Amirkabir University of Technology

(Tehran Polytechnic)

Department of Industrial Engineering & Management Systems

Exploring Search Algorithms Through Visualization

By:

Pedram Peiro Asfia – 9825006

Hamed Araab – 9925003

Professor:

Dr. Marzieh Zarrinbal Masouleh

Course:

Algorithm Design Foundation

June 2023

Contents

1. Introduction 3
2. Linear Search Algorithm 3
    2.1. Implementation 4
    2.2. Time Complexity 4
3. Jump Search Algorithm 5
    3.1. Implementation 5
    3.2. Time Complexity 6
4. Binary Search Algorithm 6
    4.1. Implementation 7
    4.2. Time Complexity 8
5. Ternary Search Algorithm 8
    5.1. Implementation 8
    5.2. Time Complexity 9
6. Interpolation Search Algorithm 10
    6.1. Implementation 10
    6.2. Time Complexity 11
7. Exponential Search Algorithm 12
    7.1. Implementation 12
    7.2. Time Complexity 13
8. Fibonacci Search Algorithm 13
    8.1. Implementation 14
    8.2. Time Complexity 15
9. Uniform Binary Search Algorithm 15
    9.1. Implementation 16
    9.2. Time Complexity 17
10. Computing Time Complexity 17

1. Introduction

Search algorithms are an essential component of computer science and play a crucial role in various applications, ranging from databases to information retrieval systems. These algorithms enable us to efficiently locate and retrieve desired elements from a given collection of data. In this final project for the algorithm design course, we will explore and visualize the time complexity of several search algorithms, including Linear Search, Jump Search, Binary Search, Ternary Search, Interpolation Search, Exponential Search, Fibonacci Search, and Uniform Binary Search.

The primary objective of this project is to analyze the efficiency and performance of these search algorithms under different scenarios and input sizes. By visualizing their time complexities, we aim to provide a comprehensive understanding of how these algorithms behave in terms of their running time as the input data grows. This knowledge will allow us to make informed decisions regarding the selection and utilization of search algorithms in practical applications.

Through this project, we will not only gain insights into the theoretical aspects of these algorithms but also gain practical experience by implementing and testing them on various input arrays. By examining their time complexity plots, we can compare their efficiency and identify the most suitable algorithm for specific scenarios. This knowledge will be invaluable in optimizing search operations in real-world applications, where time efficiency is of utmost importance.

In the subsequent sections of this project, we will delve into the details of each algorithm, discussing their key concepts, working principles, and time complexity analyses. By the end of this project, we will have a comprehensive understanding of the strengths and weaknesses of each search algorithm, enabling us to make informed decisions when selecting the most appropriate algorithm for a given problem.

Overall, this project aims to provide a thorough exploration of various search algorithms, shedding light on their time complexity and helping us understand their behavior in different scenarios. With this knowledge in hand, we can make informed decisions when designing and implementing search operations in real-world applications, ultimately leading to improved efficiency and performance.

2. Linear Search Algorithm

Linear search is a simple search algorithm that sequentially checks each element in a list or array until the target element is found or the end of the list is reached. It starts from the beginning of the list and compares each element with the target element until a match is found or the entire list is traversed.

To perform a linear search:

- Start from the first element of the array.
- Compare the current element with the target element.
- If there is a match, return the index of the element.
- If no match is found after iterating through all elements, return -1 to indicate that the target element is not present in the array.

2.1. Implementation

The linear search algorithm can be implemented using a loop that iterates through each element of the array. Here is a step-by-step explanation of the implementation:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i

    return -1
```

- The function *linear_search* takes two arguments: *arr* (the array or list to search in) and `target` (the element to search for).
- It starts by looping through each index of the array using the *for* loop and the *range(len(arr))* statement.
- Inside the loop, it compares the element at the current index (*arr[i]*) with the target element.
- If there is a match (*arr[i] == target*), it returns the index *i* where the target element is found.
- If no match is found after iterating through all elements, the loop completes and the function returns -1 to indicate that the target element is not present in the array.

2.2. Time Complexity

The Linear Search algorithm has a time complexity of $O(n)$, where n represents the number of elements in the array. This complexity signifies that the time required by the algorithm increases proportionally to the size of the input.

In the worst-case scenario, the algorithm may need to iterate through all the elements in the array until it either discovers the target element or reaches the end of the array. Consequently, the time taken to execute the linear search grows linearly with the size of the array.

It is worth emphasizing that the best-case scenario occurs when the target element is found at the beginning of the array, resulting in a time complexity of $O(1)$. However, it is important to note that the worst-case time complexity of $O(n)$ dominates the overall complexity of the algorithm, as it represents the scenario where the target element is located at the very end of the array or is not present in the array at all.

When analyzing the efficiency of the Linear Search algorithm, we primarily focus on its worst-case time complexity. This allows us to evaluate its performance when dealing with large input sizes. As the number of elements increases, the time taken by the linear search grows linearly, making it less efficient compared to algorithms with better time complexities, such as binary search ($O(log\ n)$) or hash-based searches ($O(1)$ on average).

Therefore, while the Linear Search algorithm is simple and easy to implement, it may not be the most suitable choice for large datasets or situations where optimizing search time is crucial. It serves as a basic algorithmic building block and provides a starting point for more efficient search algorithms.

3. Jump Search Algorithm

Jump Search is a search algorithm designed to locate the position of a target element in a sorted array. It operates by dividing the array into blocks of a fixed size, often determined as the square root of the array size, and then performs a linear search within each block. If the target element is not found in the current block, the algorithm jumps to the next block and repeats the process. This iterative approach continues until the target element is found or the end of the array is reached.

The key advantage of Jump Search lies in leveraging the sorted nature of the array to determine the block where the target element might reside. By reducing the number of elements to search within each iteration, Jump Search can be more efficient than a simple linear search. It strikes a balance between the simplicity of linear search and the efficiency of more complex algorithms like Binary Search.

3.1. Implementation

The implementation of the jump search algorithm can be described as follows:

```
def jump_search(arr, target):
    n = len(arr)

    left = 0
    right = int(n**0.5) + 1

    while right <= n and arr[right - 1] < target:
        left = right
        right += int(n**0.5)

    for i in range(left, min(right, n)):
        if arr[i] == target:
            return i

    return -1
```

- The function *jump_search* takes two arguments: *arr* (the sorted array to search in) and *target* (the element to search for).
- It initializes the variables *n* (the length of the array), *left* (the starting index of the current block), and *right* (the ending index of the current block).
- Inside the while loop, it performs the following steps:
- Checks if the right pointer is within the array bounds and if the last element in the current block is less than the target element (*arr[right - 1] < target*).
- If the condition is satisfied, it updates the left pointer to the current right pointer and calculates the new right pointer by jumping a fixed size (*int(n**0.5)*).

- After determining the block where the target element may be located, it performs a linear search within that block by iterating over the range from *left* to the minimum of *right* and *n*.
- If the target element is found, it returns the index *i* where the element is located.
- If the target element is not found after searching all the blocks, the function returns -1 to indicate that the target element is not present in the array.

3.2. Time Complexity

The time complexity of the Jump Search algorithm is O(sqrt(n)), where n represents the number of elements in the array. This complexity indicates that the time required by the algorithm grows at a slower rate than linear search but is not as efficient as Binary Search.

The efficiency of Jump Search stems from the division of the array into blocks and performing a linear search within each block. This approach significantly reduces the total number of comparisons needed during the search process.

The choice of the block size plays a crucial role in determining the algorithm's efficiency. Typically, the block size is set as the square root of the array size. A smaller block size increases the number of blocks and reduces the number of comparisons within each block, resulting in faster search times. However, it also leads to more frequent jumps between blocks, incurring additional overhead. On the other hand, a larger block size decreases the number of jumps required but increases the number of comparisons within each block.

Overall, Jump Search is a valuable algorithm, particularly for searching in sorted arrays, especially those with a large size and a relatively uniform distribution of elements. It offers an intermediate level of efficiency between linear search and Binary Search. While it may not be the most efficient algorithm in all scenarios, Jump Search strikes a balance between simplicity and speed, making it a practical choice in certain situations. By understanding its principles and analyzing its time complexity, we can make informed decisions when choosing the most suitable search algorithm for a given problem.

4. Binary Search Algorithm

One of the most commonly used and efficient search algorithms is Binary Search. It is particularly beneficial when working with sorted arrays, as it drastically reduces the search space with each iteration. By repeatedly dividing the search space in half, Binary Search proves to be an excellent choice for handling large datasets.

The principle underlying Binary Search revolves around comparing the target element with the middle element of the sorted array. Initially, the algorithm considers the entire array as the search space, calculating the middle index and comparing the value at that index with the target element. Based on this comparison, there are three possible outcomes to consider:

1. If the target element is equal to the middle element, the search is successful, and the index of the middle element is returned, indicating the position of the target element in the array.

2. If the target element is less than the middle element, it suggests that the target element, if present in the array, must reside in the left half. Consequently, the algorithm updates the search space to the left half of the array and repeats the process.
3. If the target element is greater than the middle element, it implies that the target element, if present, must lie in the right half. The algorithm then updates the search space to the right half of the array and repeats the process.

This process of halving the search space and narrowing down the possibilities continues until the target element is found or until the search space is exhausted, indicating that the element is not present in the array.

4.1. Implementation

The implementation of the binary search algorithm can be described as follows:

```
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] < target:
            left = mid + 1
        elif arr[mid] > target:
            right = mid - 1
        else:
            return mid

    return -1
```

- The function *binary_search* takes two arguments: *arr* (the sorted array to search in) and *target* (the element to search for).
- It initializes the variables *left* and *right* to represent the boundaries of the search space.
- Inside the while loop, it performs the following steps:
    - Calculates the middle index of the search space using the formula *mid = (left + right) // 2*.
    - Compares the target element with the middle element of the array using *arr[mid]*.
    - If the middle element is less than the target, it updates the *left* pointer to *mid + 1* to search in the right half of the array.
    - If the middle element is greater than the target, it updates the *right* pointer to *mid - 1* to search in the left half of the array.
    - If the middle element is equal to the target, it returns the index *mid*, indicating that the target element has been found.
- If the while loop completes without finding the target element, the function returns -1 to indicate that the target element is not present in the array.

4.2. Time Complexity

Binary Search exhibits an order, or time complexity, of $O(log\ n)$, where n represents the number of elements in the array. This time complexity highlights the high efficiency of Binary Search when searching in sorted arrays.

The efficiency of Binary Search stems from its repeated division of the search space in half, effectively reducing the number of elements to examine with each iteration. This logarithmic behavior enables Binary Search to efficiently locate the target element, even when dealing with large arrays.

It is crucial to note that Binary Search requires the array to be sorted in order to function correctly. If the array is not sorted, Binary Search may not produce accurate results. Additionally, Binary Search can be applied not only to arrays but also to lists, as long as the elements are arranged in a sorted order.

Overall, Binary Search stands as a fundamental and widely used search algorithm due to its efficiency and simplicity. By understanding its principles and time complexity, we can leverage Binary Search to efficiently search for target elements in sorted arrays or lists, making it a valuable tool in various domains of computer science and beyond.

5. Ternary Search Algorithm

Ternary Search is a searching algorithm specifically designed to locate the position of a target element in a sorted array. It operates by dividing the array into three parts, allowing for a more efficient search process.

The algorithm compares the target element with the values at two points in the array, which are calculated using two midpoints. If the target element matches either of these midpoints, the search is successful. If the target element is smaller than the first midpoint, the algorithm narrows the search range to the left third of the array. On the other hand, if the target element is larger than the second midpoint, the algorithm narrows the search range to the right third of the array. If the target element falls between the two midpoints, the search range is narrowed to the middle third of the array. This iterative process continues until the target element is found or the search range becomes empty.

5.1. Implementation

The implementation of the ternary search algorithm can be described as follows:

```
def ternary_search(arr, target):
    left = 0
    right = len(arr) - 1
```

```
    while left <= right:
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        if arr[mid1] == target:
            return mid1
        elif arr[mid2] == target:
            return mid2
        elif target < arr[mid1]:
            right = mid1 - 1
        elif target > arr[mid2]:
            left = mid2 + 1
        else:
            left = mid1 + 1
            right = mid2 - 1

