Fast in Technologies and AI Solutions

Website: https://fastin-technologies.com/ftas

Vol. 1 No.1 October 2025 : 28-35.

Optimization of Data Search Process Using a Combination of Merge Sort and Binary Search

Dimas R. Aulia[1*], Faris Andisa[2], Hafid Nasrullah[3], Firdiawan Hermawan[4]

[1,2,3,4]Universitas Ahmad Dahlan

**Corresponding author E-mail: 2400018245@webmail.uad.ac.id*

Received: 5 August 2025
Accepted: 7 October 2025

Abstract

The growth in data volume in the digital age makes speed in processing and searching for information an increasingly important need. This research focuses on optimizing the data search process by comparing two algorithm combination strategies. The first strategy combines recursive Merge Sort with manual Binary Search, while the second strategy uses iterative (bottom-up) Merge Sort paired with the lower bound function from the C++ library. This research employs a quantitative experimental approach by testing execution times on random datasets of 1,000, 10,000, and 100,000 elements. The test results show that bottom-up Merge Sort has faster execution times compared to the recursive version, for example, 71.2 ms versus 107.2 ms for a dataset of 100,000 elements. This speed is achieved because the iterative approach does not require calling recursive functions, which can add to the memory load. For the search process, manual Binary Search proved to be very fast for a one-time search. However, in a repeated search scenario, the lower bound function is more efficient and consistent in execution time. The combination of bottom-up Merge Sort and lower bound becomes the optimal choice for large-scale systems that require a one-time sorting process and repeated searches, such as search engines and massive data filters.
This research concludes that selecting an algorithm strategy appropriate for the frequency and needs of the search can improve system efficiency in processing large-scale data.

Keywords: *Data Search; Data Sorting; Merge Sort; Binary Search; Bottom-up and Lower Bound*

1. Introduction

In the modern digital era, which is filled with vast amounts of data, the speed of processing and searching for data has become an extremely important need. The data search process is a fundamental component in many information systems, such as file searching, user data retrieval, and real-time command processing. The main challenge currently faced is how to perform fast searches, especially as data volume continues to increase[1][2].

Search efficiency is highly dependent on the condition of the data used. Binary Search, one of the popular search algorithms, can only function optimally if the data is already sorted. That's why an initial step of data sorting is needed so that the search process can run efficiently. One of the search methods widely used in C++ programming practice is the lower bound function, which internally implements the Binary Search principle with logarithmic efficiency [3][4].

Before the search process can be carried out, the data needs to be sorted first. In this study, the bottom-up version of Merge Sort was chosen as the sorting algorithm because it is stable, does not use recursion, and has a time complexity of O (n log n). Compared to the recursive approach, bottom-up Merge Sort is more suitable for environments that avoid intensive stack usage, such as systems with certain memory limitations[5].

The combination of bottom-up Merge Sort and lower bound as a search method is a practical strategy for optimizing the data search process. Merge Sort ensures efficient data arrangement, while lower_bound speeds up the search without requiring a manual algorithm. This approach is highly relevant for large-scale systems that require high efficiency in terms of search speed and data processing [6] [7].

Data search, or searching, is a process for finding a specific value or element within a dataset of similar types. This process is crucial in data processing because it helps users or systems quickly and efficiently obtain the necessary information, whether it's basic data like numbers and letters or more complex data like arrays, structures, or objects[8][9].

Copyright © Authors. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The main goal of search algorithms is to find an element within a dataset quickly and efficiently. These algorithms are designed to speed up the search process, so users don't have to check each element one by one. In an era of increasingly large and complex data, the use of optimal search algorithms is crucial for improving system performance and minimizing access time to information[10][11].

Binary search is an efficient search method that works on sorted data. This algorithm repeatedly divides the dataset into two equal parts. If the midpoint is greater than the value being searched for, the search continues in the first half. Conversely, if the midpoint is smaller, the search continues in the second half. This elimination process continues until the desired element is found or until the search area is exhausted[12][13].

The working principle of Binary Search is to repeatedly divide the sorted dataset into two parts to speed up the process of finding a specific element[14][15].

