Foundations of Algorithm SCS1308

Dr. Dinuni Fernando
PhD

Senior Lecturer

Searching Algorithms

Table of Content

Searching Algorithm

- Searching Algorithm is an algorithm made up of a series of instructions that retrieves information stored within some data structure or calculated in the search space of a problem domain.
- Involves finding specific item(s) in vast collection of data.

Why we need a search algorithm ?

- For efficient data retrieval
  - Data size is too large to search sequentially or manually.
- For real world applications
  - Database systems – quick retrieval of records
  - File systems – locating files on storage device efficiently
  - Web search engines – retrieving relevant web pages from billions of documents.
  - Games and AI – decision-making and pathfinding eg: best move in chess or navigating a maze
- Handling large-scale problems
  - With large scale growth in applications, traditional methods of search is impractical.
- Optimizing resource usage
  - Computing resources like CPU time and memory are finite. Search algorithms minimize the computational effort and optimize resource utilization.

Types of search algorithms

- Linear Search
- Binary Search
- Jump Search
- Interpolation Search
- Exponential Search
- Ternary Search

Linear Search

- Linear Search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched

When to use linear search ?

- The dataset is unsorted or unordered.
- The dataset is small, making its simplicity advantageous.
- Searching in memory-constrained environments, as it does not require additional space.

Why ?

- Linear Search checks each element sequentially, making it the simplest search algorithm. It doesn’t depend on data structure properties like order or uniform distribution.

More about Linear Search

Practical Examples

- Searching for a name in an attendance list.
- Locating a specific value in an unsorted array.
- Finding a word in a dictionary that is stored in random order.

Limitations of Linear Search

- Inefficient for large datasets (O(n))
- Requires scanning the entire array for worst-case scenarios.

Linear Search

Algorithm:

Step1: Start from the leftmost element of array and one by one compare x with each element of array.

Step2: If x matches with an element, return the index.

Step3: If x doesn’t match with any of elements, return-1.