    return -1
```

- The function *ternary_search* takes two arguments: *arr* (the sorted array to search in) and *target* (the element to search for).
- It initializes the variables *left* and *right* to the start and end indices of the array, respectively.
- Inside the while loop, it performs the following steps:
    - Calculates two midpoints, *mid1* and *mid2*, using the formulas *left + (right - left) // 3* and *right - (right - left) // 3*, respectively.
    - Checks if either of the midpoints equals the target element. If so, it returns the index of the midpoint where the target element is found.
    - Compares the target element with the elements at the midpoints and adjusts the search range accordingly:
        - If the target element is smaller than the element at *mid1*, it updates the *right* pointer to *mid1 - 1* to search in the left third of the array.
        - If the target element is larger than the element at *mid2*, it updates the *left* pointer to *mid2 + 1* to search in the right third of the array.
        - If the target element is between the elements at *mid1* and *mid2*, it updates the *left* pointer to *mid1 + 1* and the *right* pointer to *mid2 - 1* to search in the middle third of the array.
- If the target element is not found after exhausting the search range, the function returns -1 to indicate that the target element is not present in the array.

5.2. Time Complexity

The time complexity of the Ternary Search algorithm is $O(\log_3 n)$, where n represents the number of elements in the array. This complexity indicates that the time required by the

algorithm increases at a slower rate compared to a simple linear search, but it is slightly slower than binary search.

Ternary Search achieves its efficiency by dividing the search range into three parts and progressively narrowing down the search space based on the comparison results. With each iteration, the size of the search range decreases by a factor of 3. This reduction in search space significantly reduces the number of comparisons required to find the target element.

It is important to note that the efficiency of Ternary Search is influenced by the distribution of elements within the array. When the elements are uniformly distributed, Ternary Search can provide efficient search times. However, if the elements are skewed towards one end of the array, it may require more comparisons and become less efficient.

Overall, Ternary Search proves to be a valuable algorithm for searching in sorted arrays, particularly when the number of elements is large and evenly distributed. By understanding its time complexity and considering the distribution of elements, we can effectively leverage Ternary Search to optimize search operations in various scenarios.

6. Interpolation Search Algorithm

Interpolation Search is a search algorithm designed to efficiently locate a target element in a sorted array that has uniformly distributed values. It improves upon the efficiency of a linear search by leveraging the values of the elements to estimate the probable position of the target element.

The algorithm operates by calculating the probable position of the target element using an interpolation formula. It assumes that the array is uniformly distributed, meaning that the difference between consecutive elements is approximately constant. Based on this estimation, the algorithm compares the target element with the element at the probable position. Depending on the result of the comparison, the search range is updated by moving the low and high pointers accordingly. This iterative process continues until the target element is found or the search range becomes invalid.

6.1. Implementation

The implementation of the interpolation search algorithm can be described as follows:

```
def interpolation_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high and target >= arr[low] and target <= arr[high]:
        if low == high:
            if arr[low] == target:
                return low
            else:
                return -1
```

```
        pos = low + ((target - arr[low]) * (high - low)) // (arr[high] -
arr[low])

        if arr[pos] < target:
            low = pos + 1
        elif arr[pos] > target:
            high = pos - 1
        else:
            return pos

    return -1
```

- The function *interpolation_search* takes two arguments: *arr* (the sorted array to search in) and *target* (the element to search for).
- It initializes the low pointer to the starting index of the array (0) and the high pointer to the last index of the array (*len(arr) - 1*).
- Inside the while loop, it performs the following checks:
    - It checks if the target element is within the current search range (*target >= arr[low] and target <= arr[high]*). If not, it exits the loop and returns -1 to indicate that the target element is not present in the array.
    - It checks if the low and high pointers are pointing to the same element (*low == high*). If so, it checks if that element is equal to the target element. If there is a match, it returns the index *low*; otherwise, it returns -1.
- If the above conditions are not met, it calculates the probable position of the target element using the interpolation formula: *pos = low + ((target - arr[low]) * (high - low)) // (arr[high] - arr[low]*).
- It then compares the target element with the element at the probable position (*arr[pos]*). Depending on the comparison result, it updates the low and high pointers accordingly.

6.2. Time Complexity

The time complexity of the Interpolation Search algorithm varies depending on the distribution of values within the array. In average and best-case scenarios, the time complexity approaches $O(\log \log n)$, indicating a highly efficient search algorithm for uniformly distributed data.

However, in the worst-case scenario where the values are not uniformly distributed, the time complexity can degrade to $O(n)$, similar to a linear search. It is important to consider this potential performance degradation when applying Interpolation Search to arrays with non-uniform distributions.

Overall, Interpolation Search exhibits an average time complexity of $O(\log \log n)$, which is advantageous for uniformly distributed data. By understanding its time complexity characteristics and the impact of data distribution, we can appropriately evaluate and utilize Interpolation Search in various search scenarios.

7. Exponential Search Algorithm

Exponential Search is a search algorithm designed to find the position of a target element in a sorted array. It utilizes a combination of exponential positioning and binary search techniques to efficiently locate the desired element.

The algorithm begins by comparing the target element with the first element of the array. If a match is found, the search is successful. Otherwise, the algorithm exponentially increases the position by a power of two and repeats the comparison process. This exponential positioning allows the algorithm to rapidly determine a range where the target element may be present. Once this range is identified, a binary search is performed within that range to precisely locate the element.

7.1. Implementation

The implementation of the exponential search algorithm can be described as follows:

```
def exponential_search(arr, target):
    n = len(arr)
    i = 1

    while i <= n and arr[i - 1] < target:
        i *= 2

    result = binary_search(arr[i // 2 : i], target)

    if result == -1:
        return -1
    else:
        return result + i // 2
```

- The function *exponential_search* takes two arguments: *arr* (the sorted array to search in) and *target* (the element to search for).
- It initializes the variables *n* (the length of the array) and *i* (the initial position to start the comparison).
- Inside the while loop, it performs the following steps:
    - Checks if the current position is within the array bounds and if the element at that position is less than the target element (*arr[i - 1] < target*).
    - If the condition is satisfied, it doubles the position (*i *= 2*) and continues to the next iteration.
- After determining the range where the target element might be present, it performs a binary search within that range using the *binary_search* function.
- If the binary search returns -1, indicating that the target element was not found, the function returns -1.
- Otherwise, it returns the index of the target element by adding the result of the binary search to *i // 2*.

7.2. Time Complexity

The time complexity of the Exponential Search algorithm is $O(\log n)$, where n represents the number of elements in the array. This complexity indicates that the algorithm's time requirements increase logarithmically with the size of the array, making it more efficient than a simple linear search and comparable to binary search.

Exponential Search achieves its efficiency by utilizing an exponential increase in the position, which helps identify a range where the target element is likely to be present. By rapidly narrowing down the search space, the algorithm reduces the number of comparisons required. Once the range is established, a binary search is employed within that range, further decreasing the number of comparisons and improving efficiency.

It is important to note that Exponential Search assumes the array to be sorted. Additionally, the efficiency of the algorithm can be influenced by the distribution of elements within the array. If the target element is closer to the beginning of the array, Exponential Search can find it more quickly. Conversely, if the target element is closer to the end, the algorithm may require more iterations to locate it.

Overall, Exponential Search combines the benefits of linear search and binary search, offering an efficient approach for searching in sorted arrays. By considering its time complexity and accounting for element distribution, Exponential Search can be effectively applied in various search scenarios.

8. Fibonacci Search Algorithm

Fibonacci Search is an advanced search algorithm specifically designed to efficiently locate the position of a target element within a sorted array. It improves upon binary search by dividing the array into subarrays using Fibonacci numbers, resulting in a more balanced and optimized search process.

The algorithm begins by generating a Fibonacci sequence until a value that is either equal to or slightly greater than the size of the array is obtained. These Fibonacci numbers are then utilized to determine the sizes of the subarrays for comparison. By using these non-uniform subarray sizes, Fibonacci search reduces redundant comparisons and optimizes the search process.

The steps of the Fibonacci search algorithm can be summarized as follows:

1. Generate a Fibonacci sequence until a value equal to or greater than the size of the array is obtained.
2. Compare the target element with the element in a subarray based on the Fibonacci numbers.
3. Based on the comparison result, update the Fibonacci numbers and adjust the subarray range.
4. Repeat the above steps until the target element is found or the search range becomes invalid.

8.1. Implementation

The implementation of the Fibonacci search algorithm can be described as follows:

```
def fibonacci_search(arr, target):
    n = len(arr)

    fib_prev = 0
    fib_curr = 1
    fib_next = fib_prev + fib_curr

    while fib_next < n:
        fib_prev = fib_curr
        fib_curr = fib_next
        fib_next = fib_prev + fib_curr

    offset = -1

    while fib_next > 1:
        i = min(offset + fib_prev, n - 1)

        if arr[i] < target:
            fib_next = fib_curr
            fib_curr = fib_prev
            fib_prev = fib_next - fib_curr
            offset = i
        elif arr[i] > target:
            fib_next = fib_prev
            fib_curr = fib_curr - fib_prev
            fib_prev = fib_next - fib_curr
        else:
            return i

    if n and fib_curr and arr[n - 1] == target:
        return n - 1

    return -1
```

- The function *fibonacci_search* takes two arguments: *arr* (the sorted array to search in) and *target* (the element to search for).
- It initializes the variables *fib_prev*, *fib_curr*, and *fib_next* to generate the Fibonacci sequence.
- It enters the first while loop to generate the Fibonacci sequence until *fib_next* is just greater than or equal to the size of the array.
- The algorithm then initializes the variable *offset* to -1 to keep track of the starting position of the subarray.

- The second while loop performs the search by comparing the target element with the element in the subarray.
- Inside the loop, it calculates the index i based on the current Fibonacci numbers and the offset.
- Depending on the comparison result, it updates the Fibonacci numbers and the offset to narrow down the search range.
- If a match is found, it returns the index i where the target element is located.
- If the loop finishes without finding a match, it checks for a special case where the last element of the array matches the target element.
- Finally, if no match is found, the function returns -1 to indicate that the target element is not present in the array.

8.2. Time Complexity

The time complexity, or order, of the Fibonacci Search algorithm is $O(\log n)$, where n represents the number of elements in the array. This time complexity implies that the algorithm's efficiency increases logarithmically with the size of the array. Thus, Fibonacci search offers a more efficient alternative to linear search and is comparable to binary search.

Fibonacci Search achieves its efficiency by dividing the array into non-uniform subarrays using Fibonacci numbers. This approach enables a balanced search process, minimizing the number of elements to be examined in each iteration. By eliminating redundant comparisons, Fibonacci search optimizes the search time and improves overall efficiency.

However, it is important to note that Fibonacci search assumes the array to be sorted. Furthermore, while Fibonacci search generally provides efficient results, its performance may be affected by the overhead involved in calculating Fibonacci numbers and updating the search range. In some cases, binary search may outperform Fibonacci search due to its simplicity.

Overall, Fibonacci search presents a balanced and efficient search algorithm, surpassing linear search and rivaling binary search in terms of efficiency. However, the choice between Fibonacci search and binary search should be based on the specific characteristics and requirements of the search scenario.

9. Uniform Binary Search Algorithm

Uniform Binary Search is a specialized version of the Binary Search algorithm that addresses the potential issue of suboptimal performance in scenarios where the target element is located at one extreme end of a large sorted array. This variant aims to mitigate the effect of a skewed distribution of elements by introducing a randomization factor during the search process.

The algorithm operates similarly to the traditional Binary Search, where the search range is repeatedly divided in half. However, instead of selecting the middle element as the midpoint, Uniform Binary Search randomly selects a midpoint within the search range. This randomization

helps in balancing the search process and reduces the impact of skewed distributions, ensuring a more uniform exploration of the array.

By introducing randomness into the selection of the midpoint, Uniform Binary Search aims to alleviate the worst-case performance scenarios that can occur with skewed distributions. It provides a more robust and efficient search algorithm for sorted arrays with uneven element distributions.

9.1. Implementation

The implementation of Uniform Binary Search can be described as follows:

```
def uniform_binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = random.randint(low, high)

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

- The function *uniform_binary_search* takes two arguments: *arr* (the sorted array to search in) and *target* (the element to search for).
- It initializes the variables *low* and *high* to the start and end indices of the array, respectively.
- Inside the while loop, it performs the following steps:
    - Generates a random midpoint *mid* within the range of *low* to *high* using the *random.randint()* function. This random selection aims to minimize the chances of consistently choosing the same skewed midpoint.
    - Compares the element at *mid* with the target element:
        - If they are equal, it returns the index *mid*, indicating that the target element is found.
        - If the element at *mid* is less than the target element, it updates *low* to *mid + 1* to search in the upper half of the remaining range.
        - If the element at *mid* is greater than the target element, it updates *high* to *mid - 1* to search in the lower half of the remaining range.
- If the target element is not found after exhausting the search range, the function returns -1 to indicate that the target element is not present in the array.

9.2. Time Complexity

The time complexity, or order, of the Uniform Binary Search algorithm is O(log n), where n represents the number of elements in the array. Similar to the traditional Binary Search algorithm, Uniform Binary Search achieves efficient search by dividing the search range in half at each iteration.

However, what sets Uniform Binary Search apart is its additional randomization factor during the midpoint selection process. By randomly choosing the midpoint within the search range, it aims to address the potential issue of worst-case performance when the distribution of elements is skewed. This randomization helps balance the search process and ensures a more uniform exploration of the array, leading to improved efficiency.