The value to be searched for is compared to the element in the middle position of the array; if they are the same, the search process is complete. If the value being searched for is smaller, the search continues to the left side of the array and to the right side. This procedure is repeated until the element is found or the search space is exhausted. Binary search is faster than linear search because it has a time efficiency of O (log n) since each iteration halves the amount of data that needs to be checked.

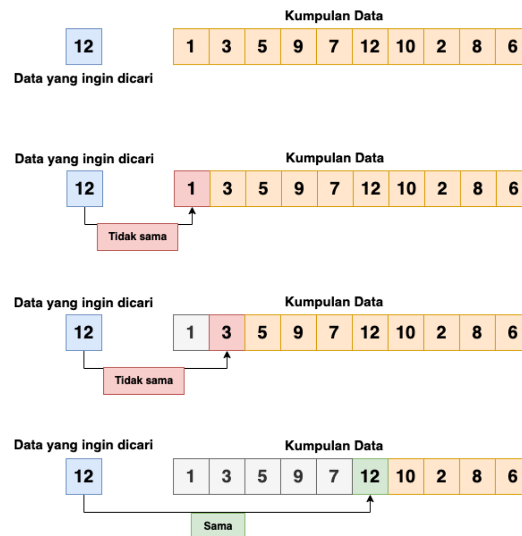

Fig. 1: Binary Search Algorithm Flowchart

Binary search is a highly efficient search algorithm because it works by dividing the search space into two parts at each step. For this algorithm to work correctly, there are several important conditions that must be met. First, the data to be searched must be sorted, either in ascending or descending order, because the search area division depends on value comparisons. Second, the data structure used must allow random access via an index, such as an array, because binary search is not suitable for structures like linked lists that only support sequential access. The elements within a dataset must be of the same type and comparable for the logical evaluation process to run consistently.

Optimizing binary search with the lower bound method is used to find the position of the smallest element in a sorted array that is equal to or greater than a specific value. The process involves repeatedly dividing the search space by comparing the midpoint values. If the middle value is smaller, the search continues to the right; if it's greater than or equal to, the search continues to the left. The final result is the index of the first qualifying element[16].

Data sorting is the process of arranging initially unordered data into a specific sequence, making the data structured according to established rules. This process can be done in ascending or descending order and can be applied to both numerical and character data[17][18].

An algorithm is a structured set of steps to achieve a specific result. This process starts from the initial condition and is then executed until it reaches the final condition. An example of algorithm application is data sorting. Sorting algorithms are one of the topics with a variety of solution methods[19].

Merge sort is an algorithm for sorting data that uses the divide and conquer strategy. The way to do this is by breaking down the data into very small sub-sections, so that each section only has one or two elements. Then, these small sections are sorted and gradually reassembled until they form a complete and ordered data list.

The working principle of Merge Sort follows three main stages: divide, conquer, and combine. First, in the divide stage, the data is recursively divided into two parts until each part has only one element. Second, in the conquer stage, these small parts begin to be sorted. Since that section only has one element, it is considered sorted. Finally, in the combine stage, the elements from the sorted sub-arrays are merged back into the main array step by step by comparing the elements from each sub-array and arranging them in order.

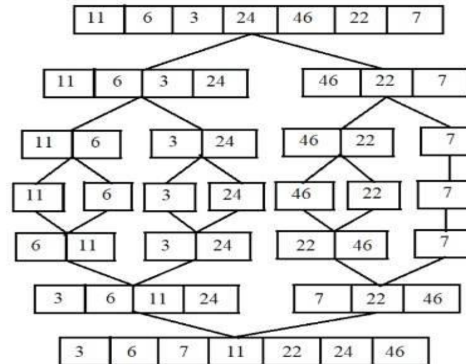

Fig. 2: Flowchart of the Merge Sort Algorithm

This process continues until all the data is successfully merged back into a single, perfectly sorted array. This principle allows Merge Sort to have stable performance with a time complexity of O(n log n) under all conditions, including best, average, and worst cases. This efficiency makes merge sort ideal for sorting large-scale data. However, because it uses extra space (temporary arrays), merge sort requires additional memory allocation, especially when implemented recursively.

