Design and Analysis of Algorithms

Algorithm Design Techniques

- **Brute Force and Exhaustive Search**
- **Decrease-and-Conquer**
- **Divide-and-Conquer**
- Transform-and-Conquer
- Space and Time Trade-Offs
- Dynamic Programming
- Greedy Technique
- Iterative Improvement
- Backtracking
- Branch-and-Bound

Brute Force Algorithms

- Selection Sort
- Bubble Sort
- Sequential Search
- Brute-Force String Matching

Exhaustive Search Problems and Algorithms

- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem
- Depth-First Search
- Breadth-First Search

Searching

- The searching problem deals with finding a given value, called a search key, in a given set.
- There are plenty of searching algorithms to choose from.
- Searching algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.
- There is no single algorithm that fits all situations best.
- Some algorithms work faster than others but require more memory.
- Some are very fast but applicable only to sorted arrays.

Sequential Search

- We have already encountered a brute-force algorithm for the general searching problem: it is called sequential search.
- The algorithm simply compares successive elements of a given array with a given search key until either a match is encountered or the list is exhausted without finding a match.
- It is the simplest searching algorithm.

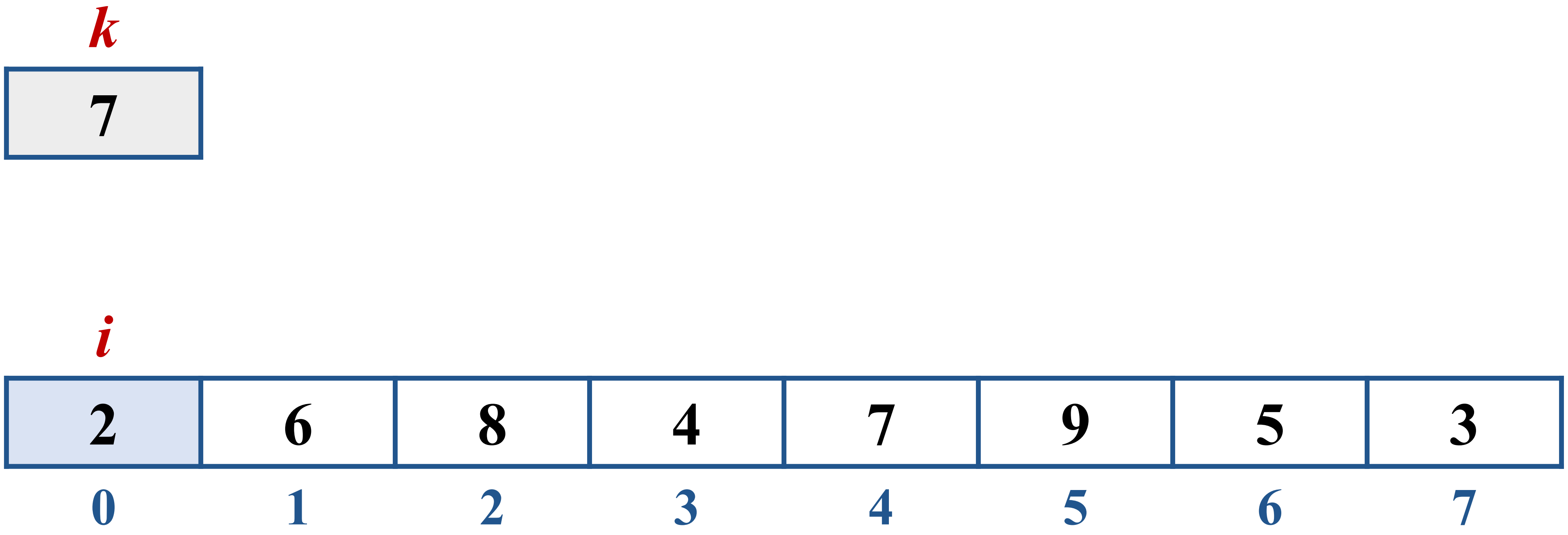
Sequential Search: Average-Case

k

7

2	**6**	**8**	**4**	**7**	**9**	**5**	**3**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Sequential Search: Average-Case



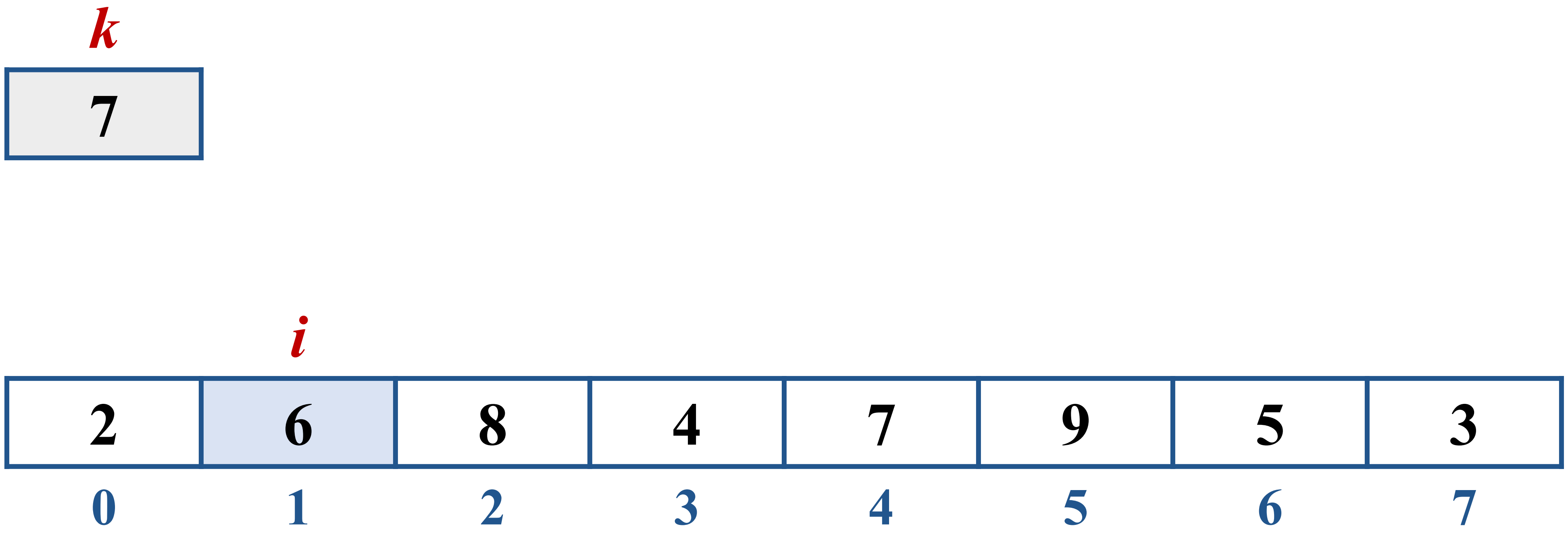
k

7

i

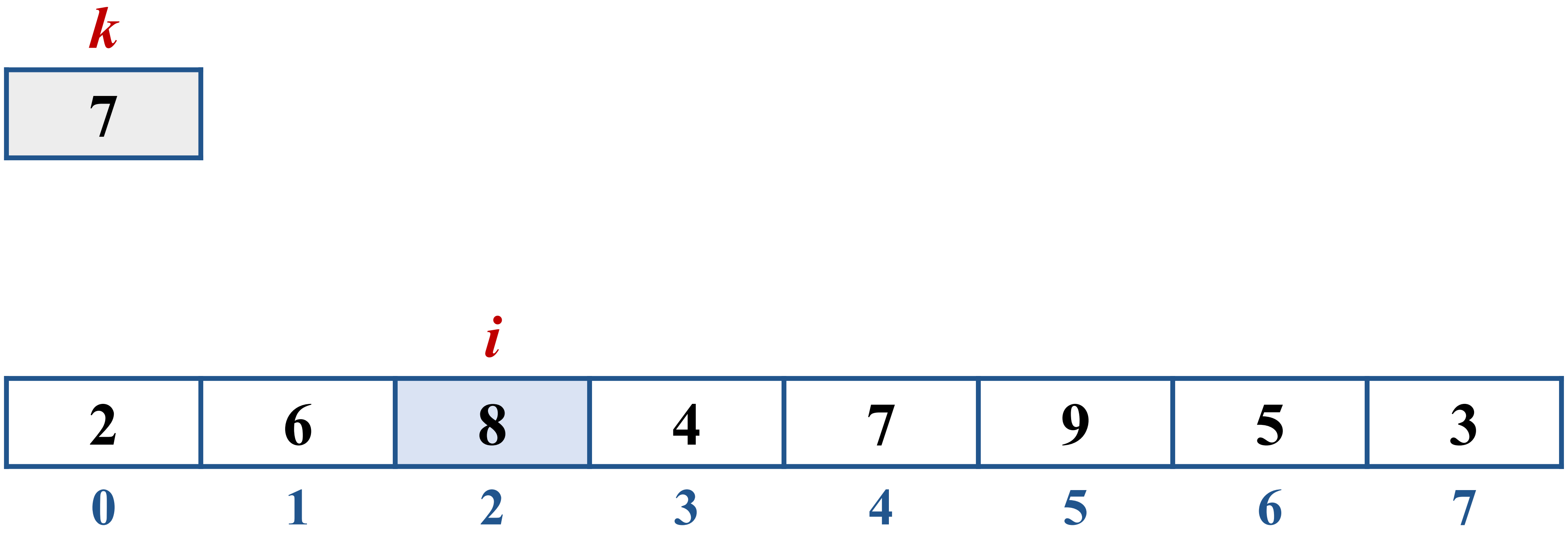
2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Average-Case



