Data Algorithms and Representation

Escuela Colombiana de Ingeniería Julio Garavito

Analysis of Sorting Algorithms

Zayra Gutiérrez Solano

Student

Rafael Alberto Niquefa Velasquez

Professor

2025-1

Contenido

Introduction 3

Search Algorithms Analysis 4

1. Linear Search 4

Algorithm Complexity 4

Characteristics 4

2. "In" Search 5

Algorithm Complexity 5

Characteristics 5

3. Binary Search 6

Algorithm Complexity 6

Characteristics 6

4. Ternary Search 7

Algorithm Complexity 7

Characteristics 7

Analysis 8

Methodology 8

Conclusion 15

Introduction

Search algorithms are fundamental in computer science, as they enable efficient retrieval of information from large datasets. Depending on the algorithm used, execution time can vary significantly, directly impacting scalability and applicability.

This study analyzes the performance of four search algorithms: linear search, the "in" operator search, binary search, and ternary search. Through experimental testing with different list sizes and maximum values, the impact of each algorithm’s computational complexity on search efficiency is evaluated.

The purpose of this analysis is to compare the execution time of each algorithm under various conditions and determine the best option for handling large datasets. The results will provide insight into the strengths and weaknesses of each method and their applicability in optimal search scenarios.

Search Algorithms Analysis

1. Linear Search

```python
# Lineal search implementation
# Array can be unsorted
3 usages  ZayraGS1403
def lineal_search(arr, target):
    for i in range(len(arr)):  # O(n)
        if arr[i] == target:  # O(1)
            return True  # O(1)
    return False  # O(1)


# O(lineal_search) = O(n) * O(1) * O(1) + O(1)
# O(lineal_search) = O(n)
```

Algorithm Complexity

- Best case: O (1) (When the target is found at the first index)
- Worst case: O(n) (When the target is at the last index or not present)
- Average case: O(n)

Characteristics

- Works on unsorted arrays.
- Simple and easy to implement.
- Not efficient for large datasets.

2. "In" Search

```python
# Search in implementation
# Array can be unsorted
2 usages  ZayraGS1403
def search_in(arr, target):
    if target in arr:  # O(n) en el peor caso
        return True  # O(1)
    else:
        return False  # O(1)


# O(search_in) = O(n) * O(1) + O(1)
# O(search_in) = O(n)
```

Algorithm Complexity

- Best case: O (1) (When the target is found at the first index)
- Worst case: O(n) (When the target is at the last index or not present)
- Average case: O(n)

Characteristics

- Works on unsorted arrays.
- Utilizes Python's built-in operator for searching.
- Performance is like linear search.

3. Binary Search

```python
# Binary search implementation
# Array must be sorted
3 usages  ZayraGS1403
def binary_search(arr, target):
    left = 0  # O(1)
    rigth = len(arr) - 1  # O(1)

    while (left <= rigth):  # It runs ( O(log n)) times because, in each iteration, the problem size is reduced by half.
        mid = left + (rigth - left) // 2  # O(1)

        if arr[mid] == target:  # O(1)
            return True  # O(1)

        elif arr[mid] < target:  # O(1)
            left = mid + 1  # O(1)

        else:
            rigth = mid - 1  # O(1)
    return False  # O(1)


# o(binary_search) = O(log n) * O(6) + O(2)
# o(binary_search) = O(log n)
```

Algorithm Complexity

- Best case: O (1) (When the target is found at the middle index)
- Worst case: O(log(n)) (When the search space is reduced to one element)
- Average case: O(log(n))

Characteristics

- Requires a sorted array.
- Efficient for large datasets.
- Reduces the problem size by half in each iteration.

4. Ternary Search

```python
# Ternary search implementation
# Array must be sorted
6 usages  ZayraGS1403
def ternary_search(arr, target):
    if len(arr) == 0:  # O(1)
        return False  # O(1)
    else:
        mid1 = len(arr) // 3  # O(1)
        mid2 = mid1 * 2  # O(1)

        if arr[mid1] == target or arr[mid2] == target:  # O(1)
            return True  # O(1)

        elif arr[mid1] > target:
            return ternary_search(arr[:mid1], target)  # O(T(n/3))

        elif arr[mid2] < target:
            return ternary_search(arr[mid2 + 1:], target)  # O(T(n/3))

        else:
            return ternary_search(arr[mid1 + 1: mid2], target)  # O(T(n/3))


# O(ternary_search) = o(log 3 n)
```

Algorithm Complexity

- Best case: O (1) (When the target is found at mid1 or mid2)
- Worst case: O (log3 (n)) (When the search space is reduced to one element)
- Average case: O (log3 (n))

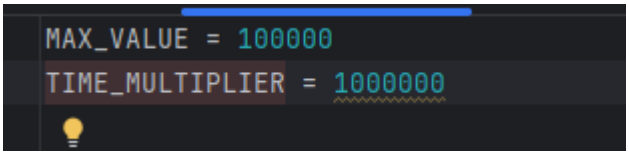
Characteristics

- Requires a sorted array.
- Divides the search space into three parts instead of two.
- Slightly less efficient than binary search due to additional comparisons.

