People's Democratic Republic of Algeria
Ministry of higher education and scientific research
University of Mohamed Boudiaf - M'sila

Faculty of Mathematics and Computer Science
Department of Computer Science

2nd Year License (2L)

Data Structures and Algorithms 3 (DSA3)

Semester : 03

2025/2026

Directed by

Dr. DABBA ALI

Contact us

alidabba@gmail.com

ali.dabba@univ-msila.dz

- In case of problems or difficulties, please contact me or your TD / TP teacher.
- We are at your disposal to help you

CHAPTER 2

Sorting Algorithms

Plan

I. Introduction
II. Complexity of Sorting Algorithms
III. Types of Sorting Algorithms
IV. Bubble Sort
V. Selection Sort
VI. Insertion Sort
VII. Quicksort
VIII. Merge Sort
IX. Analysis of Comparison Sorts

I. Introduction

- One of the **fundamental problems** in computer science is **sorting a list of items.**
- There are a large number of solutions to this issue, known as **sorting algorithms**.
- In this chapter, we try to describe the **different** sorting algorithms that have been considered the most widely used in different applications.

I. Introduction

- Sorting is **the process of arranging or rearranging** the elements of a list in ascending or descending order, which can be numerical, lexicographical or any other order defined by the user.
- The output is a **permutation or reordering of the input**.

II. Complexity of Sorting Algorithms

- The complexity of the sorting algorithm **measures the execution time** as a function of the number $\boldsymbol{n}$ of items to be stored. Each sorting algorithm
- Each sorting algorithm $\boldsymbol{S}$ will be composed of the following operations, where $\boldsymbol{A_1, A_2, A_3 \cdots, A_n}$ contains the items to be sorted and $\boldsymbol{B}$ is an auxiliary location.
  - Comparisons, which test whether $\boldsymbol{A_i < A_j}$ or test whether $\boldsymbol{A_i < B}$.
  - Exchanges that swap the contents of $\boldsymbol{A_i}$ and $\boldsymbol{A_j}$ or $\boldsymbol{A_i}$ and $\boldsymbol{B}$.
  - Assignment that defines $\boldsymbol{B}$: $\boldsymbol{A_i}$ then defines $\boldsymbol{A_j \leftarrow B}$ or $\boldsymbol{A_j \leftarrow A_i}$.

II. Complexity of Sorting Algorithms

- Normally, the complexity function **measures only the number of comparisons,** since the number of other operations is at most a constant factor of the number of comparisons.

III.1. Types of Sorting Algorithms

Sorting algorithms can be classified in two types :

- **Internal Sorting**
- **External sorting**

III.1. Types of Sorting Algorithms

Internal Sorting

- This method uses **only the primary memory** during the sorting process.
- All data items are held in main memory and **no secondary memory is required** for this sorting process.

III.1. Types of Sorting Algorithms

Internal Sorting

- Internal Sorting **is limited**;
- they **can only process a small amount of data** due to memory constraints.
- The time required for reading or writing is **not considered significant** when assessing the performance of internal sorting methods.
- E.g., Bubble Sort, Insertion Sort, Selection Sort, Shell sort, Quick Sort, Radix Sort, Heap Sort, Two-way Merge Sort etc.

III.1. Types of Sorting Algorithms

External sorting

- In contrast to internal sorting, external sorting is used when **a large quantity of data needs** to be sorted.
- This process **uses external memory** such as HDD to store data that cannot be stored in main memory. So, the main memory only contains the data currently being sorted.

III.1. Types of Sorting Algorithms

External sorting

- All external sorts are based on the merge process.
- Read and write access times are a major concern in determining the sorting performance of these methods.
- E.g., Merge Sort, Multiway Merge, Polyphase merge.

IV. Bubble Sort

- The bubble sort is the **oldest and simplest** sort in use.
- Unfortunately, it's also the **slowest**.
- The bubble sort works by comparing each item in the list with the item next to it, and **swapping** them if required.
- The algorithm **repeats this process until** it has gone through the entire list **without swapping any items** (in other words, all items are in the correct order).

IV. Bubble Sort

- A C++ implementation of the standard **bubble sort algorithm** for a vector of integers is as follows:

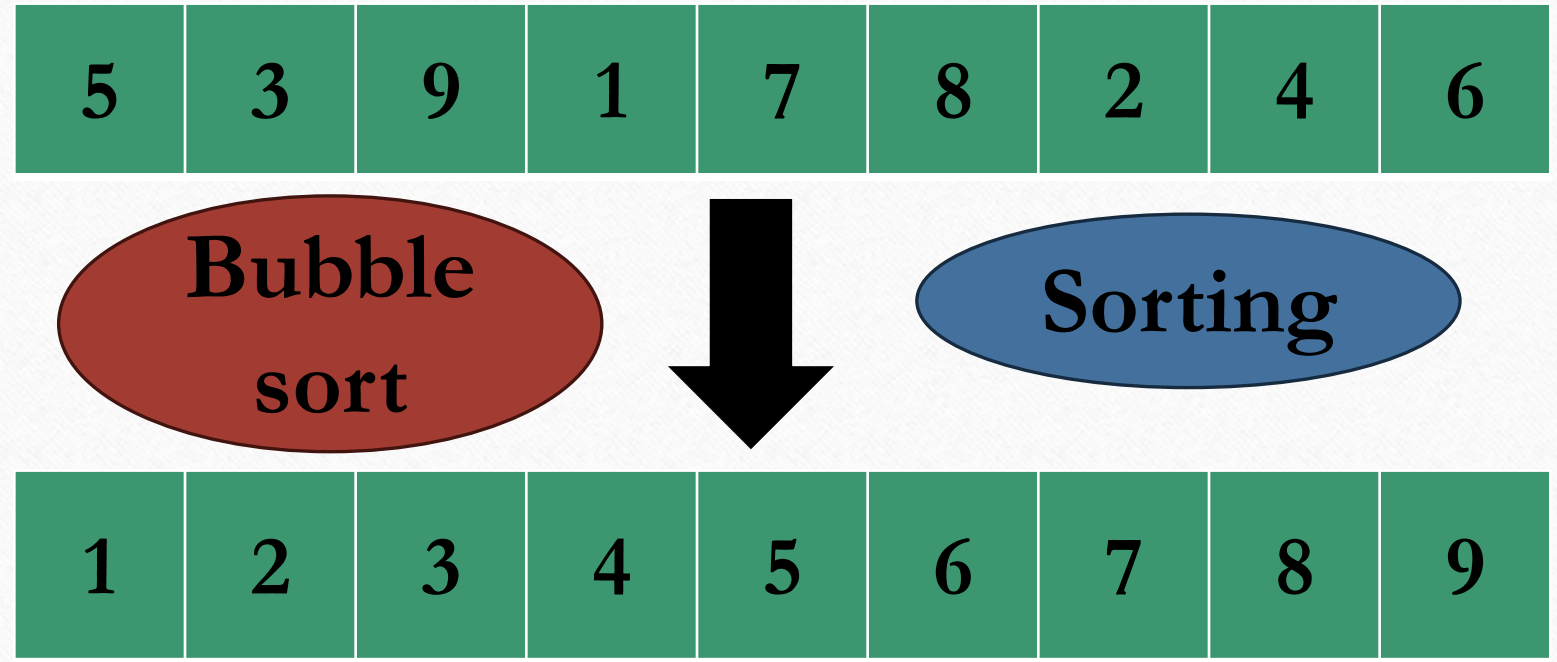
```
void bubble_sort (int Vec[], int n) {    // Bubble Sort
   for (int i=0; i<n-1; i++)             // Bubble up ith record
      for (int j=0; j<n-i-1; j++)
         if (Vec[j]> Vec[j+1])
            swap(Vec[j], Vec[j+1]);
}
```

IV. Bubble Sort

Example:

- Let's look at how bubble sort works using the following unsorted array: **[5, 3, 9, 1, 7, 8, 2, 4, 6]**.
- The bubble sort algorithm looks at two adjacent elements at a time, swapping them if they are out of order.
- At the end of each pass, the largest element "bubbles" to the end of the vector, as shown below:

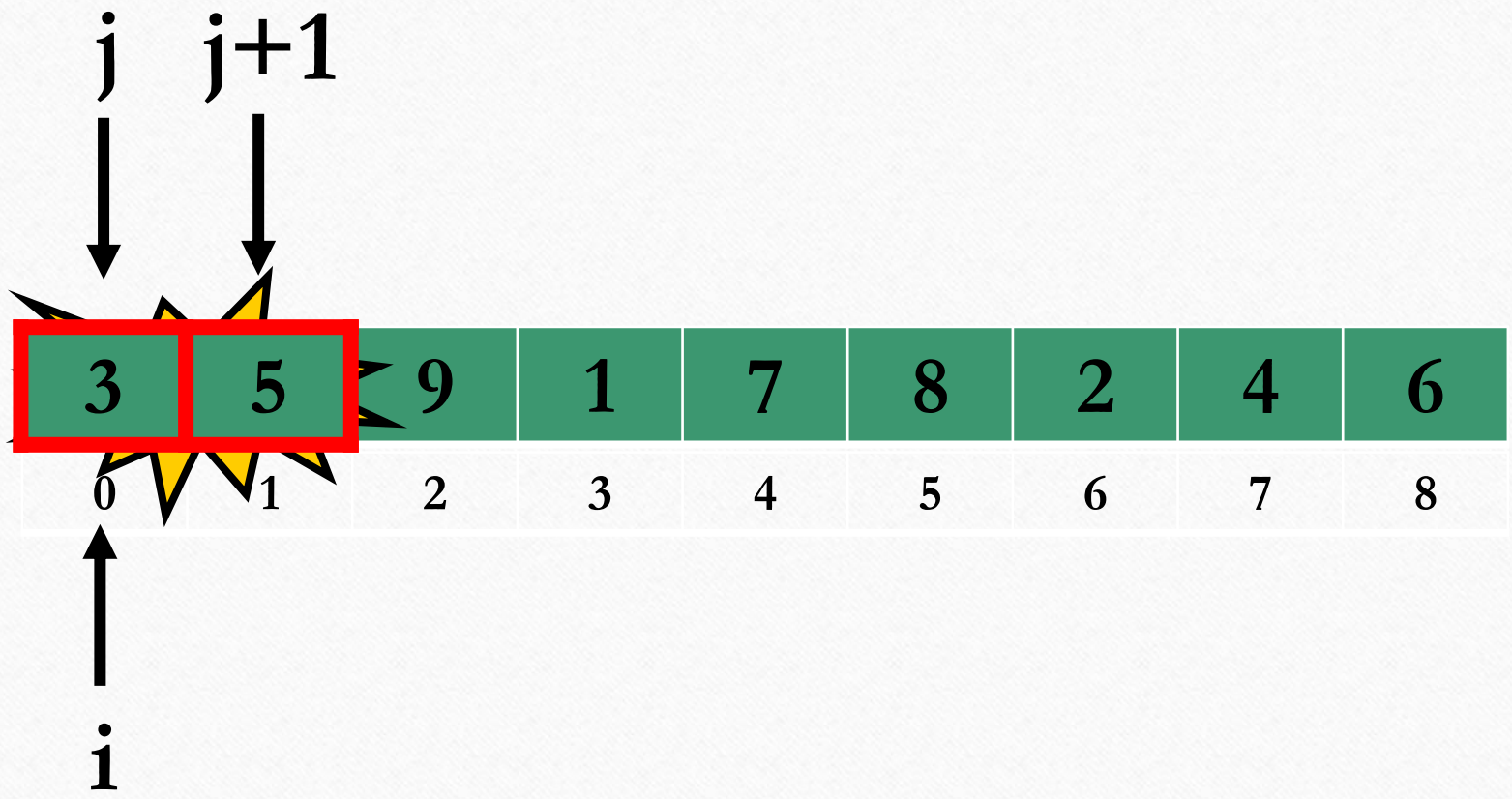
IV. Bubble Sort



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

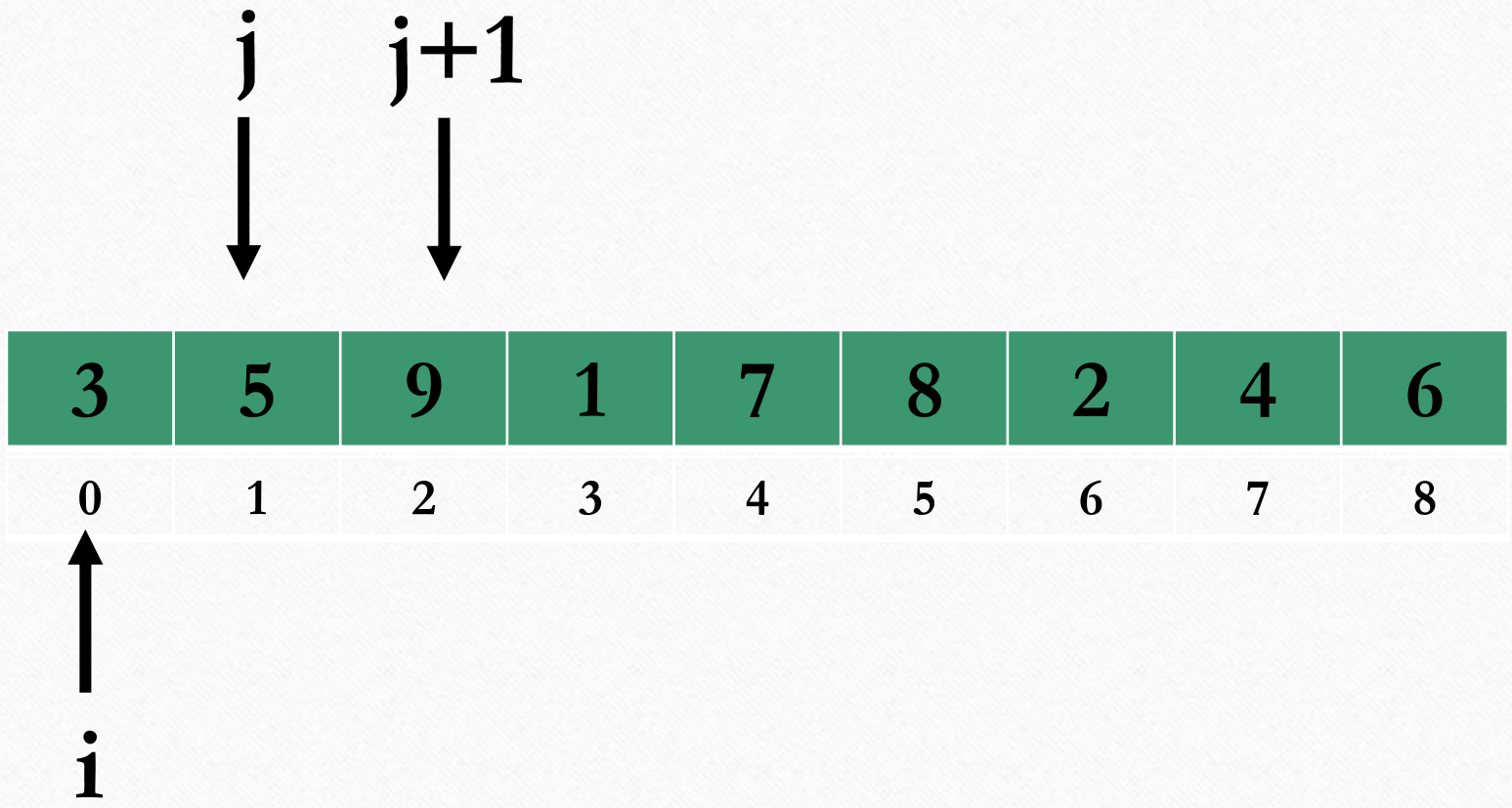
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

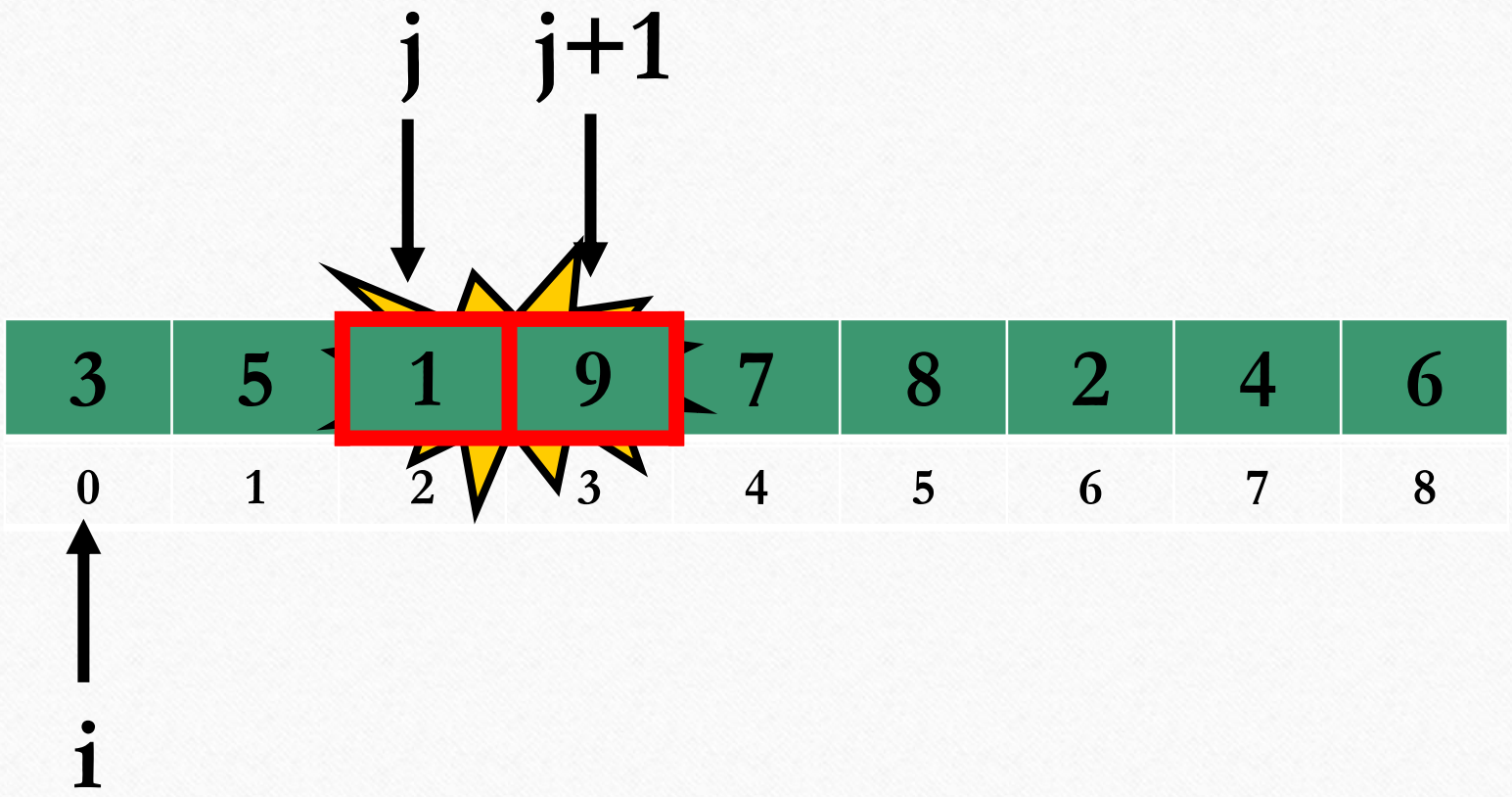
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

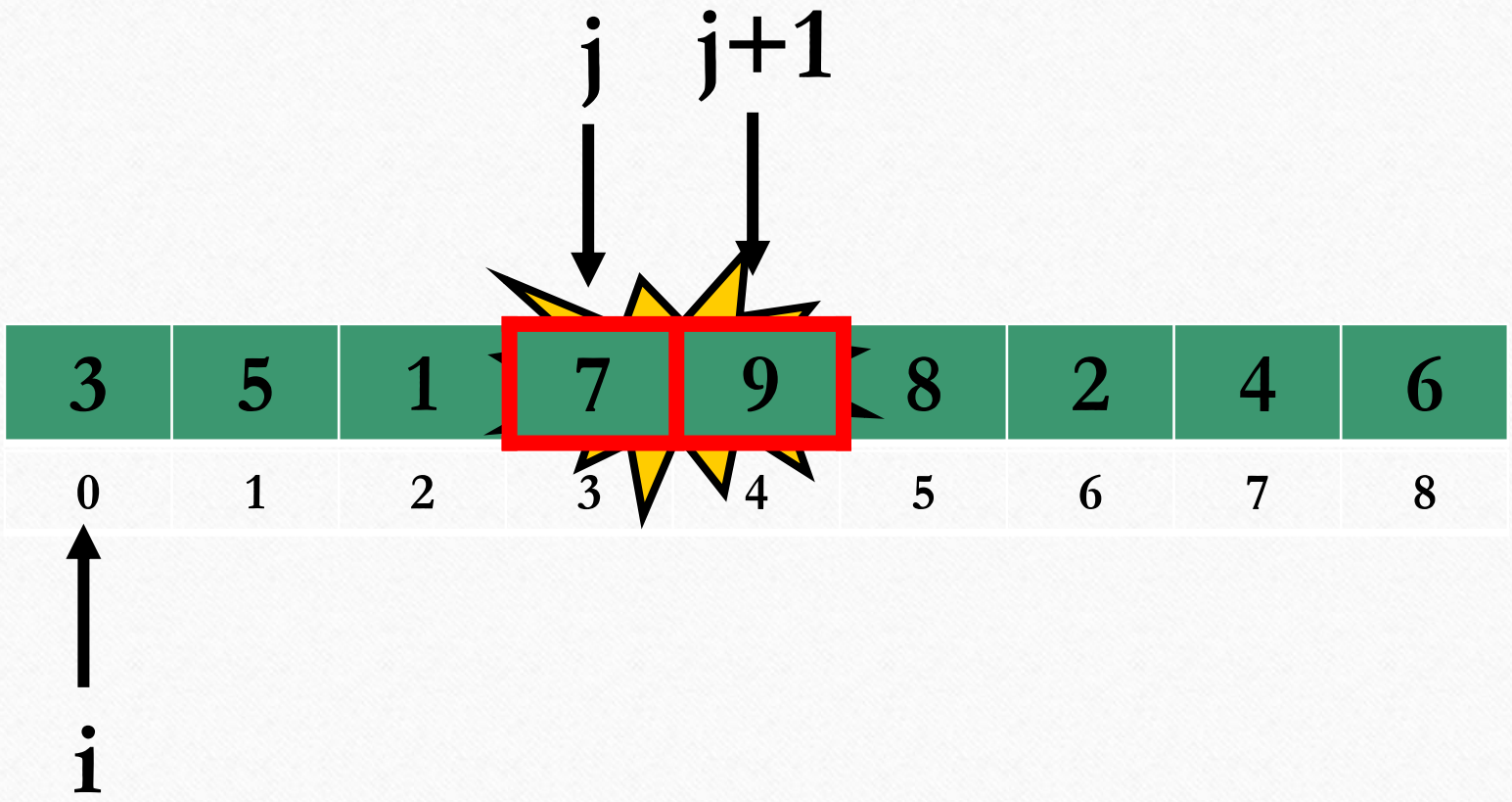
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

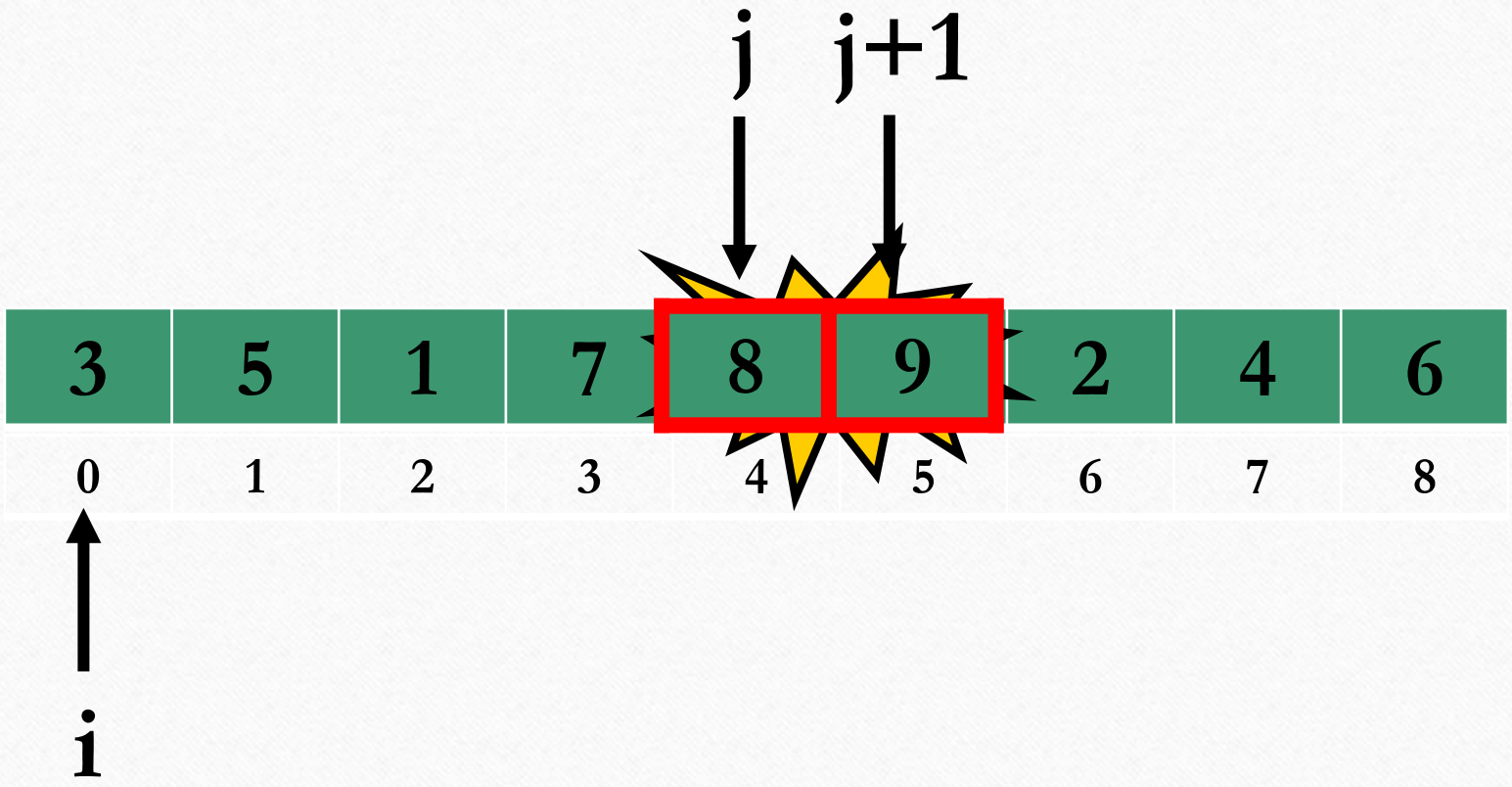
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

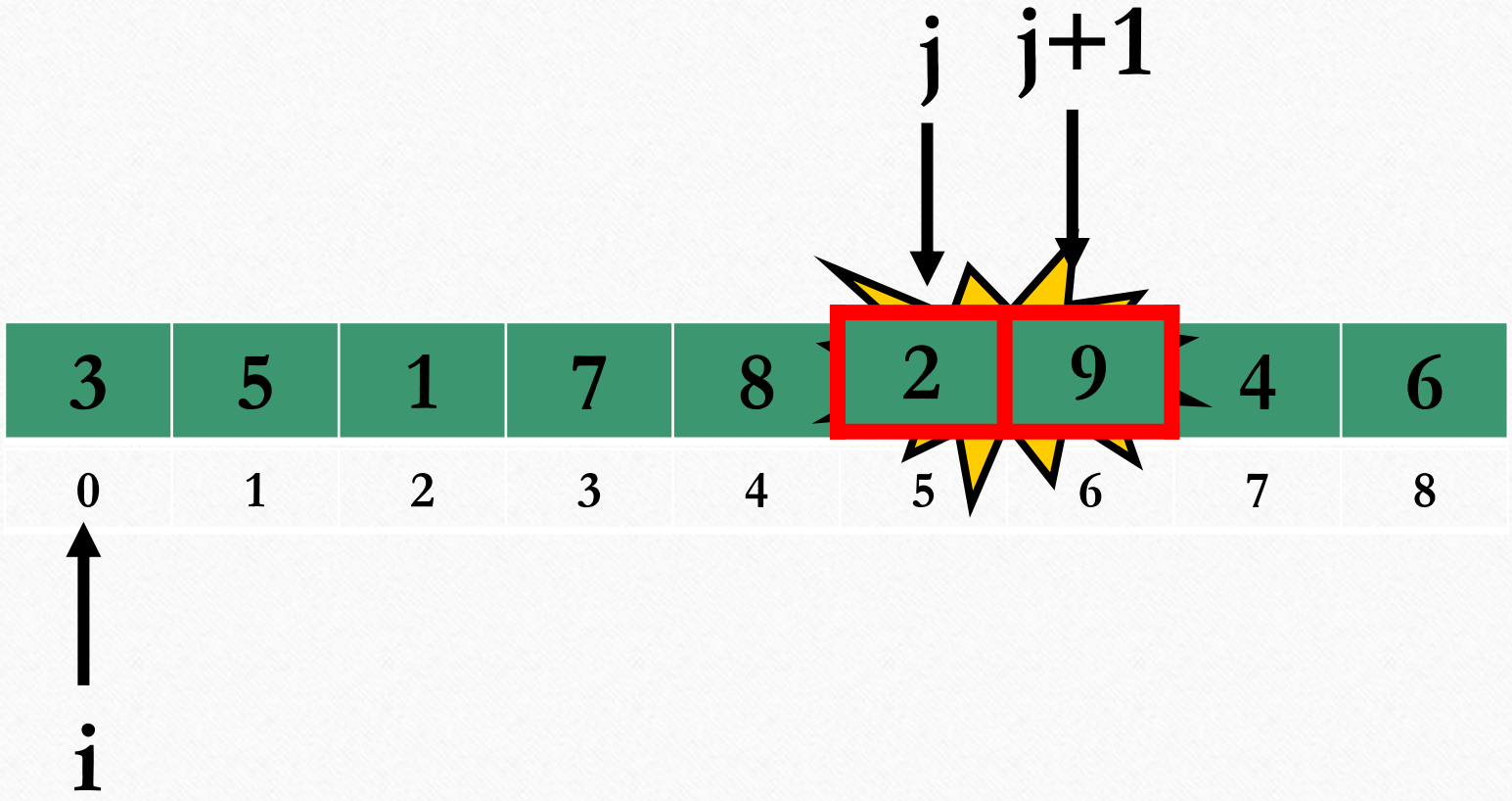
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

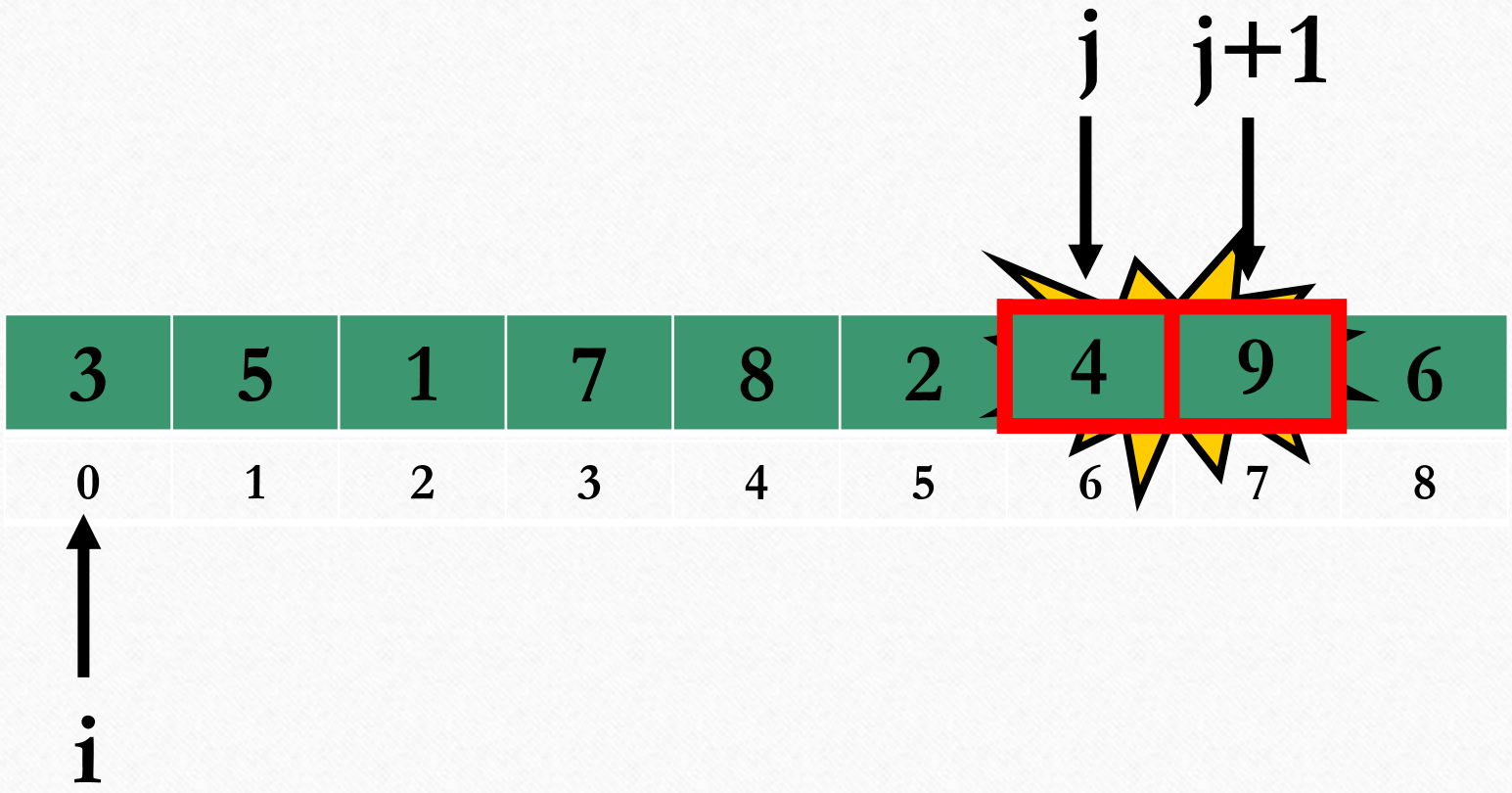
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

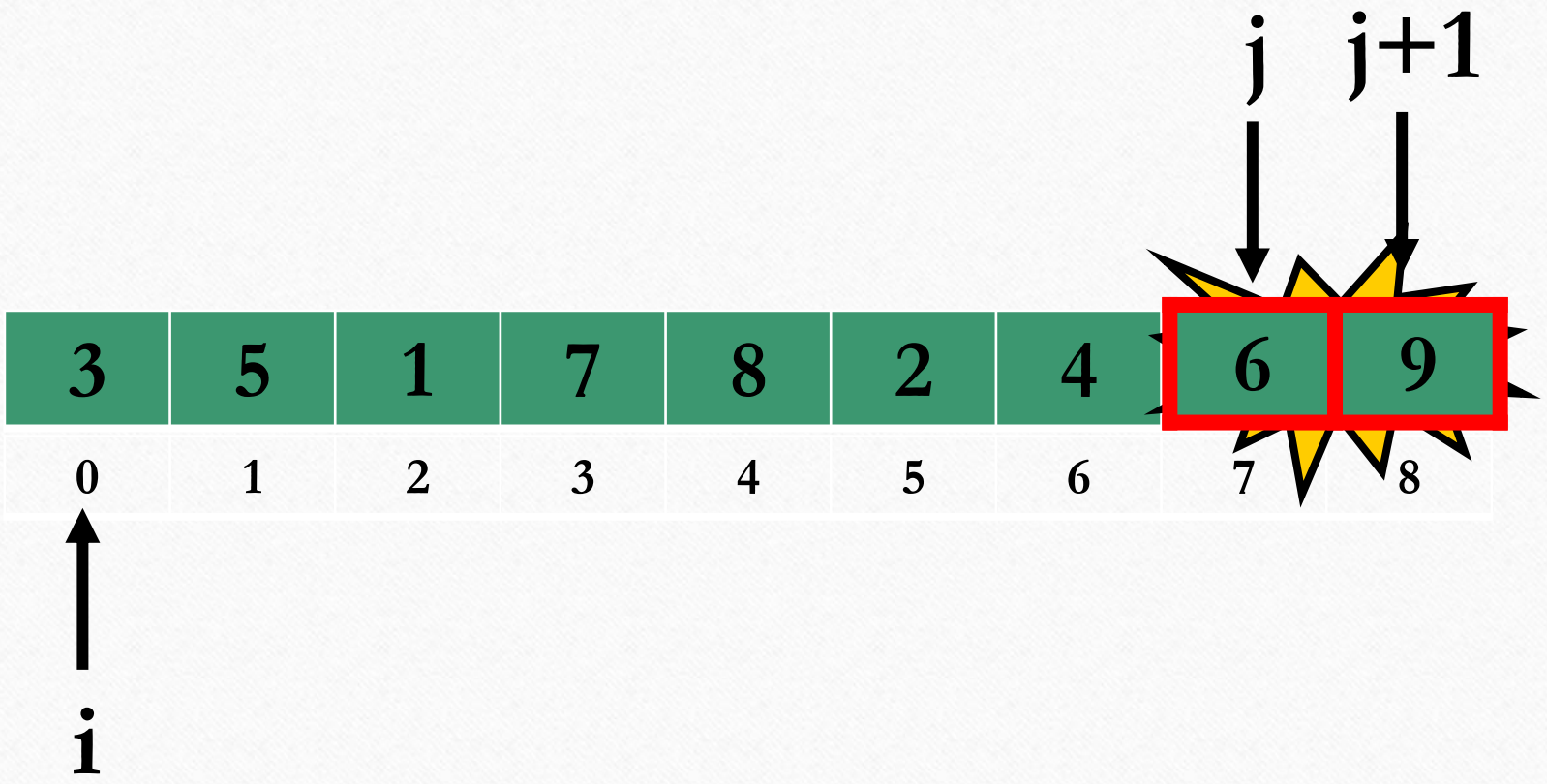
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

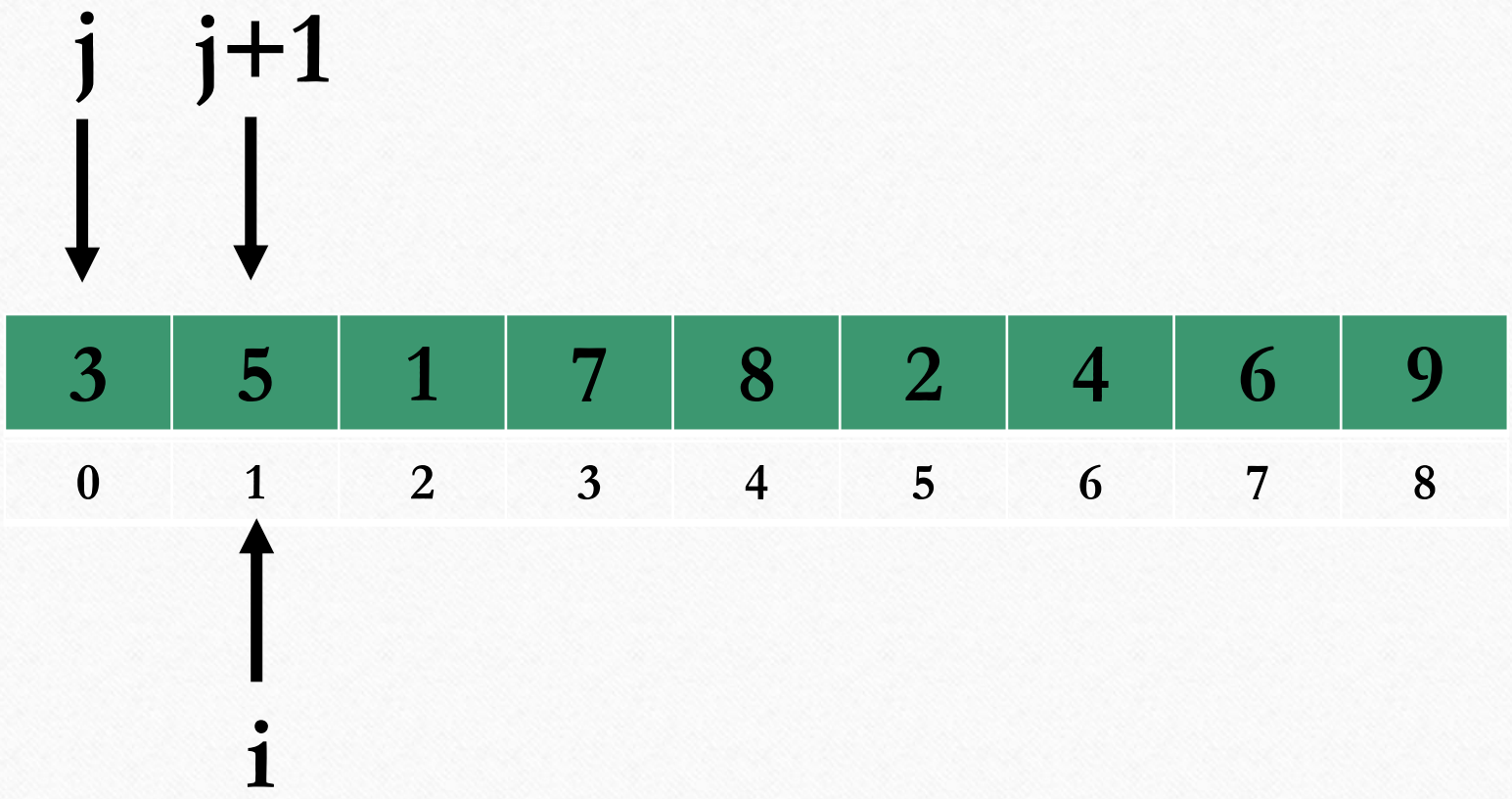
First pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

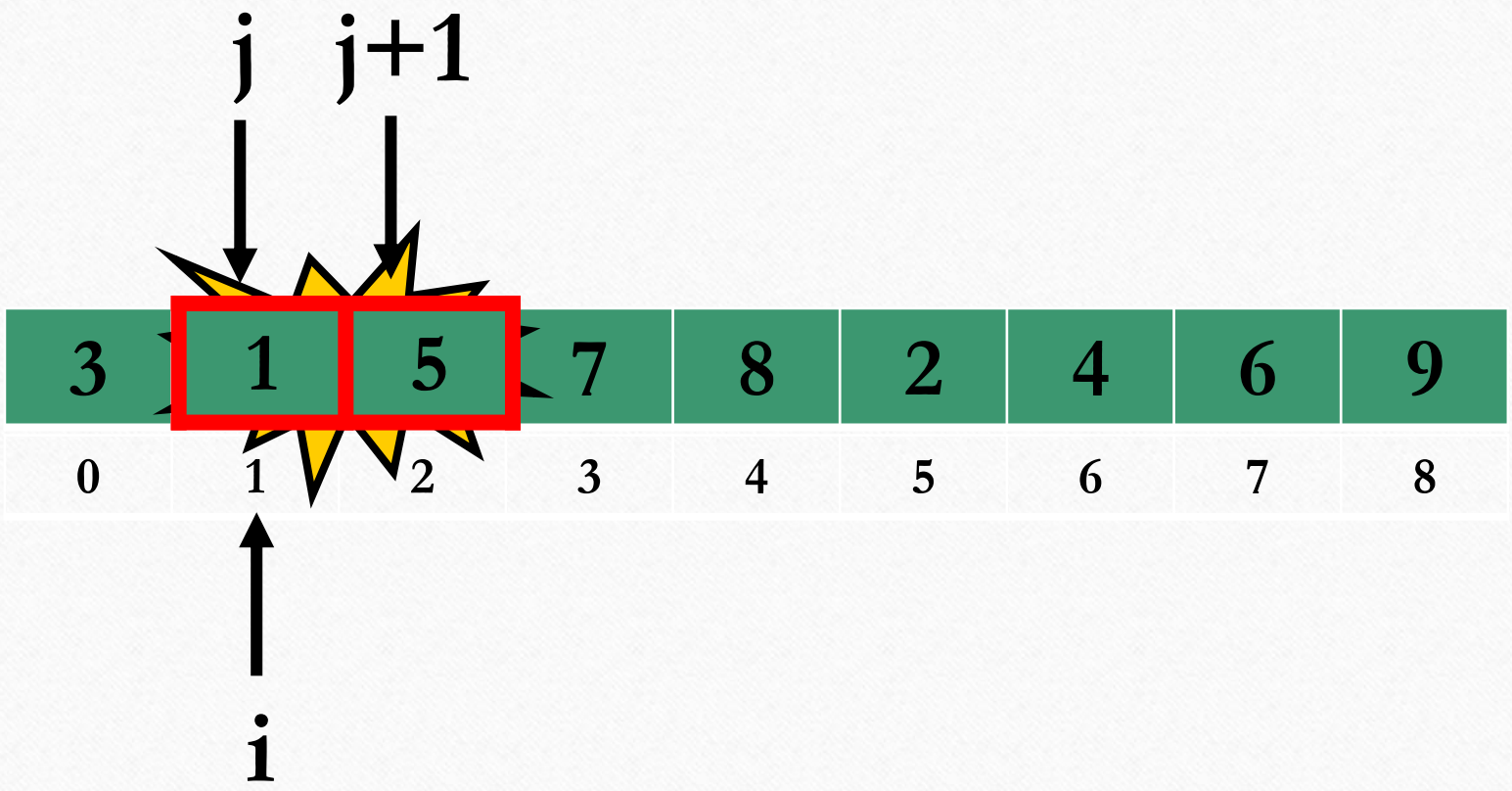
Second pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

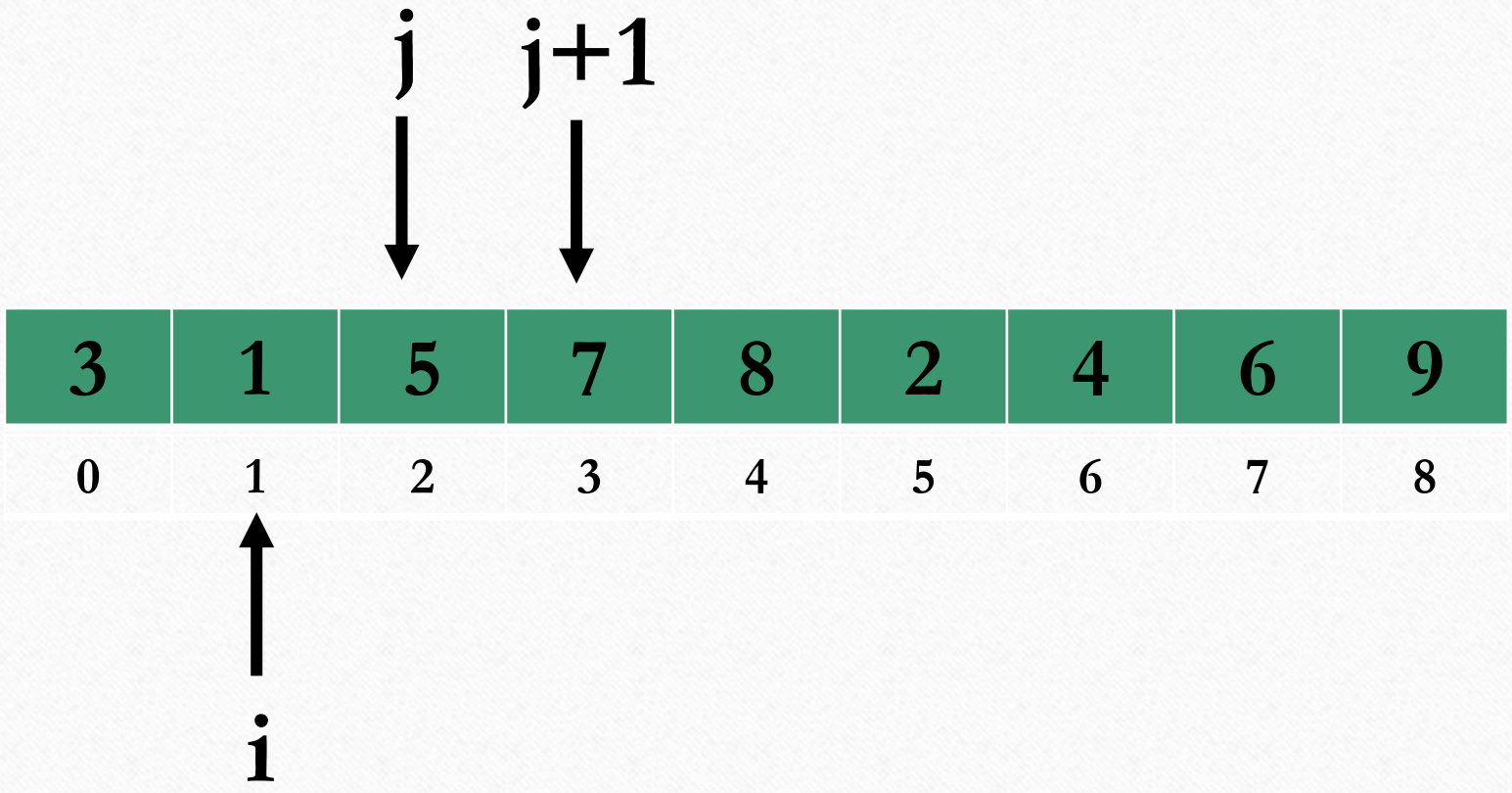
Second pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