2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Average-Case



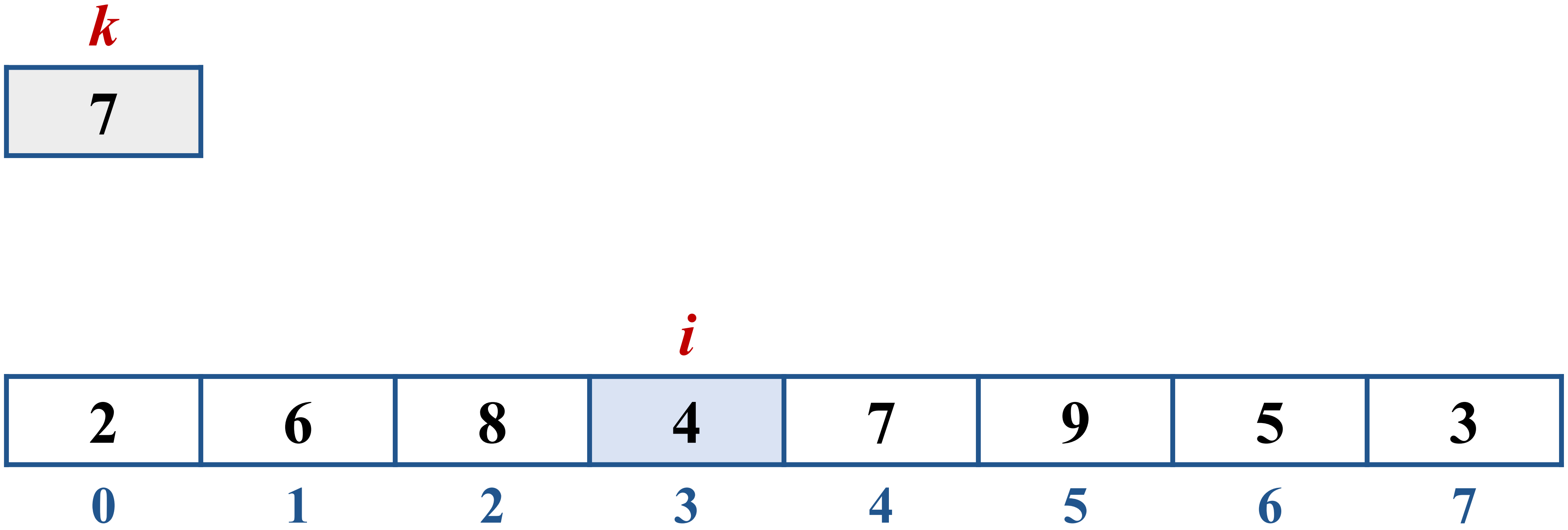
k

7

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Average-Case



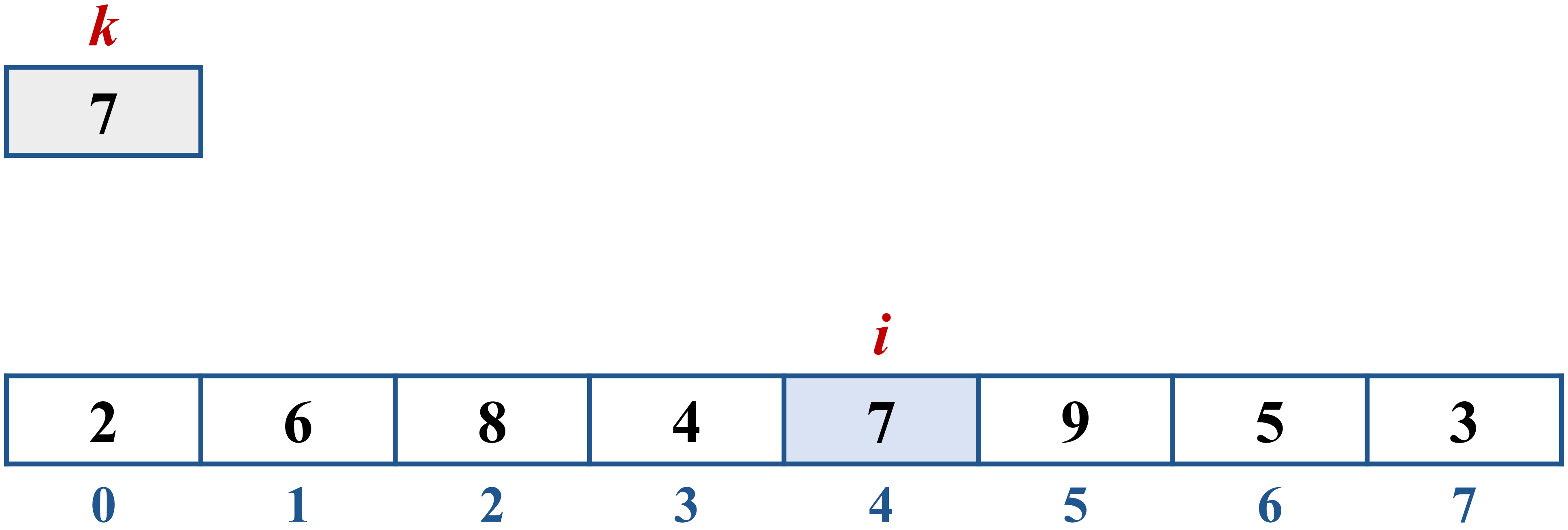
k

7

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Average-Case



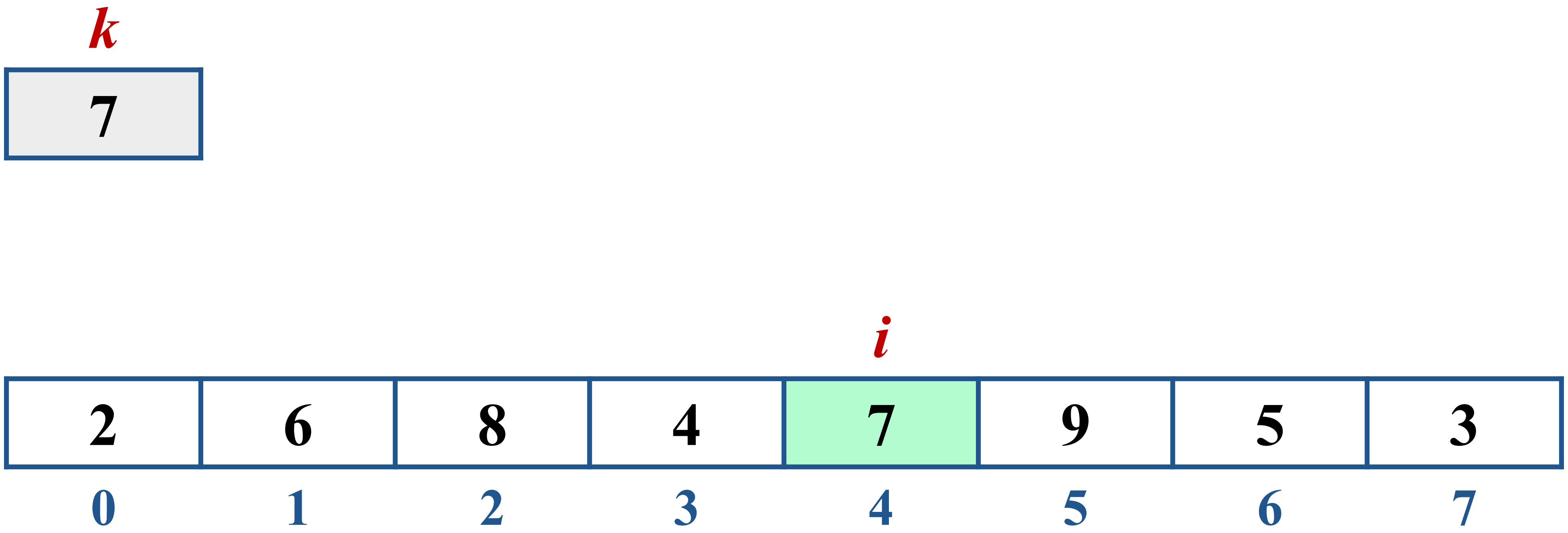
k

7

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Average-Case



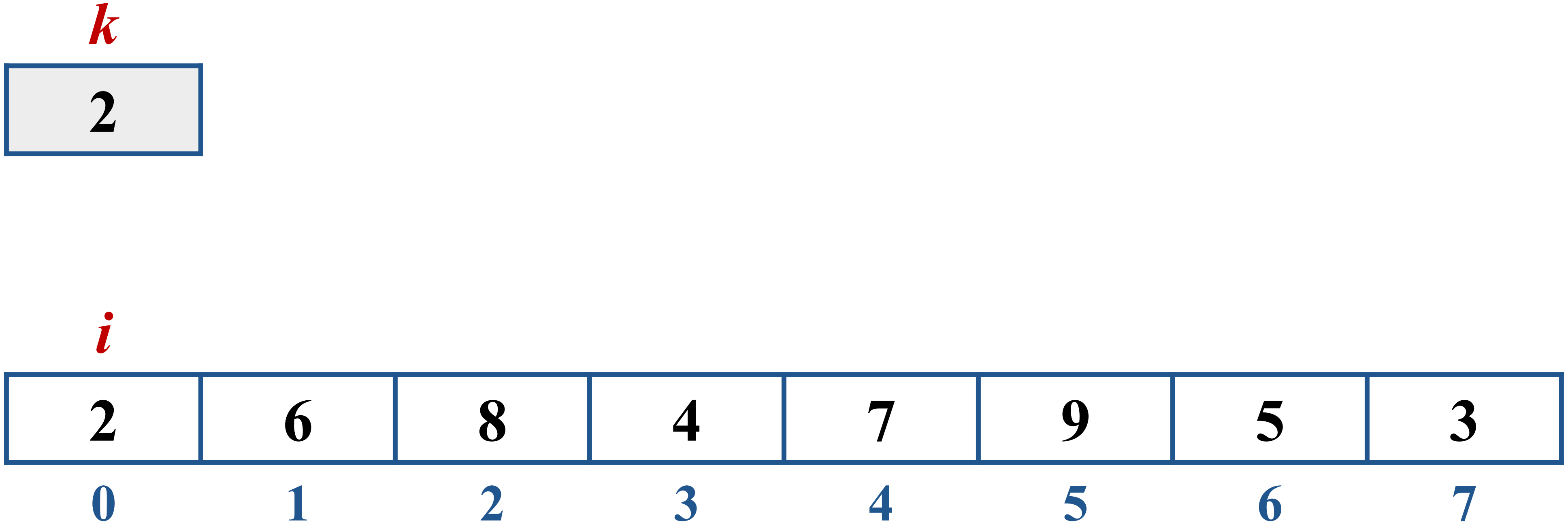
k

7

i

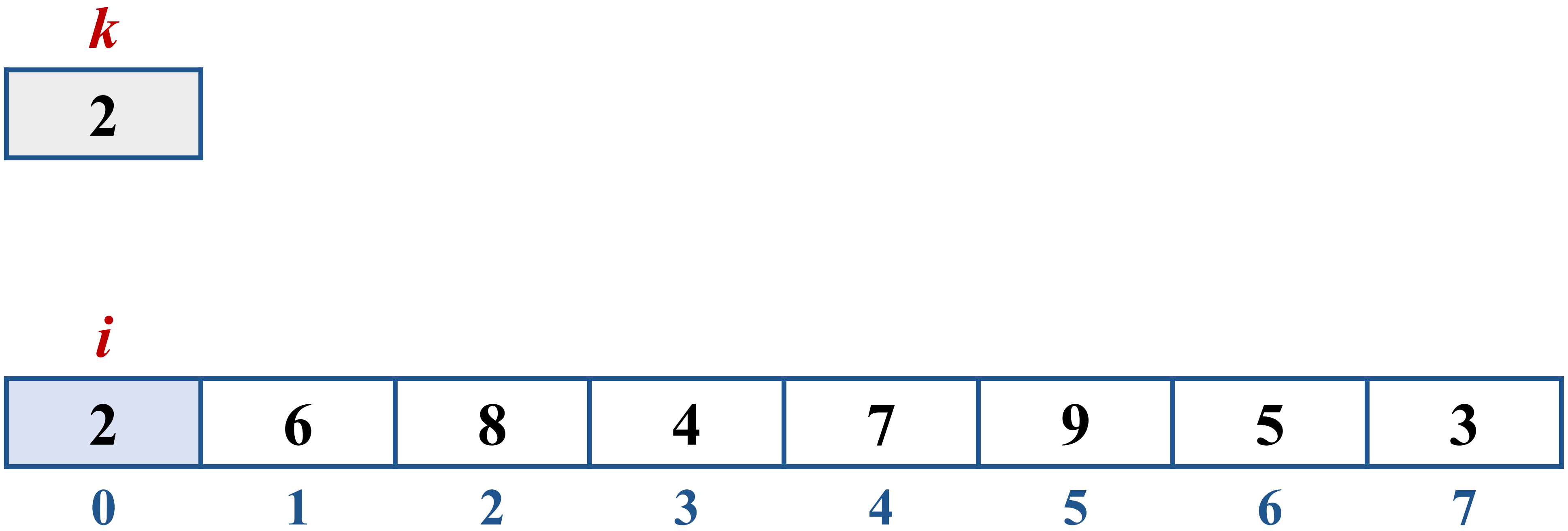
2	6	8	4	🟩 7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Best-Case



