PERFORMANCE EVALUATION AND ALGORITHMIC ANALYSIS OF ADVANCED SEARCH TECHNIQUES IN ARRAY-BASED STRUCTURES

Tuxtanazarov Dilmurod Solijonovich

Associate Professor, PhD, International Islamic Academy of Uzbekistan, Tashkent, Uzbekistan.

Ergashev Giyosjon Jurayevich

Associate Professor, PhD, International Islamic Academy of Uzbekistan, Tashkent, Uzbekistan.

Abstract: *In the contemporary landscape of high-velocity data processing, search efficiency is a primary determinant of system throughput. This paper provides an exhaustive analysis of search algorithms within array-based data structures. We dissect the transition from the O(n) linear baseline to the O(log n) efficiency of Binary Search, further exploring the O(log log n) potential of Interpolation Search.*

Introduction. The fundamental problem of locating a target element K within a collection A = {a_1, a_2, …, a_n} underpins nearly every facet of computational science. As datasets scale into the petabyte range, the choice of search algorithm dictates the latency of cloud databases, financial trading engines, and real-time operating systems. This article provides a rigorous academic evaluation of these algorithms, emphasizing the trade-offs between computational complexity and hardware-level efficiency.

The main part. Linear Search, or Sequential Search, serves as the fundamental benchmark. While often dismissed due to its O(n) complexity, it possesses unique advantages in modern hardware environments.

Linear search examines each element A[i] until a match is found or the end of the array is reached. The average number of comparisons for a successful search is (n+1)/2.

Modern CPUs utilize **SIMD (Single Instruction, Multiple Data)** and **prefetching**. Because Linear Search accesses memory contiguously, it results in a near 100% cache hit rate for small n. In high-performance C++ systems, Linear Search often outperforms Binary Search for $n < 64$ because it avoids the overhead of branch mispredictions associated with midpoint logic.

For sorted arrays, Binary Search is the definitive standard. It operates on the principle of reducing the problem size by half in each iteration.

The efficiency of Binary Search is rooted in the recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

Using the Master Theorem, we establish the complexity as $O(\log_2 n)$. This logarithmic growth is what enables Google or Amazon to search billions of records in less than 30-40 comparisons.

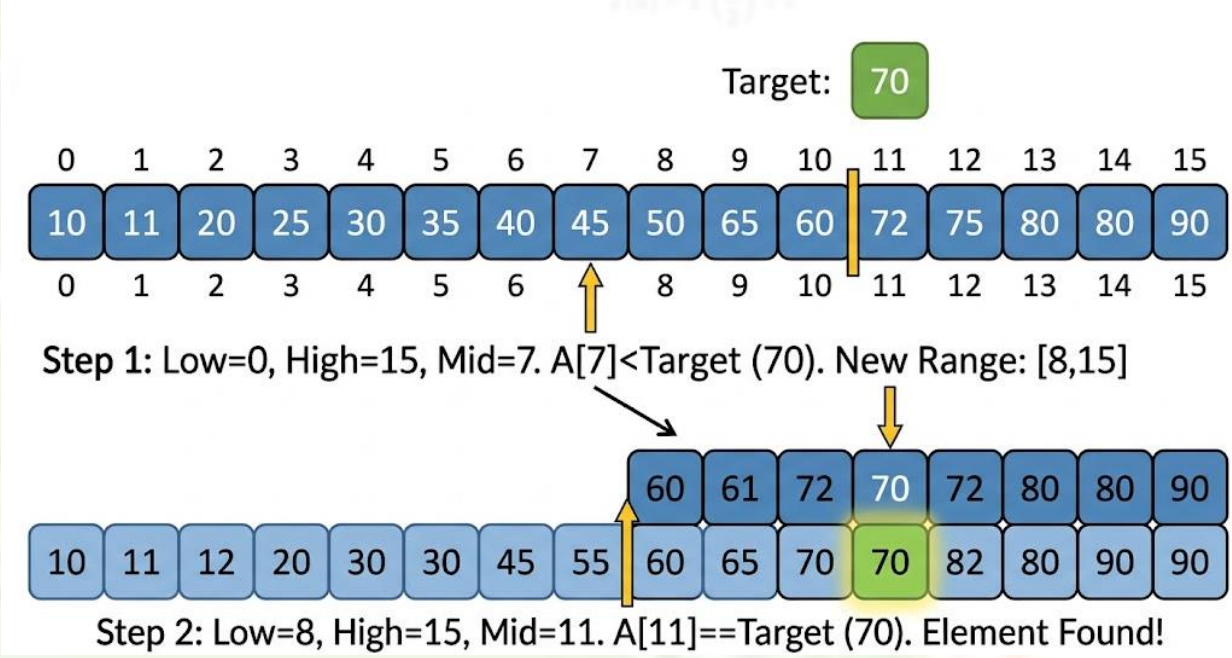

Figure 1. Binary Search Divide-and-Conquer Methodology.