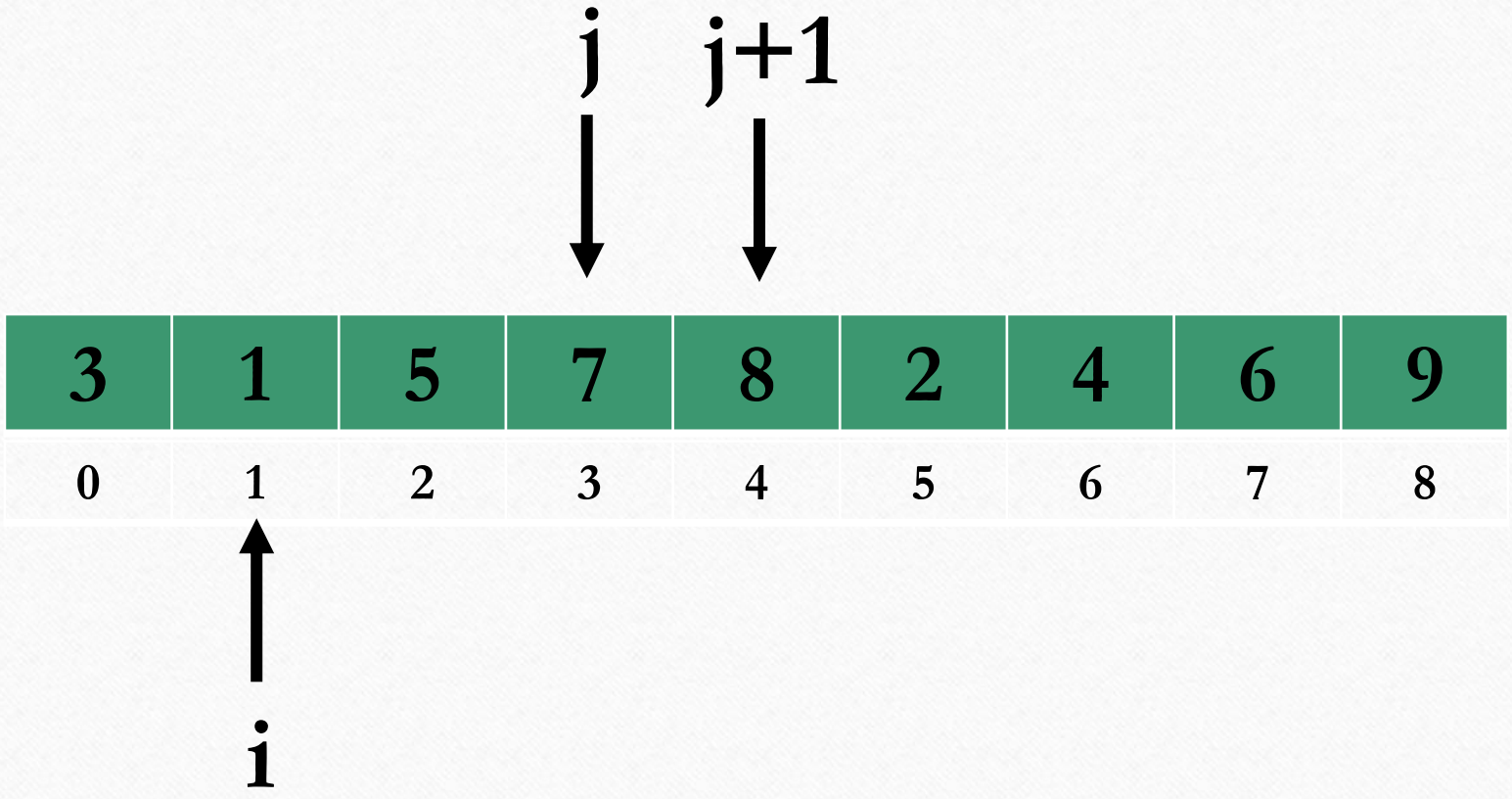
Second pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

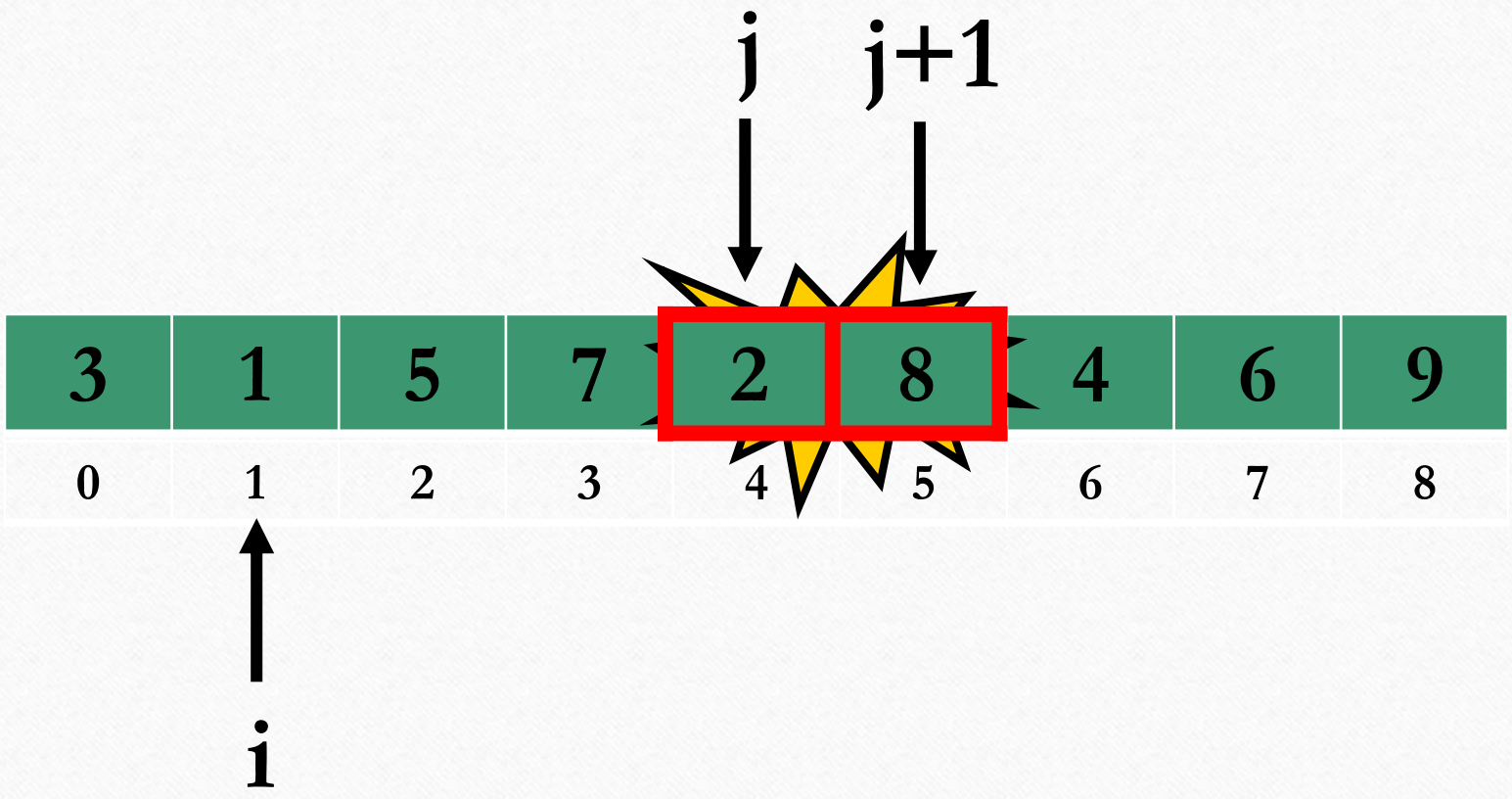
Second pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

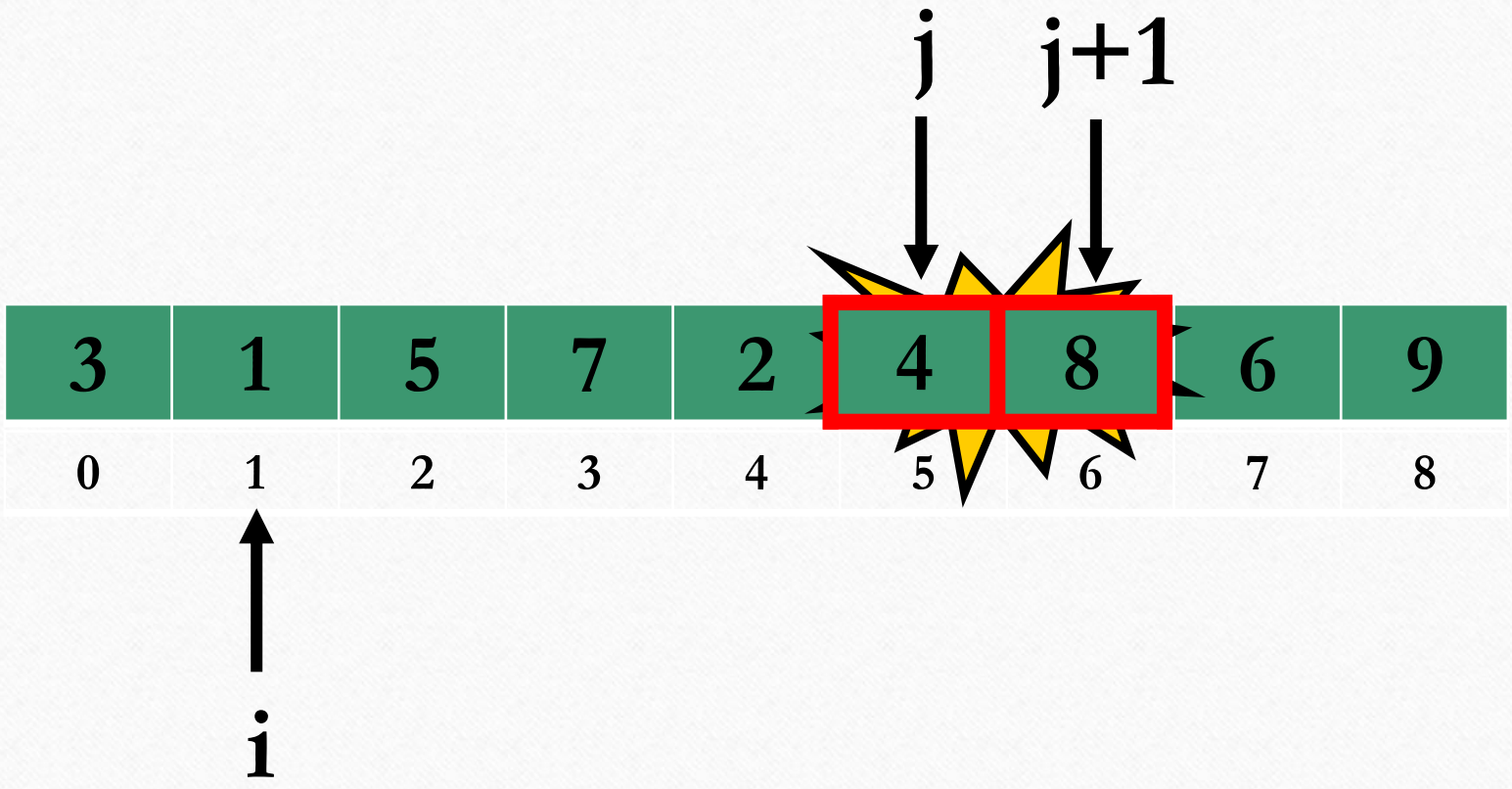
Second pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

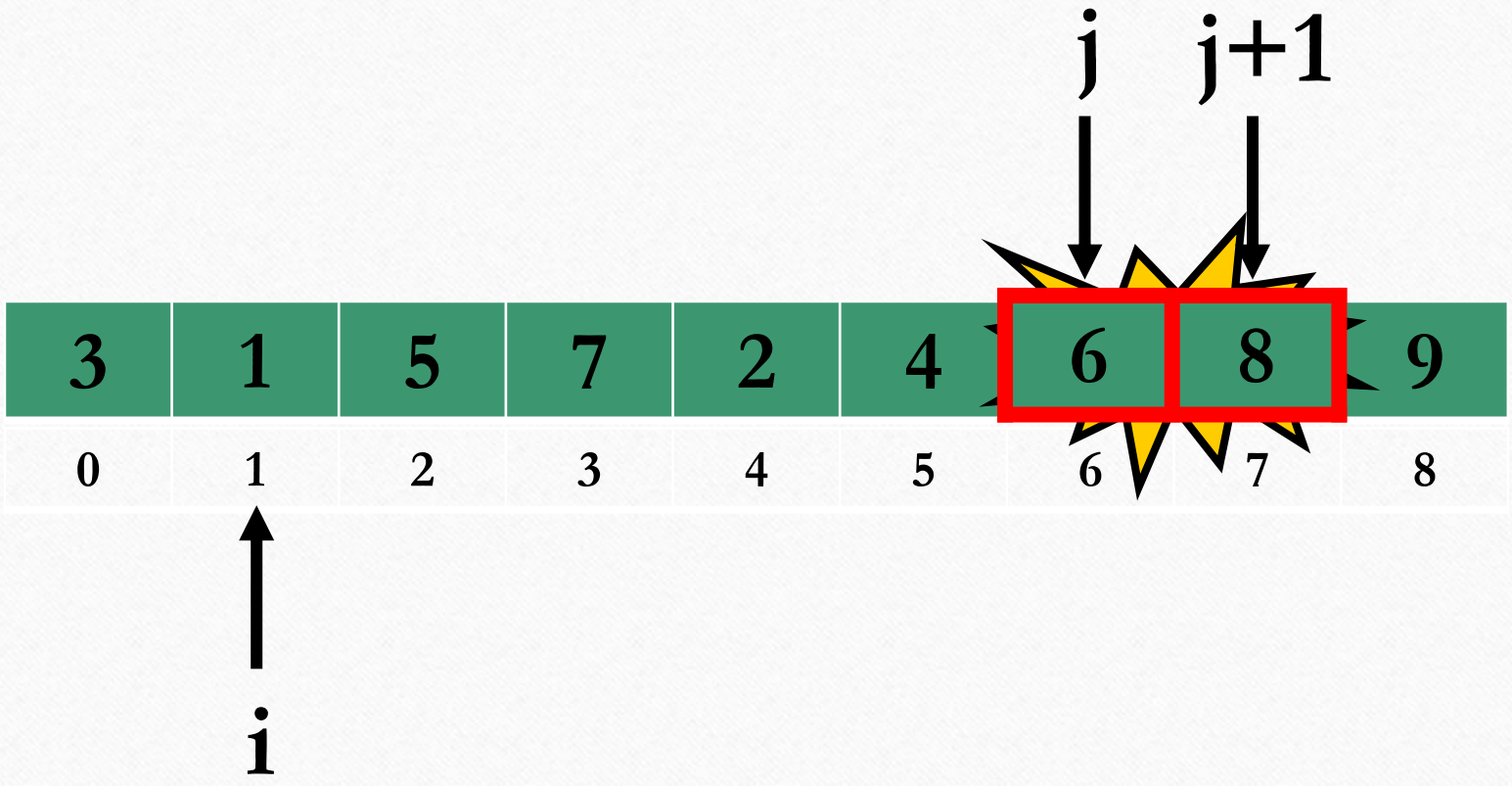
Second pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

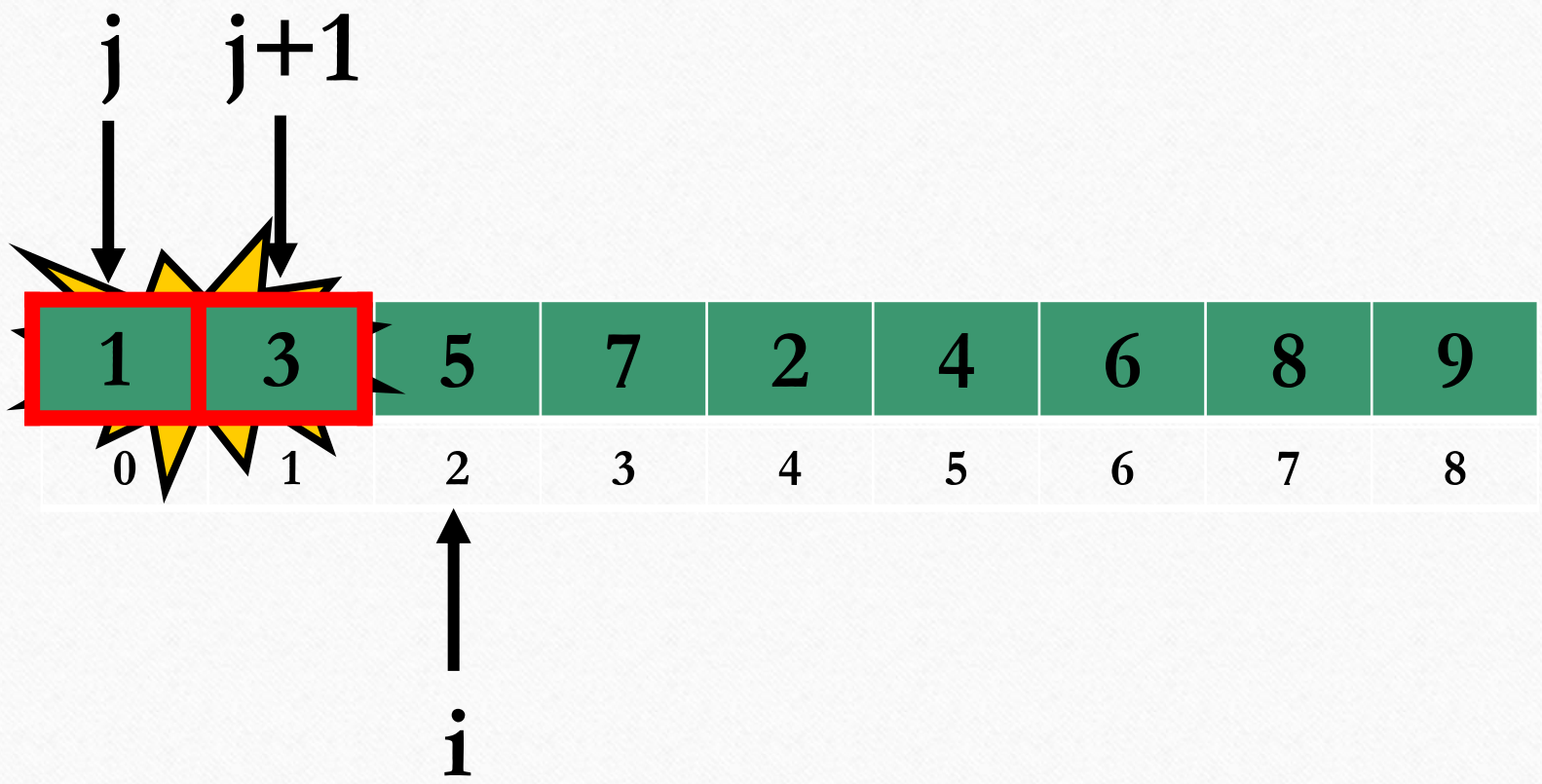
Second pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

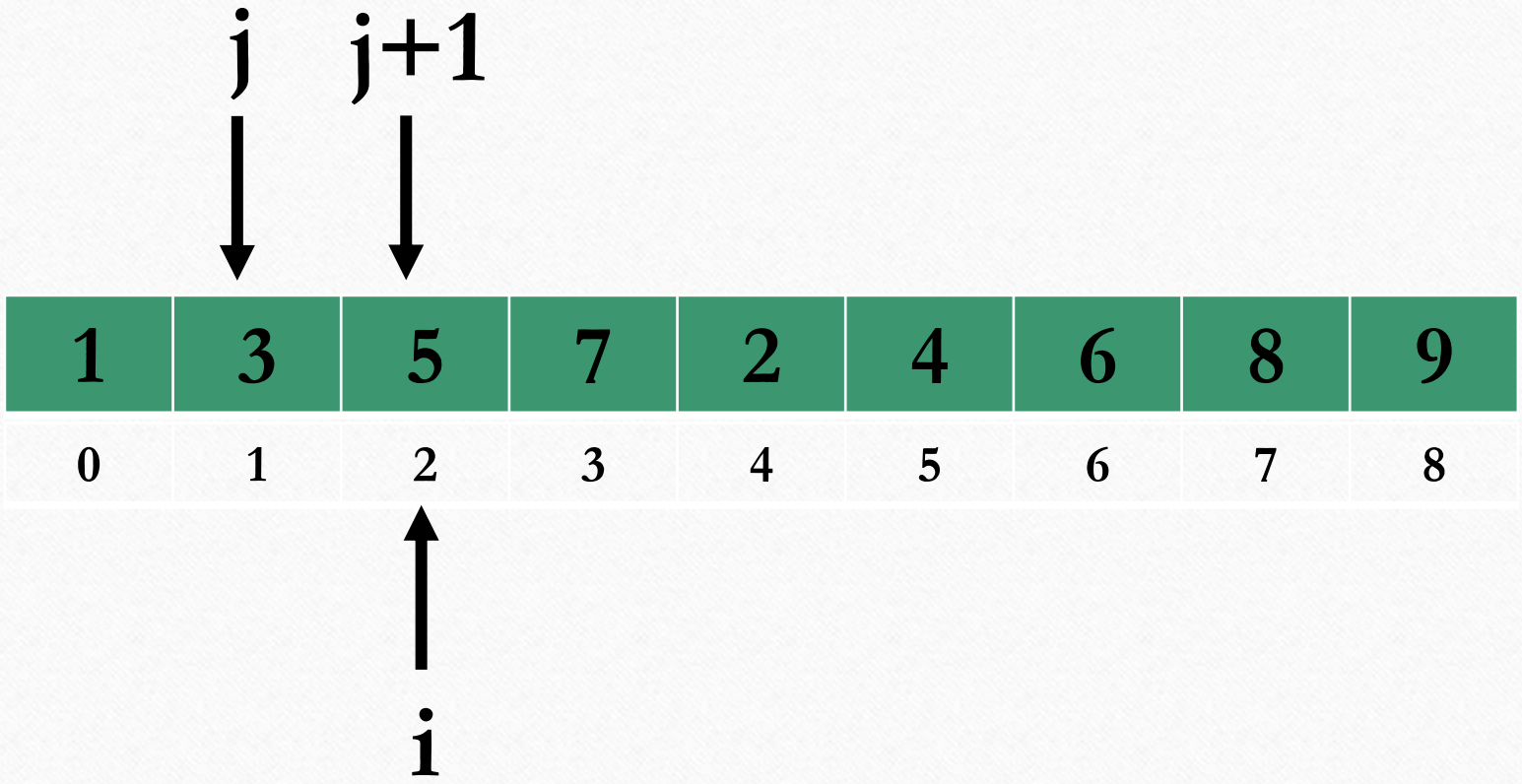
3rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

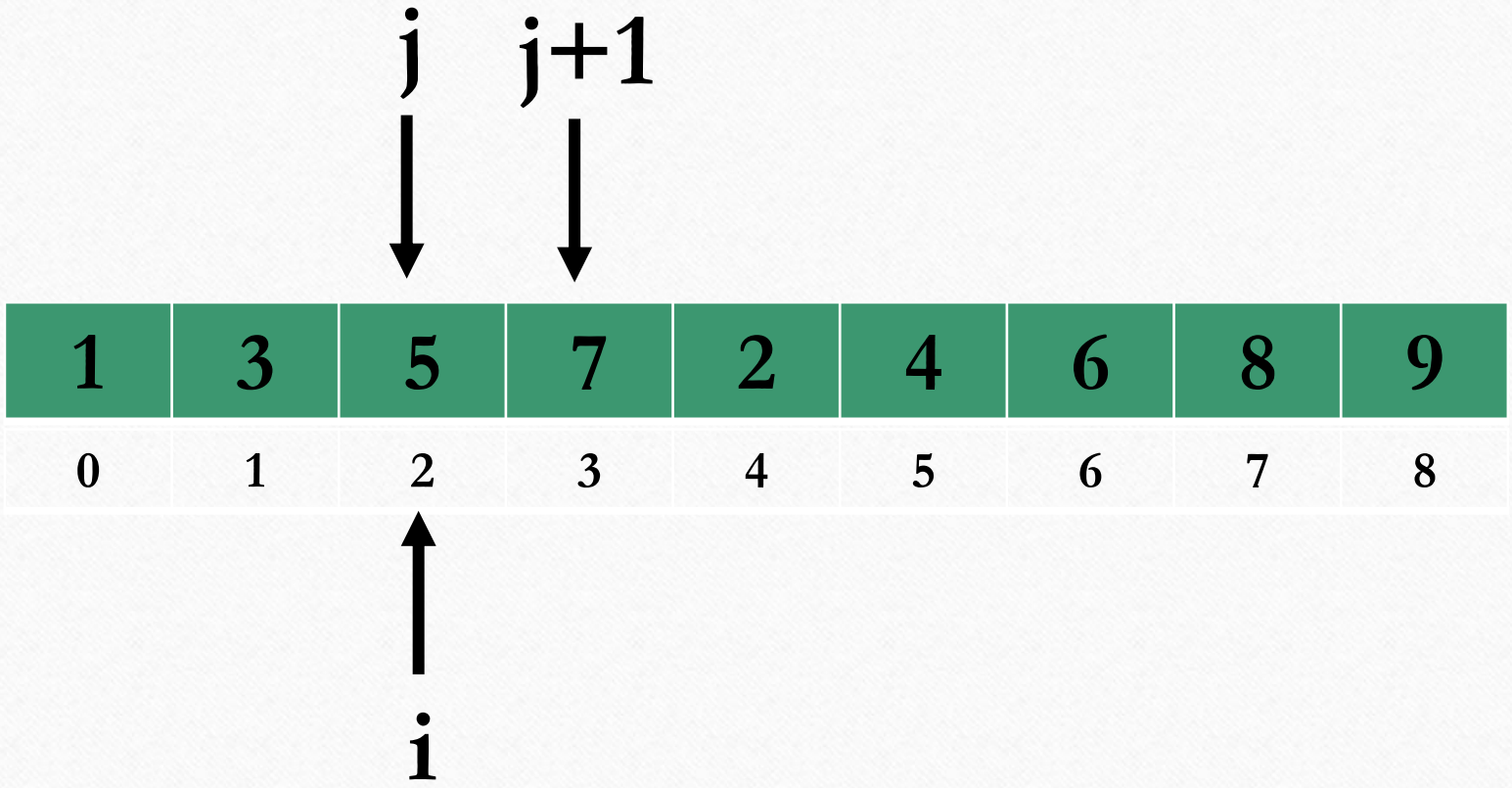
3rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

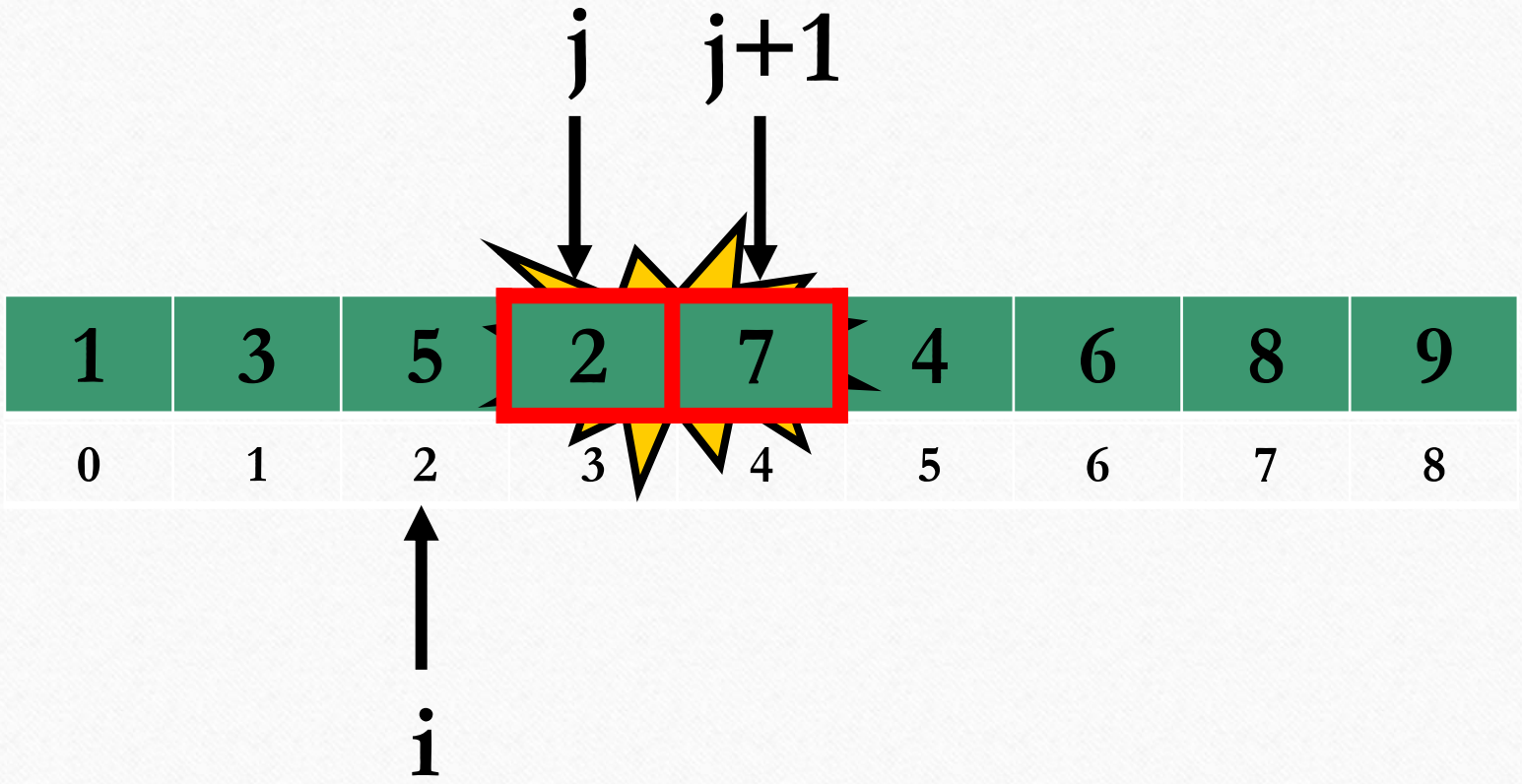
3rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

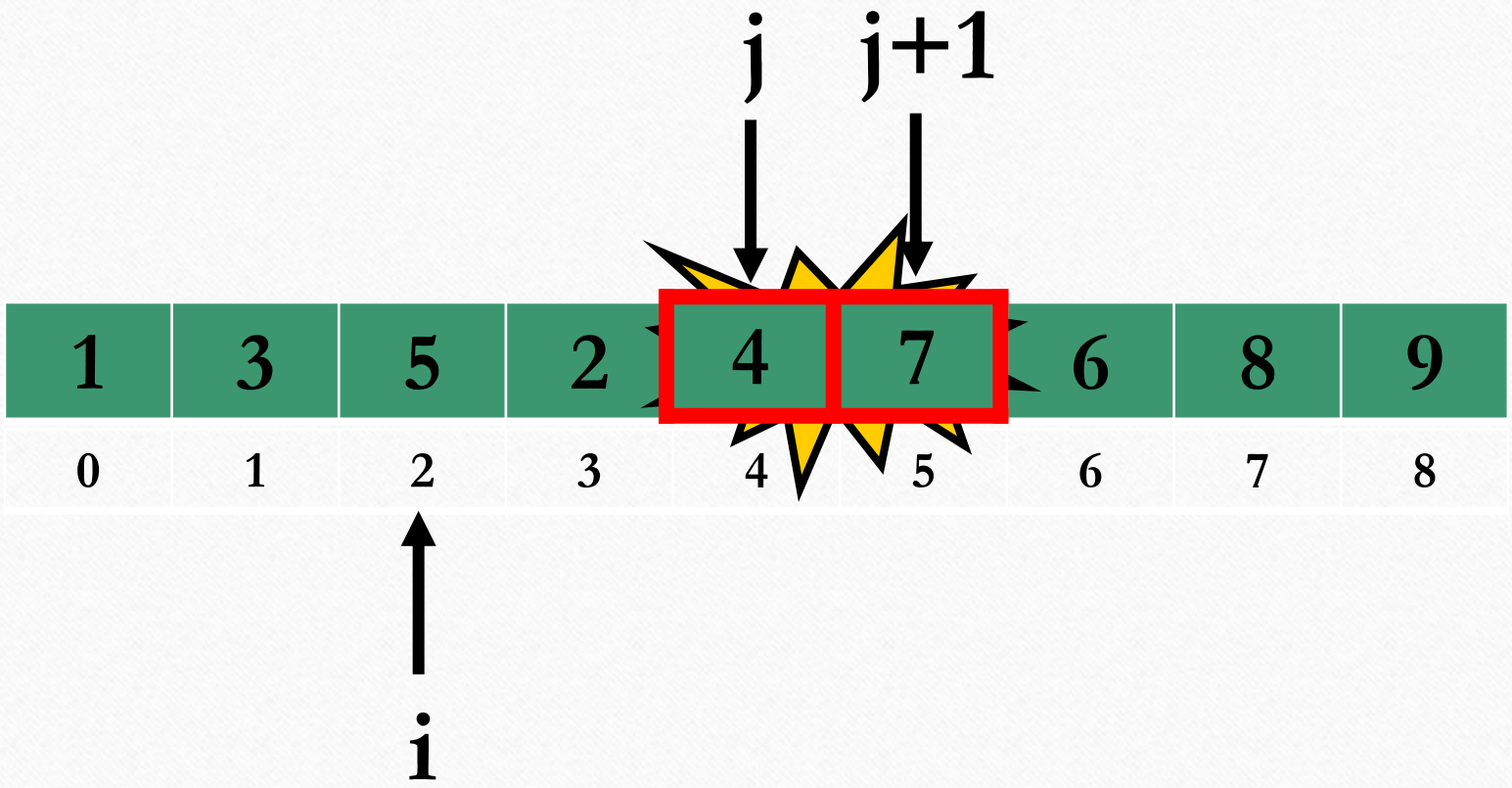
3rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

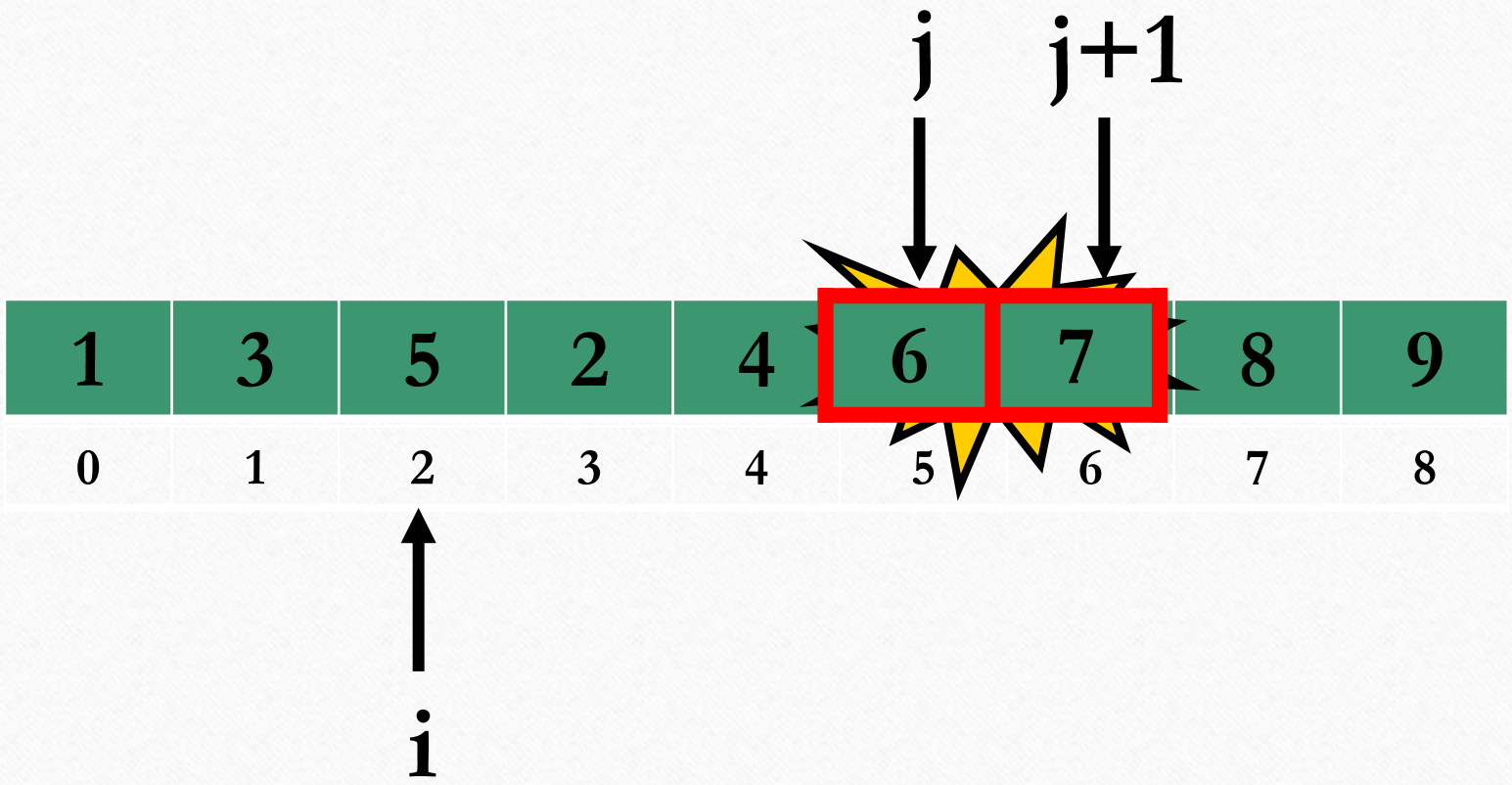
3rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

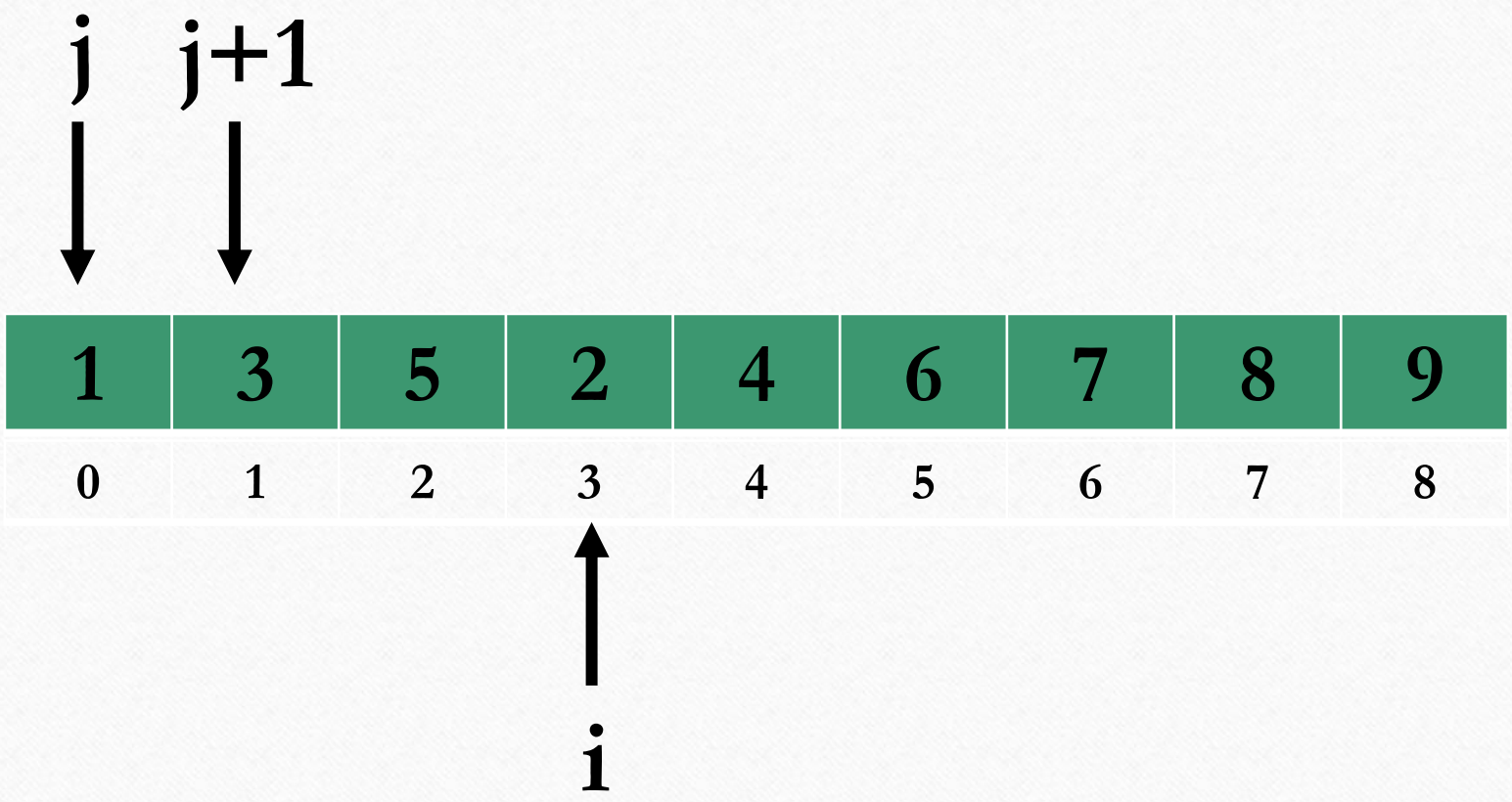
3rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

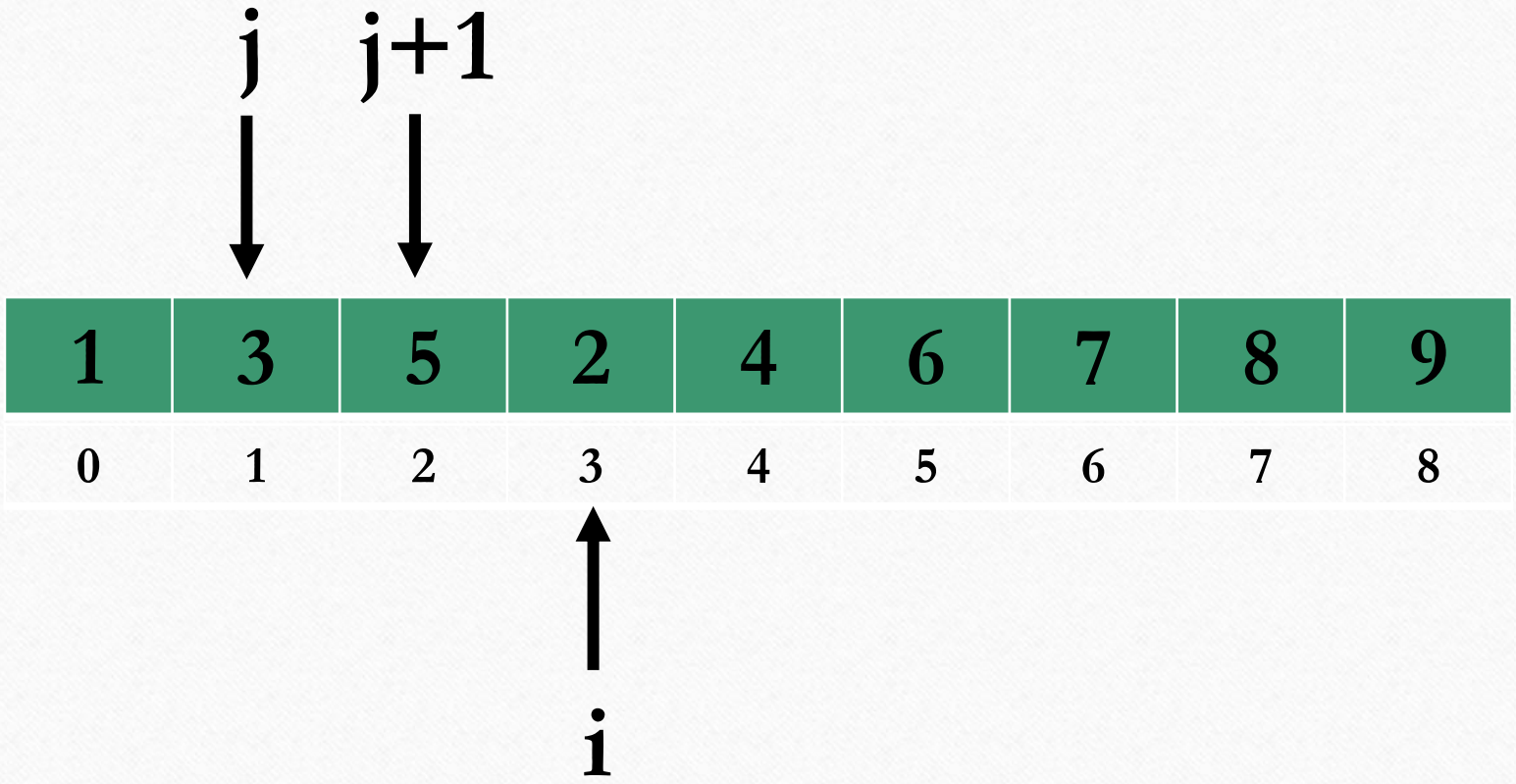
4rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

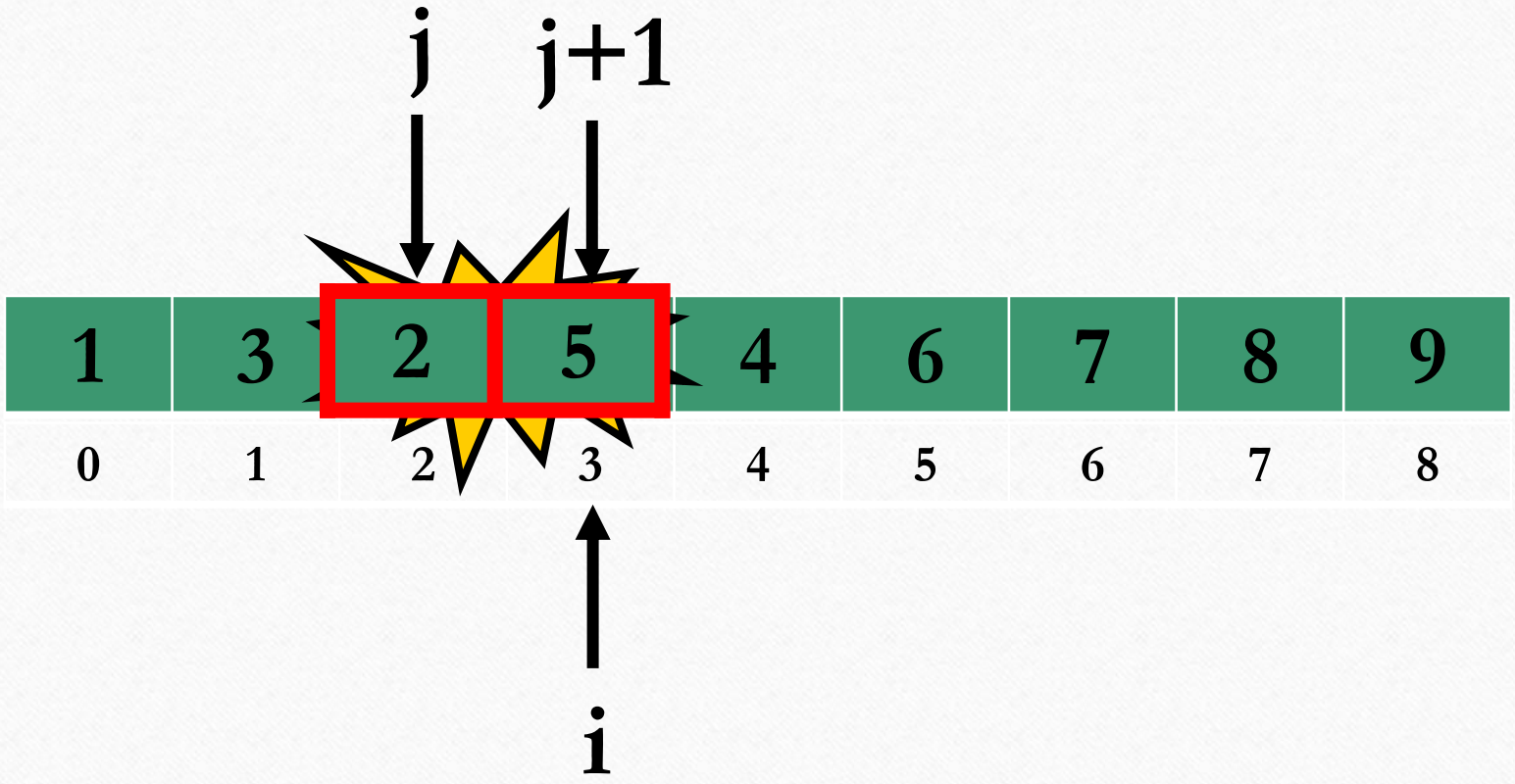
4rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

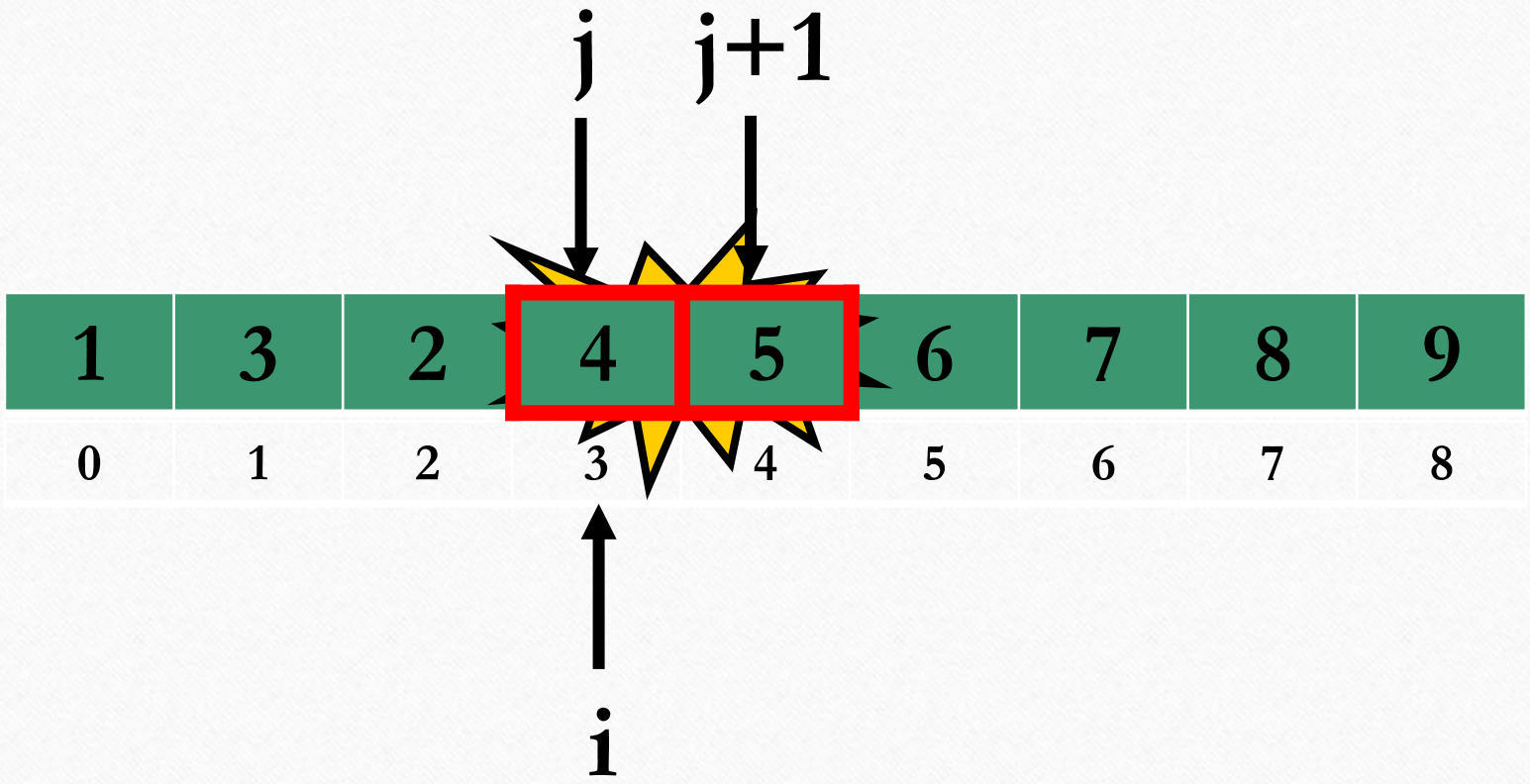
4rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