Overall, the Uniform Binary Search algorithm provides the same logarithmic time complexity as the traditional Binary Search while offering enhanced performance in scenarios with skewed element distributions. It represents an effective approach to mitigate the impact of unevenly distributed elements during the search process.

10. Computing Time Complexity

In general, the code generates a random array of a specific size and after that, it will use the benchmarking algorithm *iterations* times and take average of the time consumed and save it in the dictionary. This procedure is done for different array sizes.

```
start_size = 1
end_size = 30000
step = 5
iterations = 1000

average_times = {search_fn: [] for search_fn in search_fns}

arr = random.sample(range(0, start_size * 10), start_size)

arr.sort()

while len(arr) <= end_size:
    for search_fn in search_fns:
        # Warmup
        for _ in range(iterations // 10):
            search_fn(arr, random.choice(arr))

        total_time = timeit.timeit(
            lambda: search_fn(arr, random.choice(arr)), number=iterations
        )

        average_time = total_time / iterations * 1e6
```

```
        average_times[search_fn].append(average_time)

    extension = random.sample(range(arr[-1], arr[-1] + step * 10), step)

    extension.sort()

    arr += extension
```

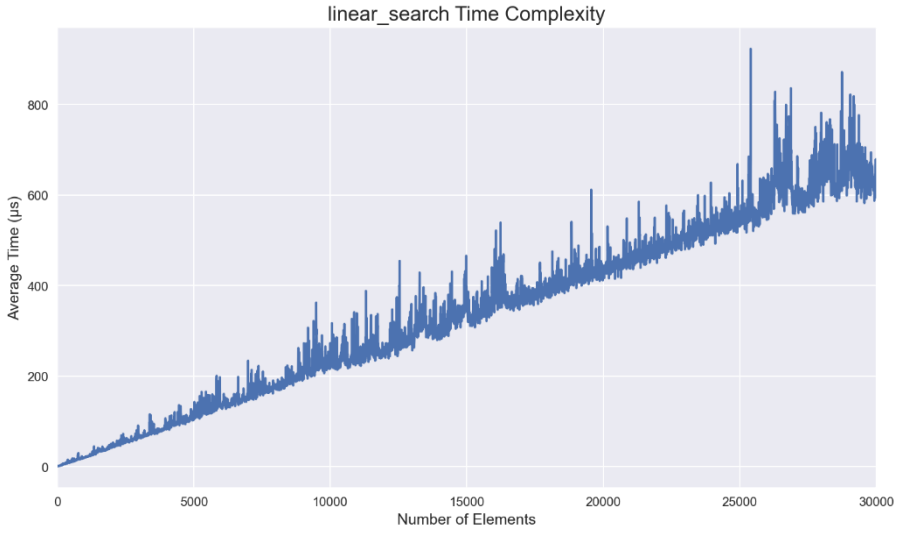
- The variables *start_size*, *end_size*, *step*, and *iterations* are initialized to control the parameters of the benchmark. *start_size* represents the initial size of the array, *end_size* represents the final size of the array, *step* determines how much the array size increases in each iteration, and *iterations* determines the number of times each search algorithm is executed and timed.
- The dictionary *average_times* is created to store the average times for each search algorithm. It uses a dictionary comprehension to initialize an empty list for each search algorithm present in the *search_fns* list.
- An array *arr* is generated by sampling random numbers from the range 0 to *start_size * 10*. It is then sorted in ascending order to ensure it is a sorted array, which is a requirement for most of the search algorithms.
- The code enters a loop that continues until the length of *arr* reaches or exceeds *end_size*. This loop is responsible for increasing the size of the array in each iteration.
- Within the loop, there is another loop that iterates over each search algorithm in the *search_fns* list.
- The inner loop performs a warm-up phase for each search algorithm. It executes the search function a fraction of *iterations* times (specifically *iterations // 10*) with a randomly chosen element from *arr*. This warm-up phase allows the search algorithm to "warm up" and potentially optimize its internal structures or cache data.
- After the warm-up phase, the search algorithm is executed *iterations* times using *timeit.timeit()*. This function measures the execution time of the given search function and calculates the total time taken.
- The total time is then divided by *iterations* and multiplied by 1 million (*1e6*) to obtain the average time per search operation in microseconds (*average_time*).
- The *average_time* is appended to the corresponding list in the *average_times* dictionary for the current search algorithm.

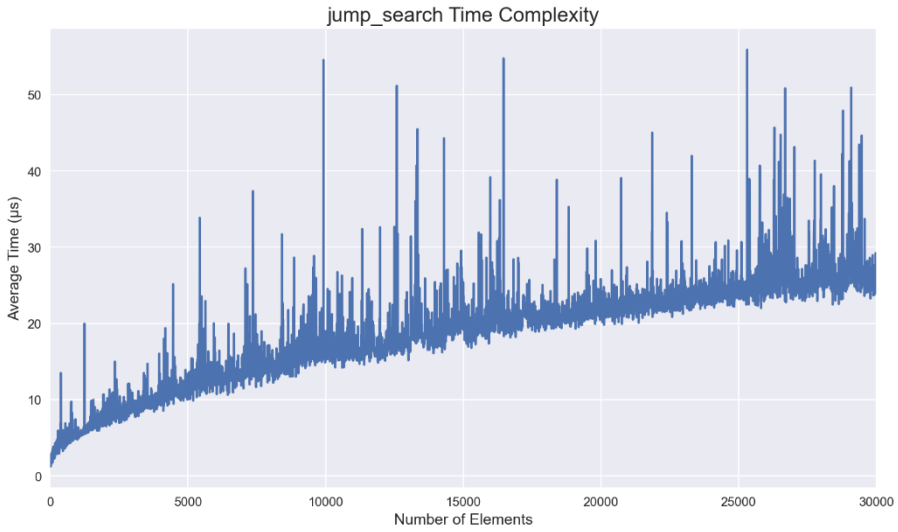
- Following the inner loop, an extension array is created by sampling random numbers from the range starting at the last element of **arr** and extending by **step * 10** elements. The extension array is then sorted to maintain the sorted property of **arr**.
- The extension array is concatenated with **arr**, increasing its size for the next iteration of the loop.

