills and deepened the understanding of computational efficiency, which are crucial for developing robust and scalable software solutions.

**References:**

**Sorting algorithms:** <https://www.cloudthat.com/resources/blog/exploring-the-sorting-algorithm-in-c>

**Sorting algorithms:** <https://medium.com/engineering-hub/https-medium-com-engineering-hub-sorting-algorithms-in-csharp-and-java-4615f6f87696>

**Searching algorithms:** <https://www.cloudthat.com/resources/blog/exploring-powerful-searching-algorithms-in-c>

**Searching algorithms:** <https://www.luigisbox.com/blog/types-of-search-algorithms/>

Used ChatGPT for code, content refactoring and helping hand during my hiccups in understanding the algorithm and other parts