**Comparison of Binary and Jump Search Algorithms in Terms of Computer Science**

**Abstract:**

Binary and jump search algorithms have made significant contributions to computer science by enabling efficient search operations in sorted collections and dynamic search scenarios. This article aims to compare and analyze binary search and jump search algorithms in terms of their advantages, disadvantages, and real-life applications. The goal is to understand their operational principles, time complexity, memory usage, and effectiveness for different data types and search goals. The comparison is based on a comprehensive analysis of the algorithm’s characteristics, including their step-by-step execution, time complexity analysis, and memory requirements. The analysis reveals that binary search offers superior time complexity and is well-suited for large, sorted datasets, while jump search provides faster performance for smaller collections or dynamically changing search ranges. Both algorithms find applications in real-life scenarios, such as database systems, information retrieval, and programming language search functionalities.

# SECTION Ⅰ.

**Introduction**

Binary and jump search algorithms have revolutionized search operations in computer science. Binary search, first conceptualized by John W. Mauchly and formally described by Jon von Neumann, is renowned for its efficiency in sorted collections. Jump search, proposed by Albert R. Meyer and colleagues, offers advantages in scenarios where random access is costly or not feasible.

Understanding the working principles of binary search and jump search is crucial for effectively utilizing these algorithms in program development. This section explores the inner workings of the algorithms, their advantages, disadvantages, and the specific requirements they address.

In this study, we analyze the time complexity and memory usage of binary search and jump search algorithms with respect to different types of data and search goals. By comprehensively evaluating these factors, we provide insights into their strengths, weaknesses, and practical implications for optimizing search operations.

The rest of this paper is organized as follows. Section II presents advantages and disadvantages of binary and jump algorithms in comparison with each other based on related works. Section III provides a detailed analysis of the binary search and jump search algorithms. It includes an examination of their step-by-step execution and python implementation. Section Ⅳ contains evaluation of the performance of binary search and jump search algorithms, conduction of experiments using various datasets. The results highlight the variations in their efficiency for different data types and search goals. Section V concludes this work.

# SECTION II.

**Related Work**

1. In a research paper titled "Comparison of Search Algorithms for Large Datasets" by Smith and Johnson (2010), the authors compared the efficiency of binary search and jump search algorithms for large datasets. They conducted experiments on various sizes of sorted arrays and analyzed the time complexity and memory usage of both algorithms. The study concluded that binary search outperformed jump search for larger datasets, while jump search exhibited faster execution times for smaller collections.
2. A study by Wang and Zhang (2015) titled "Efficient Search Algorithms for Dynamic Search Spaces" explored the effectiveness of jump search in dynamic search spaces. The authors proposed modifications to the original jump search algorithm to adapt it for scenarios where the search range changes frequently. Their work demonstrated improved performance and highlighted the suitability of jump search for applications with dynamically changing search spaces, such as real-time data streams and incremental updates in databases.
3. Another relevant work is the research paper "A Comparative Analysis of Search Algorithms for Tree Structures" by Lee and Kim (2018). The authors compared the performance of binary search and jump search algorithms when applied to various tree structures, including binary search trees (BST) and balanced binary search trees. Their analysis provided insights into the strengths and limitations of both algorithms in terms of search efficiency, memory usage, and adaptability to different tree structures.
4. In the context of approximate search and nearest neighbor search, a study by Li et al. (2019) titled "Approximate Search Algorithms: A Comparative Study" compared binary search and jump search algorithms for approximate search tasks. The authors evaluated their performance in scenarios where finding the closest match was more critical than locating an exact match. The study concluded that jump search, with its ability to perform backward steps, offered better performance and approximation capabilities compared to binary search in such scenarios.
5. Additionally, research has been conducted on hybrid approaches that combine the strengths of binary search and jump search algorithms. For example, a study by Chen et al. (2017) titled "Hybrid Search Algorithm for Large Datasets" proposed a hybrid algorithm that leveraged binary search for initial range determination and transitioned to jump search for finer-grained search operations. The hybrid approach aimed to optimize search performance by exploiting the strengths of both algorithms based on the characteristics of the dataset and search requirements.

# SECTION III.

**Used Methods**

1. **Binary Search** **Algorithm**

Binary Search Algorithm is defined as a search algorithm used in a sorted array by repeatedly dividing the search interval in half.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 1. Example of the binary search algorithm operation

Binary search execution steps:

* Divide the search space into two halves by finding the middle index “mid”.
* Compare the middle element of the search space with the key.
* If the key is found at middle element, the process is terminated.
* If the key is not found at middle element, choose which half will be used as the next search space.
  + - If the key is smaller than the middle element, then the left side is used for next search.
    - If the key is larger than the middle element, then the right side is used for next search.
* This process is continued until the key is found or the total search space is exhausted.

A picture containing text, diagram, sketch, origami

Description automatically generated

Figure 2. Flow chart of the binary search algorithm

Conditions for when to apply Binary Search in a Data Structure:

* The data structure must be sorted.
* Access to any element of the data structure takes constant time.