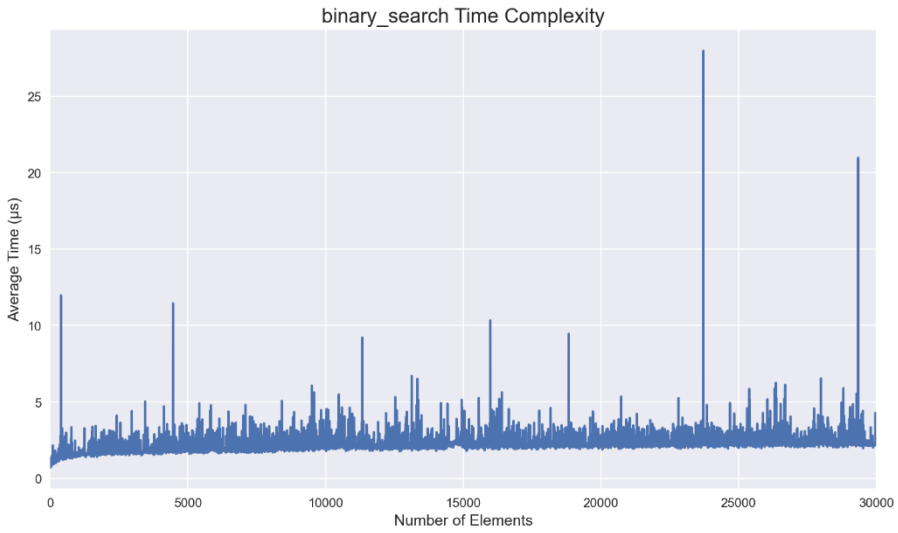
Now it's time for visualizing the results:

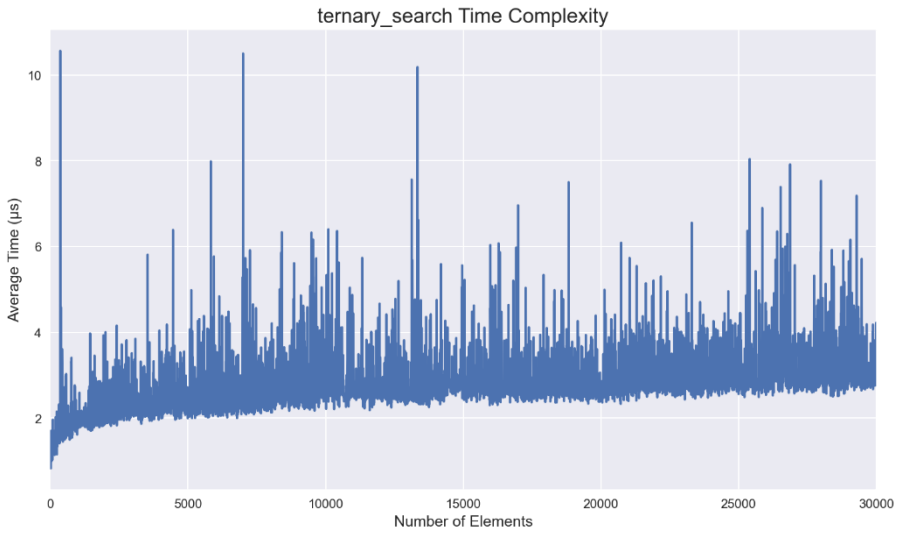
```
# Set plot style
plt.style.use("seaborn")

# Plot the time complexity for each search algorithm
for search_fn in search_fns:
    arr_size = range(start_size, end_size + 1, step)
    avg_times = average_times[search_fn]

    plt.figure(figsize=(10, 6), dpi=150)

    plt.plot(arr_size, avg_times, markersize=5, label=search_fn.__name__)

    plt.title(f"{search_fn.__name__} Time Complexity", fontsize=16)
    plt.xlabel("Number of Elements", fontsize=12)
    plt.ylabel("Average Time (µs)", fontsize=12)
    plt.xlim([0, end_size])

    plt.grid(True)
    plt.tight_layout()

    plt.show()
```

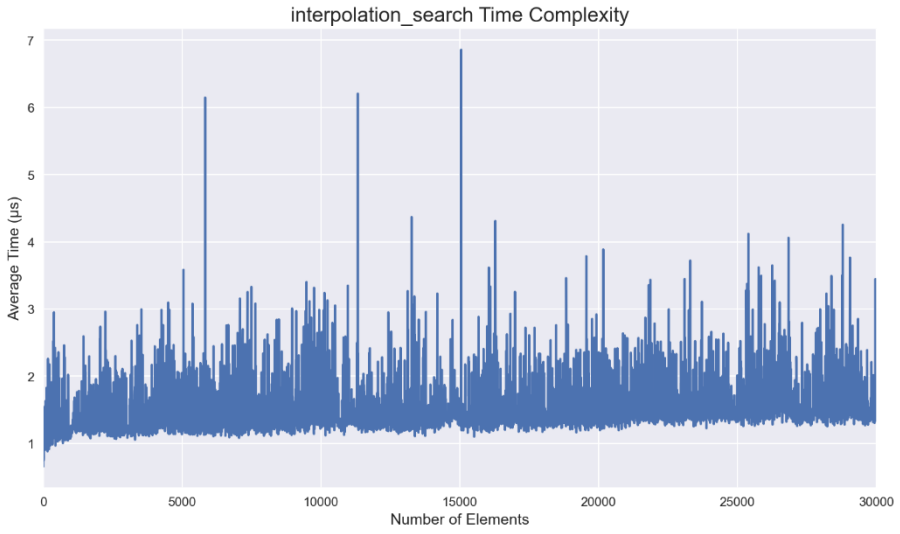
The following charts, however, are expected to be noisy due to the randomness of choosing the targets.

