# 4

# Searching algorithms

**Chapter Objectives**

After completing this chapter, you will be able to:

- Explain the purpose of searching algorithms and their role in efficient data processing
- Implement and compare a range of searching algorithms using C++ containers
- Assess performance trade-offs based on data size, ordering, and access patterns

Searching algorithms are techniques used to find a specific value inside a collection of data, such as an array or a vector. Imagine having a list of numbers or names and wanting to determine whether a particular item exists or where it is located. A searching algorithm defines the systematic way in which a computer examines that data. The simplest approach is to check each element one by one until a match is found, which works on any dataset but can become inefficient for large collections. More efficient searching methods take advantage of how the data is organized; for example, when data is sorted, a search can skip large portions of the dataset and complete much faster. In practice, searching algorithms play a critical role in enabling programs to retrieve information quickly and efficiently, making them a fundamental building block in problem solving and software development.

**Scope of the Chapter**

At the outset, it is important to note that the searching algorithms discussed in this chapter are primarily intended for data at rest, where the dataset remains stable during the search process. These algorithms assume that the collection does not change frequently while searches are being performed. When data is continuously inserted, deleted, or updated, maintaining the assumptions required by these algorithms can introduce additional overhead, making them less suitable in such dynamic environments. In applications involving constantly changing data, alternative data structures and search strategies are typically more appropriate. Such algorithms and data structures designed for dynamic environments will be discussed in the next chapter.

Before analyzing specific algorithms, it is important to understand how algorithm performance is measured. Three fundamental concepts are commonly used to evaluate and compare algorithms: time complexity, space complexity, and Big-O notation. Together, these concepts describe how an algorithm's execution time and memory usage grow as the size of the input increases.

**Time complexity**

Time complexity measures how the running time of an algorithm increases as the size of the input grows. Instead of focusing on actual execution time in seconds, time complexity counts the number of basic operations performed by the algorithm relative to the input size. This approach allows algorithms to be compared independently of hardware speed, programming language, or system environment.

**Space complexity**

Space complexity measures the amount of additional memory an algorithm requires as the input size increases. It focuses on the extra memory used by the algorithm during execution, such as temporary variables, auxiliary data structures, or recursion stacks, rather than the memory needed to store the input itself.

**Big-O notation**

Big-O notation is a mathematical way of expressing time and space complexity. It describes the upper bound on an algorithm's growth rate as the input size becomes large. Big-O notation focuses on the dominant term and ignores constants and lower-order terms, allowing algorithms to be compared based on their scalability. For example,

***Quartic Growth (Power of 4):*** If an algorithm performs $5n^4 + 2n^2 + 100n + 50$ operations, the term with the highest power of $n$, namely $n^4$, dominates the growth of the function. As the input size $n$ increases, the effects of lower-order terms and constant values become negligible in comparison. Consequently, the algorithm is classified as having a time complexity of $O(n^4)$, meaning that the number of operations increases in proportion to the fourth power of the input size.

***Cubic Growth (Power of 3):*** If an algorithm performs $7n^3 + 4n^2 + 20n + 15$, the $n^3$ term dominates the growth rate, and the algorithm is classified as $O(n^3)$.

***Quadratic Growth (Power of 2):*** If an algorithm performs $3n^2 + 10n + 25$, the dominant term is $n^2$, so the time complexity is $O(n^2)$.

***Linear Growth (Power of 1):*** If an algorithm performs $6n + 40$ operations, the dominant term is $n$. As the input size $n$ increases, the constant term becomes insignificant in comparison. Therefore, the algorithm is classified as $O(n)$, which indicates that the number of operations grows linearly with the input size.

***Constant Growth (O(1)):*** If an algorithm performs a fixed number of operations, such as 12, regardless of the input size $n$, its running time remains constant as $n$ increases. Because the execution time does not depend on the size of the input, the algorithm is classified as $O(1)$, which is known as constant-time complexity.

This chapter explores several commonly used searching algorithms, discussing how each one works, along with its advantages and limitations. The performance of these algorithms is analyzed using best-case, average-case, and worst-case scenarios, allowing readers to understand when a particular searching technique is most appropriate and how algorithm choice impacts efficiency.

## 4.1 Linear Search Algorithm

Linear search, also known as sequential search, is the simplest searching algorithm. It works by examining each element in a collection one by one, starting from the beginning and continuing until the target element is found or the end of the data is reached. Because linear search makes no assumptions about how the data is organized, it works on both sorted and unsorted datasets and is easy to understand and implement. However, this simplicity comes at a cost. For large collections, linear search can be slow because it may need to examine every element in the dataset. In addition, when duplicate elements are present, a standard linear search typically finds only the first occurrence of the target value; to find all occurrences or determine how many times the value appears, the algorithm must examine every element in the dataset. More efficient searching algorithms achieve better performance by using the structure of the data. For instance, when data is sorted, a search can skip many elements and complete much faster.