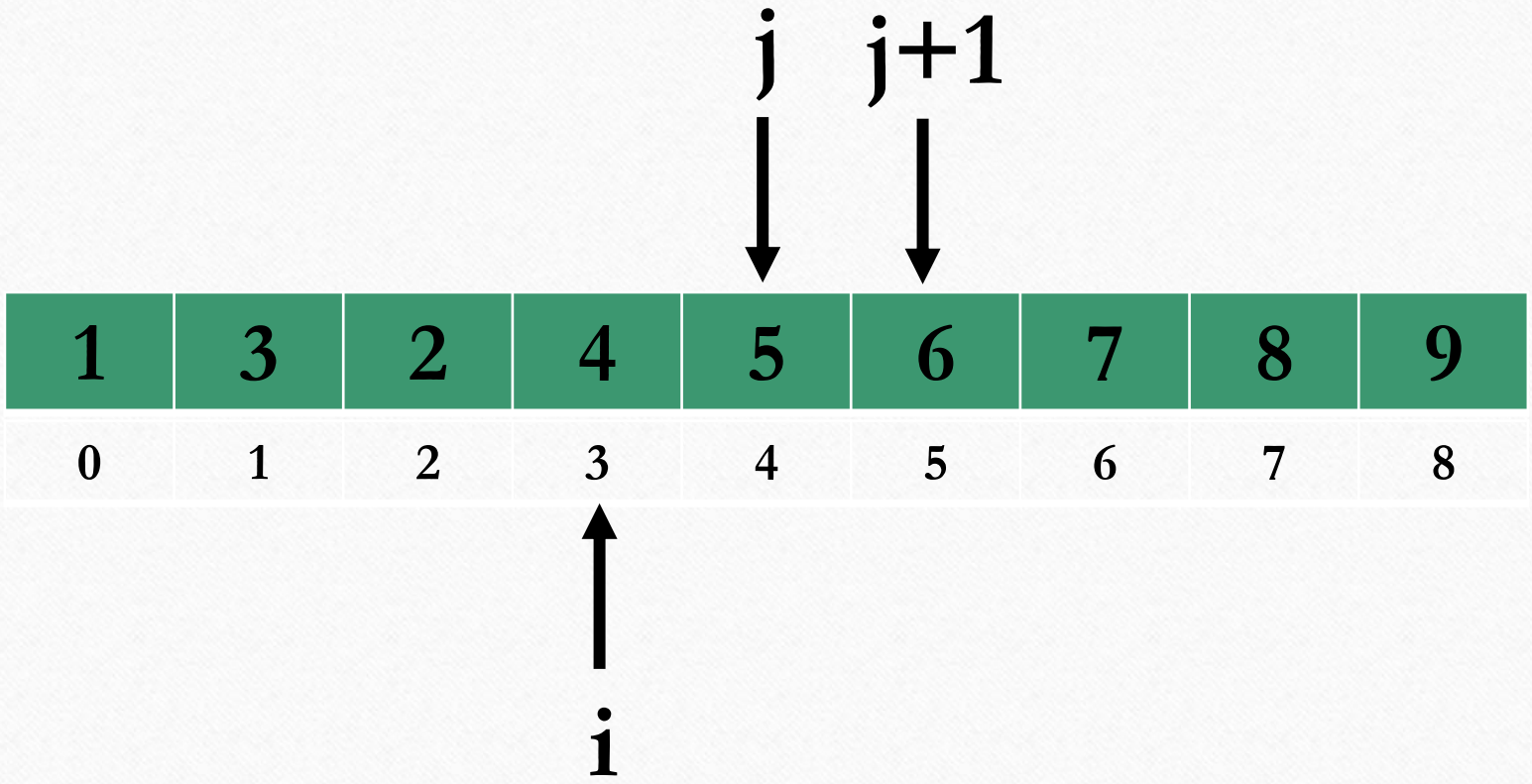
4rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

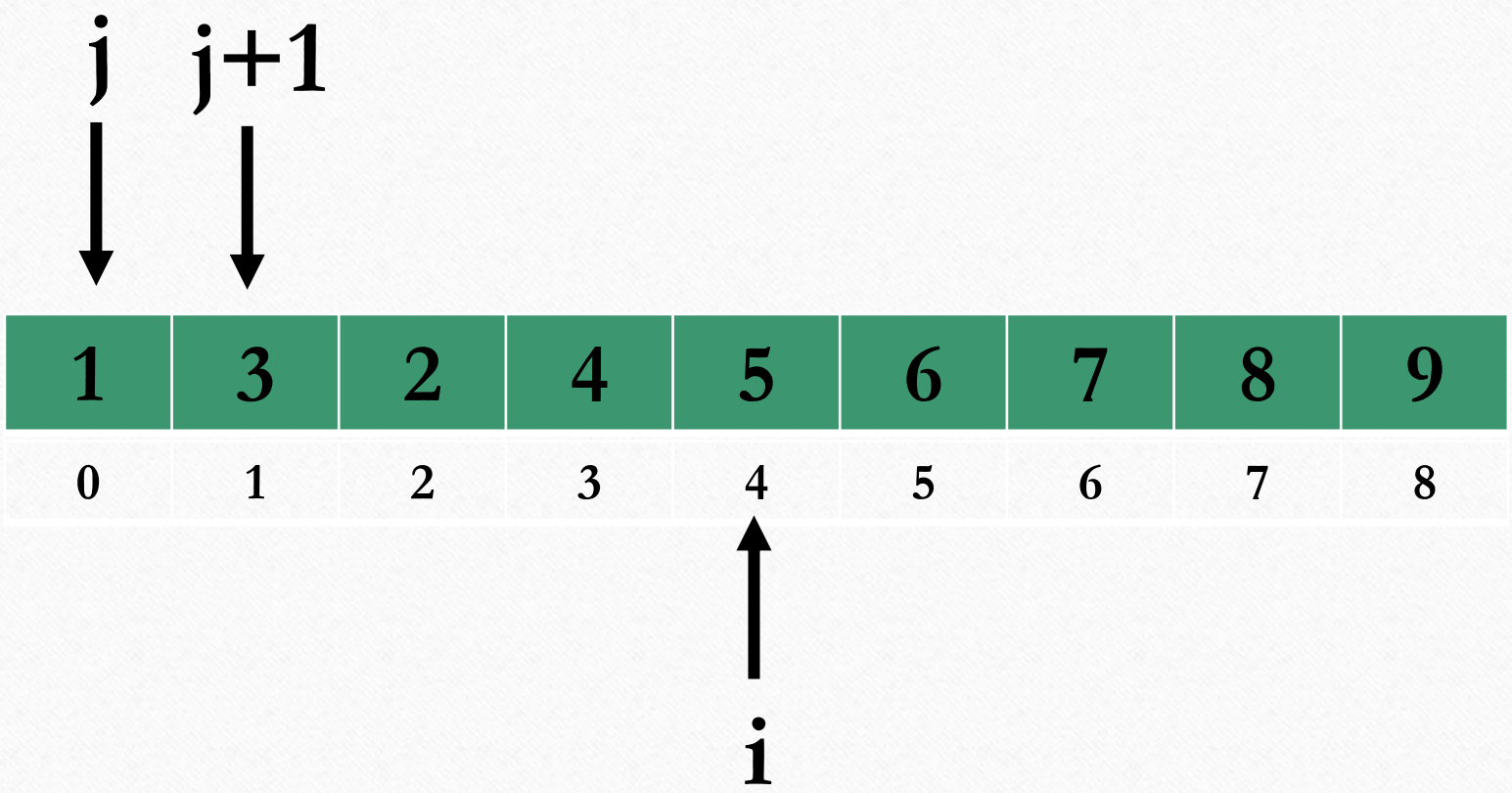
4rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

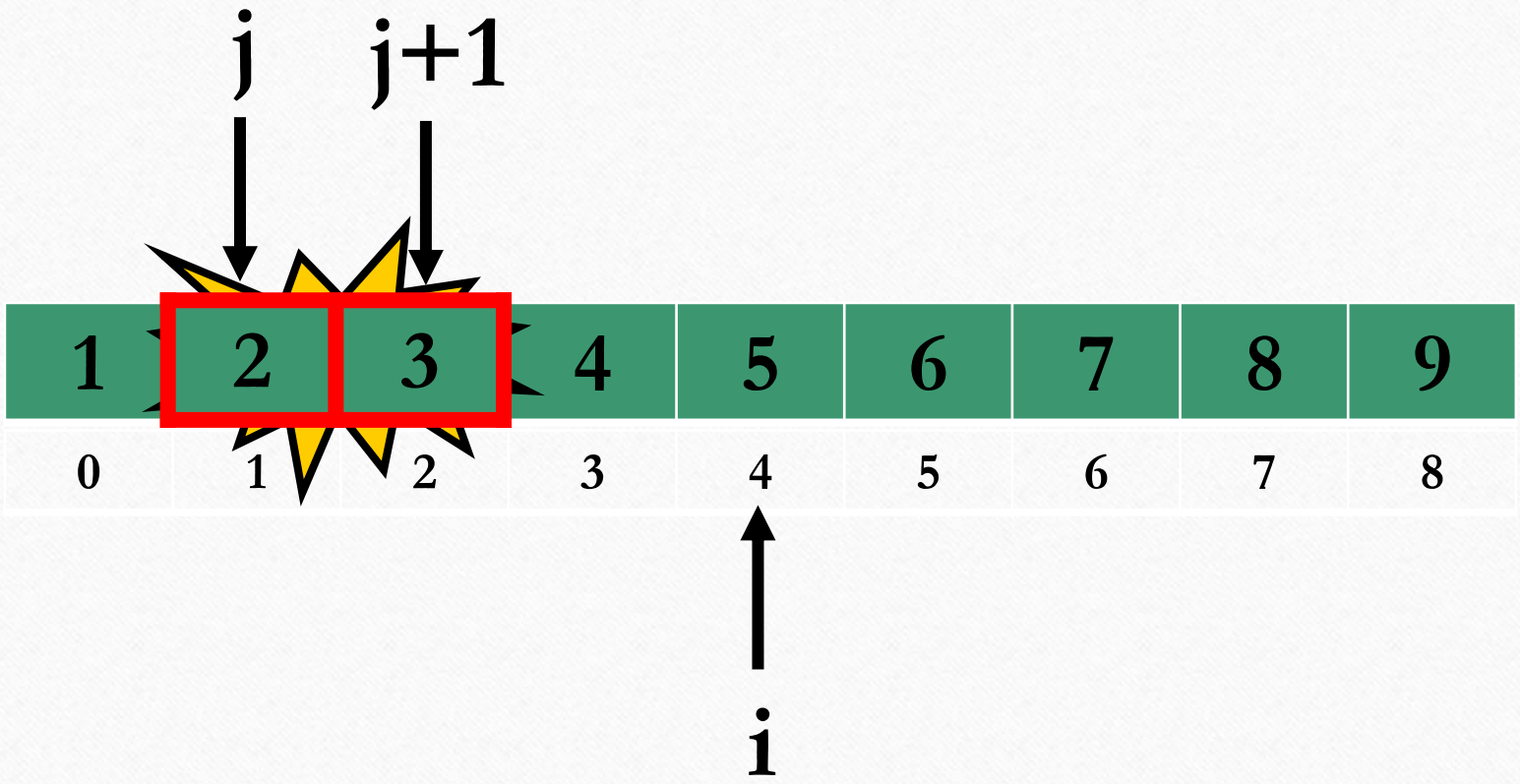
5rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

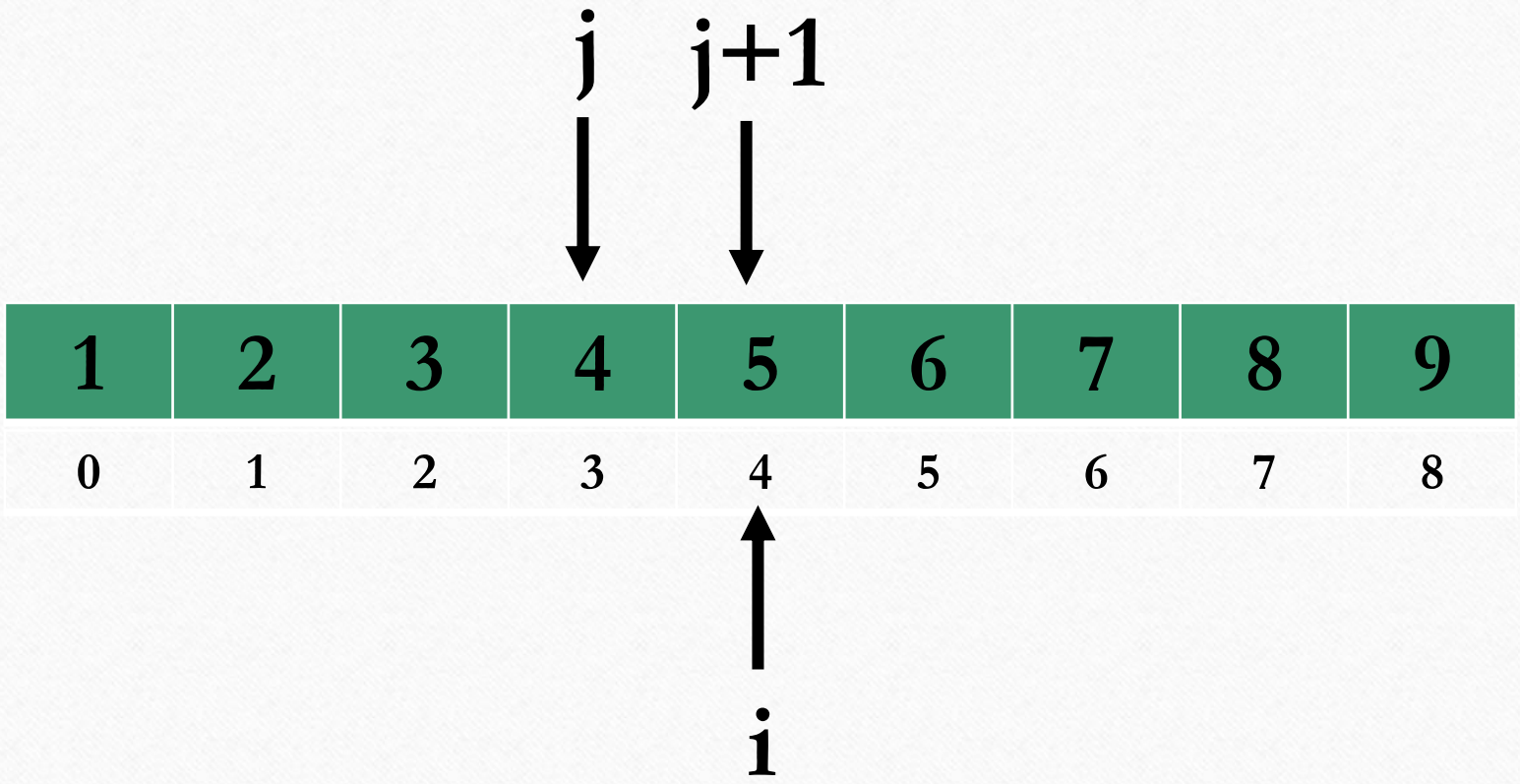
5rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

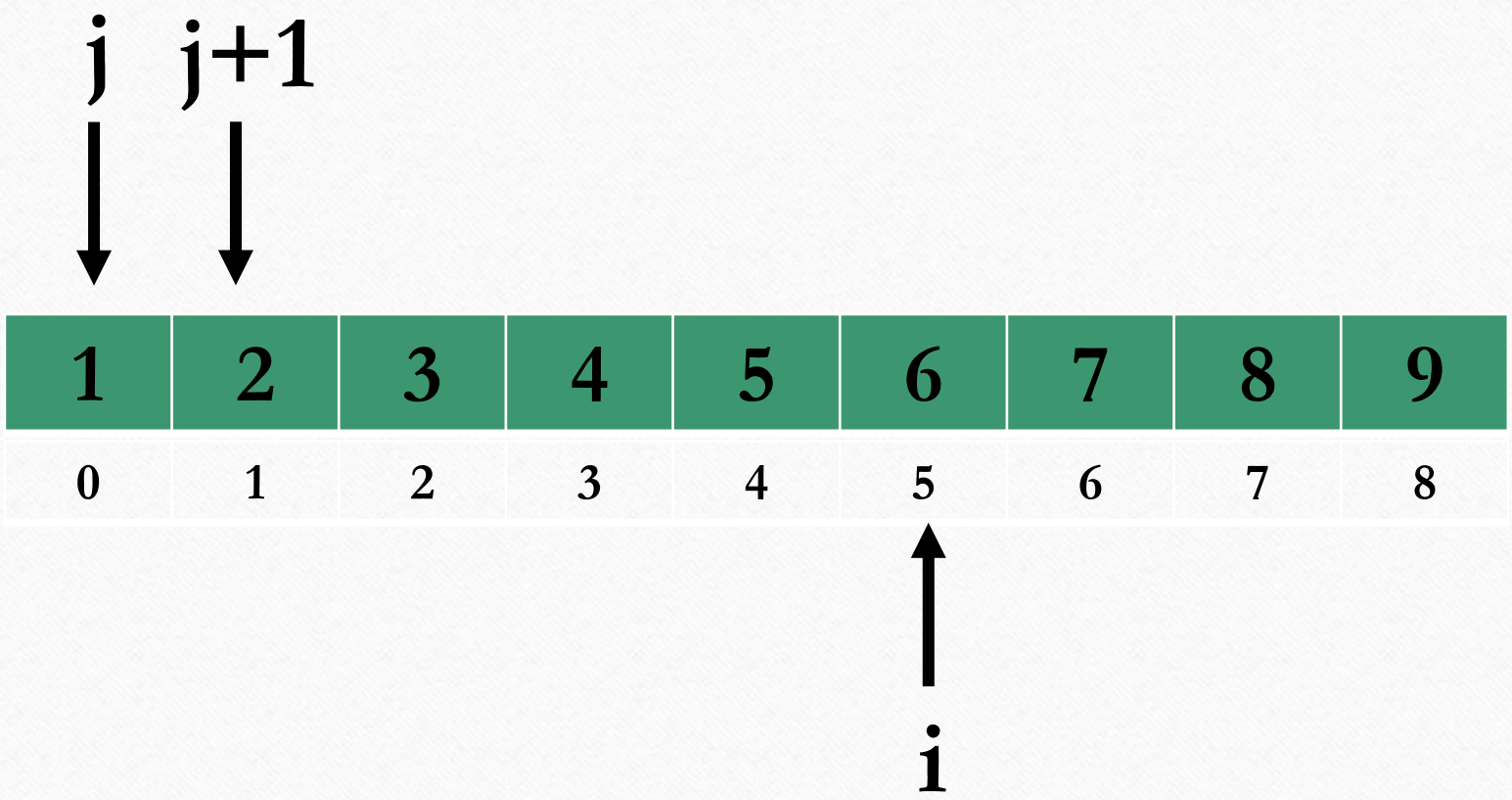
5rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

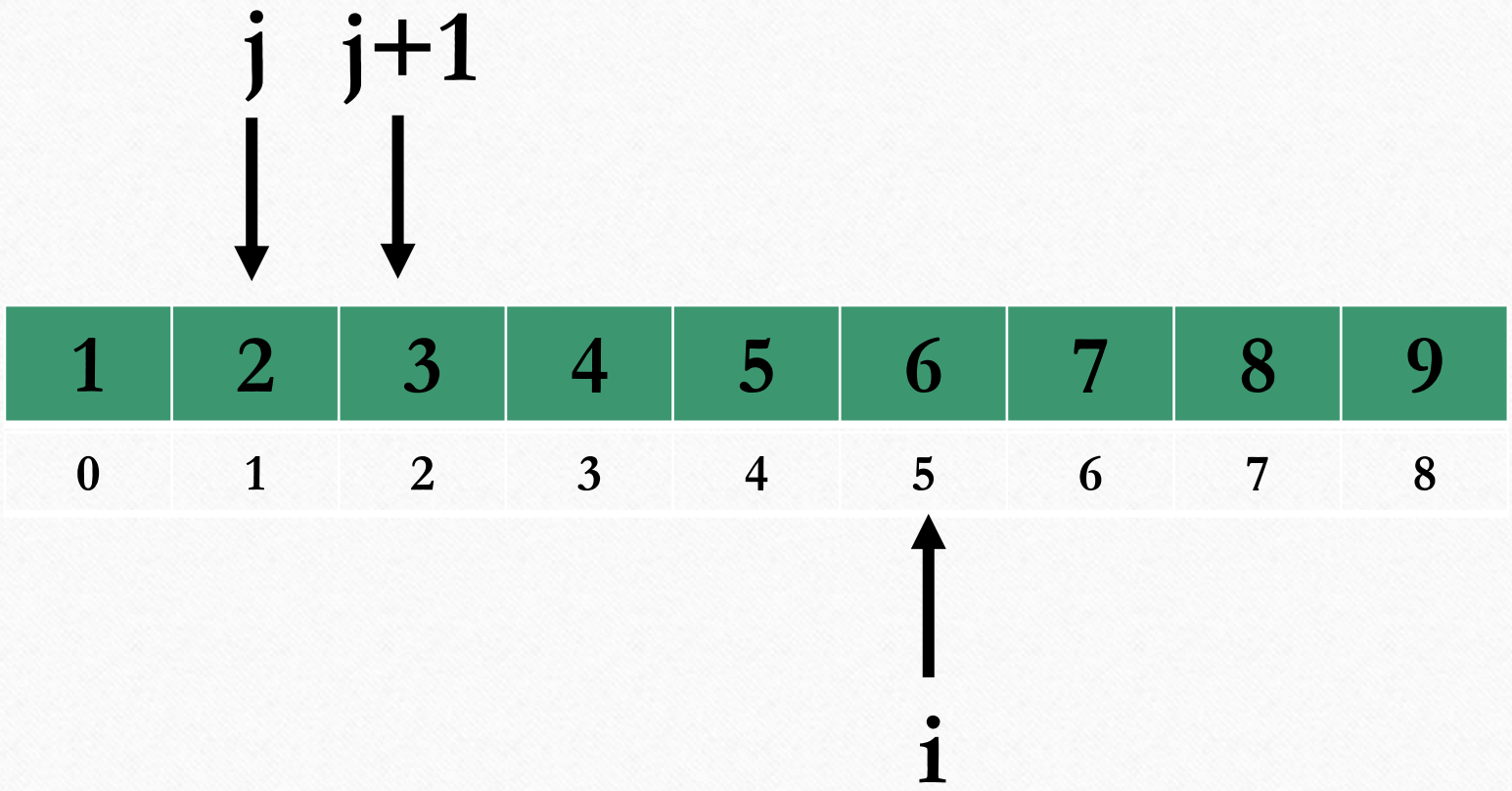
6rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

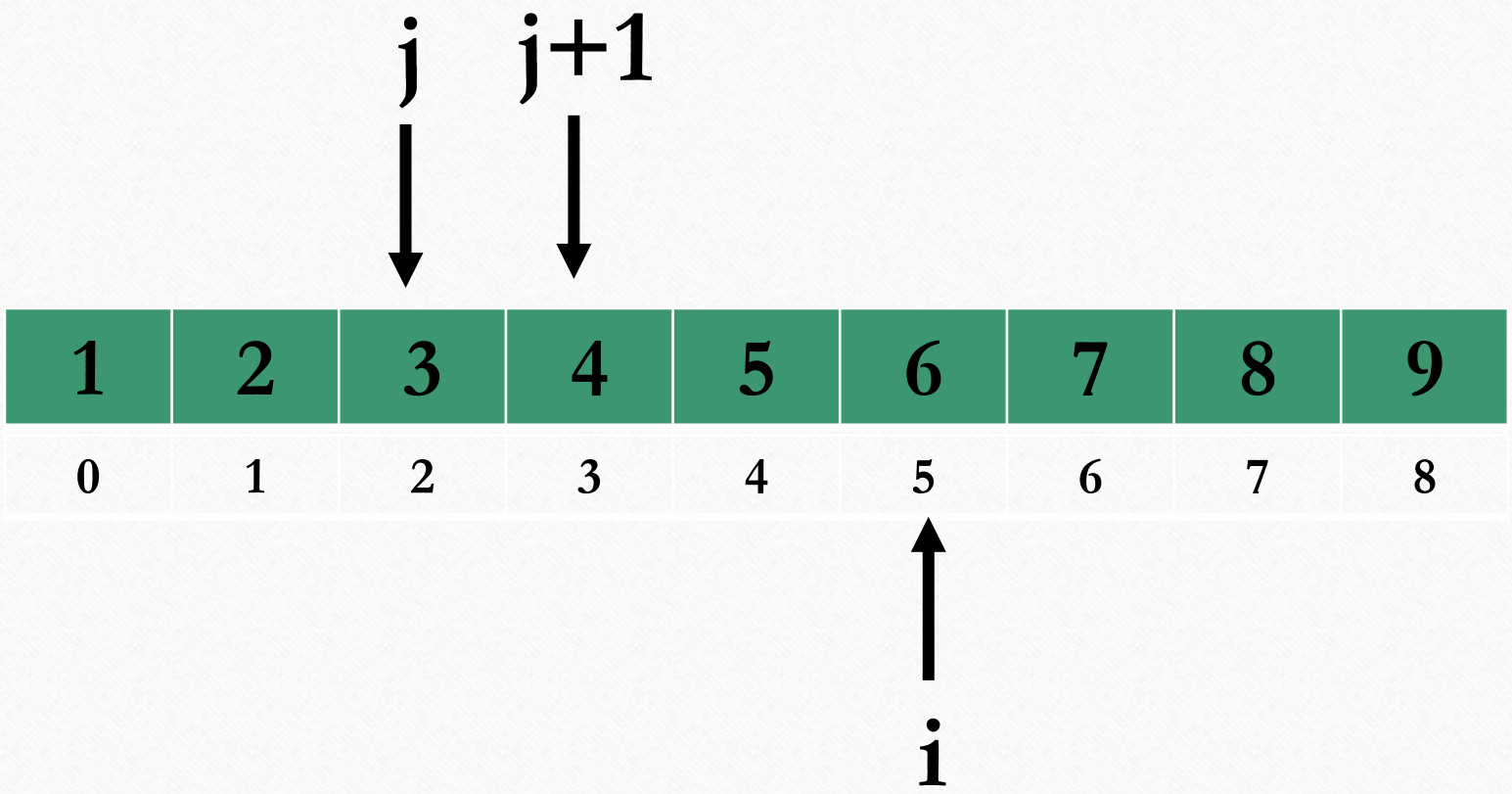
6rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

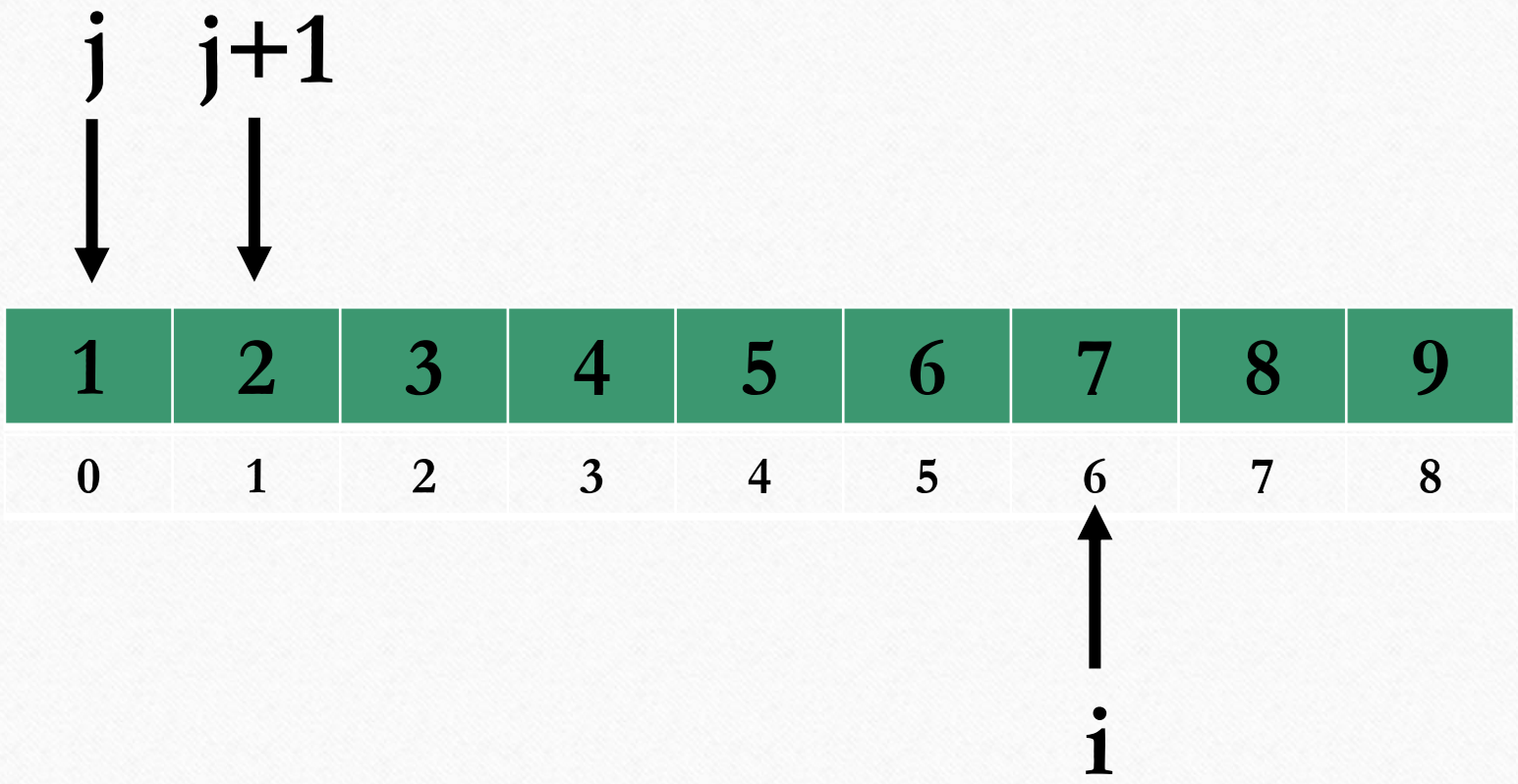
6rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

7rd pass



```
for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
```

IV. Bubble Sort

8rd pass

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

i

End of Bubble Sort

IV. Bubble Sort

Time Complexity of Bubble Sort

```
void bubble_sort (int Vec[], int n) {
    for (int i=0; i<n-1; i++)
        for (int j=0; j<n-i-1; j++)
            if (Vec[j]> Vec[j+1])
                swap(Vec[j], Vec[j+1]);
}
```

i	j
0	N-1
1	N-2
2	N-3
:::::	::::
:::::	::::
N-2	1

IV. Bubble Sort

Time Complexity of Bubble Sort

- If we define the size of the array as $\boldsymbol{n}$ then we complete $\boldsymbol{n-1}$ comparisons on the first pass, $\boldsymbol{n-2}$ comparisons on the second pass, $\boldsymbol{n-3}$ comparisons on the third pass, and so on.
- The total number of comparisons we complete is equal :

$$\boldsymbol{T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)}$$

- Thus, our current bubble sort implementation runs in $\boldsymbol{\mathcal{O}(n^2)}$ time.

V. Selection Sort

- Selection sorting is one of the **simplest** sorting algorithms:
- First, find the **smallest** item in the array and **swap** it with the **first entry** (itself if the first entry is already the smallest).
- Then find the **next smallest** item and **swap** it with the **second entry**.
- Continue in this way until the entire array has been sorted.

V. Selection Sort

- The code is shown below:

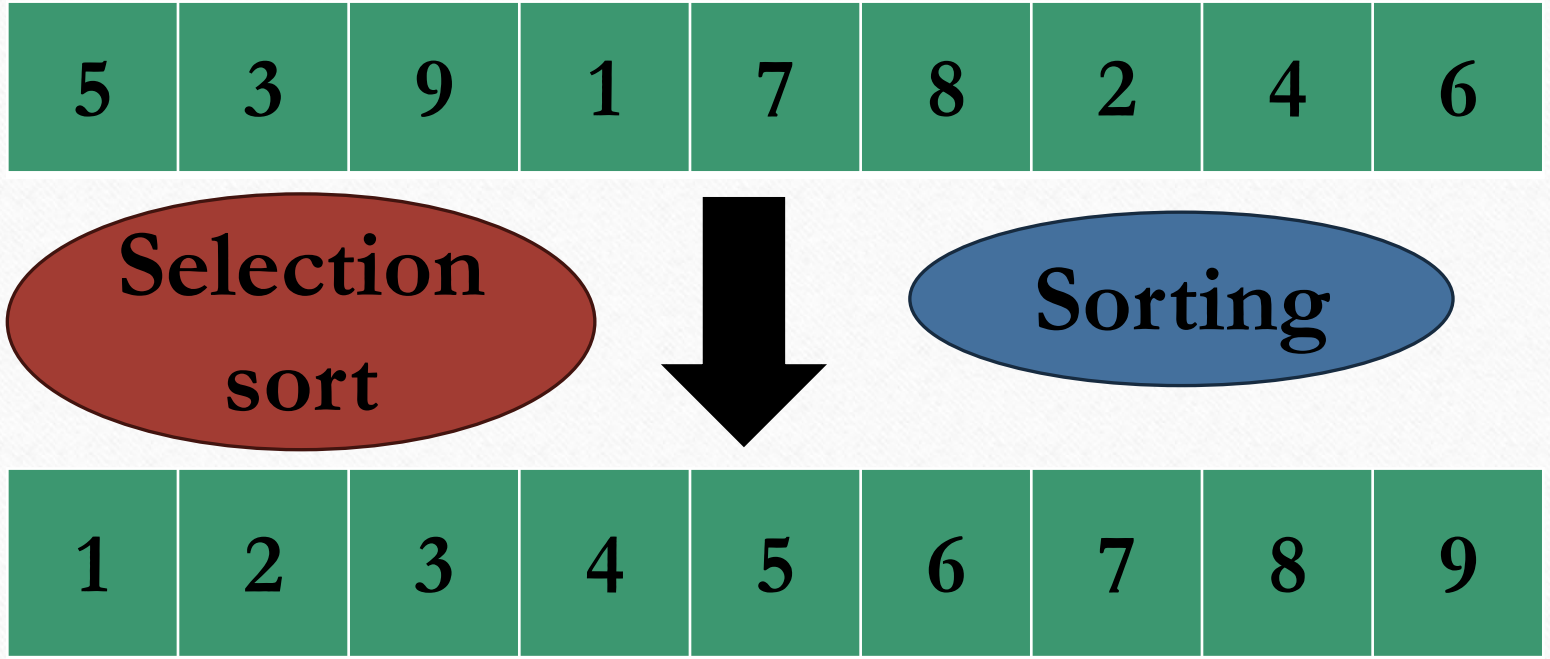
```
void selection_sort(int Vec[], int n) {
   for (int i = 0; i < n - 1; i++) {
      int min_index = i;
      for (int j = i + 1; j < n; j++)
         if (Vec[j] < Vec[min_index]) {
            min_index = j;
         }
      swap(Vec[i], Vec[min_index]);
   }
}
```

V. Selection Sort

Example:

- Let's look at selection sort in action, using the same array we used earlier: **[5, 3, 9, 1, 7, 8, 2, 4, 6]**.
- With each pass, we **find the next smallest** value and **place it** in its **correct position**.
- In the process below, the two underlined elements are swapped, and bolded elements are fixed in their correct sorted position.

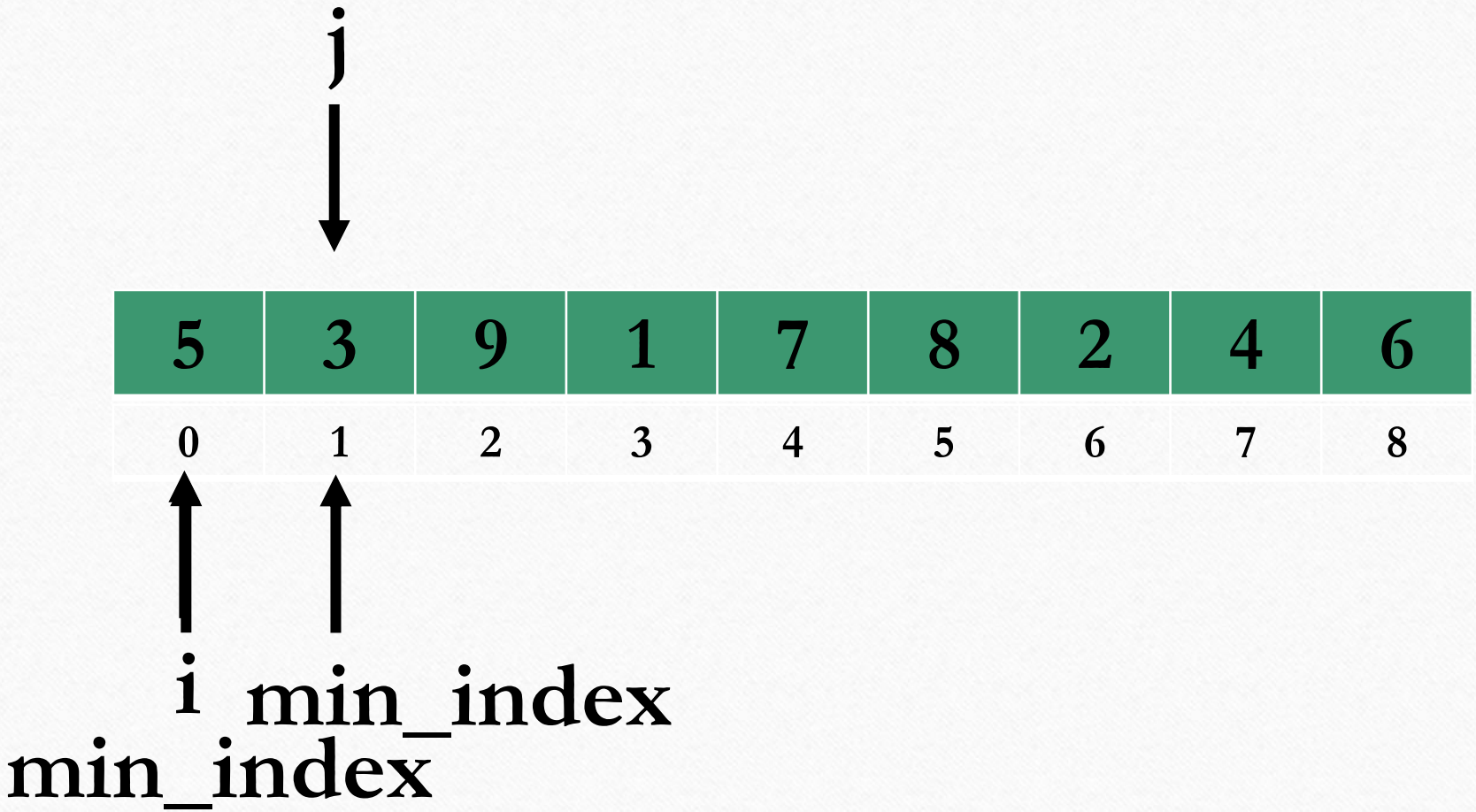
V. Selection Sort



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

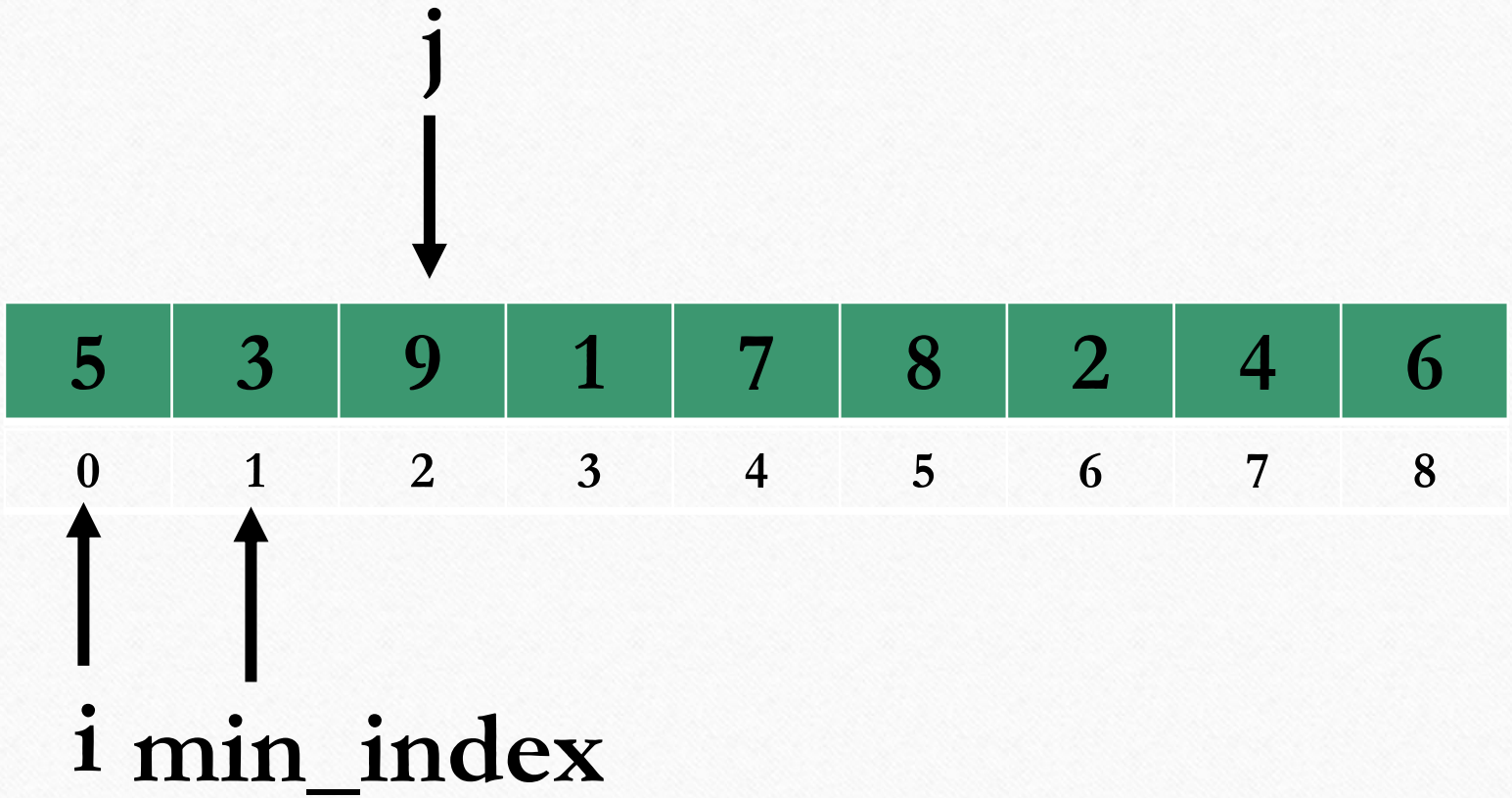
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

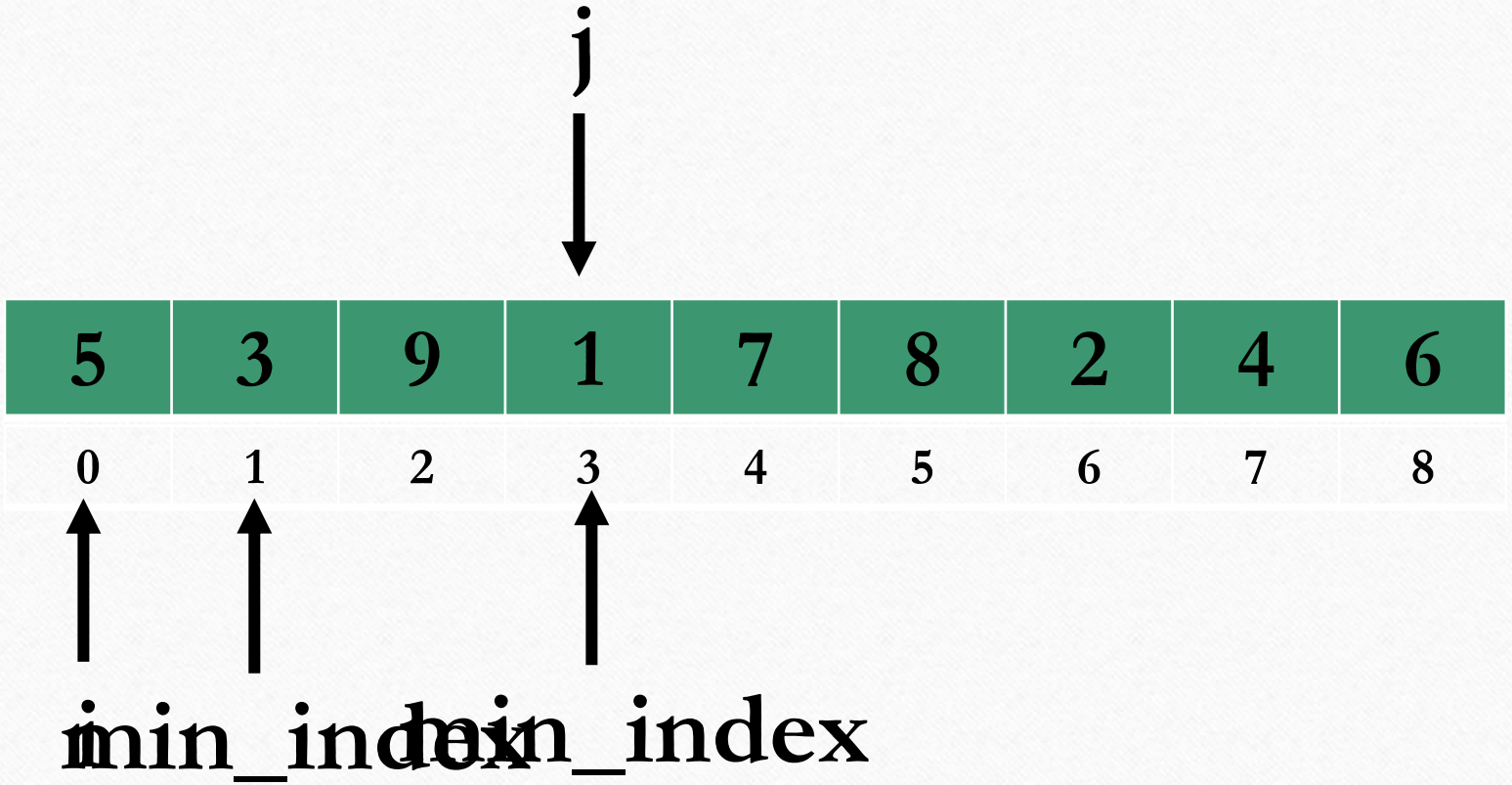
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

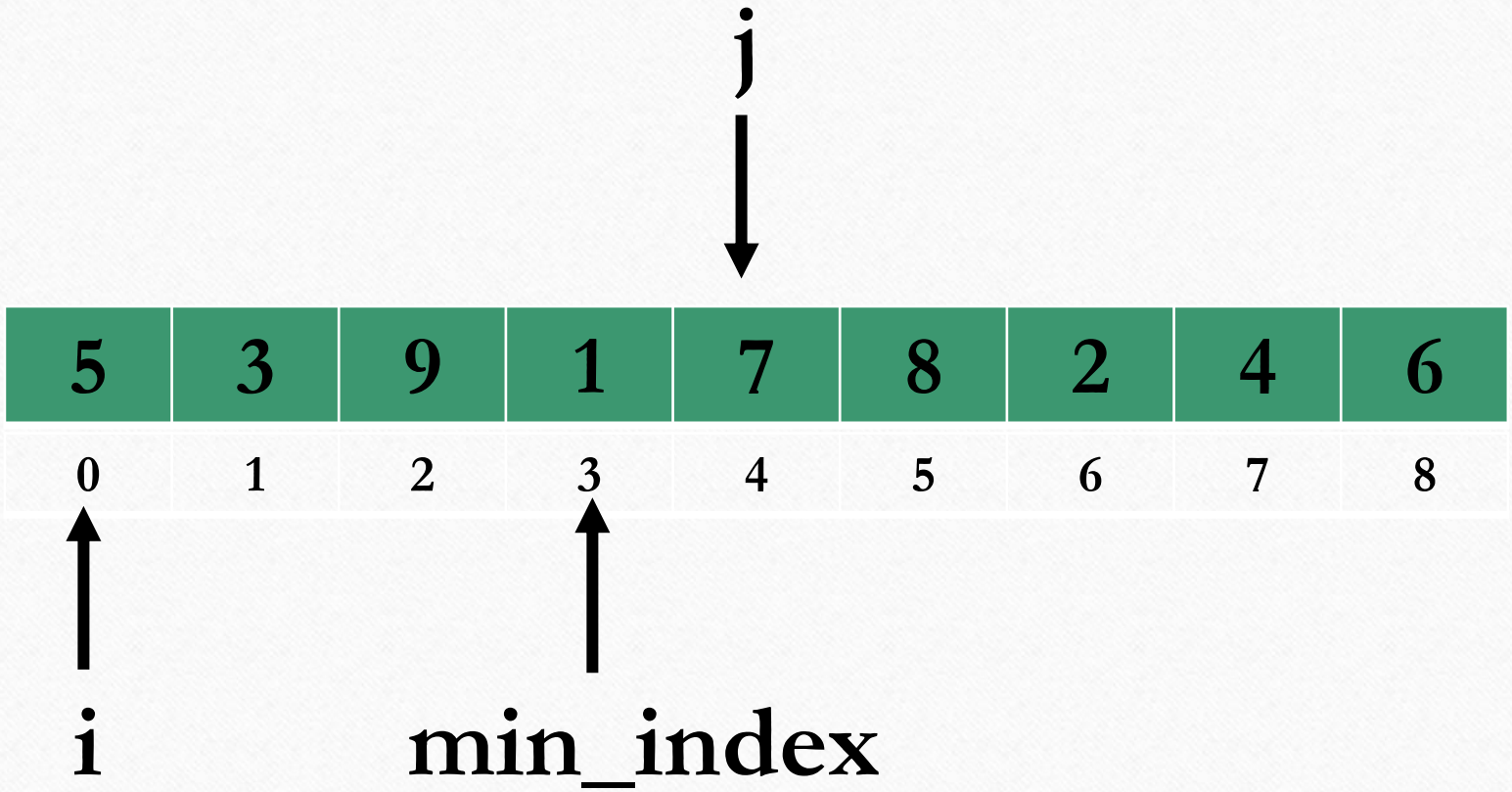
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