2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Best-Case



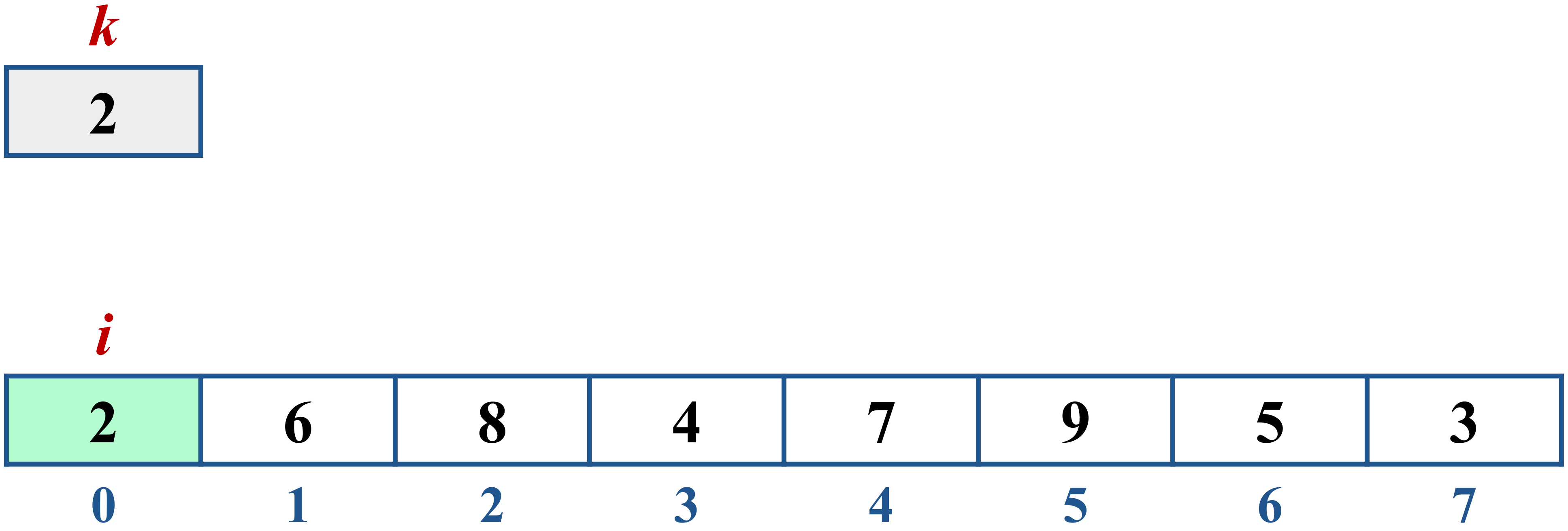
k

2

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Best-Case



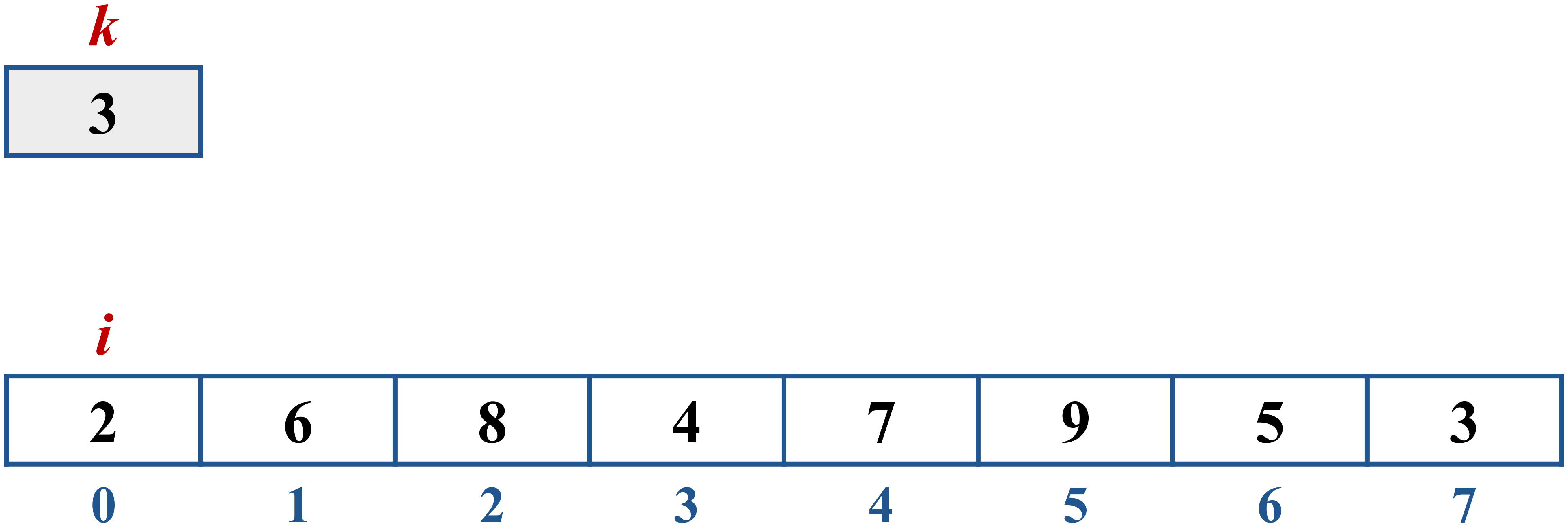
k

2

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



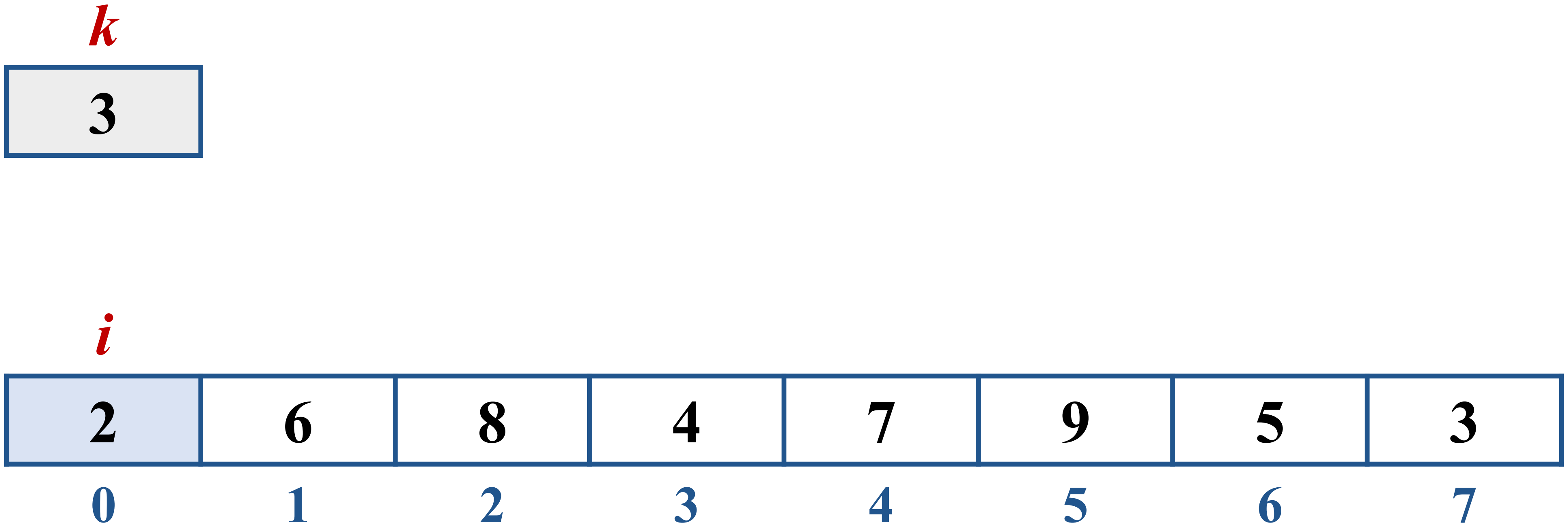
k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



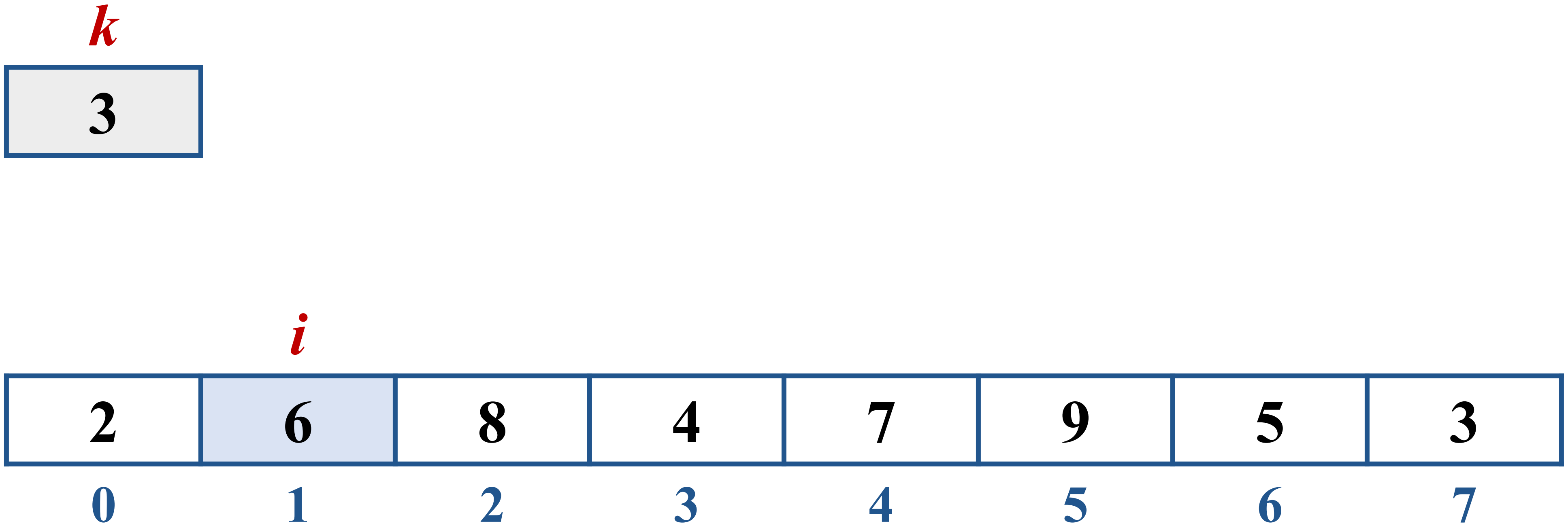
k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



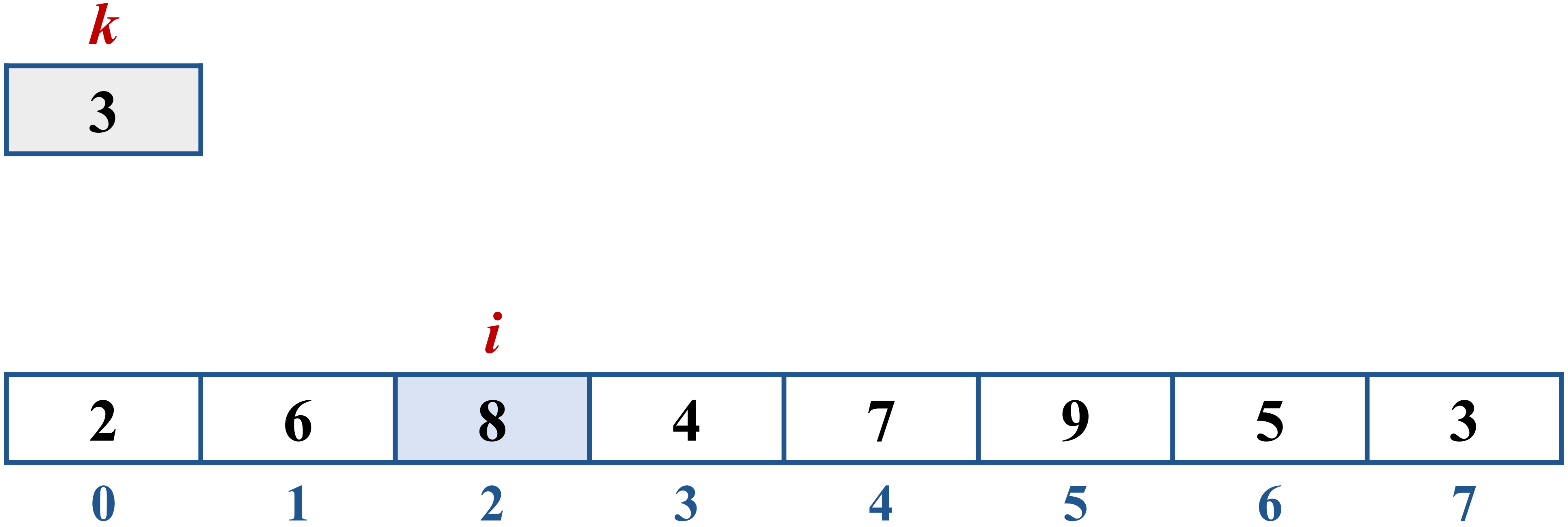
k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



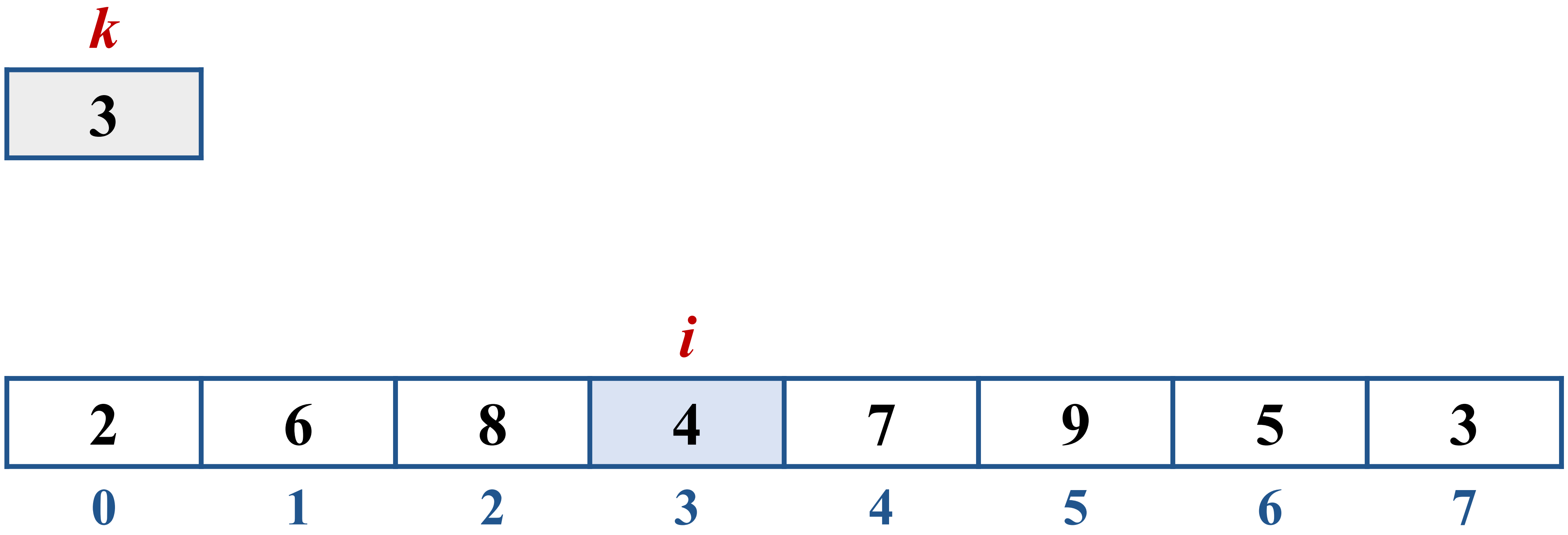
k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



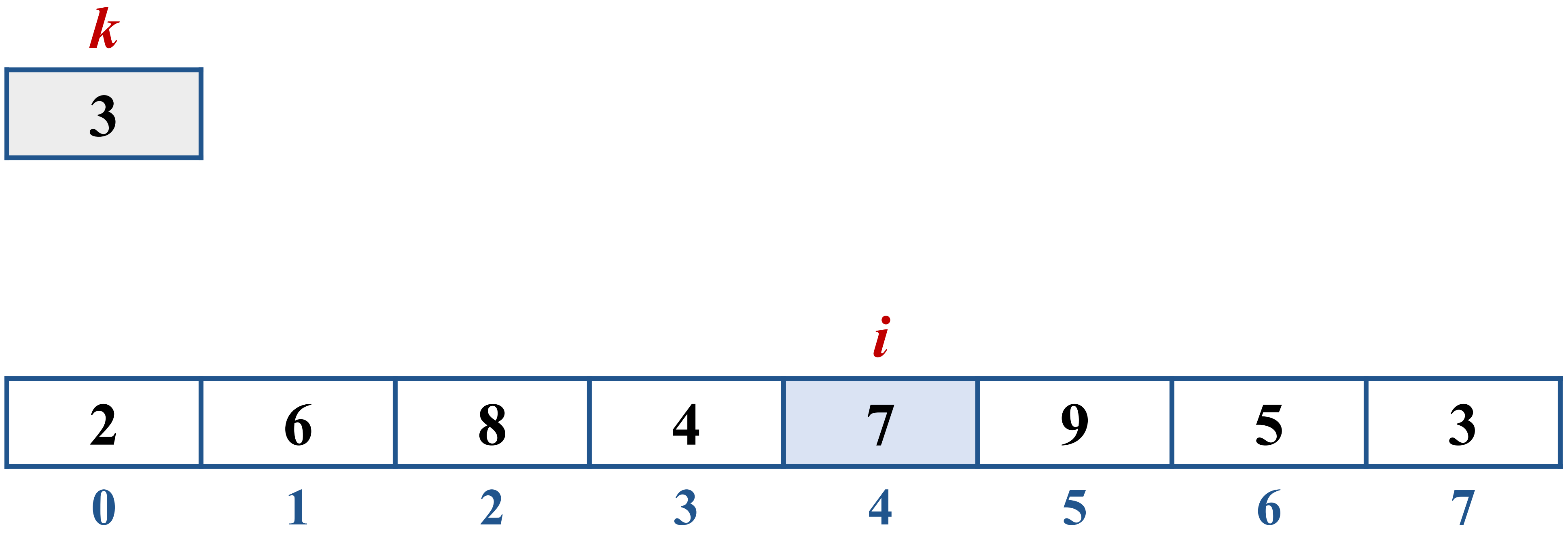
k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



