Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Lab Report

on

“Linear Search & Binary Search Time Complexity Analysis”

[Code No. : COMP 314]

Submitted by:

Prafful Raj Thapa (59)

Submitted to:

Dr. Rajni Chulyadyo

Department of Computer Science and Engineering

December 19, 2024

Contents

List of Figures

List of Tables

Chapter 1 : Linear Search

1.1 Linear Search Algorithm

Linear Search Algorithm, also known as sequential search algorithm, is a simple algorithm that traverse the provided list or array to find the target element. In Linear Search we get the algorithm to iterate through each element of the list or array, comparing it with the target element until a match is found, or the end of the list or array is reached. If the end of the list or array is reached, then it means that the target element is not present in the array. The time complexity of the linear search algorithm is $O(n)$, where n is the number of elements in the list or array. This means the time taken for searching increases linearly with the size of the list or array searched.

1.2 Source Code

```python
def Linear_Search(DATA_ARRAY:list[int], KEY:int) -> int:
    index:int = 0
    for DATA in DATA_ARRAY:
        if DATA == KEY:
            return index
        index = index + 1
    raise Exception("Value Not in Array")
```

1.3 Unit Testing

```python
import unittest
from datetime import datetime, time
from Linear_Search import Linear_Search
from Linear_Search_Data import Linear_Search_Data

class Test_Linear_Search(unittest.TestCase):
    test_data: list[int] = list(range(0,10000))

    def test_Linear_Search_01(self):
        # for best case
        # search first element of the list
        (search_key, search_key_index) = 0, 0
        searched_index:int = Linear_Search(DATA_ARRAY=self.test_data,
                                           KEY=search_key)
        self.assertEqual(searched_index, search_key_index)

    def test_Linear_Search_02(self):
        # for Average case
        # search appox. middle element of the list
        (search_key, search_key_index) = 5000, 5000
        searched_index:int = Linear_Search(DATA_ARRAY=self.test_data,
                                           KEY=search_key)
        self.assertEqual(searched_index, search_key_index)

    def test_Linear_Search_03(self):
        # for Worst Case
        # search last element of the list
        (search_key, search_key_index) = 9999, 9999
        searched_index:int = Linear_Search(DATA_ARRAY=self.test_data,
                                           KEY=search_key)
        self.assertEqual(searched_index, search_key_index)

    def test_Linear_Search_04(self):
        # search empty list
        # the value must be -1
        searched_index:int = Linear_Search(DATA_ARRAY=list(), KEY=0)
        self.assertEqual(searched_index, -1)

    def test_Linear_Search_015(self):
        # search element that is not in the list
        searched_index:int = Linear_Search(DATA_ARRAY=self.test_data,
                                           KEY=1000000000)
        self.assertEqual(searched_index, -1)

if __name__ == "__main__":
    unittest.main()
```

1.4 Time Complexity Analysis

1.4.1 Methodology

The implementation and evaluation of the linear search algorithm were conducted systematically to ensure reliable and reproducible results. The process involved the following steps:

1. **Algorithm Implementation and Testing:** The linear search algorithm was implemented in Python. Its correctness was verified using the Python `unittest` library to ensure accuracy and consistency under various conditions.

2. **Data Generation for Time Analysis:** A separate script was written to generate datasets of varying sizes, specifically for analyzing the algorithm's time complexity. The datasets included lists with sizes of 100 elements, 1,000 elements, 10,000 elements, 100,000 elements, and progressively larger sizes up to 5,000,000 elements (e.g., 500,000, 1,000,000, 1,500,000, and so on, in increments of 500,000).

3. **Case Analysis:** For each dataset, the algorithm's performance was measured in three scenarios:

   - **Best Case:** The target element was located at the very beginning of the list.
   - **Average Case:** The target element was positioned randomly within the list.
   - **Worst Case:** The target element was either at the end of the list or absent entirely.

4. **Time Measurement:** The time taken by the algorithm to search for the target element in each dataset was recorded. These timings were consolidated into a table for comparison and analysis.

5. **Visualization:** The recorded data was visualized in a graph, providing a clear representation of the relationship between dataset size and algorithm performance under each scenario.