Linear Search

Assume the following array :
Search for 9

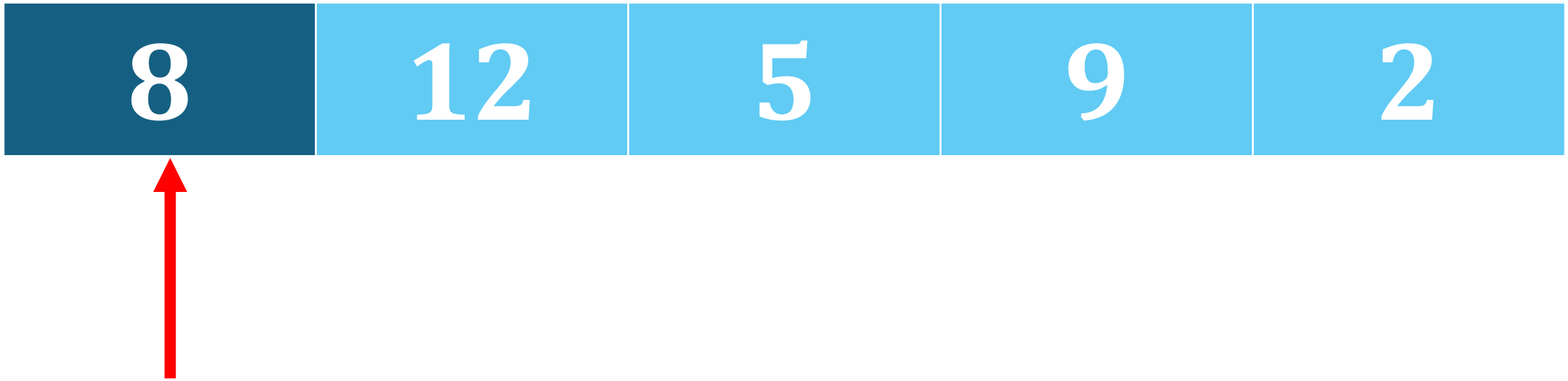
Linear Search

Compare
X = 9

Linear Search

Compare

X = 9



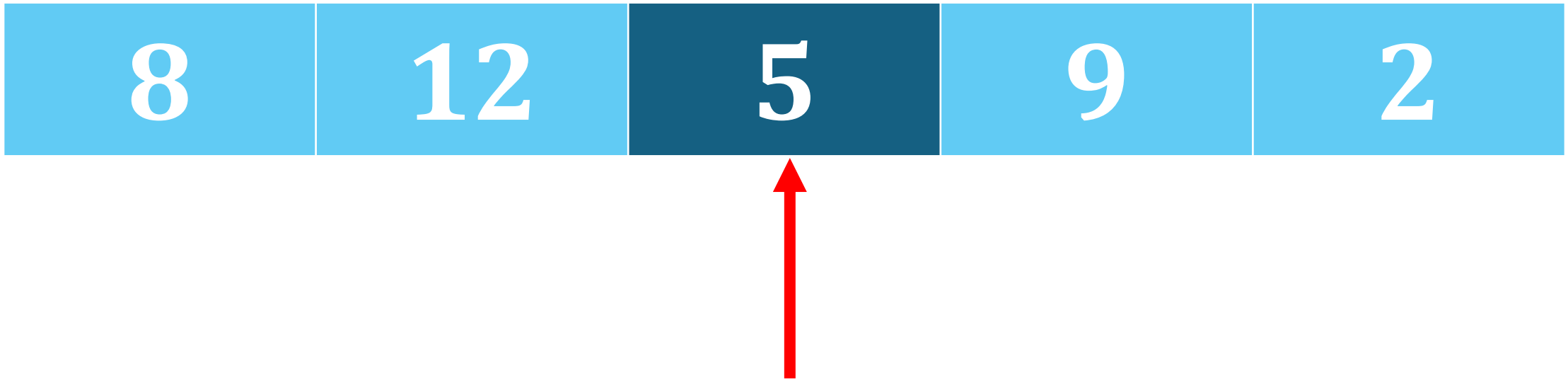
Linear Search

Compare

X = 9

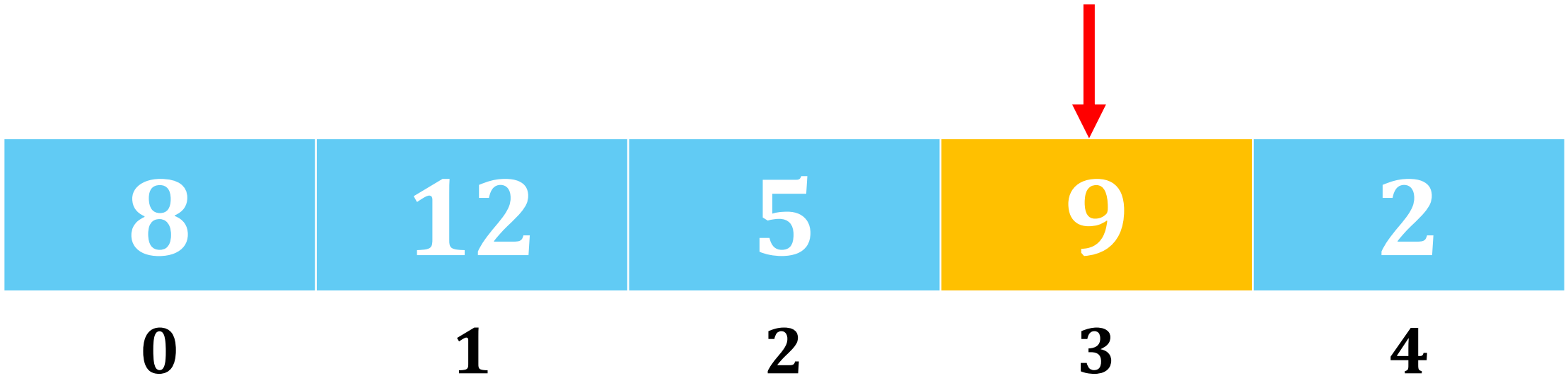
Linear Search

Compare
X = 9



Linear Search

Found at index 3



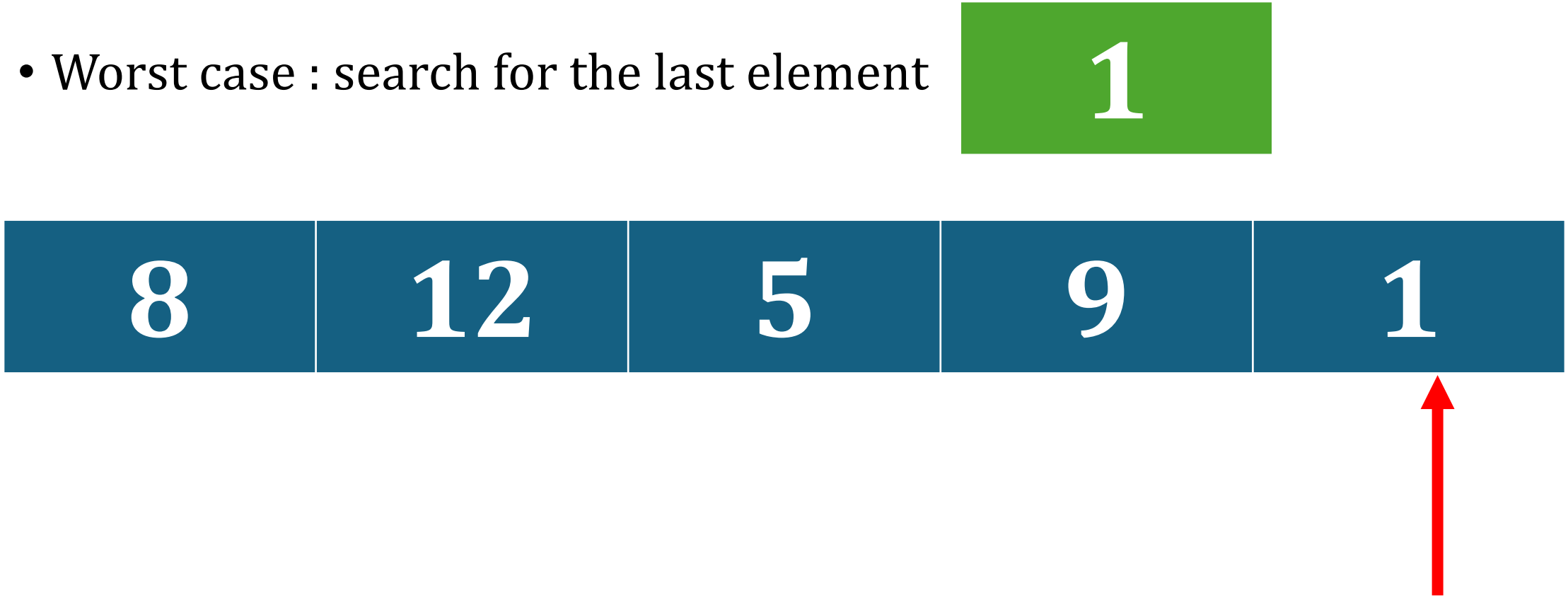
Linear search : Python code

```python
def LinearSearch(arr, x):  # Ensure parameter names match
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

# Array to search in
arr = [15, 8, 12, 45, 30]

# Target value to find
x = 45

# Call LinearSearch function
result = LinearSearch(arr, x)

# Print the result
if result != -1:
    print(f"Element {x} found at index {result}.")
else:
    print(f"Element {x} not found in the array.")
```

```
Element 45 found at index 3.
```

Linear Search

- Time Complexity O(n)
- Worst case : search for the last element



Binary Search

- Most popular search algorithm.
- It is efficient and also one of the most commonly used techniques that is used to solve problems.
- Binary search use sorted array by repeatedly dividing the search interval in half.
- Complexity $O(\log_2 n)$
  - Search space is divided in half with each comparison.
- Limitations
  - The dataset must be sorted beforehand.
  - Not suitable for linked lists due to slow middle access.
  - Additional overhead for maintaining the sorted order.