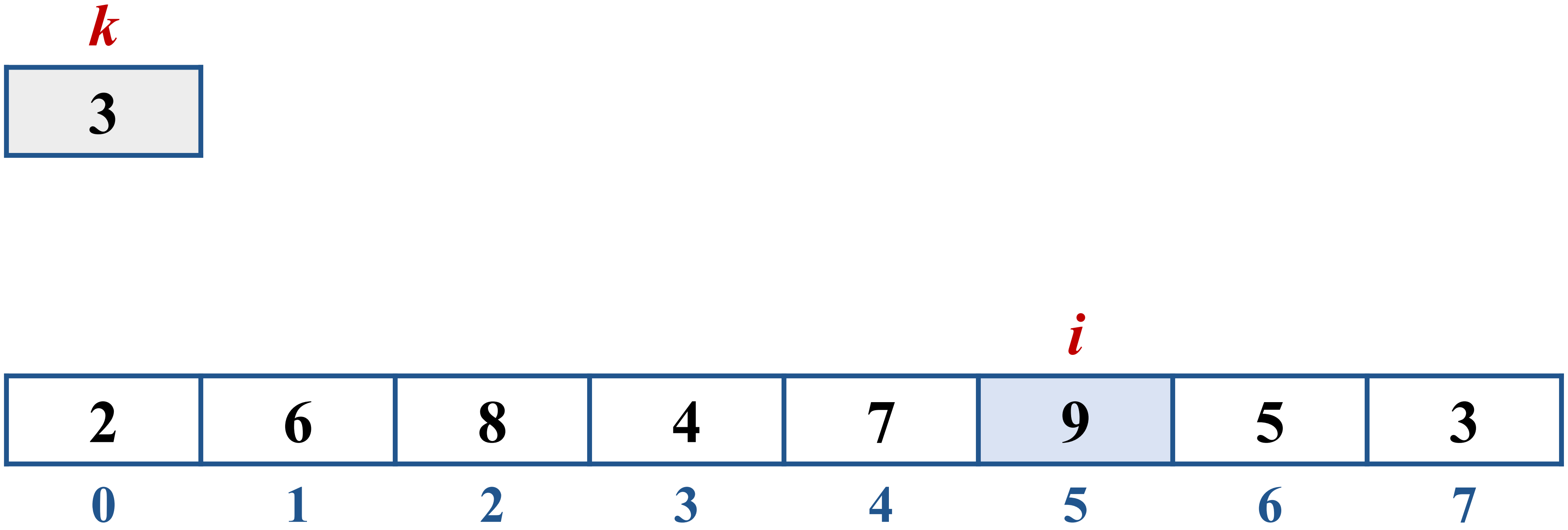
k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



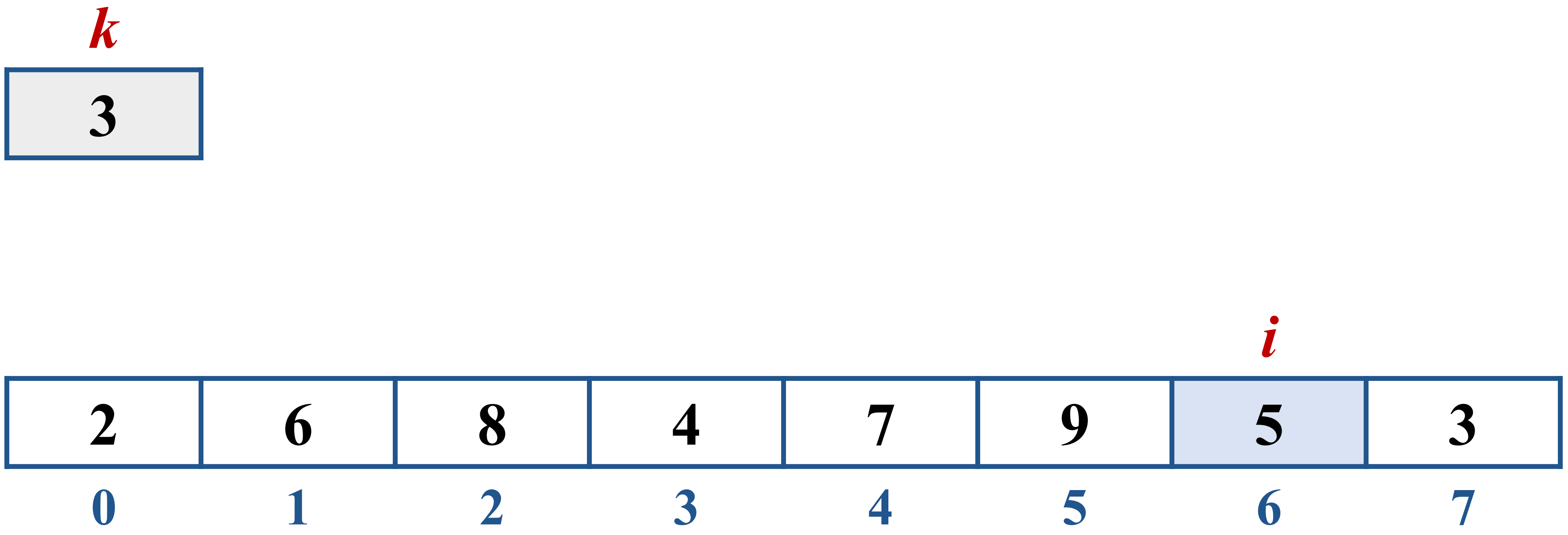
k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case

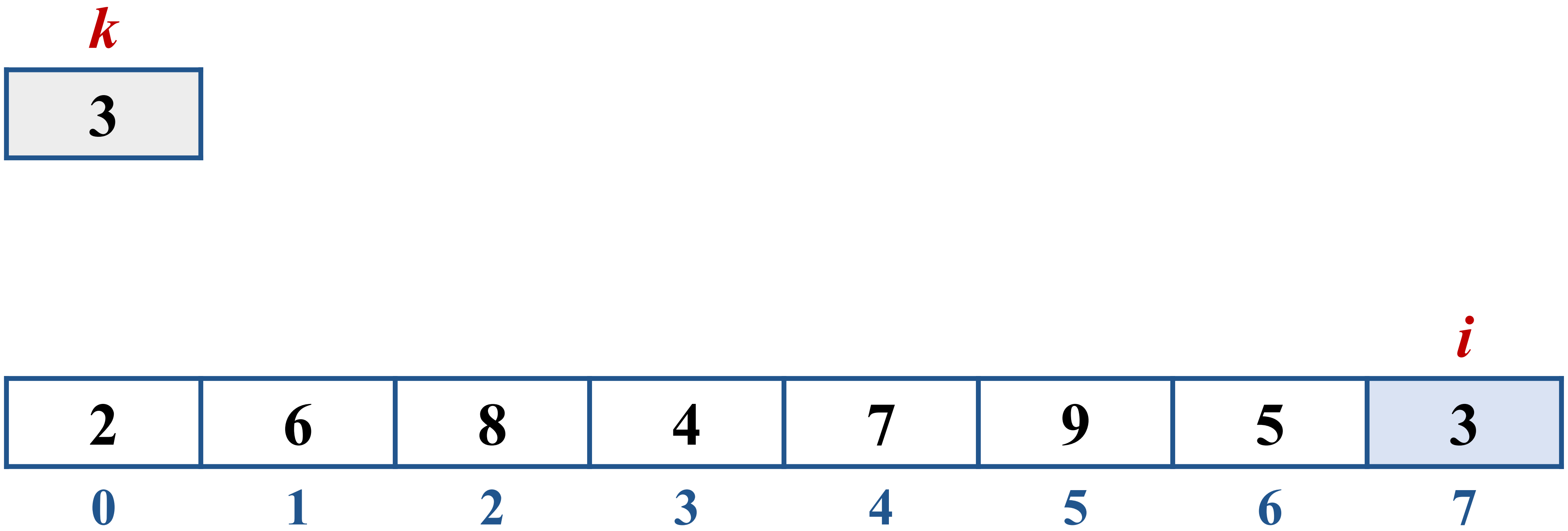


k

3

						i	
2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case

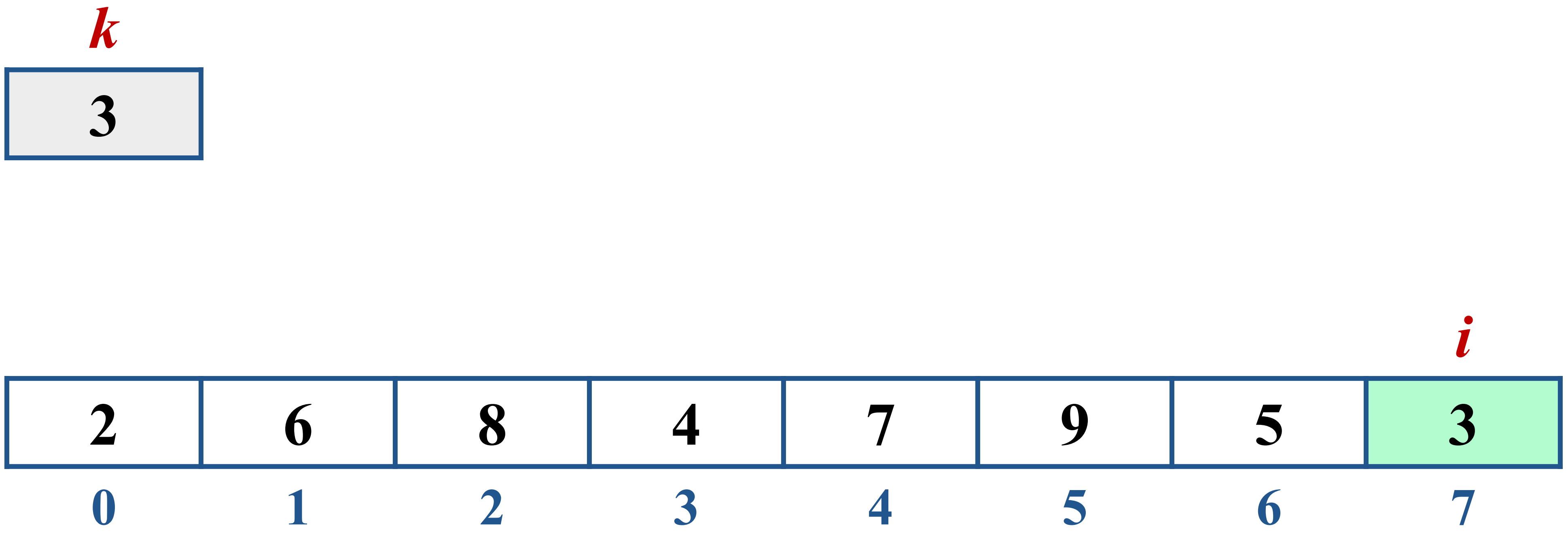


k

3

							i
2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Worst-Case



k

3

i

2	6	8	4	7	9	5	3
0	1	2	3	4	5	6	7

Sequential Search: Iterative Method

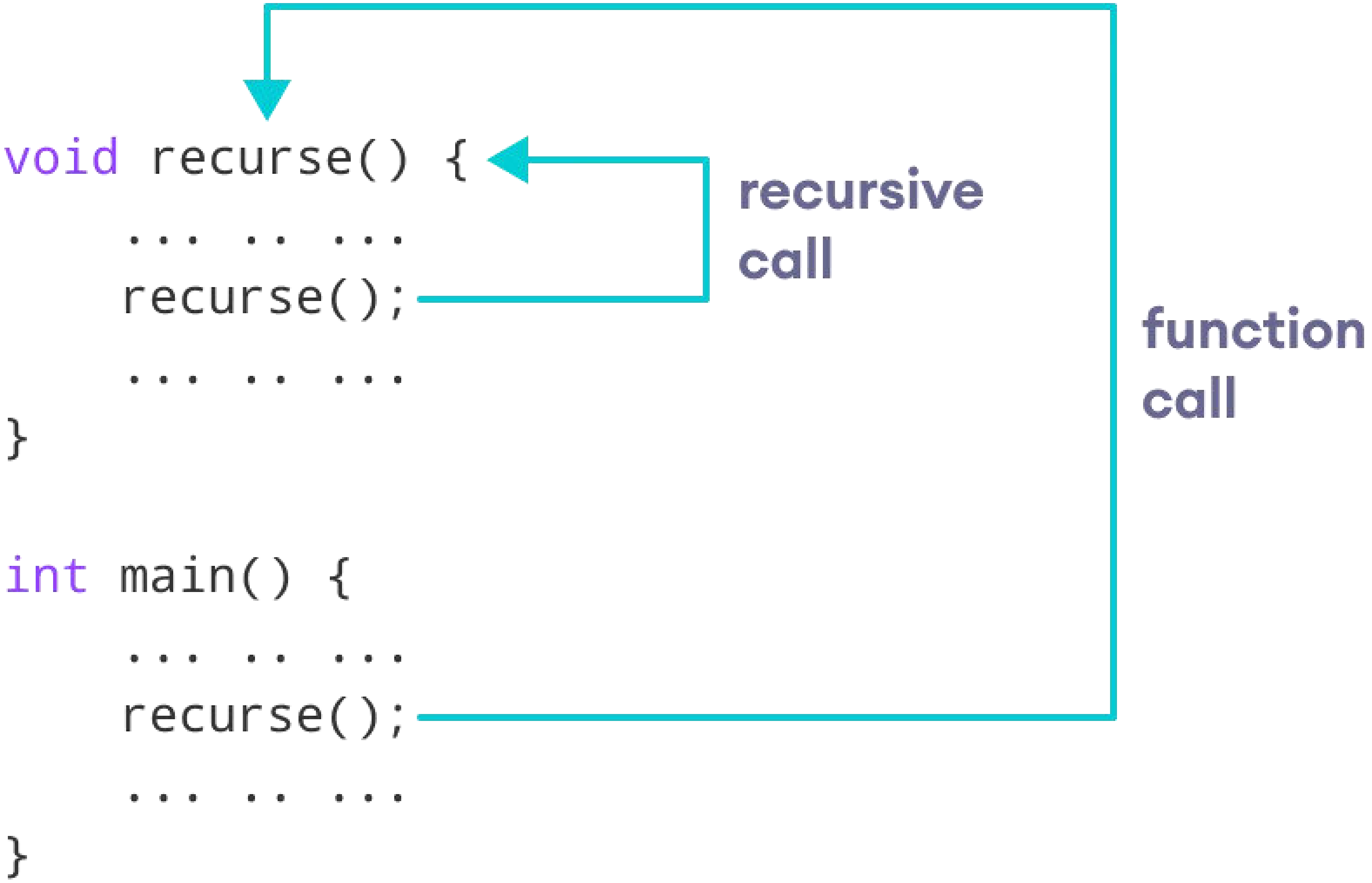
```
ALGORITHM SequentialSearch (A, $n$, K)
    for $i \leftarrow 0$ to $n - 1$ do
        // Element found, return its position
        if $A[i] = K$
            return $i$

    // Element not found in the array
    return $-1$
```

Recursion

- Recursion is a coding technique used in many algorithms.
- A recursive function is a function that calls itself.
- A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.