This approach ensured that the analysis captured the linear search algorithm's behavior across a wide range of input size, offering valuable insight into its efficiency & scalability.

1.4.2 Data

The algorithm's execution time was measured for best-case, average-case, and worst-case scenarios. The results were compiled into a table 1.1 and visualized through a graphs 1.1, 1.2, 1.3 to illustrate the relationship between input size and search performance.

Data Size	**Time Taken**		
	Best Case	**Average Case**	**Worst Case**
0.1K	2	4	5
1K	2	18	37
10K	1	198	390
100K	1	2026	4005
0.5M	2	9966	20421
1.0M	2	20332	39817
1.5M	1	30127	59930
2.0M	1	39823	86475
2.5M	1	50952	100204
3.0M	1	59963	122929
3.5M	1	69957	140315
4.0M	1	81804	160270
4.5M	1	89949	180638
5.0M	1	100124	200950

Table 1.1: Time complexity Analysis Data of Linear Search Algorithm

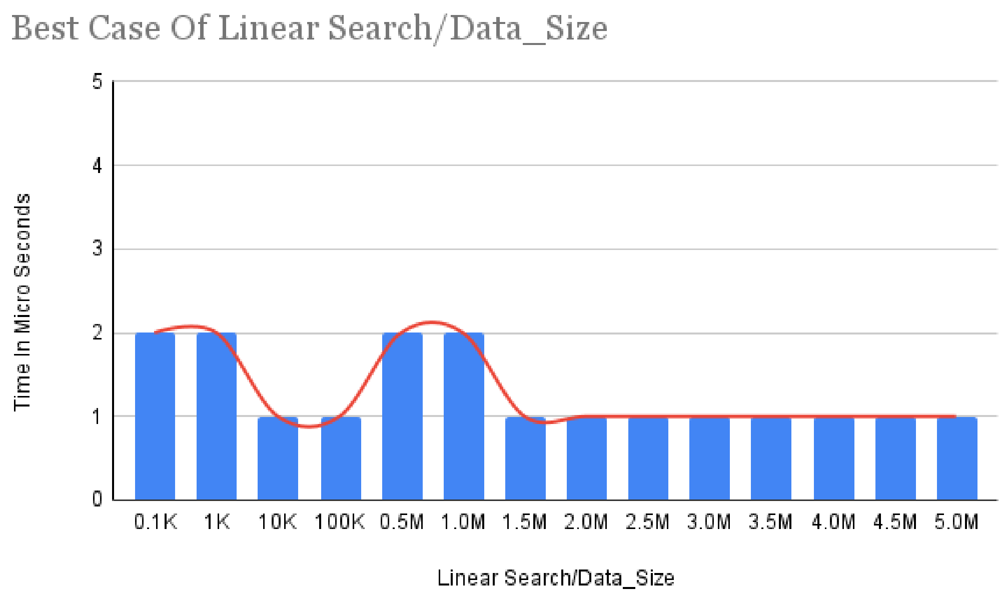

Figure 1.1: Best Case Of Linear Search Algorithm

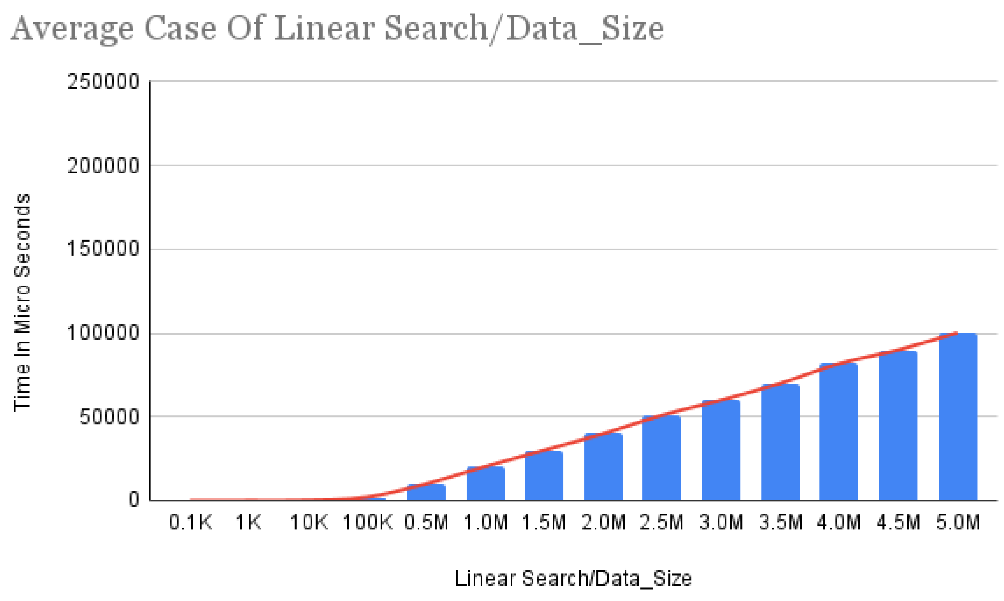

Figure 1.2: Average Case Of Linear Search Algorithm

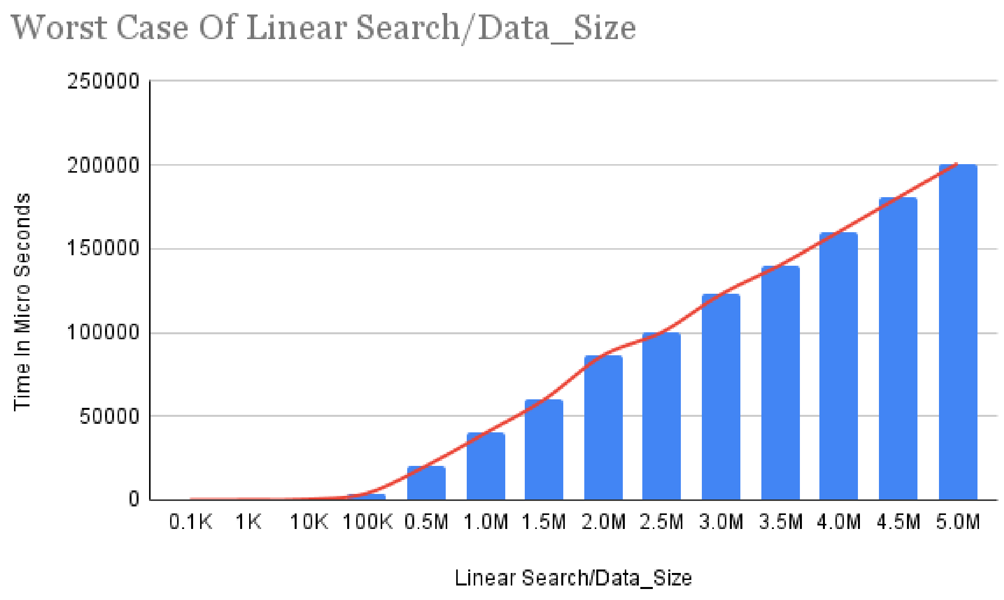

Figure 1.3: Worst Case Of Linear Search Algorithm

1.4.3 Analysis Result

The analysis of the gathered data and corresponding graphs reveals distinct patterns in the performance of the linear search algorithm across different scenarios. In the best-case scenario, where the target element is located at the beginning of the list, the algorithm consistently demonstrated near-constant execution time regardless of the dataset size. This behavior aligns with expectations, as the search terminates after a single comparison.

For the average-case scenario, the graph depicts a linear relationship between the dataset size and execution time. This outcome reflects the algorithm's need to traverse, on average, half of the list to locate the target element, leading to a time complexity proportional to the input size.

In the worst-case scenario, where the target element is either at the end of the list or absent, the graph also shows a linear relationship. However, the slope of this graph is nearly double that of the average-case scenario, as the algorithm must perform a comparison for every element in the list. This highlights the maximum workload for linear search, reinforcing its O(n) complexity.

Overall, the analysis confirms that the linear search algorithm's performance scales linearly with input size for both average and worst cases, while maintaining constant time in the best-case scenario.