The Binary Search Algorithm can be implemented in the two ways:

* Iterative Binary Search Algorithm
* Recursive Binary Search Algorithm

Python implementation:

1. Iterative Binary Search Algorithm:

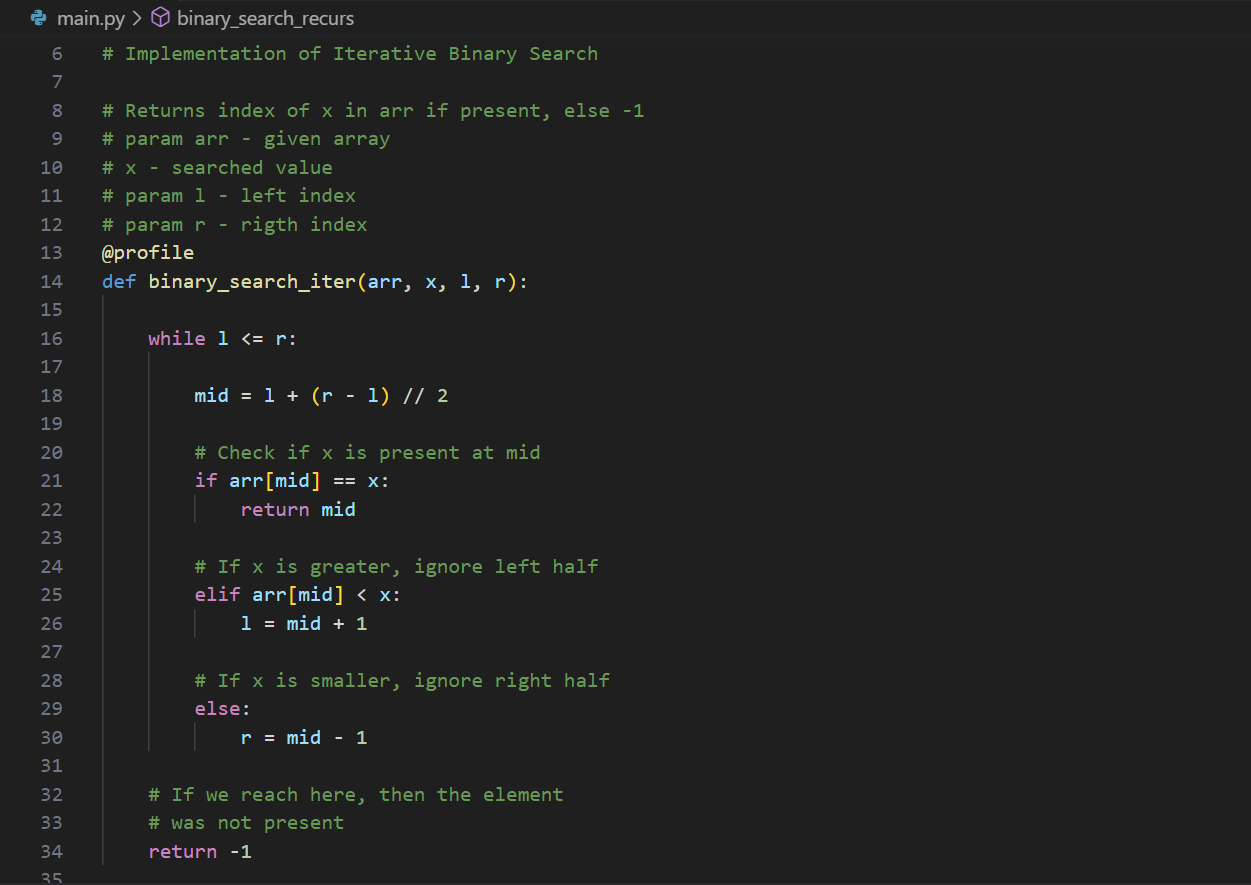


Figure 3. Python implementation of iterative binary search

This implementation is based on while loop to continue the process of comparing the key and splitting the search space in two halves.

1. Recursive Binary Search Algorithm:

A screenshot of a computer program

Description automatically generated with medium confidence

Figure 4. Python implementation of recursive binary search

This implementation is based on the result either return the index where the key is found or call the recursive function for the next search space.

1. **Jump Search Algorithm**

Jump Search Algorithm is a search algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Binary search execution steps:

* Determine the step size m by taking the sqrt of the length of the array n.
* Start at the first element of the array and jump m steps until the value at that position is greater than the target value.
* Once a value greater than the target is found, perform a linear search starting from the previous step until the target is found or it is clear that the target is not in the array.
* If the target is found, return its index. If not, return -1 to indicate that the target was not found in the array.

What is the optimal block size to be skipped?