Use cases of Binary Search

Binary Search is used when:

- The dataset is sorted.
  - Finding a word in a dictionary: The words are alphabetically sorted.
- Searching large datasets where efficiency is crucial.
  - Eg: Searching in versioned software libraries: Locating a specific version among sorted entries.
- Data retrieval is required in constant or logarithmic time.
  - Eg: Database indexing: Retrieving records using sorted primary keys.

Binary Search

Algorithm:

- Step1: Compare x with the middle element.
- Step2: If x matches with middle element, we return the mid index.
- Step3: Else If x is greater than the mid element, search on right half.
- Step4: Else If x is smaller than the mid element. search on left half.

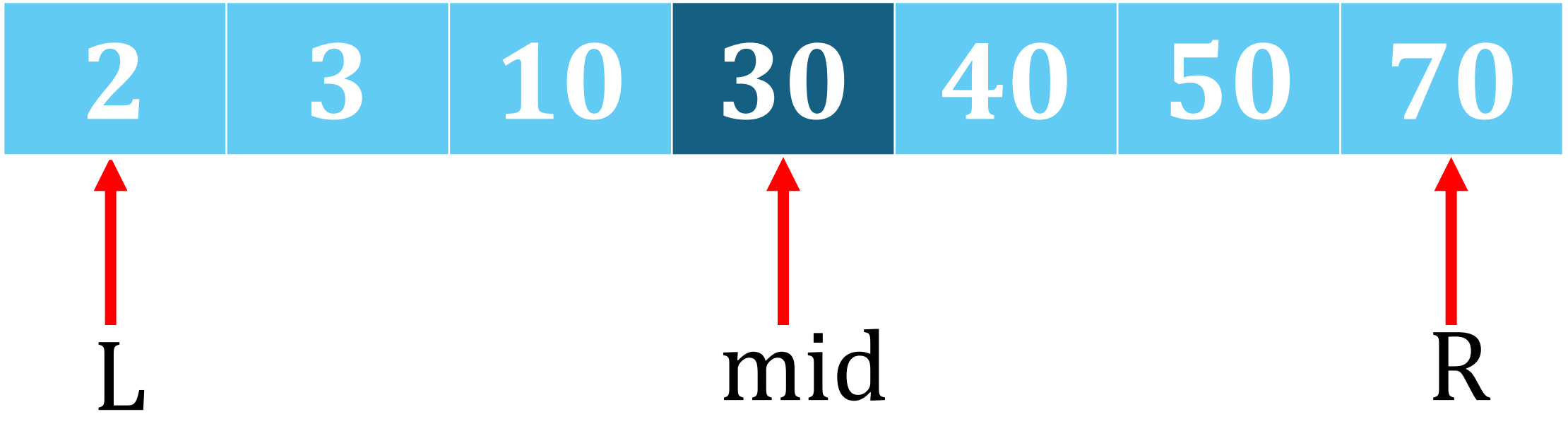
Binary Search

Assume the following array :
Search for 40

2	3	10	30	40	50	70

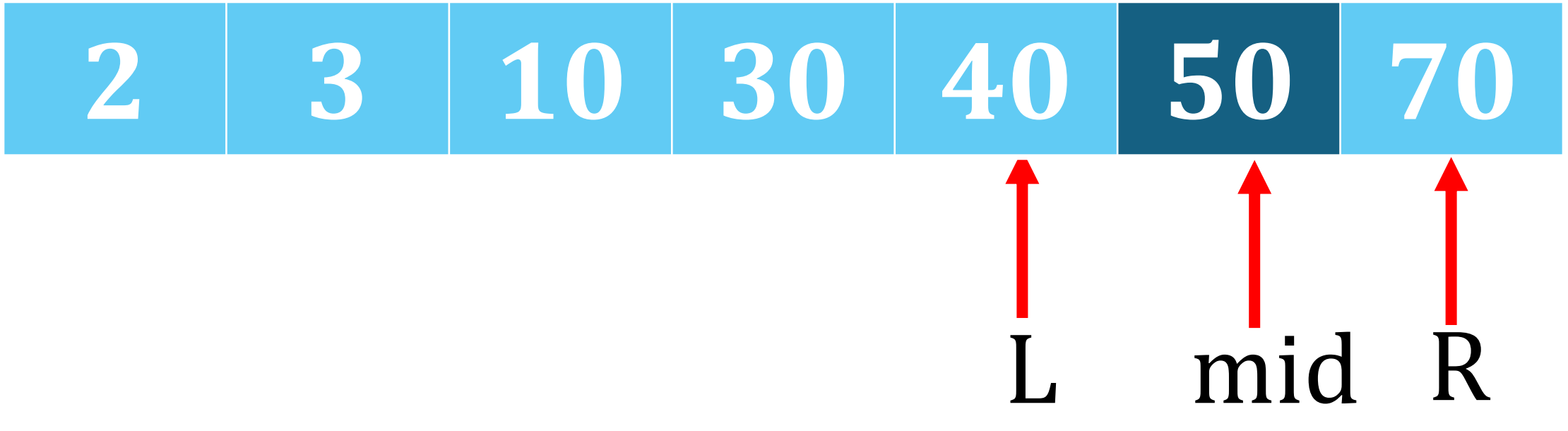
Binary Search

Compare
X = **40**



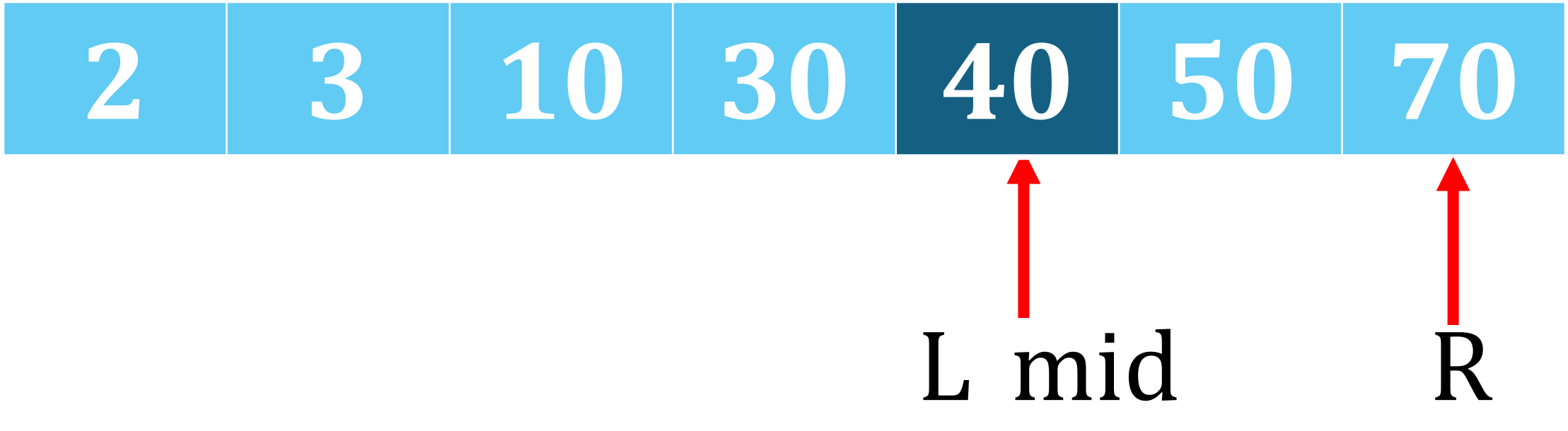
Binary Search

Compare
X = **40**



Binary Search

Compare
X = **40**



Binary Search

X = 40, found at index = 4

Binary Search

```python
def binary_search_iterative(arr, target):

    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2  # Find the middle index

        # Check if the target is at the middle
        if arr[mid] == target:
            return mid

        # If target is smaller, ignore the right half
        elif arr[mid] > target:
            right = mid - 1

        # If target is larger, ignore the left half
        else:
            left = mid + 1

    return -1  # Return -1 if the target is not found
```

```python
# Example usage
if __name__ == "__main__":
    # Sorted array with 7 elements
    arr = [10, 20, 30, 40, 50, 60, 70]
    target = 40

    result = binary_search_iterative(arr, target)

    if result != -1:
        print(f"Element {target} found at index {result}.")
    else:
        print(f"Element {target} not found in the array.")
```

Jump Search

- Is a searching algorithm for sorted arrays.
- The basic idea is to check fewer elements(that linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.
- Time complexity O(sqrt(n))
  - Jump Search skips ahead by fixed intervals (n) instead of checking each element. This reduces the number of comparisons in sorted data.

