Data Structures and Algorithms

Unit 1

Program

- A computer program is a series of instructions to carry out a particular task written in a language that a computer can understand.

- The process of preparing and loading the instructions into the computer for execution is referred as programming

Data Structure and Algorithm

- Programs consists of two things: Algorithms and data structures
- An algorithm is a step by step procedure for solving a problem
- A data structure defines a way of organizing data items, by considering the relationship between those elements

Properties of Algorithm

- An algorithm possesses the following properties:
  - It must be correct.
  - There should be no ambiguity at any step
  - It must be composed of a finite number of steps.
  - It must terminate.
  - It takes zero or more inputs
  - It results in one or more outputs

Efficiency of an algorithm

- An algorithm's Efficiency/Performance is measured by calculating the time taken and space required for running the algorithm

Time Complexity :

- Time Complexity of an algorithm is the amount of time(or the number of steps) needed by a program to complete its task. It depends on Compilation Time and Run Time.

  - Compilation time is the time taken to compile an algorithm.
  - Run Time is the time to execute the compiled program

- It should be applicable to inputs of all sizes

Time Complexity of an Algorithm

- Time complexity of a given algorithm can be defined as a total number of statements that are executed for computing the value of f(n), n represents input size and f represents function of total executable statements in the algorithm

- Time complexity of an algorithm is generally classified as three types.

  (i) Worst case

  (ii) Average Case

  (iii) Best Case

- **Worst Case:** It is the longest time that an algorithm takes on all inputs of size n for a given problem to produce the result.

- **Average Case:** It is the average time that the algorithm takes on all inputs of size n for a given problem to produce the desired result.

- **Best Case:** It is the shortest time that the algorithm takes on all inputs of size n for a given problem to produce a desired result.

Space Complexity of an Algorithm

- The memory taken by the instructions is not in the control of the programmer as its totally dependent on the compiler to assign this memory.

- But the memory space taken by the variables is in the control of a programmer. More the number of variables used, more will be the space taken by them.

Asymptotic Analysis

In asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size.

Asymptotic analysis refers to computing the running time of Algorithm in mathematical units of computation.

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm:

- O Notation (Big O)
- Ω Notation (Omega)
- θ Notation (Theta)

The Big Oh notation (O)

This notation is known as the upper bound of the algorithm. It tells us that a certain function will never exceed a specified time for any value of input n.

The function f(n) = O(g(n)) iff ∃ positive constants, c and k such that

$$\mathbf{f(n) \leq c * g(n) \quad \forall\ n \geq k}$$

Omega Notation Ω

Omega notation is used to define the lower bound of any algorithm, Which means the algorithm will take at least this much time to complete it's execution. It can definitely take more time than this too.

The function f(n) = Ω (g(n)) iff ∃ positive constants, c and k such that

$$\mathbf{f(n) \geq c * g(n) \quad \forall\ n \geq k}$$

Theta Notation Ω

When we say tight bounds, we mean that the time complexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

The function f(n) = θ(g(n)) iff ∃ positive constants, c1, c2 and k such that

$$\mathbf{c2 * g(n) \leq f(n) \leq c1 * g(n) \quad \forall\ n \geq k}$$

Bubble Sort

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are not in order.

Algorithm:

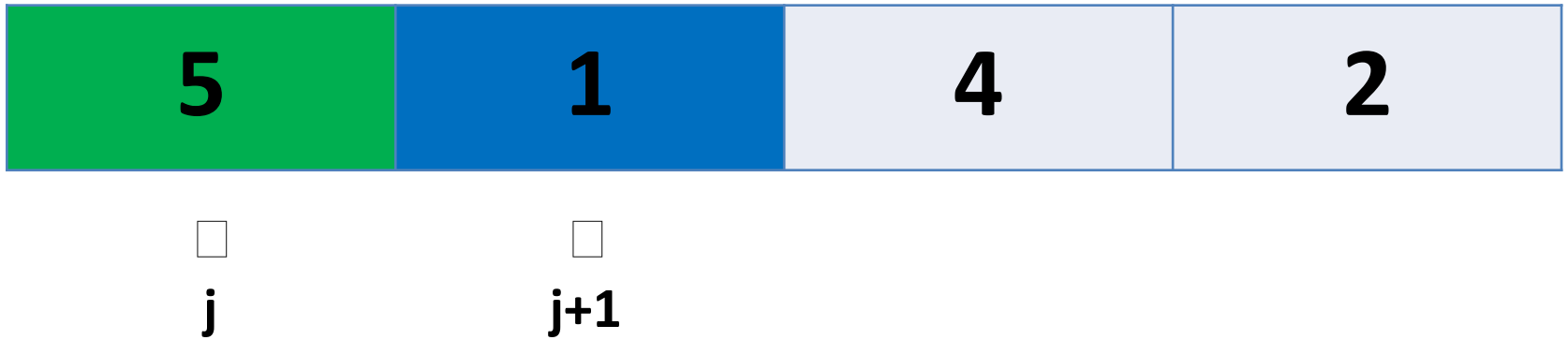
```
bubbleSort( Arr[], totat_elements)
  for i = 0 to total_elements - 1 do:
    swapped = false
    for j = 0 to total_elements - i - 2 do:
      /* compare the adjacent elements */
      if Arr[j] > Arr[j+1] then
        /* swap them */
        swap(Arr[j], Arr[j+1])
        swapped = true
      end if
    end for
    /*if no number was swapped that means array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end
```

Bubble Sort

Example 1: Consider the following Array:

Bubble Sort

- First Iteration:
- Compare



Bubble Sort

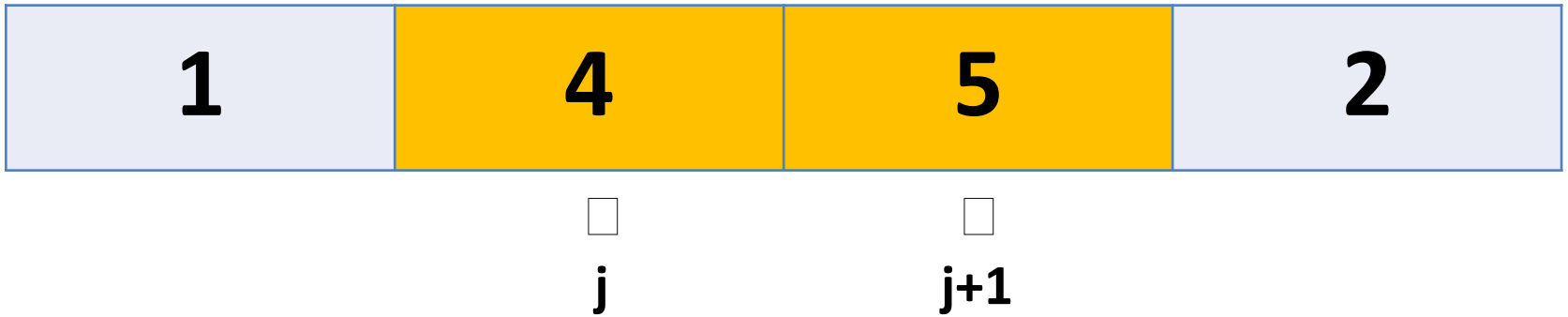
- First Iteration:
- Swap

Bubble Sort

- First Iteration:
- Compare

Bubble Sort

- First Iteration:
- Swap



Bubble Sort

- First Iteration:
- Compare

Bubble Sort

- First Iteration:
- Swap

Bubble Sort

Bubble Sort

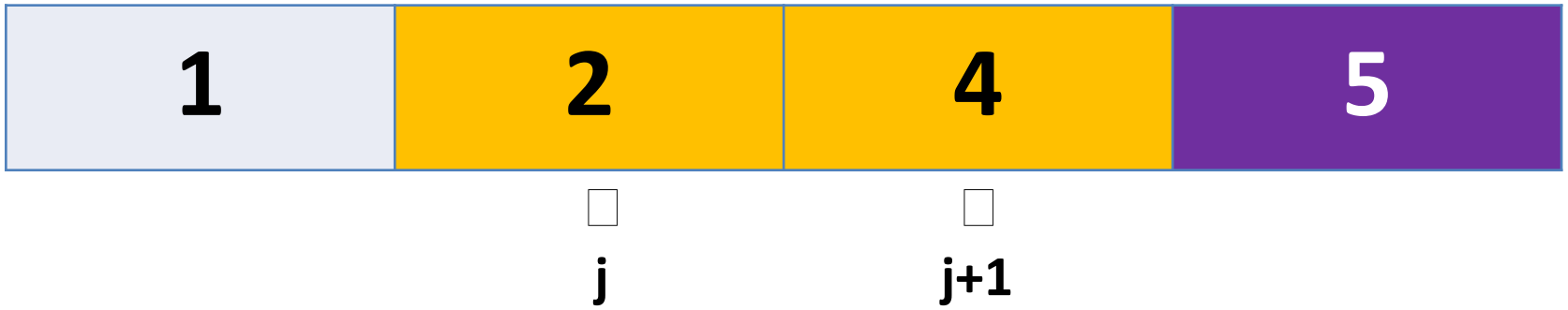
- Second Iteration:
- Compare

Bubble Sort

- Second Iteration:
- Compare

Bubble Sort

- Second Iteration:
- Swap



Bubble Sort

Bubble Sort

- Third Iteration:
- Compare

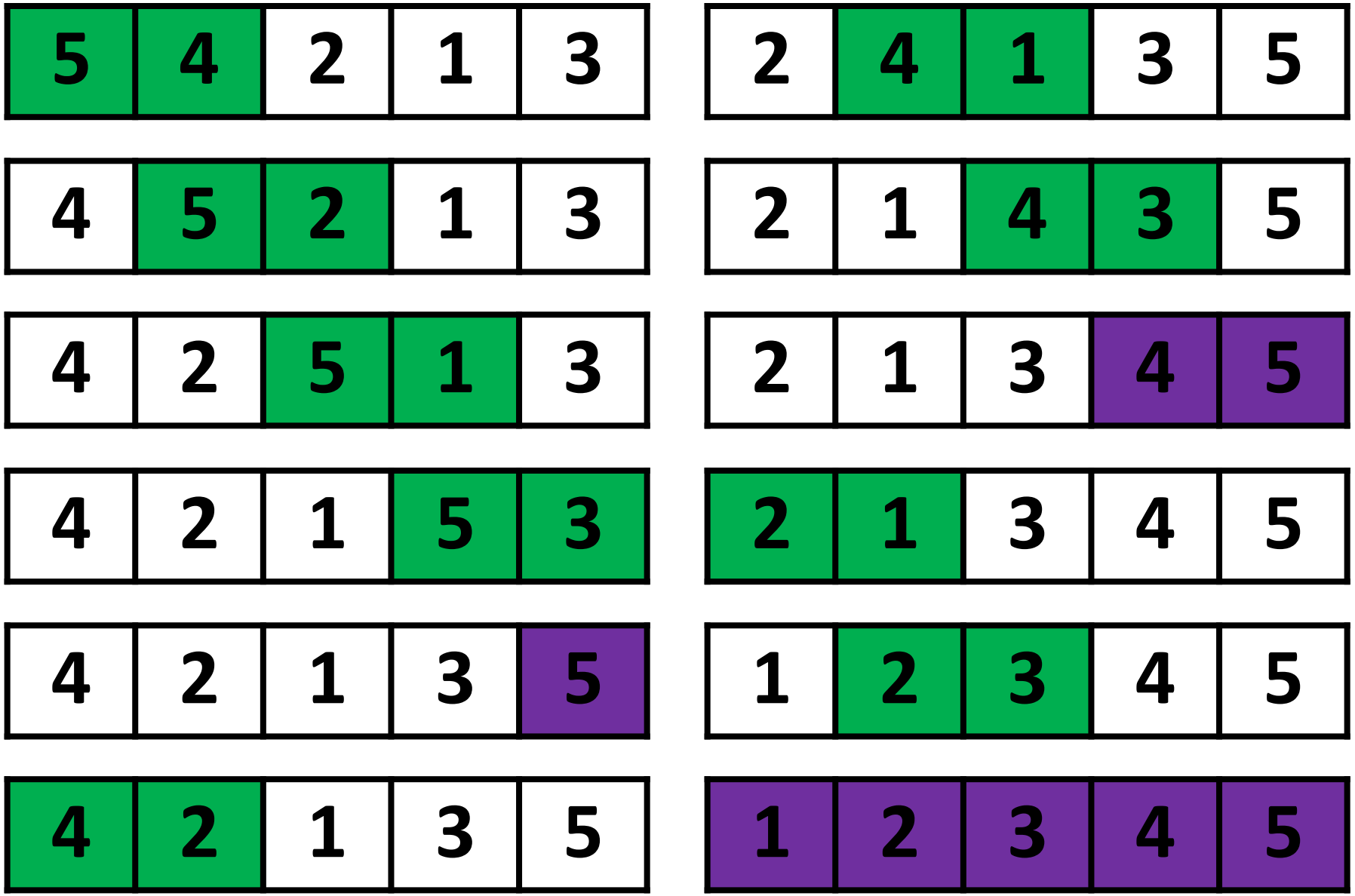
Bubble Sort

Bubble Sort

- Array is now sorted

Bubble Sort

- Example 2:



Selection Sort

Selection Sort

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and placing it at the beginning.

Selection Sort

Algorithm:

- Step1: Find the minimum value in the list
- Step2: Swap it with the value in the current position
- Step 3: Repeat this process for all the elements until the entire array is sorted

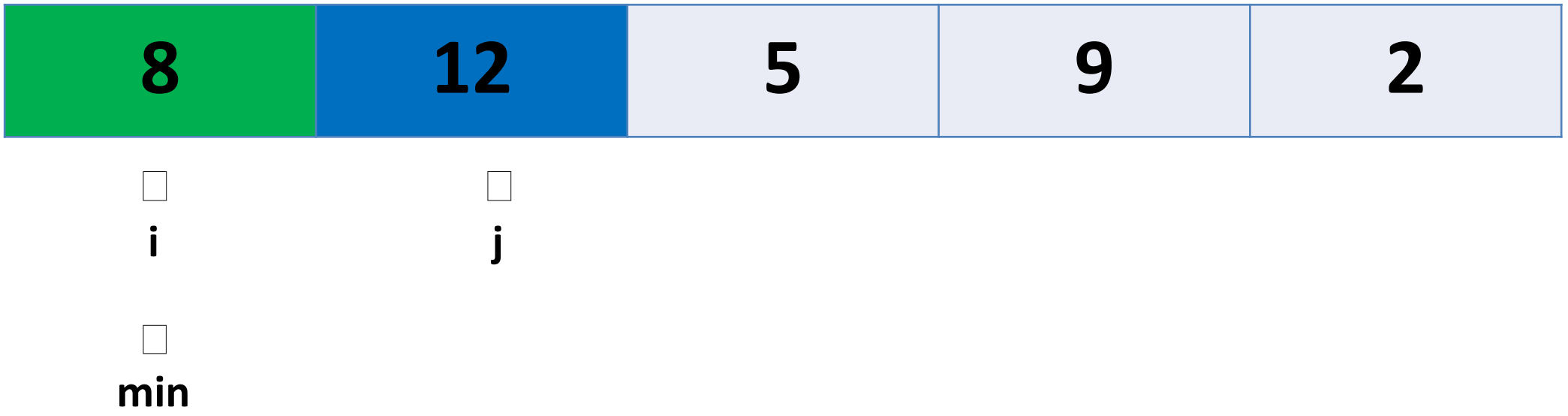
Selection Sort

- Example 1 Assume the following Array:

8	12	5	9	2

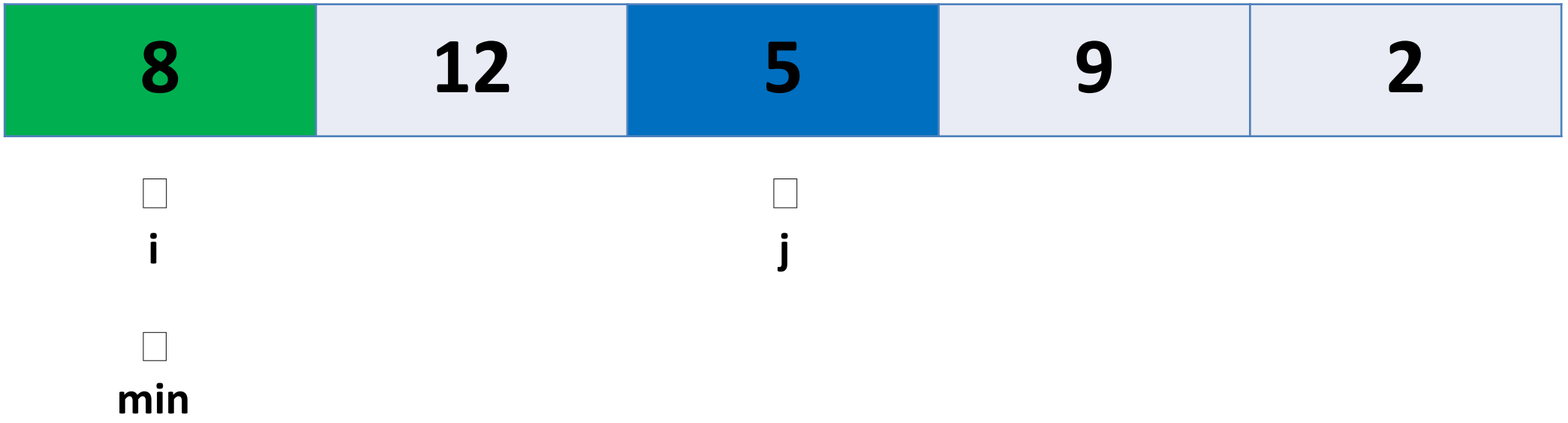
Selection Sort

- Compare



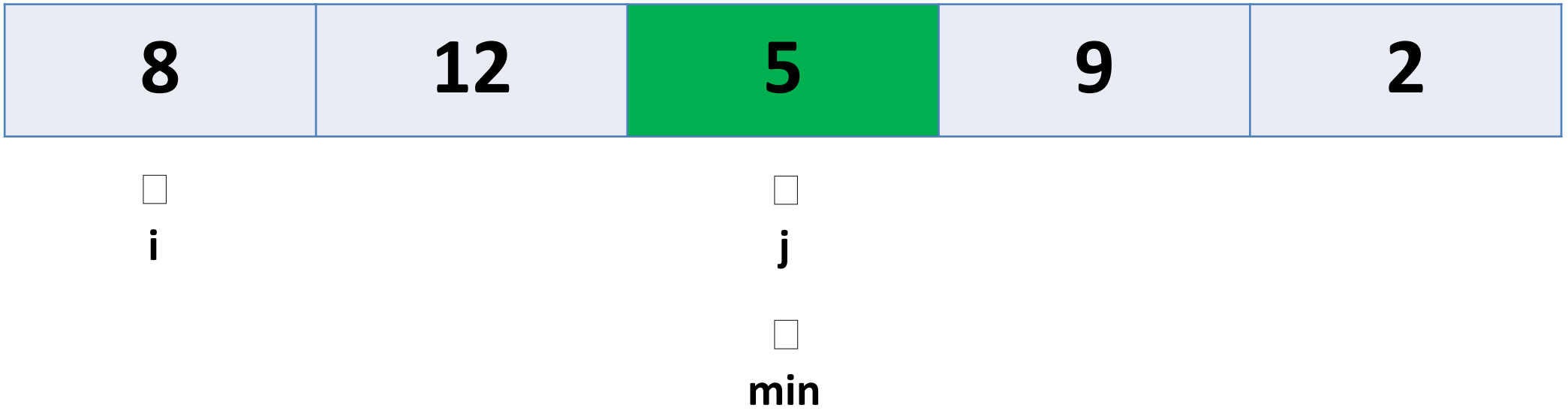
Selection Sort

- Compare



Selection Sort

- Move



Selection Sort

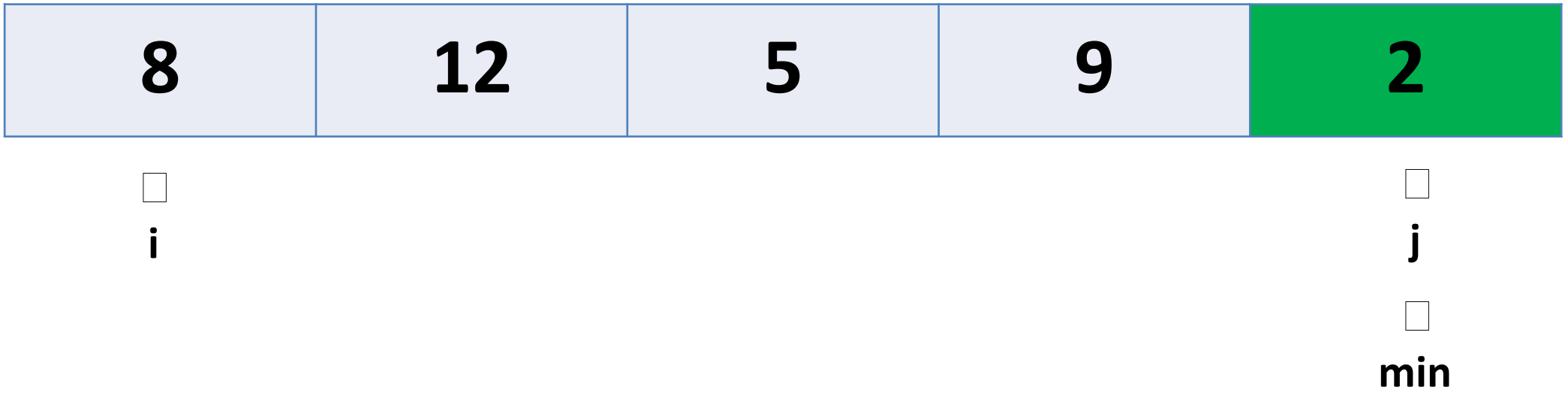
- **Compare**

Selection Sort

- Compare

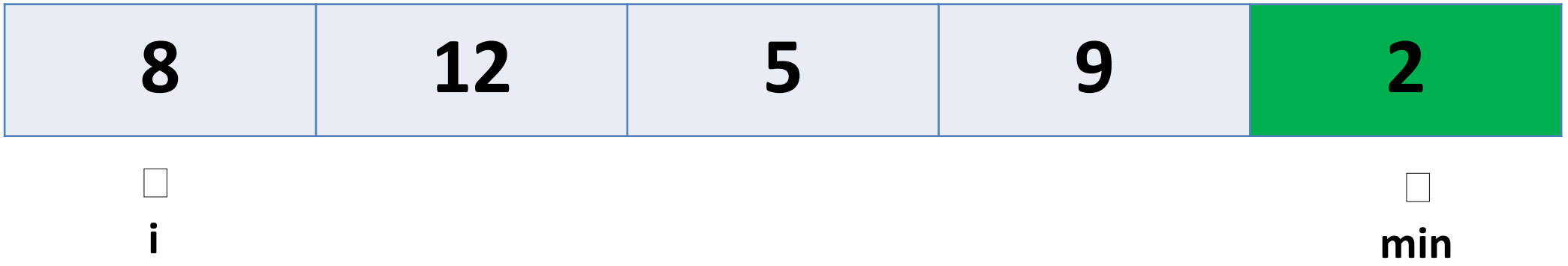
Selection Sort

- Move



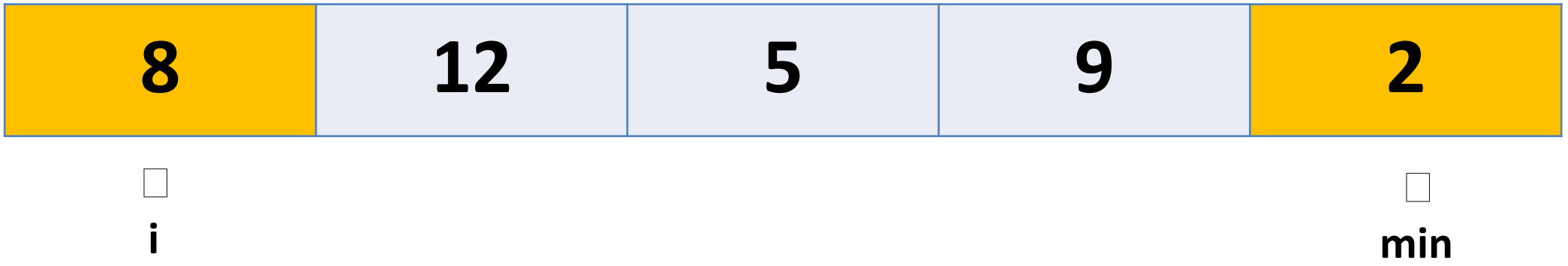
Selection Sort

- Smallest



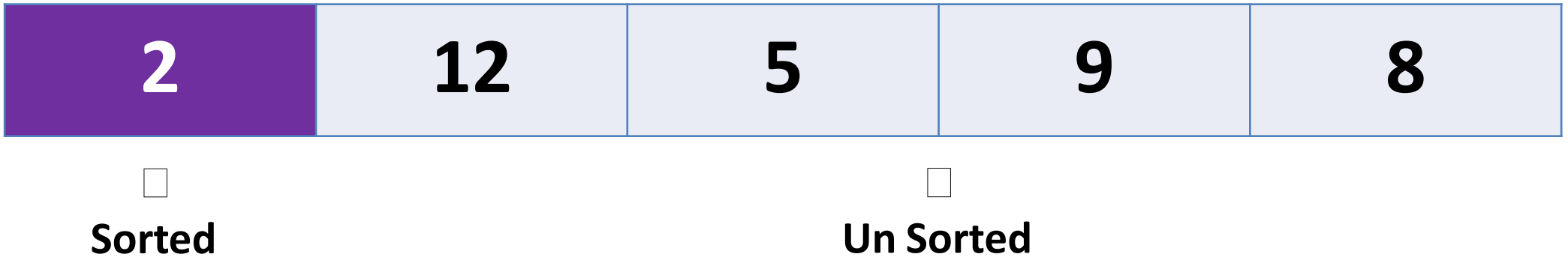
Selection Sort

- Swap



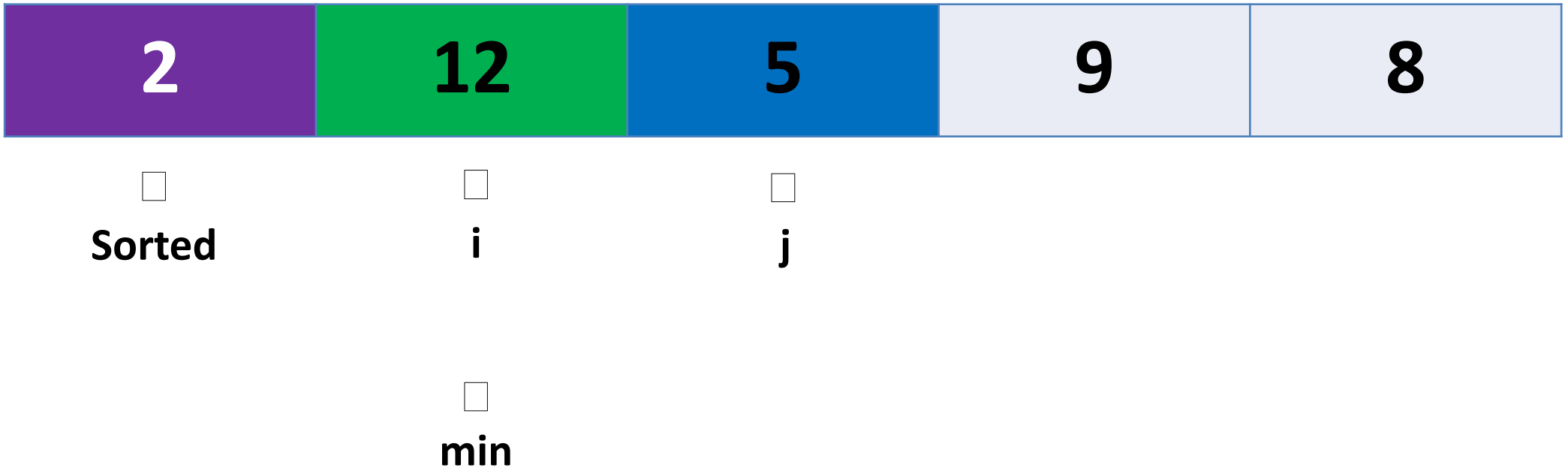
Selection Sort

- Sorted
- Un Sorted



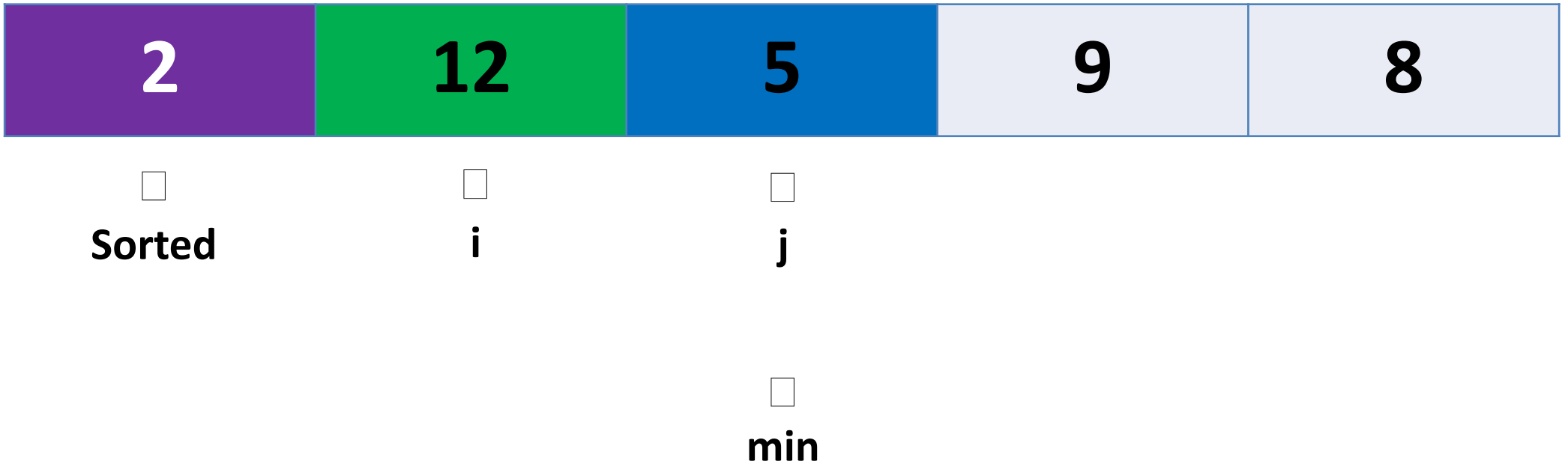
Selection Sort

❑ Compare



Selection Sort

- Move



Selection Sort

- Compare

Selection Sort

- Compare

Selection Sort

- Smallest

Selection Sort

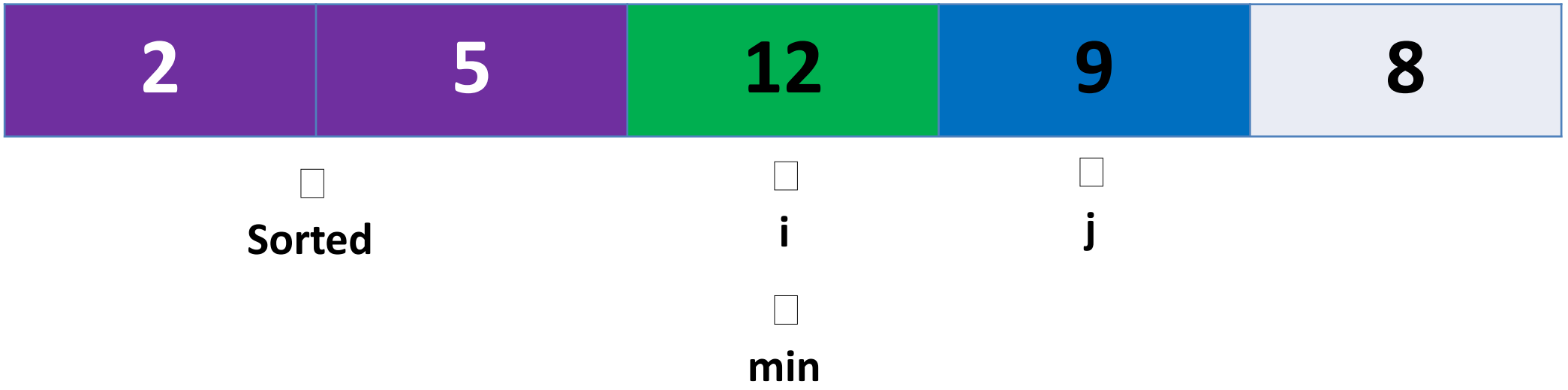
- Swap

Selection Sort

- Sorted
- Un Sorted

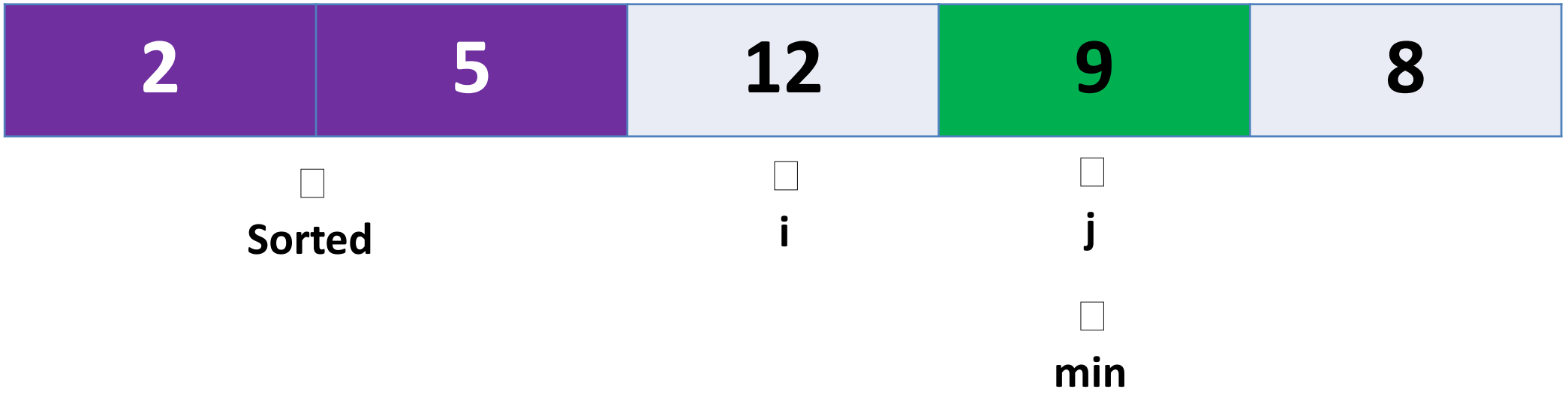
Selection Sort

- Compare



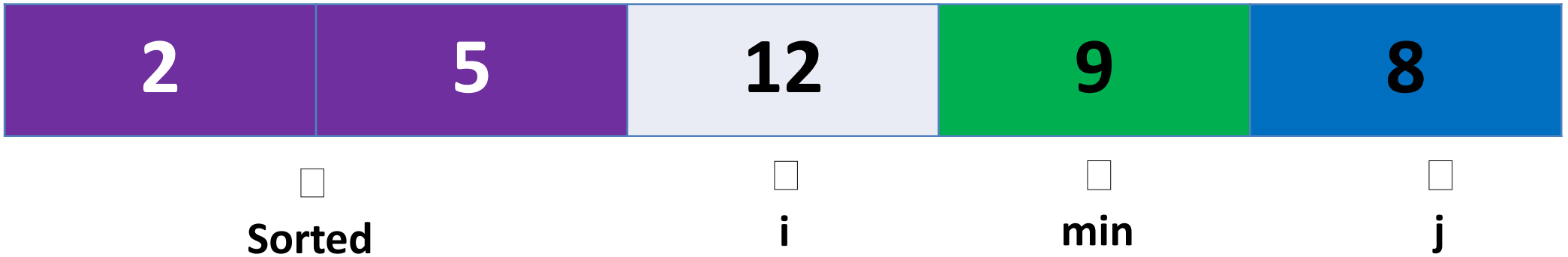
Selection Sort

- Move



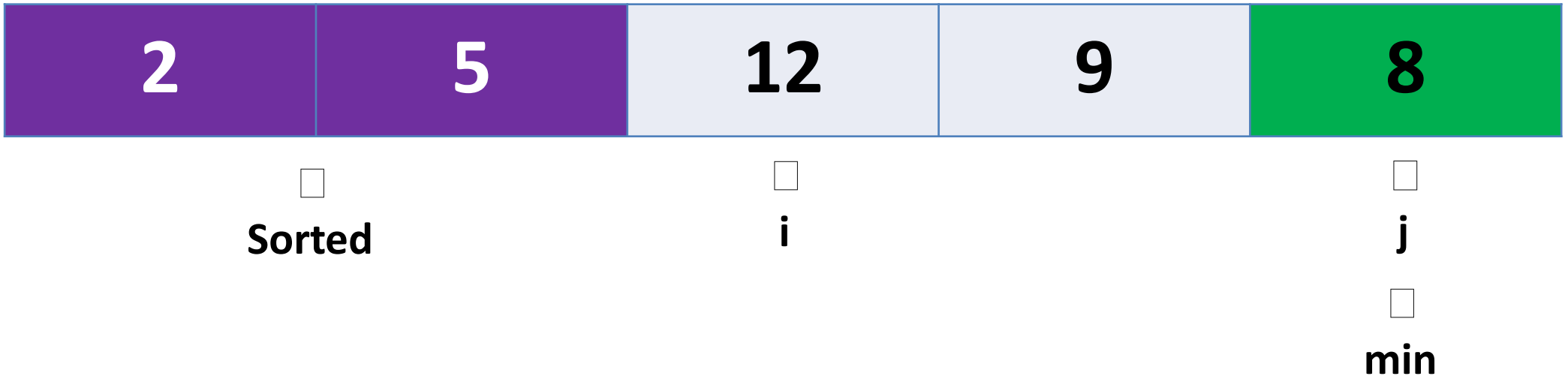
Selection Sort

- Compare



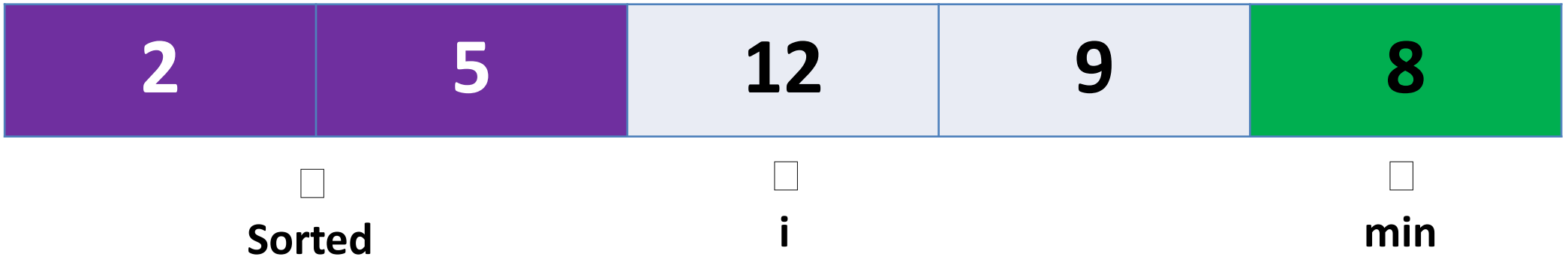
Selection Sort

- Move



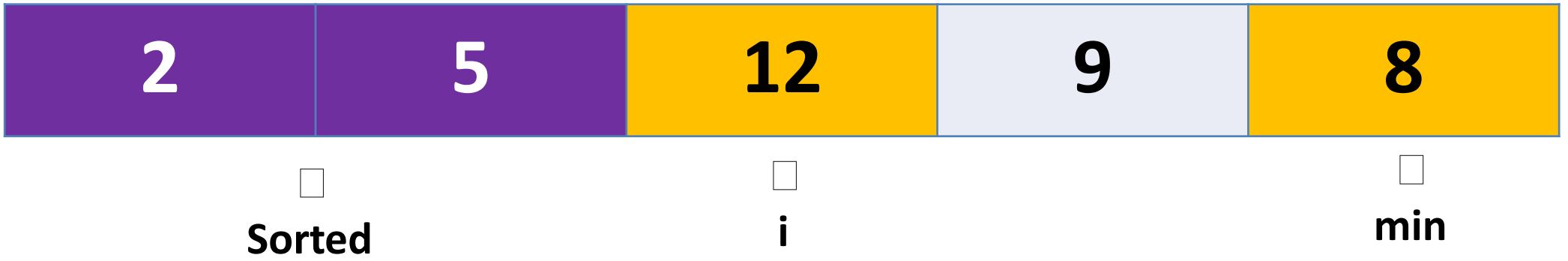
Selection Sort

- Smallest



Selection Sort

- Swap

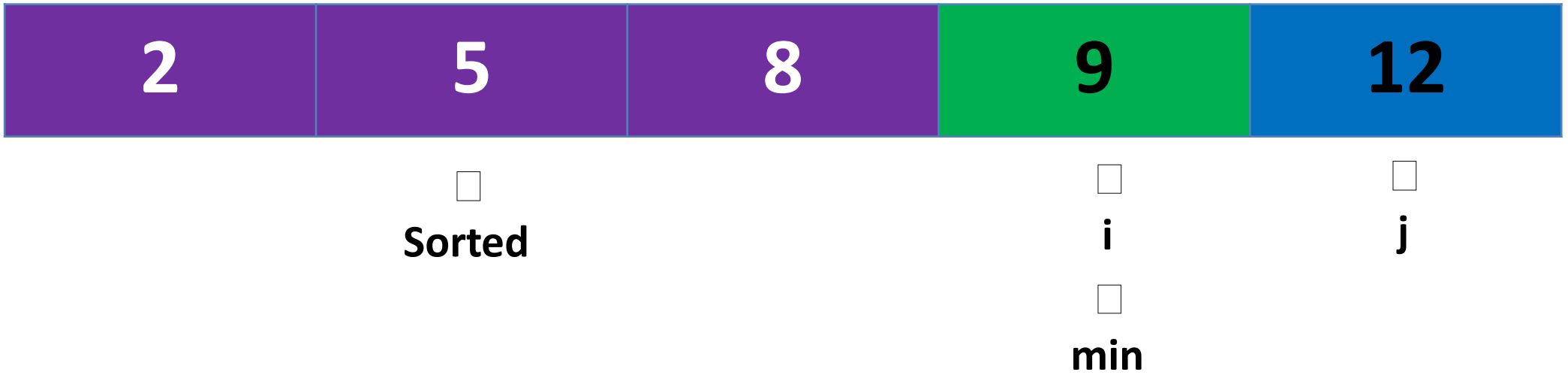


Selection Sort

- Sorted
- Un Sorted

Selection Sort

- Compare

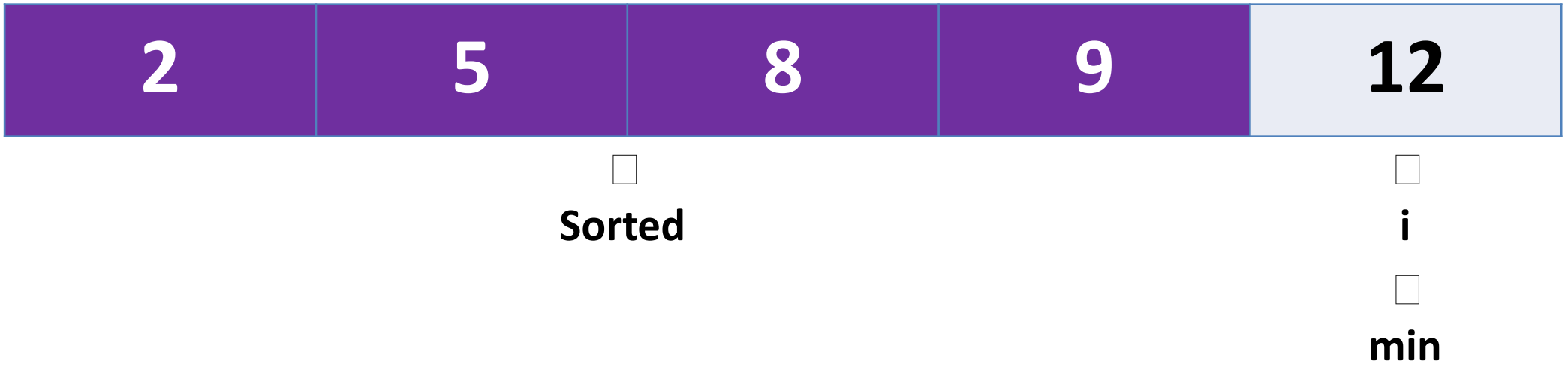


Selection Sort

- Sorted
- Un Sorted

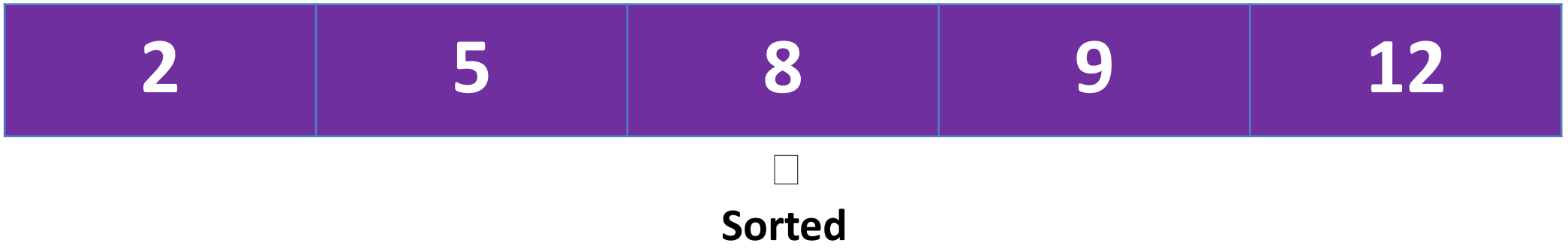
Selection Sort

- Sorted
- Un Sorted



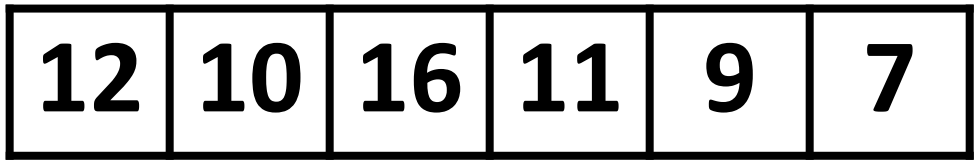
Selection Sort

- Array is now sorted



Selection Sort

- Example 2:



12	10	16	11	9	7
7	10	16	11	9	12
7	9	16	11	10	12
7	9	10	11	16	12
7	9	10	11	16	12
7	9	10	11	12	16

Insertion Sort

Insertion Sort

- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion Sort

1. If it is the first element, it is already sorted.
2. Pick the next element.
3. Compare with all the elements in sorted sub-list.
4. Shift all the the elements in sorted sub-list that is greater than the value to be sorted.
5. Insert the value.
6. Repeat until list is sorted.

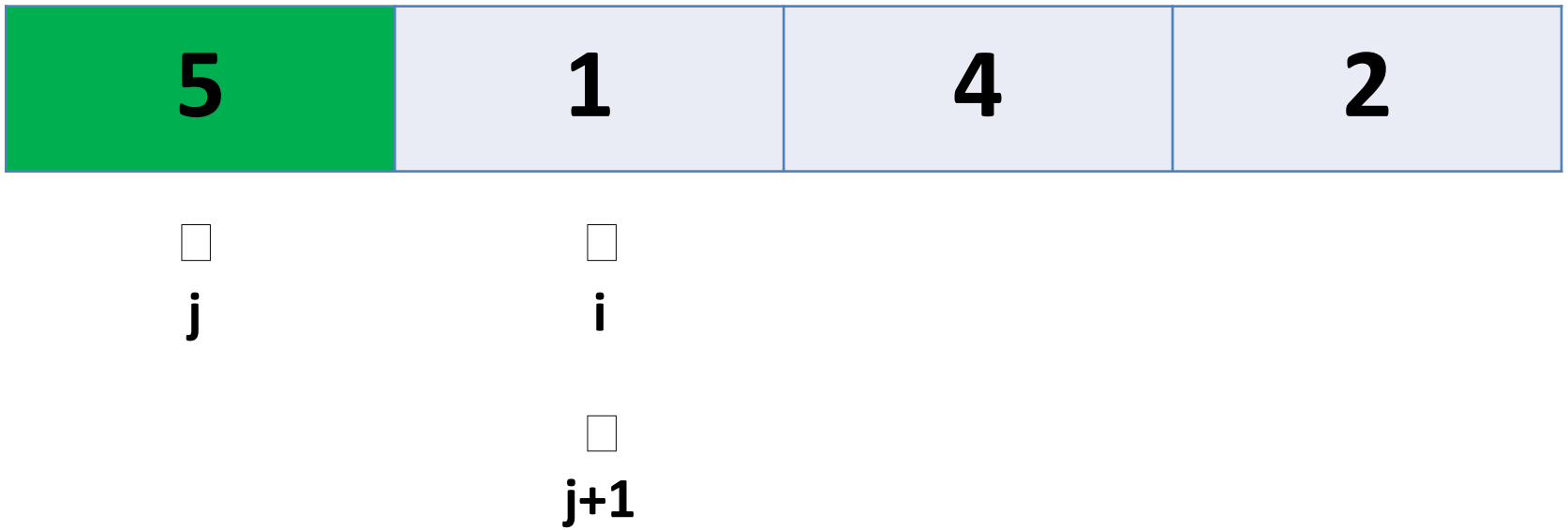
Insertion Sort

- Assume the following Array:

5	1	4	2

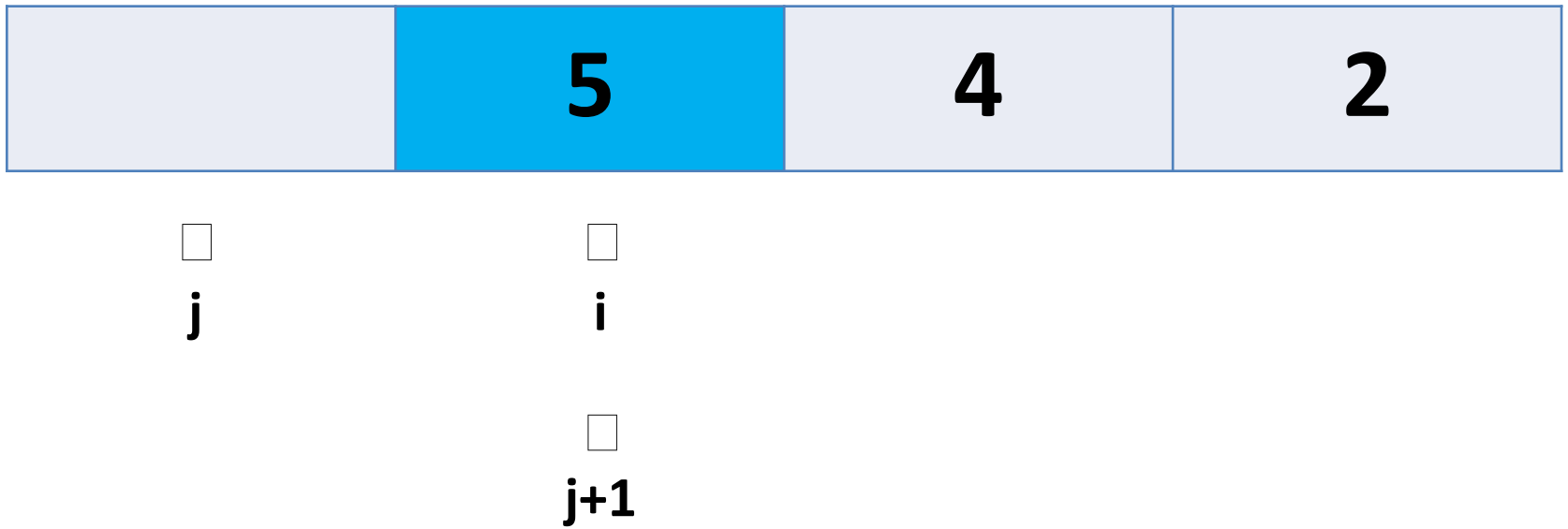
Insertion Sort

- Compare
- Store=



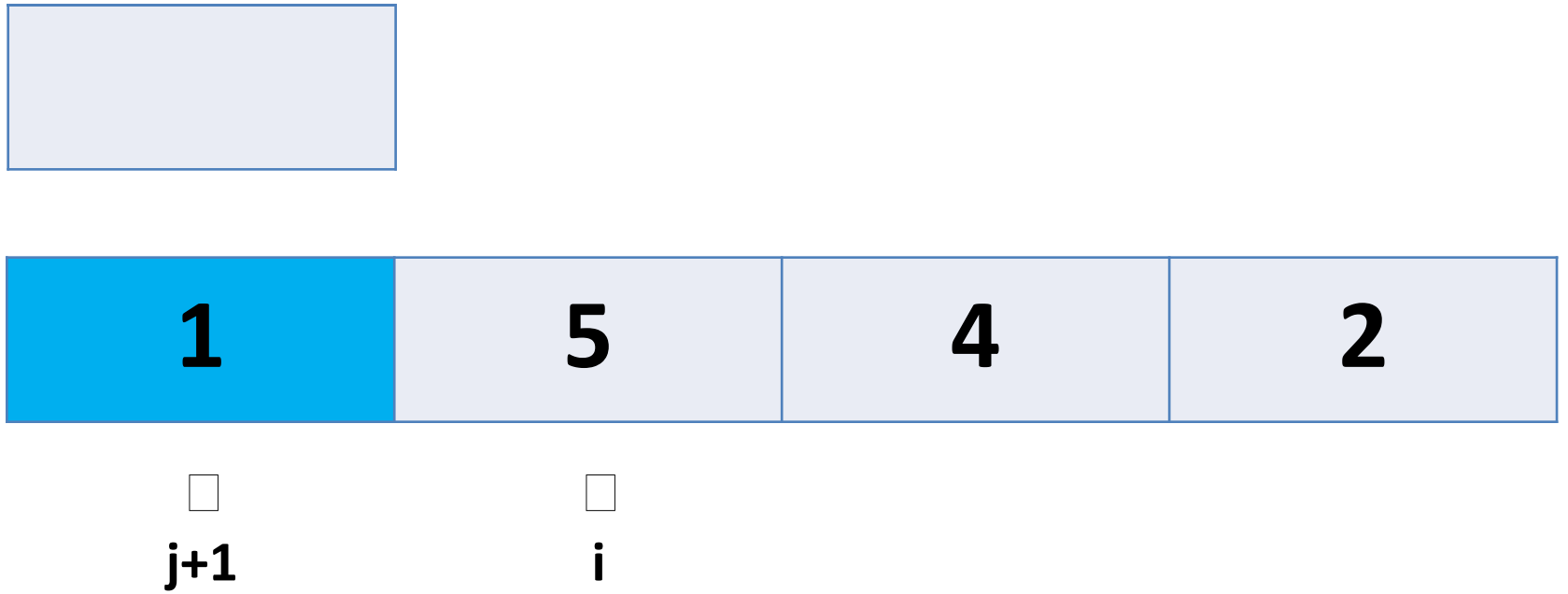
Insertion Sort

- Move
- Store=



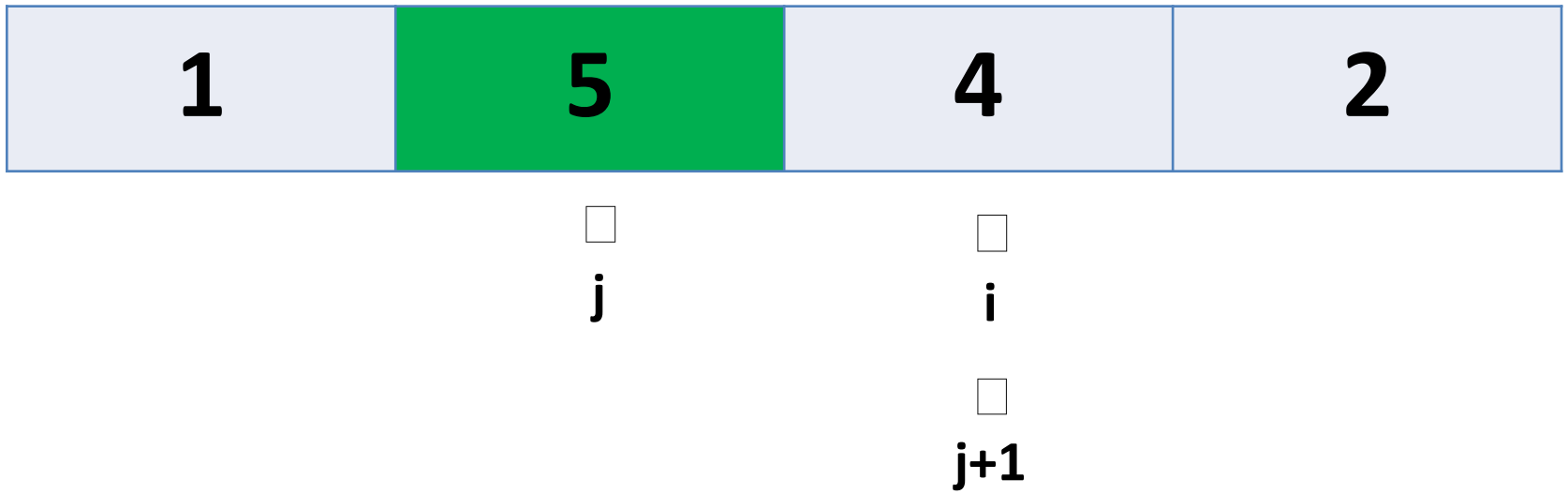
Insertion Sort

- Move
- Store=



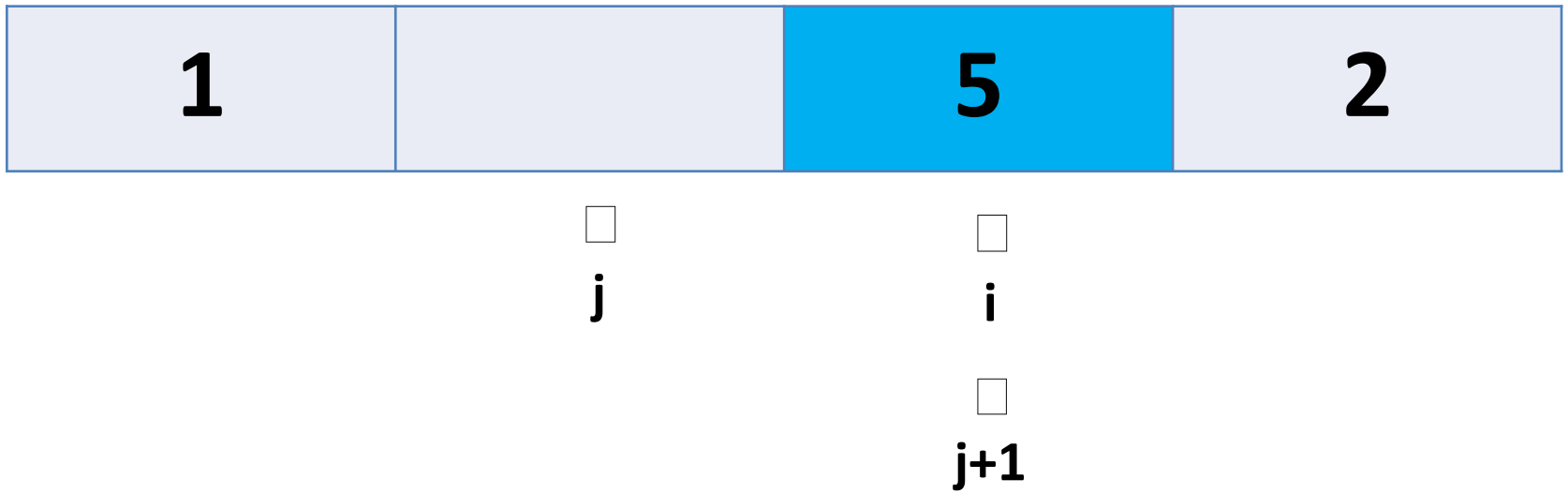
Insertion Sort

- Compare
- Store=



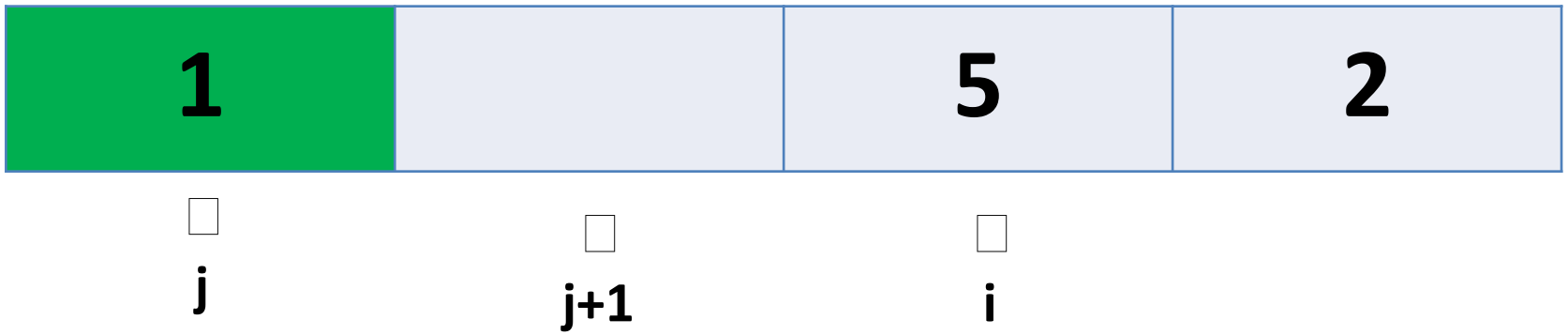
Insertion Sort

- Move
- Store=



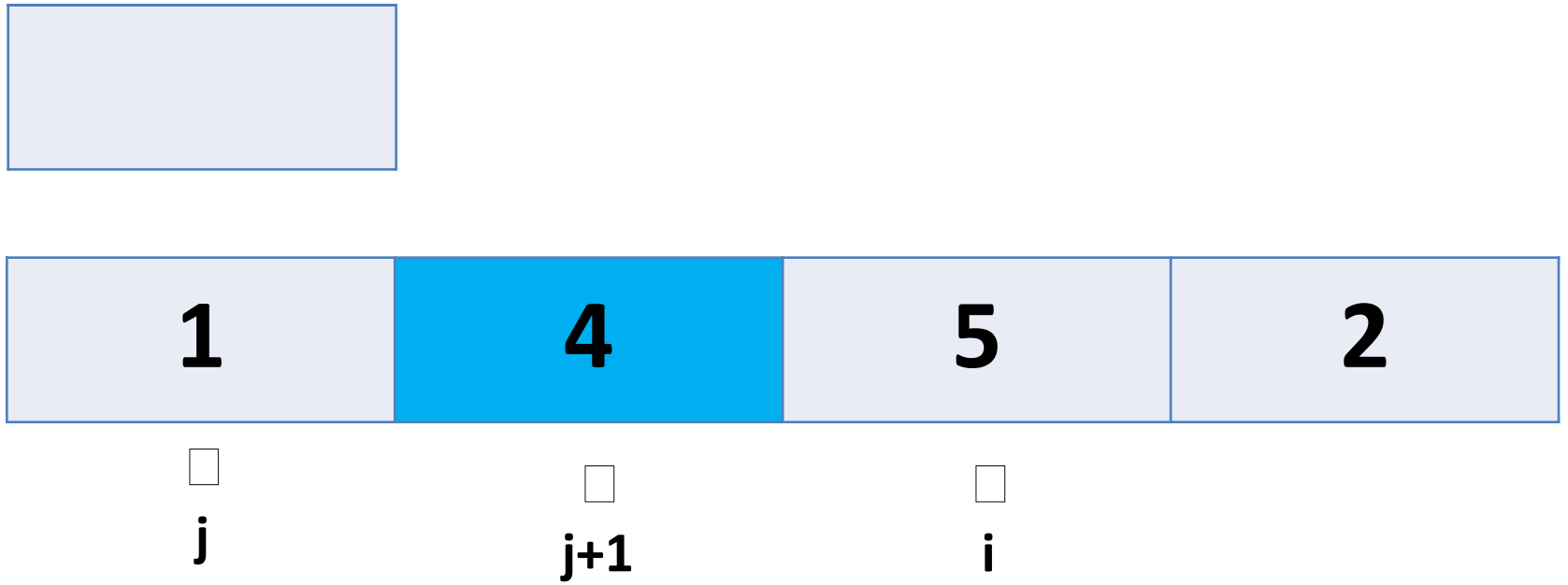
Insertion Sort

- Compare
- Store=



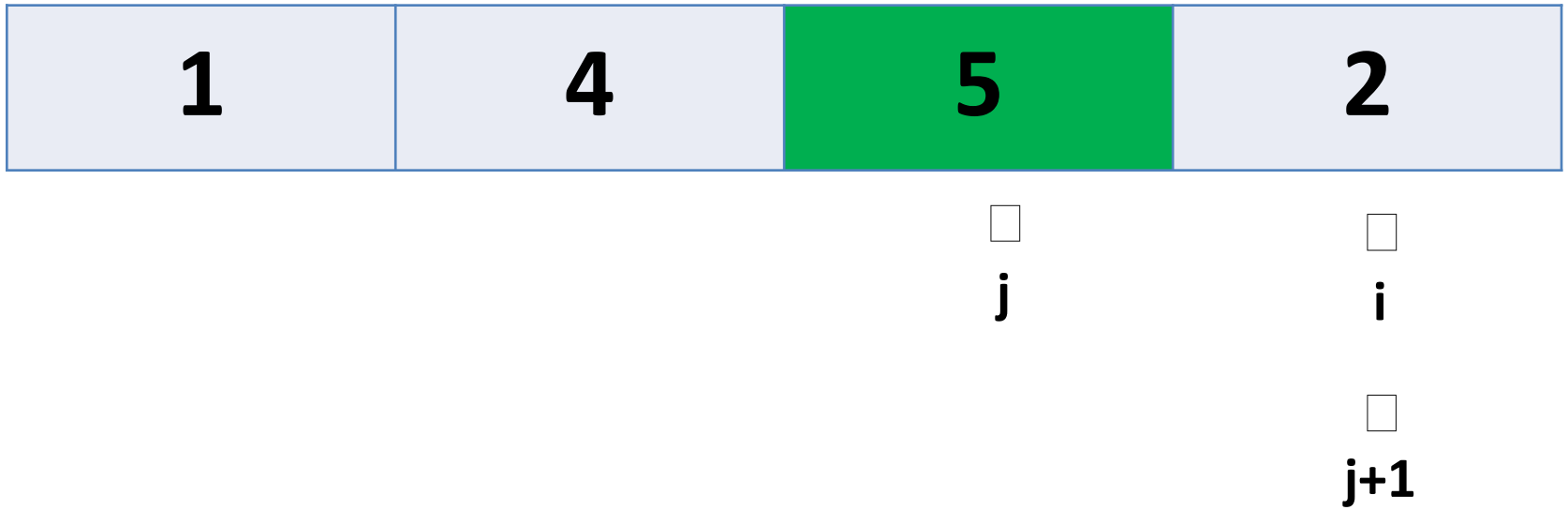
Insertion Sort

- Move
- Store=



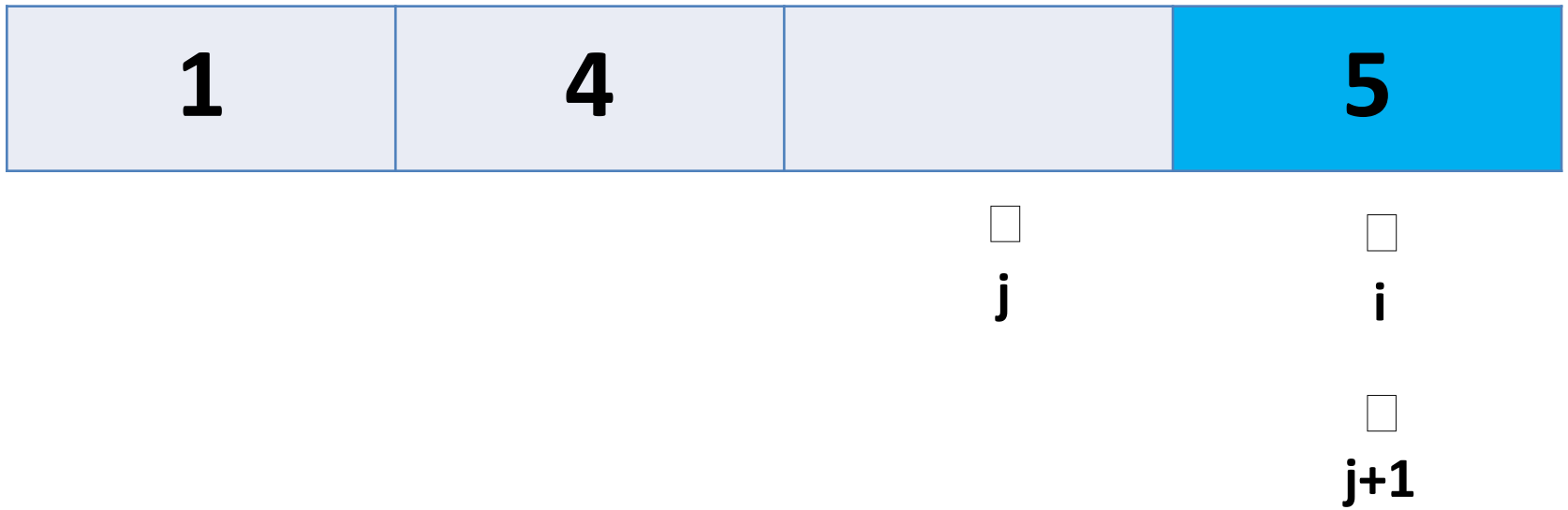