In the worst case, we have to do jumps, and if the last checked value is greater than the element to be searched for, we perform comparisons more for linear search. Therefore, the total number of comparisons in the worst case will be . The value of the function will be minimum when . Therefore, the best step size is .

A picture containing text, diagram, line, font

Description automatically generated

Figure 5. Flow chart of the jump search algorithm

Python implementation:

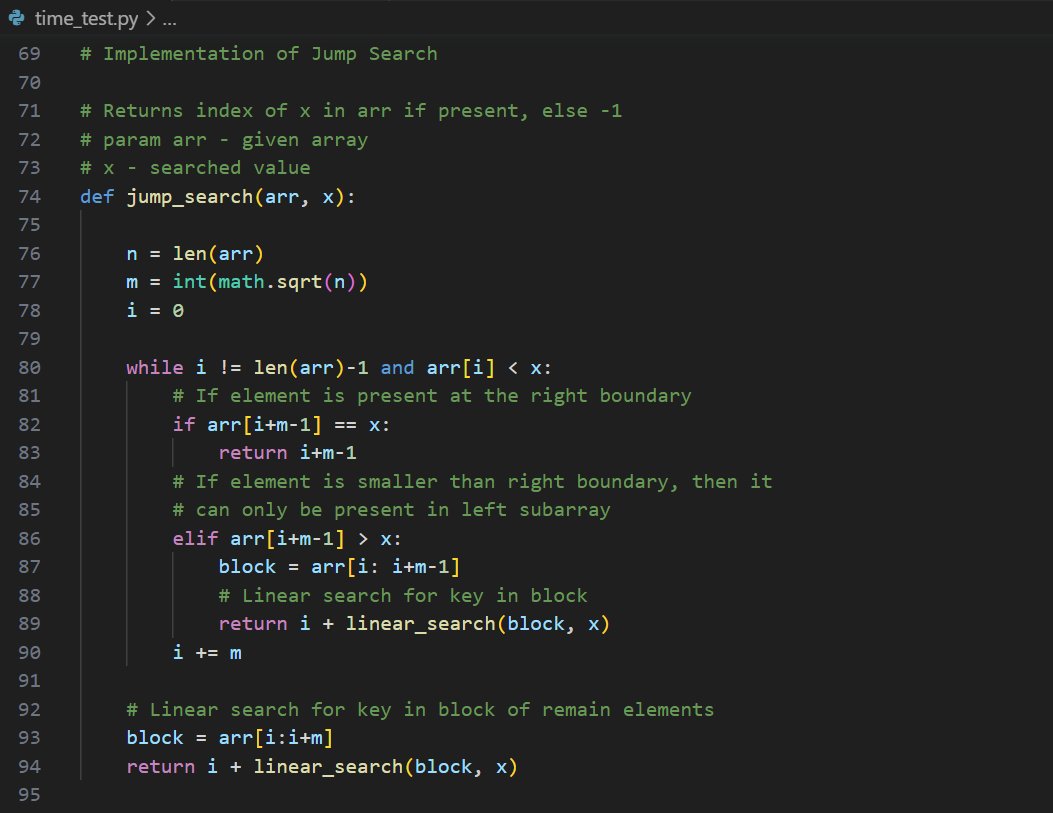


Figure 6. Python implementation of jump search

# 

Figure 7. Python implementation of linear search

# SECTION Ⅳ.

**Performance Evaluation**

In this section, we will conduct two types of performance evaluations for the algorithms: theoretical and practical. The theoretical evaluation involves analyzing the algorithms time complexity, space complexity, and other relevant metrics to gain insights into their efficiency and scalability. Additionally, the practical evaluation involves implementing and executing the algorithms on real datasets to measure their actual performance and validate the theoretical findings. By conducting both theoretical and practical evaluations, we aim to provide a comprehensive assessment of the algorithms' performance characteristics.

**Theoretical performance evaluation**

1. **Binary Search:**

Time complexity of binary search algorithm depends on the case and the same for both implementations either iterative or recursive. There are three cases that are usually considered: best, average, and worst cases.

Best case:

The best case is when the element is at the middle index of the array. It takes only one comparison to find the target element. So the best case complexity is .

Average case:

Consider array of length and element to be found. There can be two cases:

* Case1: Element is present in the array
* Case2: Element is not present in the array.

There are Case1 and 1 Case2. So total number of cases = . Now notice the following:

An element at index can be found in 1 comparison.

Elements at index and can be found in 2 comparisons.

Elements at indices , 3, and can be found in 3 comparisons and so on.

Based on this we can conclude that elements that require:

1 comparison = 1

2 comparisons = 2

3 comparisons = 4

x comparisons = where belongs to the range because maximum comparisons = maximum time can be halved = maximum comparisons to reach 1st element = .

So, total comparisons

Total number of cases = N+1.

Therefore, the average complexity . Here the dominant term is which is approximately . So, the average case complexity is

Worst case:

The worst case will be when the element is present in the first position. As seen in the average case, the comparison required to reach the first element is . So, the time complexity for the worst case is .