Bottom-up optimization has several advantages, one of which is avoiding the overhead of recursive function calls that can burden memory, especially on systems with limited stacks. This method is often easier to implement in the context of low-level systems or embedded systems. By using a bottom-up approach, the algorithm maintains a time complexity of O(n log n) but with the added efficiency of memory management and a simpler program structure. Bottom-up merge sort is a viable and optimal alternative to consider in certain scenarios.

Algorithm complexity is a fundamental concept in computer science used to measure the efficiency of an algorithm based on the resources required, particularly execution time (time complexity). This analysis is crucial for predicting algorithm performance when handling large datasets, allowing for process optimization according to system needs. Typically, the complexity of an algorithm is expressed using Big O notation, which describes how execution time increases as the input size grows in the worst-case scenario.

In the analysis of algorithm complexity, time and space efficiency are key indicators in assessing algorithm performance, especially when dealing with large-scale data. One example of an efficient sorting algorithm is Merge Sort, particularly with a bottom-up approach. This approach avoids explicit recursion by iteratively merging small sub-arrays until the entire data is sorted. Merge Sort has a time complexity of O(n log n) for the best, average, and worst cases. This makes it stable and very suitable for sorting large amounts of data. From a space perspective, Merge Sort requires an additional O(n) space to store the merged results. Its stability and consistent performance make it one of the most reliable sorting algorithms.

In the context of data searching, Binary Search is a highly efficient algorithm with a time complexity of O(log n). This value represents the lower bound for all comparison-based search algorithms in the worst-case scenario. This means that no comparative search algorithm can work faster than O(log n) if the data is already sorted. Binary Search utilizes a divide-and-conquer strategy by dividing the search space in half with each iteration, making it highly effective for sorted data. Compared to other algorithms, Merge Sort is at the upper bound of efficiency for comparison-based sorting algorithms, while Binary Search represents the lower bound of efficiency for search algorithms. Both reflect how the complexity of algorithms can be used to measure optimal efficiency in solving computational problems.

2. Research Method

The type of research used is quantitative research with an experimental method. This research focuses on measuring and comparing the performance (execution time) of several different algorithms under controlled conditions. The experiment was conducted by implementing sorting algorithms (recursive and bottom-up Merge Sort) and searching algorithms (manual and lower_bound Binary Search) to process datasets of varying sizes. The data from the execution time measurements was then analyzed quantitatively to determine which algorithm combination was the most efficient.

Gathering and studying the basic theories regarding search and sorting algorithms, particularly Binary Search and Merge Sort, as well as the concept of algorithm complexity (Big O Notation). Implementing 2 algorithm combinations using the C++ programming language, Merge Sort and Binary Search use the conventional method, Merge Sort with Bottom Up and Binary Search with Lowerdown.

Preparing a dataset consisting of random numbers with three different sizes: 1,000, 10,000, and 100,000 elements to test the scalability of the algorithm. Comparing the execution time data collected from each test. The analysis focuses on comparing the speed between the regular method and the optimized methods (bottom-up and lower bound).

```
1. void merge(vector<int>& arr, int left, int mid, int right) {
 2.     vector<int> temp(right - left + 1);
 3.     int i = left, j = mid + 1, k = 0;
 4.
 5.     while (i <= mid && j <= right)
 6.         temp[k++] = (arr[i] < arr[j]) ? arr[i++] : arr[j++];
 7.
 8.     while (i <= mid) temp[k++] = arr[i++];
 9.     while (j <= right) temp[k++] = arr[j++];
10.
11.     for (int x = 0; x < temp.size(); ++x)
12.         arr[left + x] = temp[x];
13. }
14.
15. void mergeSort(vector<int>& arr, int left, int right) {
16.     if (left >= right) return;
17.     int mid = (left + right) / 2;
18.     mergeSort(arr, left, mid);
19.     mergeSort(arr, mid + 1, right);
20.     merge(arr, left, mid, right);
21. }.
```

