|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **n=100** | **n=5000** | **n=100,000** | **n=1000,0000** |
| **Insertion Sort** | 0.0875 ms | 2.9855 ms | 994.9309 ms | 197870.0063 ms |
| **Selection sort** | 0.0867 ms | 4.317 ms | 1511.2368 ms | 267384.8833 ms |
| **Bubble Sort** | 0.1212 ms | 8.3937 ms | 10738.2656 ms | 1289654.2563 ms |
| **Merge sort** | 0.14 ms | 1.4735 ms | 15.6803 ms | 126.4236 ms |
| **Quick sort** | 0.0796 ms | 0.9454 ms | 9.4632 ms | 68.8677 ms |
| **Using Lambda** | 0.1372 ms | 0.3524 ms | 10.1566 ms | 115.8175 ms |

FSU 600 LAB 1 – Elias Kahwaji

# Part D

1- Which algorithm wins (the best one)?

Quick Sort is the winner.

2-Reflect on the results.

* Insertion Sort and Selection Sort perform well with small arrays, but their time complexity is quadratic, so as the array size grows, their performance degrades significantly.
* Bubble Sort is the slowest across all array sizes, which is expected given its inefficient algorithm for larger datasets.
* Merge Sort shows good performance and better scalability than insertion, selection, and bubble sorts due to its O(n log n) time complexity. However, it is still slow than quick sort.
* Quick Sort ends up being the best due to its approach of breaking down problems to a smaller subset of problems, which on average has a time complexity of O(n log n) but with better constants than merge sort.
* Using Lambda expressions for sorting provides an easy to write and understand code but at the cost of performance as its quality degrades as the arrays become bigger.

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

* Avoiding Mutation preserves the original data, which is crucial in scenarios where data integrity is important. It also prevents side effects, making the code more predictable and easier to debug.
* Using Delegates: Delegates allow for flexible code where functions can be passed around and used as arguments. Including the delegate as part of the DLL makes it easy to implement in any application. Delegates also make it easier to change the behavior of sections of the program without rewriting them. For Example: it allows DisplayRunningTime to be used with any sorting method, improving code reusability and abstraction.

# Part 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** | 0.0992 ms | 1.4063 ms | 3.2682 ms |
| **Binary Search** | 0.0056 ms | 0.0013 ms | 0.0018 ms |
| **Using Lambda** | 0.7102 ms | 6.4461 ms | 16.2002 ms |

1- Which algorithm wins?

Binary search is the winner.

2- Reflect on the results

* **Linear Search** shows its strength when the item you're looking for is right at the start. However, its performance drops as the target moves further away because it potentially has to go through every element until it finds what it's looking for.
* **Binary Search** excels across the board by cutting the search area in half with each step. This characteristic makes it reliable and fast, but it does need the array to be in order, which is a prerequisite for it to work.
* **Lambda Search** is all about simplicity and readability in code, which sometimes comes at the expense of speed. For bigger data sets, this type of search tends to lag behind more specialized search methods like Binary Search.

3- Describe one of these algorithms and how it works.

* Binary Search requires a sorted array to search.
* Binary Search takes the middle element as the starting element.
* If this element matches the search target, then the search concludes successfully.
* If the target is smaller than this middle element, the search continues to the left half of the array; if larger, it continues to the right half.
* Binary search uses a recursive logic to repeat the above steps by halving the array until it zeroes it gets closer to the target.
* The search ends when the target is located or when the size of the considered subarray reduces to zero, indicating the target isn't present.
* This halving approach is what grants Binary Search its **O(log n)** efficiency

4- Elias Notes:

* I think the question in part C considering The Best Case to be the first item in the array, the average case to be the middle item in the array and the worst indicating that the target is the last item in the array is potentially flawed. Since Binary search starts from the middle, the best case would be Array[(Array.length-1)/2] Which explains why the average case in the table above is the fastest.
* Also in a lot of the cases when running repetitive tests we see that modern CPUs have an effect on caching and measuring speed using the stopwatch class is also affected by the JIT compiler that .net uses so I ended up including extra options in the menus that uses a Benchmarking tool that gives extremely refined results (Hence why the project is using C# .net 4.8 since this is where the tool is supported)
* The following benchmark is using nano seconds shows that the binary search is way faster in all cases than using stopwatch and way closer.A screenshot of a computer screen

  Description automatically generated

# Part B

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Addition** | **Search** | **Deletion** | **Access By Index** |
| **Array (T[])** | **O(n)** | **O(n)** | **O(n)** | **1** |
| **Dynamic array (List)** | **1** | **O(n)** | **O(n)** | **1** |
| **Stack** | **1** | **N/A** | **1** | **N/A** |
| **Queue** | **1** | **N/A** | **1** | **N/A** |
| **Dictionary** | **1** | **1** | **1** | **N/A** |
| **SortedDictionary** | **O(log n)** | **O(log n)** | **O(log n)** | **N/A** |
| **HashSet** | **1** | **1** | **1** | **N/A** |

1-Reflect on the results.