Insertion Sort

- Compare
- Store= **2**



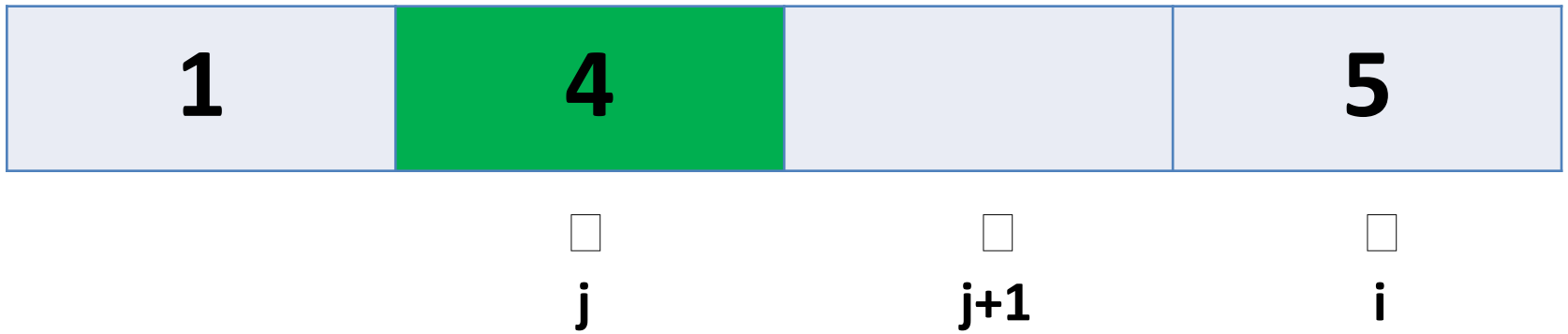
Insertion Sort

- Move
- Store=



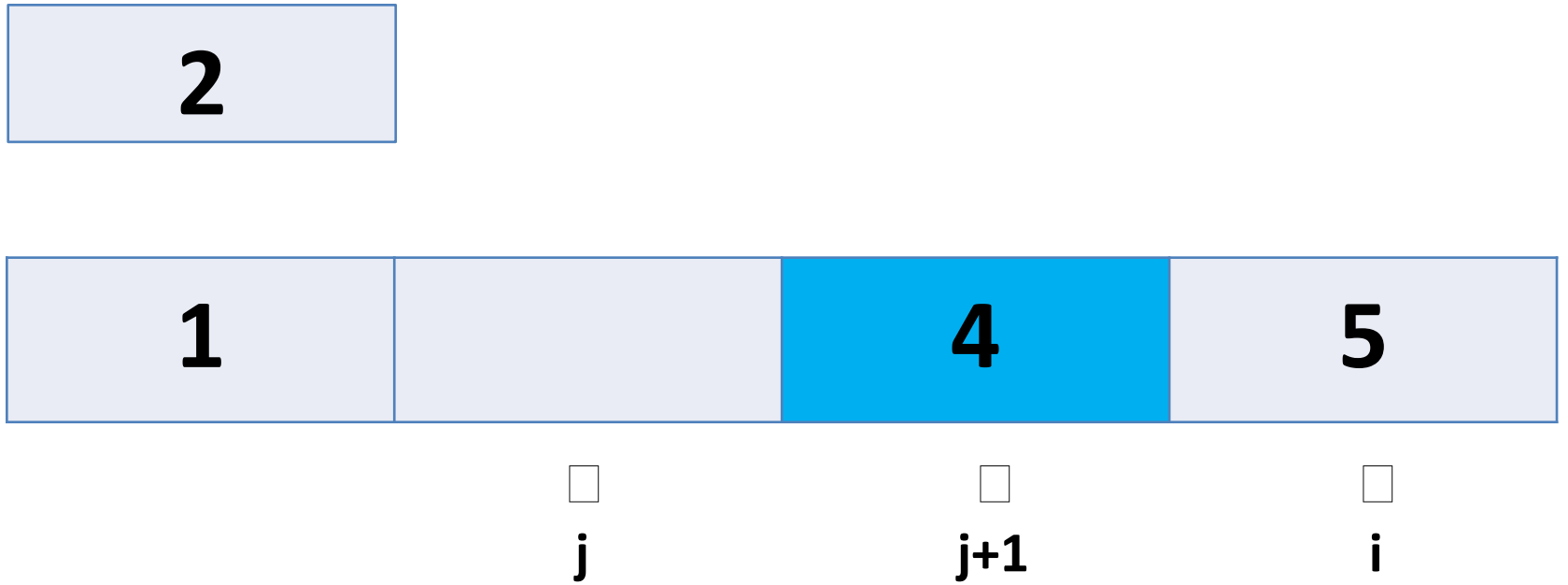
Insertion Sort

- Compare
- Store= **2**



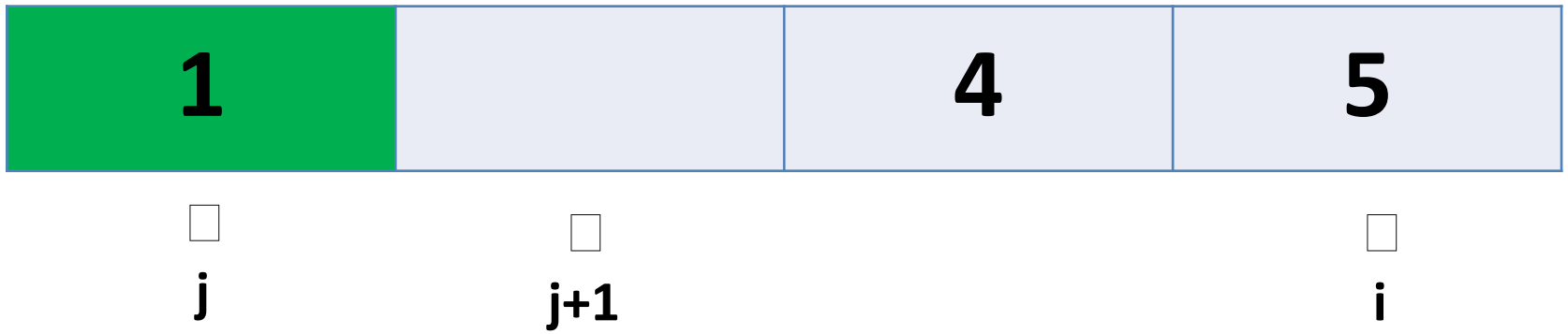
Insertion Sort

- Move
- Store=



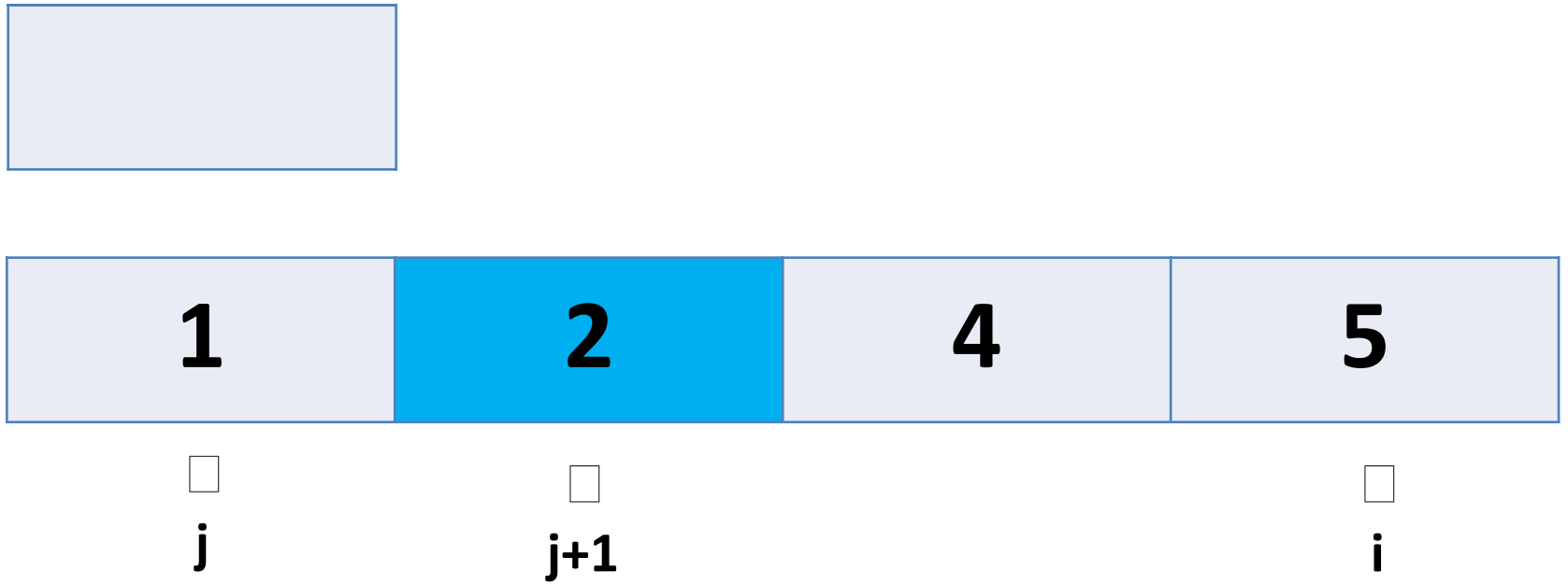
Insertion Sort

- Compare
- Store=



Insertion Sort

- Compare
- Store=

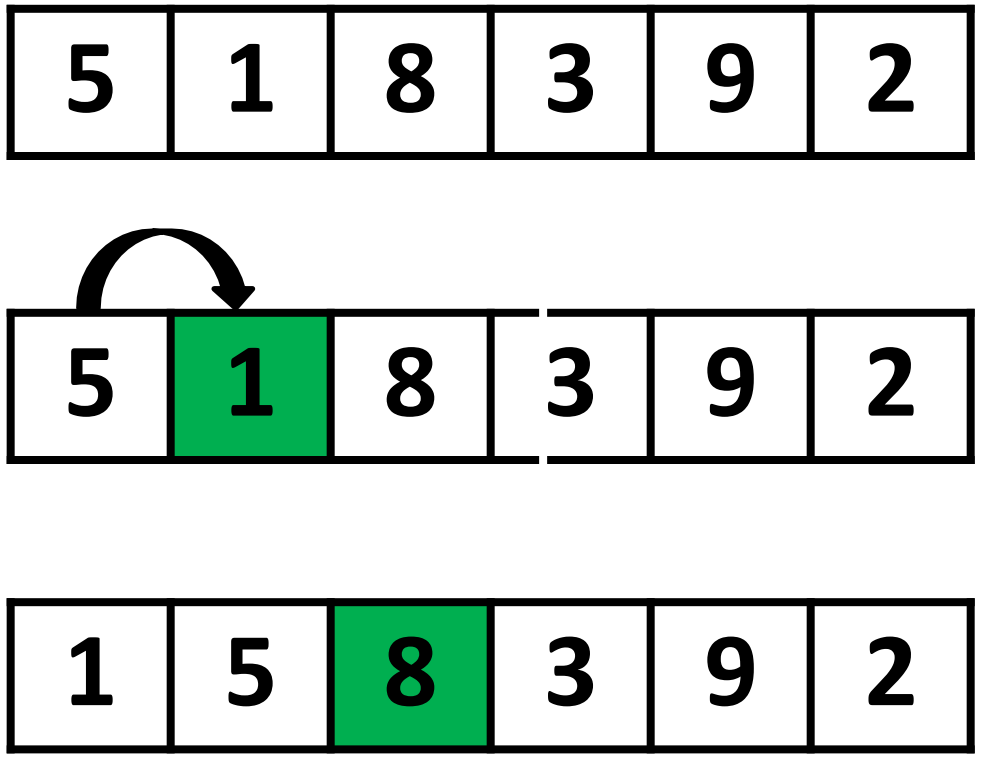


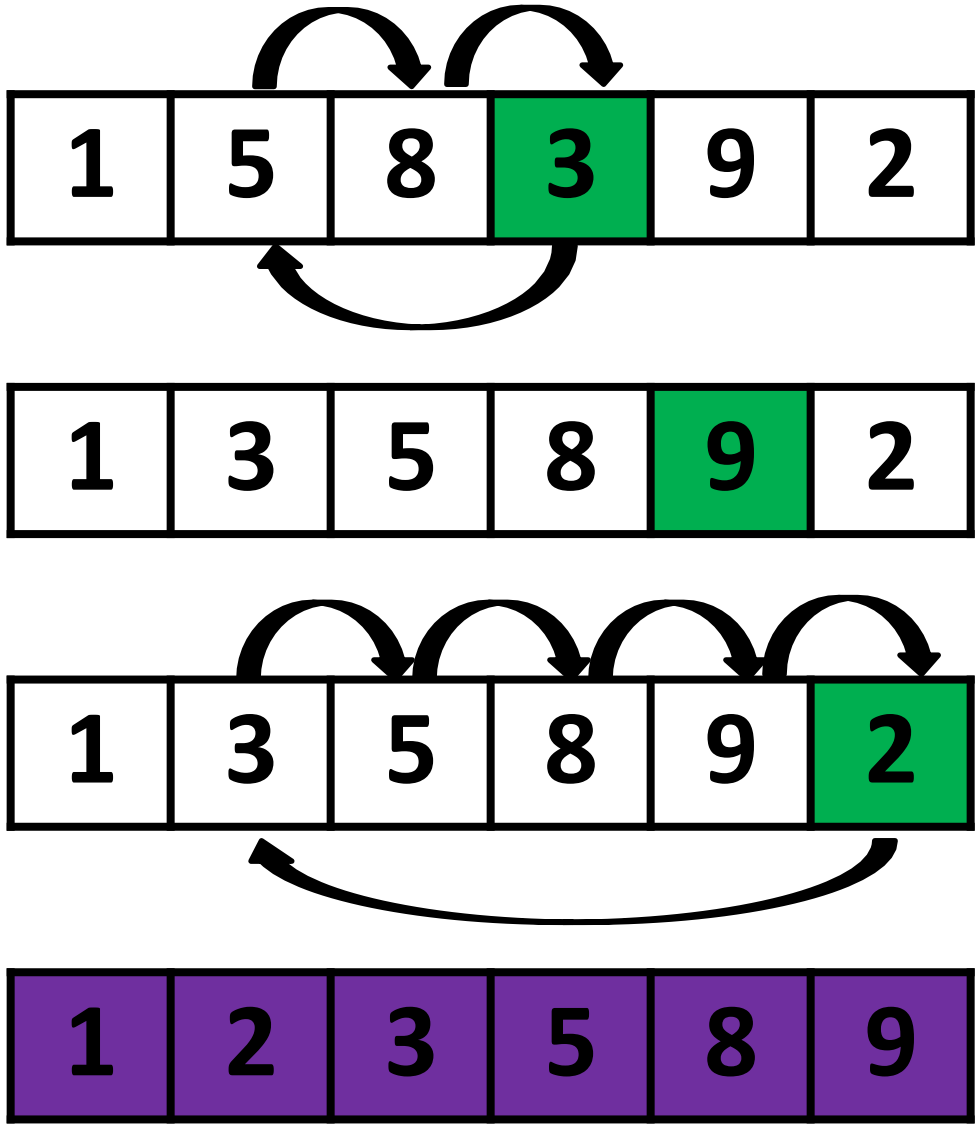
Insertion Sort

- Array is now sorted

Selection Sort

- Example 2:





Radix Sort

Radix Sort

- Radix sort is an algorithm that sorts numbers by processing digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD).
- The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

Radix Sort

❑ Algorithm:

- **Step1:** Take the least significant digit of each element
- **Step2** : Sort the list of elements based on that digit
- **Step3** : Repeat the sort with each more significant digit

Radix Sort

- Assume the following Array:

170	45	75	90	802	24	2	66

Radix Sort

- The Sorted list will appear after three steps

170	45	75	90	802	24	2	66

170	90	802	2	24	45	75	66

802	2	24	45	66	170	75	90

2	24	45	66	75	90	170	802

Radix Sort

- Step1: Sorting by least significant digit (1s place)

170	45	75	90	802	24	2	66

170	90	802	2	24	45	75	66

Radix Sort

- Step2: Sorting by next digit (10s place)

170	90	802	2	24	45	75	66

802	2	24	45	66	170	75	90

Radix Sort

- Step3: Sorting by most significant digit (100s place)

802	2	24	45	66	170	75	90

2	24	45	66	75	90	170	802

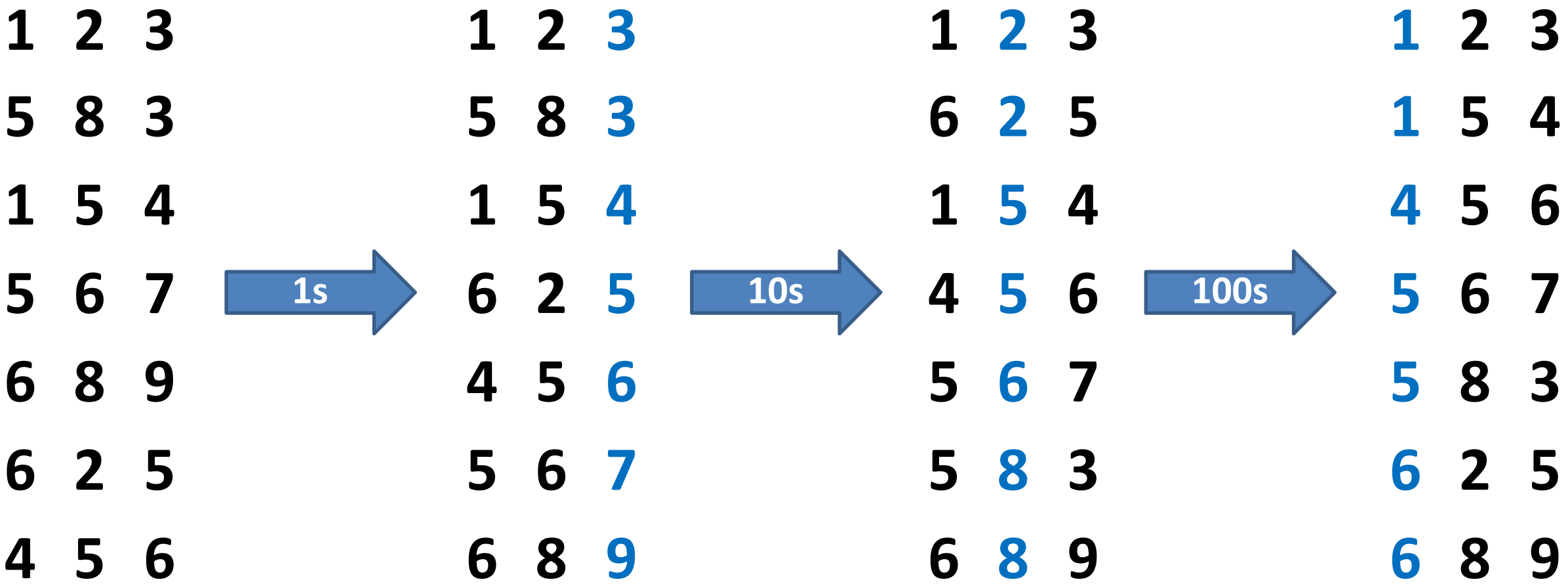
Radix Sort

- Array is now sorted

2	24	45	66	75	90	170	802

Radix Sort

❑ Example 2



Merge Sort

❑ Algorithm:

- **Step1:** Divide the list recursively into two halves until it can no more be divided
- **Step2** : Merge (**Conquer**) the smaller lists into new list in sorted order

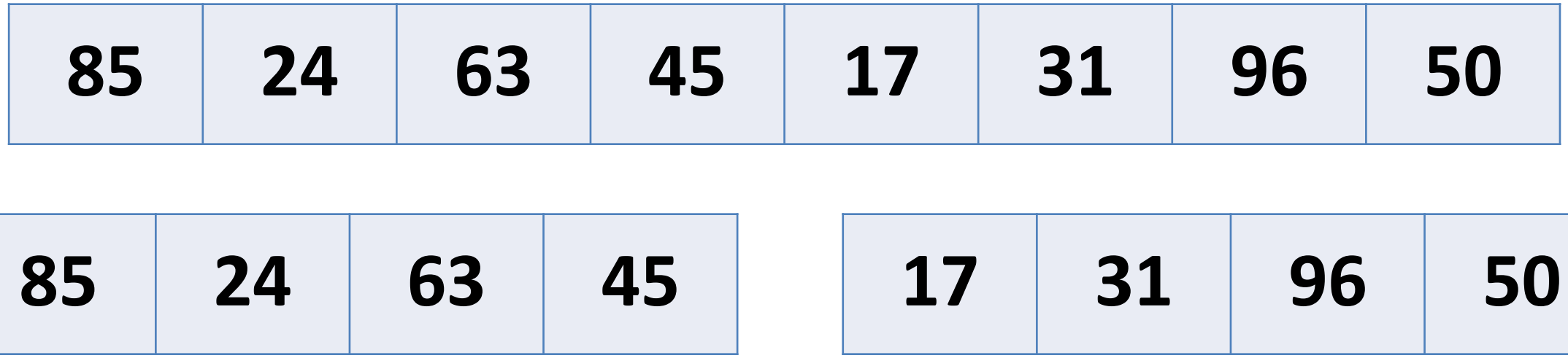
Merge Sort

- Assume the following Array:

85	24	63	45	17	31	96	50

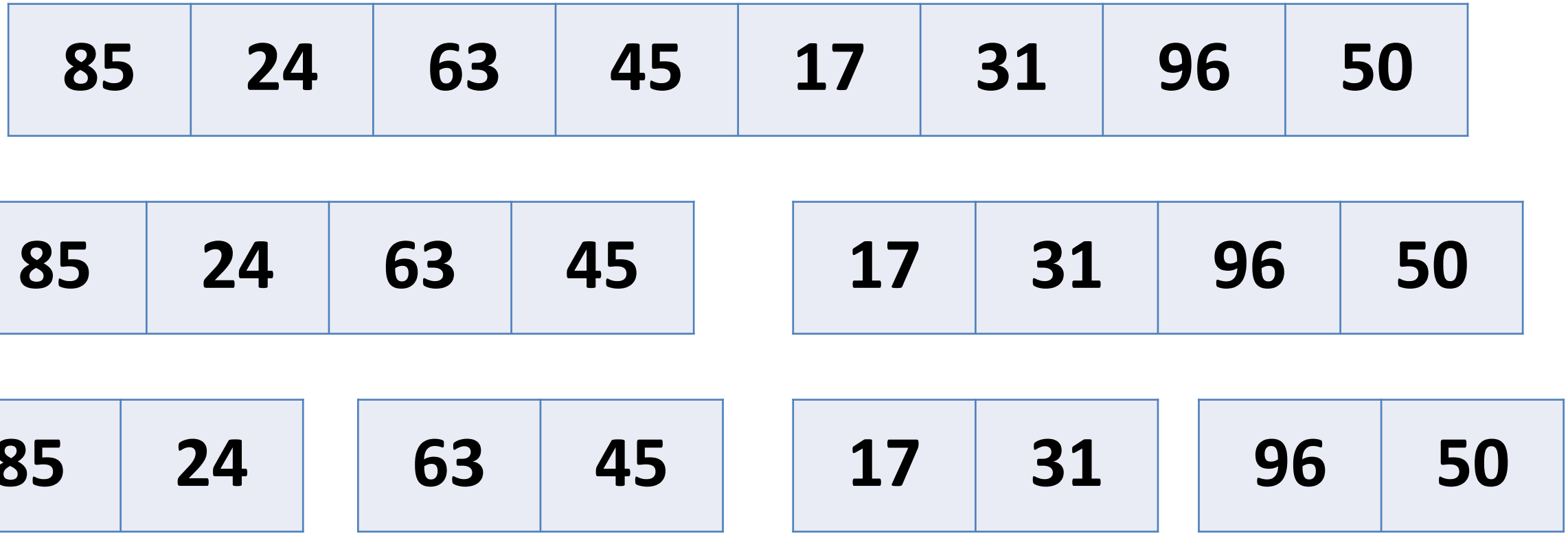
Merge Sort

- Divide



Merge Sort

- Divide



Merge Sort

- Divide

Merge Sort

- Sort & Merge

85	24	63	45	17	31	96	50

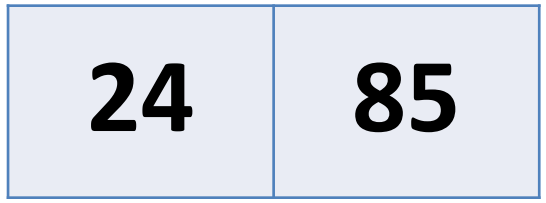
Merge Sort

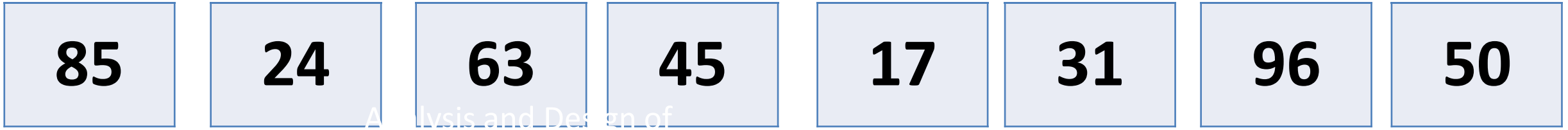
- Sort & Merge

85	24	63	45	17	31	96	50

Merge Sort

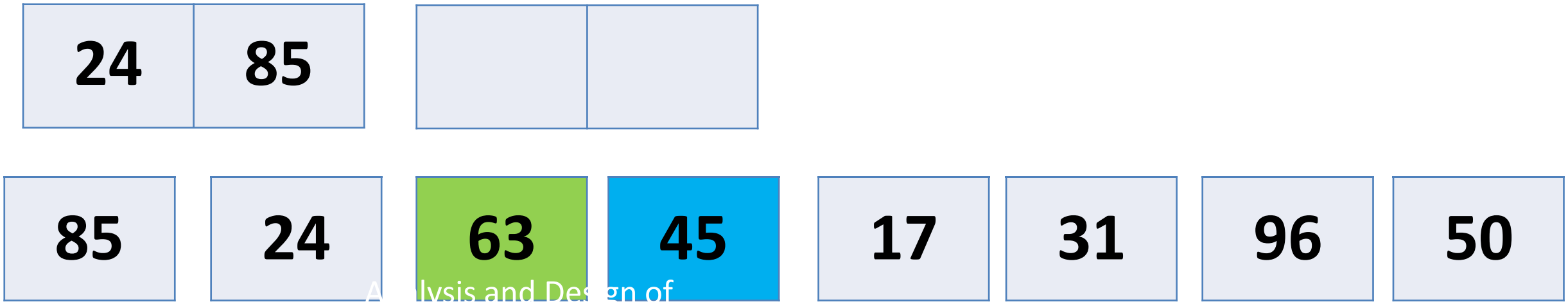
- Sort & Merge





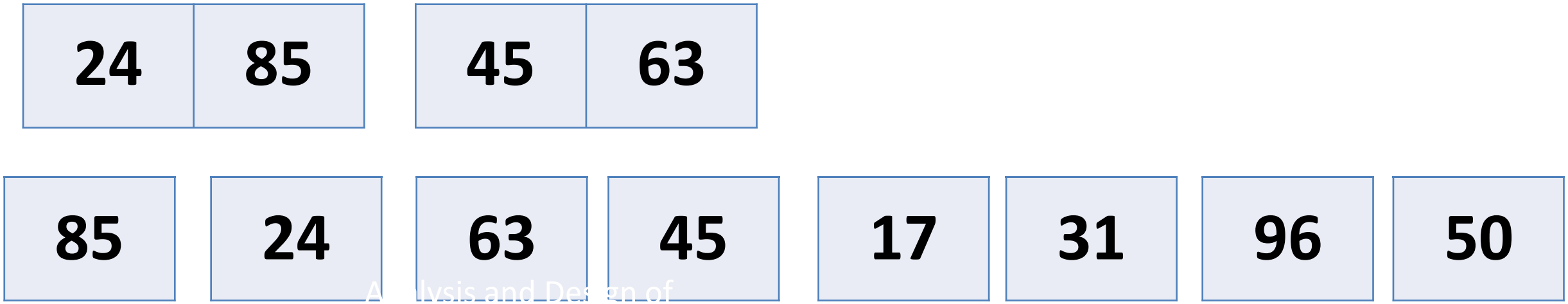
Merge Sort

- Sort & Merge



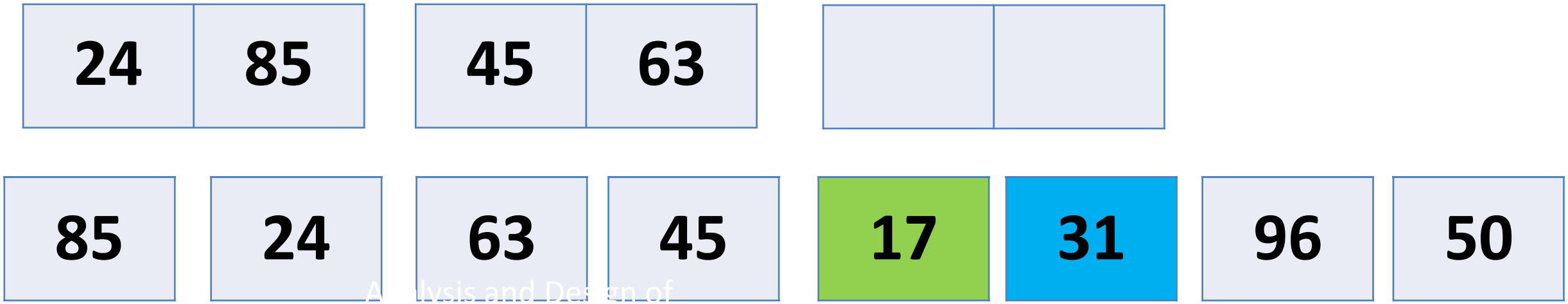
Merge Sort

- Sort & Merge



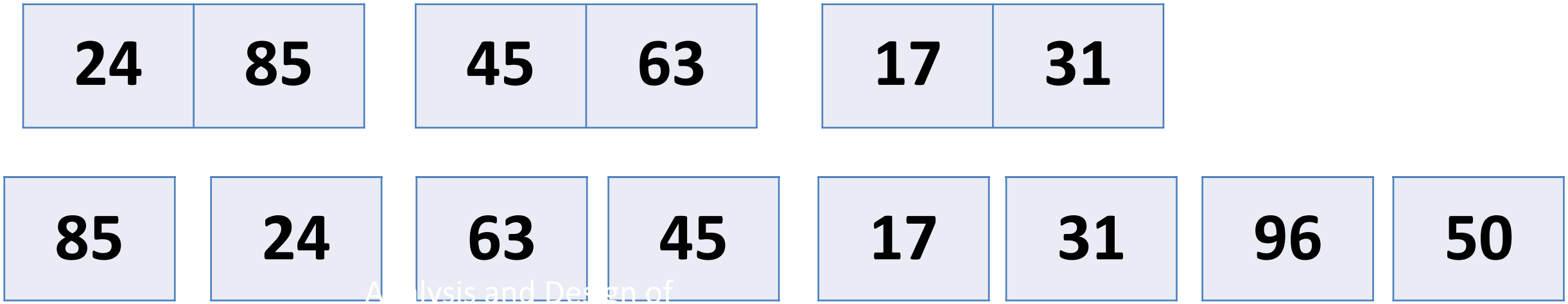
Merge Sort

- Sort & Merge



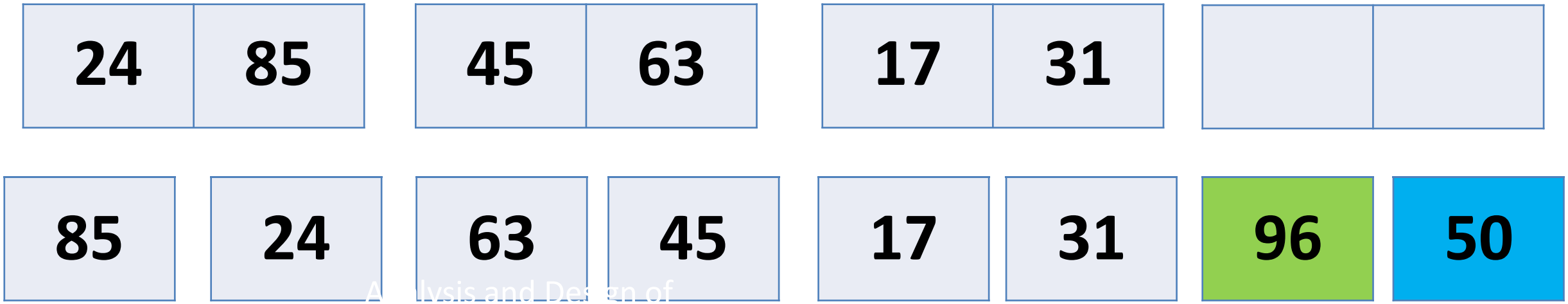
Merge Sort

- Sort & Merge



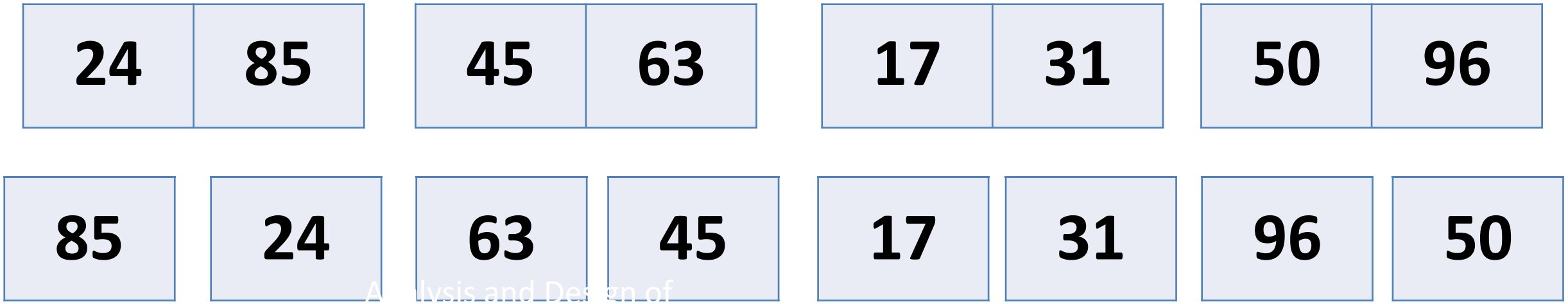
Merge Sort

- Sort & Merge



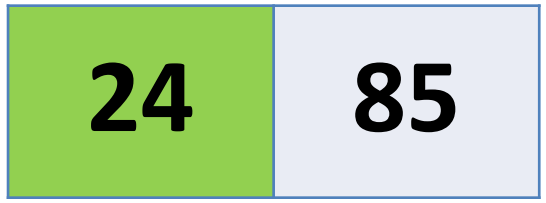
Merge Sort

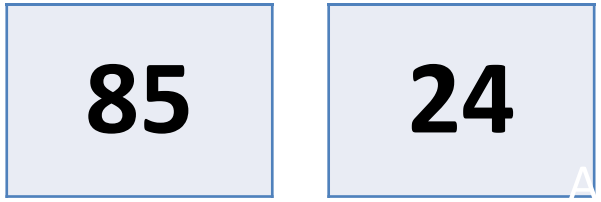
- Sort & Merge

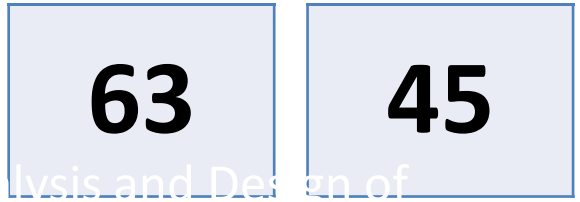


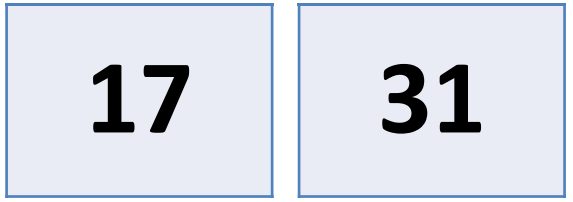
Merge Sort

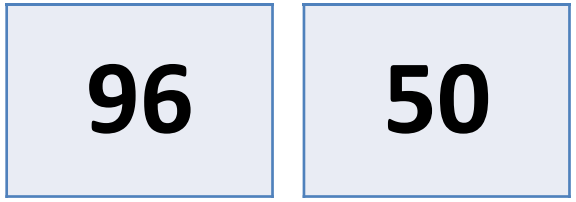
- Sort & Merge





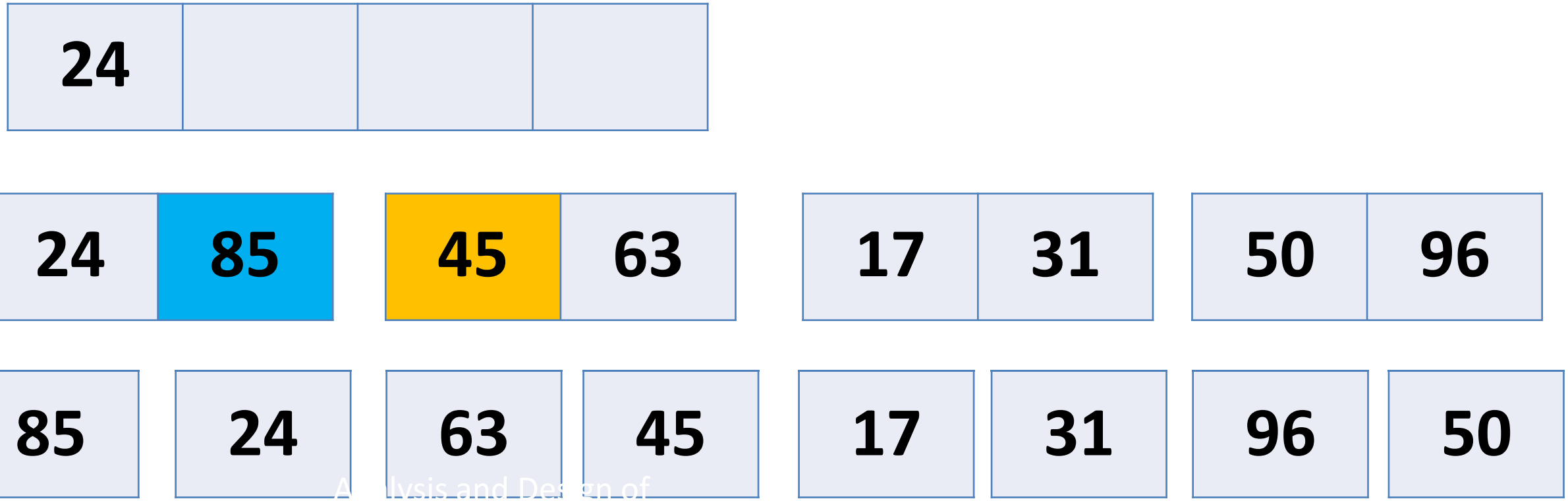






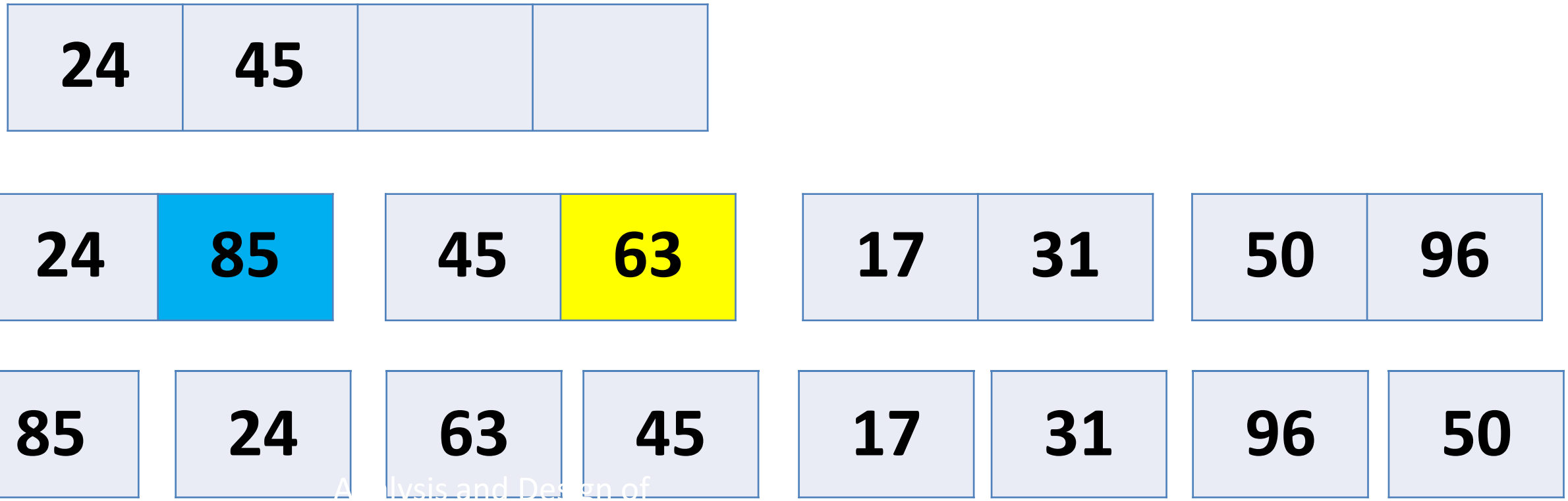
Merge Sort

- Sort & Merge



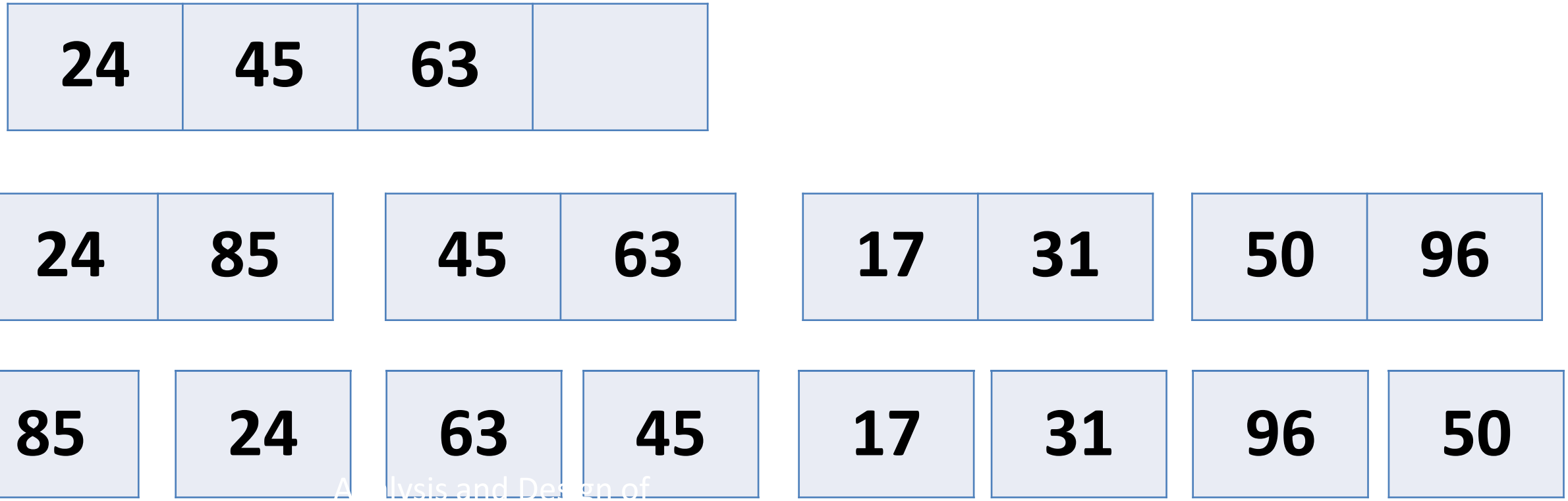
Merge Sort

- Sort & Merge



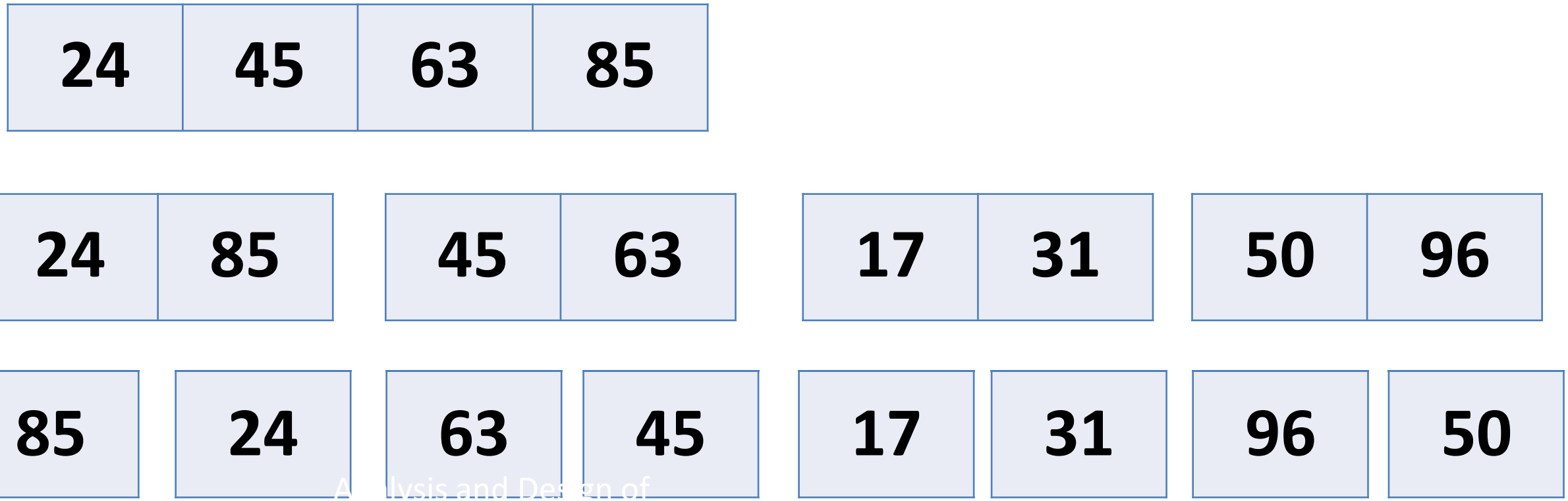
Merge Sort

- Sort & Merge



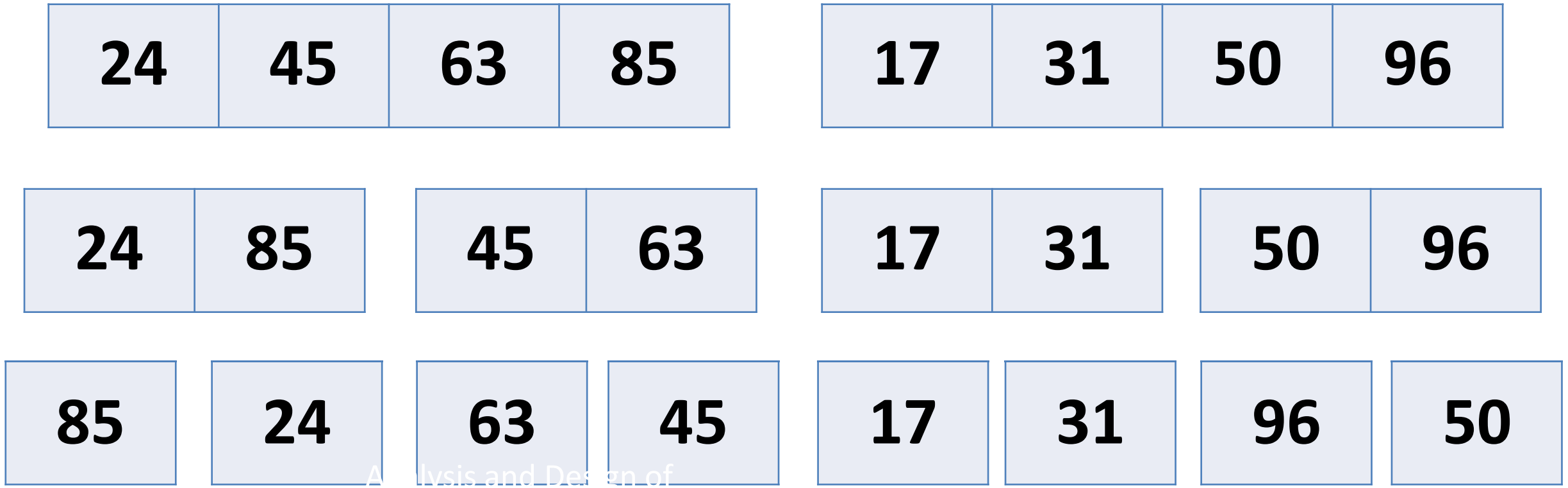
Merge Sort

- Sort & Merge



Merge Sort

- Sort & Merge



Merge Sort

- Sort & Merge

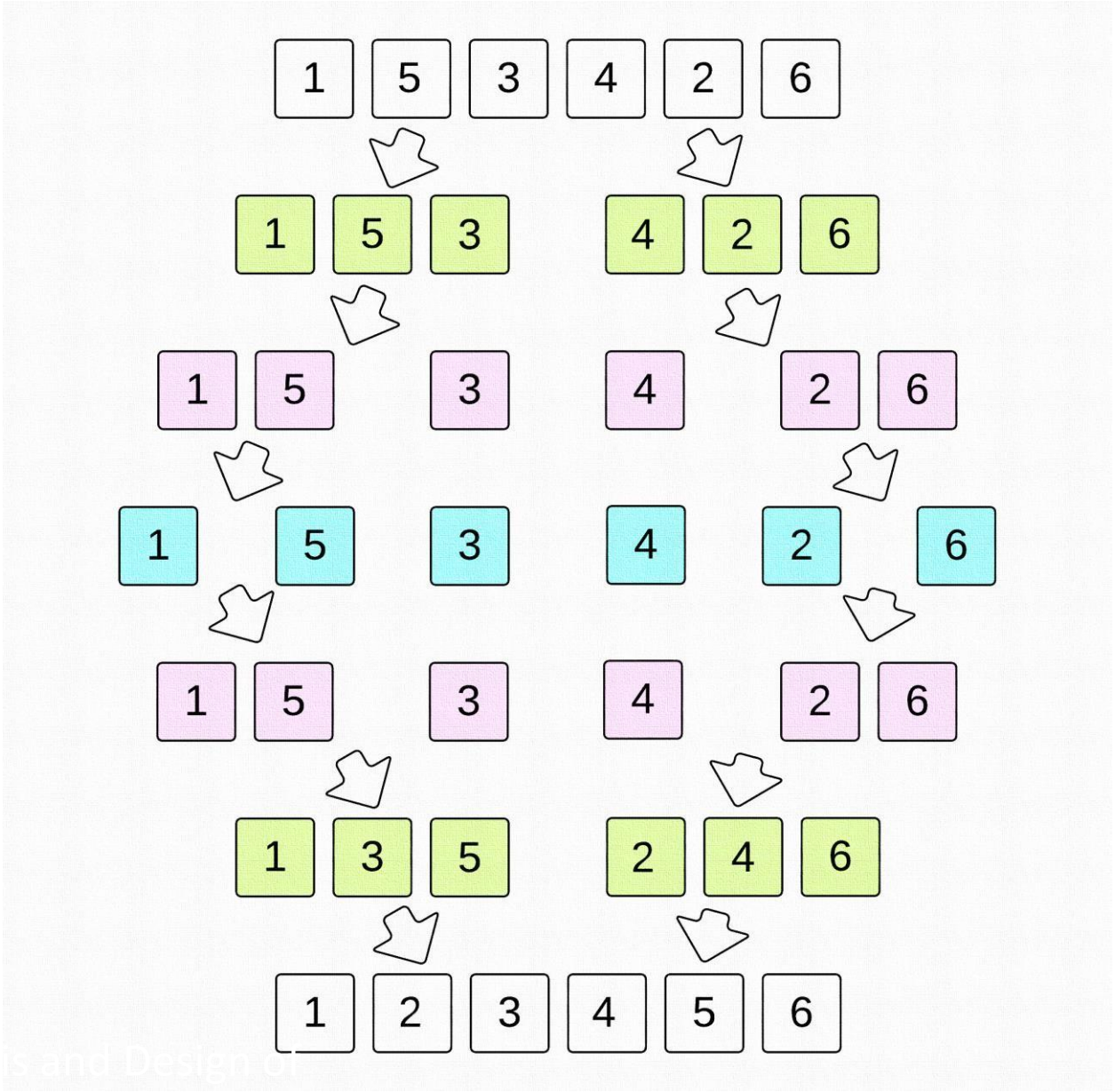
Merge Sort

- **Array is now sorted**

17	24	31	45	50	63	85	96

Merge Sort

- Example 2



Quick Sort

Algorithm

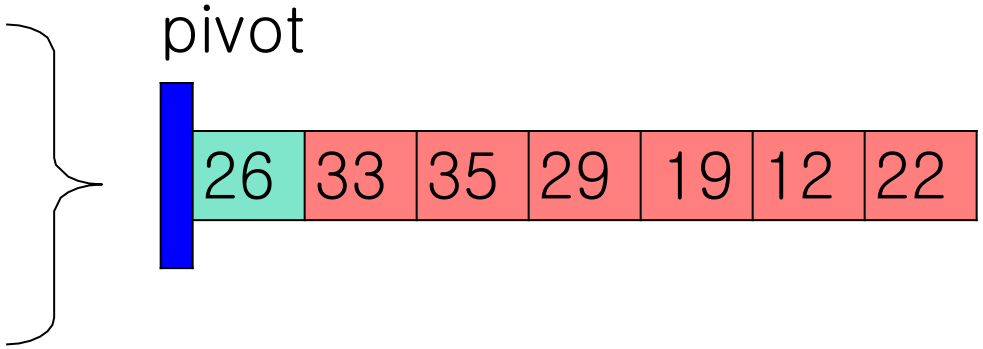
Using pivot algorithm recursively, we end up with smaller possible partitions.
Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Quicksort Step 1

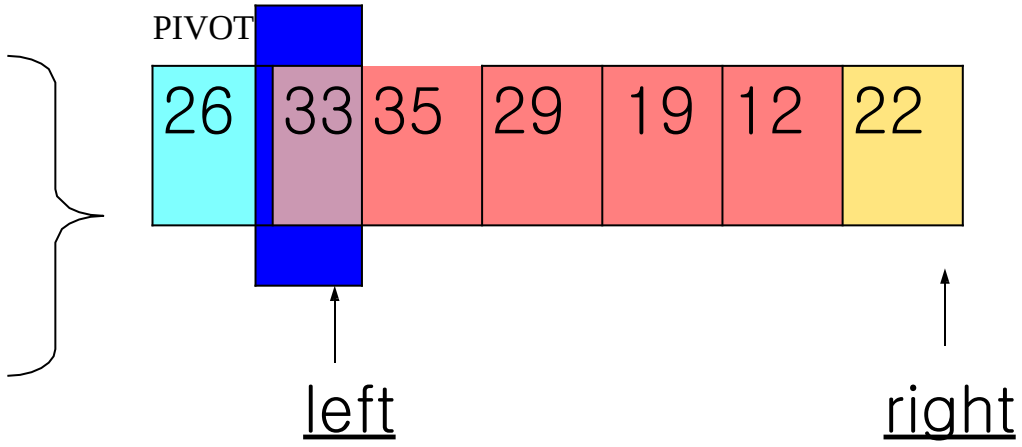
Step 1, select a pivot

Let’s take first element as Pivot



Quicksort Step 2

Step 2, start process of dividing data into LEFT and RIGHT groups:



The LEFT group will have elements less than the pivot.