The best-case scenario for linear search occurs only when the problem is defined as finding any single occurrence of the target element and the target appears at the first position in the dataset. In this situation, the algorithm performs just one comparison, resulting in a time complexity of $O(1)$.

However, when the dataset contains duplicate elements and the search task requires identifying all occurrences, counting duplicates, or ensuring completeness, linear search cannot terminate early. In such cases, the algorithm must examine every element in the dataset, and a constant-time best case does not exist; even under the most favorable conditions, the time complexity remains $O(n)$.

**Case assumptions** Best-case, average-case, and worst-case analyses are always based on specific assumptions. When discussing these cases, be clear about what the algorithm is allowed to do and what the problem requires (such as early termination or handling duplicates). Always ensure that your assumptions match the scenario you are describing.

The following program uses a linear search to scan a vector and collect the indices of all elements that match a given target value. It demonstrates how passing a vector by reference allows the function to store results directly in the caller's vector while efficiently handling duplicate elements.

**Code Example 4-1**: Linear search algorithm with duplicate member search

```
 1 #include <iostream>
 2 #include <vector>
 3 using namespace std;
 4 
 5 void linearSearch(const vector<int>& v, vector<int>& d, int key) {
 6     for (int i = 0; i < v.size(); i++) {
 7         if (v[i] == key)
 8             d.push_back(i);
 9     }
10 }
11 
12 int main() {
13     vector<int> data = {3, 8, 15, 23, 15};
14     vector<int> data_index;
15     linearSearch(data, data_index, 15);
16     for (int x : data_index) {
17         cout << "Index: " << x << endl;
18     }
19 }
```

***Lines 1-3:*** *Include the* `<iostream>` *and* `<vector>` *header files, which are required for input/output operations using* `cout` *and for using the* `std::vector` *container, respectively.*
***Line 5:*** *Defines the* `linearSearch` *function, which performs the linear search operation. The first parameter,* `const vector<int>& v`*, represents the input dataset and is passed by constant reference to prevent modification and avoid unnecessary copying. The second parameter,*

`vector<int>& d`*, is passed by reference and acts as an alias to the vector supplied by the calling function, allowing the function to store results directly in that vector. The third parameter,* `key`*, specifies the value to be searched.*
***Lines 6-10:*** *Implement a for loop that iterates through the vector* `v` *from the first element to the last. Each element is compared with the target value, and if a match is found, the current index is appended to vector* `d`*, enabling the function to record all positions where the target value appears.*
***Line 12:*** *Marks the beginning of the* `main` *function, which is the entry point of the program.*
***Line 13:*** *Initializes a vector named data with a set of integer values, including duplicates.*
***Line 14:*** *Declares an empty vector named* `data_index`*, which will be used to store the indices of all matching elements.*
***Line 15:*** *Calls the* `linearSearch` *function, passing* `data` *as the input vector,* `data_index` *as the output vector, and 15 as the target value. Because* `data_index` *is passed by reference, it is modified directly inside the function.*
***Lines 16-18:*** *Use a range-based for loop to iterate through the indices stored in* `data_index` *and print each index to the console.*

When working with unsorted data, linear search is often the most practical choice because it can be applied directly without any preprocessing. Although faster searching algorithms such as binary search exist, they require the data to be sorted first. Sorting a large dataset itself incurs additional computational cost, which may outweigh the benefits of faster searching if the search operation is performed only once or infrequently. Therefore, linear search is often preferred for unsorted data when the dataset is small or when the overhead of sorting is not justified. For large datasets with frequent search operations, investing time in sorting the data or using more advanced data structures can lead to better overall performance. These approaches and their performance trade-offs will be discussed in detail in the upcoming sections.

## 4.2 Binary Search Algorithm

The main motivation behind binary search is to search efficiently by reducing the number of comparisons needed to find an element. Instead of examining every item one by one, binary search takes advantage of sorted data to eliminate half of the remaining elements at each step. For example, if we want to find the number 144 in the sorted list [101, 115, 123, 130, 144, 150, 162, 170], a linear search may require checking many elements before reaching the target. Binary search, however, first compares the middle element (130) with 144, immediately discards the left half, and then finds 144 in the next comparison. This divide-and-conquer approach reduces the time complexity from $O(n)$ to $O(\log n)$, making binary search especially powerful for large datasets where performance and scalability matter.