Recursion



```
void recurse() {
    ... .. ...
    recurse();
    ... .. ...
}

int main() {
    ... .. ...
    recurse();
    ... .. ...
}
```

recursive call

function call

Base Case and Recursive Case

- Because a recursive function calls itself, it’s easy to write a function incorrectly that ends up in an infinite loop.
- When you write a recursive function, you have to tell it when to stop recursing.
- That’s why every recursive function has two parts: the base case, and the recursive case.
- The recursive case is when the function calls itself.
- The base case is when the function doesn’t call itself again, so it doesn’t go into an infinite loop.

Advantages of Recursion

- It makes our code shorter and cleaner.
- Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

Disadvantages of Recursion

- It takes a lot of stack space compared to an iterative program.
- It uses more processor time.
- It can be more difficult to debug compared to an equivalent iterative program.

Sequential Search: Recursive Method 1

```
ALGORITHM SequentialSearch (A, $n$, K, $i$)
    // Base case (Element not found in the array)
    if $i \geq n$
        return $-1$
    // Base case (Element found, return its position)
    if $A[i] = K$
        return $i$
    // Recursive case
    return SequentialSearch (A, $n$, K, $i + 1$)
```

Sequential Search: Recursive Method 2

```
ALGORITHM SequentialSearch (A, $n$, K)
    // Base case (Element not found in the array)
    if $n < 0$
        return $-1$
    // Base case (Element found, return its position)
    if $A[n] = K$
        return $n$
    // Recursive case
    return SequentialSearch (A, $n$ - 1, K)
```

Worst-Case, Best-Case, and Average-Case Efficiencies

- Time efficiency is measured by counting the number of times the algorithm’s basic operation is executed.
- Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size.
- For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.

Sequential Search: Worst-Case

- Clearly, the running time of this algorithm can be quite different for the same array size n.
- In the worst case, when there are no matching elements or the matching element happens to be the last one on the array, the algorithm makes the largest number of key comparisons among all possible inputs of size

$$T_{worst}(n) = n$$

Sequential Search: Best-Case

- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size.
- The best-case inputs for sequential search are arrays of size n with their first element equal to a search key.

$$T_{best}(n) = 1$$

Sequential Search: Average-Case

- It is clear from our discussion that neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input.
- This is the information that the average-case efficiency seeks to provide.
- To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n.

Sequential Search: Average-Case

- The probability of a successful search is equal to p ($0 \leq p \leq 1$).
- In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1 - p)$.

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

- If $p = 1$ (successful search), the average number of key comparisons made by sequential search is $(n + 1)/2$.

Algorithm Design Techniques

- **Brute Force and Exhaustive Search**
- **Decrease-and-Conquer**
- **Divide-and-Conquer**
- Transform-and-Conquer
- Space and Time Trade-Offs
- Dynamic Programming
- Greedy Technique
- Iterative Improvement
- Backtracking
- Branch-and-Bound

Decrease-and-Conquer Algorithms

- Insertion Sort
- Iterative Binary Search
- Interpolation Search
- Euclid’s Algorithm
- Searching and Insertion in a Binary Search Tree

Binary Search

- Binary search is a remarkably efficient algorithm for searching in a sorted array.
- It is an example of a decrease-by-a-constant-factor algorithm.
- It works by comparing a search key K with the array's middle element A[*mid*].
- If they match, the algorithm stops.
- Otherwise, the same operation is repeated recursively.
- Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too.

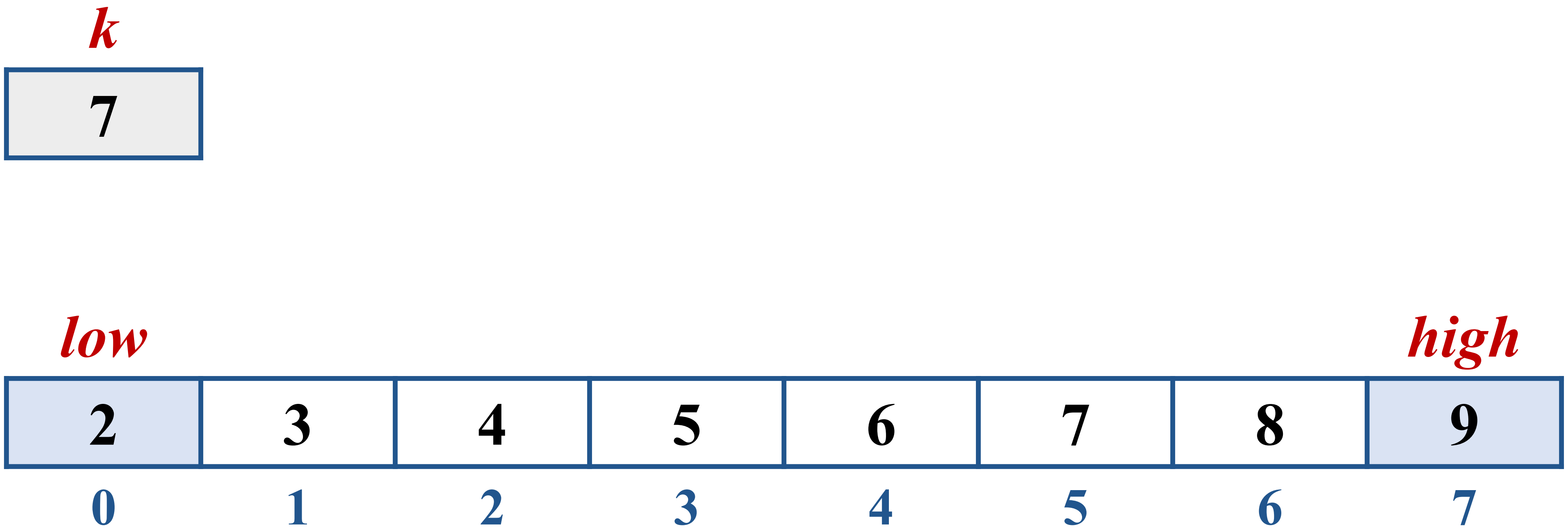
Binary Search: Example 1

k

7

2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Example 1



k

7

low							*high*
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

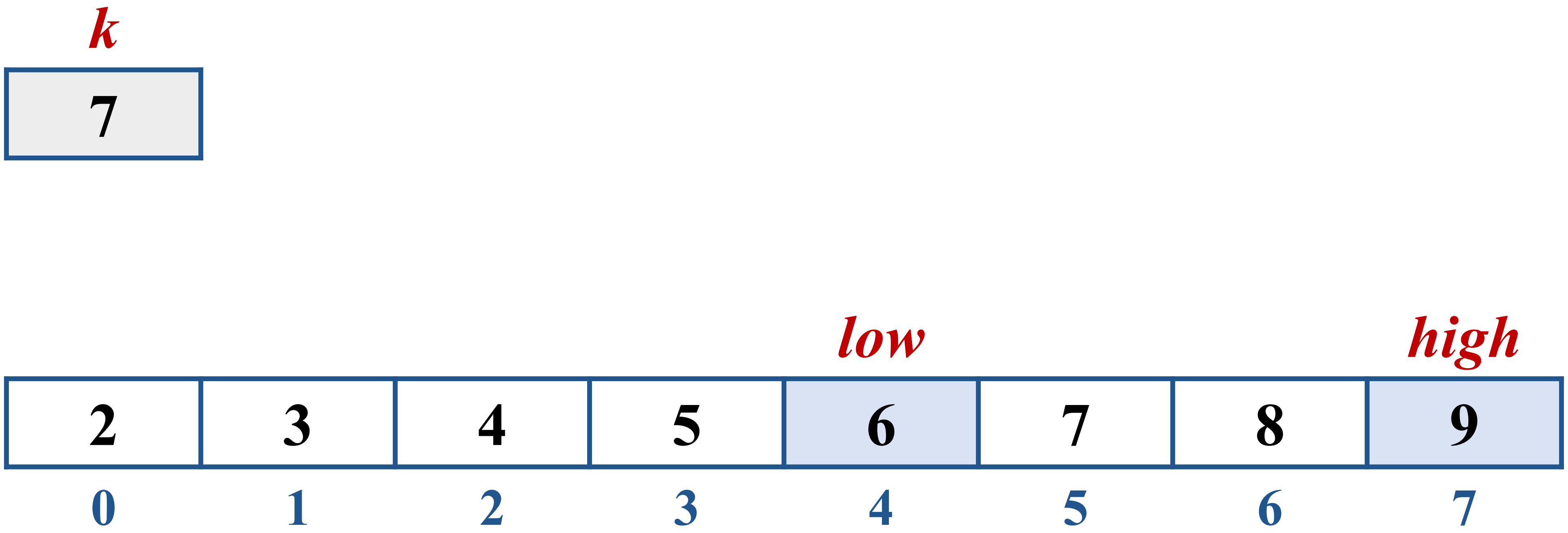
Binary Search: Example 1

k

7

low			*mid*				*high*
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7

Binary Search: Example 1

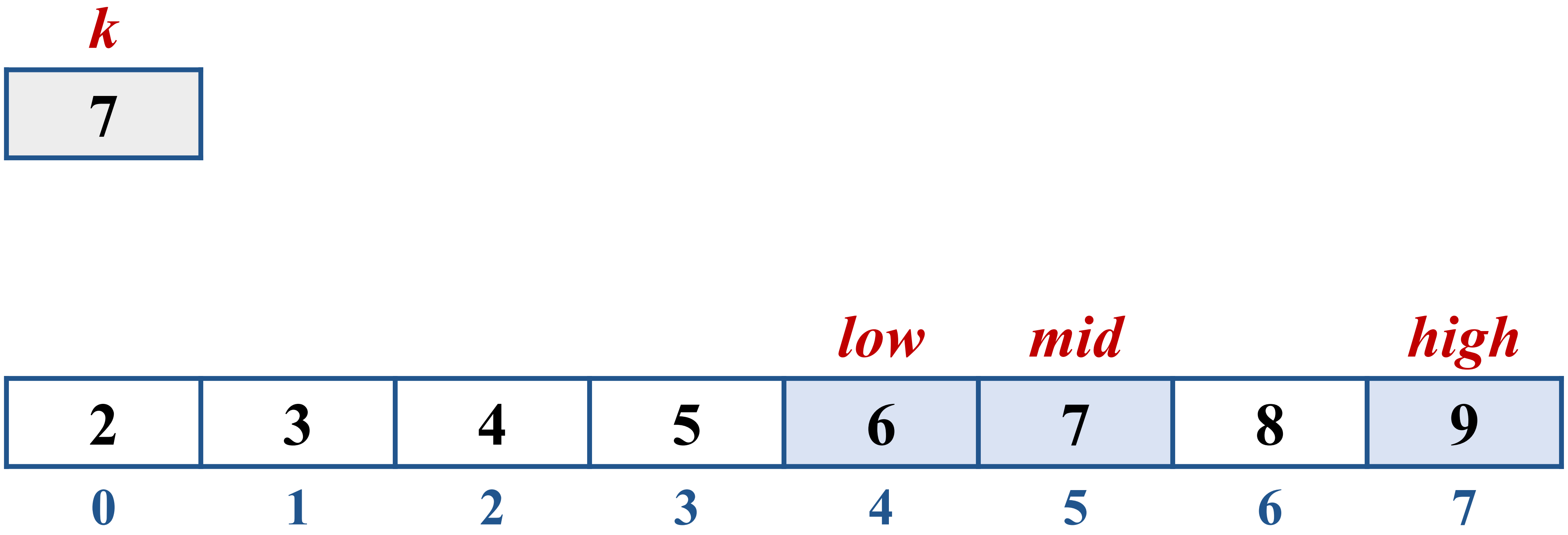


k

7

				low			***high***
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Binary Search: Example 1

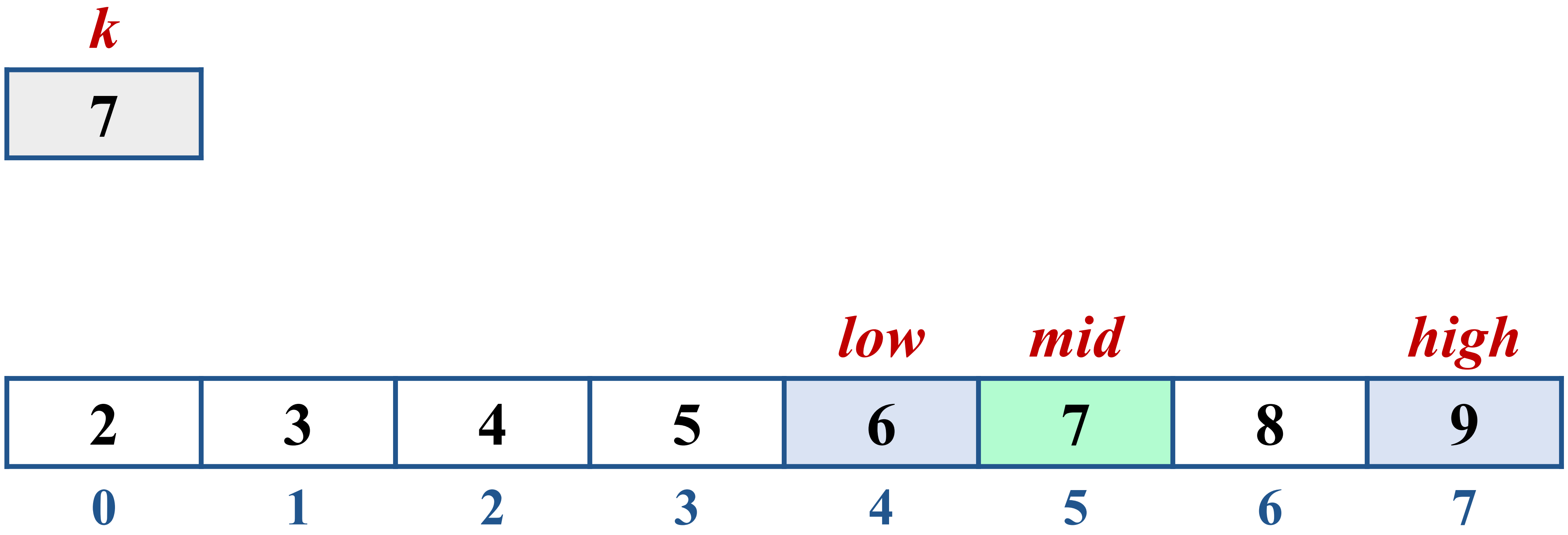


k

7

				low	*mid*		*high*
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Binary Search: Example 1