Fig. 3: Implementation of Merge Sort

The Merge Sort algorithm is recursively implemented using a divide-and-conquer approach, where the array is recursively divided in half until it reaches a single element, and then merged back in order. The purpose of this testing is to observe the efficiency of the algorithm across various dataset sizes and patterns. Execution time is recorded using the time function from the C++ library as the basis for performance measurement.

```
void mergeSortBottomUp_Corrected(std::vector<int>& arr) {
 2.   int n = arr.size();
 3.   if (n < 2) return;
 4.   std::vector<int> buffer(n);
 5.
 6.   for (int width = 1; width < n; width *= 2) {
 7.       for (int i = 0; i < n; i += 2 * width) {
 8.           int left = i;
 9.           int mid = std::min(i + width, n);
10.            int right = std::min(i + 2 * width, n);
11.
12.
13.            int p1 = left, p2 = mid, k = left;
14.            while (p1 < mid && p2 < right) {
15.
16.                buffer[k++] = (arr[p1] <= arr[p2]) ? arr[p1++] : arr[p2++];
17.            }
18.            while (p1 < mid) {
19.                buffer[k++] = arr[p1++];
20.            }
21.              while (p2 < right) {
22.                  buffer[k++] = arr[p2++];
23.              }
24.
25.
26.              for (int j = left; j < right; ++j) {
27.                  arr[j] = buffer[j];
28.              }
29.          }
30.      }
31.  }
32.
```

Fig. 4: Implementation of Merge Sort with Bottom Up

For comparison, Merge Sort is also implemented in an iterative (bottom-up) form. This approach does not use recursion, but rather divides the data into smaller subarrays and then merges them step by step until they become a single sorted array. Testing was conducted on the same dataset as in the recursive method, with a focus on time efficiency and algorithm stability in handling large datasets.

```
1. int binarySearch(const vector<int>& arr, int target) {
2.      int left = 0, right = arr.size() - 1;
3.
4.      while (left <= right) {
5.          int mid = (left + right) / 2;
6.          if (arr[mid] == target)
7.              return mid;
8.          else if (arr[mid] < target)
9.              left = mid + 1;
10.         else
11.             right = mid - 1;
12.     }
13.
14.     return -1;
15.  }
```

Fig. 5: Implementation of Binary Search

After the data is sorted, the manual Binary Search algorithm is used to find an element within the sorted array. This method is performed by finding the midpoint and repeatedly narrowing the search space. Testing was conducted to measure how quickly this method finds elements in various positions (beginning, middle, end, not found).

```
1. #include <algorithm>
2.
3.  int binarySearchLowerBound(const vector<int>& arr, int target) {
4.      auto it = lower_bound(arr.begin(), arr.end(), target);
5.      if (it != arr.end() && *it == target)
6.          return distance(arr.begin(), it);
7.      return -1;
8.  }
```

Fig. 6: Implementation of Binary Search with LowerBound

As a comparison, the lower_bound() function from the C++ library is used. This function automatically returns an iterator to the first element that is not less than the target value. Although it seems simple, this function is very efficient and is used to optimize searches within a sorted array. Testing was conducted to see if the performance was consistent or even superior to manual implementation. Tools and Environment; Programming Language C++, IDE Dev C++, Compiler MinGW-w64 (latest version of GCC), Time Measurement: Using the clock() function from or chrono from C++.

3. Results and Discussion

Based on the test results for the recursive Merge Sort algorithm, it can be seen that the execution time increases proportionally with the number of data elements. For a random dataset of 1,000 elements, the sorting time was recorded as 1.70242 milliseconds, while for 10,000 elements it increased to 9.17098 milliseconds, and reached 107.24 milliseconds for 100,000 elements. This increase is consistent with the time complexity of Merge Sort, which is O(n log n), where each exponential increase in data directly impacts the number of division and merging processes. Because this algorithm works recursively, each function call will stack up in memory, so despite being theoretically efficient, this approach has the added overhead of stack usage. Although the algorithm still shows consistent time stability across various data size scenarios, making it a viable option for systems that are not overly sensitive to recursive memory consumption.

