# Section D:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **n=100** | **n=5000** | **n=100,000** | **n=1000,0000** |
| **Insertion Sort** | 0.1691 ms | 25.188 ms | 9577.8801 ms |  |
| **Selection sort** | 0.3128 ms | 34.6696 ms | 13310.5107 ms |  |
| **Bubble Sort** | 0.5803 ms | 173.4122 ms | 73542.8113 ms |  |
| **Merge sort** | 0.6581 ms | 1.4401 ms | 26.9381 ms |  |
| **Quick sort** | 0.573 ms | 1.4962 ms | 35.884 ms |  |
| **Using Lambda** | 0.0262 ms | 1.3909 ms | 28.9444 ms |  |

**Which algorithm wins? –** Lambda Function

**Reflect on the results.**

If we consider the above table, we can say sorting with Lambda wins the test. I’m following not-in-place approach to sort the array and using OrderBy method of Linq array in this algorithm. This result does not reflect the exact output but can provide us some reference on different types of sorting algorithms and their time complexity. Since I was generating random data for the array, the speed of sorting may differ every time we test with the system. As we can see in the above table, for the array with 100 integers, insertion sort and lambda function performed really well with 0.1691 milliseconds and 0.0262 milliseconds respectively. All the other algorithms sorted in similar time. However, if we consider a little bit bigger sized array of 5000 integers, we can clearly differentiate which algorithm is better. Lambda function, quick ort and merge sort performed dramatically better than bubble sort, insertion and selection sort. We can see the time taken by bubble sort is increasing with huge margin. Where Lambda function, quick sort and merge sort sorted 5000 integers under 1.5 milliseconds, insertion and selection sort took about 25.188 and 34.66 milliseconds respectively and bubble sort took 173.41 milliseconds for the same. After this, I increased the array size to 100000 and result was surprising. If we see the table, merge sort seems to be the fastest one with just 28.9351 milliseconds whereas bubble sort is at last with 73542.8113 milliseconds. Quick sort and Lambda function following merge sort with 35.884 milliseconds and 28.9444 milliseconds whereas insertion sort performing well comparing to selection sort with 9577.8801 milliseconds and 13310.5107 milliseconds. I could not test beyond this limit because of my system limitations. However, considering the above table I can say that for smaller data below 5000 size, every algorithms seems similar but with the increase in data, merge sort, quick sort and lambda function are tends to be the fastest option.

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

Mainly there are two types of sorting approaches i.e. in-place approach and out-of-place approach. In-place approach requires low space but works slower than the out-of-place approach in which the memory resource use is higher so the processing speed. As we can see in the above table, I’m using OrderBy method of Linq in Lambda function to sort the array, which seems to be one of the most efficient method. This method implements the out-of-place approach and do not mutate the original state and create new state and store new data in that. With the availability of storage devices and RAM capacity, this approach is faster than in-place approach in which memory-consumption is low.

Using delegates make the work easier and reduce the function invoke numbers. With delegates or function as first-class values, we can assign functions to any variables and pass in the parameters and invoke whenever we want or from where ever we want by passing it through parameters or storing it in variable. It helps to keep the code clean, readable and maintainable for the long term. It also helps in responding to an even triggered by the user to do the specific task.

# Section C

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best case (first item)** | **Average case(middle item)** | **Worst case(last item)** |
| **Linear Search** | 0.4935 ms | 0.4783 ms | 0.507 ms |
| **Binary Search** | 0.5986 ms | 0.4141 ms | 0.4141 ms |
| **Using Lambda** | 0.865 ms | 0.4531 ms | 0.4576 ms |

**Which algorithm wins? –** Binary Search

**Reflect on the results.**

If we consider the above table, binary search seems to win the race against linear search and Lambda function. I have used an array of size 100000 for this test and all the numbers were randomly generated. I tried to search three different values from first index, last index and the one from middle. Since the data used in each tests were randomly generated, the result may defer test to test and we may not get the same result again. However the result gives us an insight on overall performance of three different search algorithms i.e. linear search, binary search and using lambda function. As we can see, Binary search is wining in all three cases with the lowest time i.e. 0.5986 milliseconds, 0.4141 milliseconds and 0.4141 milliseconds for first, middle and last element respectively. Linear search seems to be the slowest among these three algorithms and Lambda function in in the second position with 0.865 milliseconds, 0.4531 milliseconds and 0.4576 milliseconds for first, middle and last element respectively.

Time taken for sorting data for binary search is not considered and just the searching time is tracked.

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

Linear search algorithm is considered as one of the easiest searching algorithm available and used till date. This algorithm is also known as sequential algorithm because of its working mechanism. It starts to search for the element from the beginning to the end and goes through each element in the structure. It compares the desired element with each of the element in the structure subsequently and keep going until it reaches the desired result.

Let’s say we have an array myArray of n elements and wants to search element a in the array.

Example:

int [] myArray ={10,13,16,18,32,56,87,67,89,20,50}, a=56

Output: 5

Since the element a is present in index 5, the output

It returns -1 if we try to search any element that is not in the given array.

**How it works:**

It iterate from index 0 to n-1 and compare value of each index with ‘a’ and return index if they match else keep increasing the index.

int [] myArray ={10,13,16,18,32,56,87,67,89,20,50}, a=56

**First iteration: index 0**

10 == 56 => false => increase index

**Second iteration: index 1**

13 == 56 => false => increase index

**Second iteration: index 2**

16 == 56 => false => increase index

**Third iteration: index 3**

18 == 56 => false => increase index

**Fourth iteration: index 4**

32 == 56 => false => increase index

**First iteration: index 5**

56 == 56 => true => return 5

# Section B

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Addition** | **Search** | **Deletion** | **Access by index** |
| **Array (T[])** | O(n) | O(n) | O(n) | 1 |
| **Dynamic array (List)** | O(1) | O(n) | O(n) | O(1) |
| **Stack** | O(1) | NA | O(1) | NA |
| **Queue** | O(1) | NA | O(1) | NA |
| **Dictionary** | O(1) | O(1) | O(1) | NA |
| **Sorted Dictionary** | O(1) | O(1) | O(1) | NA |
| **HashSet** | O(1) | O(1) | O(1) | NA |

**Reflect on the results.**

The above table reflects the average time complexity for different types of operation in different data structures. I have tried to compare and find out the time complexity for different data structure by writing different codes and tracking the time. As we can see in the above table, HashSet, Sorted Dictionary and Dictionary seems to be the quickest for addition, search and deletion of the value. However, Array seems to be the quickest for accessing the data by index in comparison to dynamic array (List). Any of the other data structures doesn’t offer the option to access the element by index and can be searched. For the data addition task; List, Stack, queue, dictionary and HashSet seems to take similar time i.e. O(1) whereas, only the array is slower than others. For searching, Stack and Queue does not support the search operation and array and list seems to be equal and slower than other data structures which offers search operation. For data deletion also, Array and List seems slower than others with O(n) whereas others are offering the time complexity of O(1).

**Explain the benefits of Dictionary data structure from the functional programming perspective**

Dictionary comes under generic collection and uses the concept of hashtable. It store values based on the keys and can only store the unique keys. Some of the benefits of dictionary are given below:

* It provides the comprehensive and organized list of data
* Easy to search
* Simplifies the structure for system requirements
* No data redundancy
* Main data integrity

It provides the option of data immutability and make it easier to transport the data from one function to another function through the parameters in functional programming.