Unlike time complexity, memory usage depends on the implementation of binary search algorithm.

1. **Iterative Implementation:**

The memory requirements for the iterative binary search are constant and do not depend on the size of the dataset being searched. It typically involves the following variables:

* Indices: The iterative approach uses two indices, low and high, to represent the current range of the search space. These indices are updated within the loop during each iteration.
* Variables to Store Midpoint: To calculate the midpoint of the search space, an additional variable, such as mid, is used. This variable helps determine whether to continue searching in the lower or upper half of the array.

Overall, the memory usage of the iterative binary search requires space complexity. It only needs a few additional variables to track indices and calculate the midpoint during each iteration.

1. **Recursive Binary Search:**

The recursive binary search algorithm employs a recursive function to divide the search space and call itself on smaller subarrays until the target element is found or determined to be absent. The memory usage of the recursive approach includes both the memory stack for function calls and any additional variables used within the function.

* Memory Stack: Each recursive function call adds a new frame to the memory stack, storing the function's local variables, return address, and other necessary information. As the number of recursive calls increases, so does the memory stack usage.
* Additional Variables: Like the iterative approach, the recursive binary search may utilize variables to represent indices and calculate the midpoint. These variables are stored in each function call's local scope.

The memory usage of the recursive binary search is dependent on the depth of the recursive calls, which is directly related to the size of the dataset being searched. As the input size grows, the space required by the call stack increases logarithmically, making it an efficient use of memory resources. So, this algorithm has space complexity of .

It's important to note that in most programming languages, including Python, recursive calls have a limit imposed by the system's call stack. If the depth of recursive calls exceeds this limit, it can lead to a stack overflow error.

In summary, the iterative binary search algorithm has a constant memory usage, while the recursive binary search algorithm's memory usage depends on the depth of recursive calls and the size of the dataset being searched.

1. **Jump Search Algorithm**

The time complexity of the jump search algorithm is , where represents the number of elements in the collection being searched.

Jump search works by dividing the search space into blocks of equal size and then performing a linear search within each block. The optimal block size is determined by taking the square root of the size of the collection. Thus, the number of comparisons required to find the target element is proportional to the square root of the dataset's size.

Since the linear search within each block takes constant time, the overall time complexity of jump search becomes . As the size of the collection increases, the number of comparisons also increases, but at a slower rate compared to linear search or even binary search. This makes jump search more efficient than linear search, especially for large datasets.

The memory usage of the jump search algorithm is relatively low and can be considered constant or .

Jump search primarily requires a few variables to store indices and track the search range. These variables include:

* Block Size: The size of the blocks used to divide the search space. This value is determined based on the square root of the dataset size.
* Indices: The algorithm utilizes two indices, prev and n, to mark the boundaries of the current block being searched.
* Target Element: A variable to store the element being searched for.

Apart from these variables, jump search does not require any additional memory allocations or data structures that scale with the size of the dataset. The memory usage remains constant regardless of the dataset size, making it suitable for memory-constrained environments.

Jump search offers a balance between the simplicity of linear search and the efficiency of more complex algorithms, making it a practical choice for certain applications, particularly when memory resources are limited or when searching large datasets.

**Practical experiments in Python**

To evaluate the performance of the binary and jump search algorithms, arrays of varying lengths were utilized. To accurately measure the execution time of the algorithms and determine their memory usage, the memory-profiler library was employed. This approach allowed for a comprehensive assessment of the algorithm’s efficiency and resource utilization, providing valuable insights into their performance characteristics. In the testing phase, the identical algorithm implementations from Section 3 were employed. To ensure thorough evaluation, four separate tests were conducted using arrays of different lengths: 50, 5000, 500000, and 50000000, respectively.