The RIGHT group will have elements greater that the pivot.

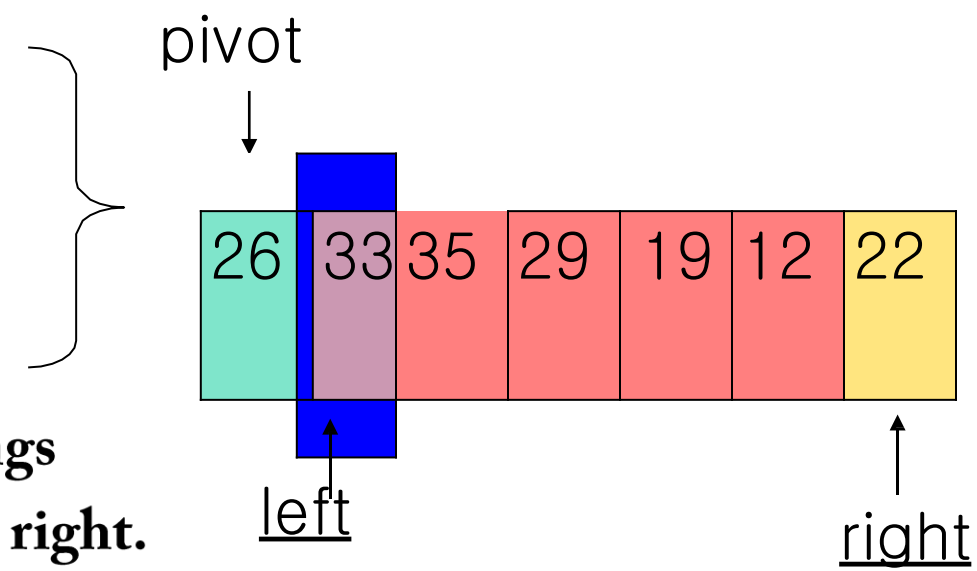
Use markers left and right

Quicksort Step 3

Step 3,
If left element belongs to LEFT group, then increment left index.

If right index element belongs to RIGHT, then decrement right.

Exchange when you find elements that belong to the other group.



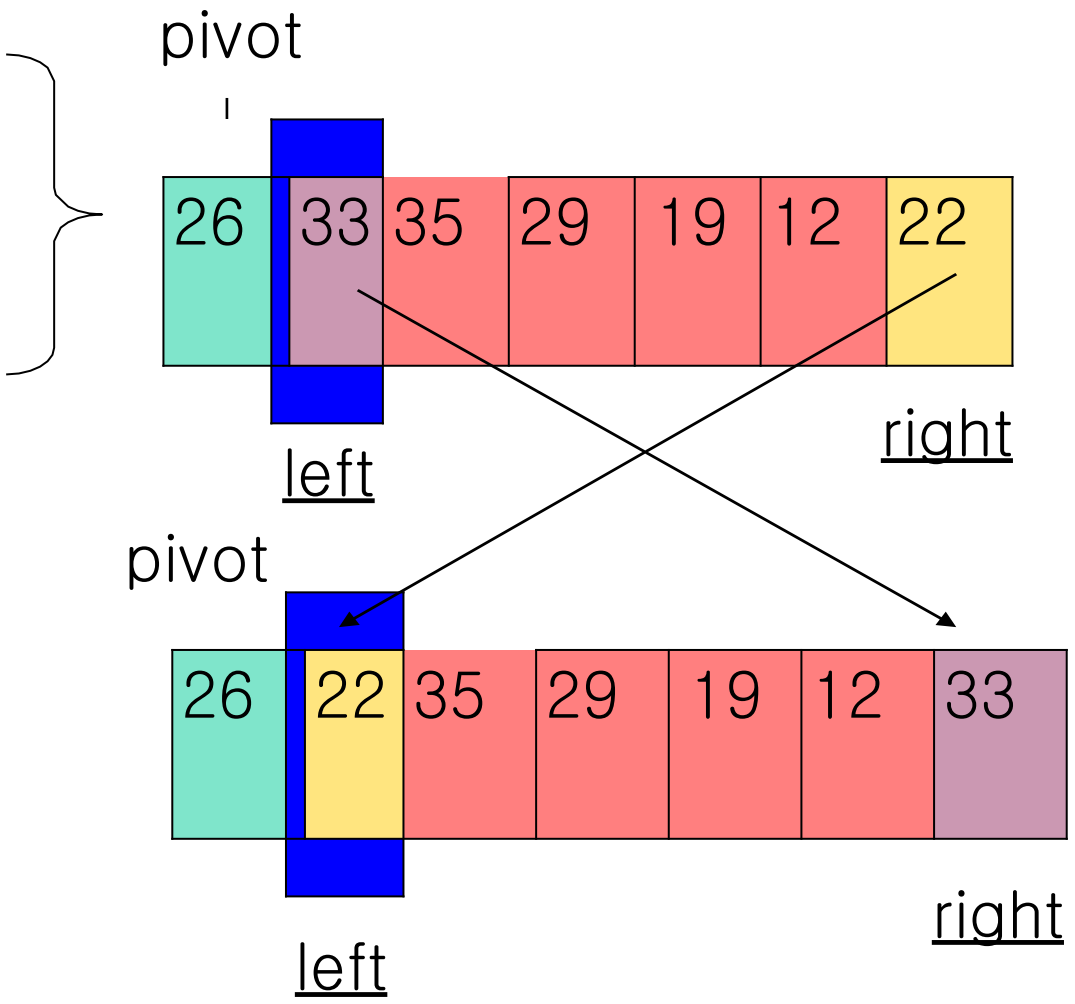
Quicksort Step 4

Step 4:

Element 33 belongs to RIGHT group.

Element 22 belongs to LEFT group.

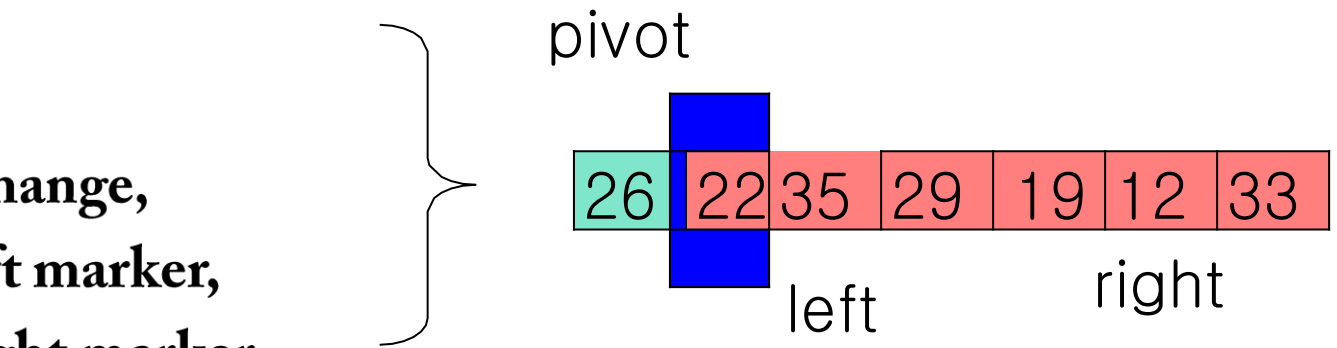
Exchange the two elements.



Quicksort Step 5

Step 5:

After the exchange, increment left marker, decrement right marker.



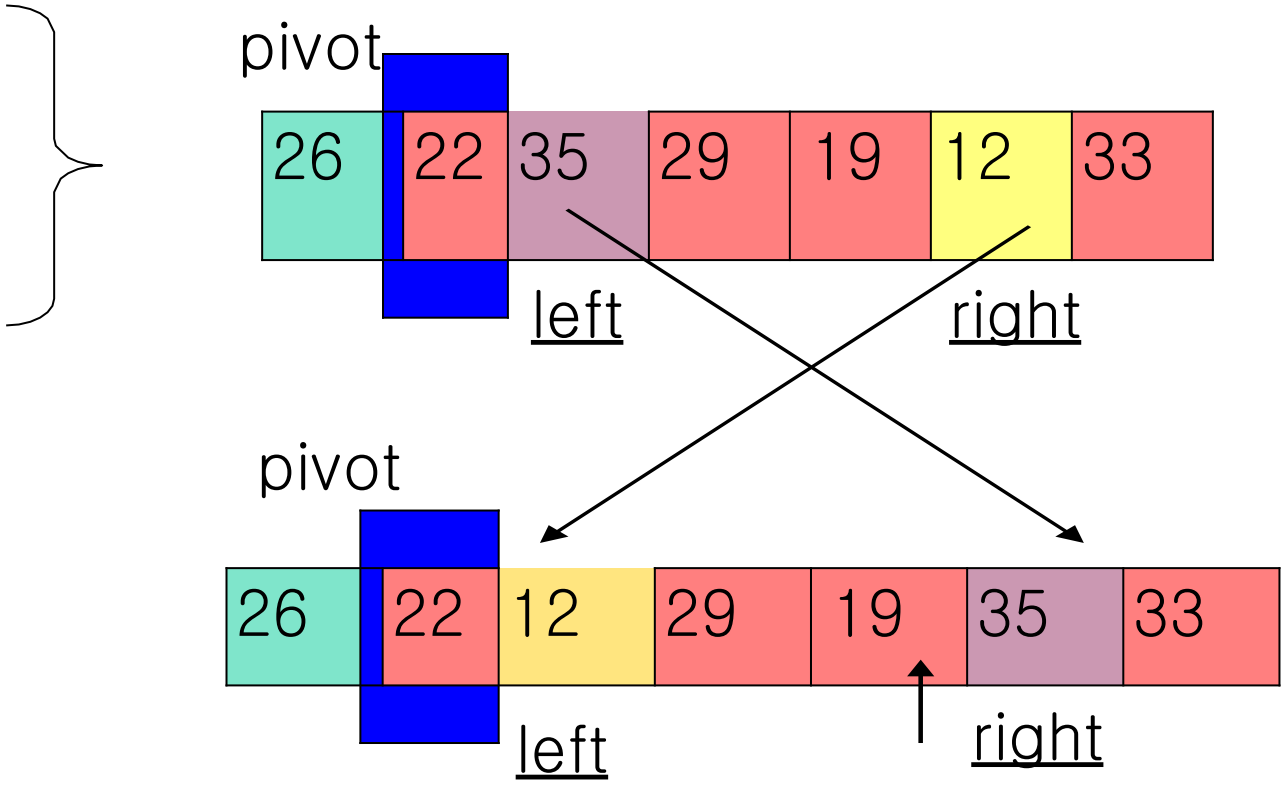
Quicksort Step 6

Step 6:

Element 35 belongs to RIGHT group.

Element 12 belongs to LEFT group.

Exchange, increment left, and decrement right.



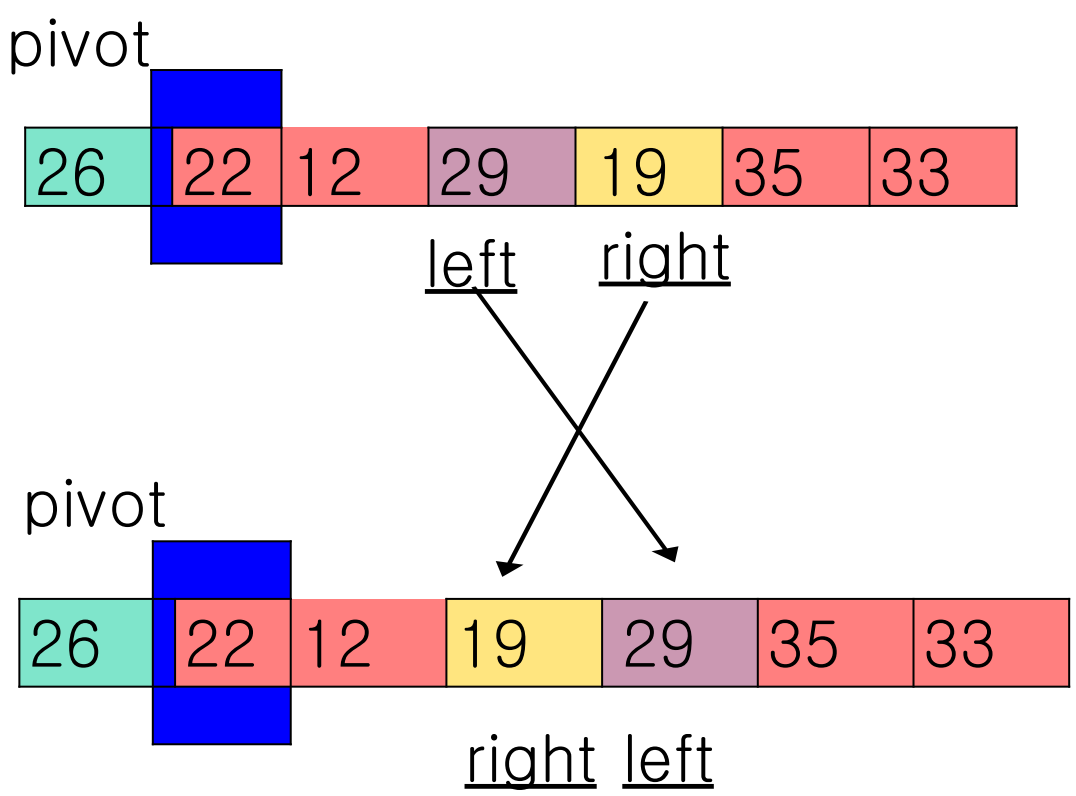
Quicksort Step 7

Step 7:

Element 29 belongs to RIGHT.

Element 19 belongs to LEFT.

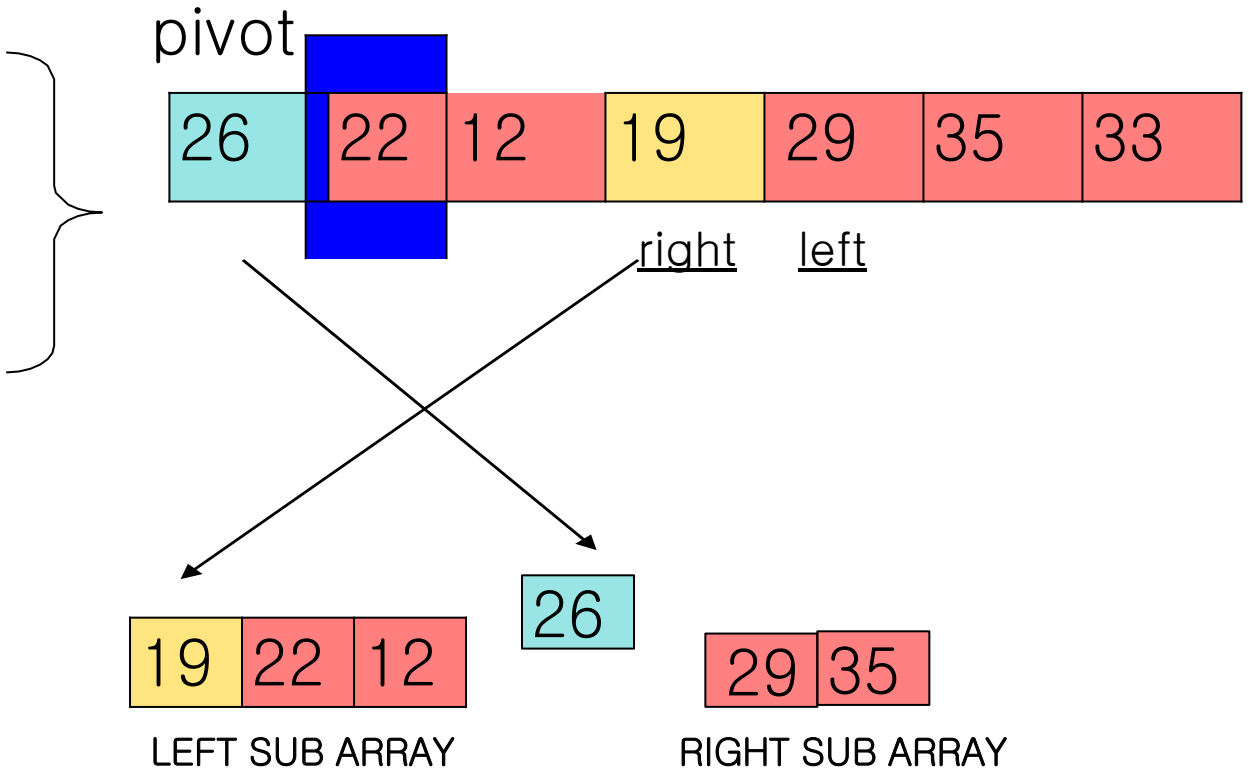
Exchange, increment left, decrement right.



Quicksort Step 8

Step 8:
When the left and right markers pass each other, we are done with the partition task.

Swap the right with pivot.



The Same procedure to be applied on LEFT and RIGHT sub array recursively . Finally all the elements will be sorted

Linear 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.

Linear Search

❑ Algorithm:

- **Step1:** Start from the leftmost element of array and compare one by one element of array with key.
- **Step2**: If key matches with an element, return the index.
- **Step3**: If key doesn’t match with any of elements, return -1.