In some textbooks, the time complexity of binary search is explicitly written as $O(\log_2 n)$ to emphasize that the algorithm repeatedly halves the search space at each step. However, in Big-O notation, the base of the logarithm is typically omitted because logarithms with different bases differ only by a constant factor. Since Big-O analysis focuses on the asymptotic growth rate of an algorithm and ignores constant multipliers, $O(\log_2 n)$ and $O(\log n)$ are considered equivalent. For example, if an array contains 1,024 elements, binary search requires at most 10 comparisons because $\log_2 1024 = 10$. Using a different base, such as base 10, gives $\log_{10} 1024 \approx 3$, which differs only by a constant factor. Despite this numerical difference, both expressions describe the same logarithmic growth behavior, which is why the standard convention in algorithm analysis is to write $O(\log n)$.

Although binary search significantly reduces the number of comparisons relative to linear search, this improvement is achieved only under specific conditions. Binary search is applicable exclusively to data that is sorted according to a well-defined ordering; without this property, the algorithm cannot guarantee correct results. Furthermore, the data must be stored in a structure that supports efficient random access, such as an array or vector, allowing constant-time access to the middle element. A consistent comparison operation is also required to establish the ordering between elements. When these preconditions are satisfied, binary search operates by repeatedly dividing the search interval in half, thereby achieving logarithmic time complexity.

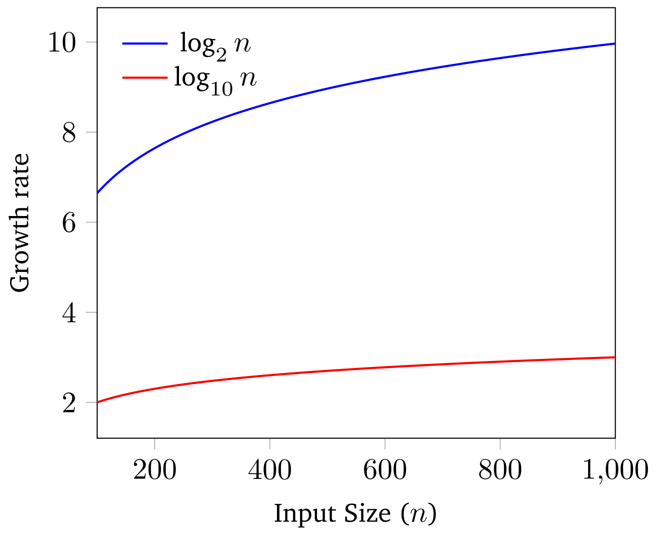

**FIGURE 4.1** *This figure compares the growth behavior of the logarithmic functions $\log_2 n$ and $\log_{10} n$ as the input size $n$ increases. The x-axis represents the input size, while the y-axis represents the growth rate in terms of the number of operations. Both curves increase slowly, illustrating the efficiency of logarithmic growth, with $\log_2 n$ growing faster than $\log_{10} n$ by a constant factor. The similar shapes of the two curves emphasize that logarithmic functions differ only by a constant multiplier, which is why logarithmic time complexity is expressed as $O(\log n)$ without specifying the base.*

Binary search works by systematically eliminating half of the remaining possibilities at each step. Instead of scanning elements one by one, the algorithm compares the target value with the element located at the middle of the search range.

When applying binary search, the method for identifying the middle element depends on whether the number of elements in the search range is odd or even. If the range contains an odd number of elements, there is a single, well-defined middle element, which is selected directly. In contrast, when the number of elements is even, no unique middle element exists; instead, two central candidates are present. In this case, binary search resolves the ambiguity by consistently selecting one of these candidates—most commonly the lower middle—using integer division when computing the midpoint. Importantly, the choice of lower or upper middle does not affect the correctness or efficiency of the algorithm, provided the selection rule is applied consistently. In both cases, the selected middle element allows binary search to partition the data into two smaller subranges, ensuring that the search space is reduced at each step and preserving the algorithm's logarithmic time complexity.

If the target is equal to the middle element, the search is complete. If the target is smaller, the algorithm ignores the entire right half of the data; if the target is larger, it ignores the entire left half. Because the data is sorted, these discarded halves are guaranteed not to contain the target.

This process repeats on the remaining half of the data, continually narrowing the search range. With each comparison, the search space is reduced by a factor of two. As a result, even for large datasets, binary search locates an element using very few comparisons, which explains its logarithmic efficiency.