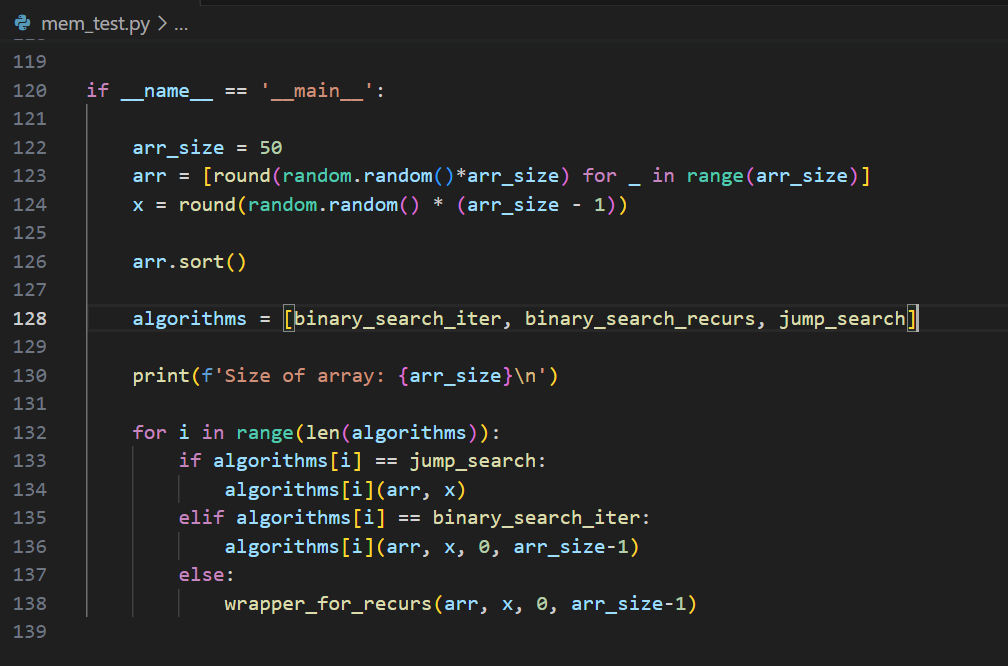


Figure 8. Python implementation of performance evaluation

1. Test1

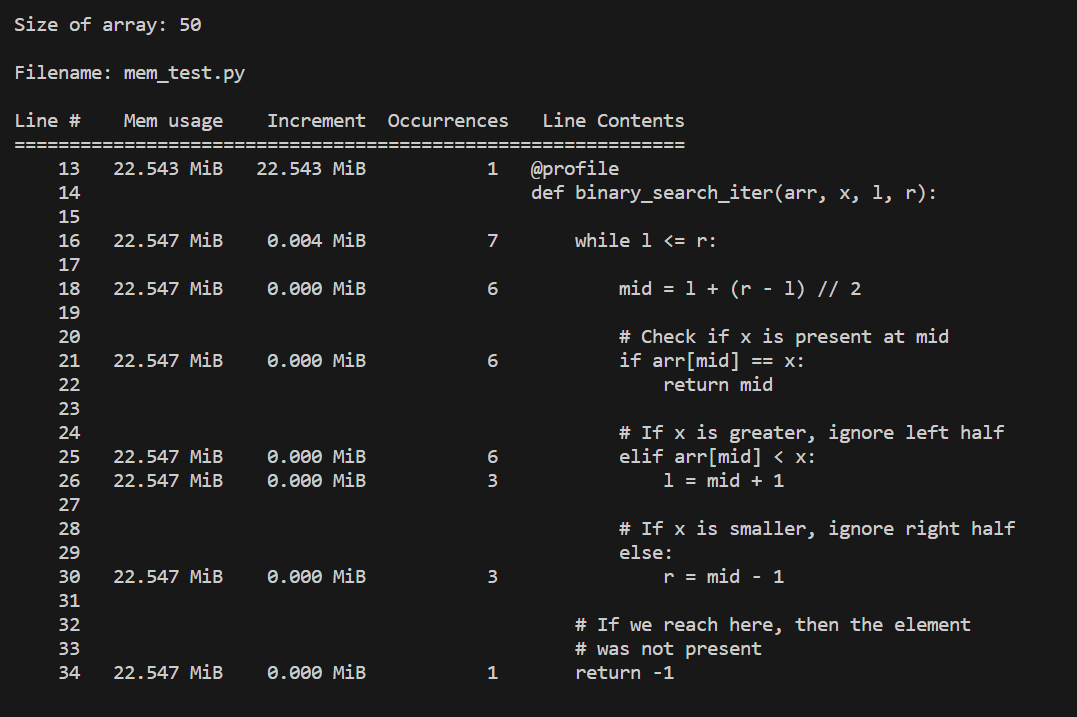


Figure 9. Test's 1 result of iterative binary search memory usage

A screenshot of a computer program

Description automatically generated with medium confidence

Figure 10. Test's 1 result of jump search memory usage

A screenshot of a computer program

Description automatically generated with low confidence

Figure 11. Test's 1 result of recursive binary search memory usage

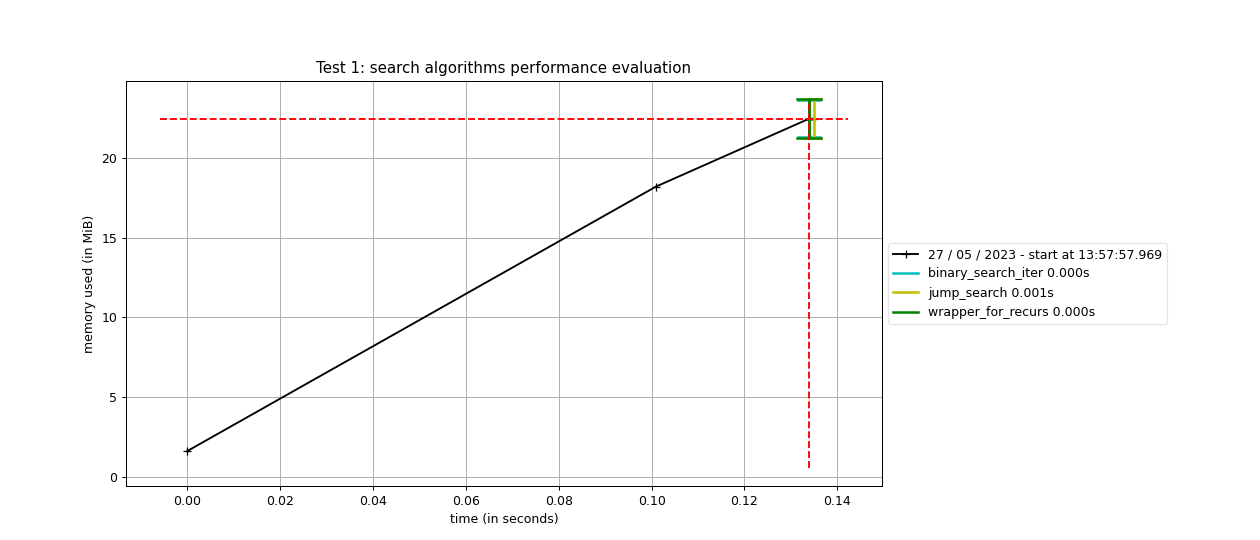


Figure 12. Graph of test 1 performance evaluation

1. Test 2

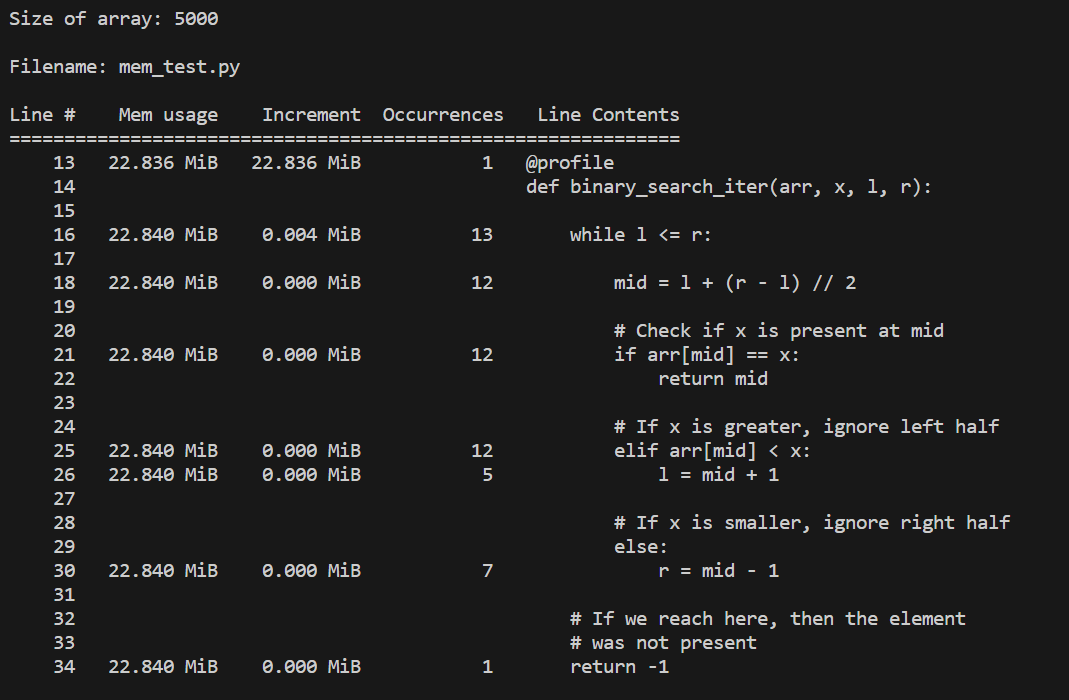


Figure 13. Test's 2 result of iterative binary search memory usage

A screenshot of a computer program

Description automatically generated with medium confidence

Figure 14. Test's 2 result of jump search memory usage

A screenshot of a computer

Description automatically generated with medium confidence

Figure 15. Test's 2 result of recursive binary search memory usage

A picture containing text, line, diagram, plot

Description automatically generated

Figure 16. Graph of test 2 performance evaluation

1. Test 3

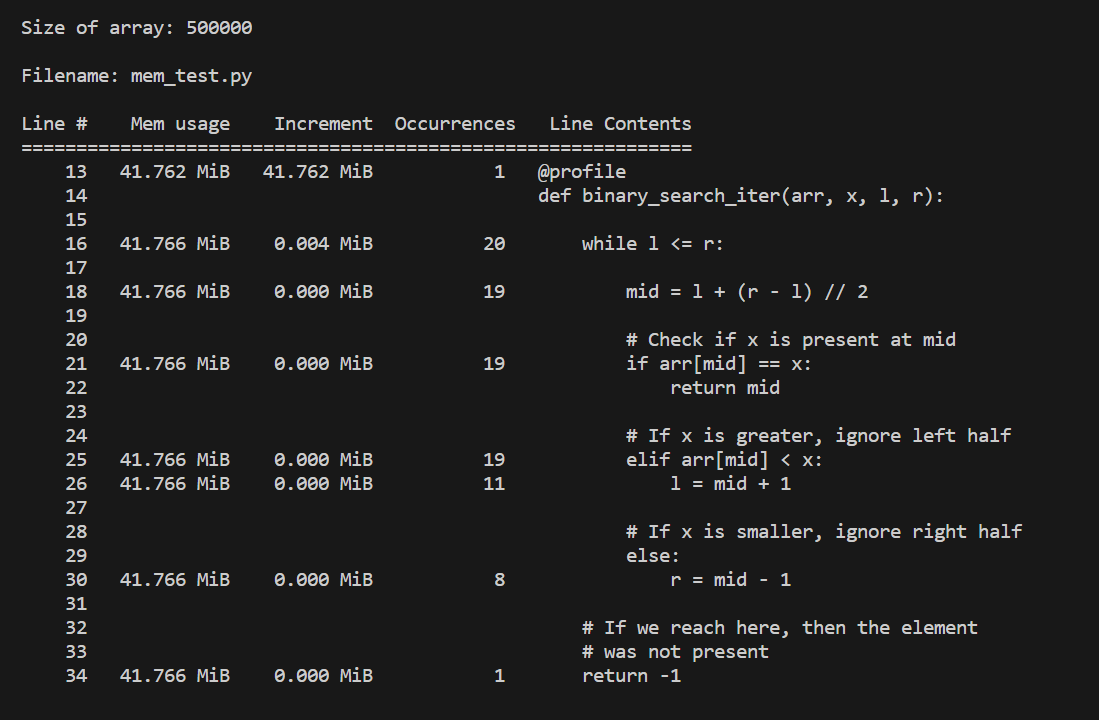


Figure 17. Test's 3 result of iterative binary search memory usage