Analysis

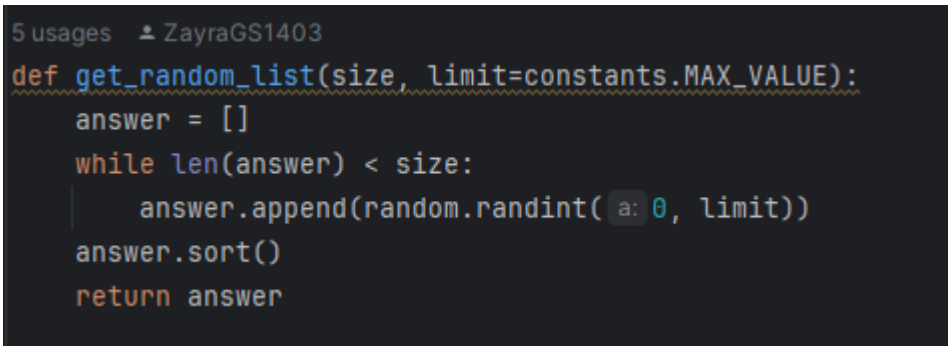
Methodology

Defines constants used throughout the project, such as MAX_VALUE (maximum value for random numbers) and TIME_MULTIPLIER (used for execution time measurement).



Contains functions for generating random lists of numbers.

The function get_random_list(size, limit) creates a sorted list of random integers up to a specified limit, which is essential for testing search algorithms.



Measures the execution time of different search algorithms.

Uses functions like take_execution_time() to run tests on various list sizes and collect execution times.

Calls take_time_for_algorithm() to measure the median execution time for each algorithm, ensuring a fair comparison.

```
1 usage   ZayraGS1403
def take_execution_time(minimum_size, maximum_size, step, samples_by_size):
    return_table = []

    for size in range(minimum_size, maximum_size + 1, step):
        table_row = [size]
        times = take_times(size, samples_by_size)
        return_table.append(table_row + times)

    return return_table
```

```
"""
    It will return three values, one for each algorithm: The execution time for that size on each algorithm
"""


1 usage   ZayraGS1403
def take_times(size, samples_by_size):
    samples = []
    for _ in range(samples_by_size):
        samples.append(data_generator.get_random_list(size))

    return [
        take_time_for_algorithm(samples, algorithms.lineal_search),
        take_time_for_algorithm(samples, algorithms.seach_in),
        take_time_for_algorithm(samples, algorithms.binary_search),
        take_time_for_algorithm(samples, algorithms.ternary_search)
    ]
```

```
"""
    Returns the median of the execution time measures for a sorting approach given in
"""


4 usages   ZayraGS1403
def take_time_for_algorithm(samples_array, sorting_approach):
    times = []

    for sample in samples_array:
        start_time = time.time()

        sorting_approach(sample.copy(), random.randint( a: 0, constants.MAX_VALUE))
        times.append(int(constants.TIME_MULTIPLIER * (time.time() - start_time)))

    times.sort()
    return times[len(times) // 2]
```

This is the main script that executes the performance analysis of different search algorithms.

It imports execution_time_gathering to measure execution times for various search methods.

Calls take_execution_time() to gather execution times for each algorithm.

Outputs a table displaying the results for Linear Search, "In" Search, Binary Search, and Ternary Search.

```
if __name__ == "__main__":
    minimum_size = 10000
    maximum_size = 100000
    step = 5000
    samples_by_size = 7

    table = execution_time_gathering.take_execution_time(
        minimum_size, maximum_size, step, samples_by_size
    )

    print("Size | LinealSearch | Search In | Binary Search | Ternary Search")
    for row in table:
        print(row)
```

Data

1.

- Minimum size 100000
- Maximum size 1000000
- Step100000 examples per size
- MAX_VALUE = 100000