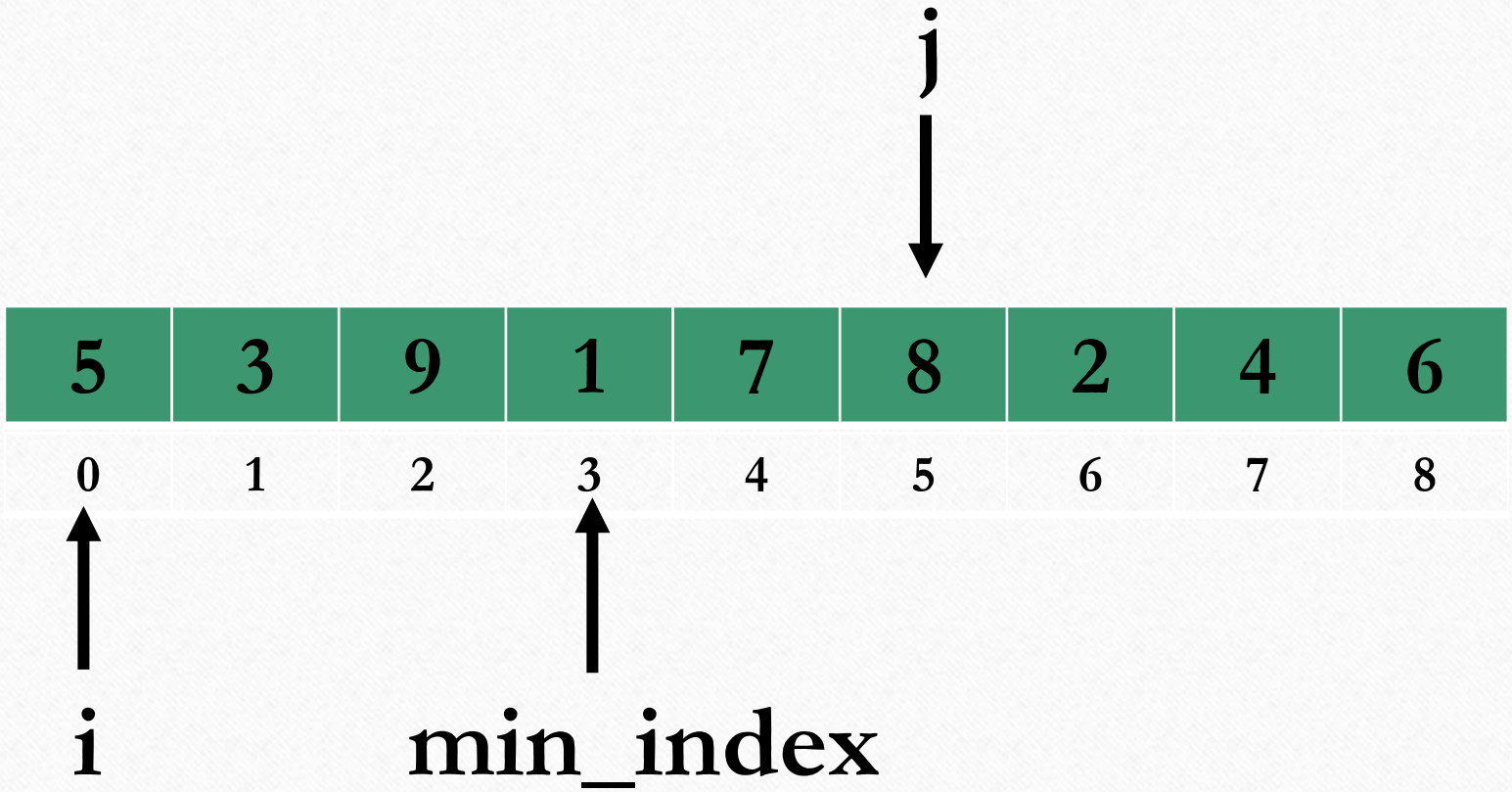
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

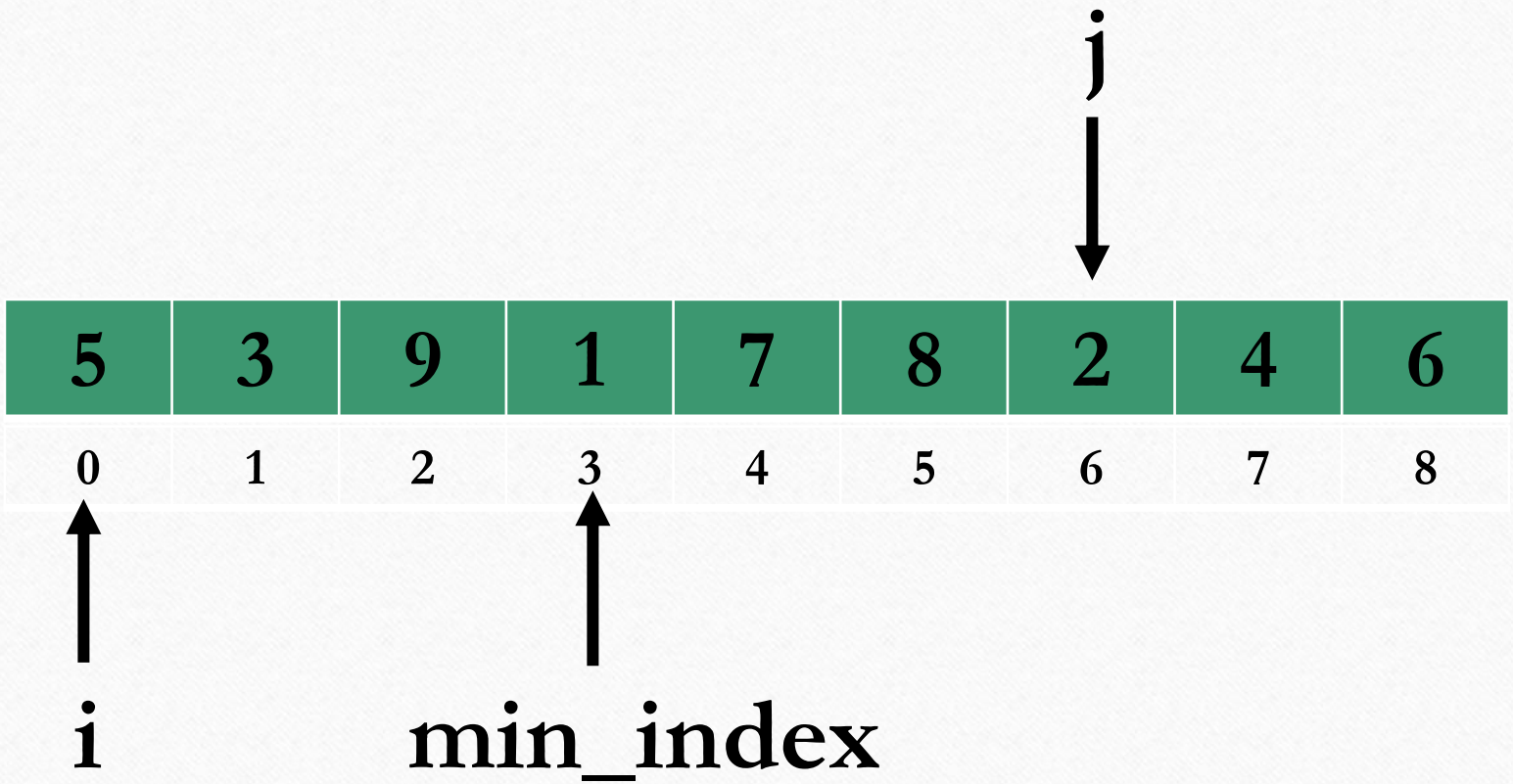
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

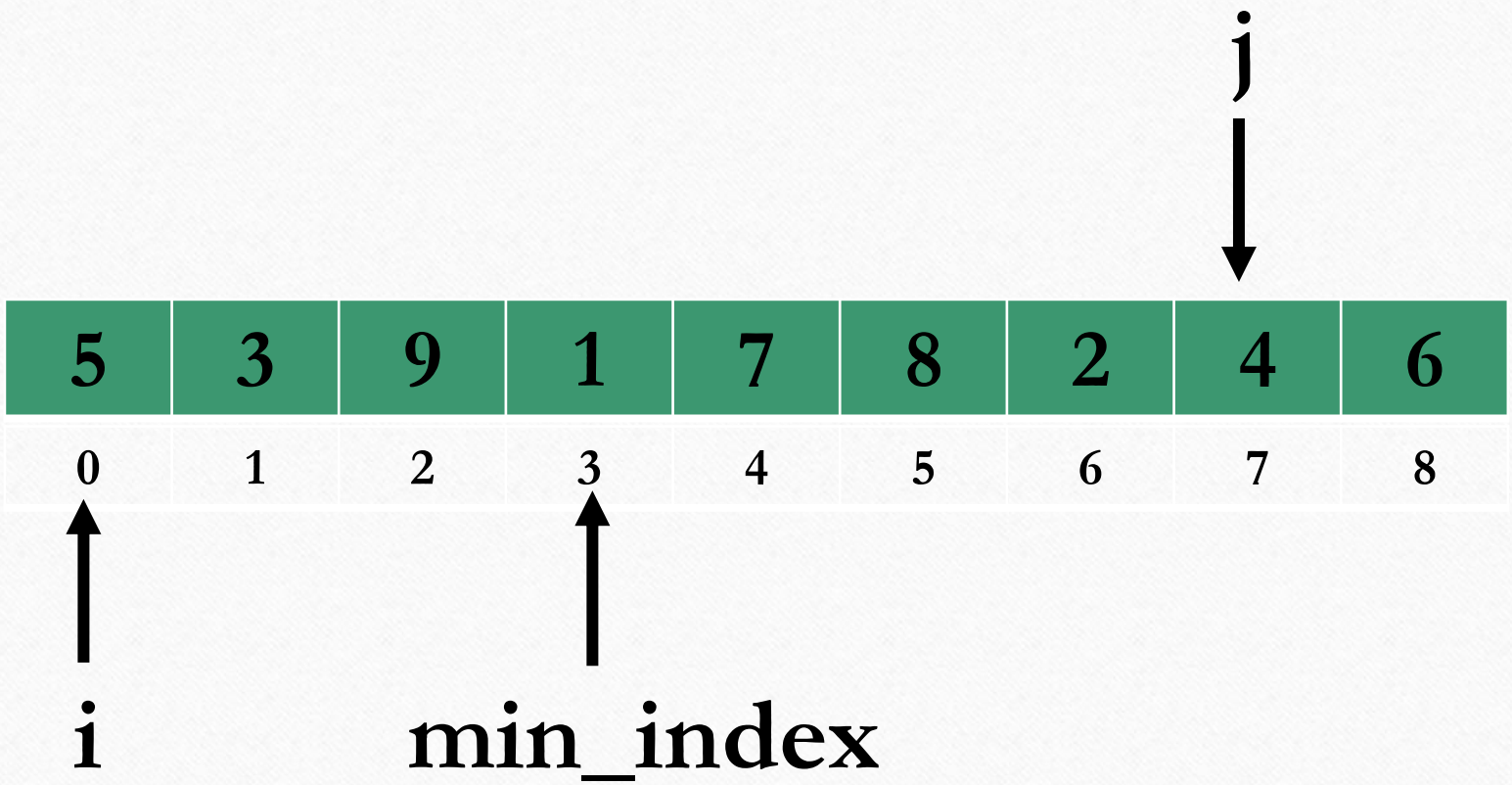
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

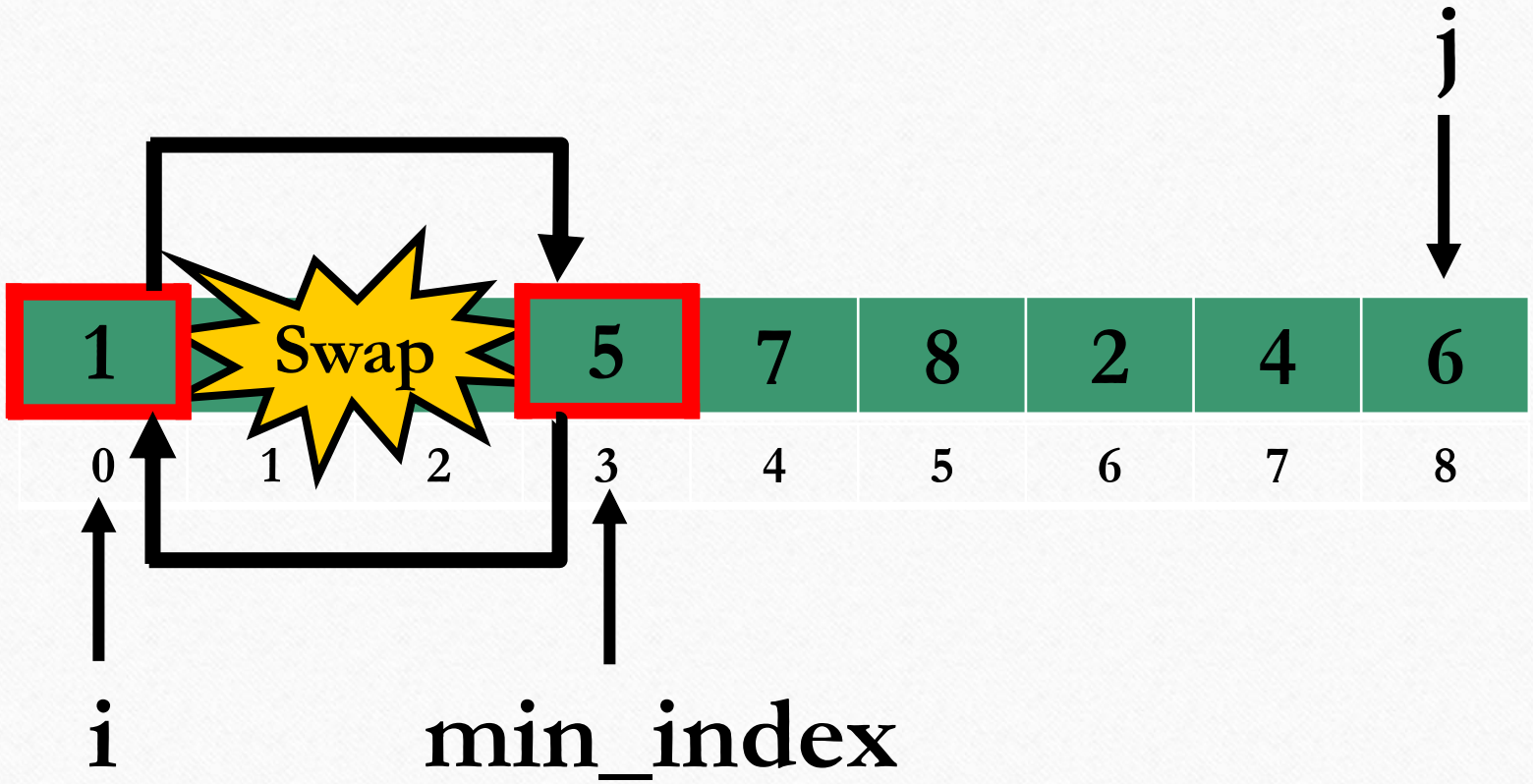
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

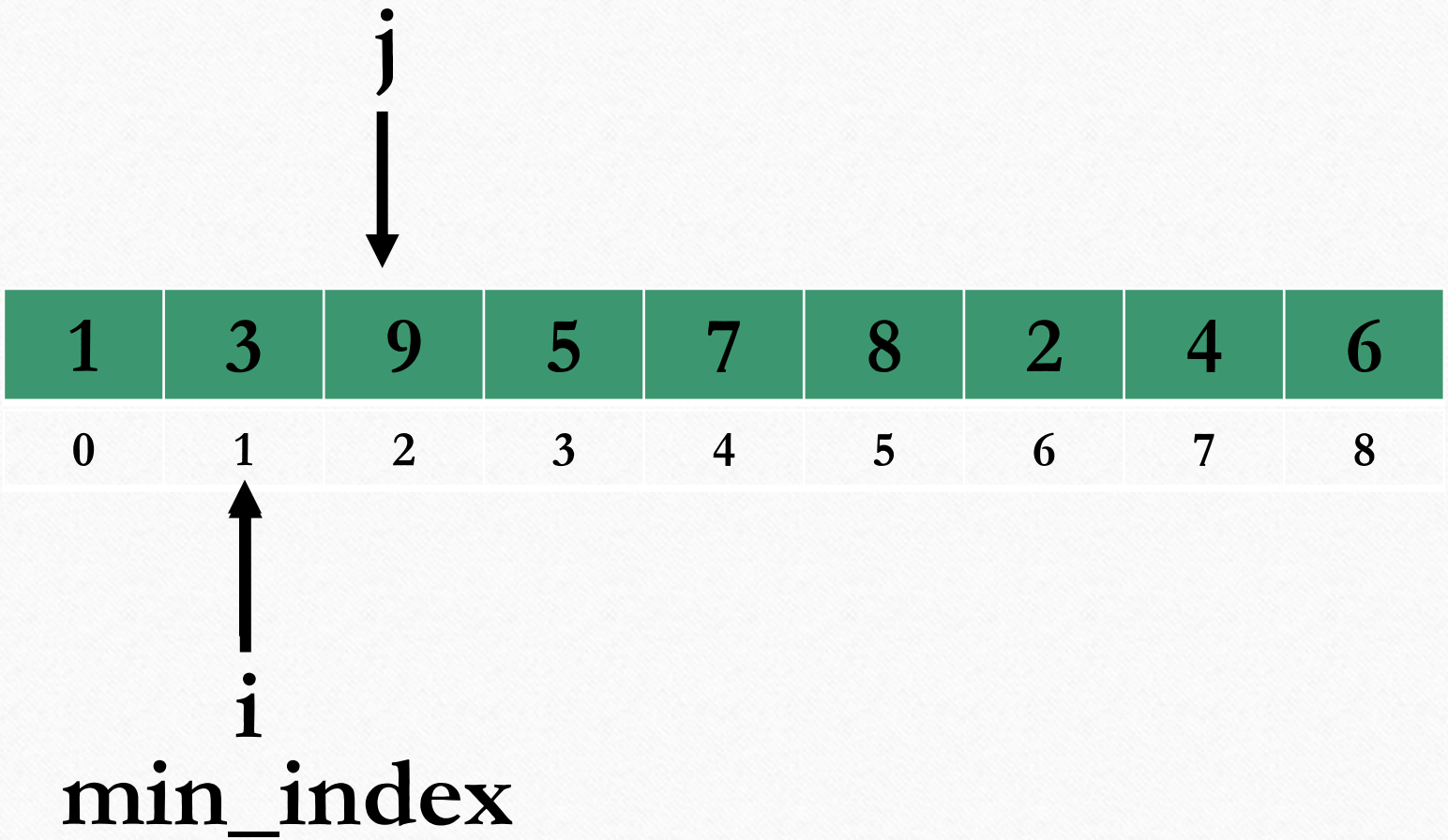
First pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

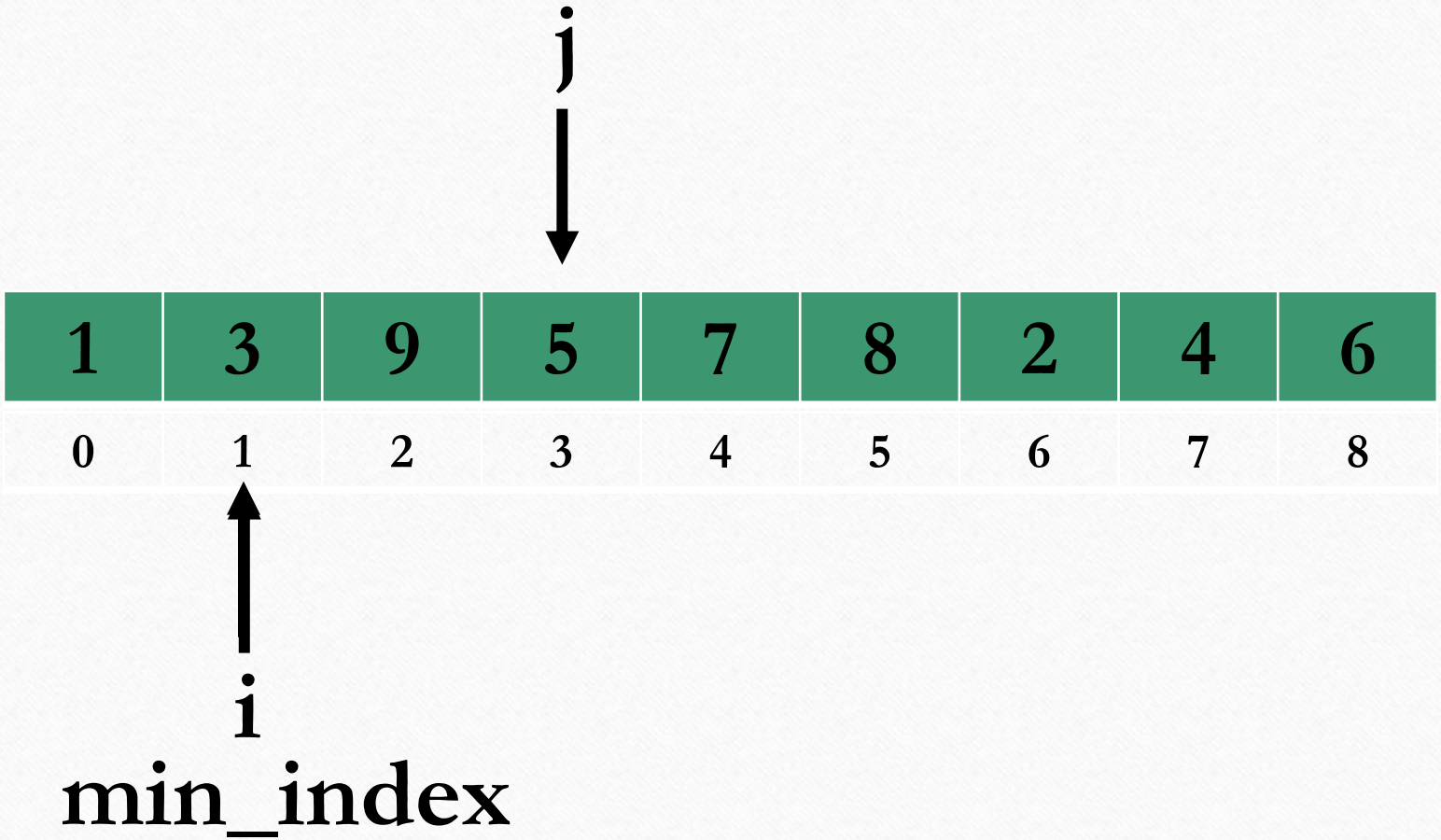
Second pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

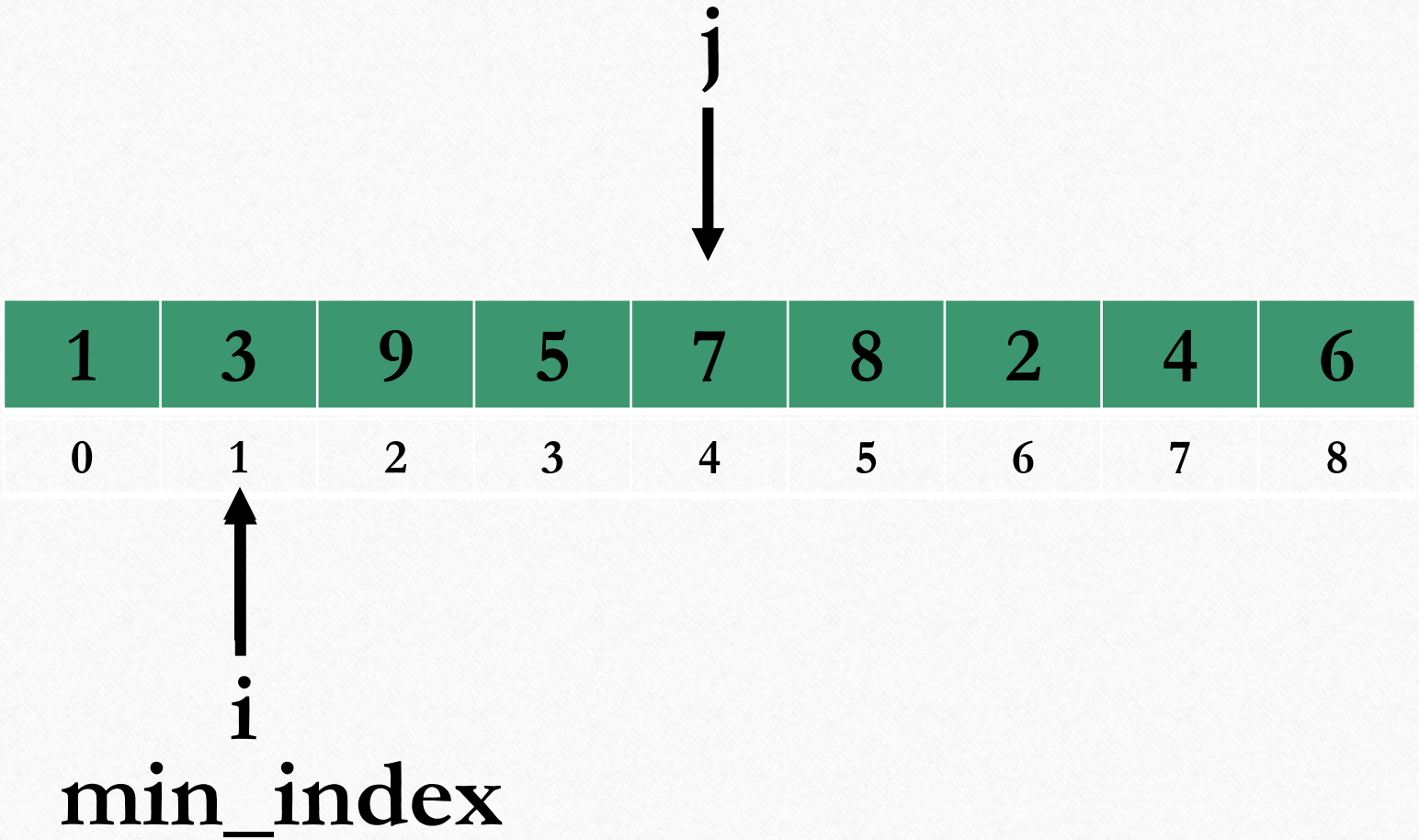
Second pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

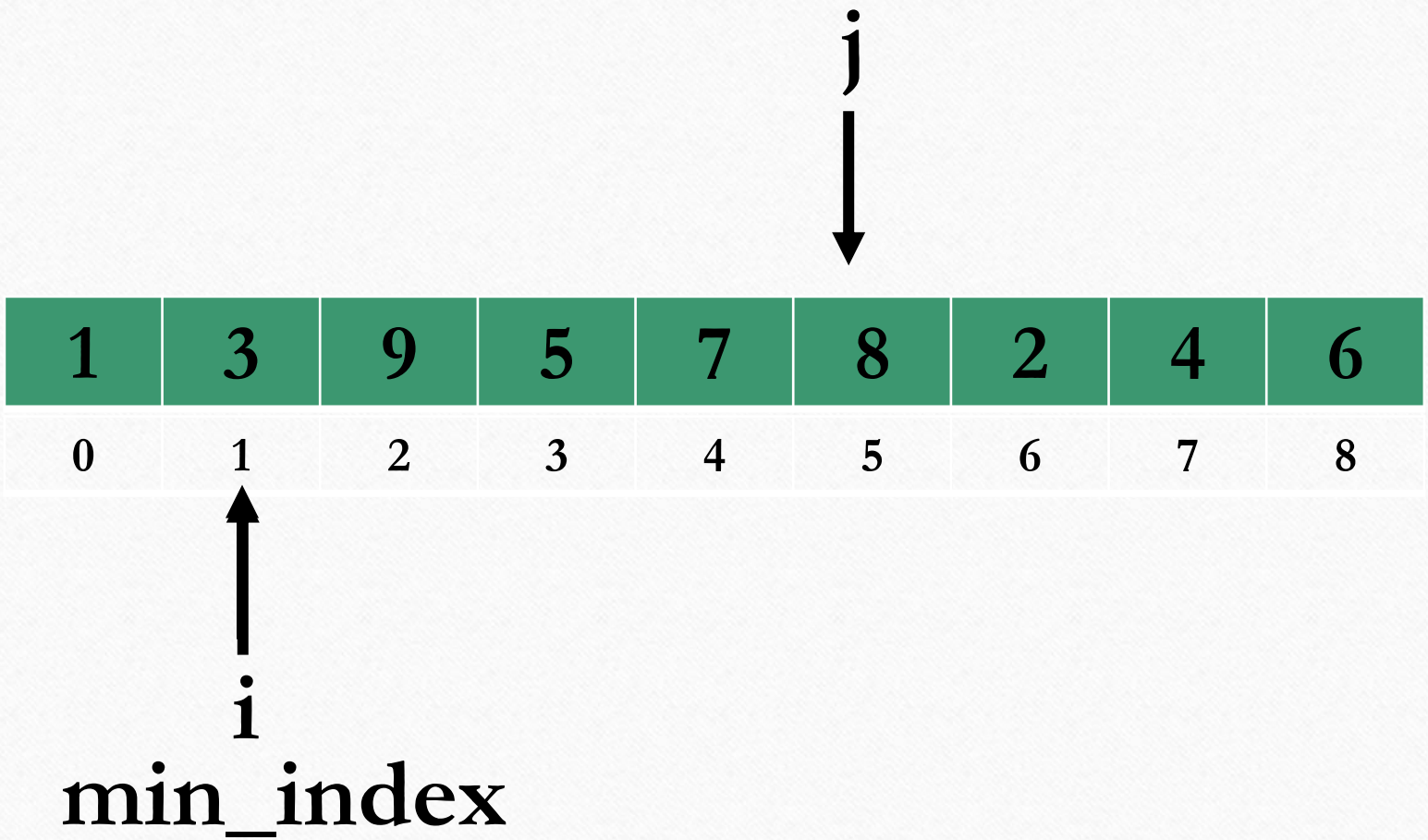
Second pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

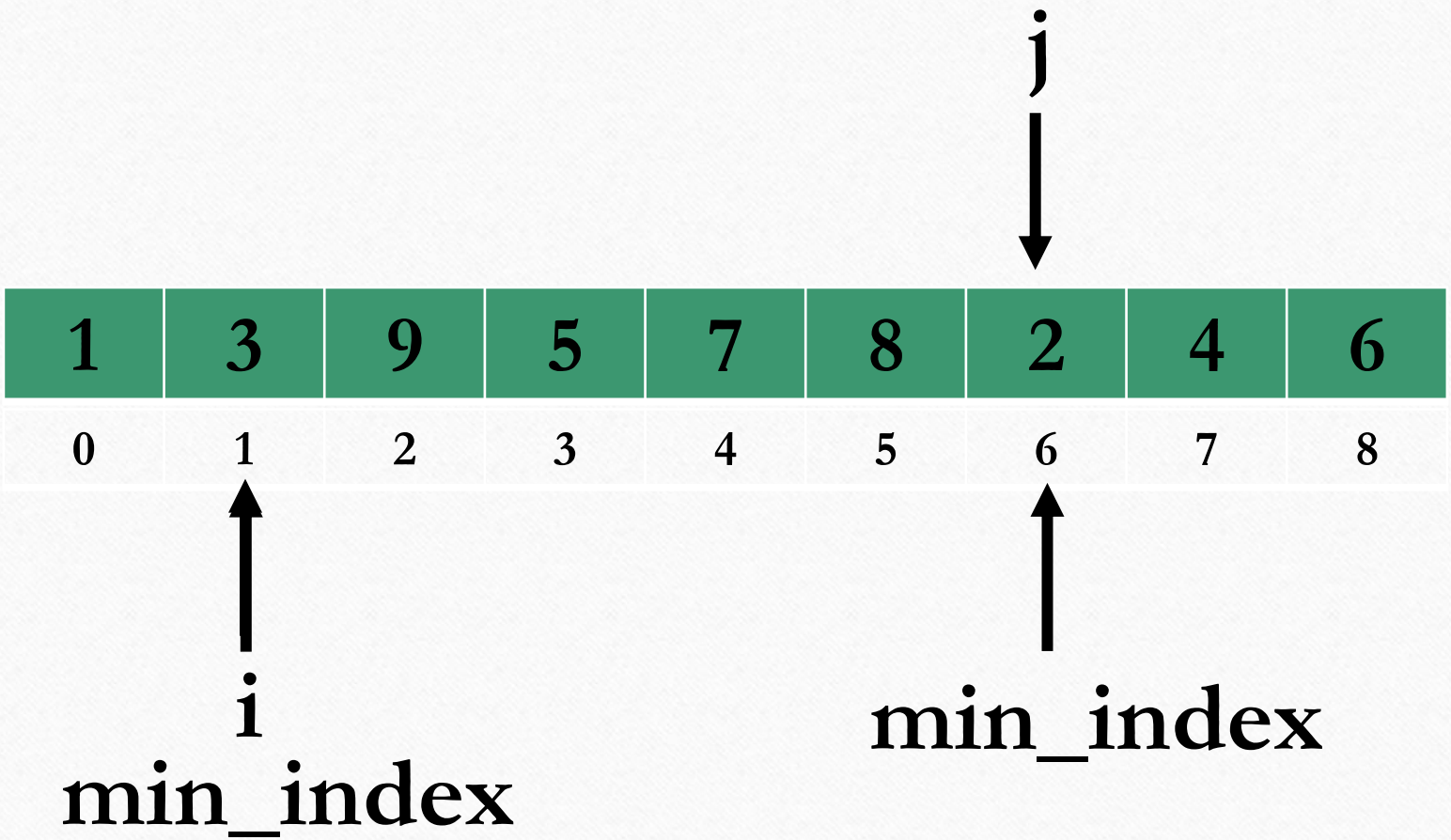
Second pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

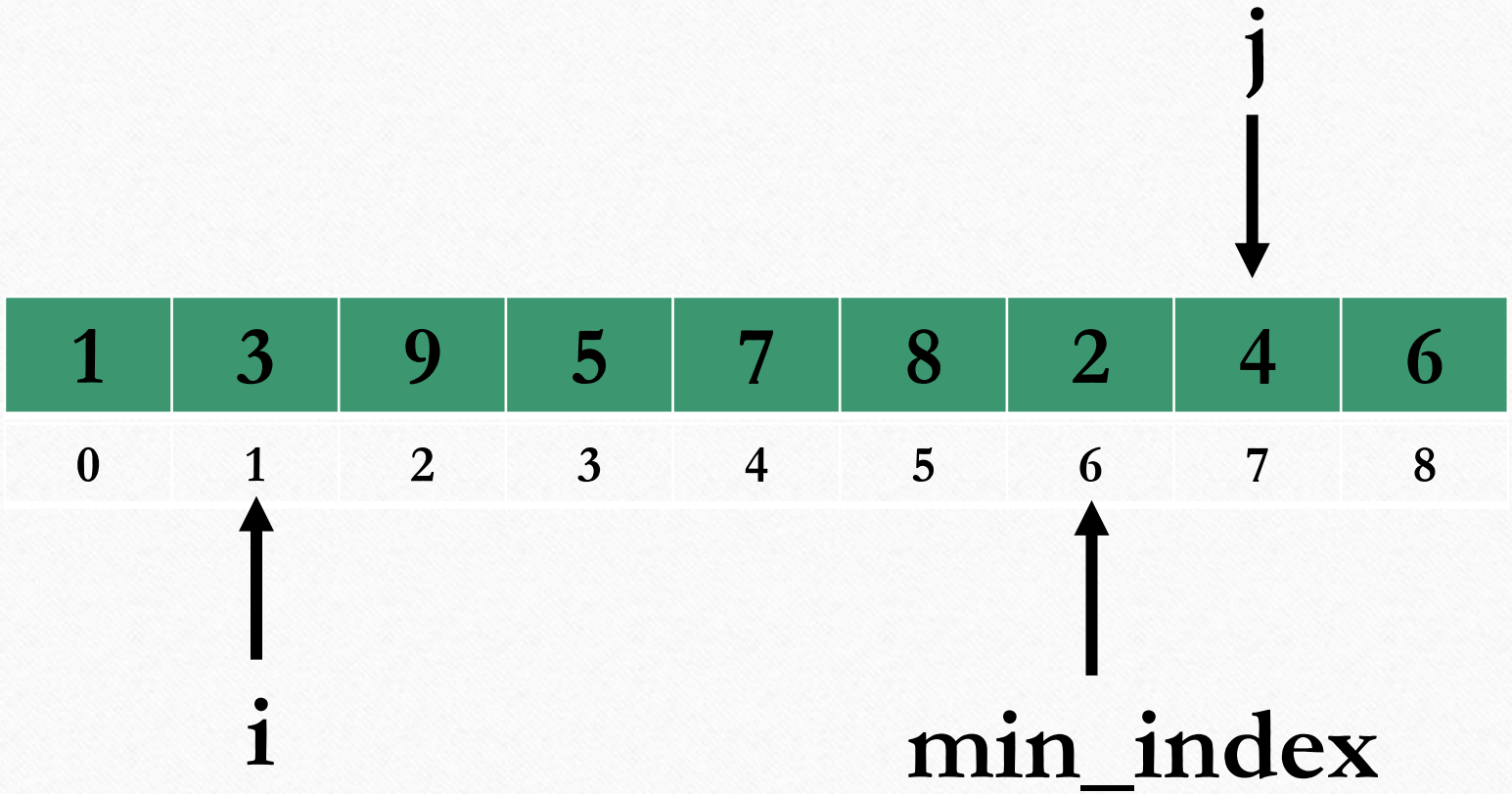
Second pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

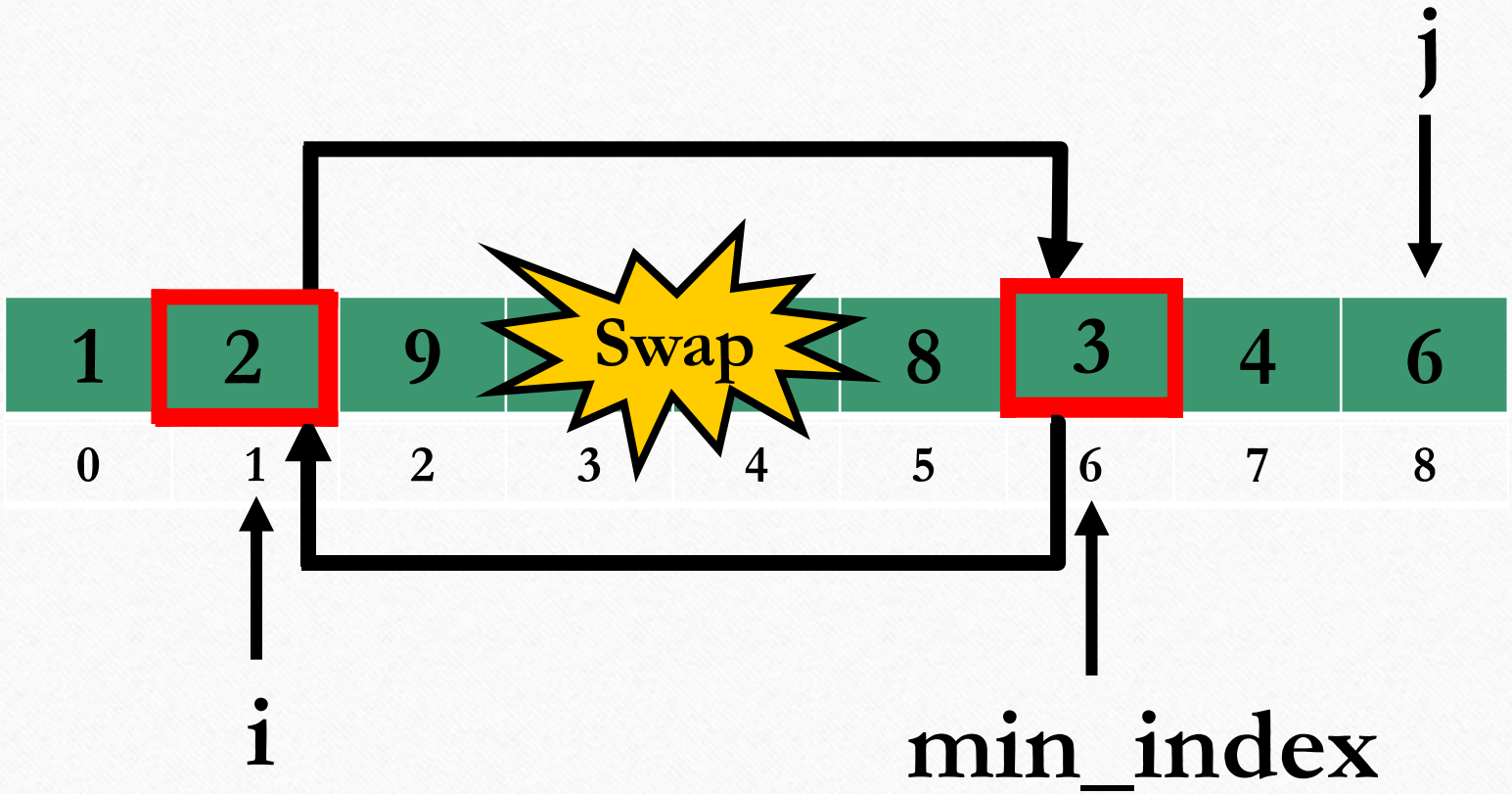
Second pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

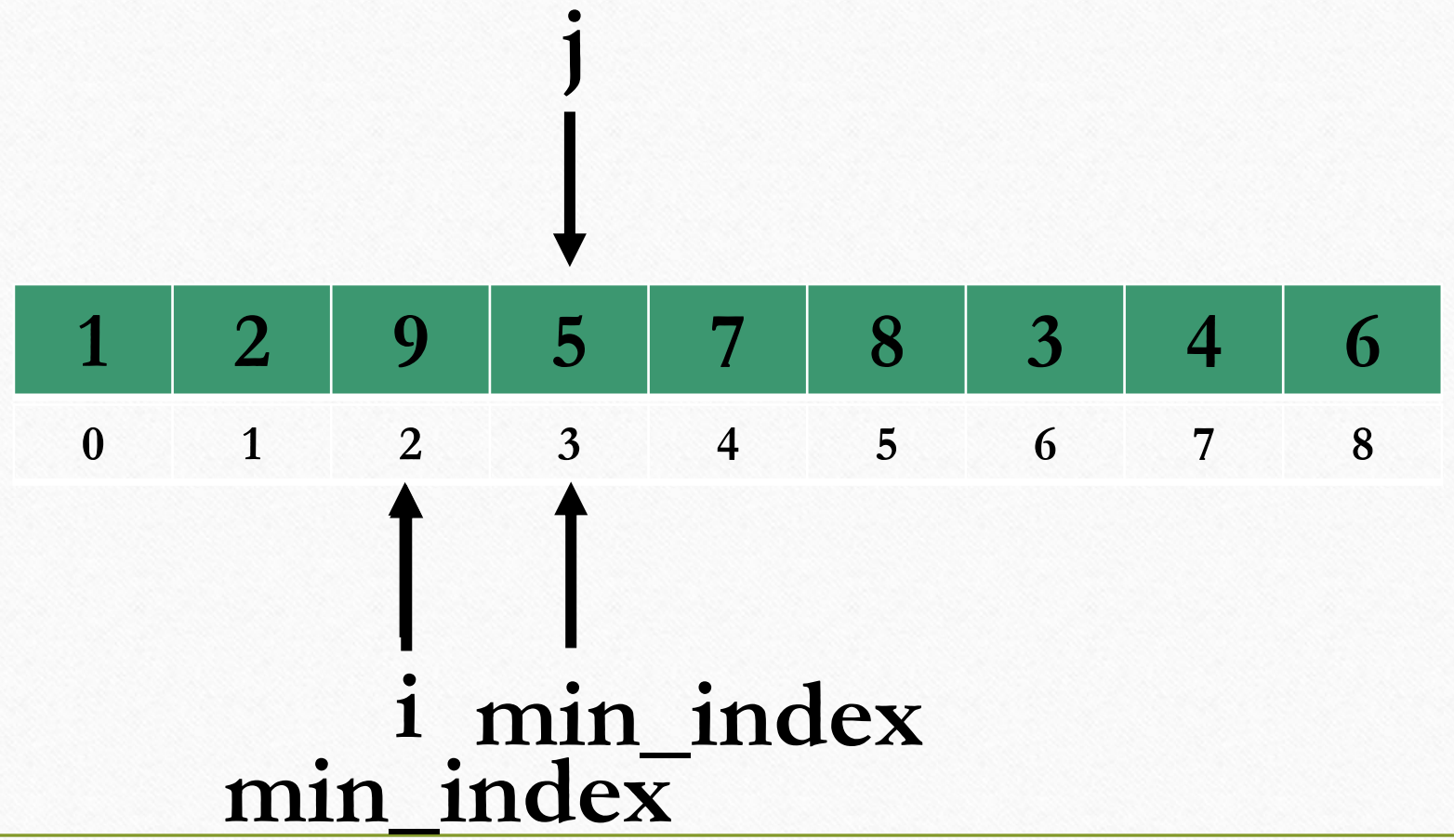
Second pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

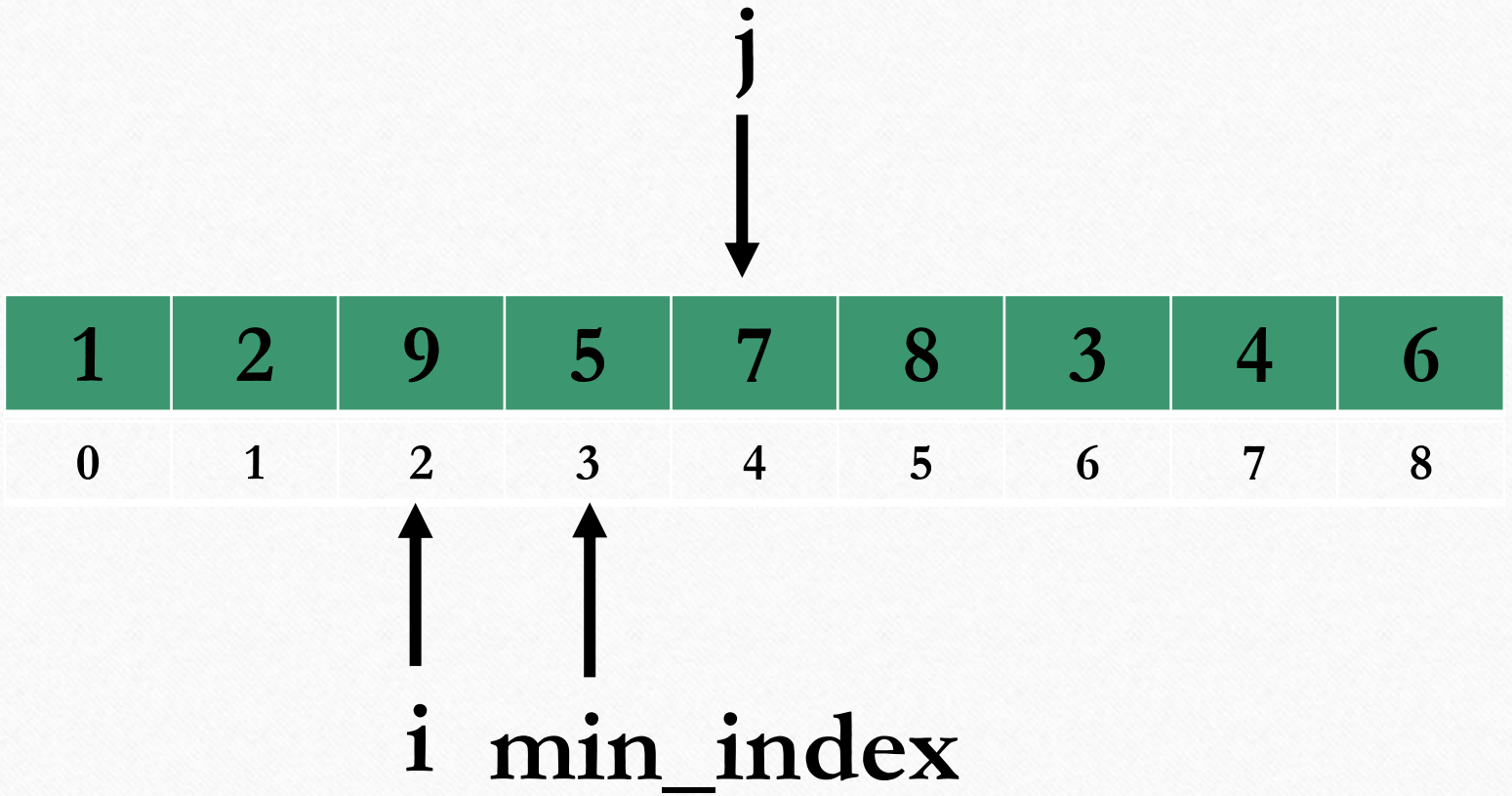
3rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

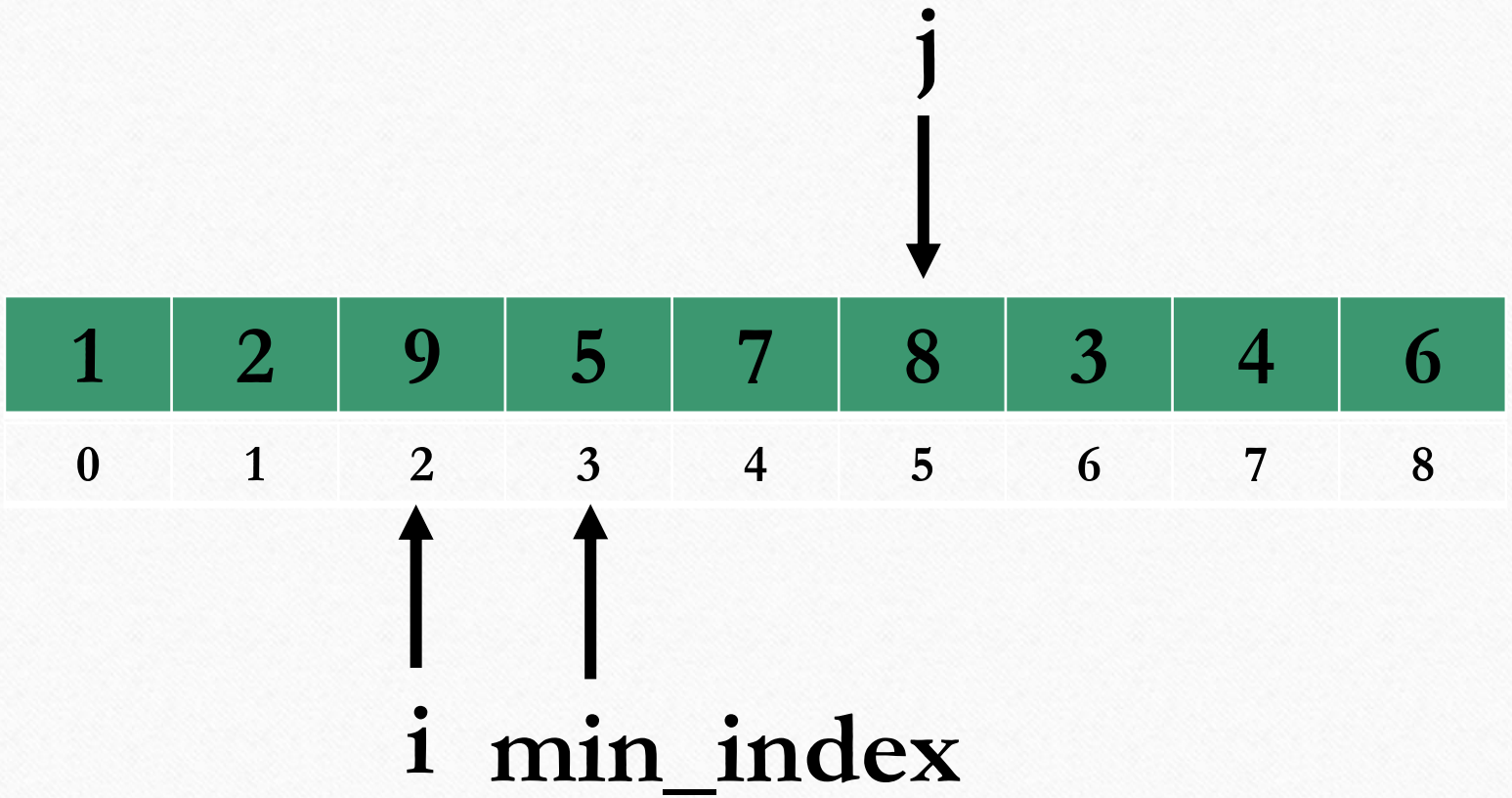
3rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