+ Faster than linear search – due to jumping, it skips over large sections of the array reducing unnecessary comparisons.

+ Efficient for sorted data – exploits sorted nature to minimize search efforts.

+ Better than binary search for certain data structures (eg: linked lists)

Use cases of Jump Search

Jump Search is used when:

- The dataset is sorted.
  - Library catalog systems: Finding books in sorted shelves.
- Optimizing search speed for a large dataset.
- Systems where the overhead of random access is acceptable.
- Limitations
  - Requires the dataset to be sorted.
  - Performance depends on the jump size (sqrt(n))
  - Less efficient for small datasets compared to Binary Search.

Jump Search

Algorithm

- Step1: Calculate Jump size
- Step2: Jump from index i to index i+jump
- Step3: If x = = arr[i+jump] return x

  Else jump back a step
- Step4: Perform linear search

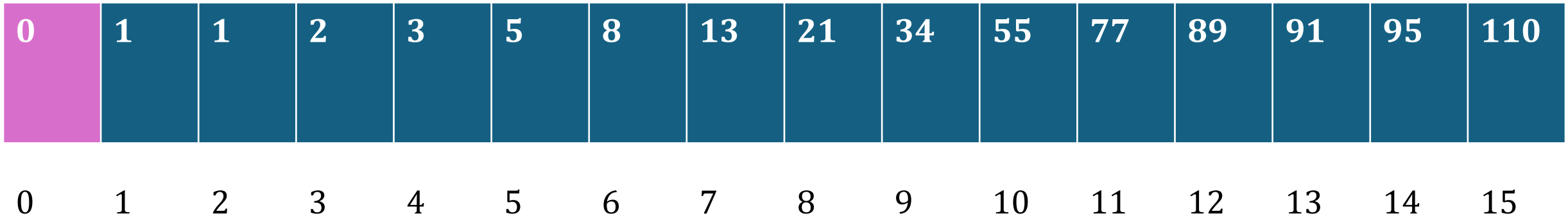
Jump Search

- Assume the following sorted array
- Search for 77

0	1	1	2	3	5	8	13	21	34	55	77	89	91	95	110

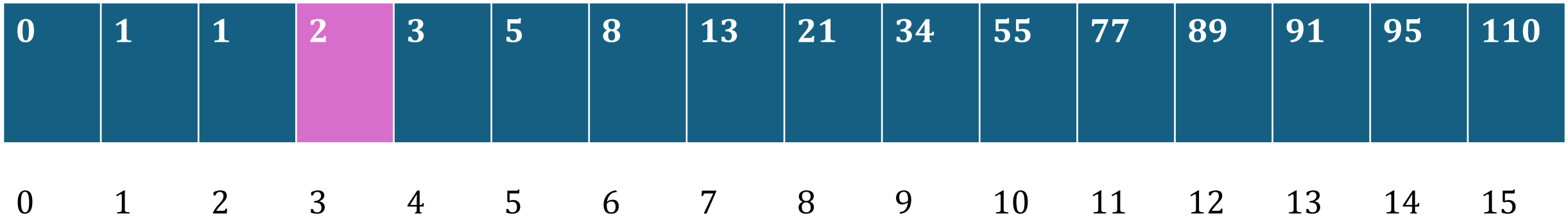
Jump Search : search element 77

- Jump size = 4
- Search from index 0
- Compare index value with search number 0 > 77