Table 1: Processing Time for Randomized Data - Merge Sort

Data Set	Processing Time Randomized Data
1000	1.70242 ms
10000	9.17098 ms
100000	107.24 ms

Testing results for Merge Sort with a bottom-up approach show better performance compared to the recursive version, especially when handling large datasets. The sorting time for 1,000 elements is only 0.38292 milliseconds, significantly faster than the recursive method. For 10,000 elements, the time required is 5.62208 milliseconds, and for 100,000 elements, it's only 71.2689 milliseconds, demonstrating consistent efficiency and time savings. The bottom-up approach does not use recursion, but rather merges small sub-arrays step by step using loops, thus avoiding the overhead of the call stack. This makes it more memory-friendly and more stable for systems with stack limitations, such as embedded systems. While maintaining a time complexity of O(n log n), bottom-up Merge Sort becomes a highly optimized alternative in terms of memory management and execution speed, and is well-suited for large-scale systems with high data processing frequencies.

Table 2: Processing Time for Randomized Data Using Bottom-Up Merge Sort

Data Set	Processing Time Randomized Data
1000	0.38292 ms
10000	5.62208 ms
100000	71.2689 ms

Testing the manual Binary Search algorithm showed remarkable efficiency in single searches, with search times ranging from only 0.0002 ms to 0.0004 ms for datasets of 1,000 to 100,000 elements. However, when the search was performed repeatedly 100,000 times, the total time increased linearly with the amount of data, namely 1.10 ms for 1,000 elements, 1.40 ms for 10,000 elements, and 1.80 ms for 100,000 elements. This shows that although manual search is very fast in individual scenarios, its efficiency begins to decline when used in large-scale repeated searches because it does not utilize internal optimizations like standard functions. Nevertheless, due to its simple implementation and deterministic results, this method remains ideal for occasional searches or on systems that require full control over the search process.

Table 3: Processing Time for Binary Seacrh

Data Set	Search Time	
	1 Time	100000 Time
1000	0.0002 ms	1.10 ms
10000	0.0003 ms	1.40 ms
100000	0.0004 ms	1.80 ms

The lower_bound function from the C++ library shows slightly slower performance in a single search compared to manual Binary Search, with a difference of approximately 0.0013 ms to 0.0015 ms. However, it demonstrates a significant advantage when used for repeated searches 100,000 times. The total time required is 1.00 ms for a dataset of 1,000 elements, 1.20 ms for 10,000 elements, and 1.30 ms for 100,000 elements, which is consistently faster than the manual method in the context of mass searches. This indicates that lower_bound contains internal optimizations that make it more stable and efficient at scale. Using this built-in function also simplifies code implementation and guarantees the reliability of results through the standard library, making it highly suitable for applications requiring high speed and scalability in intensive data searches.

Table 4: Binary Search Time Using Lowerdown

Data Set	Search Time	
	1 Time	100000 Time
1000	0.0013 ms	1.00 ms
10000	0.0014 ms	1.20 ms
100000	0.0015 ms	1.30 ms

The combination of the recursive Merge Sort algorithm with manual Binary Search in a one-time search scenario demonstrates good efficiency in terms of total execution time. The test results show that for a dataset of 1,000 elements, the total sorting and searching time is 1.70262 ms; for 10,000 elements, it is 9.17128 ms; and for 100,000 elements, it reaches 107.2404 ms. The manual search time is only between 0.0002–0.0004 ms, so the contribution of search time hardly affects the total time, and the dominant performance comes from the sorting algorithm. This testing is only focused on single searches because in many real-world cases, data is only searched occasionally after the initial sorting process, for example, in batch processing, one-way report analysis, or simple validation processes. This combination is considered effective and efficient for systems that emphasize the final result of a single search query, rather than systems with repeated search queries.

