Jurnal Inovasi Informatika dan Bisnis Digital (JIIBD)

Website: https://ioinformatic.org/index.php/JIIBD

Juli 2026. Vol. 1. No. 3

Analysis of Binary Search with the Decrease and Conquer Method Using Java NetBeans

Victor Maruli Pakpahan[1*], Alya Syafira[2], Nurul Khairunnisa[3]

[1]Universitas Mahkota Tricom Unggul, [2,3]STMIK Kaputama, Binjai, Indonesia
victor.pakpahan@gmail.com[1]*, nurulkhairunnisa10032006@gmail.com[3]

Abstract

The increasing volume of digital data requires search procedures that are not only correct but also efficient in terms of execution time and computational operations. This article analyzes the Decrease-and-Conquer paradigm through the implementation of the Binary Search algorithm using Java in the NetBeans development environment. The study is developed from two student papers and reorganized into a journal-style article by combining conceptual analysis, algorithm design, source-code implementation, and functional testing. The method follows a descriptive-computational approach: the theoretical concepts of algorithms and Decrease-and-Conquer are first reviewed, then the Binary Search mechanism is formalized into pseudocode, converted into Java code, and tested using several input scenarios. The discussion is extended by adding a step-by-step trace table, comparison with Linear Search, best-average-worst case complexity analysis, and practical evaluation of algorithm limitations. The findings show that Binary Search works by maintaining left and right boundaries, calculating the middle index, comparing the middle element with the target, and discarding one half of the search space in each iteration. This reduction strategy produces O(1) complexity in the best case and O(log n) complexity in both the average and worst cases. However, the algorithm is only valid when the input data are sorted. The Java implementation successfully returns the target index when data exist and returns -1 when data are absent. Overall, Binary Search is proven to be an effective and scalable search algorithm for ordered data structures.

***Keywords**:* Binary Search, Decrease and Conquer, Java, NetBeans, algorithm complexity, ordered array

1. Introduction

Digital information systems rely heavily on data retrieval operations. Academic information systems, databases, inventory systems, digital dictionaries, document archives, and file-management applications all require a search mechanism to locate a specific value among stored data. When the number of records increases, inefficient search procedures may reduce system responsiveness and increase computational workload. Therefore, algorithm selection is an essential component of software design and data-structure implementation [1], [2].
One of the fundamental paradigms in algorithm design is Decrease and Conquer. This paradigm solves a problem by reducing its size into a smaller instance until the solution becomes easier to obtain. Unlike Divide and Conquer, which divides a problem into multiple subproblems, Decrease and Conquer generally continues with only one smaller subproblem. This property makes it suitable for search problems in which a large part of the search space can be eliminated after each comparison [2], [10].

Binary Search is a classic algorithm that applies the Decrease-and-Conquer principle. The algorithm searches for a target value in a sorted array by comparing the target with the middle element. If both values are equal, the search process stops. If the target is smaller, the search continues in the left half; if the target is larger, the search continues in the right half. Each iteration eliminates half of the remaining candidates, which makes Binary Search significantly faster than Linear Search for large sorted datasets [1], [3], [11].
This article reformulates and expands two student papers on the implementation of Binary Search using Java NetBeans. The journal version improves the original reports by presenting a clearer research method, a more systematic algorithmic trace, comparative complexity analysis, and a more structured discussion aligned with the Journal of Optimization and Decision Science template.

2. Research Method

2.1. Research Design

This study uses a descriptive-computational design. The descriptive part explains the theoretical foundation of Decrease and Conquer, Binary Search, algorithmic requirements, and complexity analysis. The computational part implements the algorithm in Java and evaluates whether the program produces correct outputs under several testing scenarios. The study does not measure hardware-level execution time; instead, it focuses on algorithmic behavior, functional correctness, and theoretical efficiency [4], [12].

2.2. Data Sources and Document Integration

The primary sources of this article are two student papers discussing Decrease and Conquer and the Java NetBeans implementation of Binary Search. Both papers contain similar core materials, including background, theoretical review, pseudocode, source-code explanation, program testing, advantages, disadvantages, and conclusions. In this article, those materials are synthesized into a single academic structure to avoid repetition and to strengthen coherence.

Table 1: Integration of the Two Source Papers into the Journal Structure

Source Material	Original Content	Journal Development
Paper 1	Background, Binary Search concept, pseudocode, source code, testing, and conclusion.	Converted into introduction, algorithm mechanism, implementation, testing, and conclusion.
Paper 2	Algorithm concept, research objectives, Decrease-and-Conquer theory, complexity, and references.	Used to enrich the theoretical framework, method section, complexity analysis, and discussion.
Template 2270	JODS-style article format with abstract, keywords, method, results, discussion, and references.	Applied as the formatting model for the final English journal article.