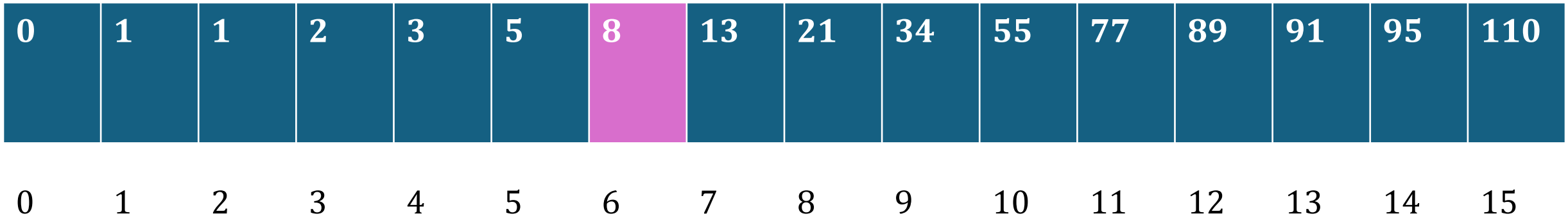
Jump Search

- Jump size = 4
- Search from index 0 to index 3
- Compare index value with search number 2 > 77



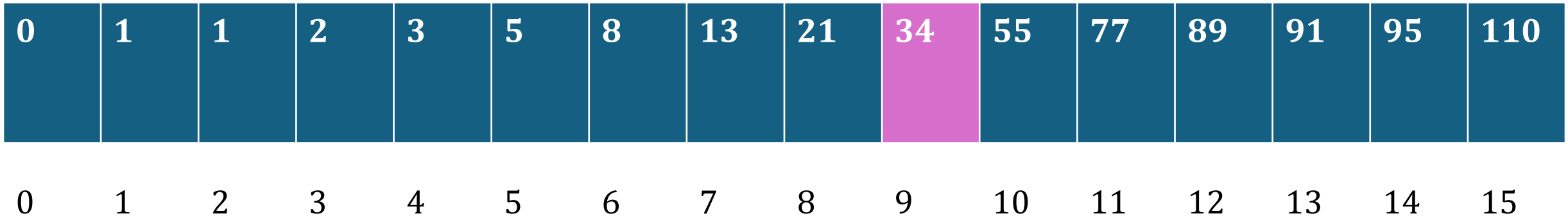
Jump Search

- Jump size = 4
- Search from index 3 to index 6
- Compare index value with search number 8 <77



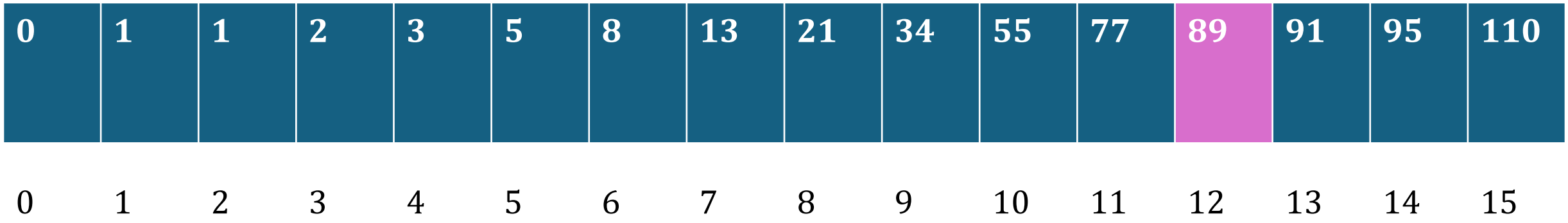
Jump Search

- Jump size = 4
- Search from index 6 to index 9
- Compare index value with search number 34 < 77



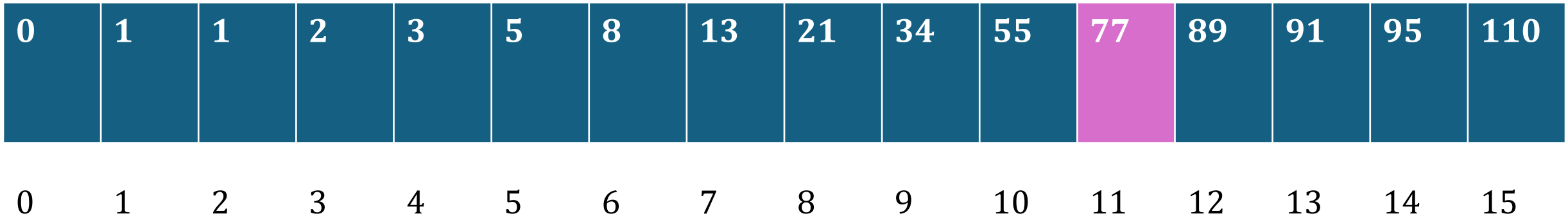
Jump Search

- Jump size = 4
- Search from index 9 to index 12
- Compare index value with search number 89 > 77