Table 5: Total Time for the Combined Merge Sort and Binary Search

Data Set	Processing Time Randomized Data	Search Processing Time 1 time	Total Time Combination
1000	1.70242 ms	0.0002 ms	**1.70262 ms**
10000	9.17098 ms	0.0003 ms	**9.17128 ms**
100000	107.24 ms	0.0004 ms	107.2404 ms

The combination of bottom-up Merge Sort and the lower_bound function is optimized for search scenarios performed repeatedly in large quantities. From the test results, the total processing time for a 1,000-element dataset is 1.38292 ms, for 10,000 elements it is 6.82208 ms, and for 100,000 elements it is only 72.5689 ms. The sorting time using bottom-up Merge Sort contributes significantly at the beginning, but the real advantage is seen when the lower_bound function is used to perform 100,000 searches. This function has high stability and efficiency, keeping the total search time low even with a very large number of searches. This testing is indeed focused on 100,000 searches

because this scenario represents real-world system needs such as search engines, massive data filtering, or search features used repeatedly within a single application execution cycle. This combination was chosen as the optimal approach for systems with high search intensity after a single sorting process.

Table 6: Total Time for the Combined Merge Sort and Binary Search Optimization

Data Set	Processing Time Randomized Data	Search Processing Time 100000 time	Total Time Combination
1000	0.38292 ms	1.00 ms	**1.38292 ms**
10000	5.62208 ms	1.20 ms	**6.82208 ms**
100000	71.2689 ms	1.30 ms	72.5689 ms

A comparison between two data search algorithm combination strategies, namely recursive Merge Sort with manual Binary Search (for 1 search) and bottom-up Merge Sort with lower_bound (for 100,000 searches), shows that time efficiency is highly dependent on search frequency. In the first strategy, the total process time is dominated by sorting, as the search time is very small and has almost no impact on the overall result, making it suitable for systems that only require occasional searches. Conversely, in the second strategy, testing is performed in a mass search scenario because the lower_bound shows more stable and efficient performance when used repeatedly, with relatively constant search times even when run in large numbers. This comparison was done separately because both represent different needs in real-world implementation: a single search reflects a one-way analysis system or simple data validation, while 100,000 searches represent the workload of a database system, search engine, or application with high data access intensity.

4. Conclusion

Based on the results of the research and experimental analysis that have been conducted, it can be concluded that the efficiency of the data search process is highly dependent on the combination of algorithms used and the context of the number of searches. The combination of recursive Merge Sort with manual Binary Search proved more optimal for single searches because the search time was very small and the total execution time was almost entirely determined by the sorting speed. Conversely, the combination of bottom-up Merge Sort and the lower_bound function showed significantly better performance in large-scale repeated search scenarios, with more stable and efficient execution times. These results confirm that the optimal data search strategy must be tailored to the system's needs: if searches are infrequent, a manual approach is efficient enough, whereas for systems requiring continuous searching, optimized approaches like bottom-up Merge Sort and lower_bound are more recommended. This research shows that selecting the right combination of algorithms can significantly improve performance in large-scale data processing systems.

References

[1] P. Zhang *et al.*, "NetSHa: In-Network Acceleration of LSH-Based Distributed Search," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2213–2229, Sep. 2022, doi: 10.1109/TPDS.2021.3135842.

[2] R. Perego, G. E. Pibiri, and R. Venturini, "Compressed Indexes for Fast Search of Semantic Data," *IEEE Trans Knowl Data Eng*, vol. 33, no. 9, pp. 3187–3198, Sep. 2021, doi: 10.1109/TKDE.2020.2966609.

[3] P. Bose, K. Douïeb, J. Iacono, and S. Langerman, "The Power and Limitations of Static Binary Search Trees with Lazy Finger," *Algorithmica*, vol. 76, no. 4, pp. 1264–1275, Dec. 2016, doi: 10.1007/S00453-016-0224-X/METRICS.

[4] T. Brown, A. Prokopec, and D. Alistarh, "Non-blocking interpolation search trees with doubly-logarithmic running time," *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pp. 276–291, Feb. 2020, doi: 10.1145/3332466.3374542.

[5] W. Qiao, L. Guo, Z. Fang, M. C. F. Chang, and J. Cong, "TopSort: A High-Performance Two-Phase Sorting Accelerator Optimized on HBM-Based FPGAs," *IEEE Trans Emerg Top Comput*, vol. 11, no. 2, pp. 404–419, Apr. 2023, doi: 10.1109/TETC.2022.3228575.