2.3. Implementation Procedure

The implementation procedure consists of five stages. First, the input array is prepared in ascending order because Binary Search requires sorted data. Second, the left boundary is initialized at index 0 and the right boundary at the last index. Third, the middle index is calculated in every iteration. Fourth, the middle value is compared with the target value to determine whether the search is completed or whether the search range should move to the left or right part of the array. Fifth, the program returns the index if the target is found or returns -1 if the search range becomes empty.

2.4. Evaluation Criteria

The algorithm is evaluated using three criteria: correctness, traceability, and complexity. Correctness refers to whether the Java program returns the expected output for both existing and non-existing targets. Traceability refers to whether each step of the algorithm can be followed through boundary changes. Complexity refers to the number of comparisons required as the data size grows. These criteria are appropriate because Binary Search is primarily evaluated through its logical behavior rather than through visual interface performance.

3. Result and Discussion

3.1. Algorithmic Principle of Decrease and Conquer

The Decrease-and-Conquer paradigm reduces the original problem into a smaller problem instance. In Binary Search, the reduction factor is one half. After the target is compared with the middle element, one half of the array is logically removed from consideration. This mechanism is more efficient than checking every element sequentially because the number of remaining candidates decreases exponentially rather than linearly.

For example, an array containing 1,024 elements requires at most 1,024 comparisons in Linear Search when the target is located at the last position or does not exist. In contrast, Binary Search requires at most about 10 comparisons because log2(1024) = 10. This difference becomes increasingly significant as the dataset grows.

3.2. Binary Search Working Model

The Binary Search working model can be represented through three variables: left, right, and middle. The variable left marks the first possible index, right marks the last possible index, and middle is the candidate index being examined. The algorithm repeatedly updates the left or right boundary until the target is found or until left becomes greater than right. This boundary update is the core of the Decrease-and-Conquer process.

```
Algorithm: BinarySearch(data, target)
Input : A sorted integer array data and a target value
Output: The index of target, or -1 if target is absent

left  <- 0
right <- length(data) - 1

while left <= right do
   middle <- (left + right) / 2

   if data[middle] = target then
      return middle
   else if data[middle] < target then
      left <- middle + 1
   else
```

```
        right <- middle - 1
    end if
end while

return -1
```

Gambar 1: Pseudocode of the Binary Search algorithm

3.3. Step-by-Step Algorithm Trace

To make the process more transparent, the algorithm can be traced using the sorted array {13, 15, 18, 19, 26, 33, 38, 56, 82, 91} and target value 38. The trace demonstrates how the search range is reduced until the target is located.

Tabel 2: Binary Search Trace for Target 38

Iteration	Left	Right	Middle	data[Middle]	Decision
1	0	9	4	26	26 < 38, move left to middle + 1
2	5	9	7	56	56 > 38, move right to middle - 1
3	5	6	5	33	33 < 38, move left to middle + 1
4	6	6	6	38	Target found at index 6

3.4. Java NetBeans Implementation

The following Java code shows the core implementation of Binary Search. The method binarySearch receives an integer array and a target value. It returns the index when the target is found and returns -1 when the target does not exist in the array. The main method prepares a sorted array, accepts user input, calls the binarySearch method, and prints the result.

```
import java.util.Scanner;

public class BinarySearch {
    public static int binarySearch(int[] data, int target) {
        int left = 0;
        int right = data.length - 1;

        while (left <= right) {
            int middle = (left + right) / 2;

            if (data[middle] == target) {
                return middle;
            } else if (data[middle] < target) {
                left = middle + 1;
            } else {
                right = middle - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int[] data = {13, 15, 18, 19, 26, 33, 38, 56, 82, 91};

        System.out.print("Enter target value: ");
        int target = input.nextInt();
        int result = binarySearch(data, target);

        if (result != -1) {
            System.out.println("Data found at index " + result);
        } else {
            System.out.println("Data not found");
        }
    }
}
```

Gambar 2: Java implementation of Binary Search in NetBeans

3.5. Functional Testing

Functional testing is conducted to verify whether the program can handle different target positions. The tests include a target in the middle area, a target at the final boundary, a target at the first boundary, and a target that does not exist in the array. These cases are important because they represent common and edge-case scenarios.