Linear Search

- Assume the following Array:
- Search for 9

8	12	5	9	2

Linear Search

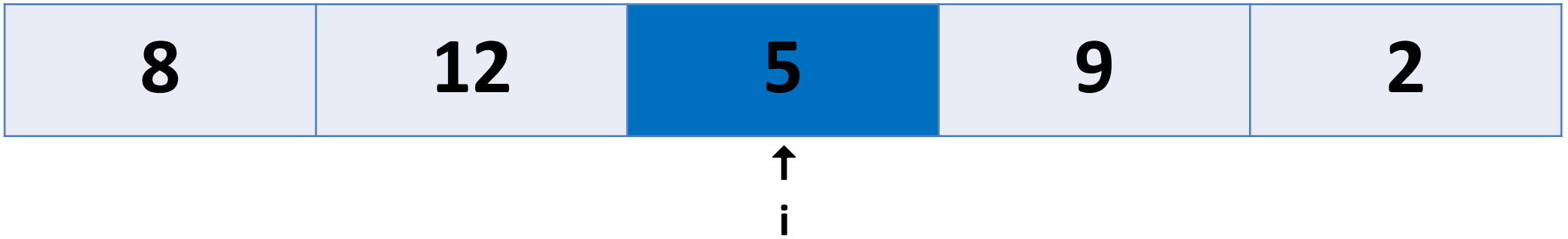
- Compare
- Key = **9**

Linear Search

- Compare
- Key = 9

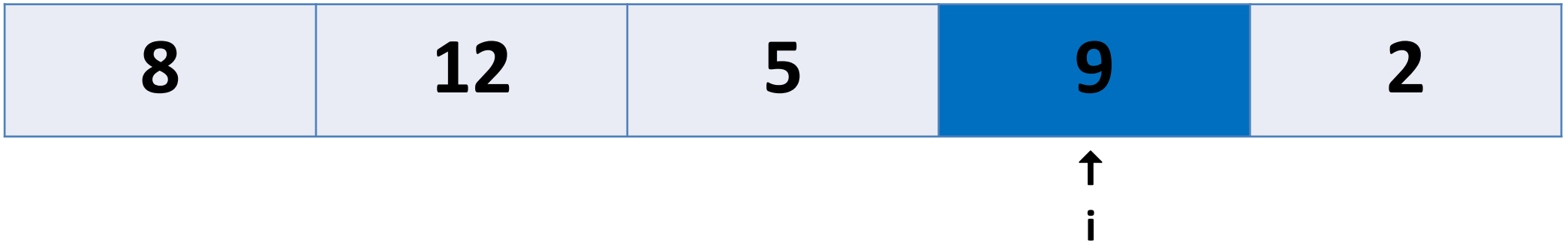
Linear Search

- Compare
- Key = 9



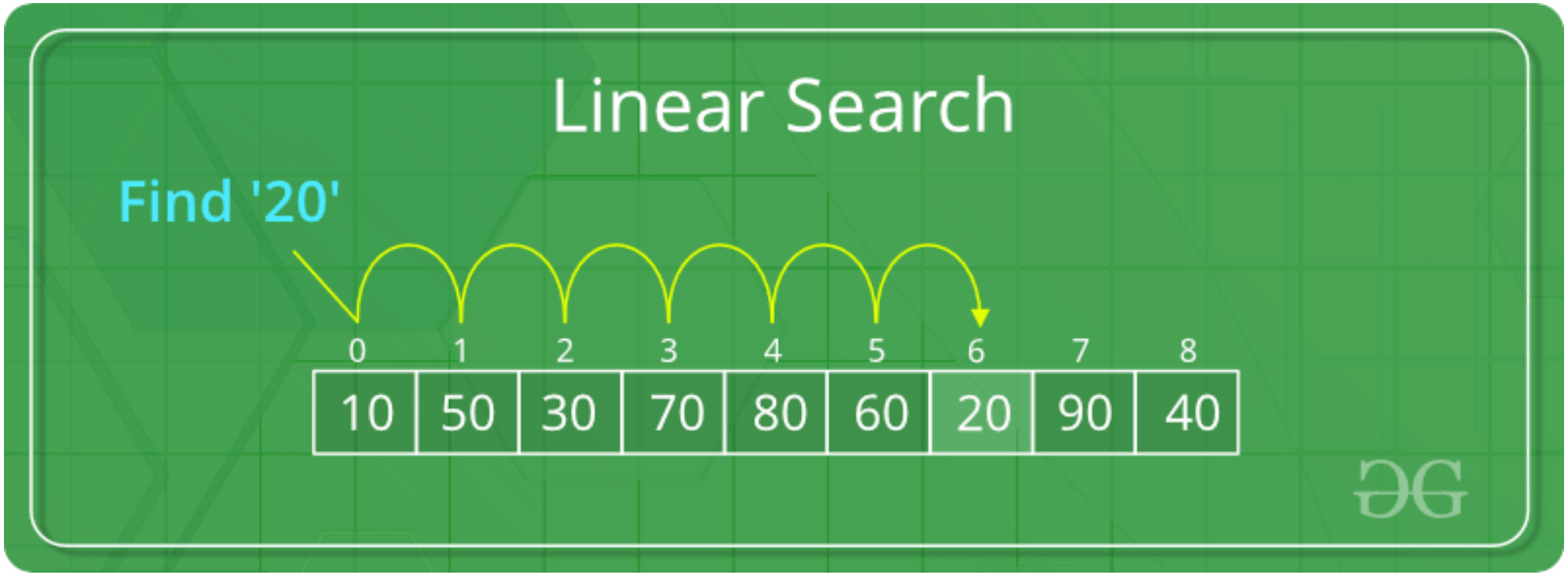
Linear Search

- Compare
- Key = **9**



Linear Search

- Found at index = 3


Linear Search
Find '20'
0
1
2
3
4
5
6
7
8
10
50
30
70
80
60
20
90
40

Binary Search

Binary Search

- **Binary Search** is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques. Binary search use sorted array by repeatedly dividing the search interval in half.

Binary Search

- Algorithm:

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

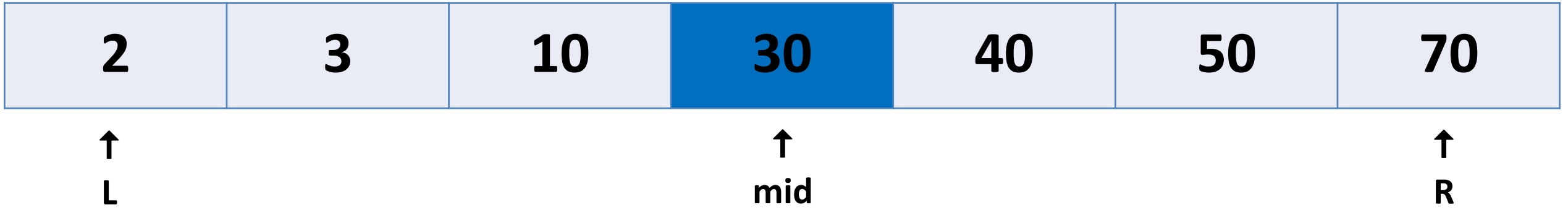
Binary Search

- Assume the following Array:
- Search for 40

2	3	10	30	40	50	70

Binary Search

- Compare
- Key = **40**

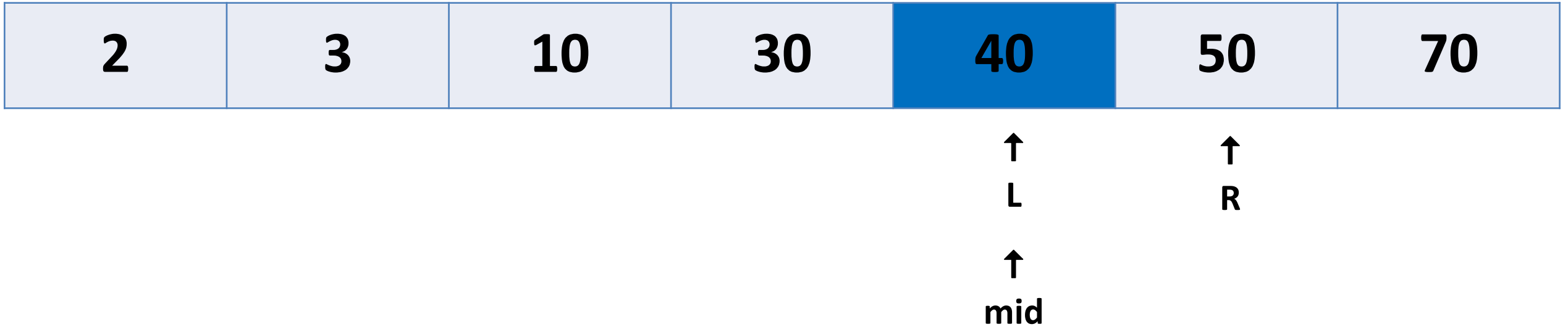


Binary Search

- Compare
- Key = **40**

Binary Search

- Compare
- Key = **40**

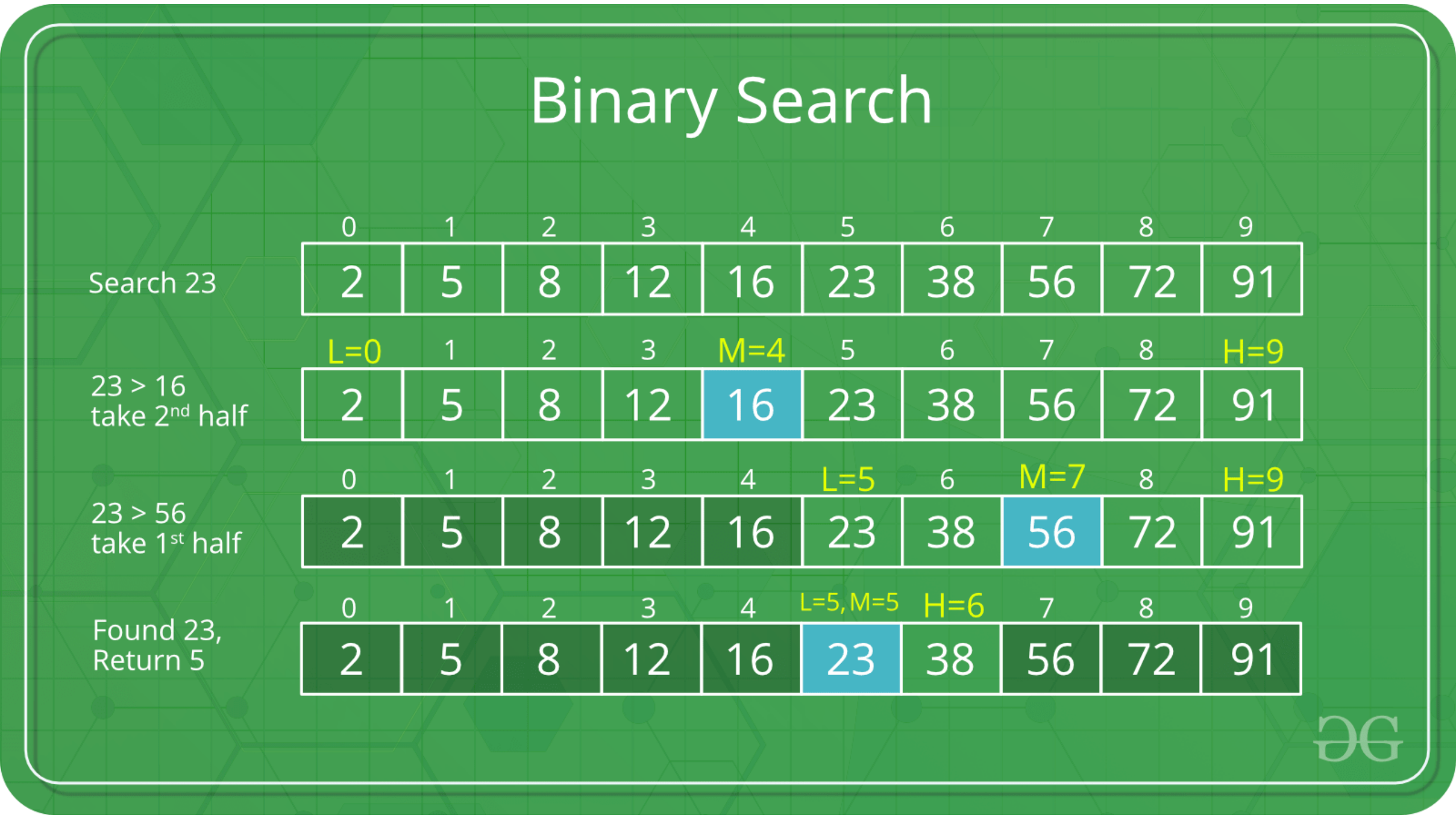


Binary Search

- x=40 , found at index = 4

2	3	10	30	40	50	70

Binary Search



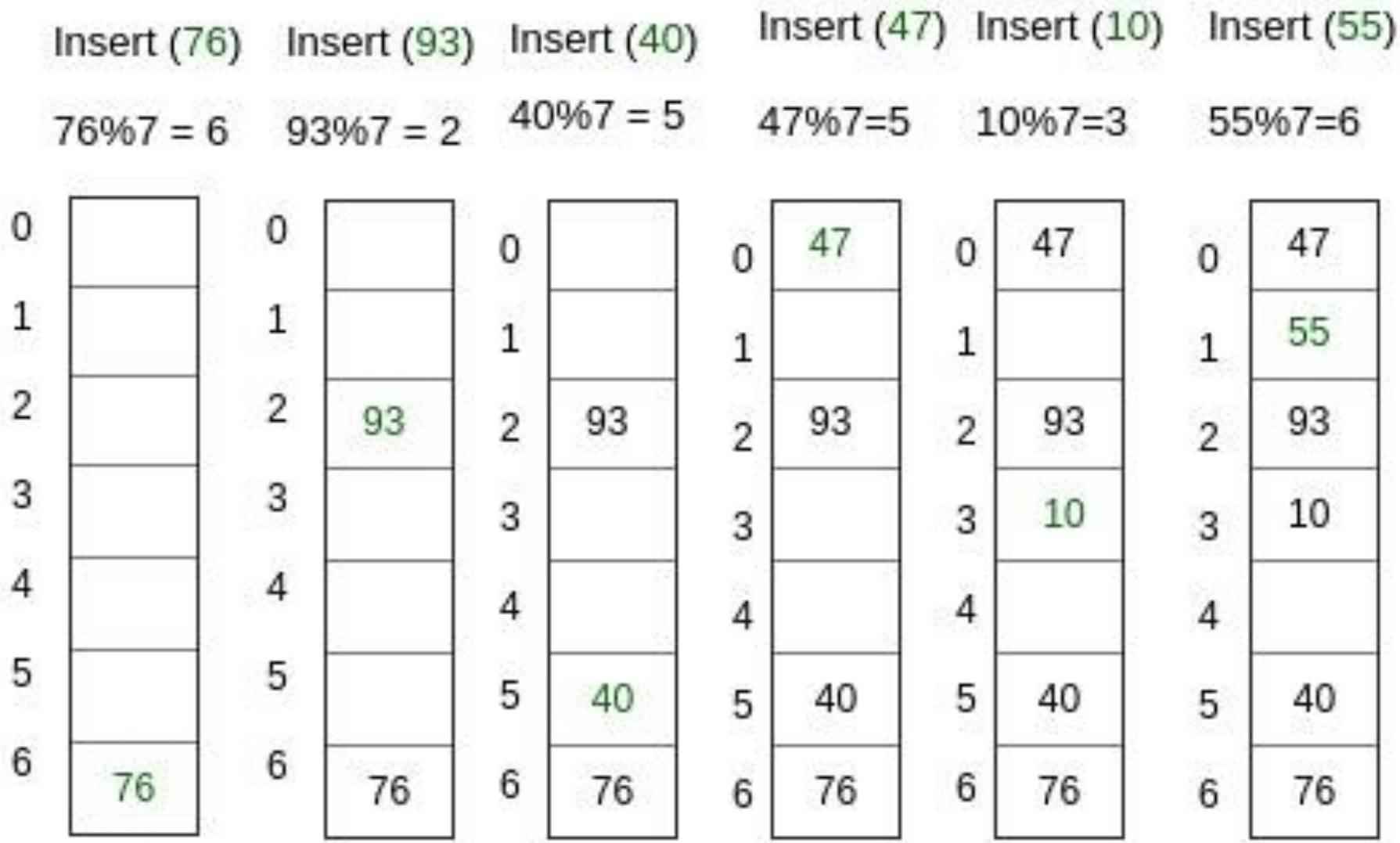
HASHING

Hashing is a Data Structure that uses Hash function. Hash function takes the given value and return a Hash key This Hash Key can be used as an index to store that value. The efficiency of Hash Table depends on the Hash Table size and Hash function.

Hash Table

A Hash Table is a collection of data values which are stored in such a way that finding any element in the Hash Table will be easy. Each block of the hash table is called be Hash bucket/ Hash cell/slot

Linear Probing Example



Hash function

A hash function can be any function that is used to map a given Key value to corresponding index in the Hash Table. The value returned by a hash function is called hash value/hash Index/hash code.

There exist different Hash Functions

Division method

The Key K is divided by some number m and the remainder is used as Hash Index to store K

H(K) = K % m

Folding Method

The Key K is partitioned into different parts with same length as required. The parts are then added together omitting final carry is used as Hash Index

EX - 942781356

P1 - 942 P2 - 781 p3 – 356 [942 + 781 + 356 = 2079, omit the final carry 2]

HashIndex = 079

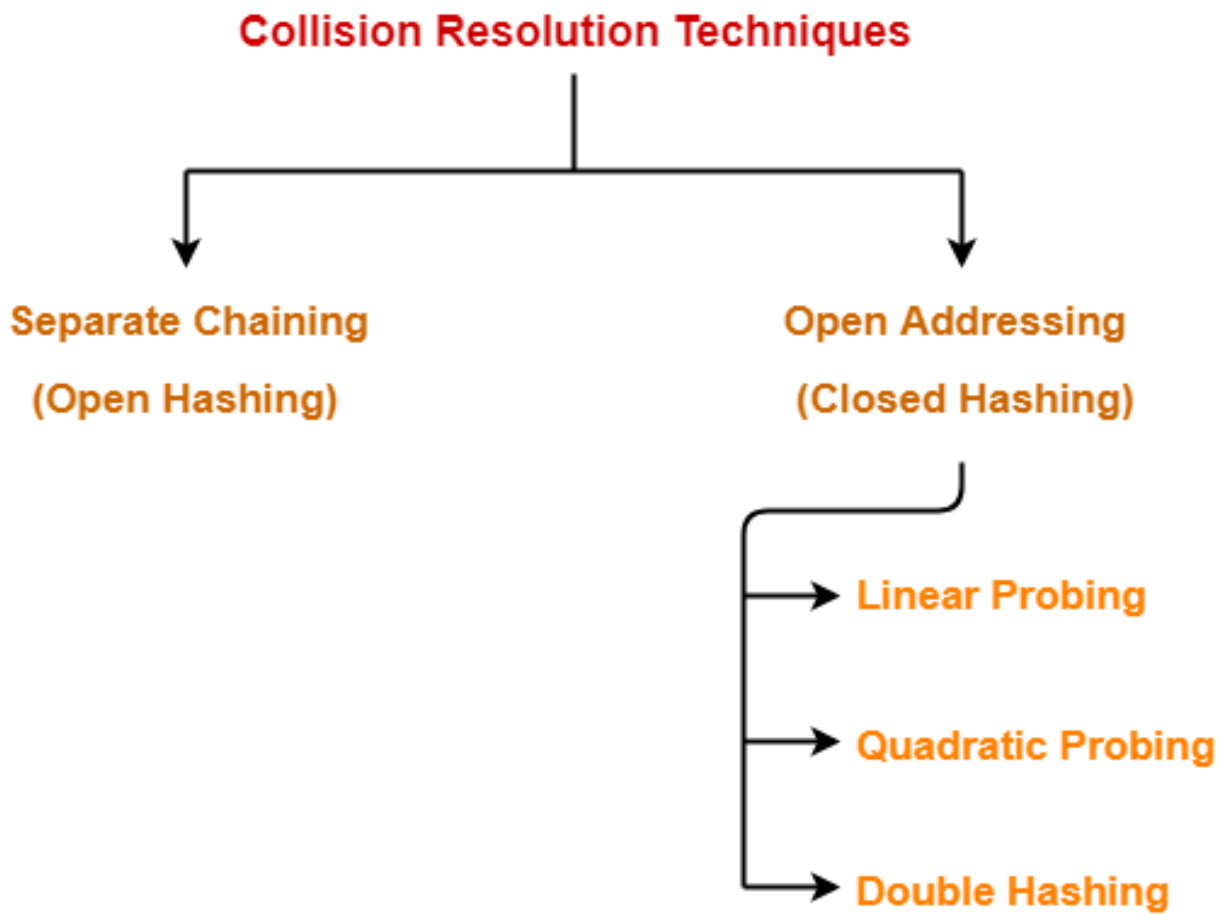
MidSquare method

The Key K is multiplied by itself and the middle part produced from result will be taken as Hash Index

Ex - 50 50*50 = 2,500, 2 50 0

Collision

When the hash value of a key maps to an already occupied bucket of the hash table, it is called as a **Collision**. Whenever collision occurs, we apply Collision Resolution Techniques to handle collision and obtain a new index



Separate Chaining-

To handle the collision,

We will create a linked list at the slot where collision occurs.

The new key is then inserted in to this linked list.

whenever collision occurs again linked list grows like a chain and known to be **separate chaining**.

Open Addressing

In Open Addressing, all elements are stored in the hash table itself. When Collision occurs, we will find out a new Hash index for the key to store

Linear Probing: In linear probing, When a collision occurs, we will check for the next empty cell linearly and will place that value

Quadratic Probing:

In quadratic probing, When collision occurs we probe for i^2th bucket in i^{th} iteration. probing continues until an empty bucket is found.

Double Hashing :

n double hashing, We use another hash function hash2(x) and look for i * hash2(x) bucket in i^{th} iteration. It requires more computation time as two hash functions need to be computed.

Searching through Hashing Technique

Hashing: In Hashing to insert an element we need to find the index and then store value at that index

***Insertion*:**

Apply Hash function to find Index/HashIndex and then store the value, it will take only unit of time to calculate index

***Searching*:**

Apply hash function for **Key** element to be searched, Find the index and verify value Exist in table or not

- If found, Known to be best case and Searching takes 1 unit of time O(1)

- If not found, Search the key following collision resolution technique applied while inserting. if we had to search for all indices, because of collision then here also it will take O(n) Time complexity. It is known to be worst case for Hashing