k

7

				low	***mid***		***high***
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

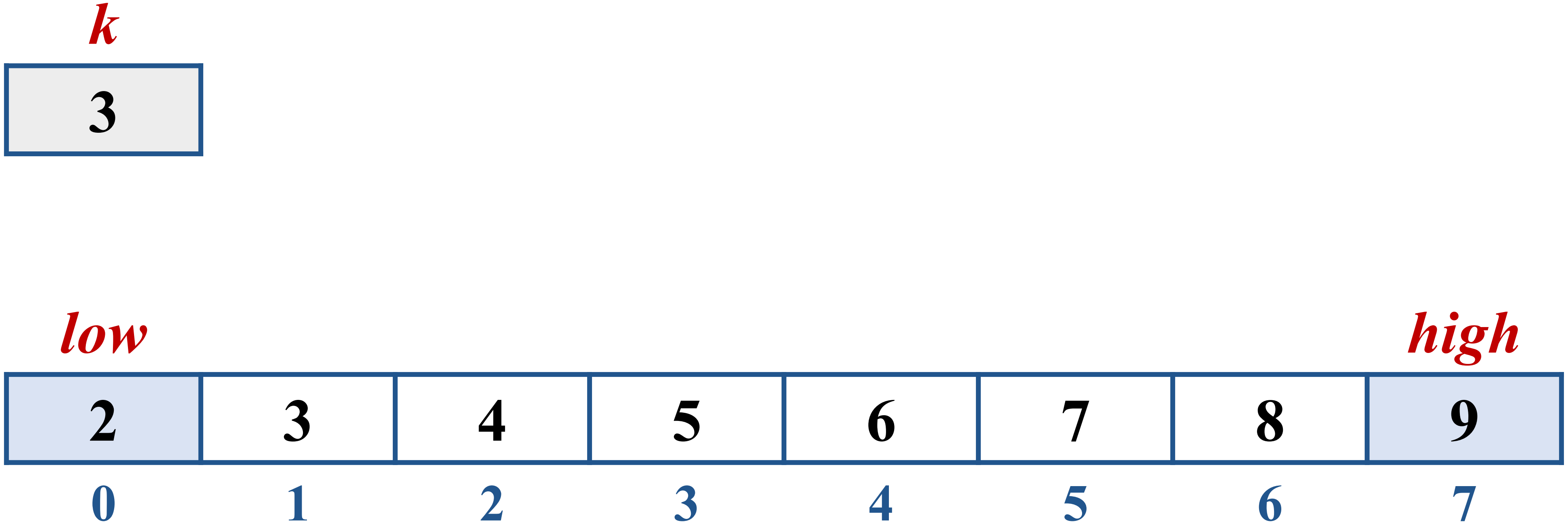
Binary Search: Example 2

k

3

2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Example 2

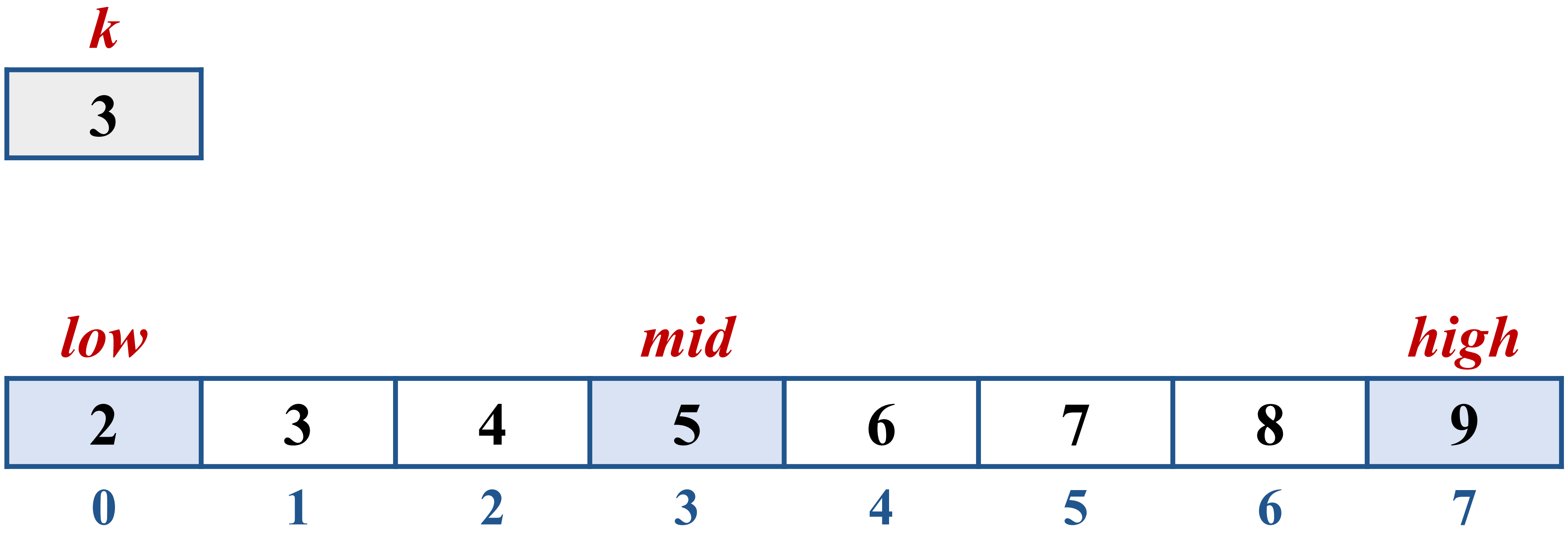


k

3

low							*high*
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Example 2

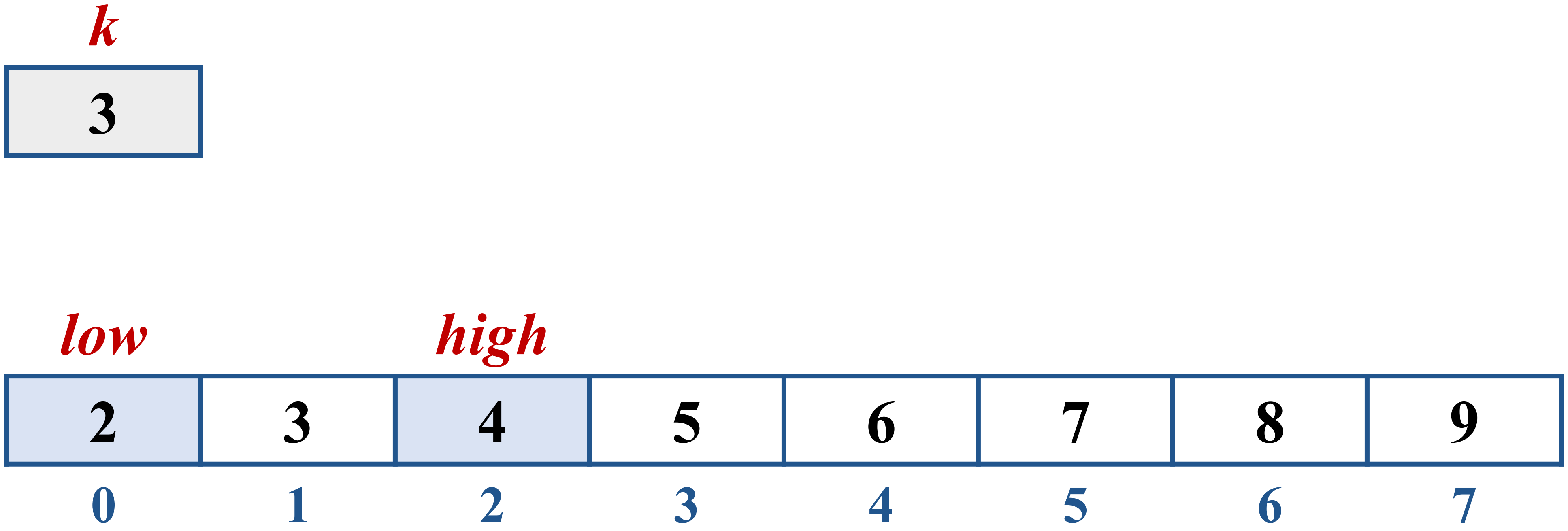


k

3

low			*mid*				*high*
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Example 2

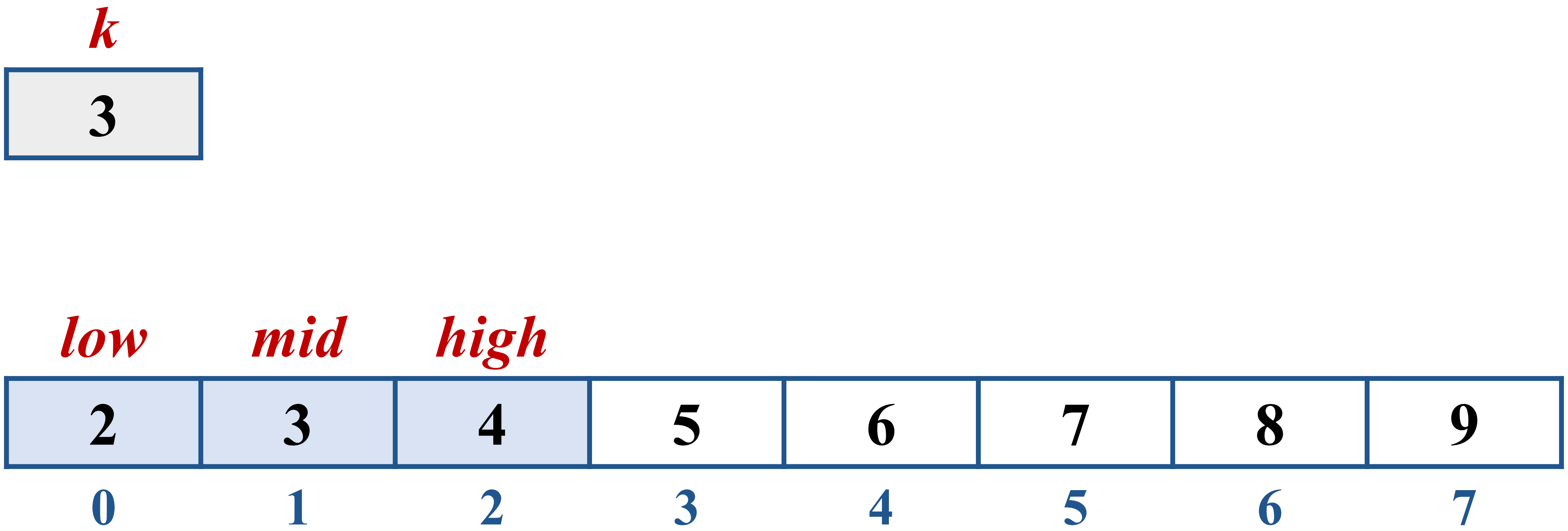


k

3

low		*high*					
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	2	3	4	5	6	7

Binary Search: Example 2

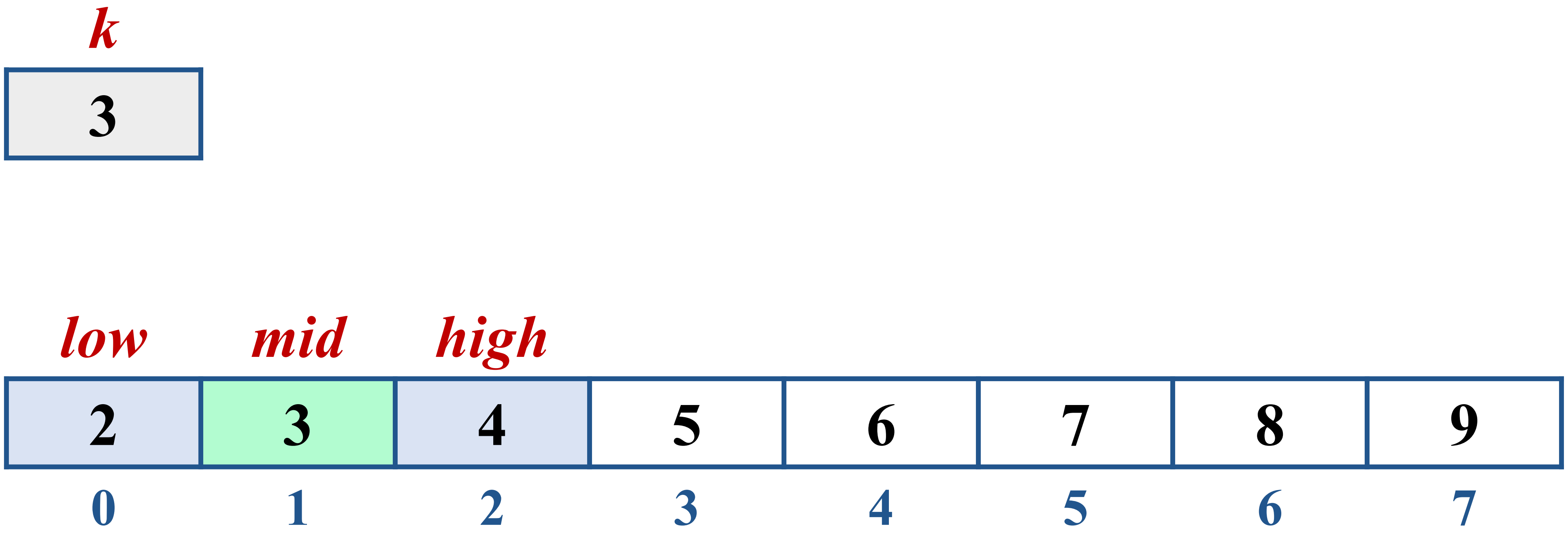


k

3

low	*mid*	*high*					
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Example 2