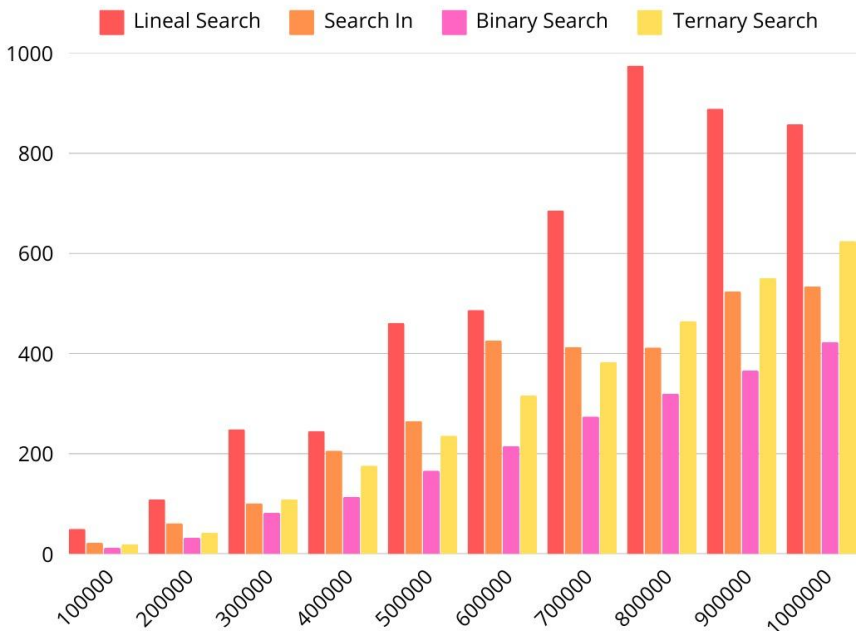
```
Size | LinealSearch | Search In | Binary Search | Ternary Search
[100000, 50, 22, 12, 19]
[200000, 109, 61, 32, 42]
[300000, 249, 101, 82, 109]
[400000, 245, 206, 114, 176]
[500000, 461, 265, 166, 236]
[600000, 487, 426, 215, 316]
[700000, 686, 413, 274, 383]
[800000, 975, 412, 320, 465]
[900000, 889, 524, 366, 551]
[1000000, 858, 534, 423, 625]
```

The execution time data highlights clear performance trends among the search algorithms. As expected, algorithms with linear complexity O(n), such as Linear Search and the "in" operator search, show a significant increase in execution time as the list size grows.

Linear Search exhibits the highest growth, reaching 858 at size 1,000,000, while the "in" operator performs slightly better at 534, yet still scales inefficiently for large datasets.

In contrast, logarithmic complexity O(log (n)) in Binary Search demonstrates far superior scalability, with execution times increasing gradually from 12 to 423, making it the most efficient option for handling large lists. Ternary Search, despite also having logarithmic complexity O (log_3 n), consistently underperforms compared to Binary Search, reaching 625 at size 1,000,000.

This suggests that splitting the list into three parts instead of two introduces unnecessary overhead, reducing efficiency. The results confirm that for large-scale searching, Binary Search is the optimal choice, while Linear Search and the "in" operator become impractical due to their poor scalability.



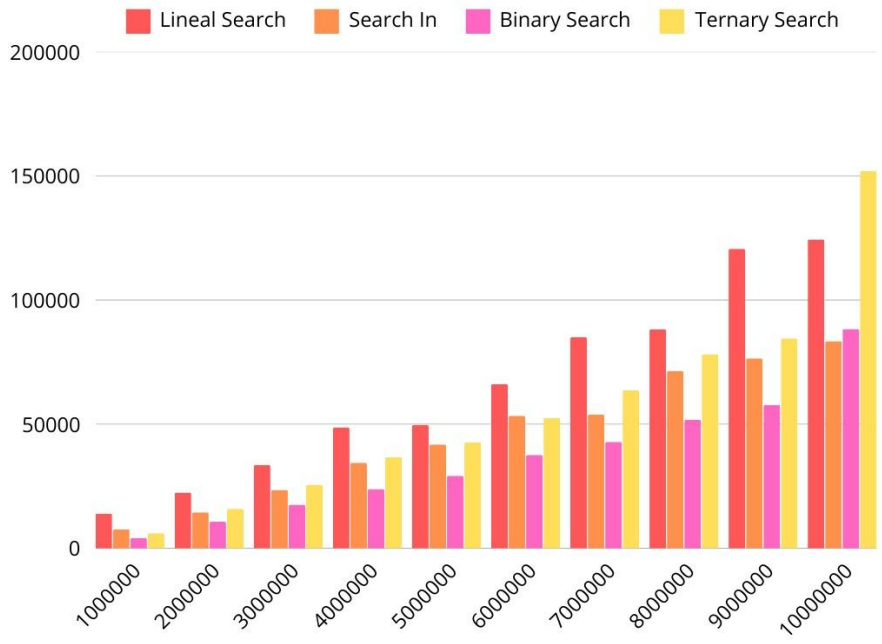
2.