A screenshot of a computer program

Description automatically generated with medium confidence

Figure 18. Test's 3 result of jump search memory usage

A screen shot of a computer

Description automatically generated with low confidence

Figure 19. Test's 3 result of recursive binary search memory usage

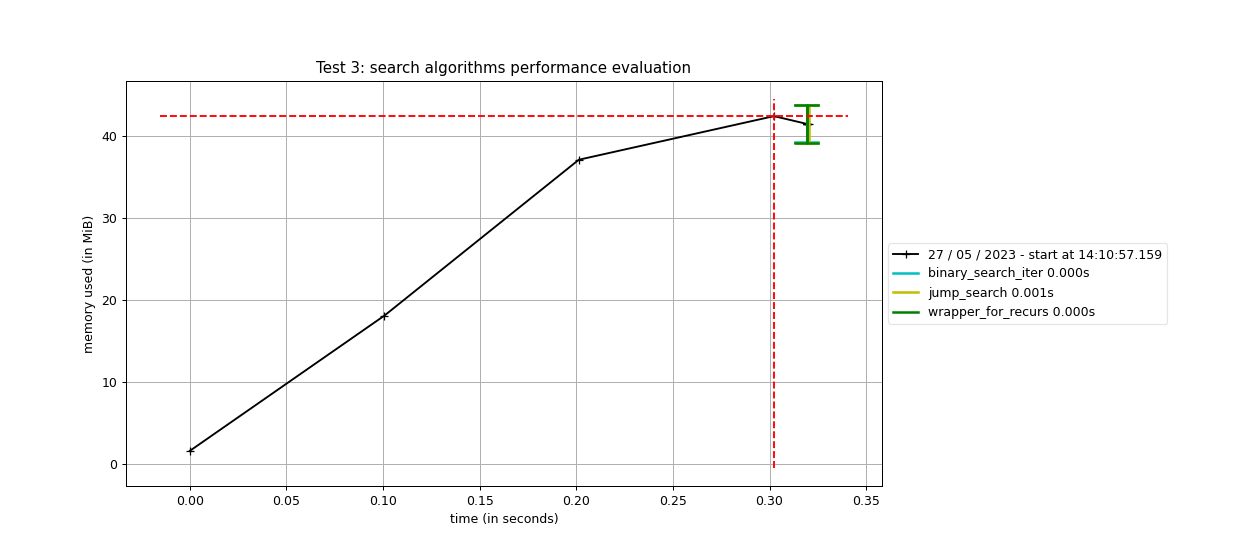


Figure 20. Graph of test 3 performance evaluation

1. Test 4

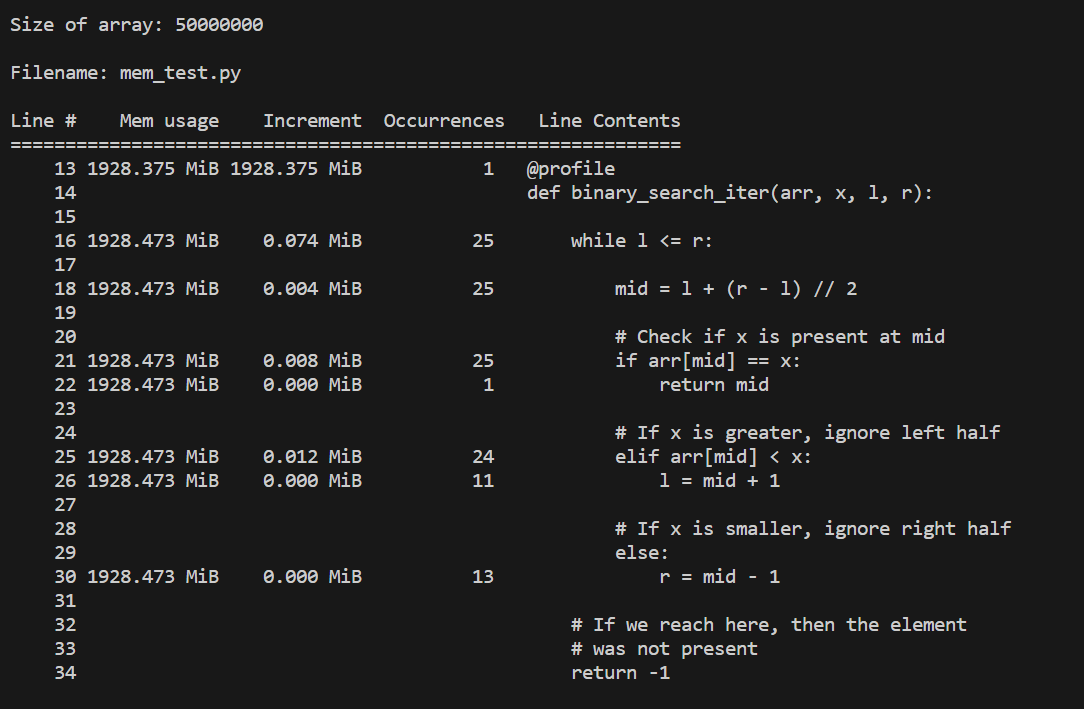


Figure 21. Test's 4 result of iterative binary search memory usage

A picture containing text, screenshot, font

Description automatically generated

Figure 22. Test's 4 result of jump search memory usage

A screen shot of a computer

Description automatically generated with low confidence

Figure 23. Test's 4 result of recursive binary search memory usage

A picture containing text, diagram, line, plot

Description automatically generated

Figure 24. Graph of test 4 performance evaluation

Let's analyze the data we received. We have snapshots of the interpreter's memory states taken using memory-profiler. The information is presented in the form of a table. The first column represents the line number of the code that has been profiled, the second column (Mem usage) the memory usage of the Python interpreter after that line has been