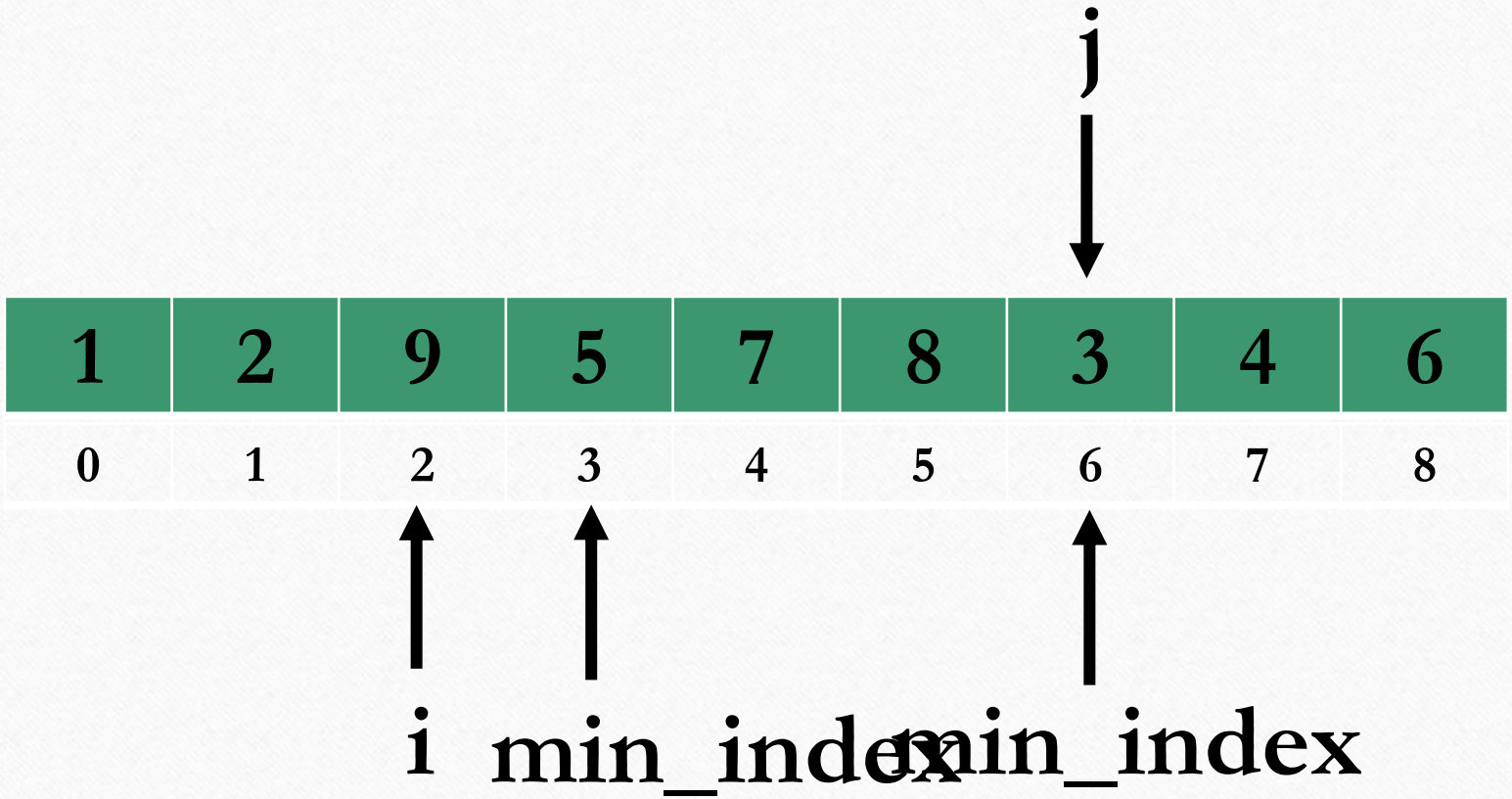
3rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

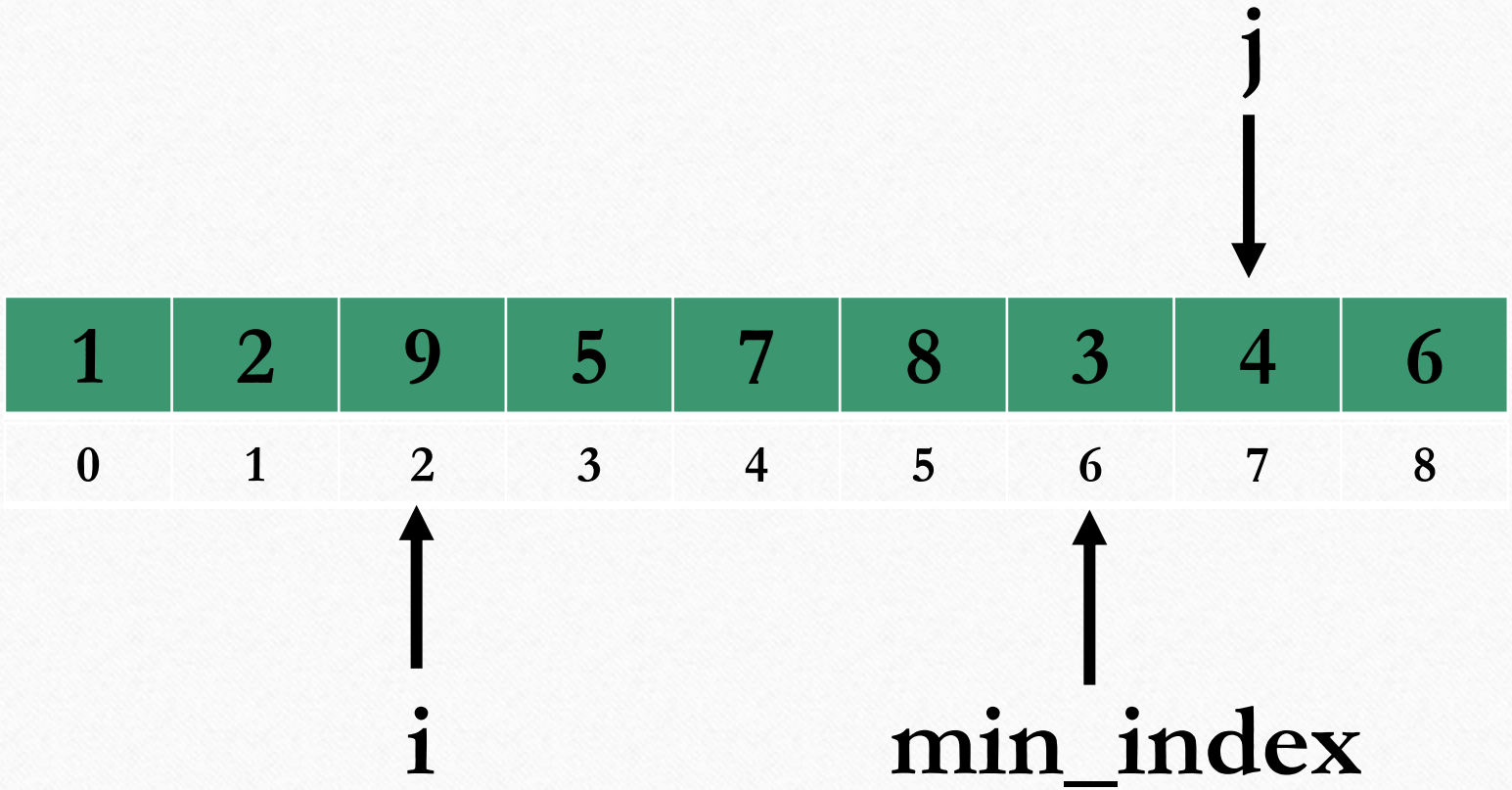
3rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

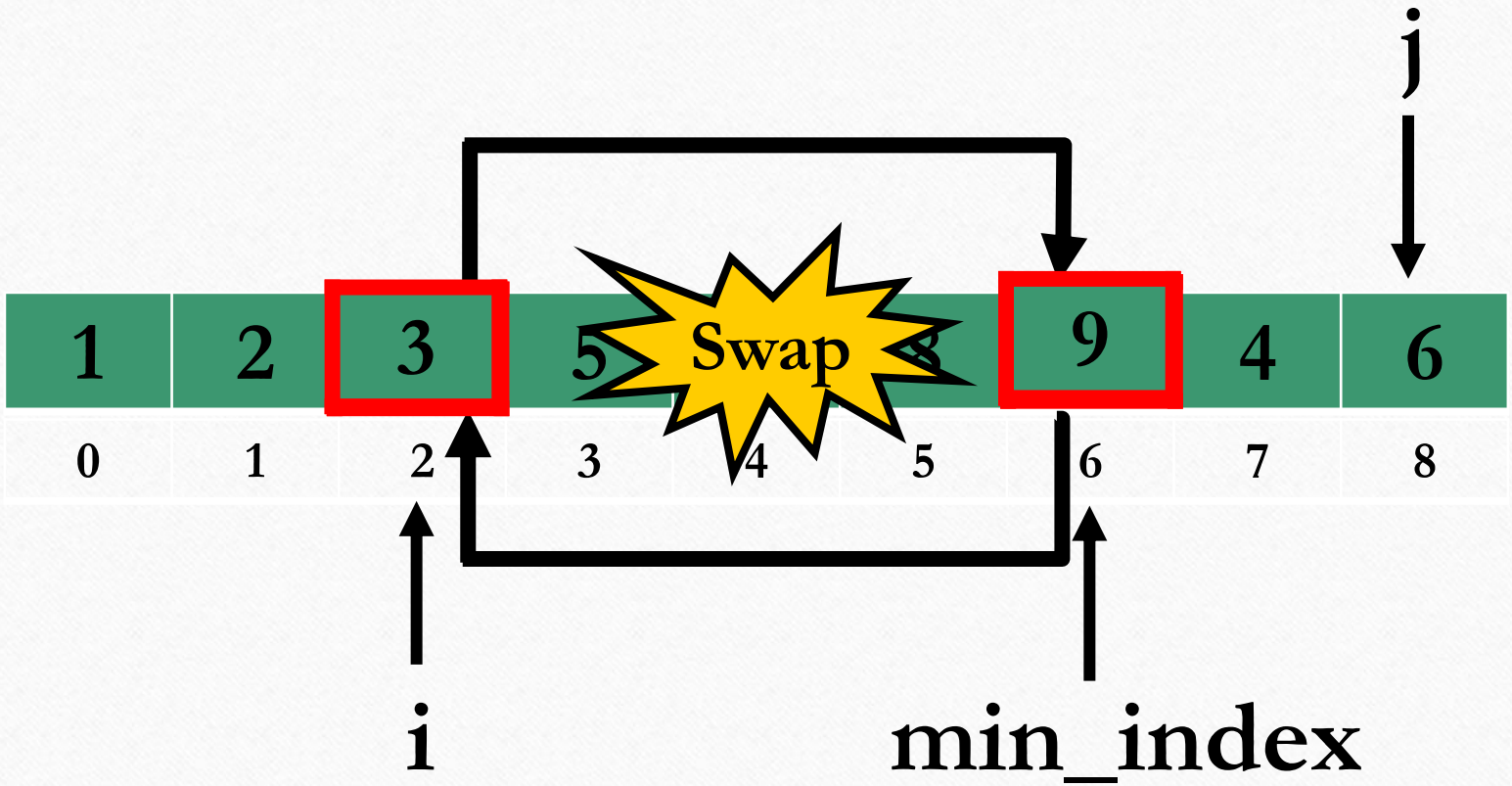
3rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

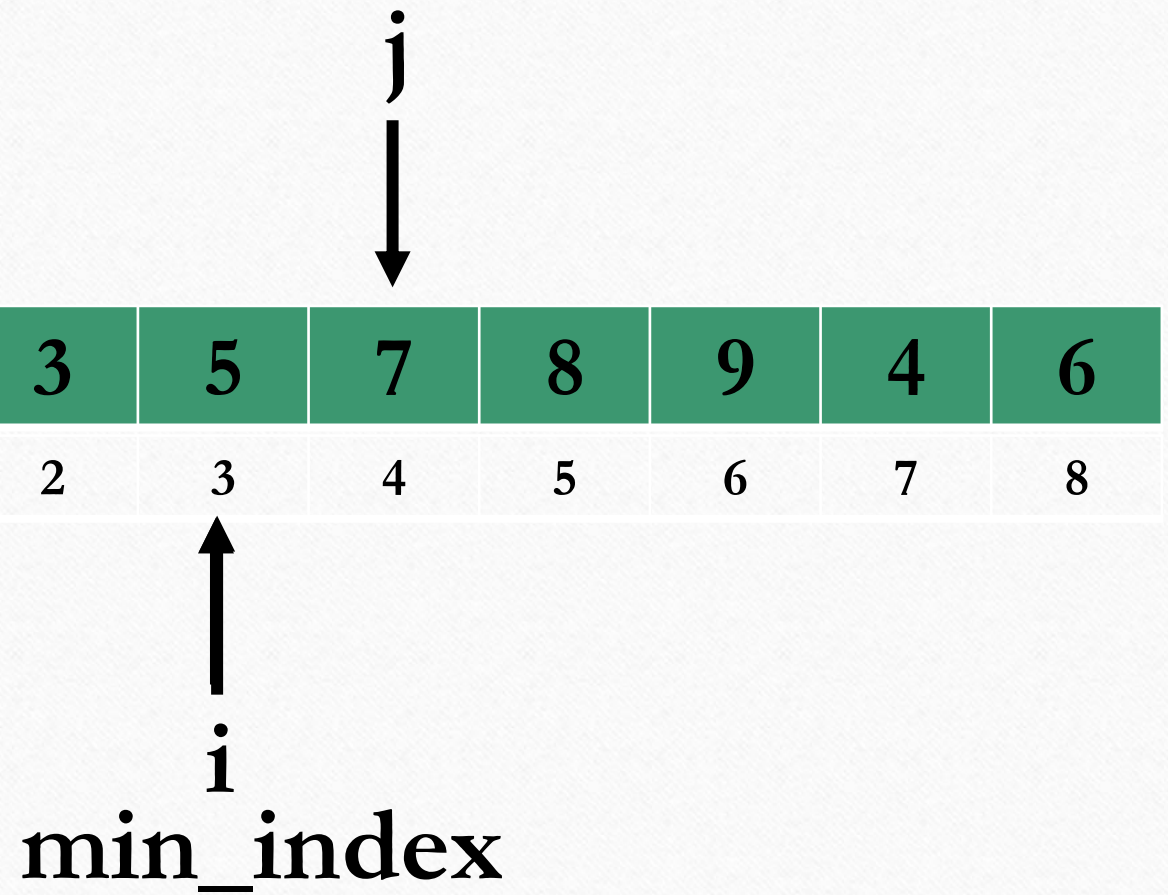
3rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

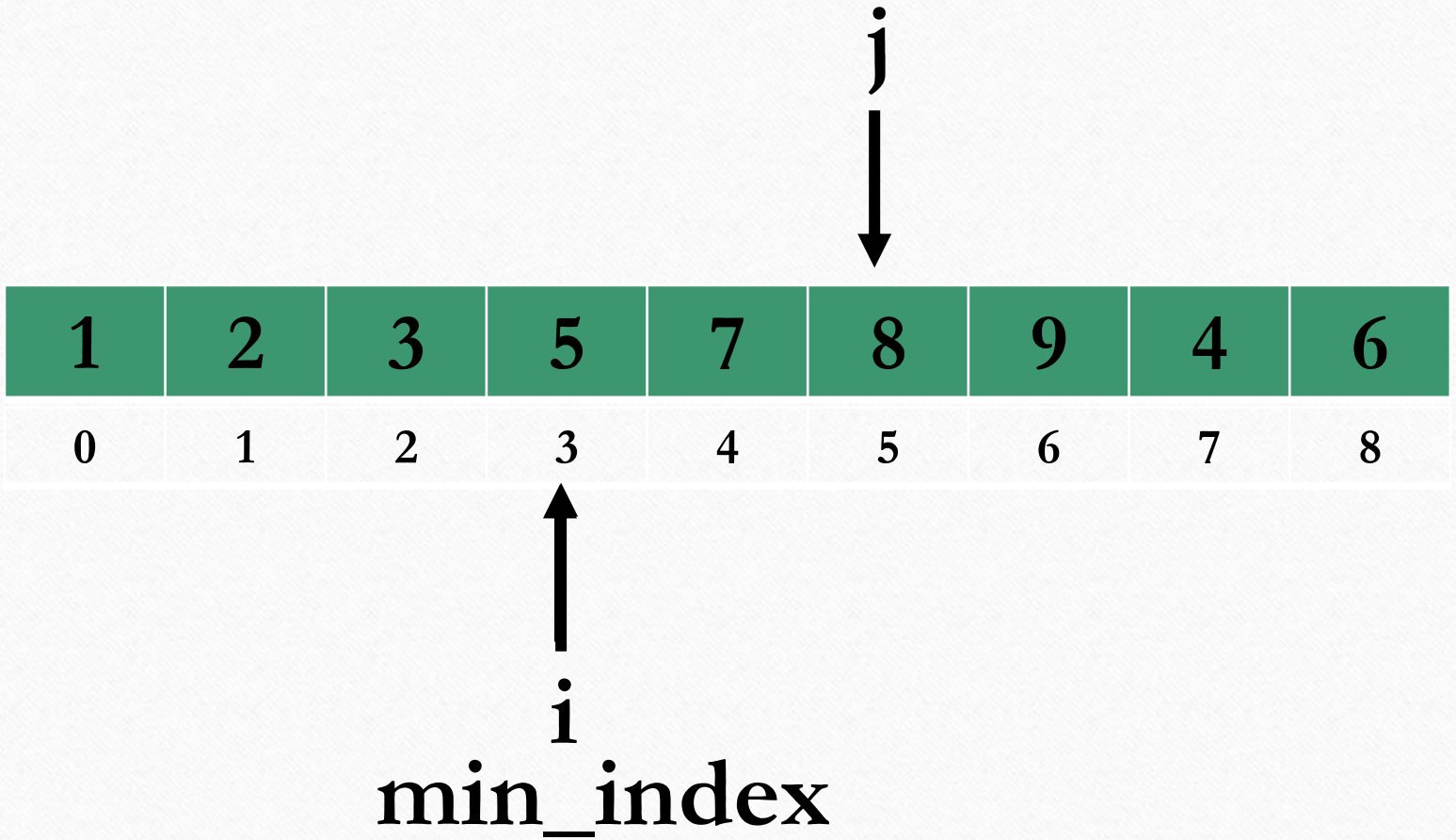
4rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

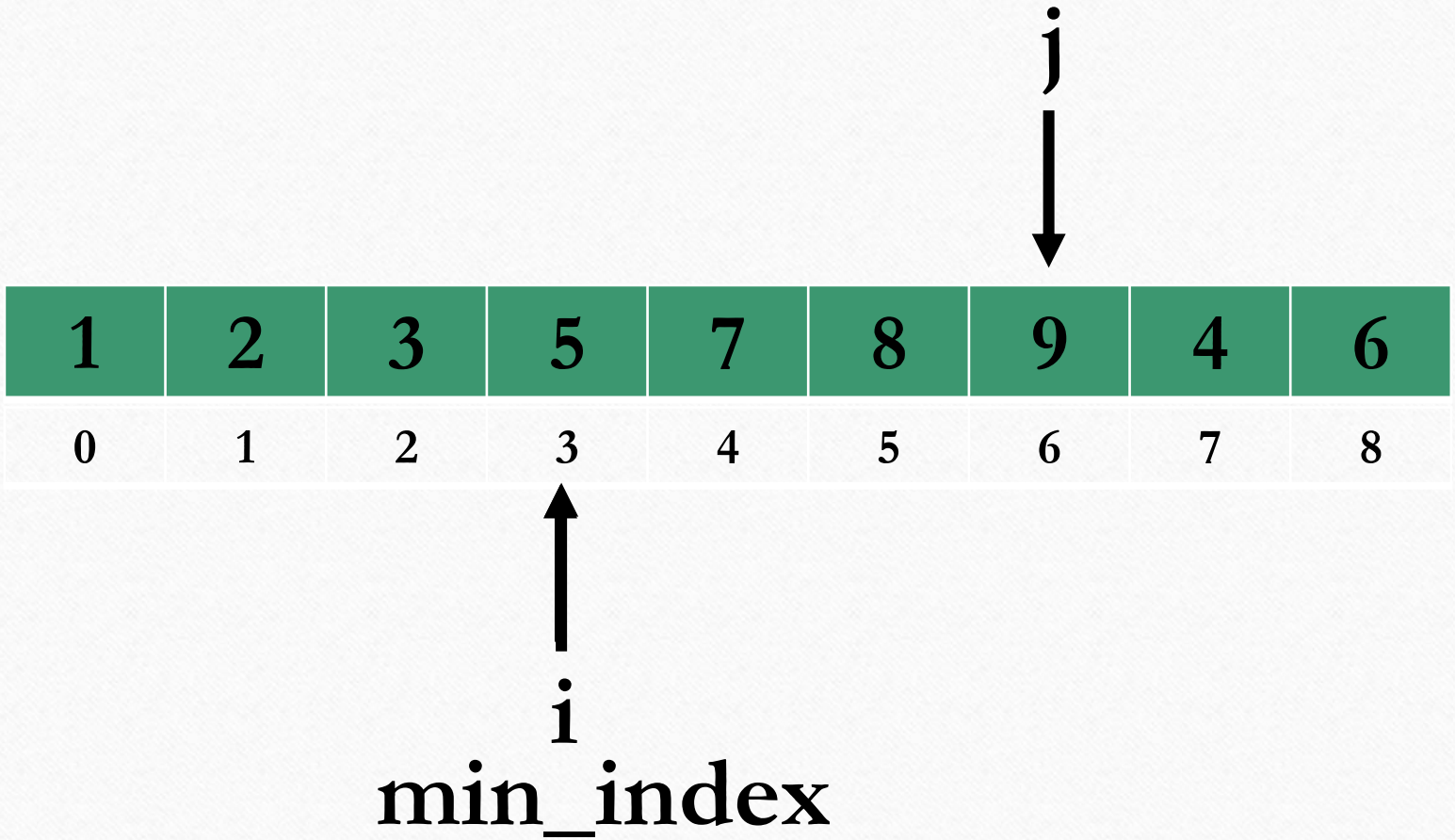
4rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

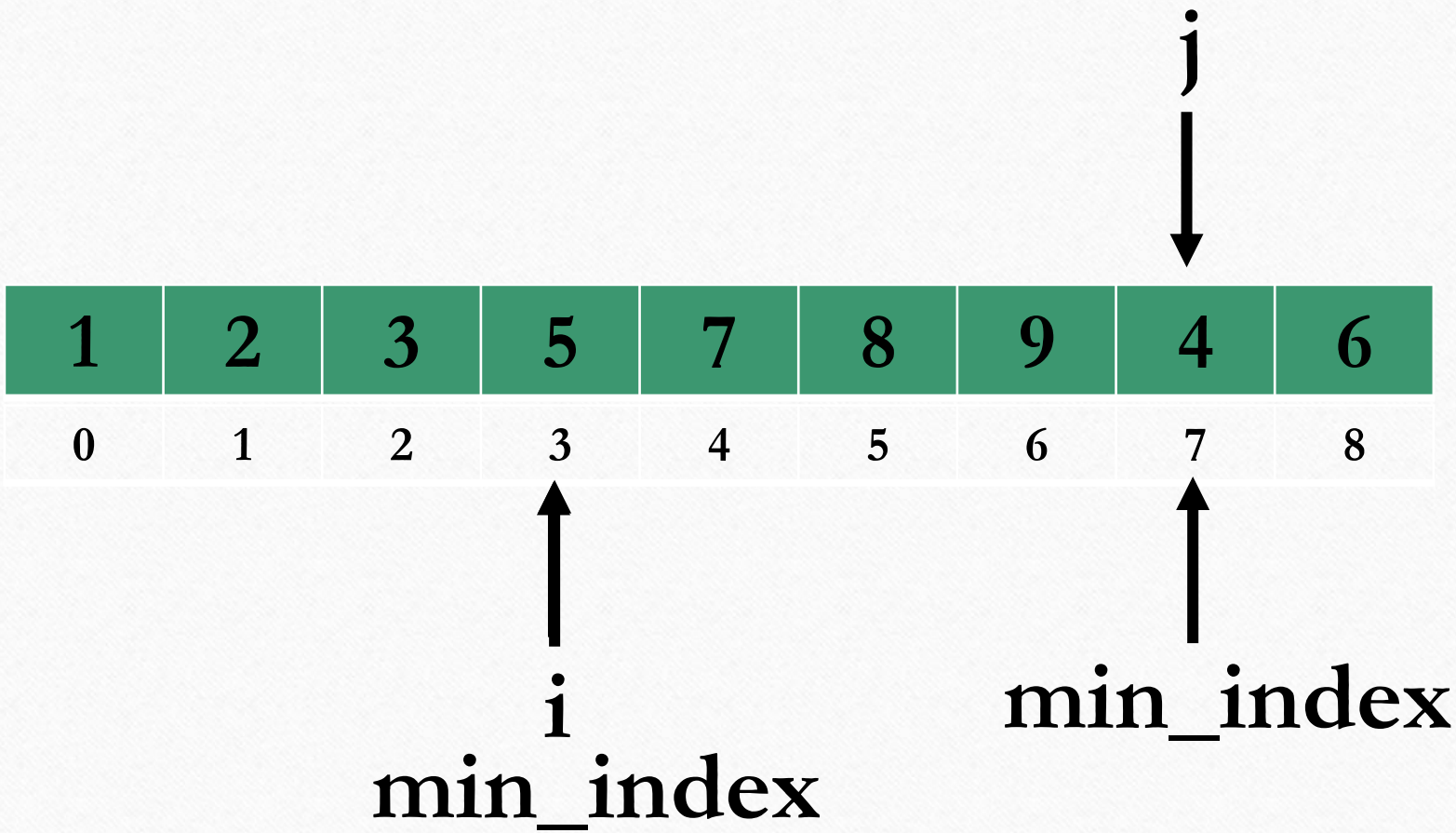
4rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

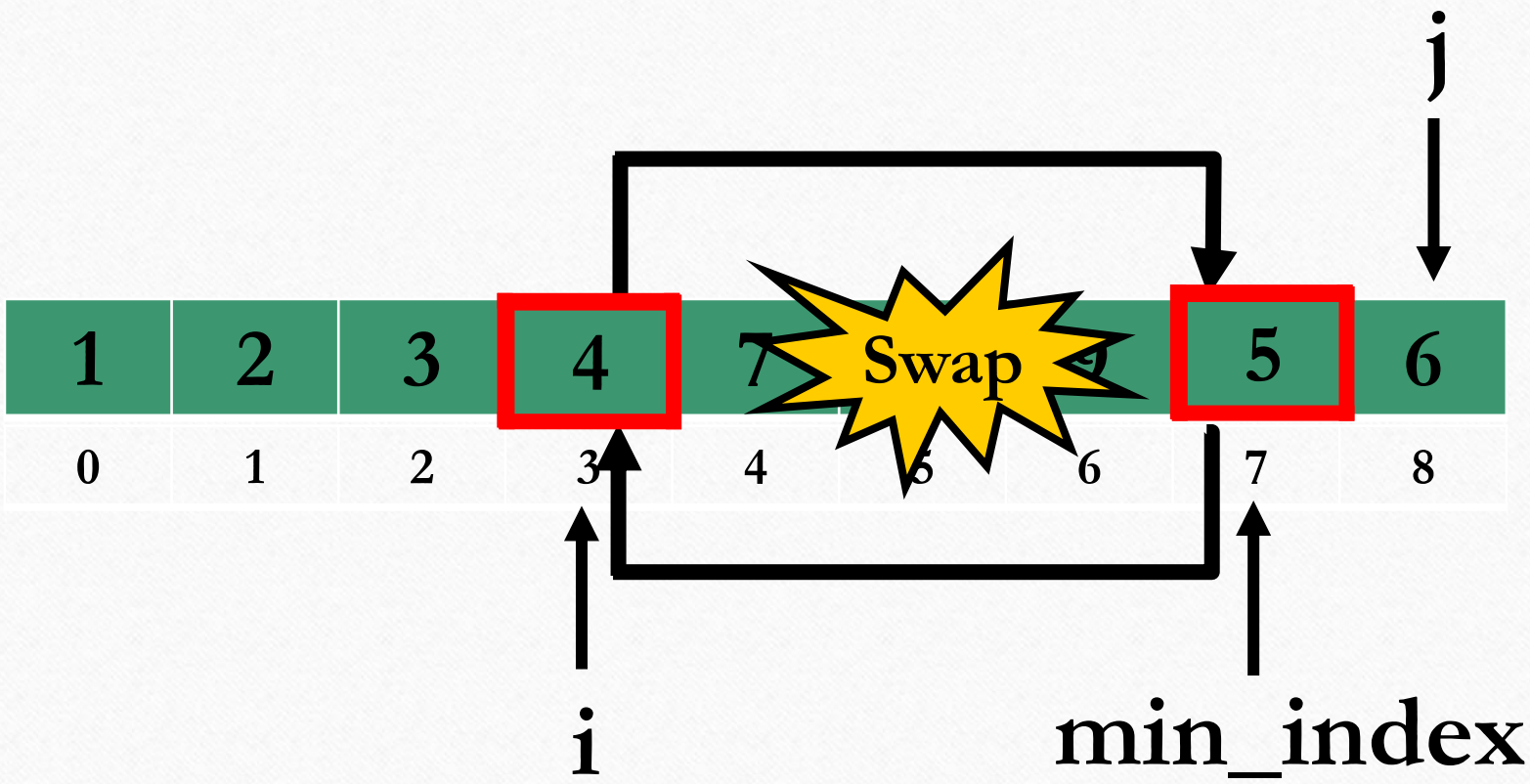
4rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

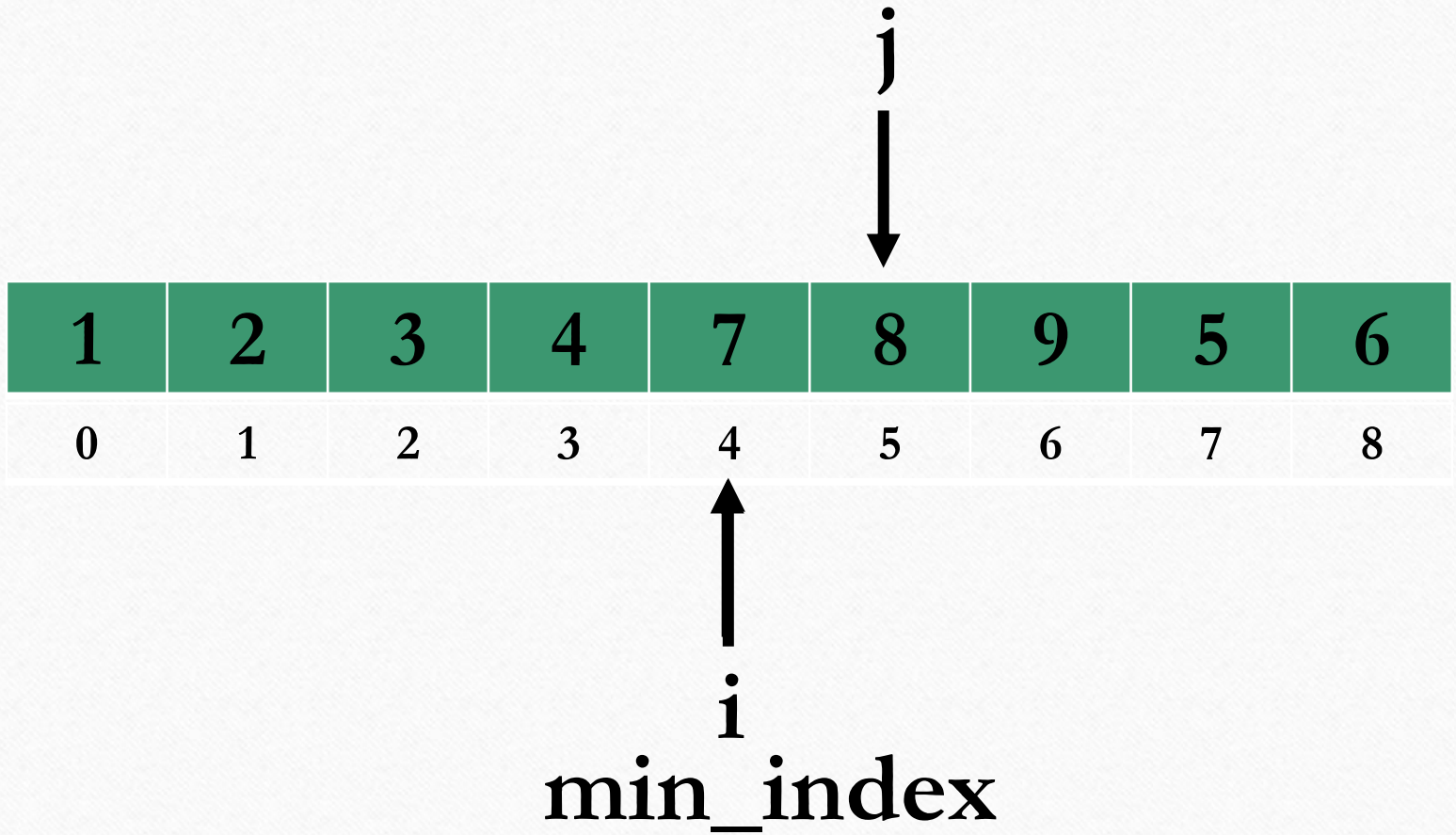
4rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

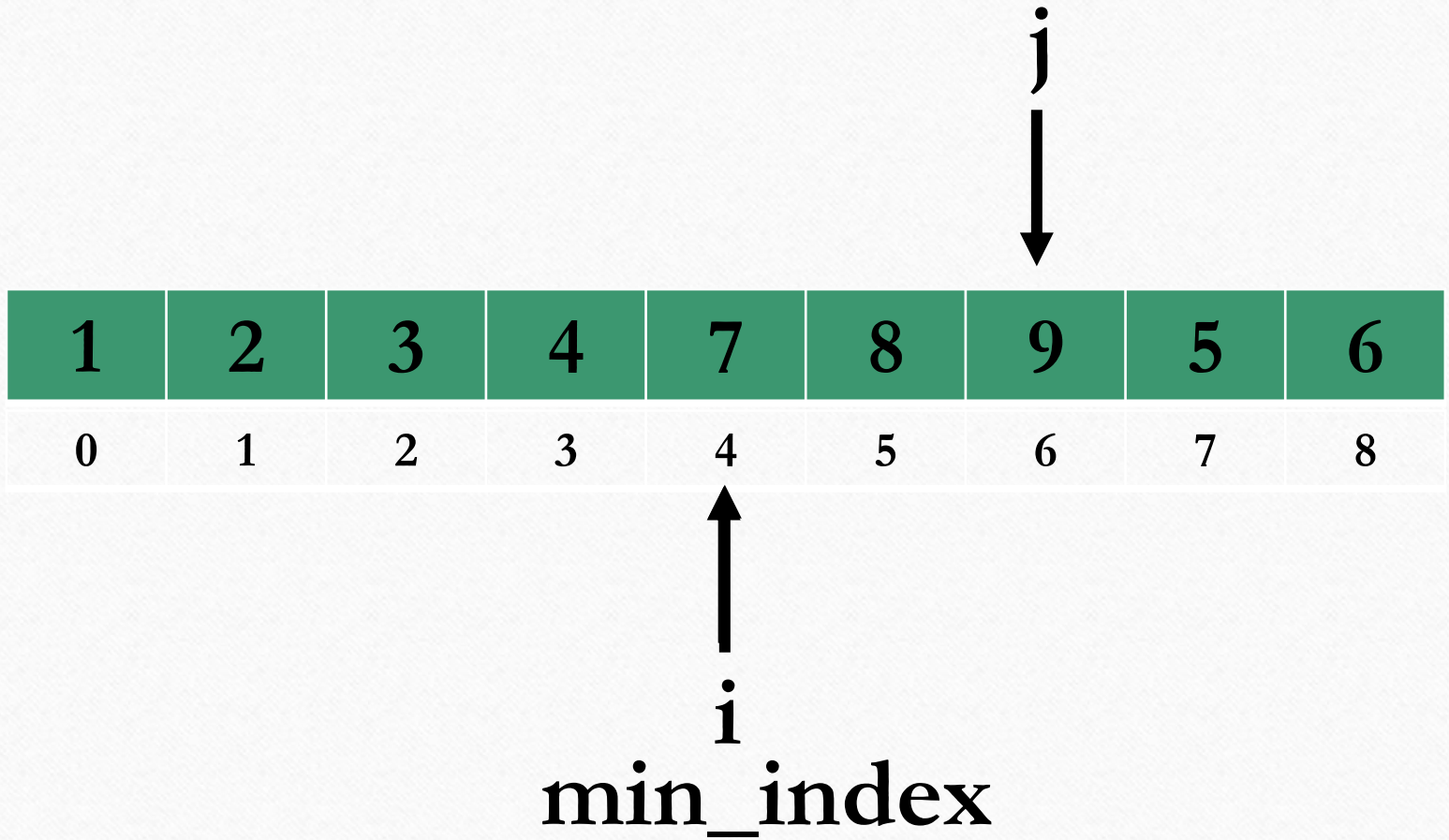
5rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

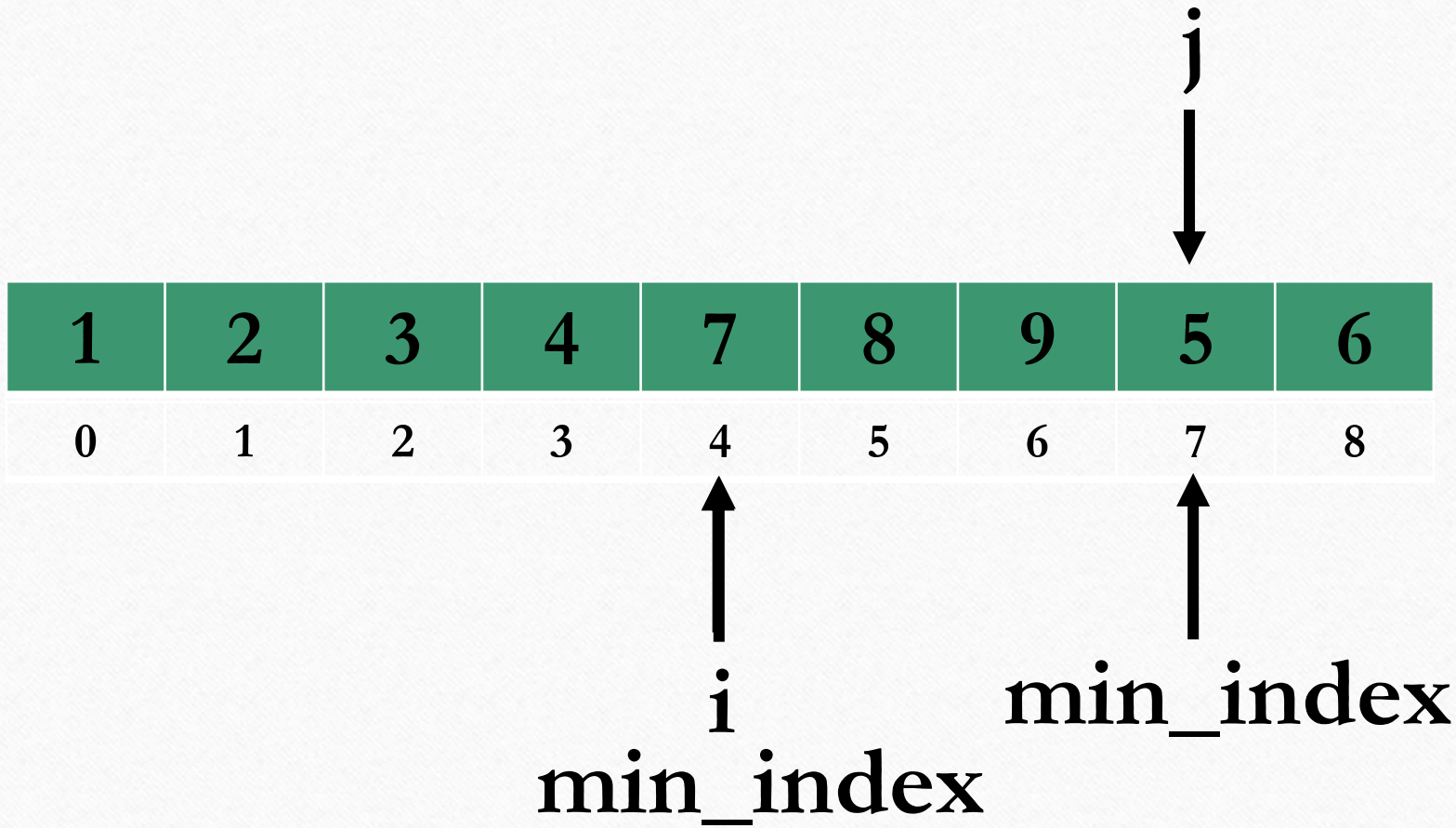
5rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

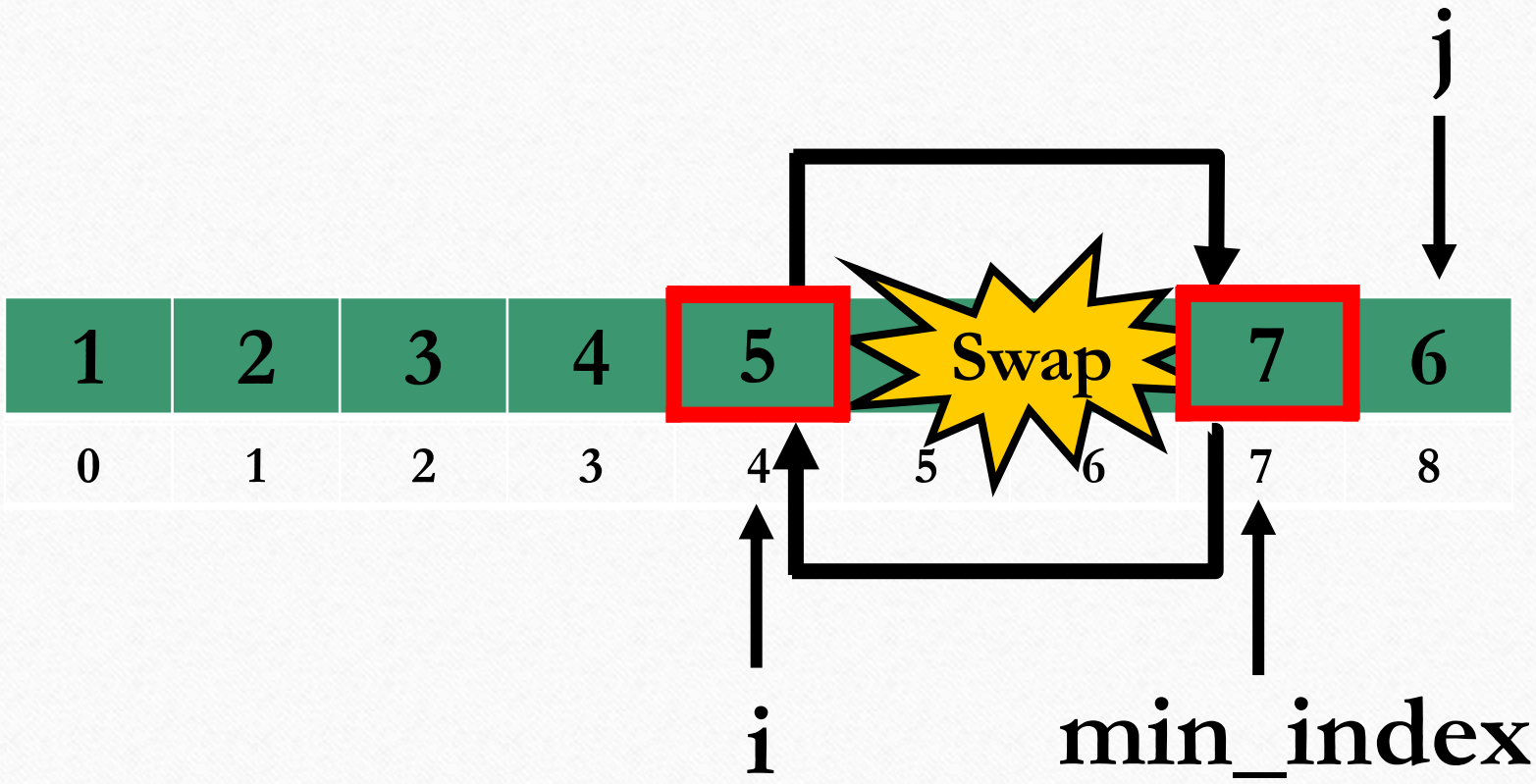
5rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

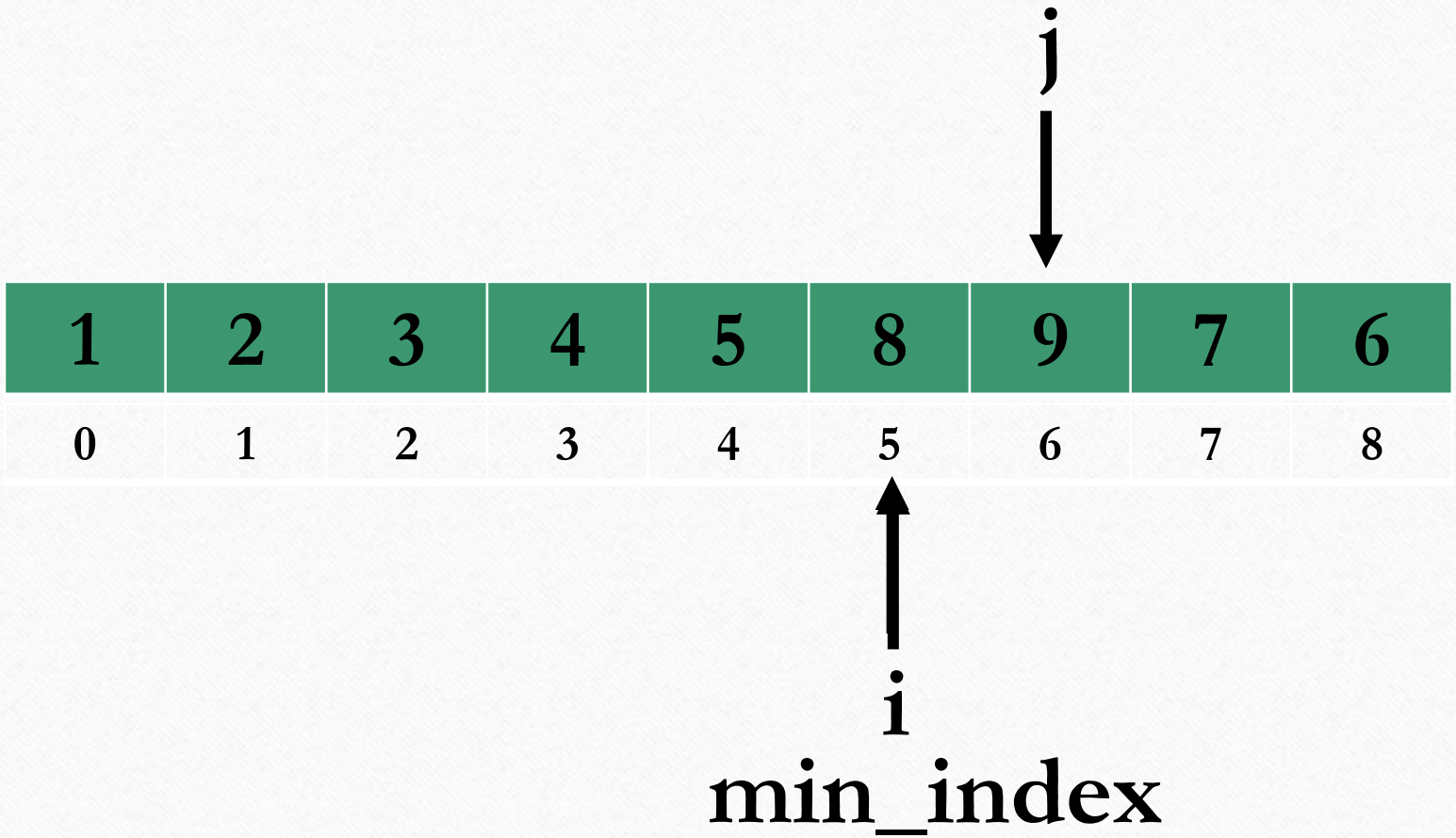
5rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