[6] L. Wu, H. Huang, K. Su, S. Cai, and X. Zhang, "An I/O efficient model checking algorithm for large-scale systems," *IEEE Trans Very Large Scale Integr VLSI Syst*, vol. 23, no. 5, pp. 905–915, May 2015, doi: 10.1109/TVLSI.2014.2330061.

[7] P. Michiardi, D. Carra, and S. Migliorini, "Cache-Based Multi-Query Optimization for Data-Intensive Scalable Computing Frameworks," *Information Systems Frontiers*, vol. 23, no. 1, pp. 35–51, Feb. 2021, doi: 10.1007/S10796-020-09995-2/METRICS.

[8] M. AbdelNaby, M. E. Khalefa, Y. Taha, and A. Hassan, "Towards efficient top-k fuzzy auto-completion queries," *Alexandria Engineering Journal*, vol. 61, no. 7, pp. 5783–5791, Jul. 2022, doi: 10.1016/J.AEJ.2020.06.012.

[9] R. T. Rimi, K. M. A. Hasan, and T. Tsuji, "Multidimensional query processing algorithm by dimension transformation," *Sci Rep*, vol. 13, no. 1, pp. 1–11, Dec. 2023, doi: 10.1038/S41598-023-31758-7;SUBJMETA.

[10] M. Yu, C. Chai, and G. Yu, "A Tree-Based Indexing Approach for Diverse Textual Similarity Search," *IEEE Access*, vol. 9, pp. 8866–8876, 2021, doi: 10.1109/ACCESS.2020.3022057.

[11] N. Sultana, S. Paira, S. Chandra, and S. K. S. Alam, "A brief study and analysis of different searching algorithms," *Proceedings of the 2017 2nd IEEE International Conference on Electrical, Computer and Communication Technologies, ICECCT 2017*, no. March, 2017, doi: 10.1109/ICECCT.2017.8117821.

[12] J. Clarkson, K. Y. Lin, and K. D. Glazebrook, "A Classical Search Game in Discrete Locations," *https://doi.org/10.1287/moor.2022.1279*, vol. 48, no. 2, pp. 687–707, Jul. 2022, doi: 10.1287/MOOR.2022.1279.

[13] A. Lin, "Binary search algorithm," *WikiJournal of Science*, vol. 2, no. 1, pp. 1–13, 2019, doi: 10.15347/wjs/2019.005.

[14] M. N. Saeed *et al.*, "Empirical Analysis of Quaternary and Binary Search," pp. 1–6, 2024.

[15] P. Bose, J. Cardinal, J. Iacono, G. Koumoutsos, and S. Langerman, "Competitive online search trees on trees," *Proc West Mark Ed Assoc Conf*, vol. 2020-January, pp. 1878–1891, 2020, doi: 10.1137/1.9781611975994.115.

[16] A. Mehmood, "ASH Search: Binary Search Optimization," *International Journal of Computer Applications*, vol. 178, no. 15, pp. 10–17, 2019, doi: 10.5120/ijca2019918788.

[17] Q. Liu, Y. Ren, Z. Zhu, D. Li, X. Ma, and Q. Li, "RankAxis: Towards a Systematic Combination of Projection and Ranking in Multi-Attribute Data Exploration," *IEEE Trans Vis Comput Graph*, vol. 29, no. 1, pp. 701–711, Jan. 2023, doi: 10.1109/TVCG.2022.3209463.

[18] Z. Chen and A. Zhang, "A Survey of Approximate Quantile Computation on Large-Scale Data," *IEEE Access*, vol. 8, pp. 34585–34597, 2020, doi: 10.1109/ACCESS.2020.2974919.

[19] M. Markina and M. Buzdalov, "Hybridizing non-dominated sorting algorithms: Divide-and-conquer meets best order sort," *GECCO 2017 - Proceedings of the Genetic and Evolutionary Computation Conference Companion*, vol. 2017-January, pp. 153–154, Jul. 2017, doi: 10.1145/3067695.3076074.