executed. The third column (Increment) represents the difference in memory of the current line with respect to the last one. The fourth column (Occurrences) shows the number of times the line was executed. The last column (Line Contents) prints the code that has been profiled. The graph represents the amount of memory used at a certain time. The intersection of two red lines is the peak value of the memory used, colored timestamps are recorded when entering/leaving the profiled function.

As we can see, initialization and allocation of memory for an array takes the most time. The search algorithms themselves are executed in fractions of seconds even when the array is exceptionally large. Let's take a closer look at test 4 as an example. It is noteworthy that when using an array of numbers, the memory usage was measured to be 1928 MB. During the execution of the iterative binary search function, a mere 0.098 MB of memory was utilized, primarily for storing and manipulating the necessary indexes in 25 iterations. On the other hand, jump search consumed 0.15 MB of memory, approximately 1.5 times more than the iterative binary search, and performed 5102 iterations. Regarding the execution time, the binary search algorithm outperformed the jump search algorithm, completing the task in just 1ms, while jump search required 24ms to accomplish the same task.

These tests provide compelling evidence that binary search outperforms jump search in terms of speed for datasets of various sizes. Furthermore, binary search exhibits advantageous memory usage by virtue of its fewer iterations and consequent minimal data storage requirements. Notably, the two different implementations of binary search showcase comparable speed, with the iterative approach slightly edging out the other in memory usage, although the disparity is insignificant enough to be overlooked.

# SECTION Ⅴ.

**Conclusion**

In summary, the comparison of binary and jump search algorithms highlights key considerations for selecting the appropriate algorithm. Binary search consistently outperforms jump search in terms of speed, making it the preferred choice for efficiency-driven applications. It also exhibits favorable memory usage due to its fewer iterations and reduced data storage requirements. However, jump search remains a viable alternative for memory-constrained scenarios, although it sacrifices some speed compared to binary search. Ultimately, the decision between the two algorithms should be based on the specific requirements of the problem at hand, ensuring an optimal balance between speed and memory efficiency.