* Arrays
  + Access time is consistent, which is important for operations that require predictable timing.
  + Manipulating arrays is often resource consuming because it requires cloning the array in memory modifying the size and then deleting the previous one.
  + Speed:
    - Addition: O(n) adding an element to an array that requires resizing (like inserting at a position that doesn't exist). If adding to the end without resizing, it would be O(1).
    - Search: O(n) because you may have to look through each element.
    - Deletion: O(n) because, after deleting an element, you may need to shift the elements to fill the gap.
    - Access By Index: O(1) because arrays provide constant time access due to direct memory allocation.
* Lists (Dynamic Arrays)
  + Lists can automatically resize, which makes them ideal for collections where the number of elements is unknown or changes frequently. It’s also great for addition and deletion operations.
  + Speed:
    - Addition: Typically O(1) on average when adding at the end, but O(n) if the array needs to be resized or if adding to the middle.
    - Search: O(n) as you might have to traverse the whole list to find an item.
    - Deletion: O(n) for similar reasons to searching—removing an element typically requires elements to be shifted.
    - Access By Index: O(1) due to direct index access.
* Stacks
  + Order of elements in a stack ensures a Last-In-First-Out (LIFO) order, which is crucial for certain algorithms, like depth-first search or undo features in applications.
  + Simplicity: Stacks have a simple and intuitive set of operations (push, pop), making them easy to use and implement.
  + Speed:
    - Addition: O(1) for pushing an item onto the stack. This operation doesn't depend on the size of the stack, so it's always done in a fixed amount of time.
    - Search: N/A. You can technically search a stack but you’d have to pop every element and you have to store these elements somewhere else it’s not a very straightforward application.
    - Deletion: O(1) for popping the top item off the stack. Like pushing, popping doesn't depend on the size of the stack because it always involves just the top element.
    - Access By Index: Is not a stack operation and is not supported.
* Queues
  + Unlike stacks Queues enforce a First-In-First-Out (FIFO) that is ideal for processing tasks in the order they're received.
  + Speed:
    - Addition: O(1) for enqueueing an item at the end of the queue. The complexity remains constant regardless of the queue's size because the operation targets the end of the queue directly.
    - Search: N/A You can technically search a queue just like stacks but you’d have to pop every element and you have to store these elements somewhere else it’s not a very straightforward application.
    - Deletion: O(1) for de-queueing the item at the front of the queue. The operation is quick and independent of the queue's size because it always involves the front element only.
    - Access By Index: Like stacks, not supported.
* Dictionaries
  + Dictionaries associate keys with values, enabling the creation of look-up tables.
  + They are versatile and can store various types of data, and keys such as numbers, strings, or other objects, allowing for complex data structures.
  + Dictionaries often use hash tables underneath, offering fast operations for many elements.
  + Speed:
    - Addition: Generally O(1), When adding a new key-value pair to a Dictionary, the key is hashed using a hash function, which efficiently determines the index at which to store the value in the underlying array.
    - Search: O(1) for looking up a value by its key; A hash function is used to determine the index. Just like addition and that causes very fast lookup times.
    - Deletion: O(1) for removing an element by its key. A hash function is used to determine the index.
    - Access By Index: Not applicable (N/A), as dictionaries are accessed by key, not by index.
* Sorted Dictionaries
  + Sorted Dictionaries automatically sort entries, saving time and resources on having to order the list after each insertion.
  + Search Efficiency Offers efficient searching capabilities, as it takes advantage of binary search principles due to underlying tree structures.
  + Speed:
    - Addition: O(log n) due to maintaining the sorted order.
    - Search: O(log n) When searching for a specific key, the Sorted Dictionary takes advantage of its tree structure. Instead of checking every entry, it follows a path down the tree, cutting down the number of comparisons needed to find the key.
    - Deletion: O(log n) After removal of an entry, the dictionary might need to rearrange itself slightly to maintain its sorted order and tree structure. The time this takes is proportional to the height of the tree, leading to the O(log n) complexity.
    - Access By Index: Not typically applicable (N/A) as Sorted Dictionaries are accessed by Key
* HashSet:
  + HashSets are similar to dictionaries without storing associated values. They're optimized for quickly determining whether an element is present in the set.
  + Speed:
    - Addition: O(1) on average, Uses a hash function to calculate the index so access time is a constant time.
    - Search: O(1) for checking the presence of an element. Uses a hash function to calculate the index so search is a constant time.
    - Deletion: O(1) for removing an element. Uses a hash function to calculate the index so knowing the location of the item is very quick.
    - Access By Index: Not applicable (N/A), as sets are not indexed collections.

2-Explain the benefits of Dictionary data structure from the functional programming perspective.

* Use of Immutable Structures:
  + Functional programming often designs dictionaries to be unchangeable once created since we follow the immutability concept. Changes like additions or deletions result in a new dictionary, aiding in maintaining clear and predictable code.
* Support for Pure Functions:
  + Dictionaries enable the creation of pure functions that don’t alter state or produce side effects, ensuring consistent outputs for given inputs. This consistency is fundamental in functional programming and simplifies both testing and debugging.
* Compatibility with Higher-Order Functions:
  + Functional programming often utilizes functions like map, filter, and fold (reduce), which are ideal for working with dictionaries.
* Efficient data access and Modification:
  + The ability of dictionaries to allow swift data retrieval and updates is particularly useful in functional programming, especially when handling large datasets since it follows the concepts of parallel processing of data, making code more readable, simplifying access to nested and complex data structures.
* Facilitating Recursive Solutions:
  + Dictionaries use key-value pairs, which allows the utilization of recursive techniques, particularly useful for nested or complex data structures.
* Suitability for Concurrency:
  + The thread-safe nature (Ex: Accessing data from multiple threads without data corruption) of immutable dictionaries allows easier development of applications involving concurrency.