Chapter 2 : Binary Search

2.1 Binary Search Algorithm

Binary Search Algorithm a divide and conquer based efficient algorithm for finding a target value within an sorted list or array. It works by repeatedly dividing the search interval in half. Binary Search compares the target value to the middle element of the array. if they are equal, the search is successful. If the target value is less than the middle element, the algorithm strips the list or array after middle element, and the search continues in the lower half of the array. If the targe is greater, the lower half is striped and search continues in the upper half. This process repeats recursively until the target value is found in the list or array , or the search interval is empty. The time complexity of the Binary Search Algorithm is $O(log_2 n)$, where n is the number of element in the list or array. This is because the size of search interval is half-ed in each step.

2.2 Source Code

```
def Binary_Search(DATA_ARRAY:list[int], KEY:int, start:int=0, end:int
                                              =-1) -> int:
    if end == -1:
        end = (DATA_ARRAY.__len__() - 1)

        middle:int = (start + end) // 2

        if DATA_ARRAY[middle] == KEY:
            return middle

        elif KEY < DATA_ARRAY[middle]:
            return Binary_Search(DATA_ARRAY, KEY, start=start, end=(
                                                        middle-1)) # search
                                                        the left half

        elif KEY > DATA_ARRAY[middle]:
            return Binary_Search(DATA_ARRAY, KEY, start=(middle+1),
                                                        end=end)
        else:
            raise Exception("Value Not found")
```

2.3 Unit Testing

```python
import unittest
from datetime import datetime
from Binary_Search import Binary_Search
from Binary_Search_Data import Binary_Search_Data

class Test_Binary_Search(unittest.TestCase):
    test_data: list[int] = list(range(0,10000))

    def test_Binary_Search_01(self):
        # for best case
        # search middle element of the list
        (search_key, search_key_index) = 4999, 4999
        searched_index:int = Binary_Search(DATA_ARRAY=self.test_data,
                                           KEY=search_key)
        self.assertEqual(searched_index, search_key_index)

    def test_Binary_Search_02(self):
        # for Average Case
        # search any element of the list expect middle element or
                                          start, end element
        (search_key, search_key_index) = 2000, 2000
        searched_index:int = Binary_Search(DATA_ARRAY=self.test_data,
                                           KEY=search_key)
        self.assertEqual(searched_index, search_key_index)

    def test_Binary_Search_03(self):
        # for Worst Case
        # search start or end element of the list
        (search_key, search_key_index) = 0, 0
        searched_index:int = Binary_Search(DATA_ARRAY=self.test_data,
                                           KEY=search_key)
        self.assertEqual(searched_index, search_key_index)

    def test_Binary_Search_01(self):
        # for empty list as input
        searched_index:int = Binary_Search(DATA_ARRAY=list(), KEY=0)
        self.assertEqual(searched_index, -1)

    def test_Binary_Search_01(self):
        # for empty list as input
        searched_index:int = Binary_Search(DATA_ARRAY=self.test_data,
                                           KEY=10000)
        self.assertEqual(searched_index, -1)

if __name__ == "__main__":
    unittest.main()
```

2.4 Time Complexity Analysis

2.4.1 Methodology

The implementation and evaluation of the binary search algorithm were conducted systematically to ensure reliable and reproducible results. The process involved the following steps:

1. **Algorithm Implementation and Testing:** The binary search algorithm was implemented in Python. Its correctness was verified using the Python `unittest` library to ensure accuracy and consistency under various conditions.

2. **Data Generation for Time Analysis:** A separate script was written to generate datasets of varying sizes, specifically for analyzing the algorithm's time complexity. The datasets included lists with sizes of 100 elements, 1,000 elements, 10,000 elements, 100,000 elements, and progressively larger sizes up to 5,000,000 elements (e.g., 500,000, 1,000,000, 1,500,000, and so on, in increments of 500,000).

3. **Case Analysis:** For each dataset, the algorithm's performance was measured in three scenarios:

   - **Best Case:** The target element was located at the very beginning of the list.
   - **Average Case:** The target element was positioned randomly within the list.
   - **Worst Case:** The target element was either at the end of the list or absent entirely.

4. **Time Measurement:** The time taken by the algorithm to search for the target element in each dataset was recorded. These timings were consolidated into a table for comparison and analysis.