k

3

low	*mid*	*high*					
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

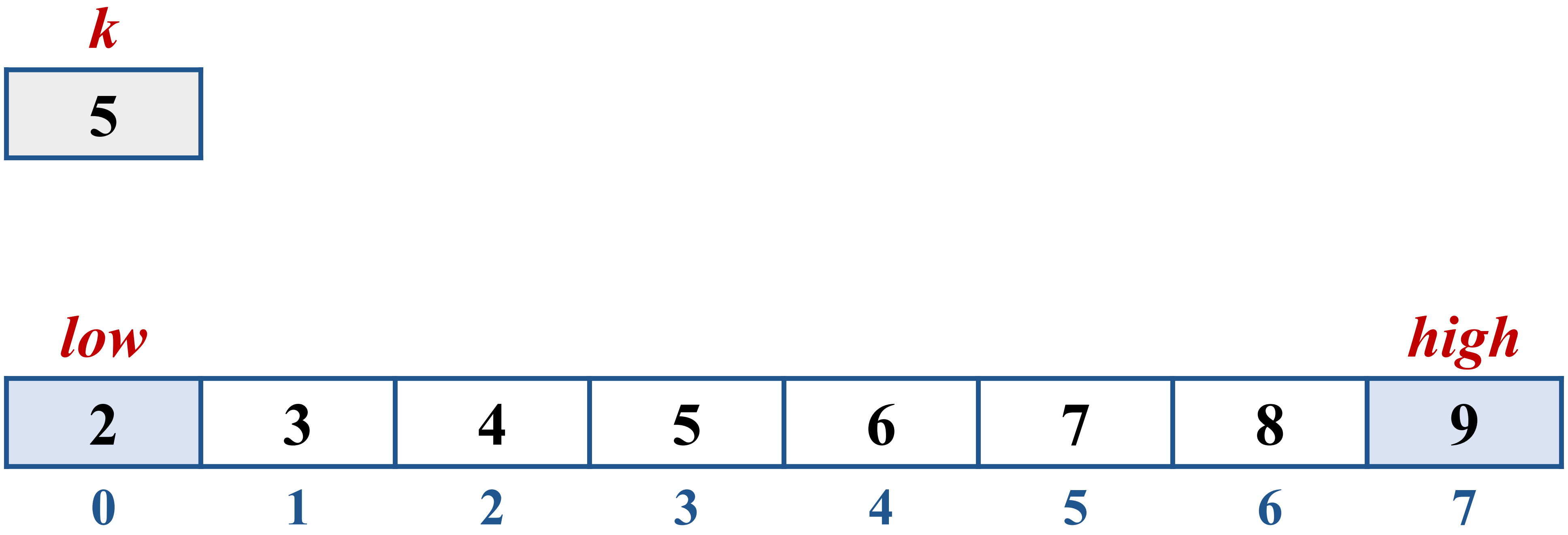
Binary Search: Best-Case

k

5

2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Best-Case

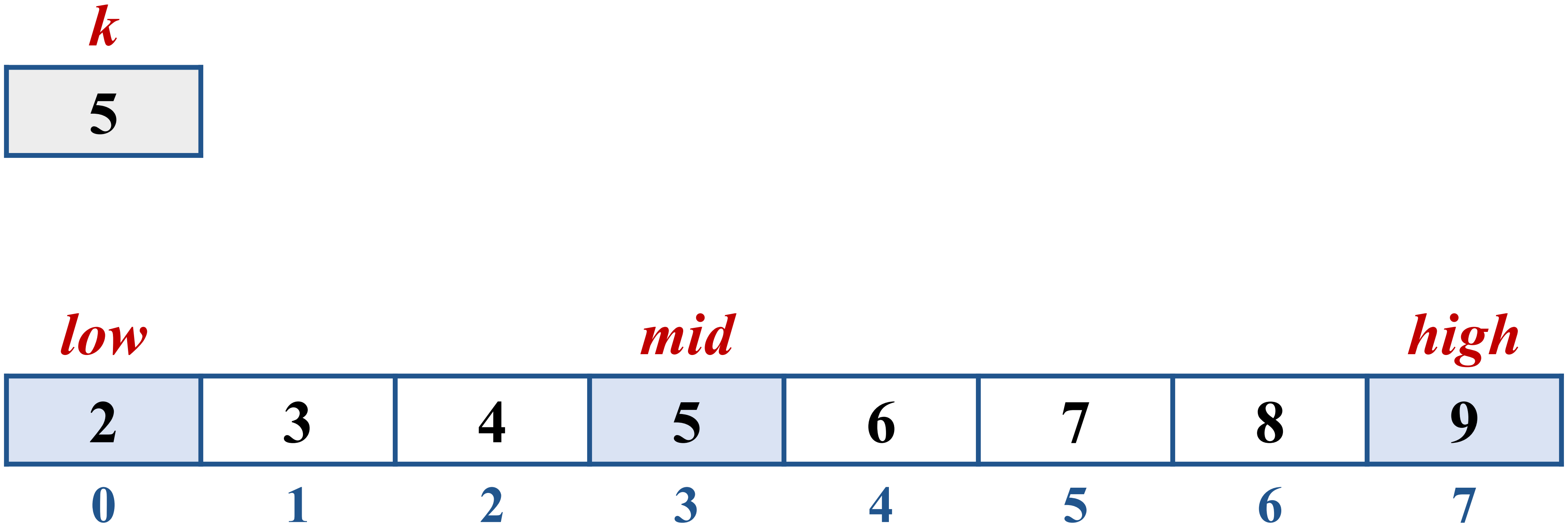


k

5

low							*high*
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Best-Case

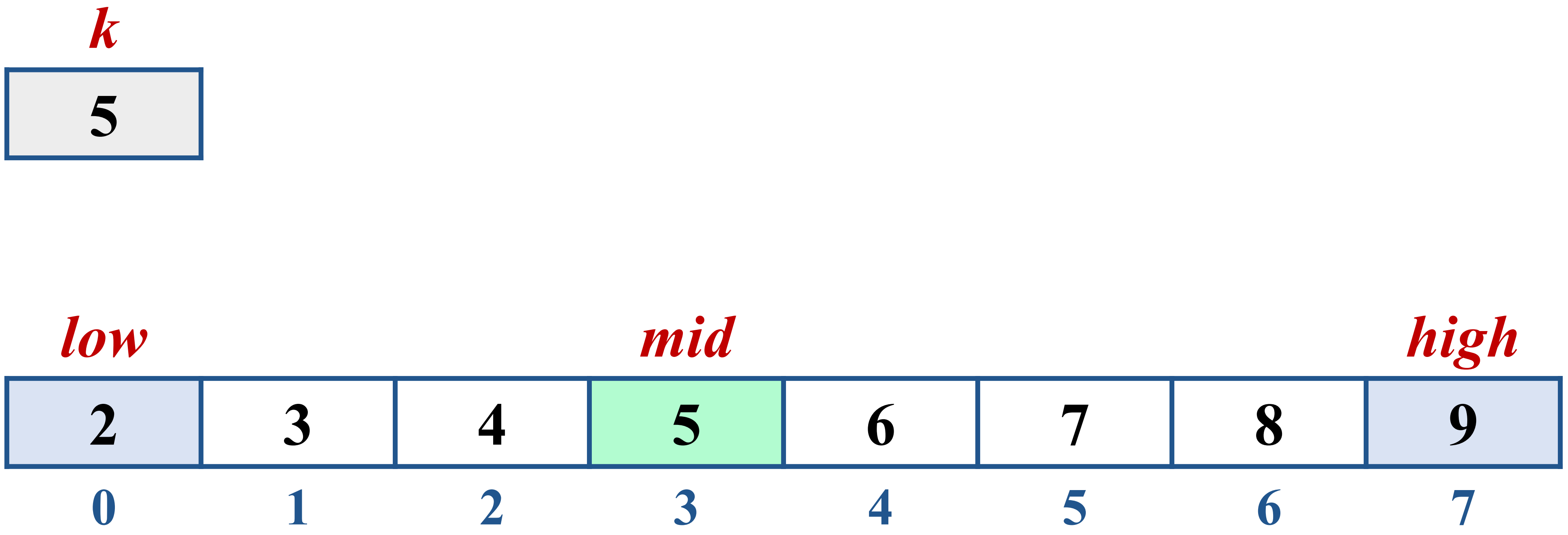


k

5

low			*mid*				*high*
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Best-Case



k

5

low			*mid*				*high*
2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7

Binary Search: Iterative Method

```
ALGORITHM BinarySearch(A[0 .. n − 1], K)
    low ← 0;
    high ← n − 1
    while high ≥ low do
        mid ← ⌊(low + high) / 2⌋
        if K = A[mid]
            return mid
        else if K < A[mid]
            high ← mid − 1
        else
            low ← mid + 1
    return −1
```

Algorithm Design Techniques

- **Brute Force and Exhaustive Search**
- **Decrease-and-Conquer**
- **Divide-and-Conquer**
- Transform-and-Conquer
- Space and Time Trade-Offs
- Dynamic Programming
- Greedy Technique
- Iterative Improvement
- Backtracking
- Branch-and-Bound

Divide-and-Conquer Algorithms

- Mergesort
- Quicksort
- Recursive Binary Search
- Strassen’s Matrix Multiplication Algorithm
- Cooley–Tukey Fast Fourier Transform (FFT) Algorithm
- Multiplication of Large Integers

Binary Search: Recursive Method

- In fact, some people consider such algorithms as binary search degenerate cases of divide-and-conquer, where just one of two subproblems of half the size needs to be solved.
- It is better not to do this and consider decrease-by-a-constant-factor and divide-and-conquer as different design paradigms.
- Binary Search algorithm can be implemented in two ways:

  1. Iterative Method
  2. Recursive Method

- The recursive method follows the divide-and-conquer approach.

Binary Search: Recursive Method

```
ALGORITHM BinarySearch(A, K, low, high)
    if $high \geq low$
        $mid \leftarrow \lfloor (low + high) / 2 \rfloor$
        if $K = A[mid]$
            return $mid$
        else if $K < A[mid]$
            return BinarySearch(A, K, low, mid - 1)
        else
            return BinarySearch(A, K, mid + 1, high)
    else
        return $-1$
```

Analysis of Binary Search

- The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array.
- For simplicity, we will assumes that after one comparison of K with A[*mid*], the algorithm can determine whether K is smaller, equal to, or larger than A[*mid*].
- The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches.

Analysis of Binary Search

- Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation

$$T(n) = T(n/2) + 1, \qquad T(1) = 1$$

- The worst-case time efficiency of binary search is in $\Theta(\log_2 n)$.
- The logarithmic function grows so slowly that its values remain small even for very large values of *n*.
- 20 comparisons to search any sorted array of size one million!