**Example (Even Number of Elements)**

Consider the following sorted array with 8 elements: Suppose we want to search for 18.

```
Index:  0   1   2   3   4   5   6   7
Data:   3   7  10  14  18  21  25  30
```

***Step 1: Choose the Middle Element***

```
- Low index = 0
- High index = 7
- Middle index = (0 + 7) // 2 = 3 → this is an integer division
- Middle element = 14
```

Since 18 > 14, all elements to the left of index 3 (including 14) are discarded.

***Step 2: Search the Right Half***

```
Index:  4   5   6   7
Data:  18  21  25  30
- Low index = 4
- High index = 7
- Middle index = (4 + 7) // 2 = 5 → this is an integer division
- Middle element = 21
```

Since $18 < 21$, all elements to the right of index 5 (including 21) are discarded.

***Step 3: Final step***

```
Index:  4
Data:   18
```

The target value 18 is found.

Binary search computes the middle index using integer division because it operates on discrete index positions in an array or similar data structure, where indices must be integers and any fractional result is discarded. When the number of elements is odd, this calculation yields a unique middle index; when the number is even, it consistently selects one of the two central indices. Binary search does not require a unique middle element for correctness—what matters is that the chosen index allows the algorithm to eliminate part of the search space at each step. This deterministic selection ensures steady progress and preserves the correctness and logarithmic time complexity of the algorithm.

Binary search exhibits different performance characteristics depending on when the target element is found within the search process. In the best case, the target element is located at the middle position of the search range during the very first comparison. In this situation, binary search completes immediately, resulting in a time complexity of $O(1)$.

In the average case, the target element is found after several iterations, as the algorithm repeatedly halves the search space. On average, binary search requires a logarithmic number of comparisons relative to the size of the dataset, giving an average-case time complexity of $O(\log n)$.

In the worst case, the target element is either located at the deepest level of the search process or is not present in the data at all. Even in this scenario, binary search continues halving the search space until it is exhausted, leading to a worst-case time complexity of $O(\log n)$. This consistent logarithmic behavior across average and worst cases highlights the efficiency of binary search for large, sorted datasets.

Keep this in mind that binary search requires the data to be sorted before it can be applied, which introduces an additional preprocessing step. If the data is initially unsorted, it must first be arranged according to a well-defined ordering, and this sorting process incurs an extra computational cost. However, once the data is sorted, binary search can be applied efficiently and repeatedly, achieving logarithmic time complexity. In situations involving multiple search operations on the same dataset, the one-time cost of sorting is often offset by the significant performance gains of optimized binary searches.

The following pseudocode presents a language-independent description of the binary search algorithm. It outlines the fundamental logic of the algorithm without relying on the syntax of any specific programming language, allowing the focus to remain on the underlying problem-solving approach.

**Code Example 4-2**: Pseudocode of binary search algorithm

```
BinarySearch(A, n, key)  //A:Array), n:array length, key:element to find
    low ← 0
    high ← n - 1

    while low ≤ high do
        mid ← (low + high) / 2        // integer division

        if A[mid] = key then
            return mid               // key found
        else if A[mid] < key then
            low ← mid + 1            // search right half
        else
            high ← mid - 1           // search left half
        end if
    end while

    return -1                        // key not found
```

### 4.2.1 Handling Duplicate Elements in Binary Search

A standard binary search can only tell you whether a value exists and may return any one of its occurrences when duplicates are present. It cannot directly return all indices of repeated elements. When the data is sorted, duplicates naturally cluster together. The following example illustrates the binary search process in the presence of duplicate elements.

**Example (Sorted array with duplicate members)**

```
Index:   0   1   2   3   4   5   6
Array:   3   5   7   7   7   9  12
Key = 7
```

*Part A: Find the First Occurrence (Leftmost 7)*

```
Step 1
- low = 0, high = 6
- mid = (0+6)/2 = 3
- A[mid] = A[3] = 7  (match found)
But we keep going left to find the first one.
- record ans = 3
- move left: high = mid - 1 = 2
```

```
Step 2
- low = 0, high = 2
- mid = (0+2)/2 = 1
- A[mid] = A[1] = 5
Since 7 > 5, go right: low = mid + 1 = 2
```

```
Step 3
- low = 2, high = 2
- mid = 2
- A[mid] = A[2] = 7  (match)
- record ans = 2
- move left: high = mid - 1 = 1
Stop (low > high)
First occurrence = 2
```