5. **Visualization:** The recorded data was visualized in a graph, providing a clear representation of the relationship between dataset size and algorithm performance under each scenario.

This approach ensured that the analysis captured the binary search algorithm's behavior across a wide range of input size, offering valuable insight into its efficiency & scalability.

2.4.2 Data

The algorithm's execution time was measured for best-case, average-case, and worst-case scenarios. The results were compiled into a table 2.1 and visualized through a graphs 2.1, 2.2, 2.3 to illustrate the relationship between input size and search performance.

Data Size	**Time Taken**		
	Best Case	**Average Case**	**Worst Case**
0.1K	2	3	3
1K	3	4	3
10K	2	4	5
100K	7	10	10
0.5M	11	19	22
1.0M	12	19	25
1.5M	14	22	24
2.0M	10	22	28
2.5M	12	26	28
3.0M	14	22	27
3.5M	13	26	29
4.0M	12	24	28
4.5M	12	27	27
5.0M	13	25	30

Table 2.1: Time complexity Analysis Data of Binary Search Algorithm

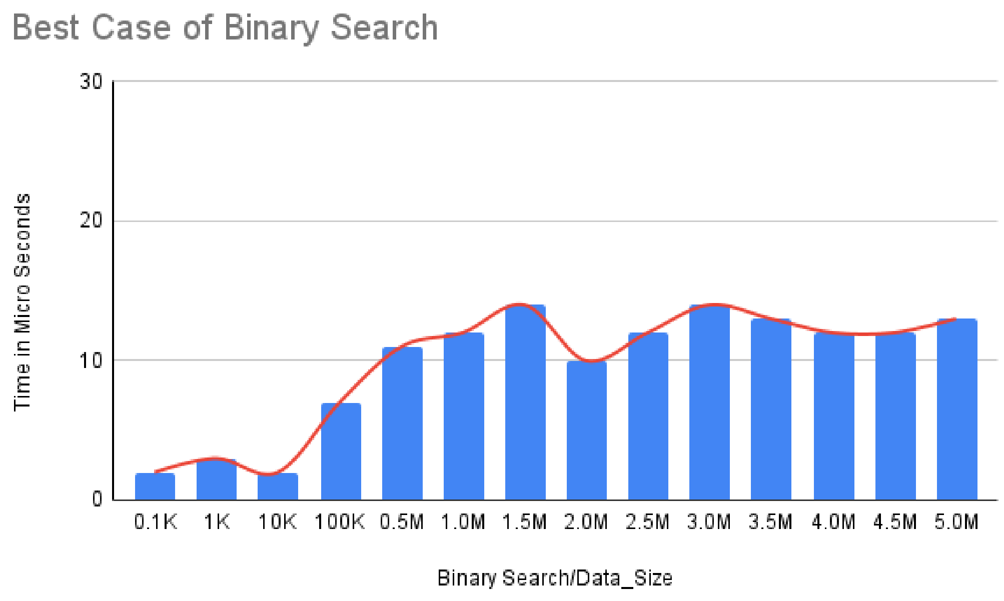

Figure 2.1: Best Case Of Binary Search Algorithm

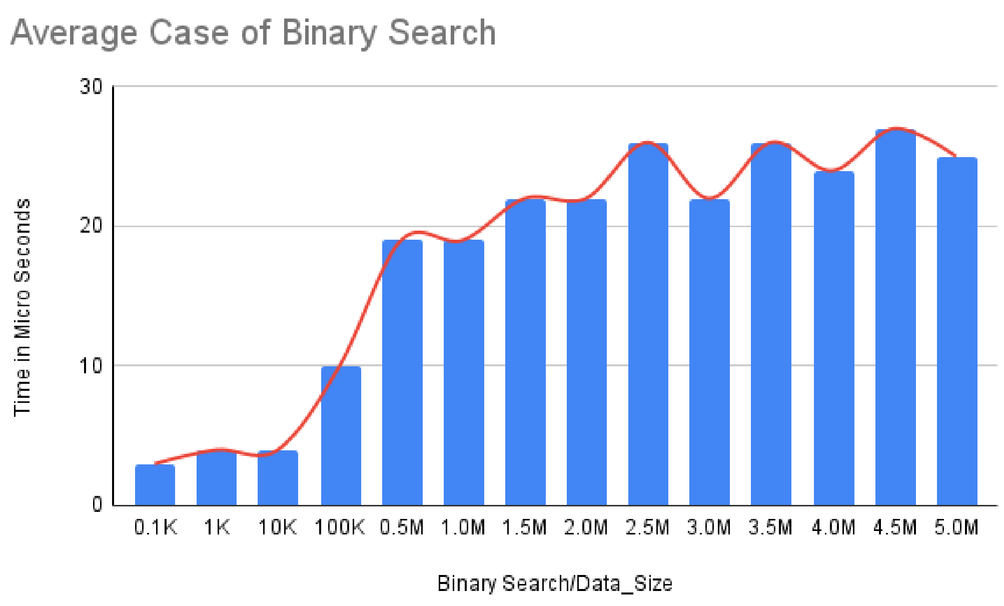

Figure 2.2: Average Case Of Binary Search Algorithm

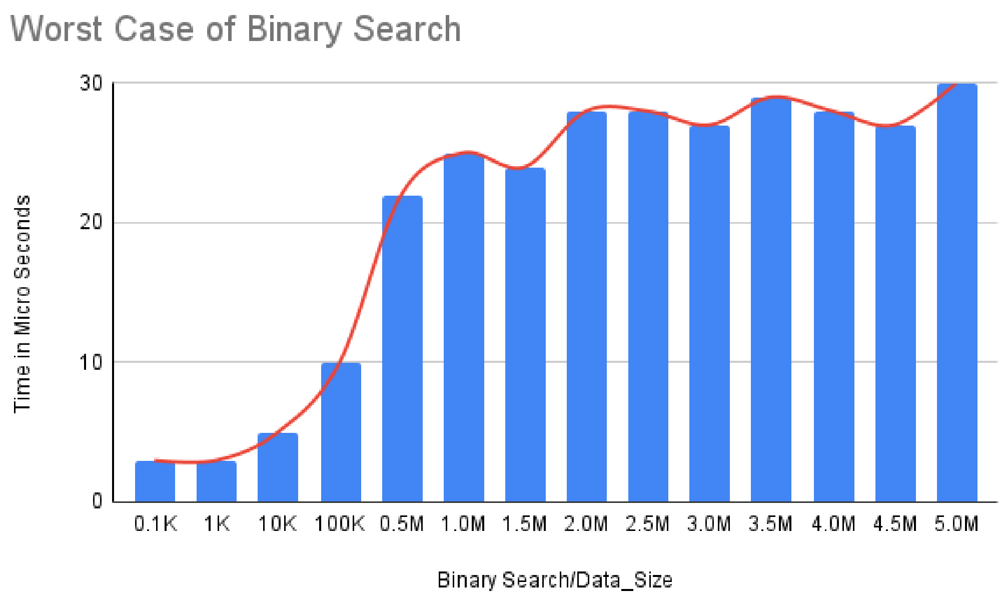

Figure 2.3: Worst Case Of Binary Search Algorithm

2.4.3 Analysis Result

The analysis of the gathered data and corresponding graphs highlights the efficiency of the binary search algorithm across different scenarios. In the best-case scenario, where the target element is located at the middle of the list during the first comparison, the algorithm exhibited almost constant execution time regardless of the dataset size. This is expected, as the search concludes in a single iteration.

For the average-case scenario, the graph illustrates a logarithmic relationship between the dataset size and execution time. This behavior is consistent with the binary search algorithm's strategy of halving the search space with each comparison, resulting in a time complexity of O(log n) for locating the target element in a sorted dataset.

In the worst-case scenario, where the algorithm exhaustively narrows down the search space to a single element before identifying the target or determining its absence, the graph also shows a logarithmic growth. The slope of this graph is slightly higher than that of the average case, reflecting the maximum number of comparisons required to complete the search process.

Overall, the analysis confirms that the binary search algorithm performs exceptionally well, with logarithmic scaling in both average and worst cases, while maintaining constant time in the best-case scenario. This highlights its efficiency for searching in sorted datasets compared to linear search.