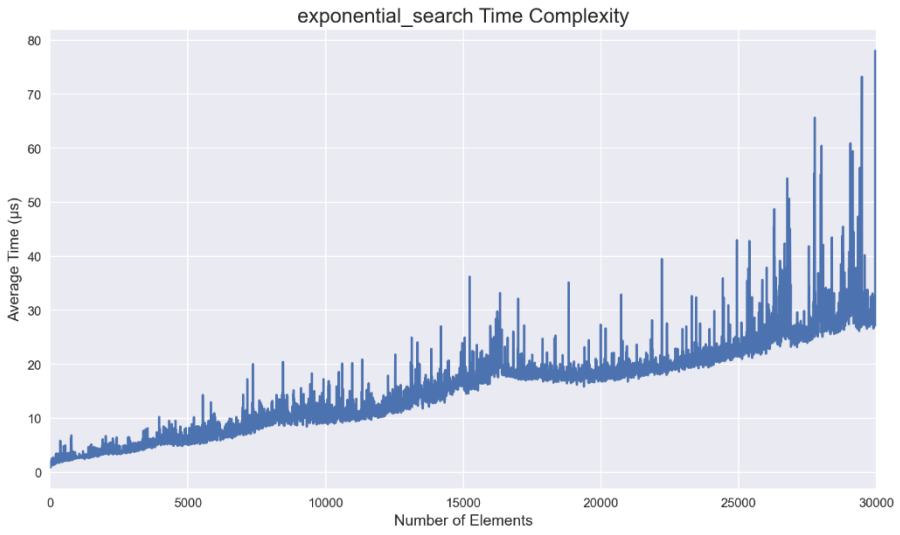

linear_search Time Complexity
Average Time (µs)
0
200
400
600
800
0
5000
10000
15000
20000
25000
30000
Number of Elements

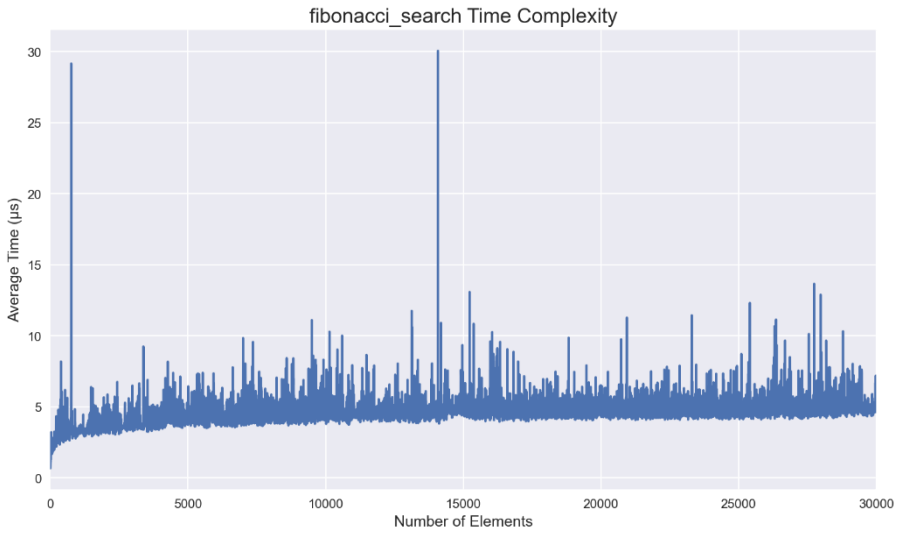

jump_search Time Complexity
Average Time (µs)
0
10
20
30
40
50
0
5000
10000
15000
20000
25000
30000
Number of Elements

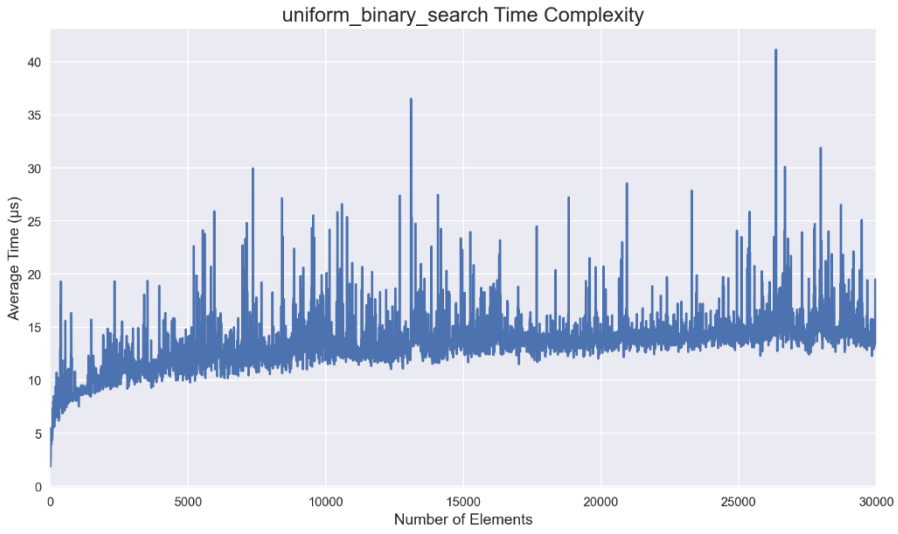

binary_search Time Complexity
Average Time (µs)
0
5
10
15
20
25
0
5000
10000
15000
20000
25000
30000
Number of Elements


ternary_search Time Complexity
Average Time (µs)
Number of Elements
0
5000
10000
15000
20000
25000
30000
2
4
6
8
10


interpolation_search Time Complexity
Average Time (µs)
Number of Elements
0
5000
10000
15000
20000
25000
30000
1
2
3
4
5
6
7


exponential_search Time Complexity
Average Time (µs)
Number of Elements
0
5000
10000
15000
20000
25000
30000
0
10
20
30
40
50
60
70
80





To smooth out the plots, the following code is implemented:

```
def plot_smoothed_results(time_complexities):
    # Set plot style
    plt.style.use("seaborn")

    # Create the figure and axes
    _, ax = plt.subplots(figsize=(13, 6), dpi=150)

    # Set the title and axis labels
    ax.set_title("Time Complexity", fontsize=16)
    ax.set_xlabel("Number of Elements", fontsize=12)
    ax.set_ylabel("Average Time (µs)", fontsize=12)
```

```
    # Plot the time complexity for each search algorithm
    for search_fn, time_complexity in time_complexities.items():
        average_time = average_times[search_fn]

        # Apply Savitzky-Golay filter for smoothing
        smoothed_average_time = savgol_filter(average_time, len(average_time), 2)

        # Plot the smoothed line
        (line,) = ax.plot(arr_size, smoothed_average_time,
label=f"{search_fn.__name__}: {time_complexity}")

        # Add the time complexity as text annotation to the line
        x_pos = arr_size[-1] + 50  # Adjust the position of the text
        y_pos = smoothed_average_time[-1]
        ax.text(x_pos, y_pos, time_complexity, fontsize=10,
color=line.get_color())

    # Add legend
    ax.legend()
    ax.set_xlim(left = 0)
    # Show the plot
    plt.tight_layout()
    plt.show()


plot_smoothed_results(
    {
        linear_search: "O(n)",
        jump_search: "O(sqrt(n))",
        binary_search: "O(log(n))",
        ternary_search: "O(log(n))",
        interpolation_search: "O(log(log(n)))",
        exponential_search: "O(log(n))",
        fibonacci_search: "O(log(n))",
        uniform_binary_search: "O(log(n))",
    }
)
```

In the code provided, after collecting the average execution times for each search function and array size combination, the data is further processed to improve its visual representation. Specifically, a Savitzky-Golay filter is applied to smooth the average time data.