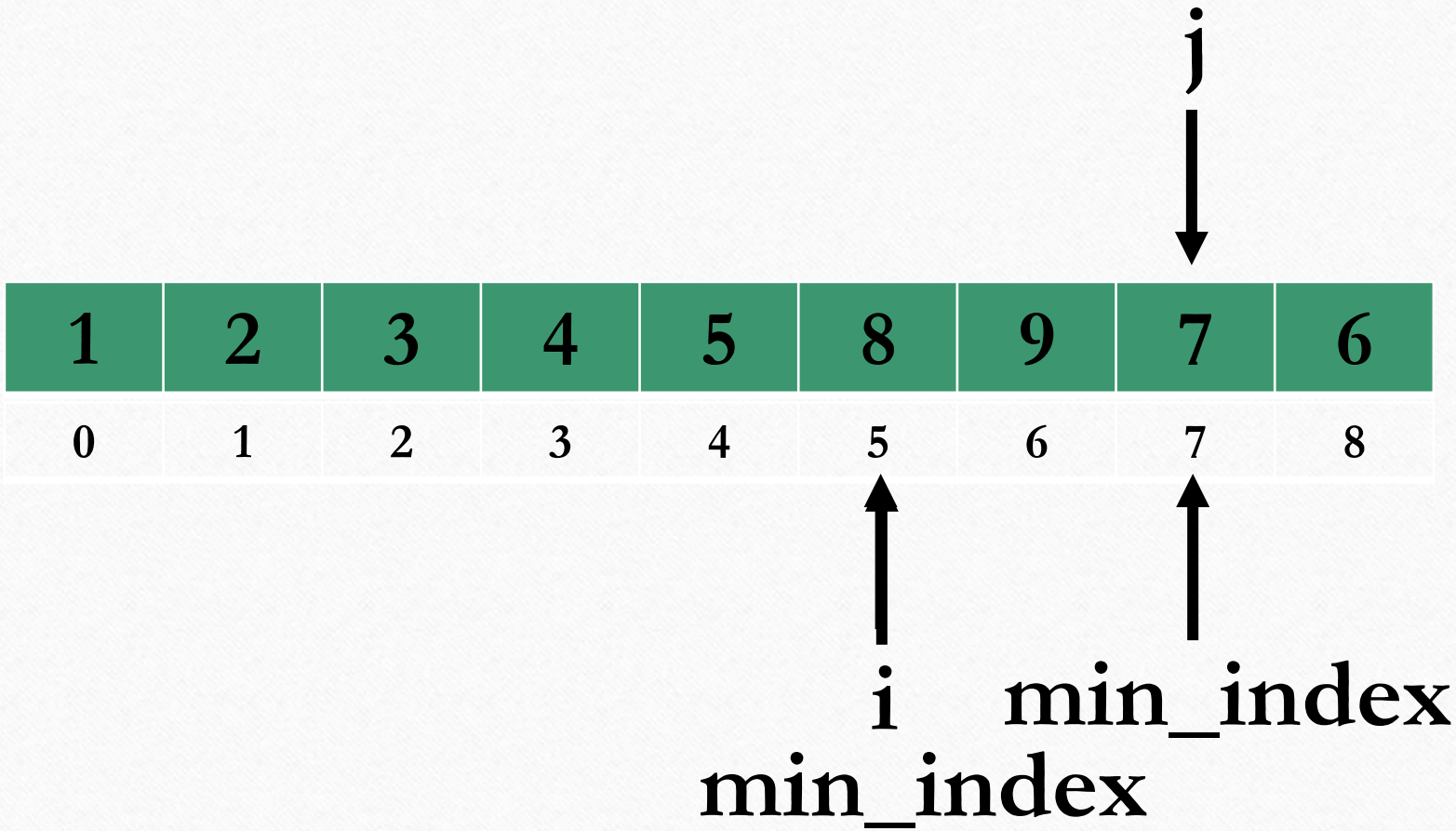
6rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

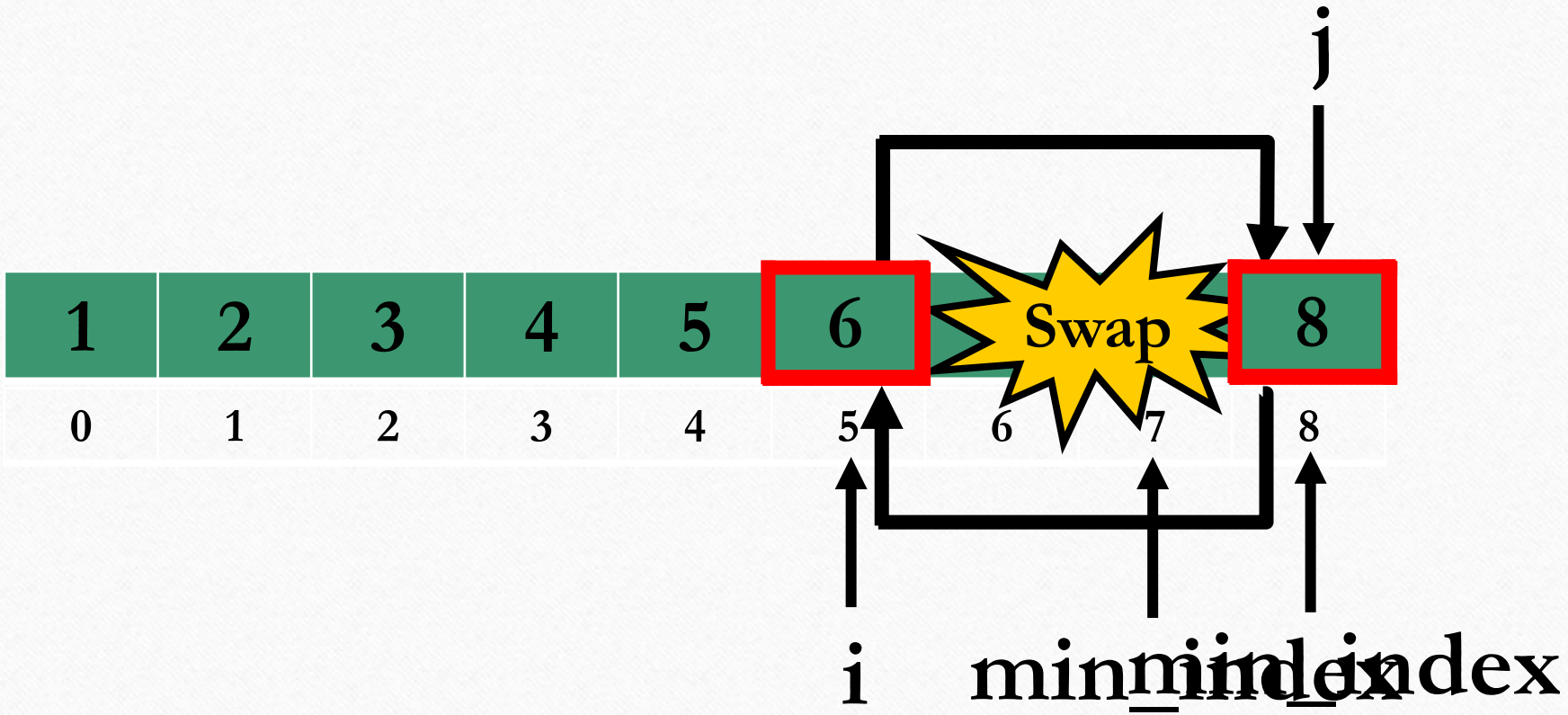
6rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

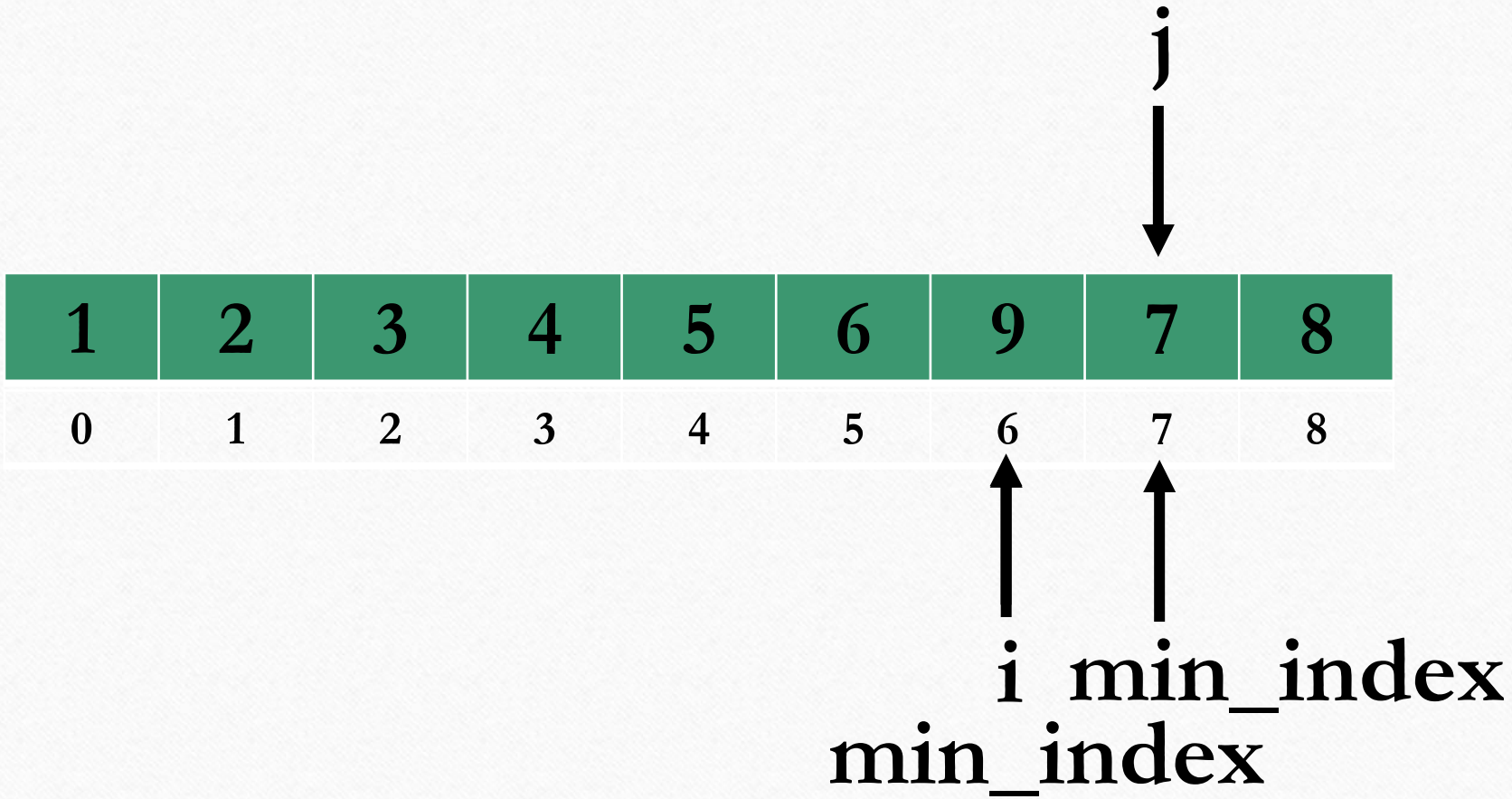
6rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

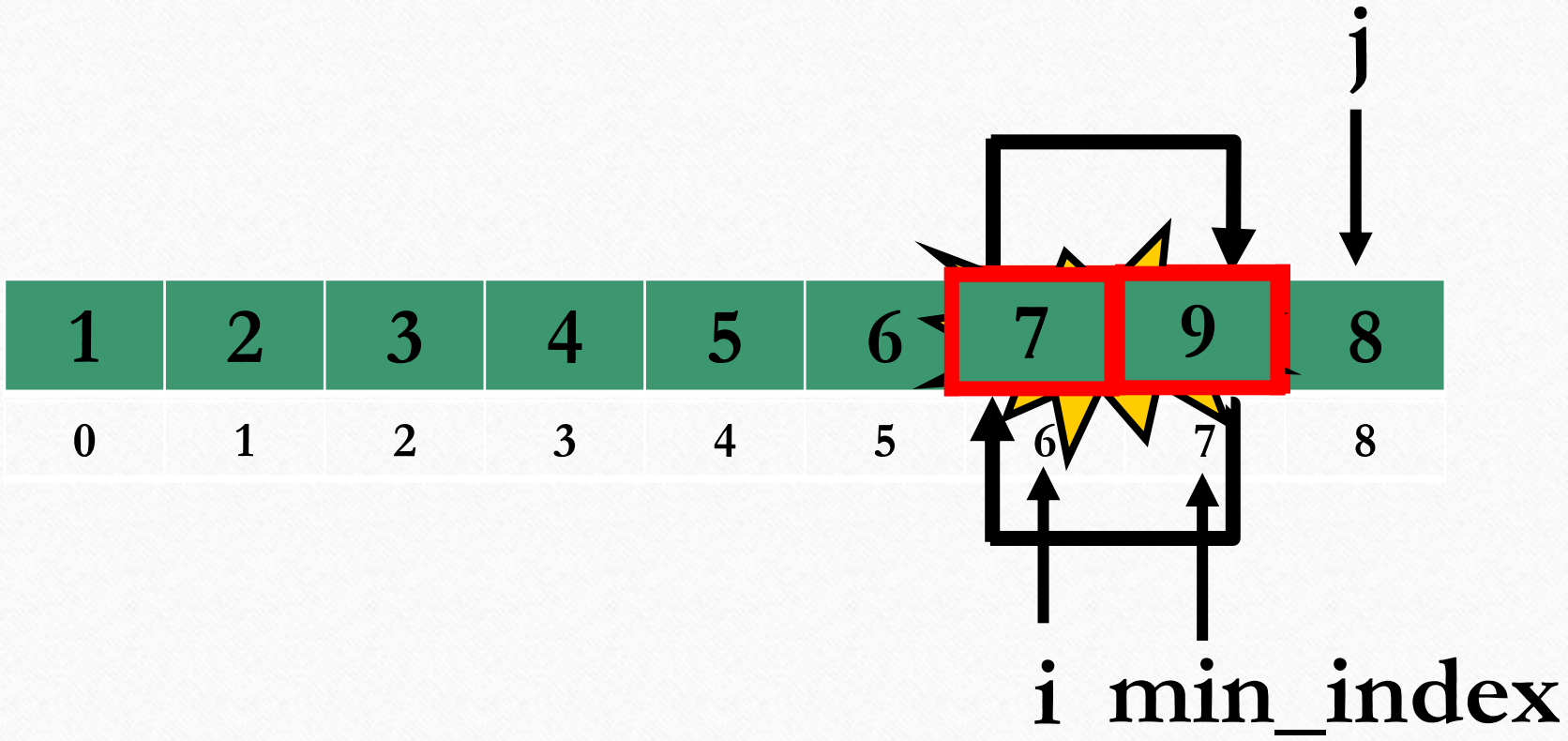
7rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

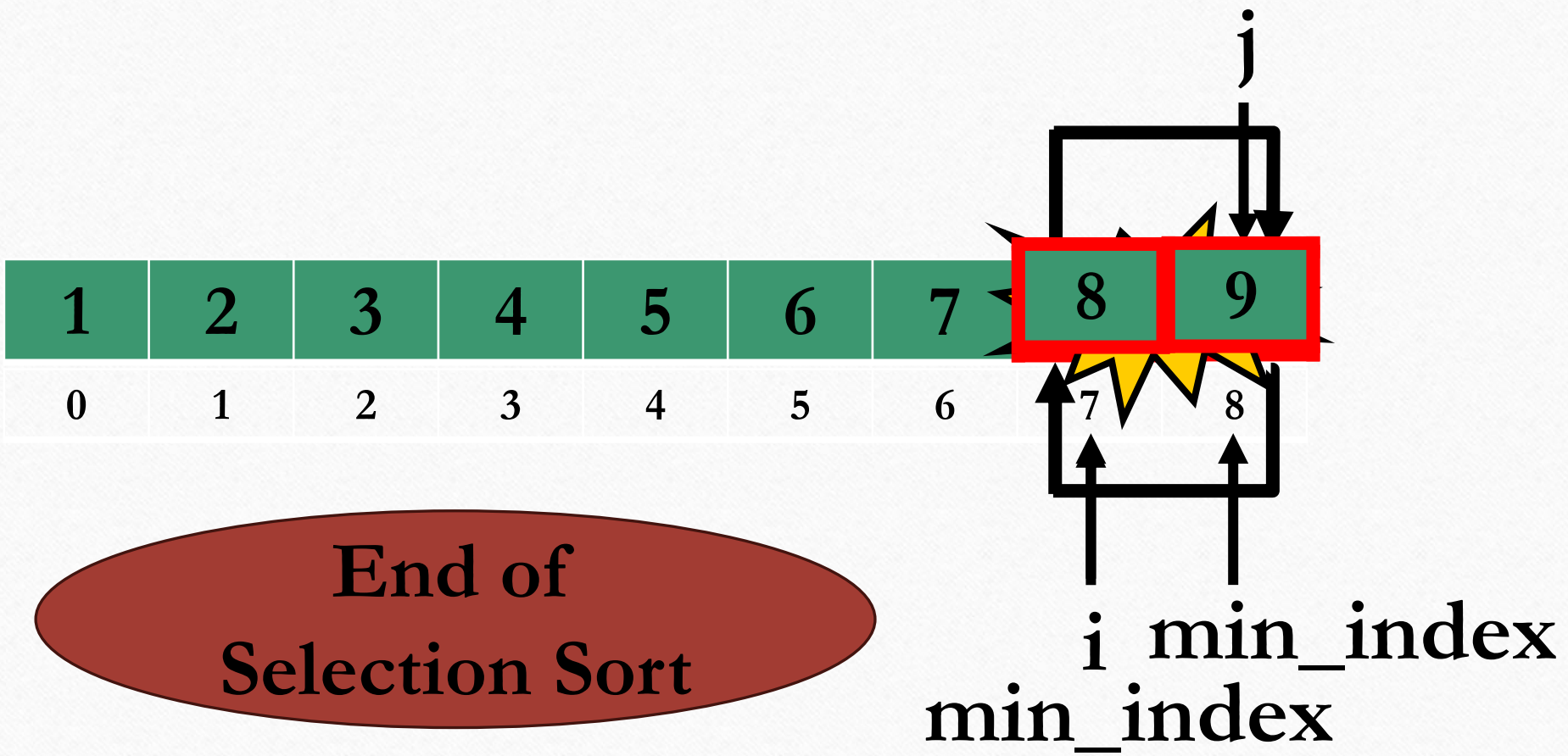
7rd pass



```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (Vec[j] < Vec[min_index]) {
                min_index = j;
            }
        swap(Vec[i], Vec[min_index]);
}
```

V. Selection Sort

8rd pass



End of Selection Sort

V. Selection Sort

Time Complexity of Selection Sort

```
void selection_sort(int Vec[], int n) {
   for (int i = 0; i < n - 1; i++) {
      int min_index = i;
      for (int j = i + 1; j < n; j++)
         if (Vec[j] < Vec[min_index]) {
            min_index = j;
         }
      swap(Vec[i], Vec[min_index]);
   }
}
```

i	j
0	N-1
1	N-2
2	N-3
.....	....
.....	....
N-2	1

V. Selection Sort

Time Complexity of Selection Sort

- Similar to bubble sort, we need to make $\boldsymbol{n-1}$ comparisons on the first pass, $\boldsymbol{n-2}$ comparisons on the second pass, and so on. Thus, the total number of comparisons can be found by:

$$\boldsymbol{T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)}$$

- Therefore, our selection sort implementation runs in $\boldsymbol{\mathcal{O}(n^2)}$ time.

VI. Insertion Sort

- This is the sorting **technique most commonly used by card players.**
- It consists of examining the cards one by one and inserting them in their place among those already examined (thus sorting them).
- Insertion sorting is a **simple sorting** algorithm that constructs the final sorted array (or list) **one item at a time**.

VI. Insertion Sort

- The list is **divided into two parts**, one composed of sorted items and the other of unsorted items.
- Each item of the unsorted list is considered one by one and **inserted into** the sorted list at **its appropriate position**.
- In each pass, the sorted list is **traversed in the backward direction** to find the position where the unsorted item can be inserted.

VI. Insertion Sort

- The implementation of insertion sorting is illustrated below:

```
void insertion_sort(int Vec[], int n) {
   for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
         Vec[j + 1] = Vec[j];
         j--;
      }
      Vec[j + 1] = current;
   }
}
```

VI. Insertion Sort

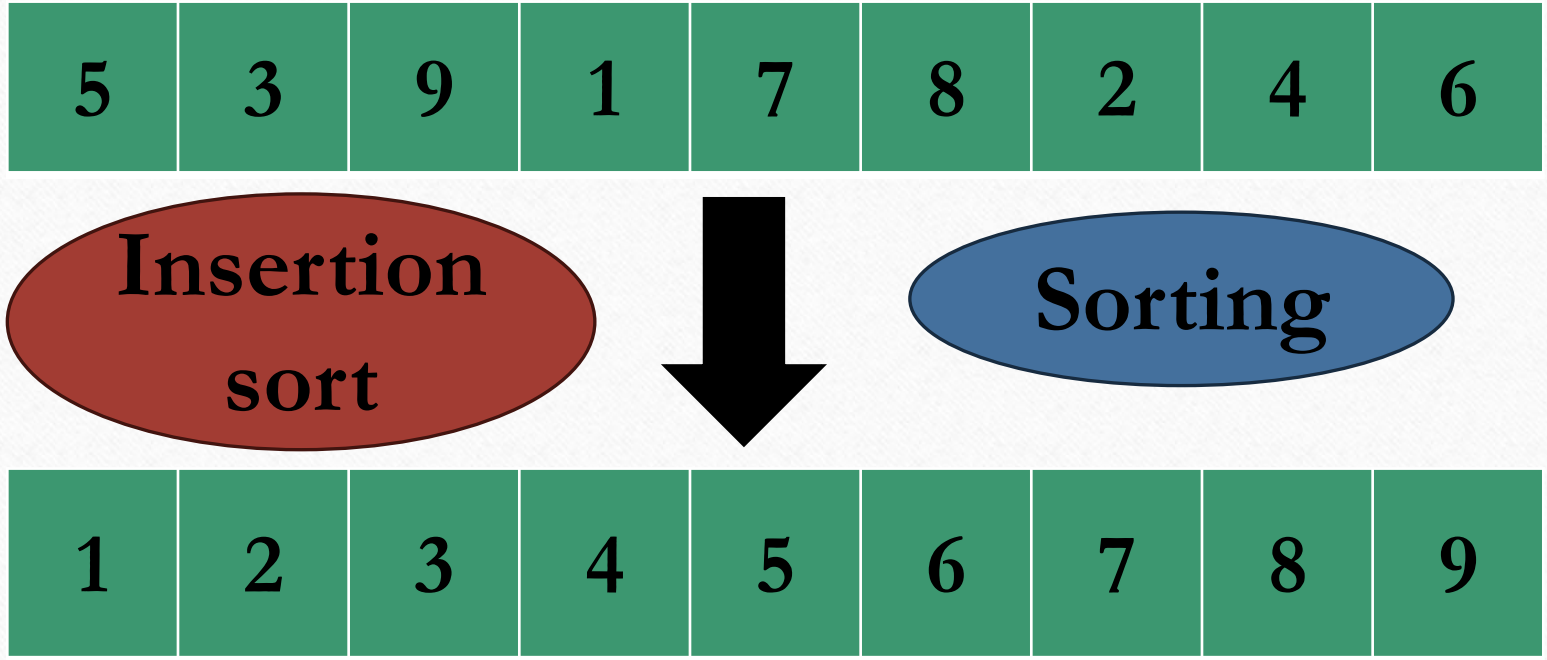
- As we iterate through the array, we **look at each element and compare it with the elements that come before it.**
- If **the element before the current element is out of position compared to the current element** (i.e., there is an inversion), we **shift** that element **one to the right.**
- This allows us to identify where the **current element should be placed relative** to the elements that come before it.

VI. Insertion Sort

Example:

- Let's analyze insertion sort using the same unsorted array we used earlier: **[5, 3, 9, 1, 7, 8, 2, 4, 6]**.
- We will start from **the left** end of the array and consider each element one at a time.

VI. Insertion Sort

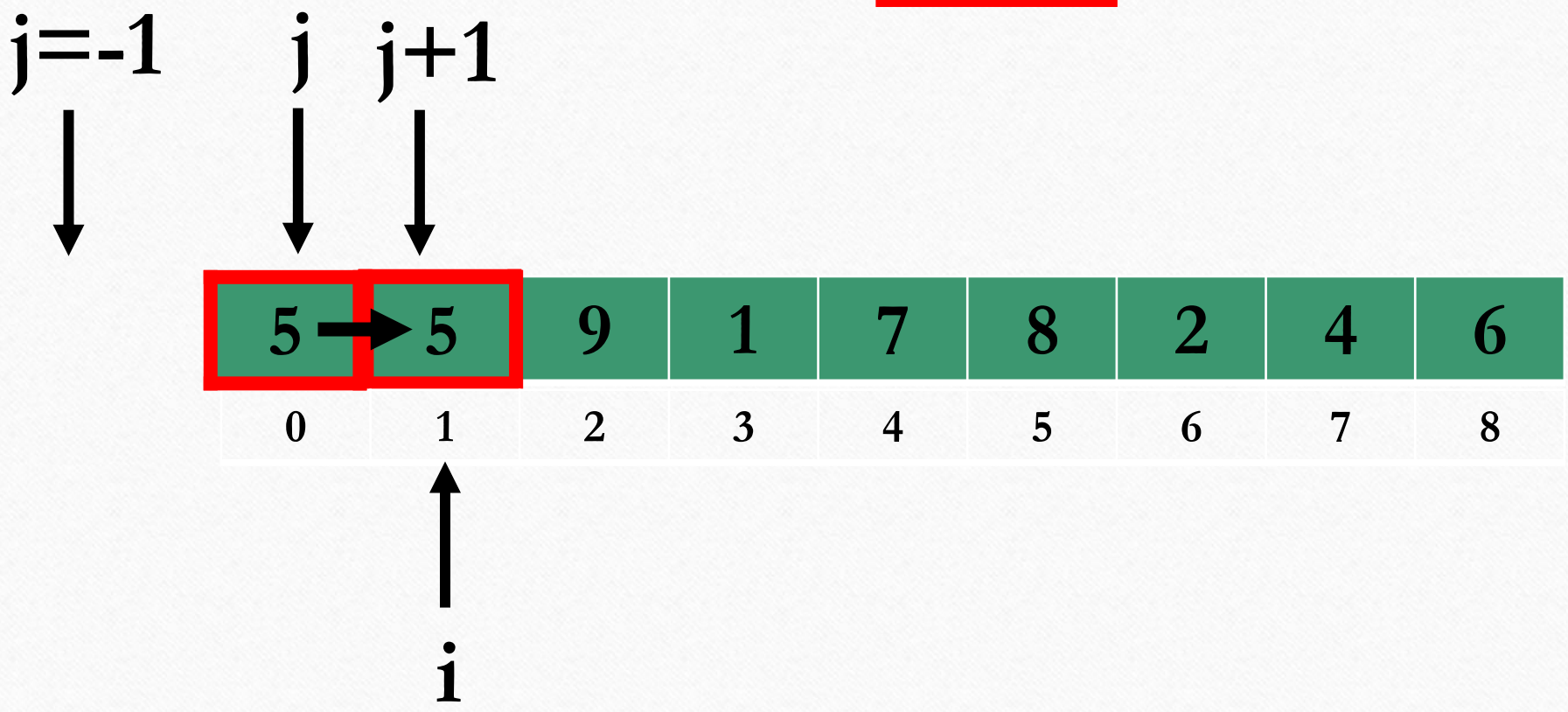


```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

First pass

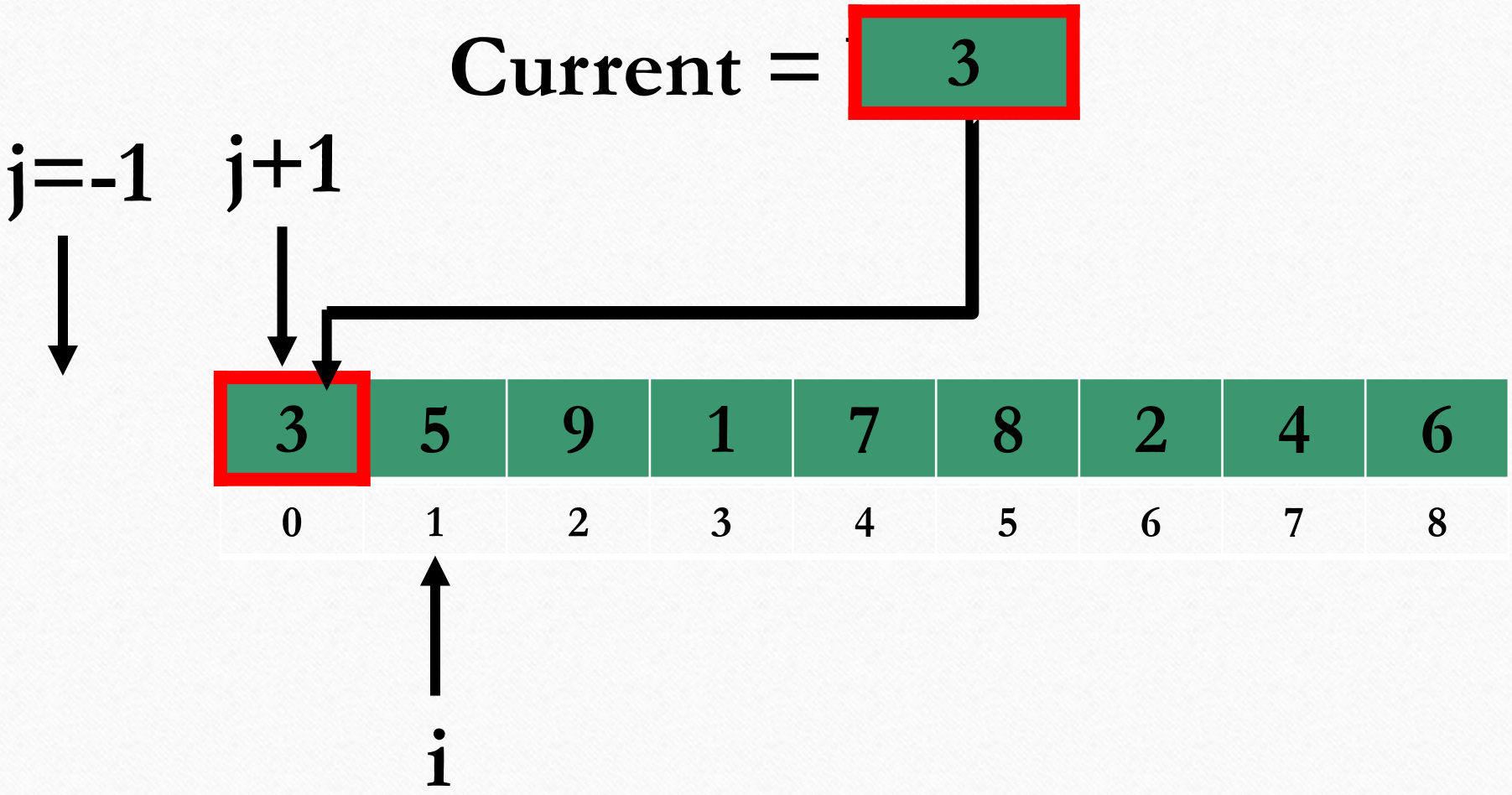
Current = 3



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

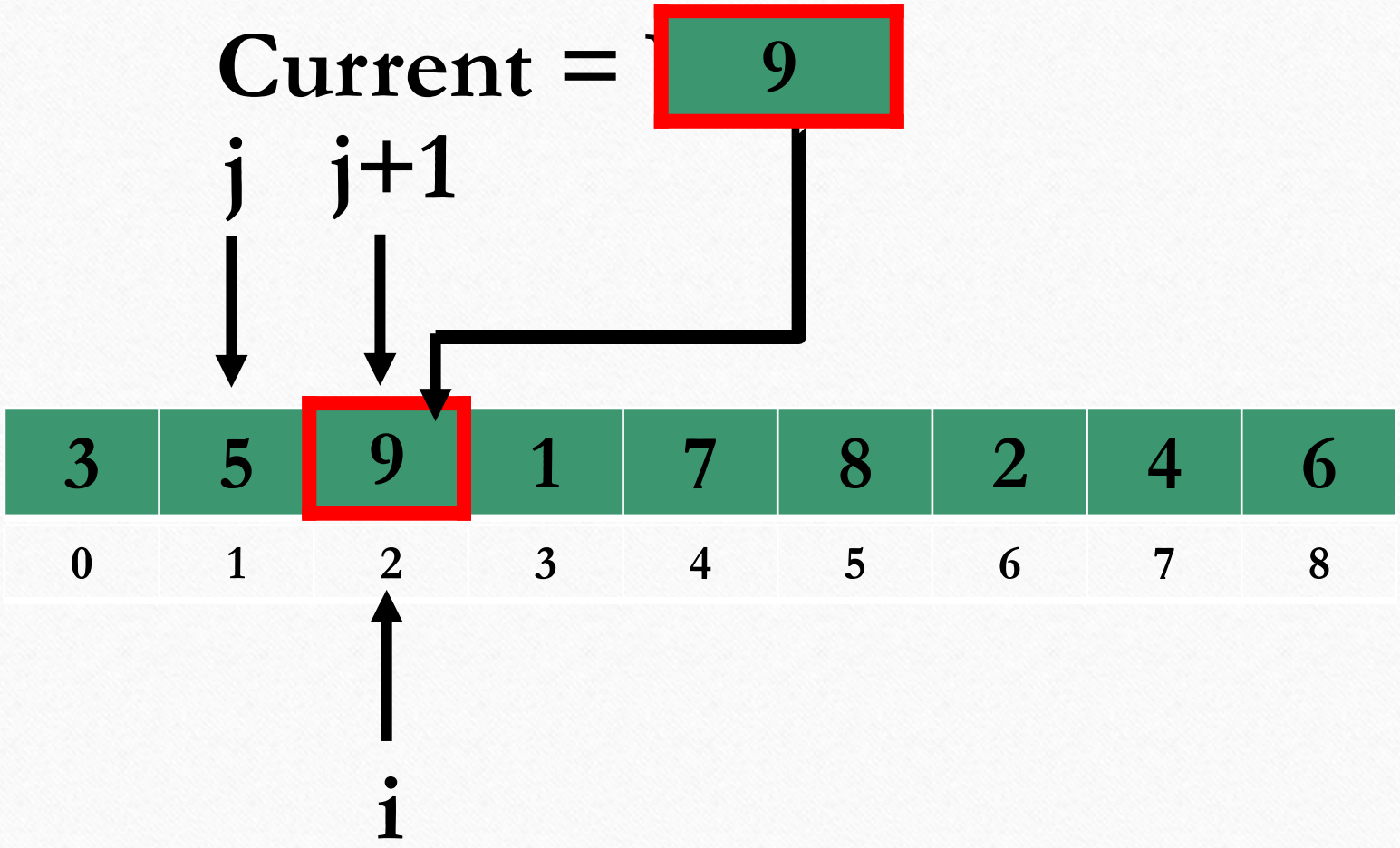
First pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

Second pass

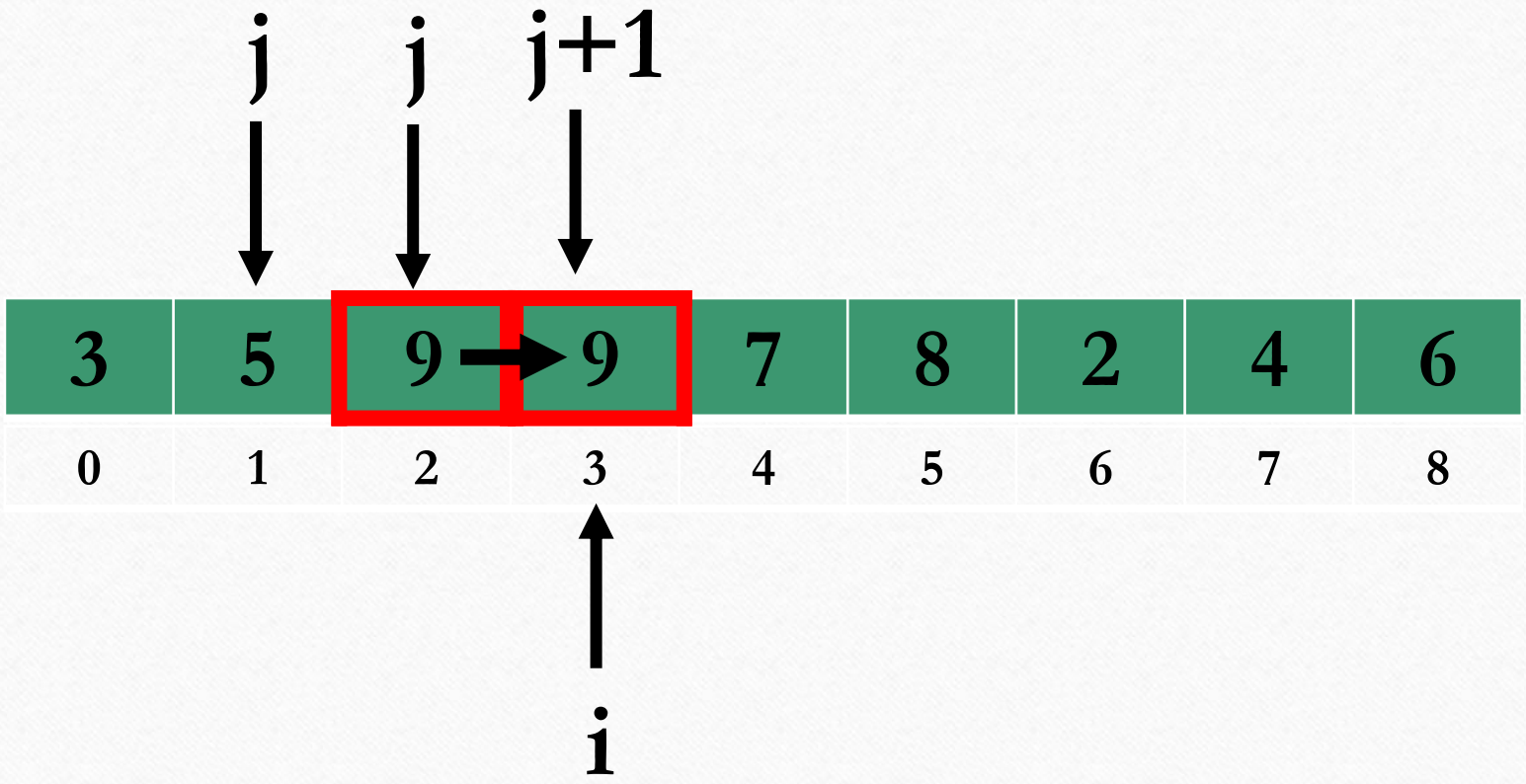