Analysis of Binary Search

$$T(n) = T(n/2) + 1, \qquad T(1) = 1$$

Let $n = 2^k \;\Leftrightarrow\; k = \log_2(n)$

$$T(2^k) = T(2^{k-1}) + 1$$

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

$$T(2^k) = \left[T(2^{k-2}) + 1\right] + 1 = T(2^{k-2}) + 2$$

$$T(2^{k-2}) = T(2^{k-3}) + 1$$

$$T(2^k) = \left[T(2^{k-3}) + 1\right] + 2 = T(2^{k-3}) + 3$$

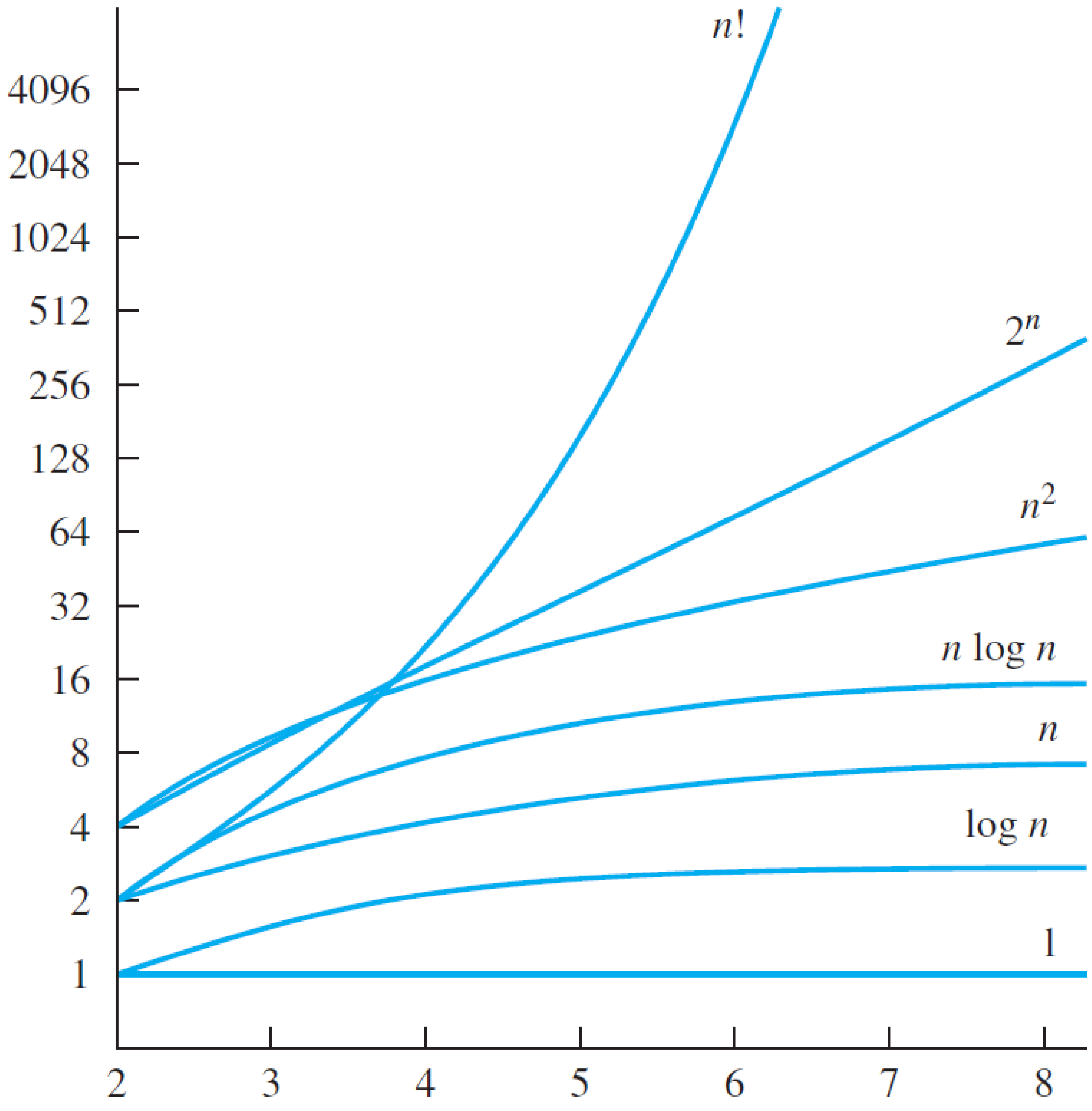
$$T(2^k) = T(2^{k-i}) + i$$

$$T(2^k) = T(2^{k-k}) + k$$

$$T(2^k) = T(1) + k = 1 + k$$

$$T(n) = 1 + \log_2(n) \qquad \in \Theta(\log_2(n))$$

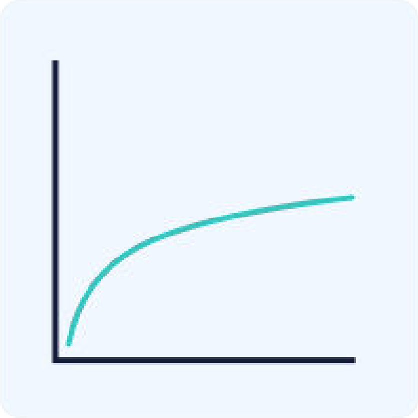
Orders of Growth



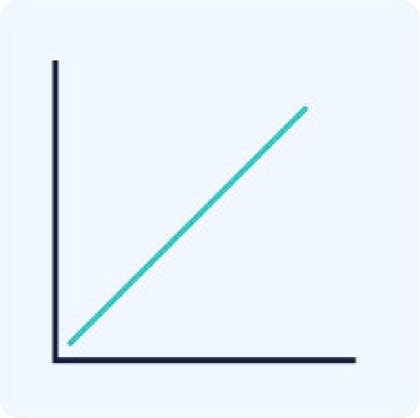
Orders of Growth

$\Theta(1)$

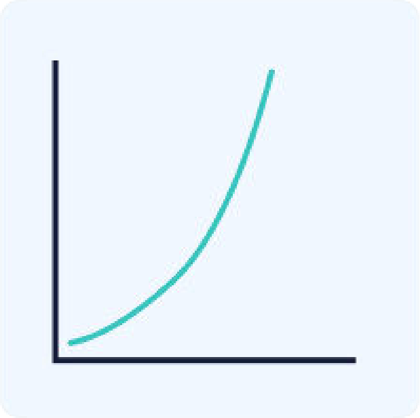
$\Theta(\log N)$



$\Theta(N)$



$\Theta(N \log N)$

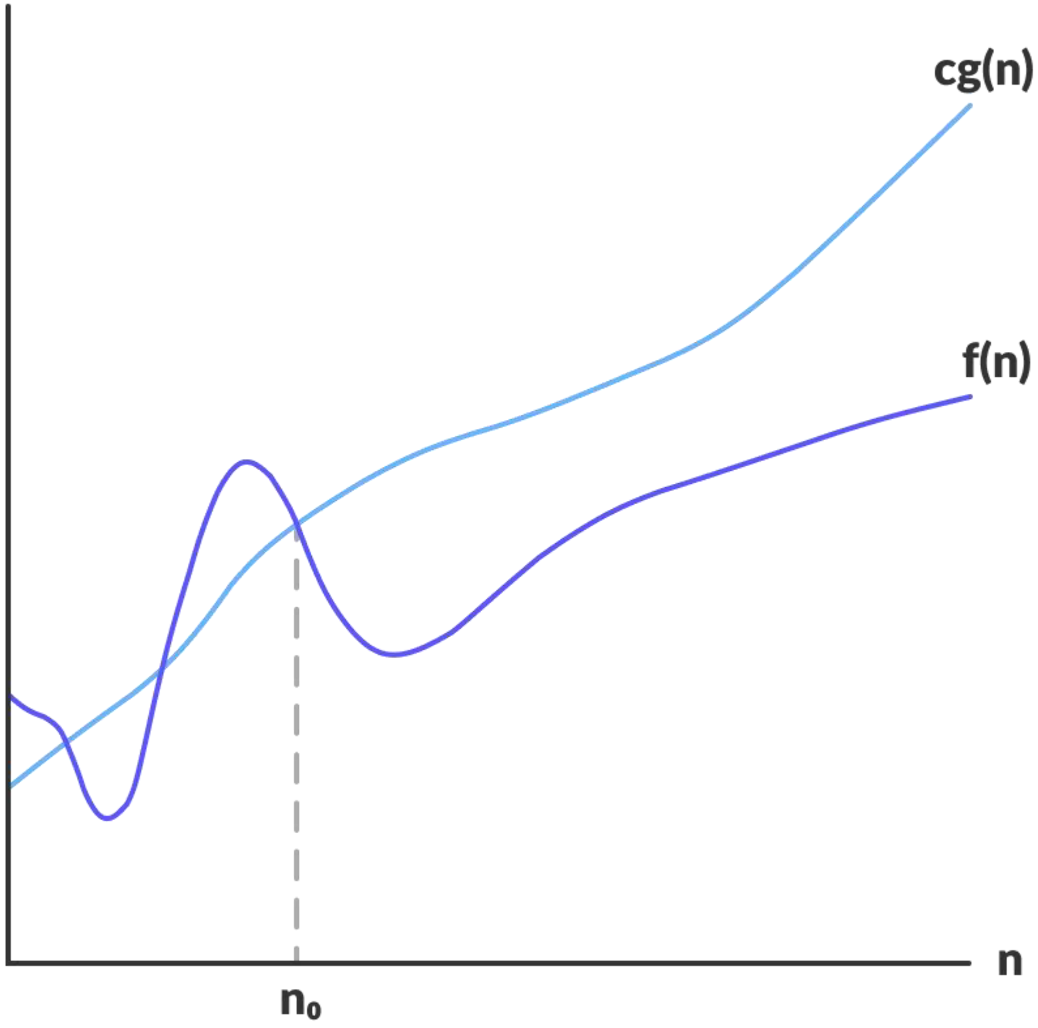


$\Theta(N^2)$

$\Theta(2^N)$

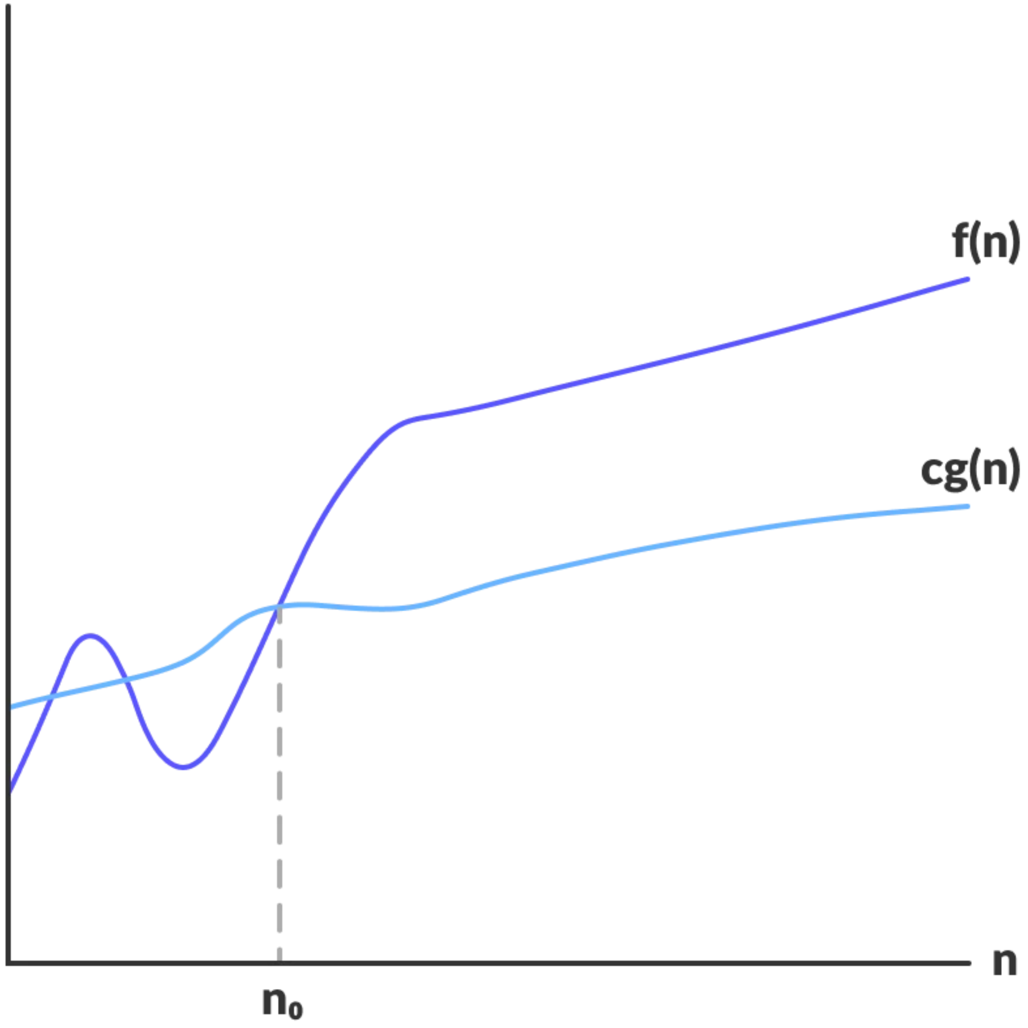
$\Theta(N!)$

Asymptotic Notations: O-notation



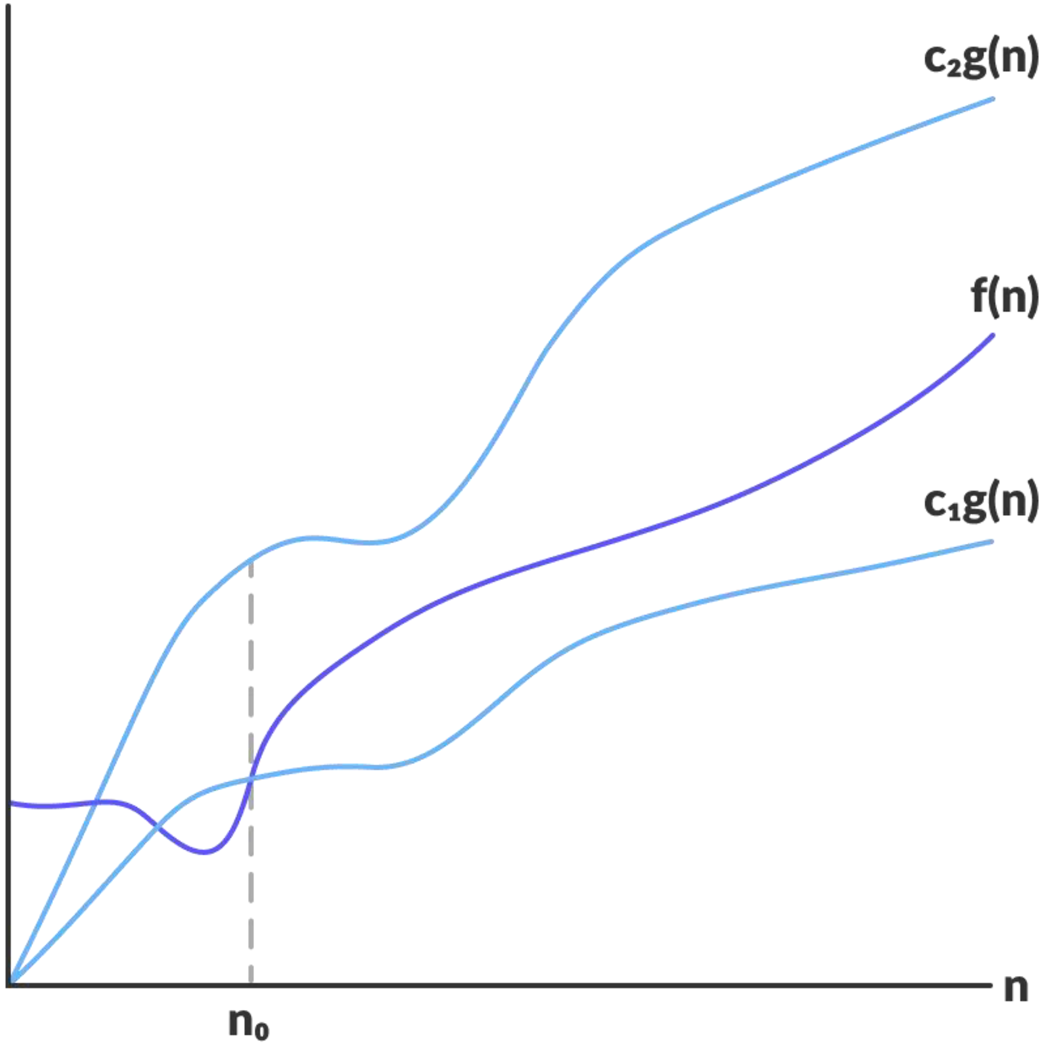
f(n) = O(g(n))

Asymptotic Notations: Ω-notation



$f(n) = \Omega(g(n))$

Asymptotic Notations: Θ-notation



$f(n) = \Theta(g(n))$

Asymptotic Notations

$T(n) \in O(g(n))$ is like $b \geq a$

$T(n) \in \Omega(g(n))$ is like $b \leq a$

$T(n) \in \Theta(g(n))$ is like $b = a$

$T(n) \in o(g(n))$ is like $b > a$

$T(n) \in \omega(g(n))$ is like $b < a$

Searching Algorithms

Algorithm	**Worst-Case Running Time**	**Average-Case Running Time**	**Best-Case Running Time**	**Requires Sorted Array?**
Linear search	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	No
Binary search	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$\Theta(1)$	Yes

Searching Algorithms

- **Linear Search**
- **Binary Search**
- Interpolation Search
- Jump Search
- Exponential Search
- Fibonacci Search

Sorting Algorithms

- Bubble Sort
- **Selection Sort**
- **Insertion Sort**
- **Merge Sort**
- Heap Sort
- Quicksort
- Counting Sort
- Radix Sort
- Bucket Sort
- Shell Sort