Jump Search

- Jump back to step.
- Perform linear search
- Compare found at index 11



```python
import math

def jump_search(arr, target):

    n = len(arr)
    step = int(math.sqrt(n))  # Block size to jum
    prev = 0

    # Jump through the array blocks
    while arr[min(step, n) - 1] < target:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1  # Target not found

    # Linear search within the block
    for i in range(prev, min(step, n)):
        if arr[i] == target:
            return i

    return -1  # Target not found
```

```python
if __name__ == "__main__":
    # Sorted array with 9 elements
    arr = [3, 6, 9, 12, 15, 18, 21, 24, 27]
    target = 15

    result = jump_search(arr, target)

    if result != -1:
        print(f"Element {target} found at index {result}.")
    else:
        print(f"Element {target} not found in the array.")
```

Interpolation Search

- Is an efficient search algorithm that works on sorted arrays.
  - Complexity O(loglogn) for uniform data
- It improves over binary search by estimating the probable position of the target based on the distribution of values in the array.
  - Depending on the searching key, search may go to different locations.
- It is particularly effective when the elements of the array are uniformly distributed.
- Limitations
  - Requires the dataset to be sorted.
  - Performs poorly if the data is not uniformly distributed.
  - Overhead of interpolation calculations may not be worth it for small datasets.

Use cases of interpolation search

Interpolation Search is used when:

- The dataset is sorted and uniformly distributed.
- Searching large datasets where data has predictable intervals.
- High-speed retrieval is required, and the dataset follows a numeric distribution.
- Practical examples
  - Database indexing: Searching numeric keys like employee IDs or phone numbers.
  - Searching in sorted financial data: Stock prices or revenue trends with evenly distributed intervals.
  - IoT sensor data: Locating specific readings in time-series data with consistent intervals.

Interpolation Search

- Step1: In a loop, calculate the value of “pos” using the position formula.
- Step2: If it is a match, return the index of the item, and exit.
- Step3: If the item is less than arr[pos], calculate the position of the left sub-array. Otherwise calculate the same in the right sub-array.
- Step4: Repeat until a match is found or the sub-array reduces to zero.

Interpolation Search

- Unlike binary search- also chooses mid element
- Interpolation search calculates the position of the target element based on search interval range , value of the target element.

$$\text{pos} = \text{low} + \left(\frac{\text{target} - \text{arr}[\text{low}]}{\text{arr}[\text{high}] - \text{arr}[\text{low}]} \times (\text{high} - \text{low}) \right)$$

- Target = element to be search
- arr[] – array
- Low - start arr[] index
- high - end arr[] index

Interpolation Search

10	20	30	40	50	60	70	80	90

Target : 70

$$\text{pos} = \text{low} + \left(\frac{\text{target} - \text{arr[low]}}{\text{arr[high]} - \text{arr[low]}} \times (\text{high} - \text{low}) \right)$$

Process:

1. Start with low = 0 , high = 8
2. pos $= 0 + \frac{70-10}{90-10} x(8-0) = 6$
3. arr[6] =70
   Target found at index 6.

Comparison of Searching Algorithms

Aspect	Linear Search	Binary Search	Jump Search	Interpolation Search
Dataset	Unsorted/Small	Sorted	Sorted	Sorted and Uniform
Time Complexity	$O(n)$	$O(\log n)$	$O(\sqrt{n})$	$O(\log \log n)$ (Uniform)
Best Use Case	Small, unsorted datasets	Large, sorted datasets	Large, sorted datasets	Uniformly distributed datasets
Practical Example	Attendance lists	Database indexing	Library catalogs	Financial or IoT data
Key Limitation	Inefficient for large n	Needs sorted data	Depends on jump size	Requires uniform data

Thank you!