```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

3rd pass

Current = 1

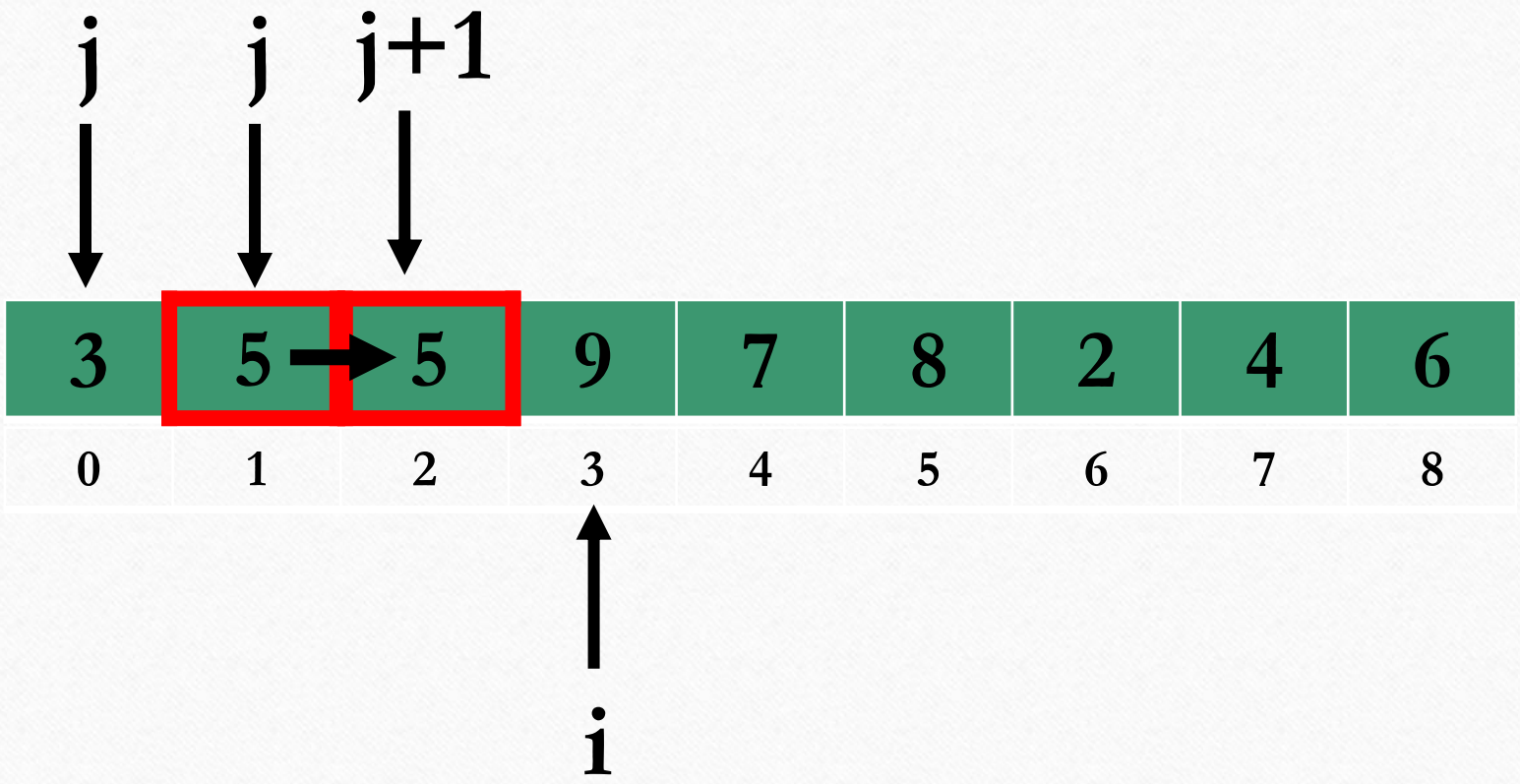


```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

3rd pass

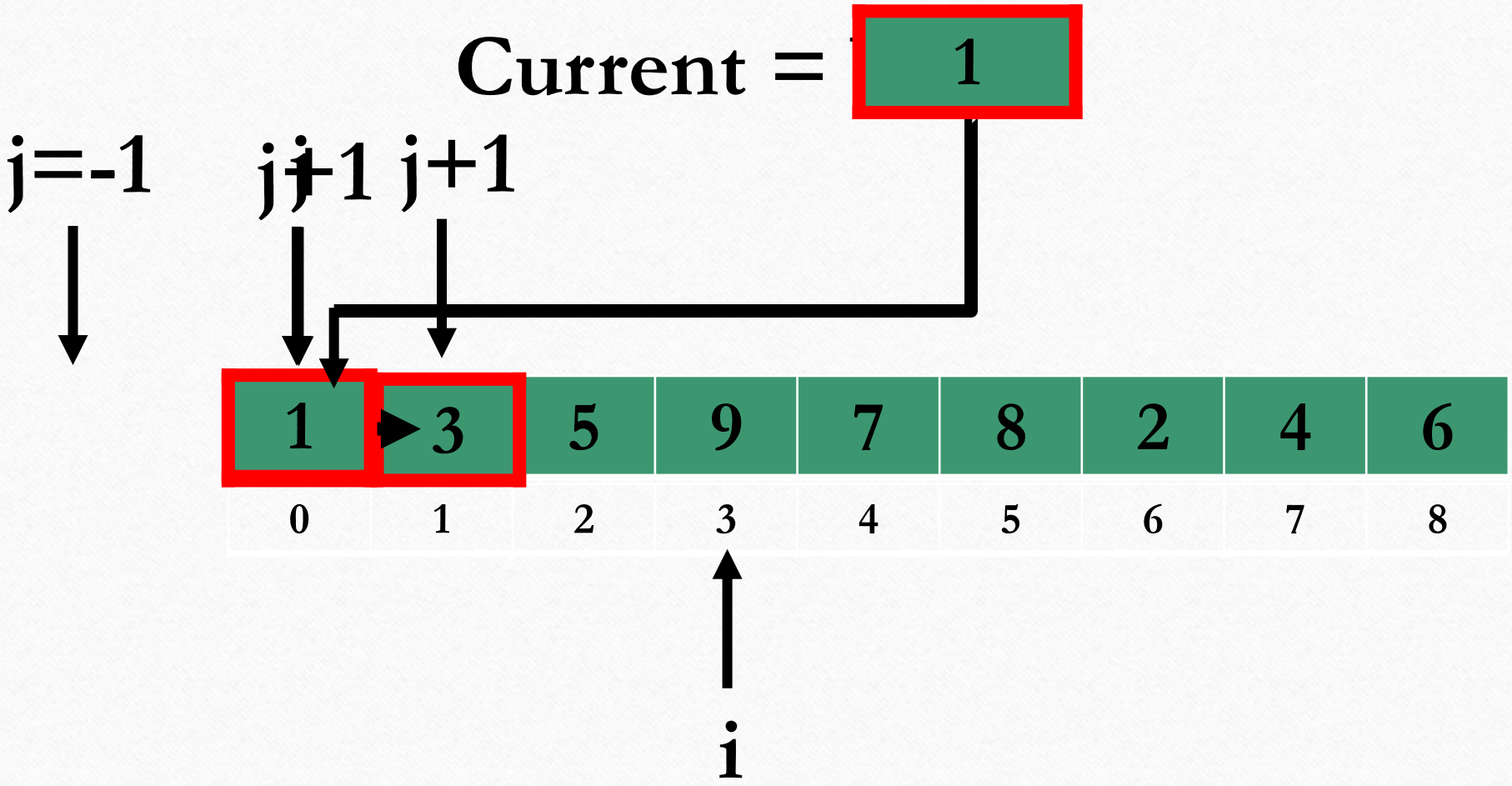
Current = 1



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

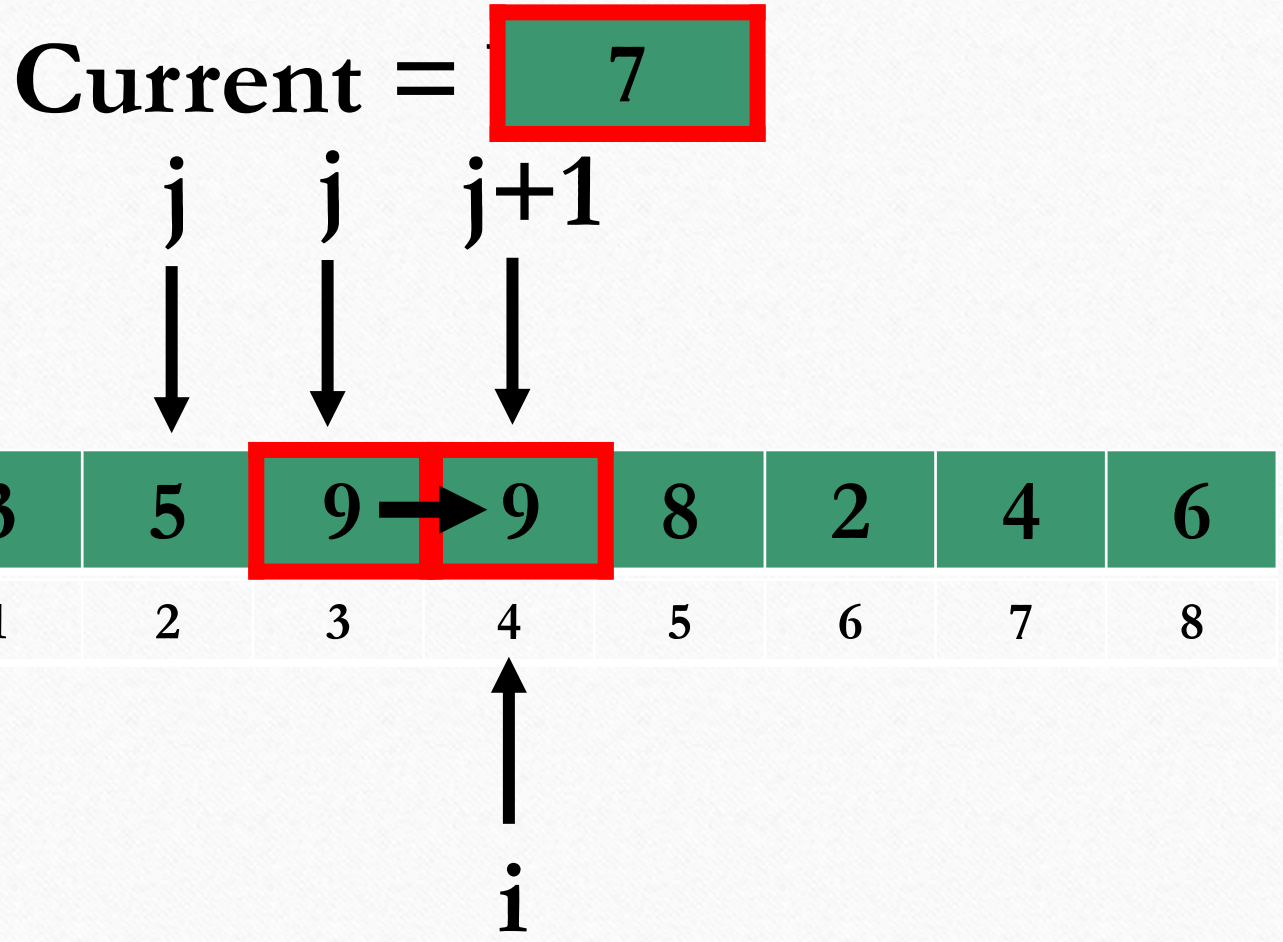
3rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

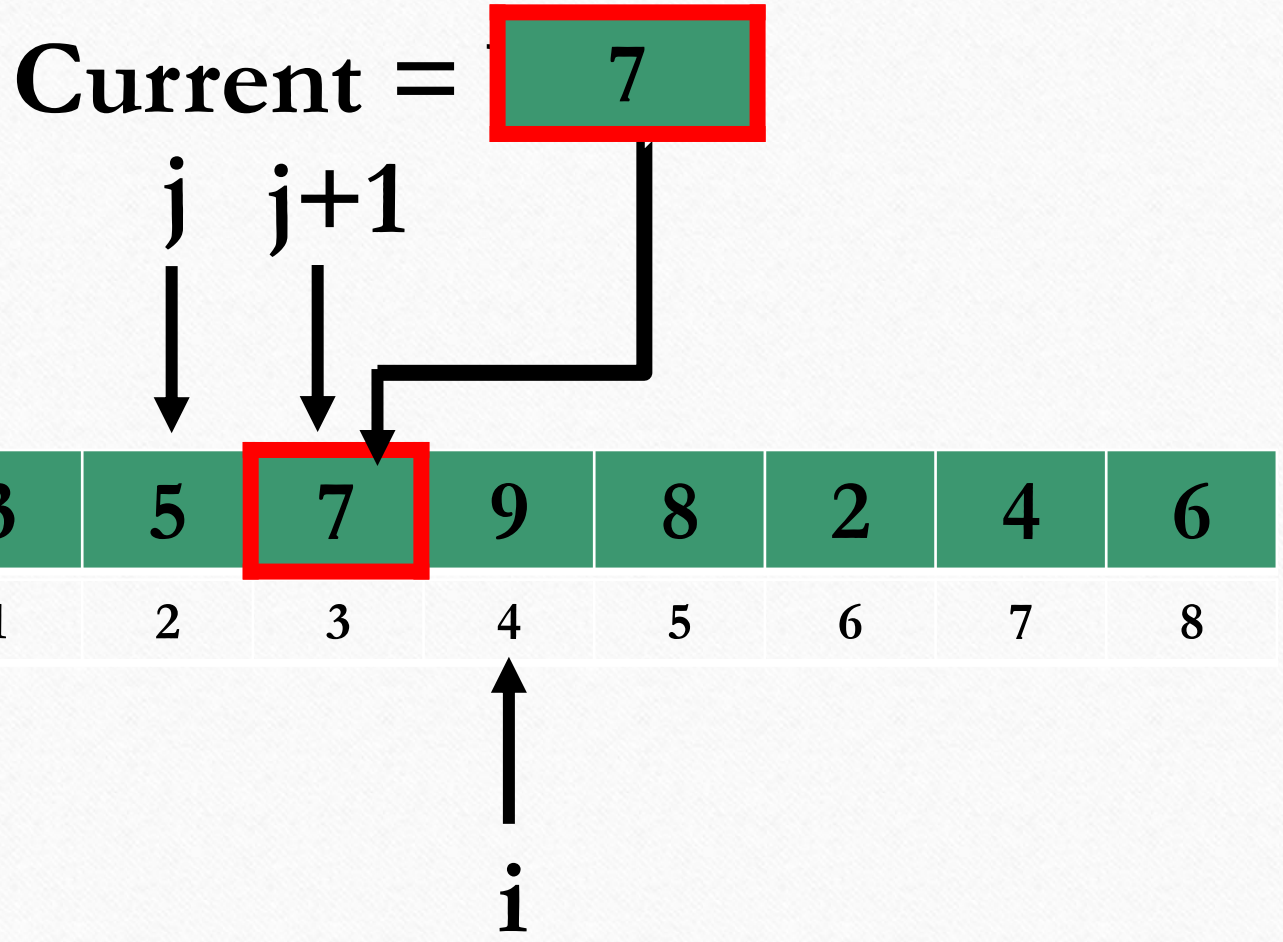
4rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

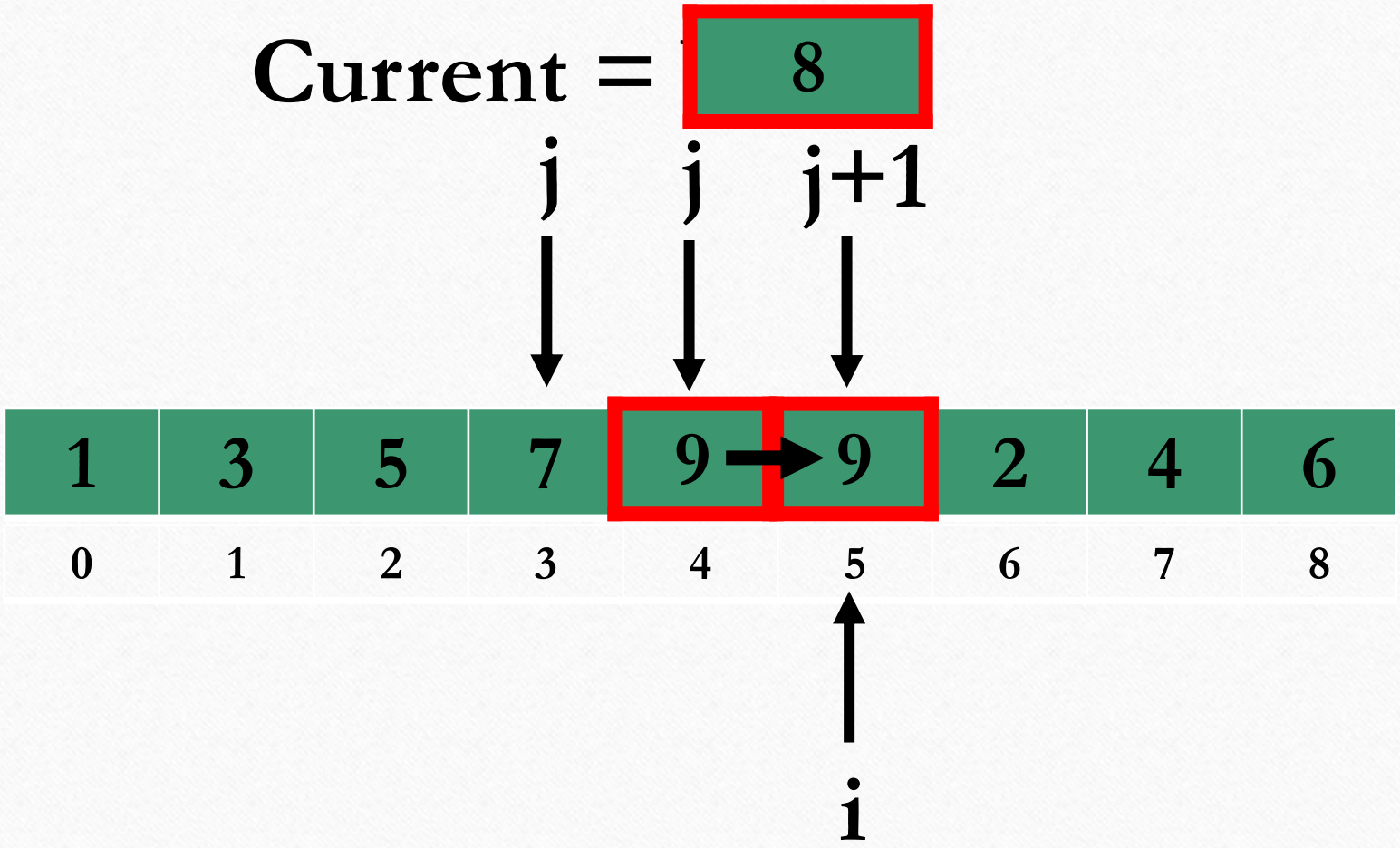
4rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

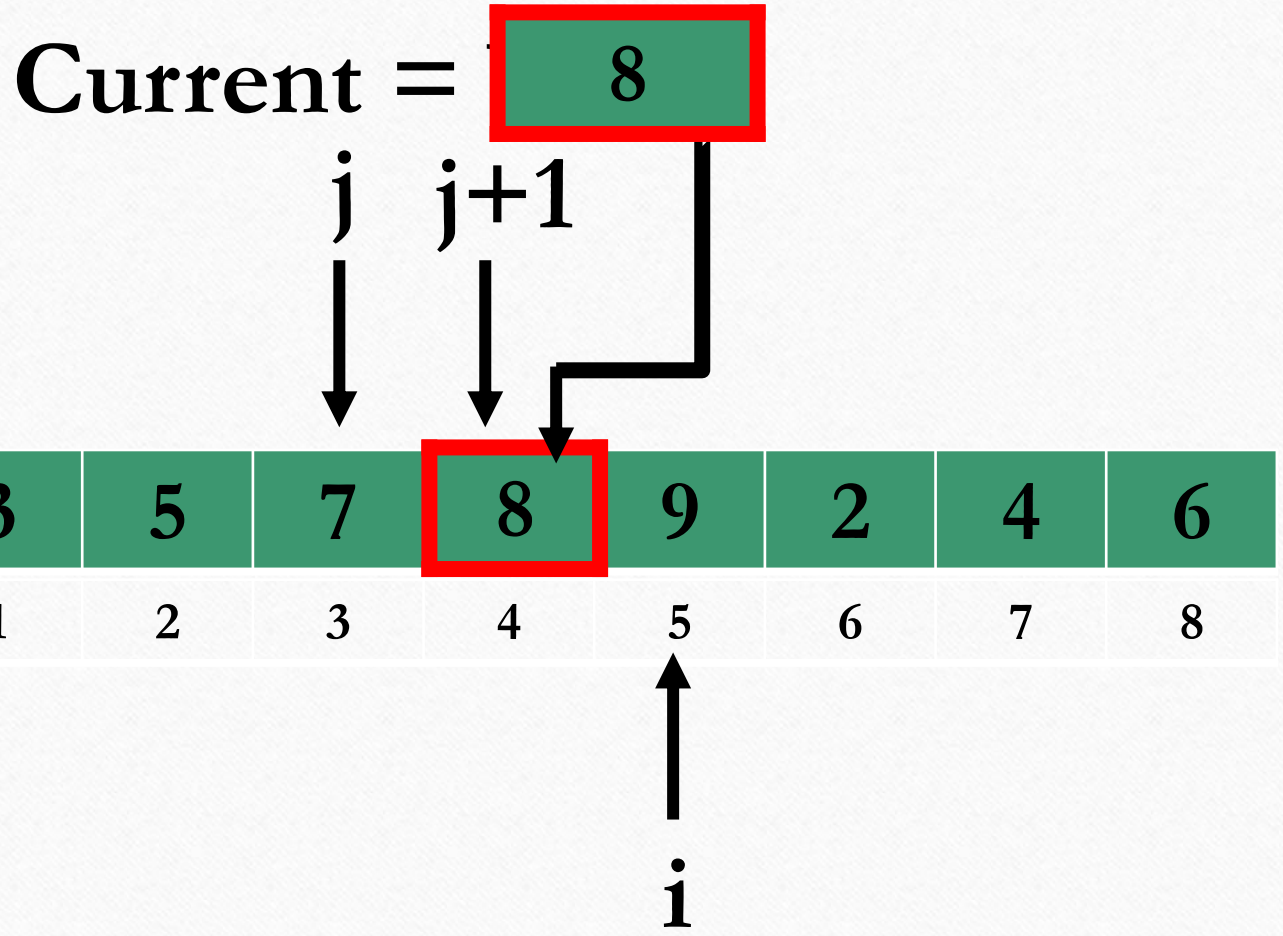
5rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

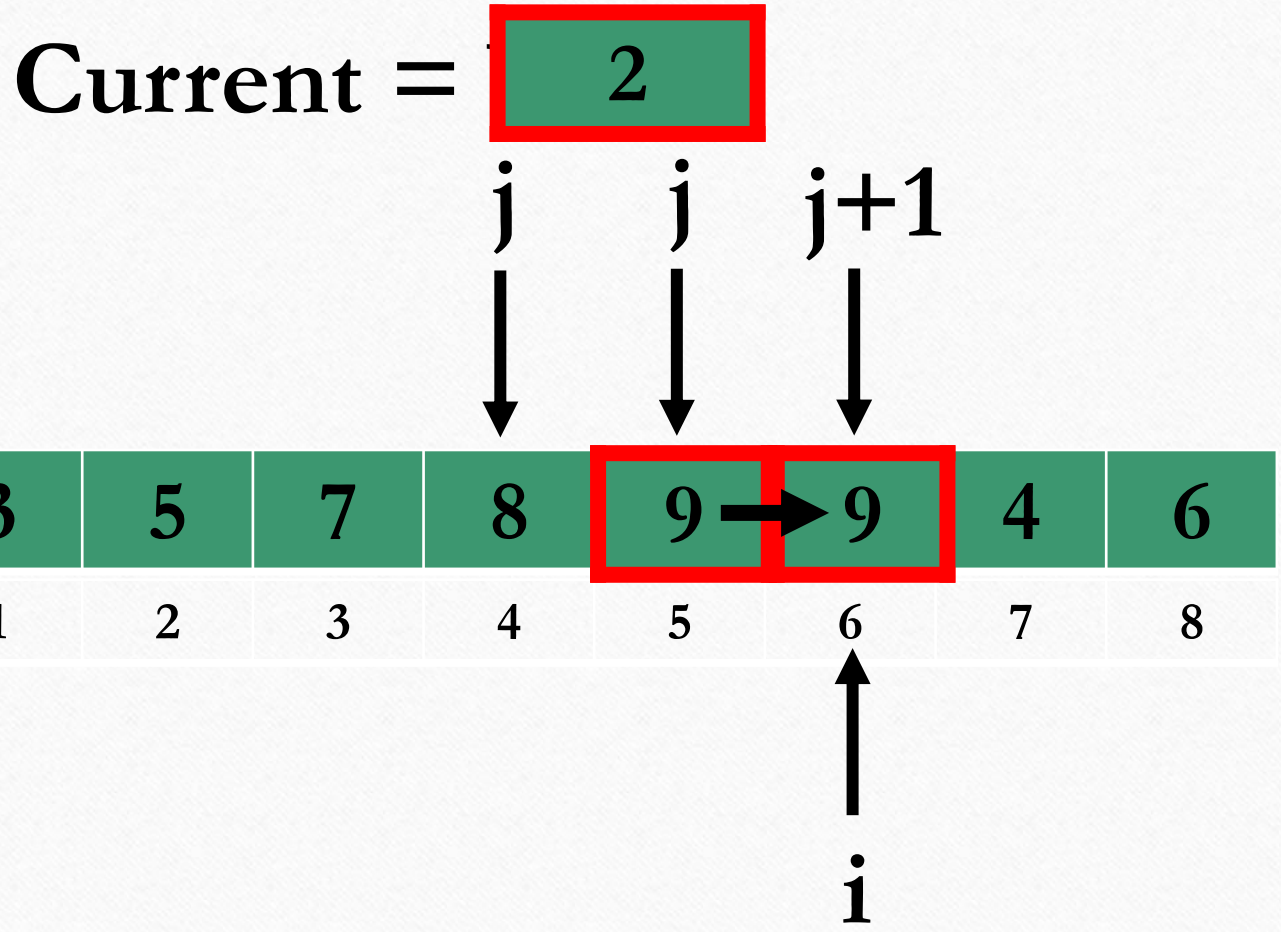
5rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

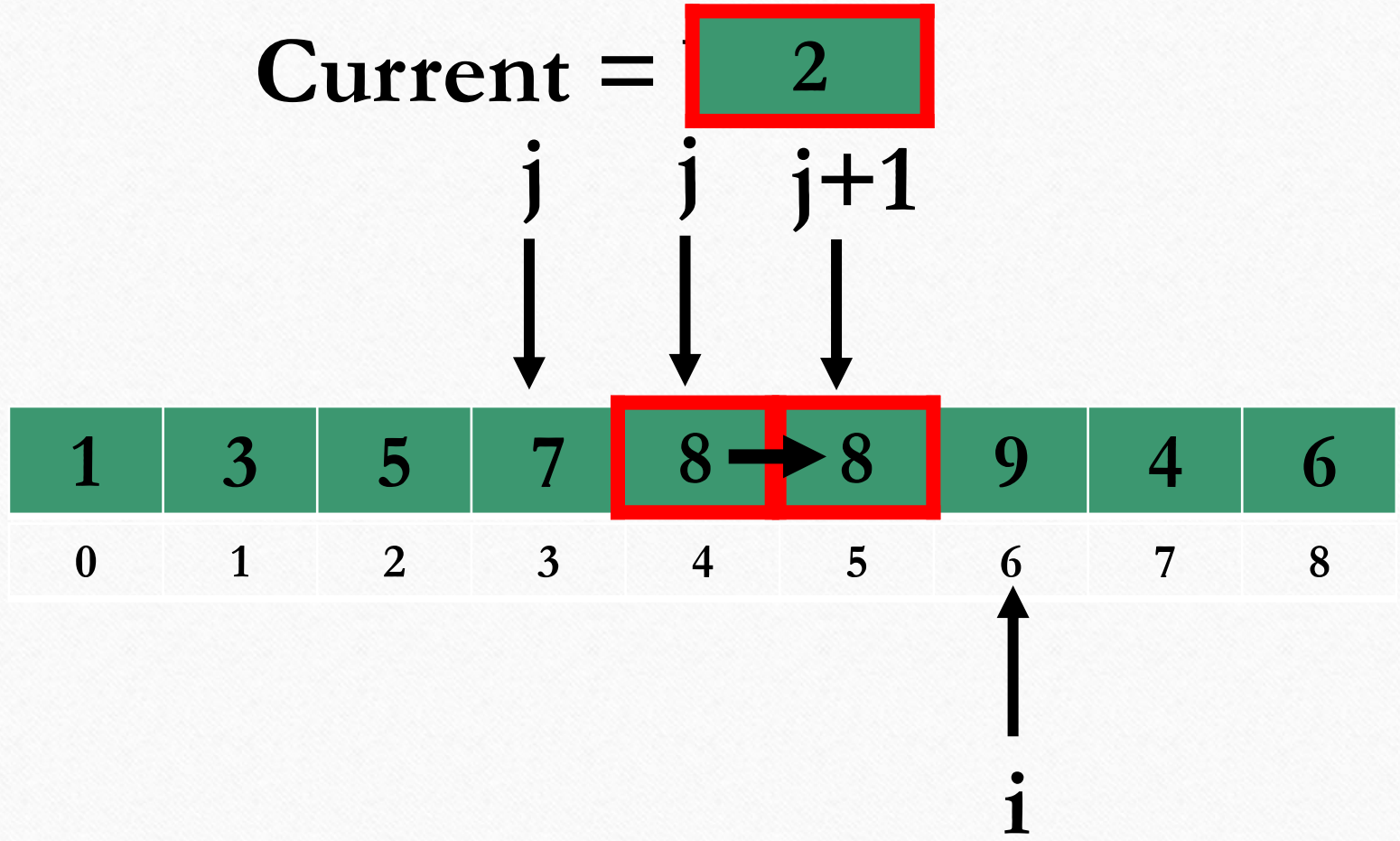
6rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

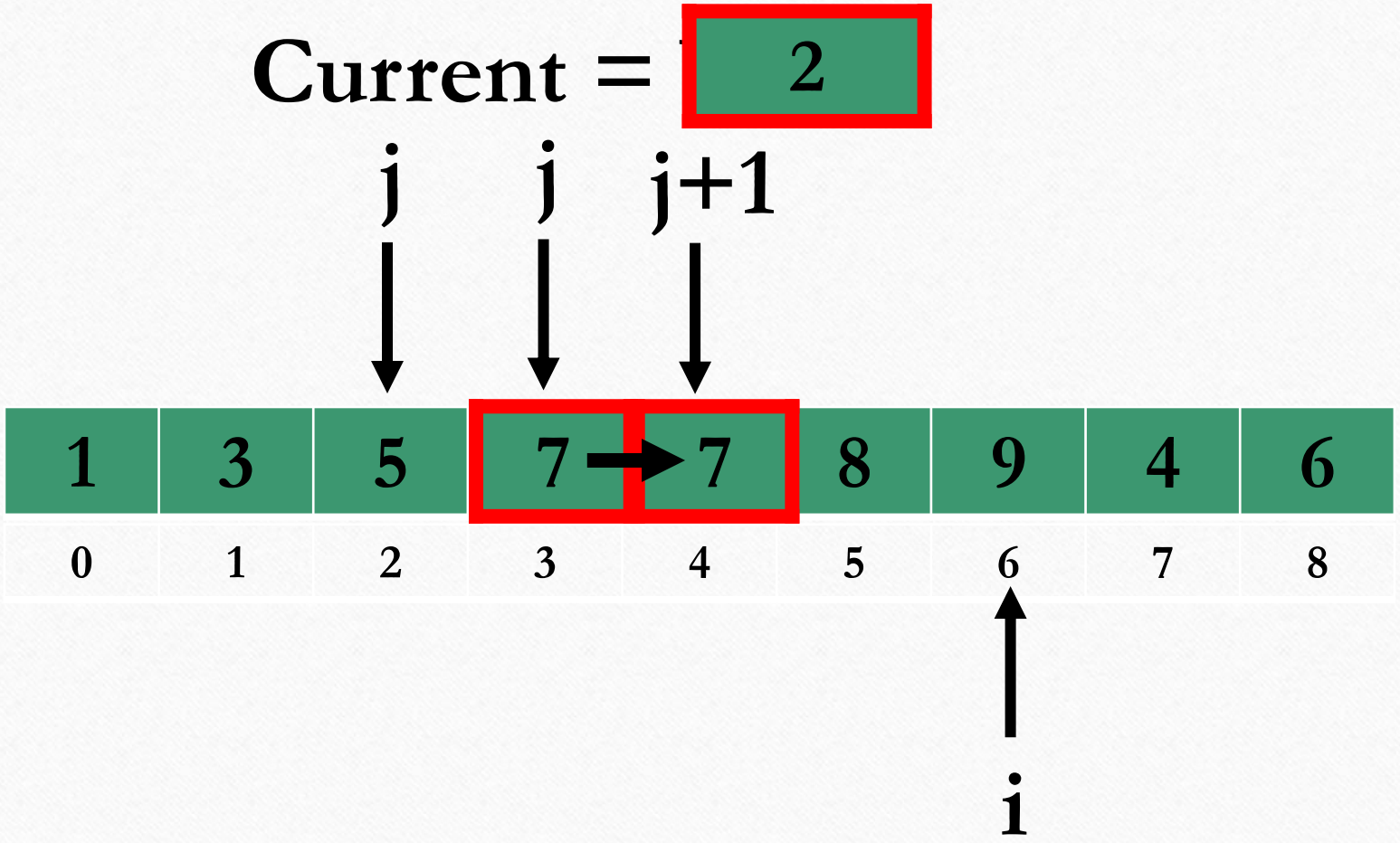
6rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

6rd pass

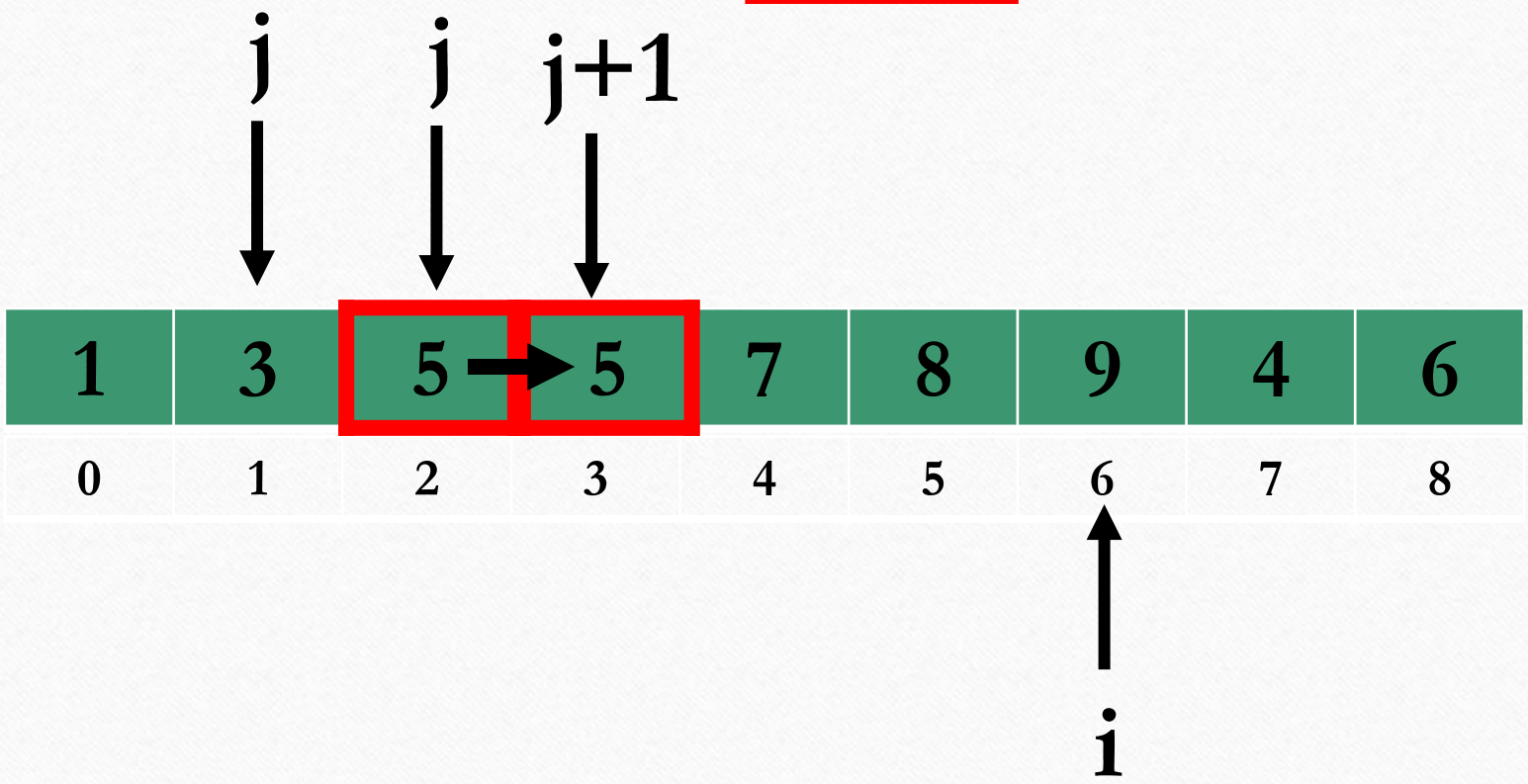


```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

6rd pass

Current = 2

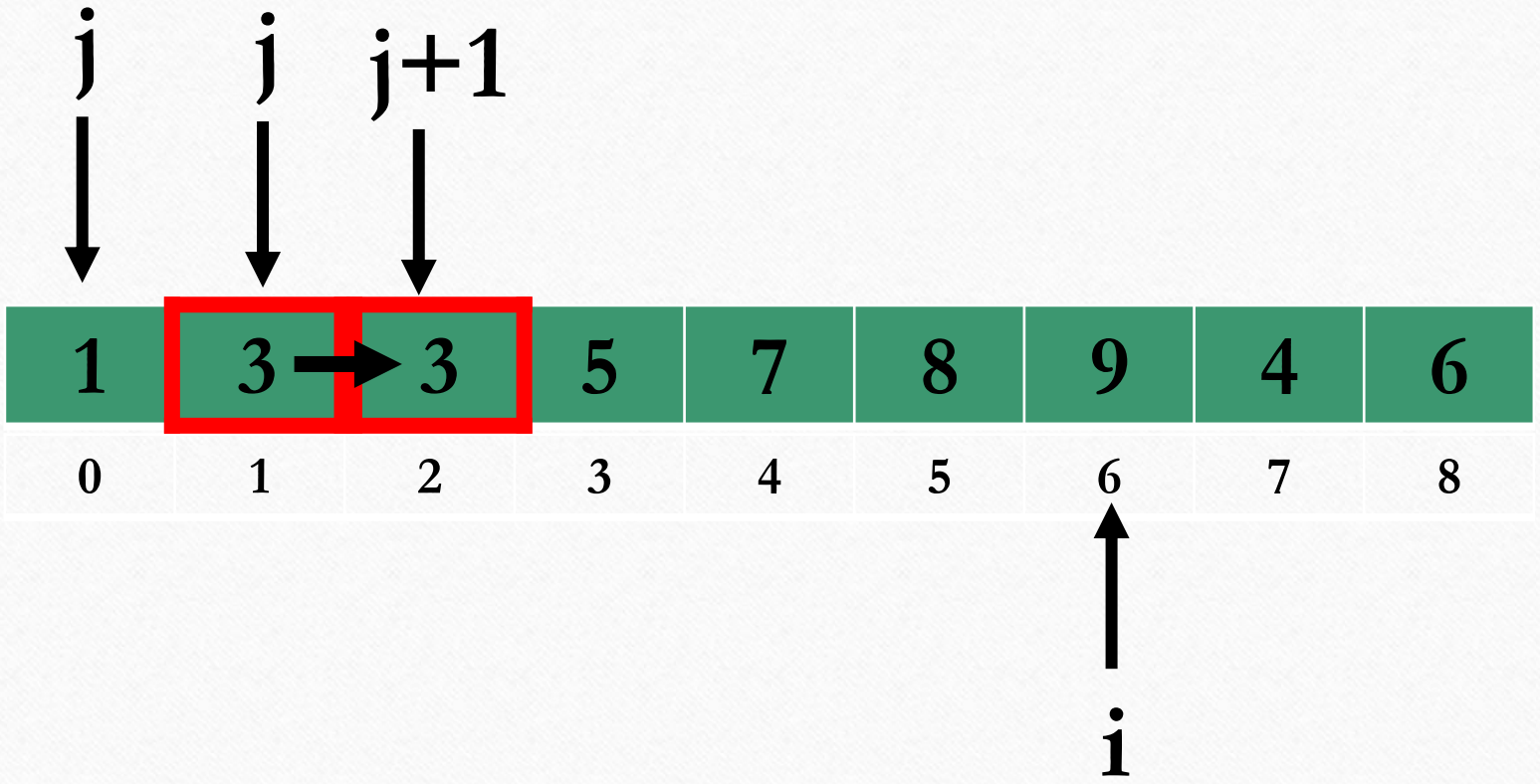


```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

6rd pass

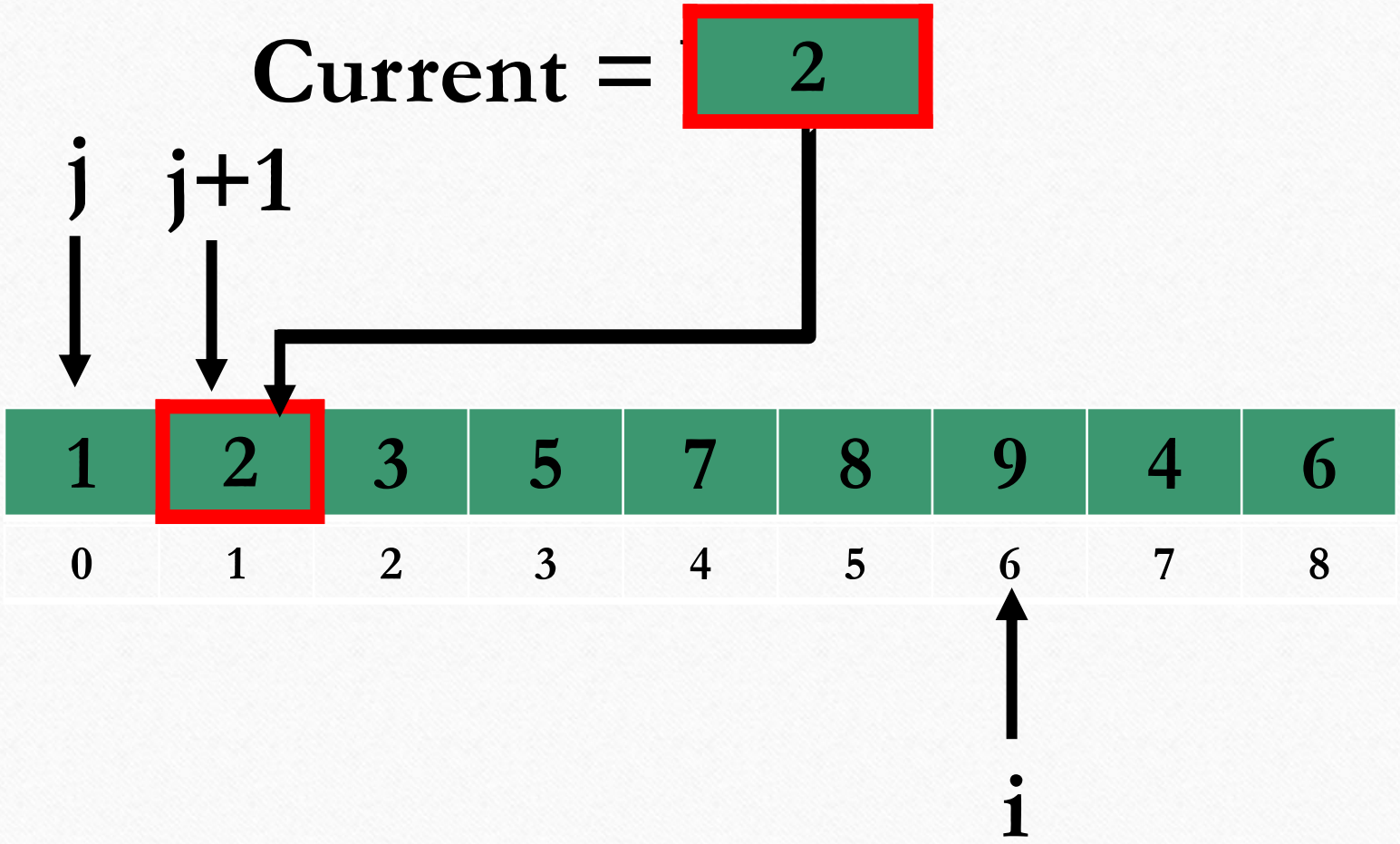
Current = 2



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

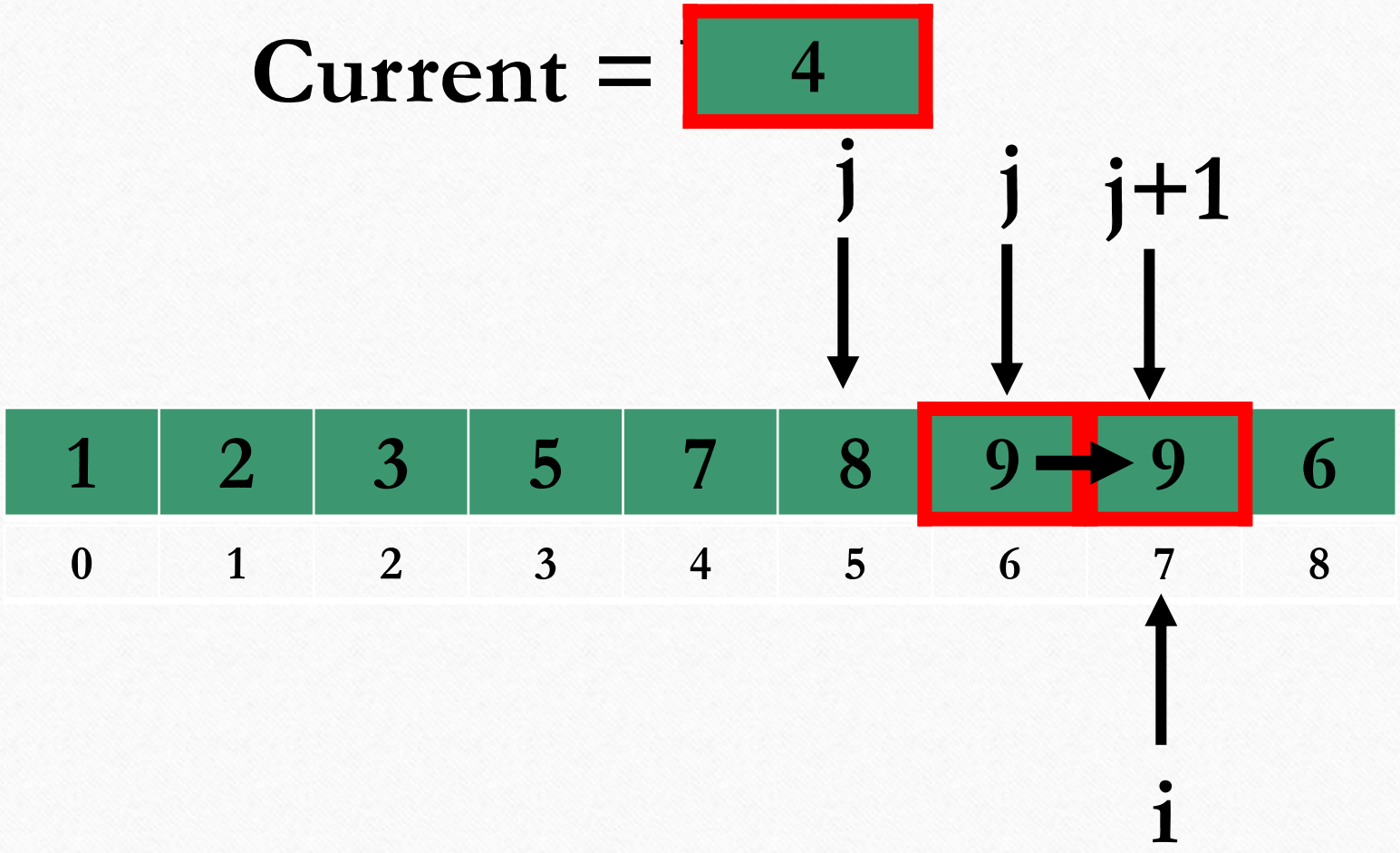
6rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

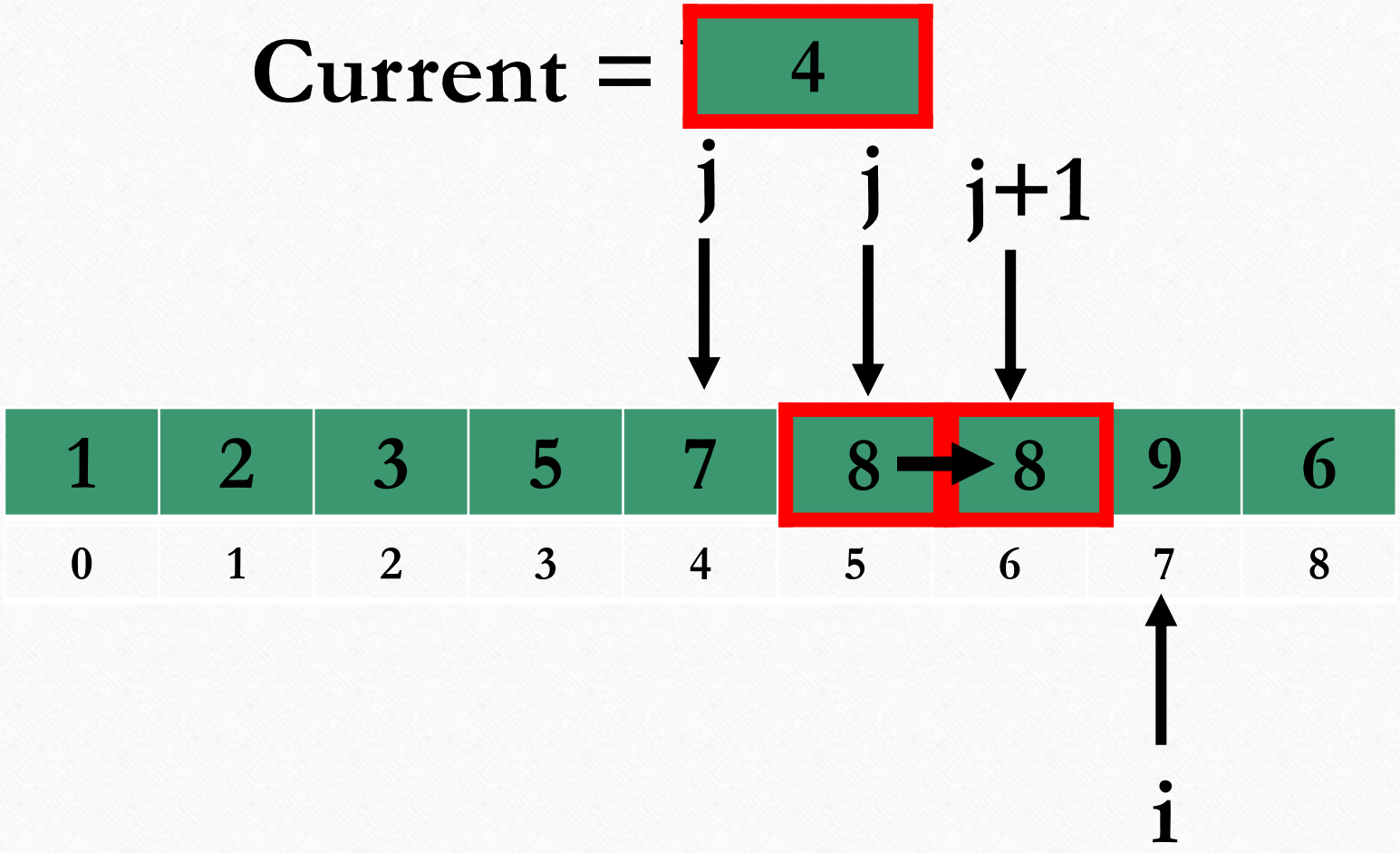
7rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

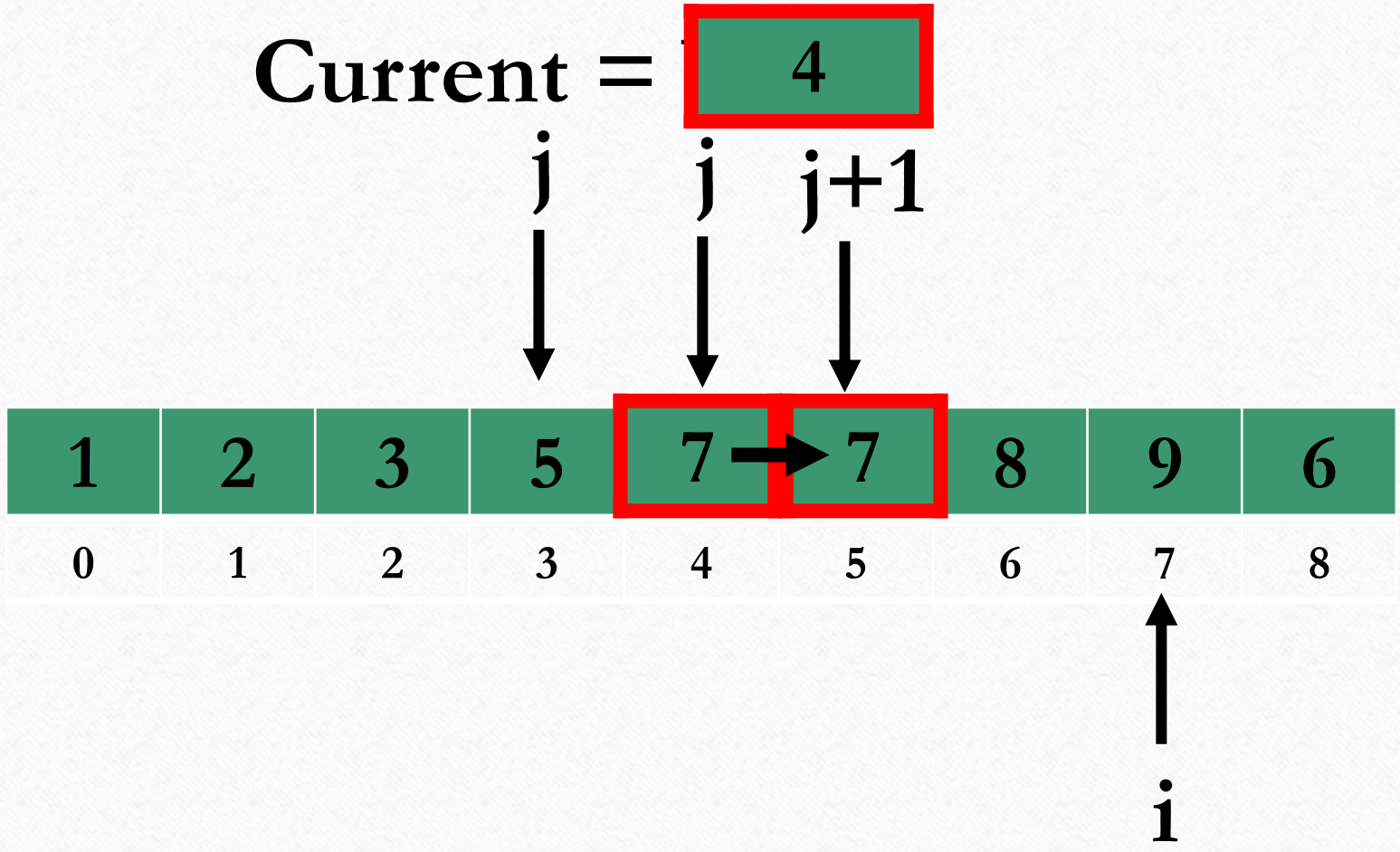
7rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

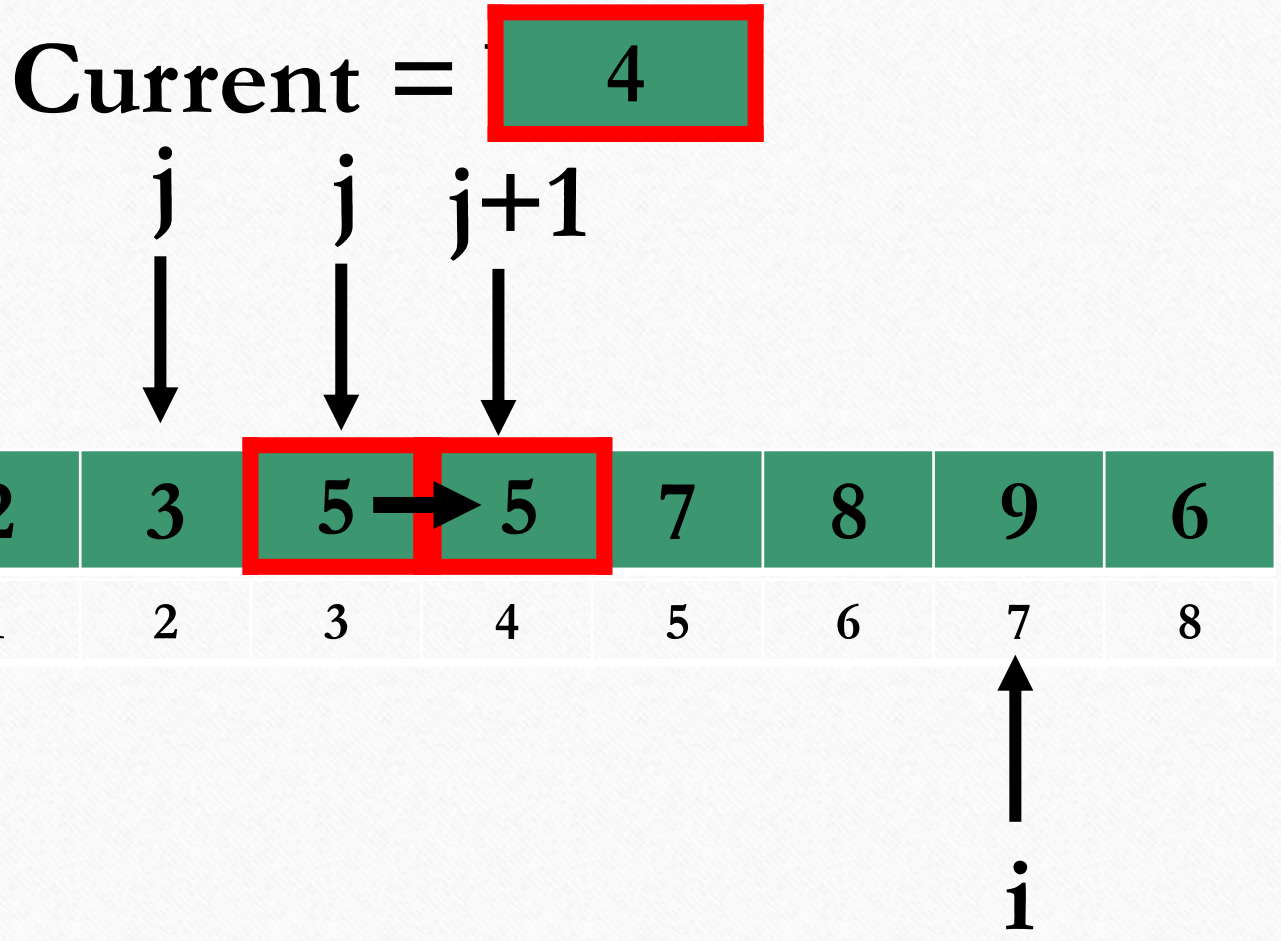
7rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

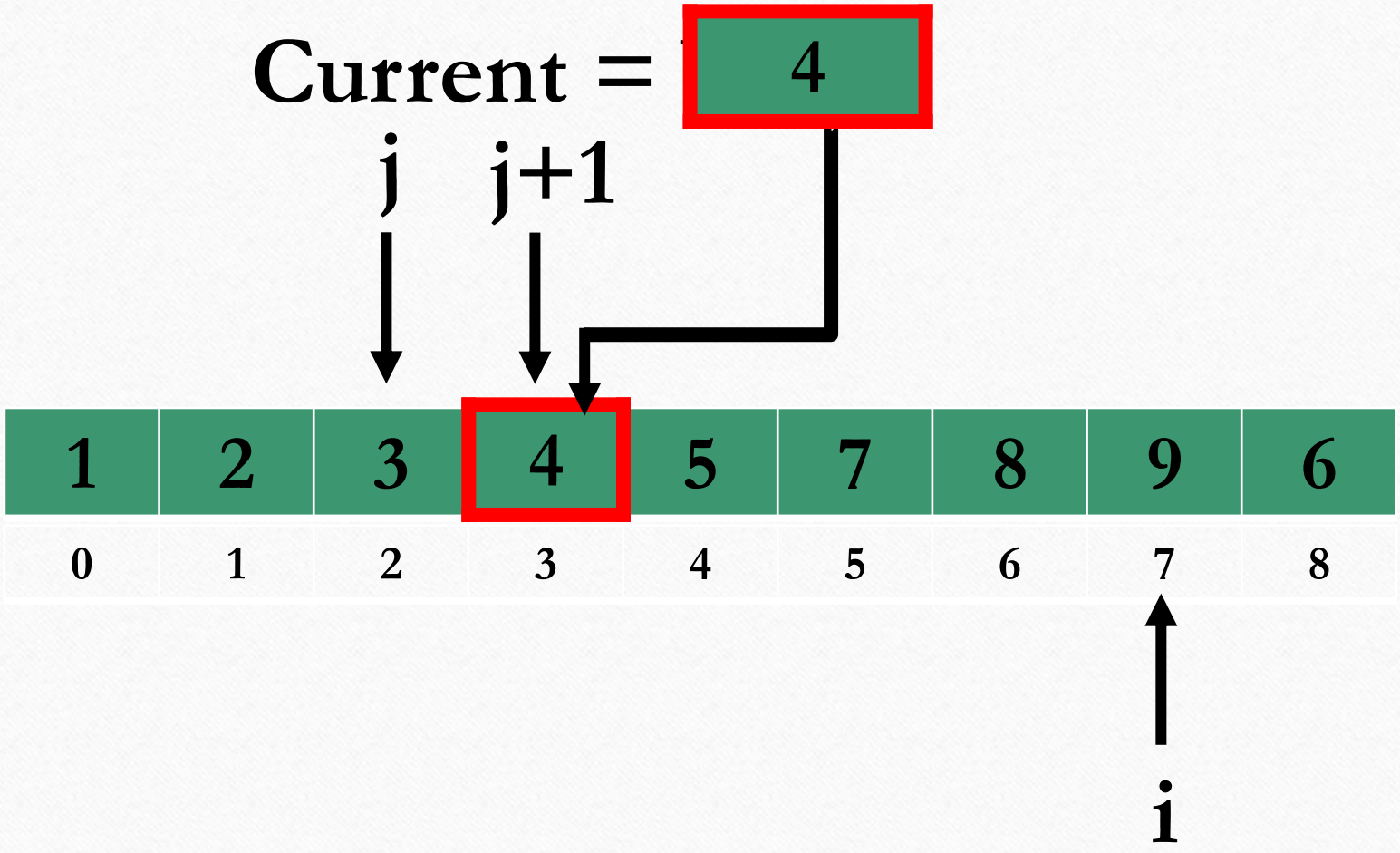
7rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

7rd pass

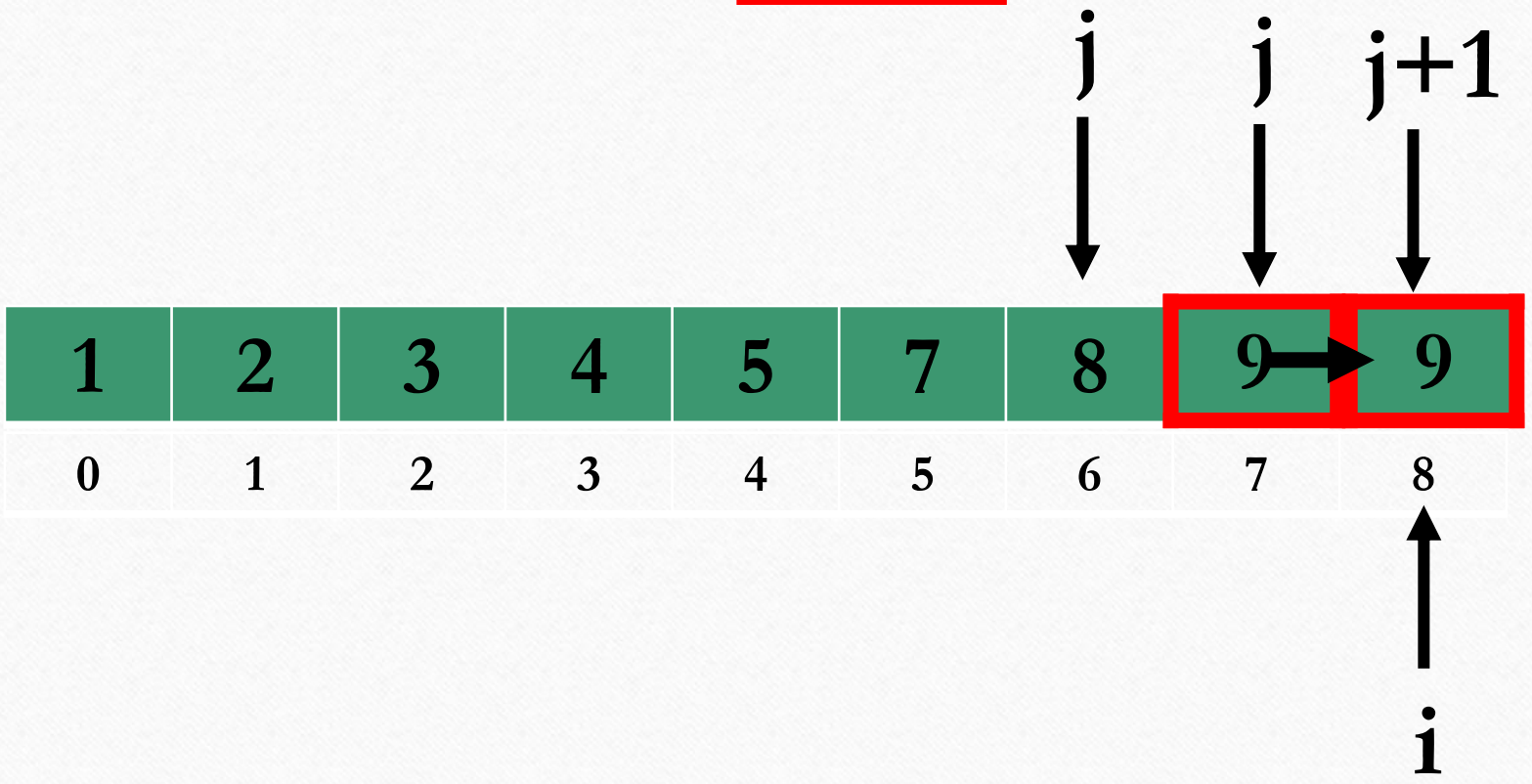