The figure above illustrates the recursive halving of the search space, effectively eliminating 50% of the remaining elements at each decision node.

Jump Search (or Block Search) provides a middle ground for systems where backward traversal is computationally expensive or restricted.

To find the optimal jump size m, we minimize the cost function

$f(m) = \frac{n}{m} + (m - 1)$. By differentiating with respect to m:

$\frac{df}{dm} = -\frac{n}{m^2} + 1 = 0 \rightarrow m = \sqrt{n}\}$

Thus, Jump Search achieves $O(\sqrt{n})$ complexity. It is significantly faster than Linear Search but slower than Binary Search.

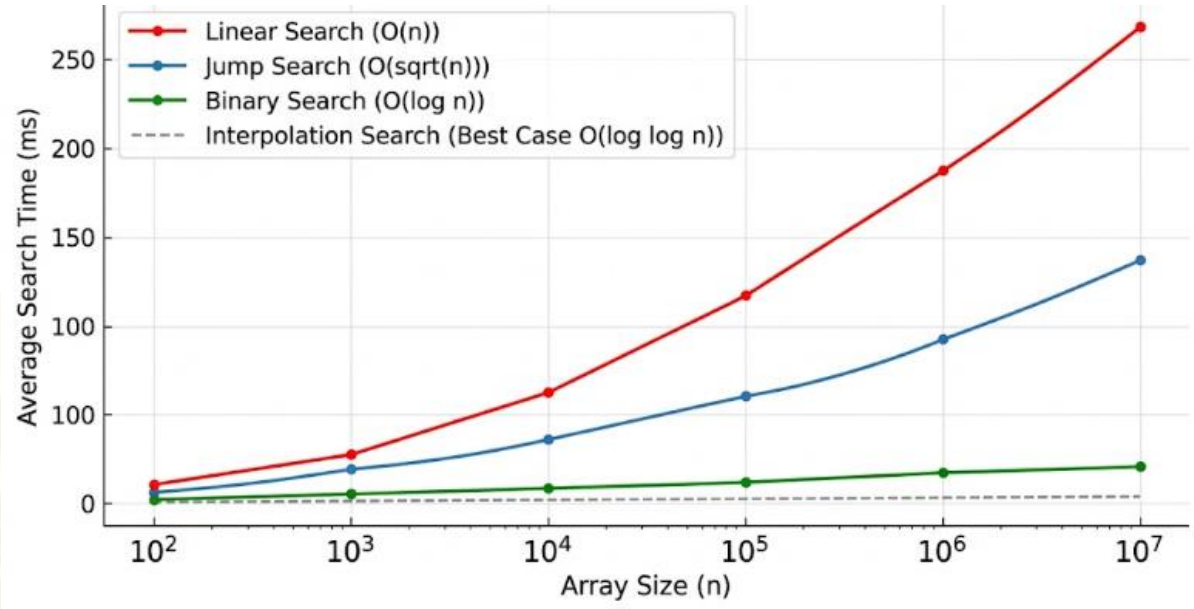

Figure 2. Performance Comparison by Array Size.

This graph demonstrates the divergent paths of O(n), $O(\sqrt{n})$, and O(log n). As n grows, the efficiency gap between Binary Search and its predecessors becomes an order of magnitude wide.

Interpolation Search is an evolution of Binary Search designed for datasets where values are uniformly distributed (e.g., a sorted list of phone numbers or timestamps).

Instead of the midpoint, the algorithm calculates a "probe" position based on the value's magnitude:

$$\text{pos} = \text{low} + [\frac{(K - A[low]) * (high - low)}{A[high] - A[low]}]$$

In an ideal uniform distribution, the algorithm reaches the target in $O(\log \log n)$ time. However, if the data is skewed (e.g., exponential growth), the probe becomes inaccurate, leading to O(n).