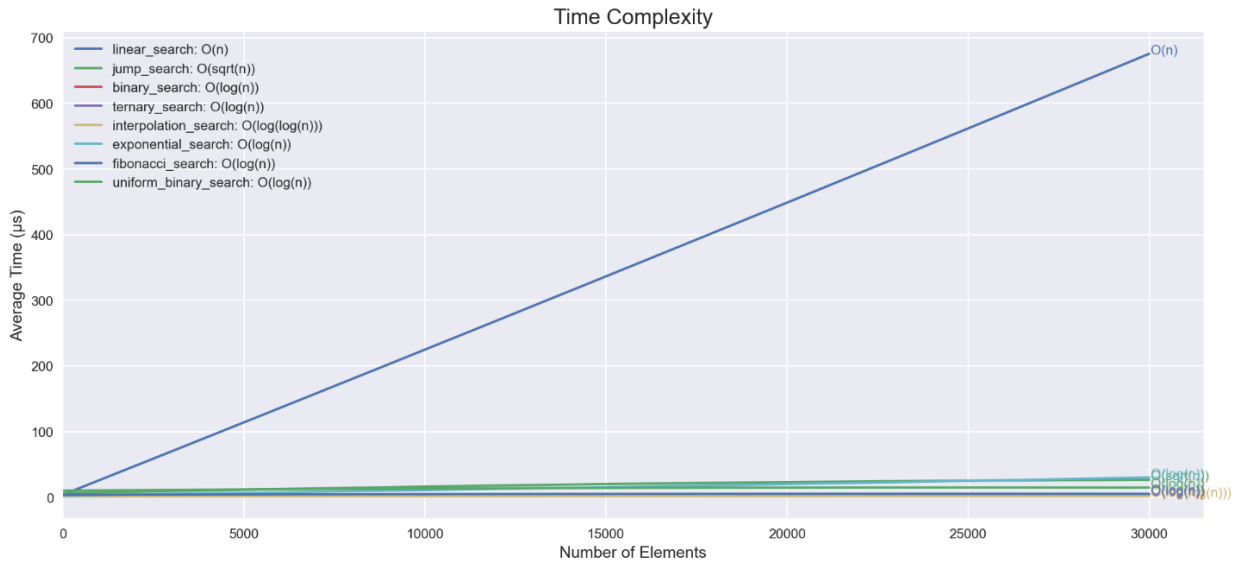
The Savitzky-Golay filter is a digital filter commonly used for noise reduction and smoothing in time series data. It helps eliminate noise and variability in the measurements, enabling a clearer understanding of underlying trends.

To apply the Savitzky-Golay filter, the following parameters are utilized:

- *window_length*: The size of the window used for smoothing. In this case, the window length is set to *len(average_time)*, which means the entire length of the average time data is considered as the window size. By using the complete range of data points, the filter captures the overall trends more accurately.
- *polyorder*: The order of the polynomial used in the filter. A higher *polyorder* value allows the filter to capture more complex trends in the data. In this scenario, a *polyorder* of 2 is chosen, indicating that a second-degree polynomial is used for smoothing.

By applying the Savitzky-Golay filter with these parameters, the average time data is processed to produce a smoothed representation. This smoothed data helps reveal the underlying performance trends of each search function as the array size increases.

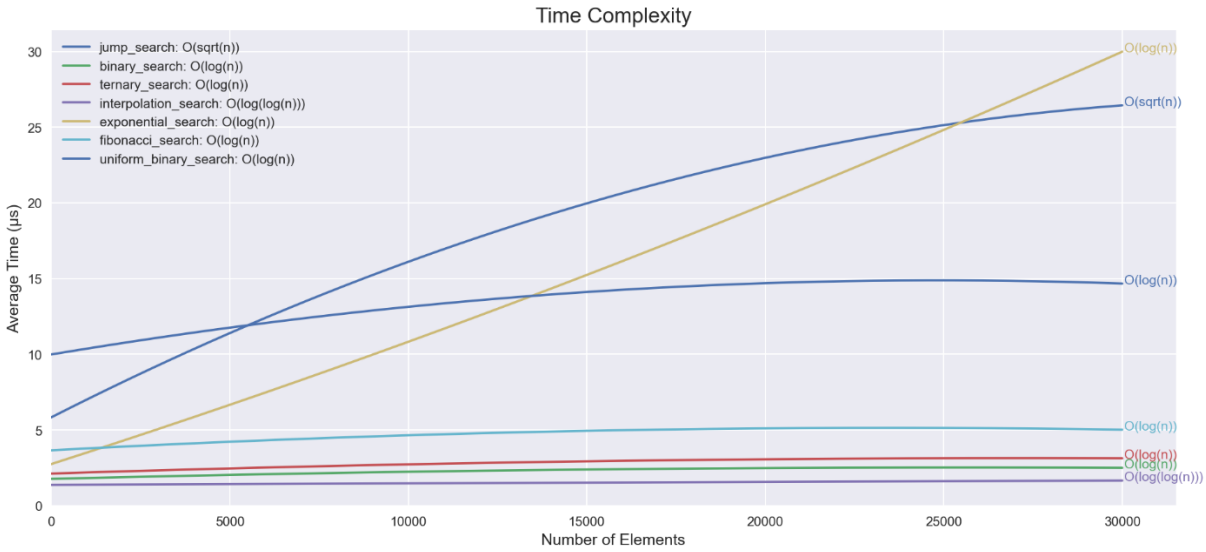
The resulting smoothed average time data is then plotted, providing a visually appealing representation of the relative efficiency and performance of the different search algorithms. It allows for a clearer understanding of how the execution time varies with the size of the array for each search function.



In order to prioritize algorithms with similar time complexity, we will exclude *linear_search* from the visualization.

```
plot_smoothed_results(
    {
        jump_search: "O(sqrt(n))",
        binary_search: "O(log(n))",
        ternary_search: "O(log(n))",
        interpolation_search: "O(log(log(n)))",
        exponential_search: "O(log(n))",
```

```
        fibonacci_search: "O(log(n))",
        uniform_binary_search: "O(log(n))",
    }
)
```



As we can see in the chart above, Linear Search has the worst performance while Ternary Search, Binary Search, and Interpolation Search have performed the best.