*Part B: Find the Last Occurrence (Rightmost 7)*

```
Step 1
- low = 0, high = 6
- mid = 3
- A[mid] = 7 (match)
But we keep going right to find the last one.
- record ans = 3
- move right: low = mid + 1 = 4
```

```
Step 2
- low = 4, high = 6
- mid = (4+6)/2 = 5
- A[mid] = A[5] = 9
Since 7 < 9, go left: high = mid - 1 = 4
```

```
Step 3
- low = 4, high = 4
- mid = 4
- A[mid] = 7  (match)
- record ans = 4
- move right: low = mid + 1 = 5
Stop (low > high)
Last occurrence = 4
```

*Final Result: All Indices*

```
Now we have:
- first = 2
- last = 4
So all occurrences are indices:
2, 3, 4
```

We have already discussed the Big-O notation of binary search in the general case. The following examines how the presence of duplicate elements influences that analysis and under what conditions the overall time complexity changes.

Let $n$ denote the total number of elements in a sorted array, and let $k$ denote the number of occurrences of a given key. When binary search is used solely to determine the presence of a key, the algorithm repeatedly halves the search space, resulting in a time complexity of

$$T(n) = O(\log n) \tag{4.1}$$

regardless of whether duplicate elements are present.

When duplicate elements exist and the objective is to locate a specific boundary—such as the first or last occurrence—the algorithm can be modified to continue searching after a match is found. Each such boundary search still requires $O(\log n)$ time, as the search space is reduced by a factor of two at each step.

However, if the objective is to identify all occurrences of the key, the analysis changes. After determining the first and last occurrence indices in $O(\log n)$ time, all duplicate elements within this range must be examined or reported. This additional step incurs a cost proportional to the number of duplicates, $K$. Therefore, the total time complexity becomes

$$T(n, k) = O(\log n + k) \tag{4.2}$$

If the number of duplicates is bounded by a constant, that is, $k = O(1)$, the complexity simplifies to $O(\log n)$.

In contrast, if the number of duplicates grows with the input size, the linear term dominates. In the worst case, where all elements are identical $(k = n)$, the complexity reduces to

$$T(n, k) = O(\log n + n) = O(n) \tag{4.3}$$

This case-by-case analysis demonstrates that while binary search itself maintains logarithmic efficiency, retrieving all duplicate elements introduces an additional cost that depends directly on the number of duplicates.

The Table 4.1 summarizes how the time complexity of binary search varies when duplicate elements are present in the dataset and different search objectives are considered.

**TABLE 4.1** *Time Complexity of Binary Search in the Presence of Duplicate Elements*

| Scenario | Objective | Time Complexity |
|---|---|---|
| No duplicates | Find key in sorted array | $O(\log n)$ |
| Duplicates present | Find any one occurrence | $O(\log n)$ |
| Duplicates present | Find first occurrence | $O(\log n)$ |
| Duplicates present | Find last occurrence | $O(\log n)$ |
| Duplicates present | Find all occurrences | $O(\log n + k)$ |
| Worst case ($k = n$) | All elements identical | $O(n)$ |

Table 4.2 provides a comparative overview of linear search and binary search, highlighting their performance characteristics under different scenarios.

**TABLE 4.2** *Time Complexity Comparison of Linear Search and Binary Search*

| Search Algorithm | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Linear Search | $O(1)$[1] | $O(n)$ | $O(n)$ |
| Binary Search | $O(1)$[1,2] | $O(\log n)$ | $O(\log n)$ |

Binary search is actively used today in many real-world systems whenever data is ordered and queried repeatedly. Some common practical applications include:

***Standard library functions:*** Many programming languages provide built-in binary search utilities (for example, searching within sorted arrays or vectors). These are heavily used in performance-critical applications where fast lookups are required.

***Searching time-ordered data:*** System logs, transaction histories, and monitoring data are often stored in sorted order by timestamp. Binary search is used to quickly locate the first or last record within a given time range.

***Debugging and fault isolation:*** Developers use binary-search principles to isolate problems efficiently. A well-known example is narrowing down which change introduced a bug by repeatedly halving the range of possible causes.

***Databases and indexing systems:*** Although databases rely on advanced index structures, the core idea of binary search—rapidly narrowing a sorted search space—underpins many index lookup operations.

[1] The best-case time complexity of $O(1)$ assumes that the target element is uniquely identifiable and encountered immediately (for linear search) or at the first midpoint comparison (for binary search).

[2] Binary search requires the data to be sorted in advance. If the data is initially unsorted, an additional preprocessing step is required, typically with a time complexity of $O(n \log n)$. This sorting cost is often amortized when multiple search operations are performed on the same dataset.