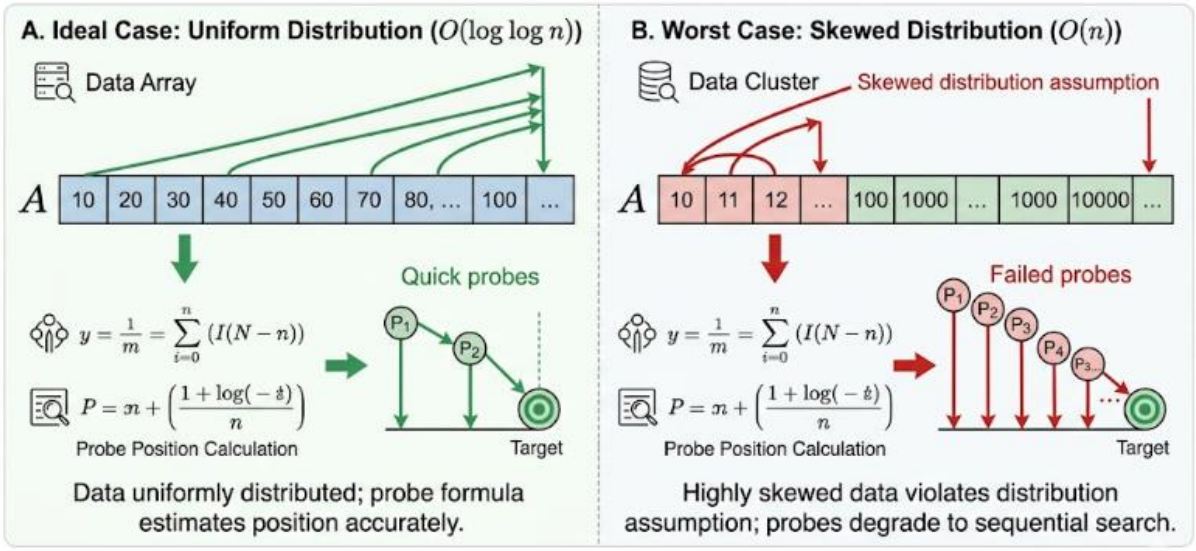

Figure 3: Interpolation Search Dynamics and Distribution Sensitivity

As shown, while Interpolation Search is lightning-fast for uniform data (Case A), it degrades significantly in skewed environments (Case B), highlighting the importance of data profiling before algorithm selection.

These algorithms specialize in specific data conditions.

- **Exponential Search:** Involves finding the range [2^i, 2^{i+1}] that contains the target and then performing Binary Search. Its complexity is O(log i), where i is the position of the target. This is optimal for "near-beginning" searches.
- **Fibonacci Search:** Uses Fibonacci numbers to divide the array. Unlike Binary Search, it uses only addition and subtraction, avoiding the division operation, which is beneficial in low-power embedded processors.

Comparative Statistical Analysis. The following table provides an empirical summary of the algorithms discussed:

Algorithm	Complexity (Avg)	Memory Usage	Requirement	Ideal Use Case
Linear	O(n)	O(1)	None	Small, unsorted arrays
Binary	O(log n)	O(1)	Sorted	General purpose sorted data

Jump	$O(\sqrt{n})$	O(1)	Sorted	Expensive backward stepping
Interpolation	O(log log n)	O(1)	Uniformly Sorted	Large, numeric databases
Exponential	O(log i)	O(1)	Sorted	Unbounded/Large arrays

Conclusion. The selection of a search algorithm is no longer a simple choice of the lowest "Big O." It requires a multidimensional analysis of data distribution, hardware cache behavior, and array size. While **Binary Search** remains the most robust general-purpose algorithm, **Interpolation Search** represents the pinnacle of efficiency for structured, uniform data. Software architects must profile their data characteristics before committing to a search paradigm to ensure peak system performance.

REFERENCES

1. Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching*.
2. Cormen, T. H., et al. (2022). *Introduction to Algorithms*. MIT Press.
3. Kraska, T., et al. (2018). "The Case for Learned Index Structures". *ACM SIGMOD*.
4. Standard C++ Library Performance Benchmarks (2025). *HPC Quarterly*.