Table 3: Functional Testing Results

No.	Test Scenario	Target	Expected Output	Result
1	Target exists in the middle-right area of the array.	38	Data found at index 6	Valid
2	Target exists at the last element of the array.	91	Data found at index 9	Valid
3	Target exists at the first element of the array.	13	Data found at index 0	Valid
4	Target does not exist in the array.	1	Data not found	Valid

3.6. Complexity and Comparative Evaluation

The time complexity of Binary Search is expressed by the recurrence relation $T(n) = T(n/2) + 1$. The term $n/2$ shows that the input size is reduced by half after each iteration, while +1 represents one comparison at the current middle element. Solving the recurrence gives $O(\log n)$, which indicates logarithmic growth. This is the main reason Binary Search is highly efficient for large sorted arrays.

Table 4: Complexity Comparison between Linear Search and Binary Search

Aspect	Linear Search	Binary Search
Input requirement	Data may be sorted or unsorted.	Data must be sorted before searching.
Best case	O(1), when the target is the first element.	O(1), when the target is the middle element.
Average case	O(n), because elements are checked sequentially.	O(log n), because the search space is halved.
Worst case	O(n), when the target is last or absent.	O(log n), when the target is near a boundary or absent.
Practical use	Simple and suitable for small or unsorted data.	Efficient for large ordered datasets and database-like structures.

The comparison shows that Binary Search is not universally superior in all situations. For very small arrays or unsorted data, Linear Search may be easier to apply because it does not require preprocessing. However, when the data are already sorted or when sorting is part of the system design, Binary Search offers a substantial efficiency advantage. This makes it appropriate for applications such as student-data search, indexed records, dictionaries, inventory lookups, file systems, and other structured data repositories [5], [6], [7].

3.7. Strengths, Limitations, and Practical Implications

The main strength of Binary Search is its logarithmic time complexity, which allows the algorithm to scale well as the number of data items increases. The algorithm is also memory efficient because the iterative implementation only uses a small number of variables. In addition, the logic is relatively simple and can be implemented in many programming languages.

The main limitation is the sorted-data requirement. If data are frequently inserted, deleted, or modified without maintaining order, additional sorting or balanced data structures may be required. Another limitation is that a basic Binary Search returns one matching index, not necessarily the first or last occurrence when duplicate values exist. For duplicate datasets, the algorithm can be modified to continue searching toward the left or right after a match is found.

From a practical perspective, Binary Search should be taught as a foundation for understanding more advanced search structures, such as binary search trees, indexed arrays, and database indexing. Its Decrease-and-Conquer pattern also helps students understand why reducing the problem size strategically can significantly improve computational efficiency.

4. Conclusion

This article concludes that Binary Search is an effective implementation of the Decrease-and-Conquer paradigm. The algorithm reduces the search space by half in every iteration by comparing the target value with the middle element of a sorted array. The Java NetBeans implementation demonstrates that the algorithm correctly returns the target index when the value exists and returns -1 when the value is absent. The testing scenarios confirm that the program works for middle, boundary, and unavailable targets. The complexity analysis shows that Binary Search has O(1) best-case complexity and O(log n) average and worst-case complexity, making it far more efficient than Linear Search for large sorted datasets. Nevertheless, its use is limited by the requirement that the data must be sorted. Future development can include recursive implementation, duplicate-value handling, automatic sorting validation, and empirical runtime measurement using larger datasets.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
[2] Levitin, A. (2012). Introduction to the Design and Analysis of Algorithms (3rd ed.). Pearson.
[3] Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
[4] Schildt, H. (2019). Java: The Complete Reference (11th ed.). McGraw-Hill Education.
[5] Dharmawan, R., & Pratama, A. (2023). Analisis penerapan algoritma binary search pada sistem pencarian data berbasis Java. Jurnal Teknologi Informasi dan Komputer, 7(2), 45-52.
[6] Prasetyo, D. A. (2022). Implementasi algoritma binary search dalam pencarian data mahasiswa menggunakan Java. Jurnal Informatika dan Rekayasa Perangkat Lunak, 4(1), 33-40.

[7] Rahman, F., & Wijaya, H. (2021). Analisis kompleksitas algoritma binary search pada pengolahan data terurut. Jurnal Sistem Informasi dan Teknologi, 5(3), 101-108.
[8] Sari, N. P. (2024). Perbandingan linear search dan binary search dalam proses pencarian data. Jurnal Ilmu Komputer dan Teknologi Informasi, 9(1), 12-19.
[9] Fitriyani, A., Achmad, A. H., Lestari, D., & Fitriawati, N. (2025). Penerapan algoritma binary search pencarian data pada sistem supply barang. JSI (Jurnal Sistem Informasi) Universitas Suryadarma.
[10] Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.
[11] Horowitz, E., Sahni, S., & Rajasekaran, S. (2008). Fundamentals of Computer Algorithms (2nd ed.). Universities Press.
[12] Weiss, M. A. (2012). Data Structures and Algorithm Analysis in Java (3rd ed.). Pearson.