- Minimum size 1.000.000
- Maximum size 10.000.000
- Step 1.000.000 examples per size
- MAX_VALUE = 1.000.000

```
Size | LinealSearch | Search In | Binary Search | Ternary Search
[1000000, 13966, 7666, 4140, 6100]
[2000000, 22460, 14458, 10679, 15854]
[3000000, 33540, 23537, 17577, 25556]
[4000000, 48679, 34430, 23841, 36709]
[5000000, 49727, 41833, 29201, 42646]
[6000000, 66163, 53371, 37602, 52440]
[7000000, 85055, 53928, 42835, 63655]
[8000000, 88189, 71415, 51803, 78134]
[9000000, 120602, 76521, 57709, 84572]
[10000000, 124425, 83438, 88355, 151969]
```

Linear search methods become inefficient and impractical for large datasets, as their execution time increases significantly with dataset size. Linear search, in particular, demonstrates its limited scalability, while the "in" operator, although slightly more efficient, still faces the same constraints.

For datasets reaching millions of elements, binary search remains the best option, as its execution time grows minimally due to its logarithmic complexity. In contrast, ternary search, despite having a similar complexity, introduces unnecessary overhead by dividing the search space into three parts instead of two, making it the worst-performing method. This inefficiency becomes even more evident with the sharp increase in execution time when the dataset size reaches 10,000,000 elements.



3.

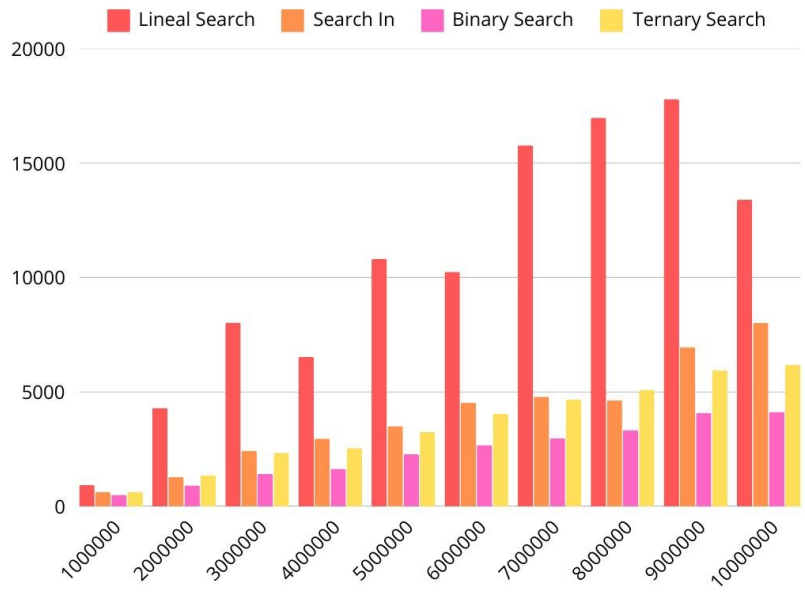
- Minimum size 1.000.000
- Maximum size 10.000.000
- Step 1.000.000 examples per size
- MAX_VALUE = 100

```
Size | LinealSearch | Search In | Binary Search | Ternary Search
[1000000, 931, 618, 500, 619]
[2000000, 4296, 1280, 915, 1356]
[3000000, 8028, 2428, 1424, 2338]
[4000000, 6533, 2960, 1643, 2549]
[5000000, 10806, 3506, 2275, 3250]
[6000000, 10249, 4533, 2674, 4040]
[7000000, 15769, 4787, 2972, 4666]
[8000000, 16981, 4639, 3336, 5092]
[9000000, 17792, 6955, 4081, 5935]
[10000000, 13393, 8018, 4123, 6187]
```

Reducing the maximum value to 100 significantly improved the performance of linear search algorithms and the "in" operator, as the number of unique values in the list decreased drastically, increasing the likelihood of finding the target in early positions. This explains the reduction in execution times compared to the previous results.

Despite this improvement, binary search remains the most efficient option, with controlled execution time growth and consistently lower values than the other methods. Ternary search, while better than linear searches, still performs worse than binary search due to the additional cost of dividing the search space into three parts instead of two.

This experiment demonstrates that the distribution of values within the dataset affects the performance of linear search algorithms, but even with this optimization, linear methods remain inefficient for large datasets, and binary search continues to be the best choice for scalability and efficiency.



Conclusion

The analysis of search algorithms confirms that the efficiency of each method varies significantly depending on the size and distribution of the data. Linear search methods, including sequential search and the "in" operator, become impractical for large datasets, as their execution time grows linearly with dataset size. Although reducing the value range improves performance, they remain inefficient compared to logarithmic approaches.

On the other hand, binary search proves to be the best option for large datasets, thanks to its logarithmic execution time growth, making it highly scalable and efficient. Ternary search, although also logarithmic, performs worse due to the additional cost of dividing the search space into three parts instead of two, introducing unnecessary overhead.

In conclusion, binary search is the most efficient algorithm for handling large volumes of data, while linear searches become inefficient, and ternary search does not provide significant improvements over binary search. These results emphasize the importance of choosing the right algorithm based on scalability and dataset size.