```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

8rd pass

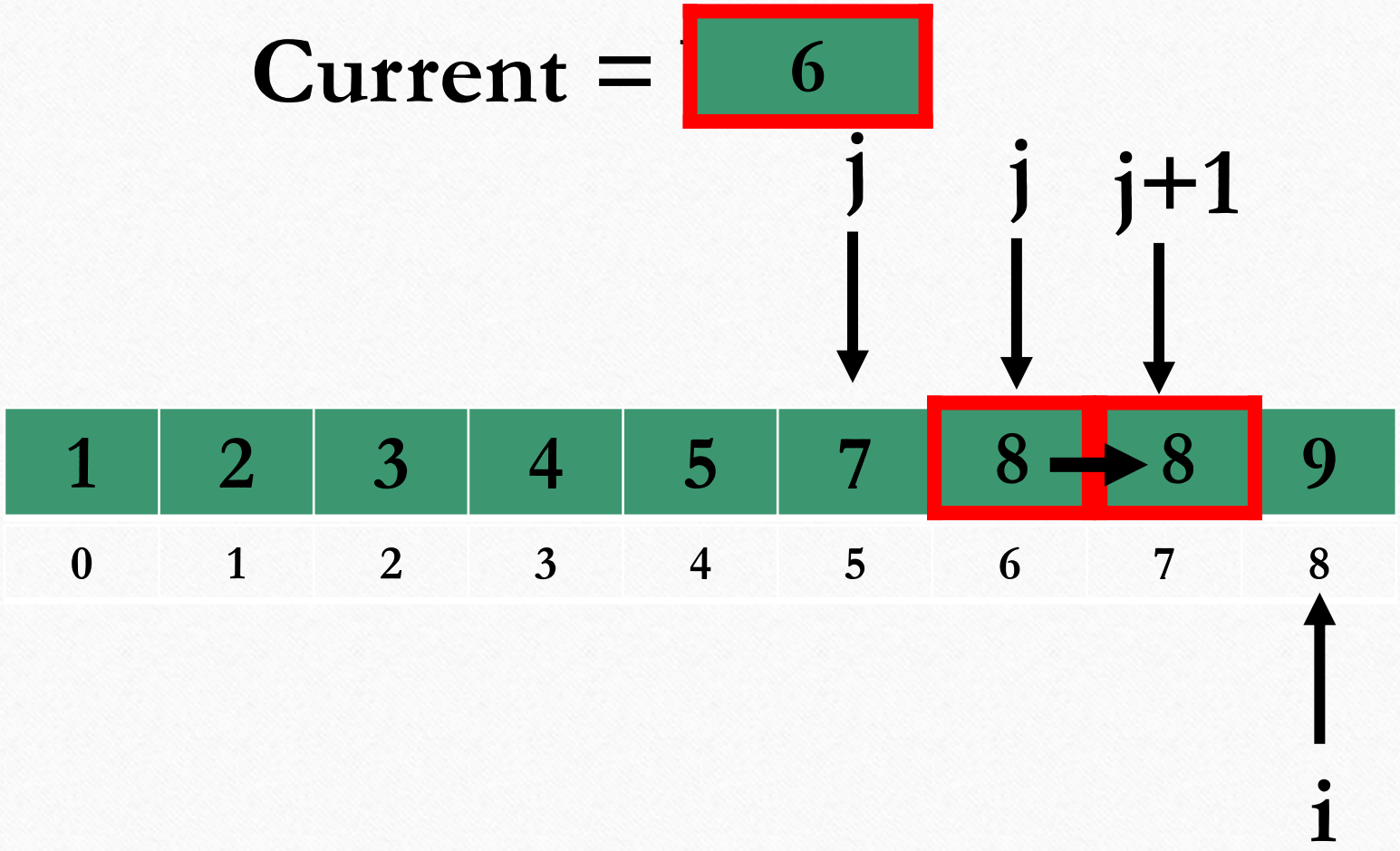
Current = 6



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

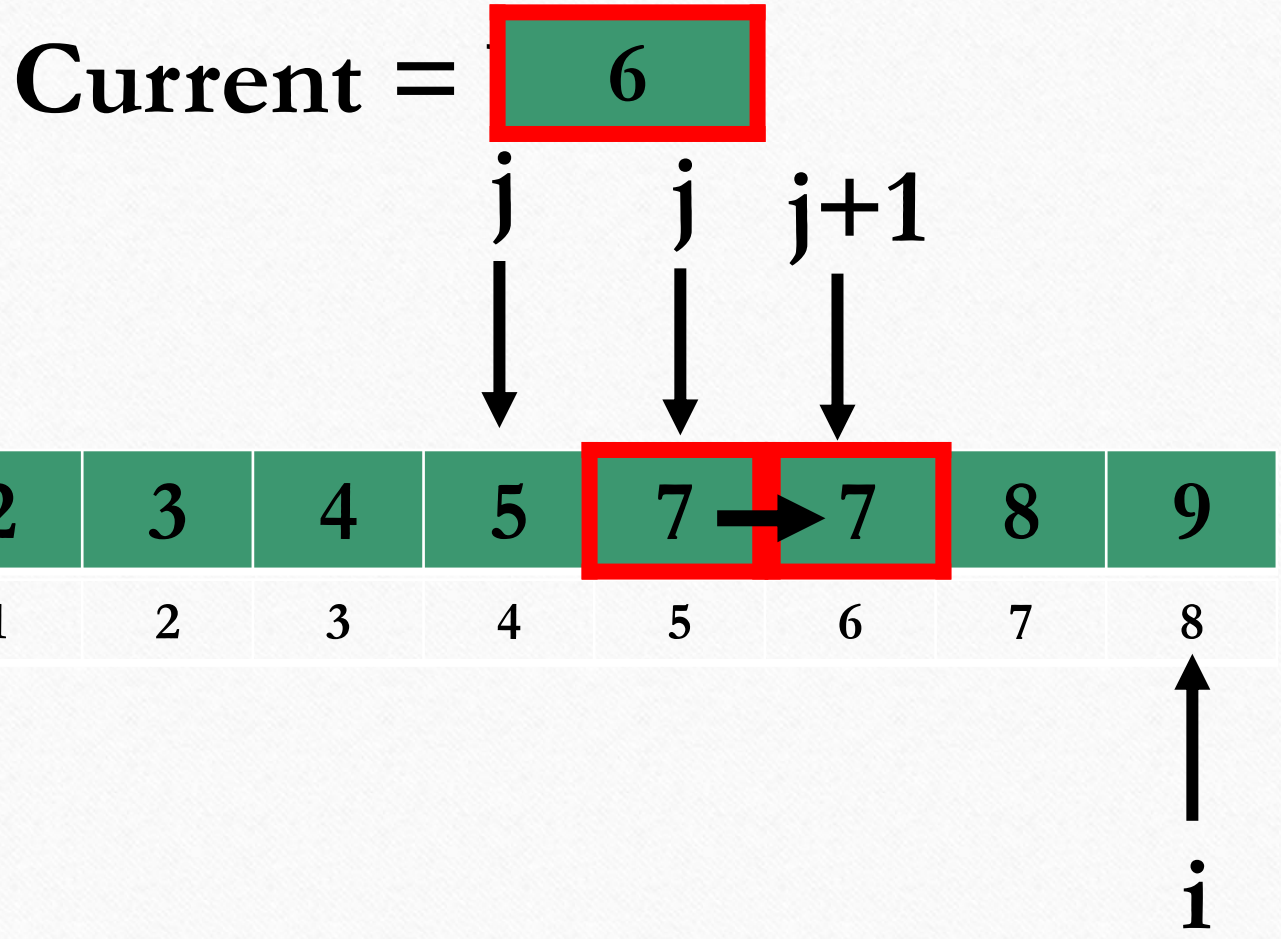
8rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

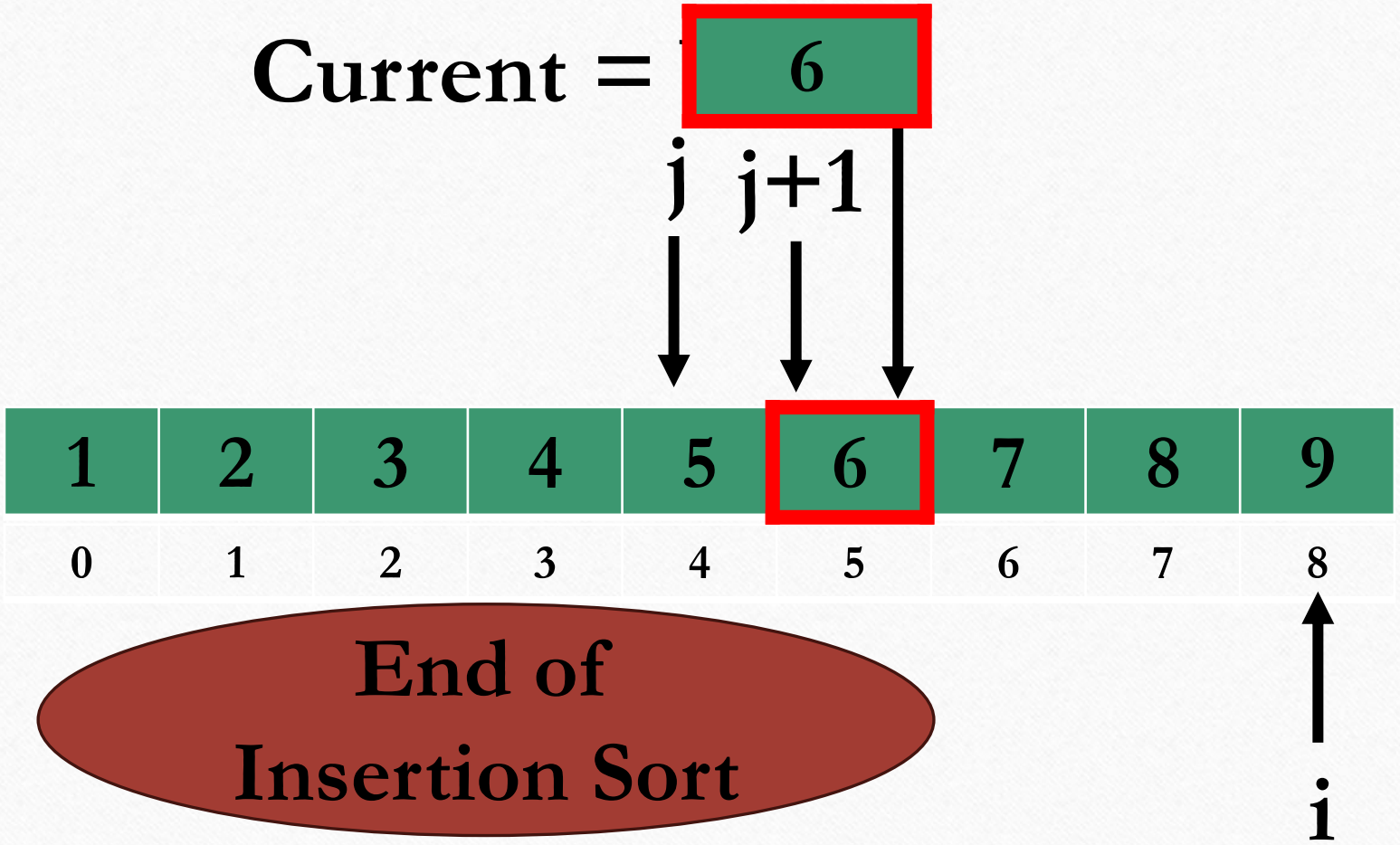
8rd pass



```
for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
            Vec[j + 1] = Vec[j];
            j--;
      }
      Vec[j + 1] = current;
}
```

VI. Insertion Sort

8rd pass



VI. Insertion Sort

Time Complexity of Insertion Sort

```
void insertion_sort(int Vec[], int n) {
   for (int i = 1; i < n; i++) {
      int current = Vec[i];
      int j = i - 1;
      while (j >= 0 && Vec[j] > current) {
         Vec[j + 1] = Vec[j];
         j--;
      }
      Vec[j + 1] = current;
   }
}
```

i	j
1	1
2	2
.....	....
.....	....
N-2	N-2
N-1	N-1

VI. Insertion Sort

Time Complexity of Insertion Sort

- If we define the size of the array as $\boldsymbol{n}$ then in the first pass, it will make $\mathbf{1}$ comparisons; in the second pass, it will do $\mathbf{2}$; in the third pass, it will do $\mathbf{3}$ and so on.
- Thus, the total number of comparisons can be found by:

$$\boldsymbol{T(n) = 1 + 2 + \cdots + (n-2) + (n-1) = \frac{n(n-1)}{2} = \mathcal{O}(n^2)}$$

- Thus, our current insertion sort implementation runs in $\boldsymbol{\mathcal{O}(n^2)}$ time.

VII. Quicksort

- Quicksort is a **partition-based sorting** technique developed by British computer scientist Tony Hoare in 1959 and published in 1961.
- It is also known as **partition exchange sorting**.
- The basic concept of the quicksort process is **to pick an element** from an array and **rearrange the other elements around it.**

VII. Quicksort

- The quicksort algorithm consists of the following **two steps**, which are repeated until the array is fully sorted:

1) **A pivot element is selected** (the method of selecting the pivot depends on the implementation, but a common strategy is to select the **last element as the pivot** — this is the method that will be used in these notes).

VII. Quicksort

- The quicksort algorithm consists of the following **two steps**, which are repeated until the array is fully sorted:

  2) The **remaining elements in the array are partitioned** so that all elements **to the left of the pivot are lesser** than the pivot, and **all elements to the right of the pivot are greater** than the pivot.

- This process is **repeated recursively until sub-lists consist of only one element.**

VII. Quicksort

- An outline of the quicksort algorithm is shown below:

```
void quick_sort(int Vec[], int left, int right) {
   if (left + 1 >= right) {
      return;
   } // if
   int pivot = partition(Vec, left, right);
   quick_sort(Vec, left, pivot);
   quick_sort(Vec, pivot + 1, right);
} // quick_sort()
```

VII. Quicksort

- The ***partition()*** function takes in an index range **[left, right)** and selects the **last element as the pivot.**
- The remaining elements are partitioned so that elements to **the left of the pivot are lesser, and elements to the right are greater.**

VII. Quicksort

```
int partition(int Vec[], int left, int right) {
 int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot])
        right--;}
    if (left >= right) { break;}
    swap(Vec[left], Vec[right - 1]);
 }
 swap(Vec[left], Vec[pivot]);
 return left;
} // partition()
```

VII. Quicksort

Example:

- Let's look at what this code does using an example. Consider the following unsorted array:

Vec :

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

- Call Function: `quick_sort(Vec, 0, 9); //9 size`

VII. Quicksort

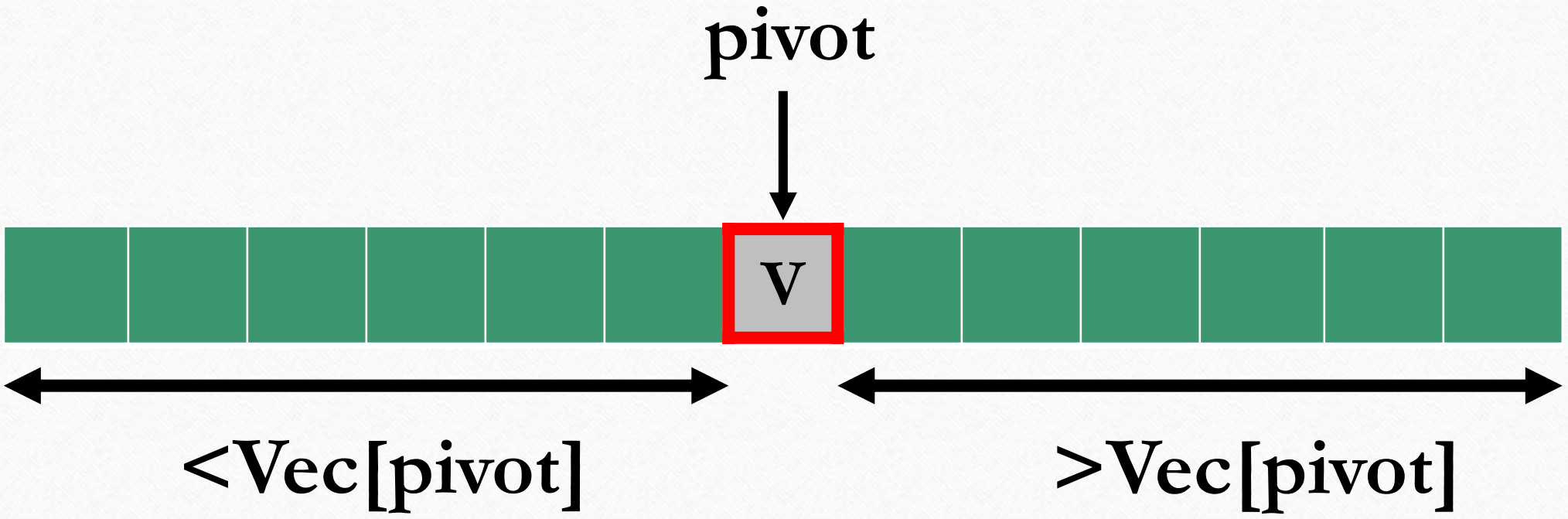
Example:

- Call : **quick_sort(Vec, 0, 9) //9 size**

```
if ( 0 + 1 >= 9 ) {
    return;
} // if

int ? = partition(Vec, left, right);
quick_sort(Vec, 0, ?);
quick_sort(Vec, ? + 1, 9);
```

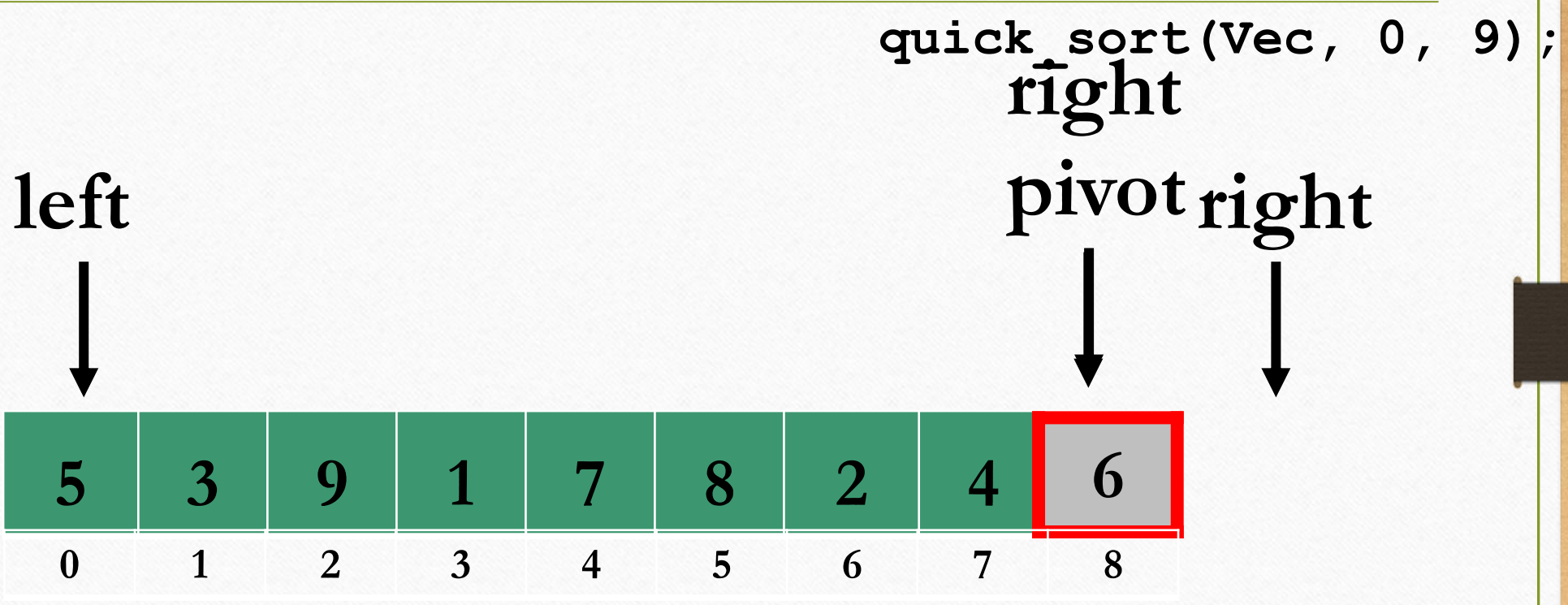
VII. Quicksort



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

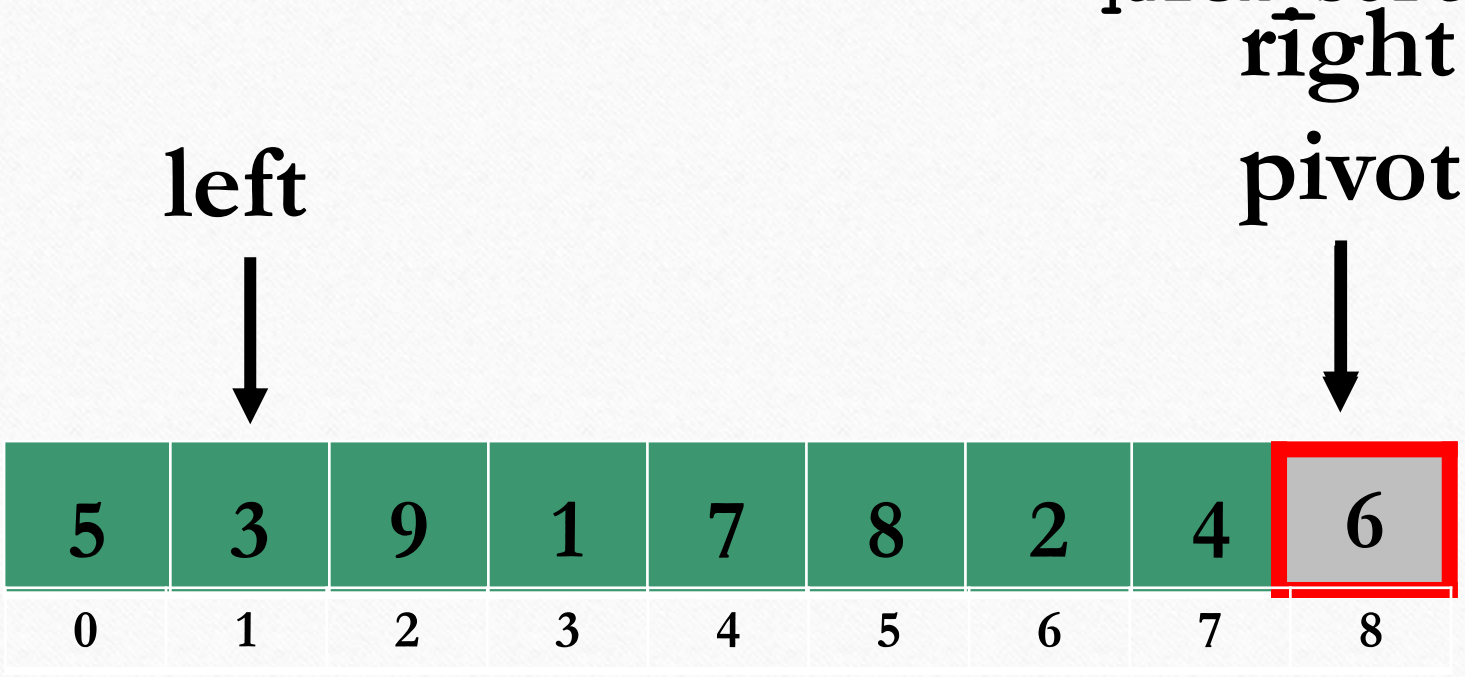


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 9);
```

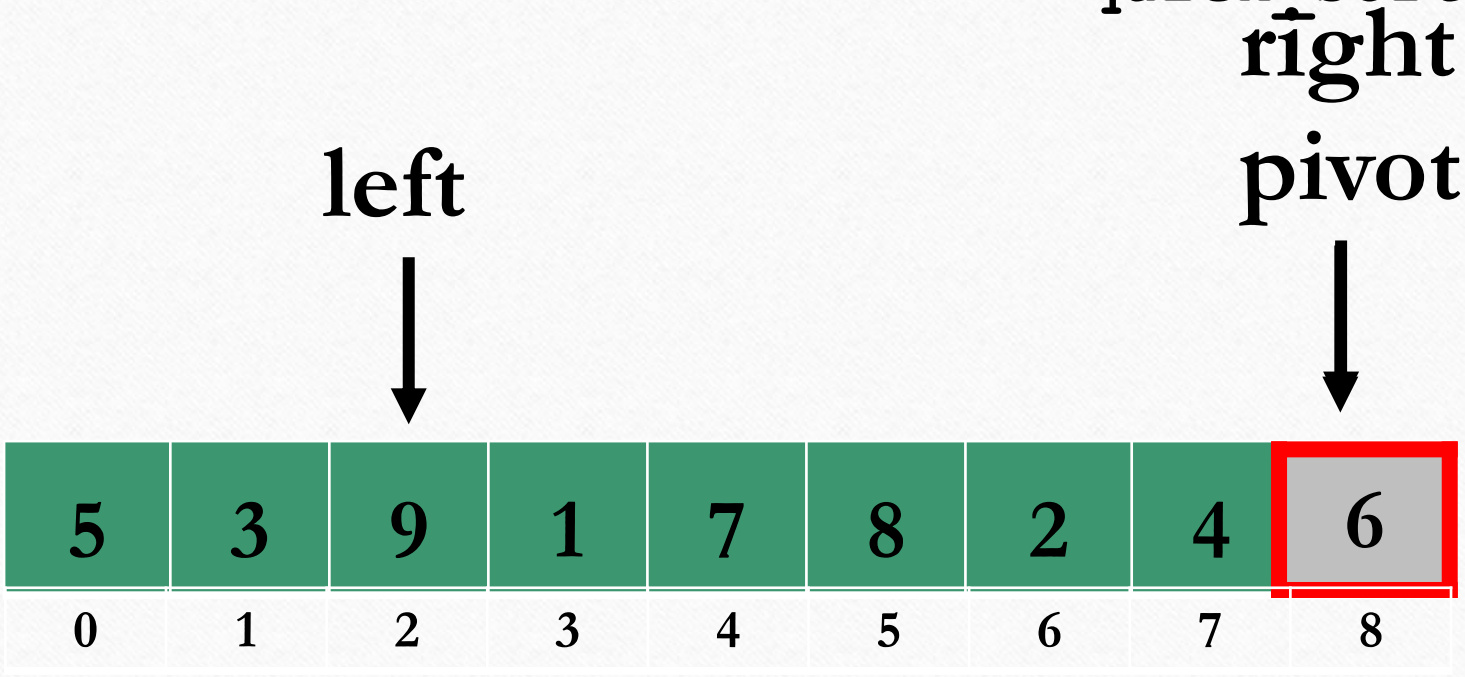


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 9);
```

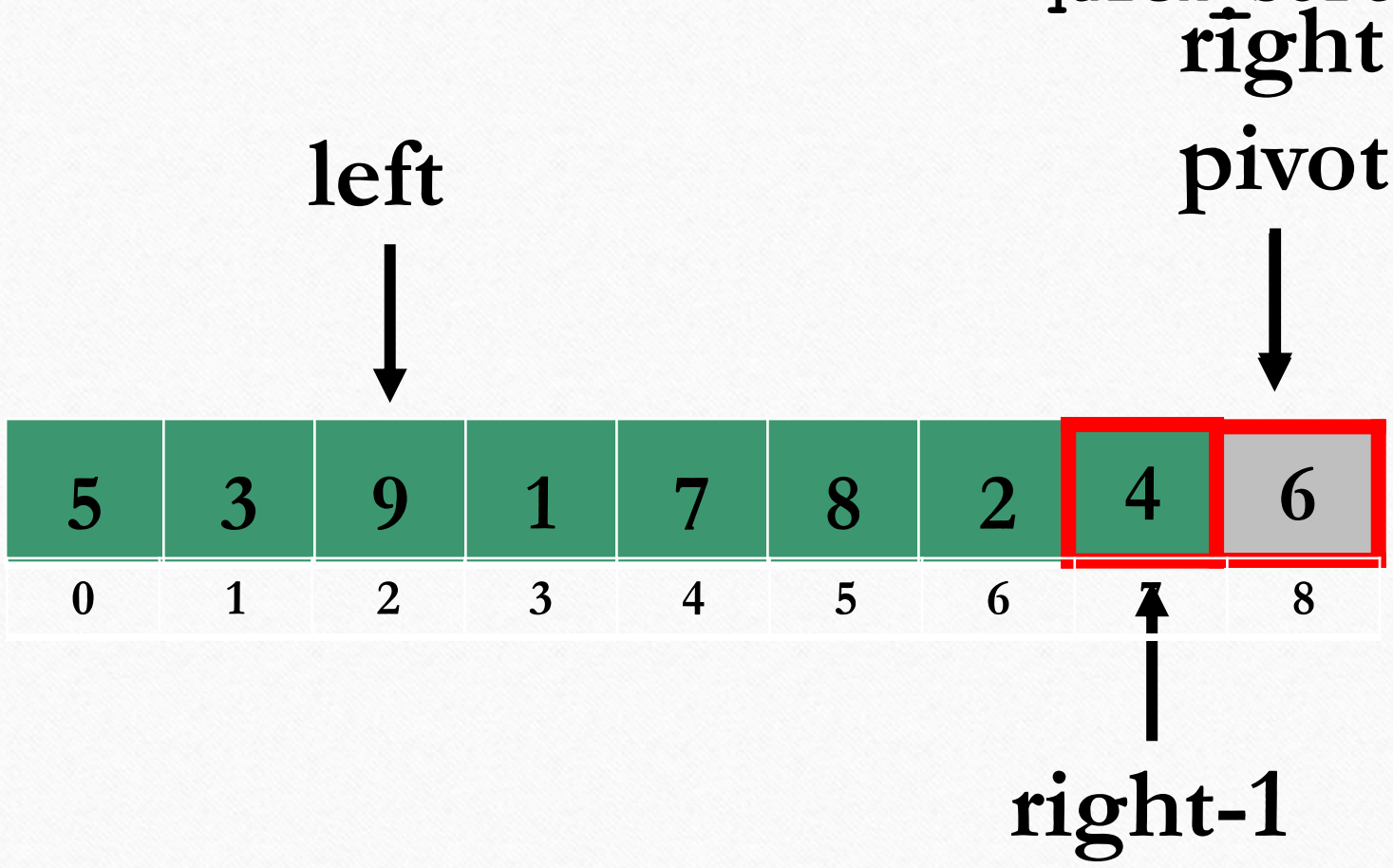


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

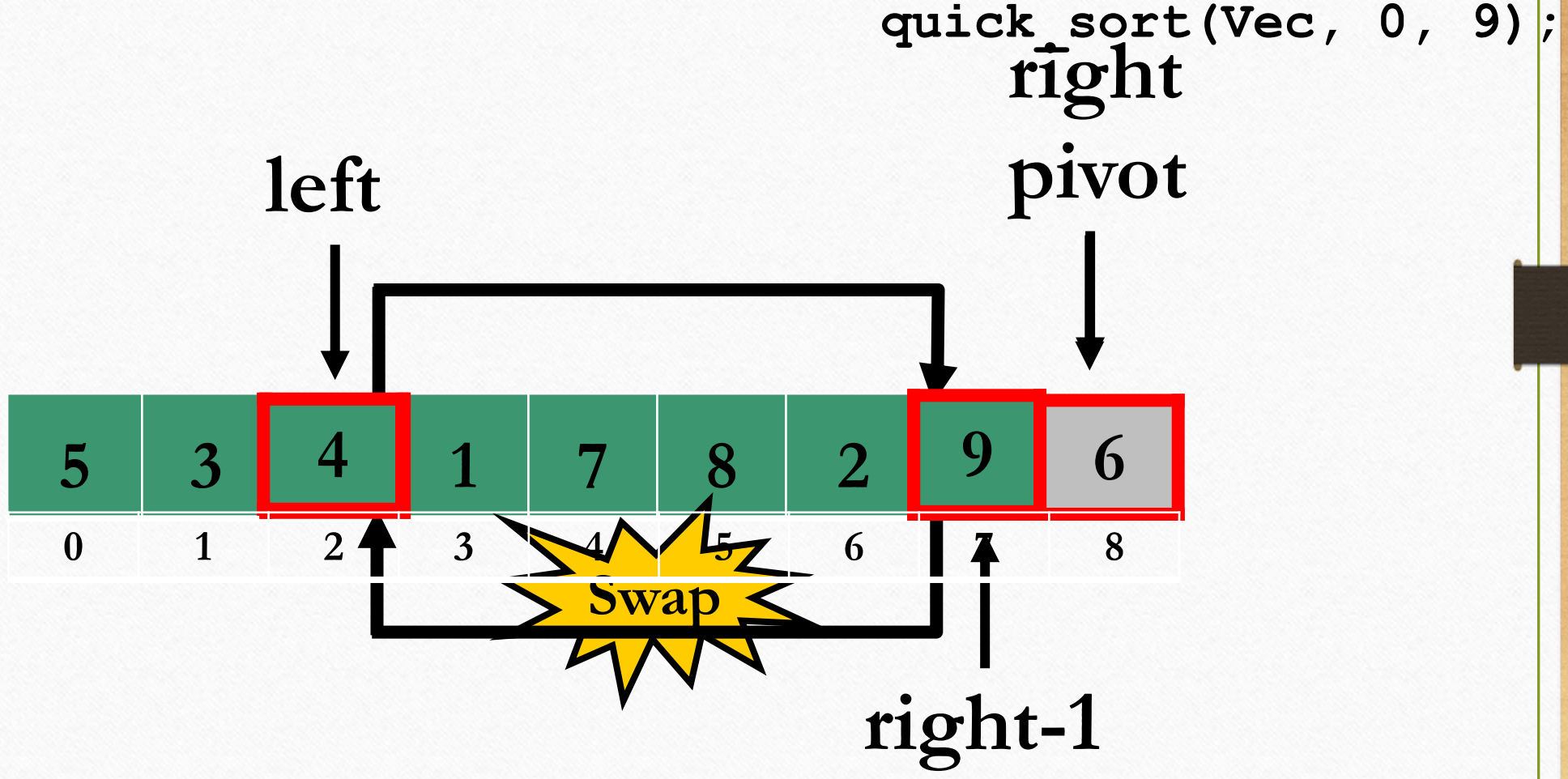
```
quick_sort(Vec, 0, 9);
```



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

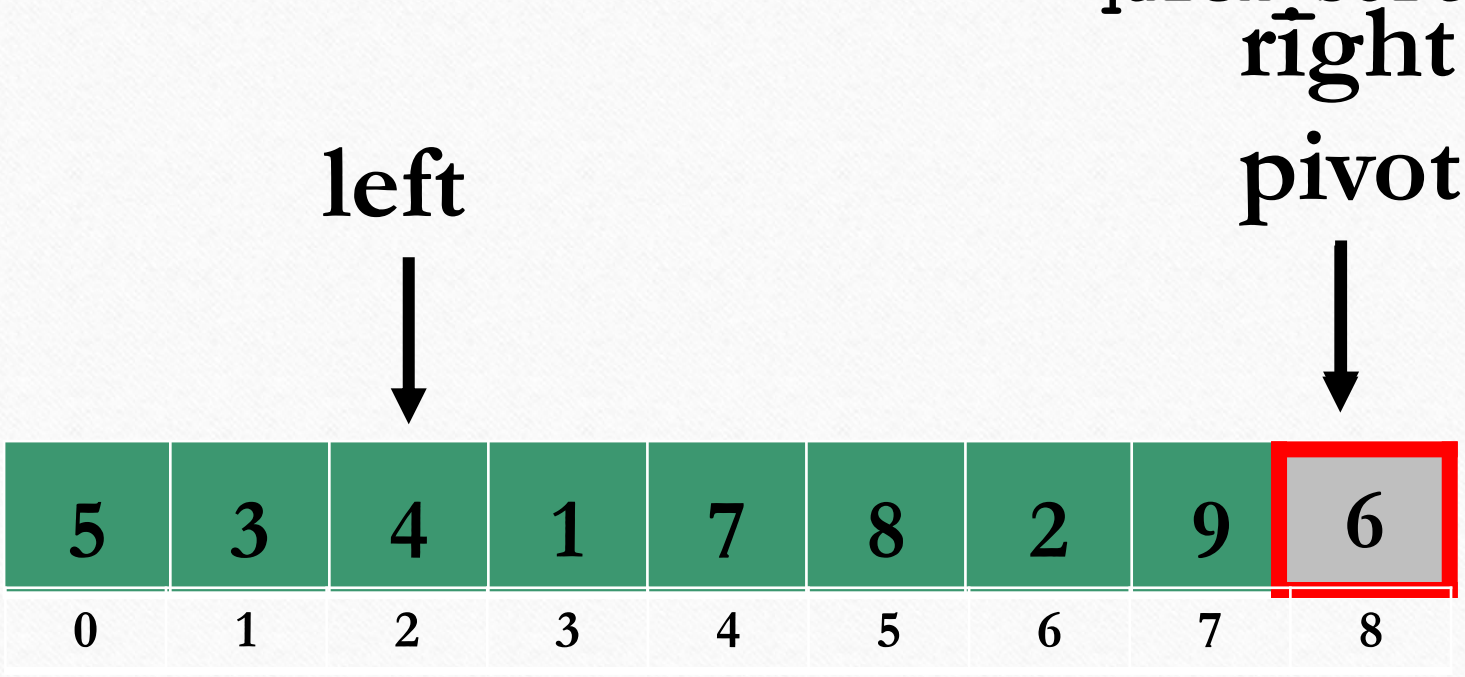


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 9);
```

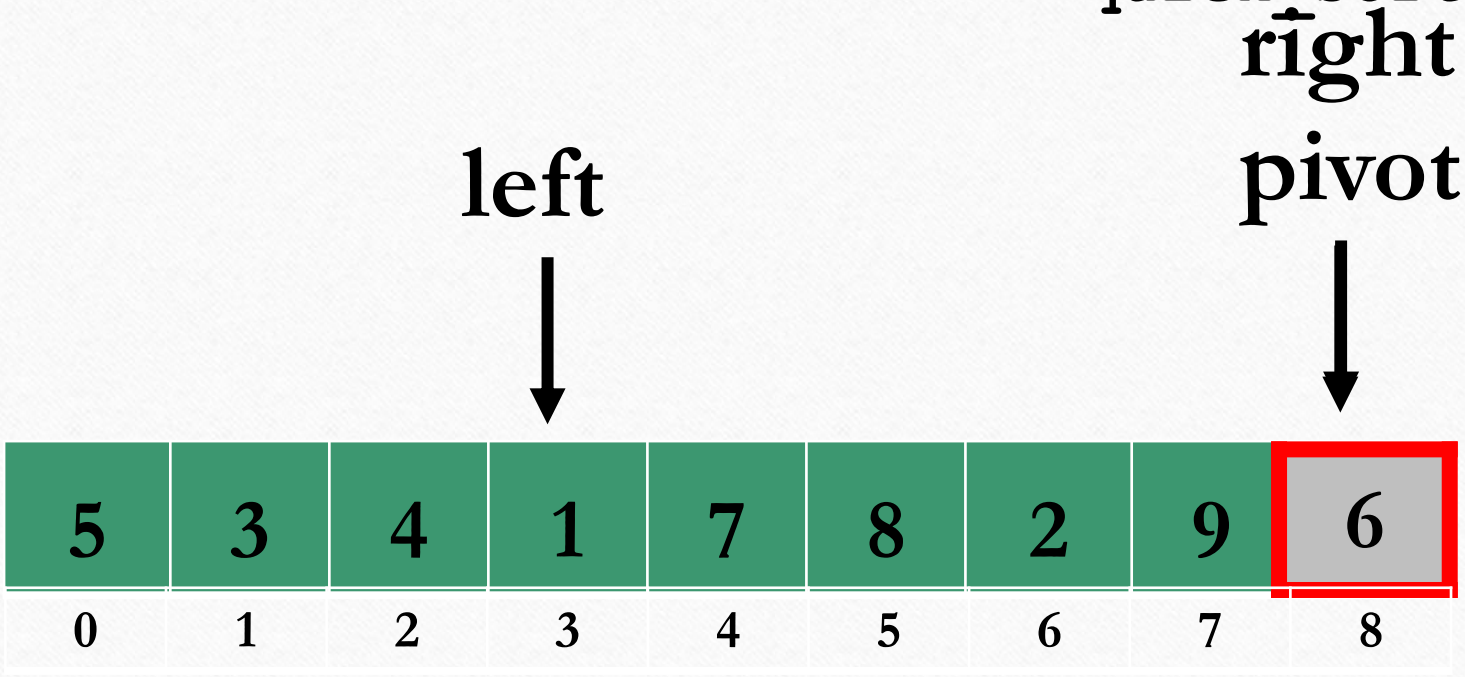


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 9);
```

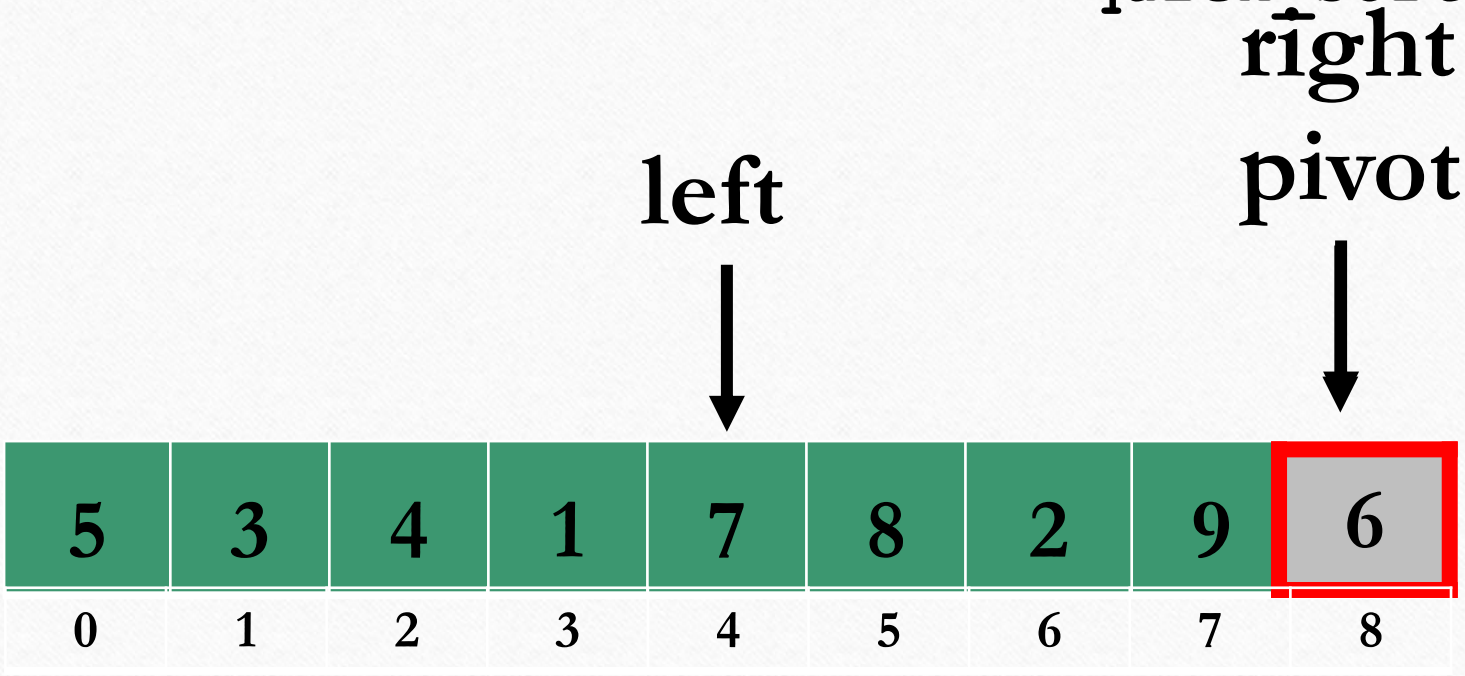


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

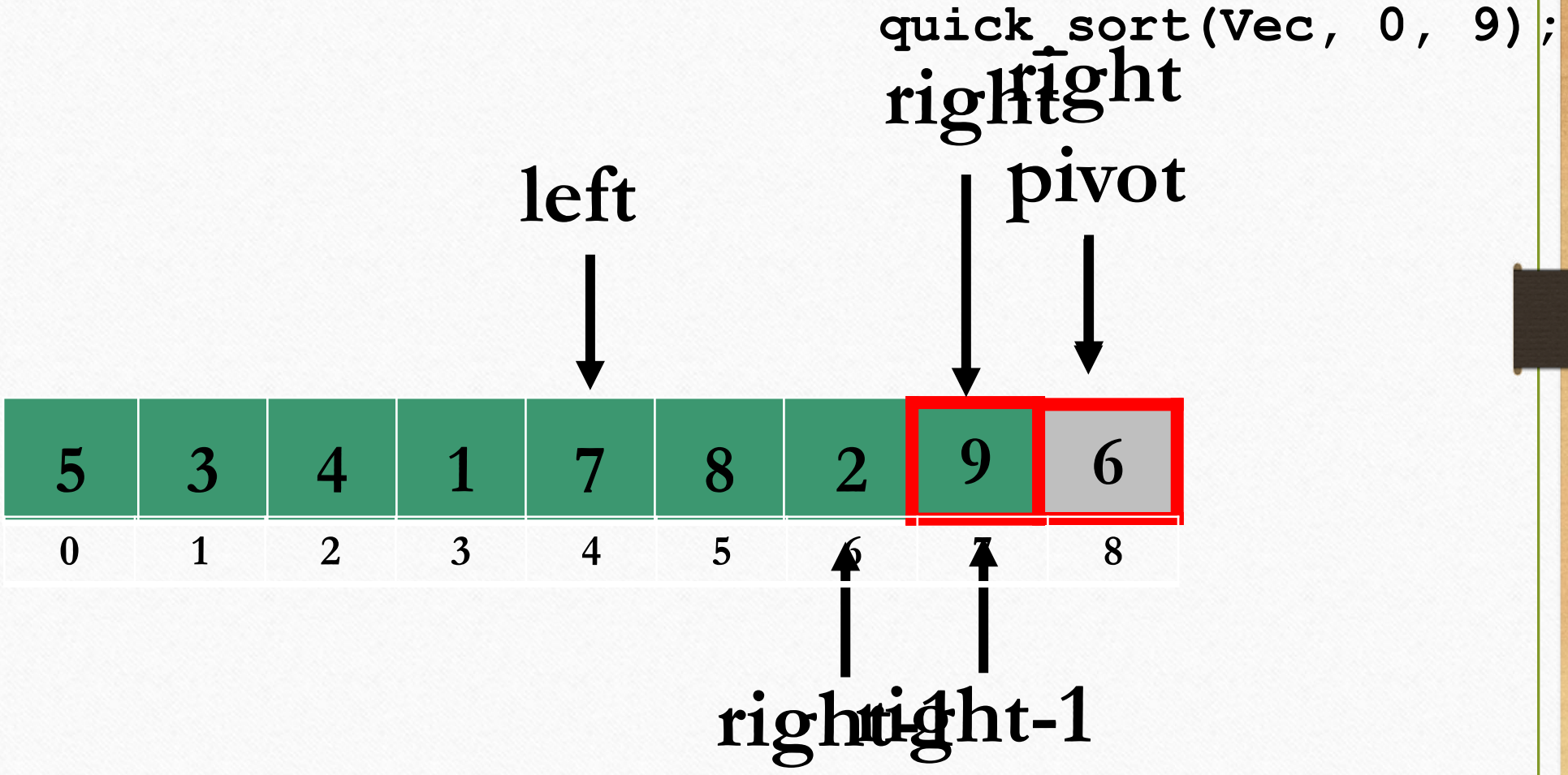
```
quick_sort(Vec, 0, 9);
```



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

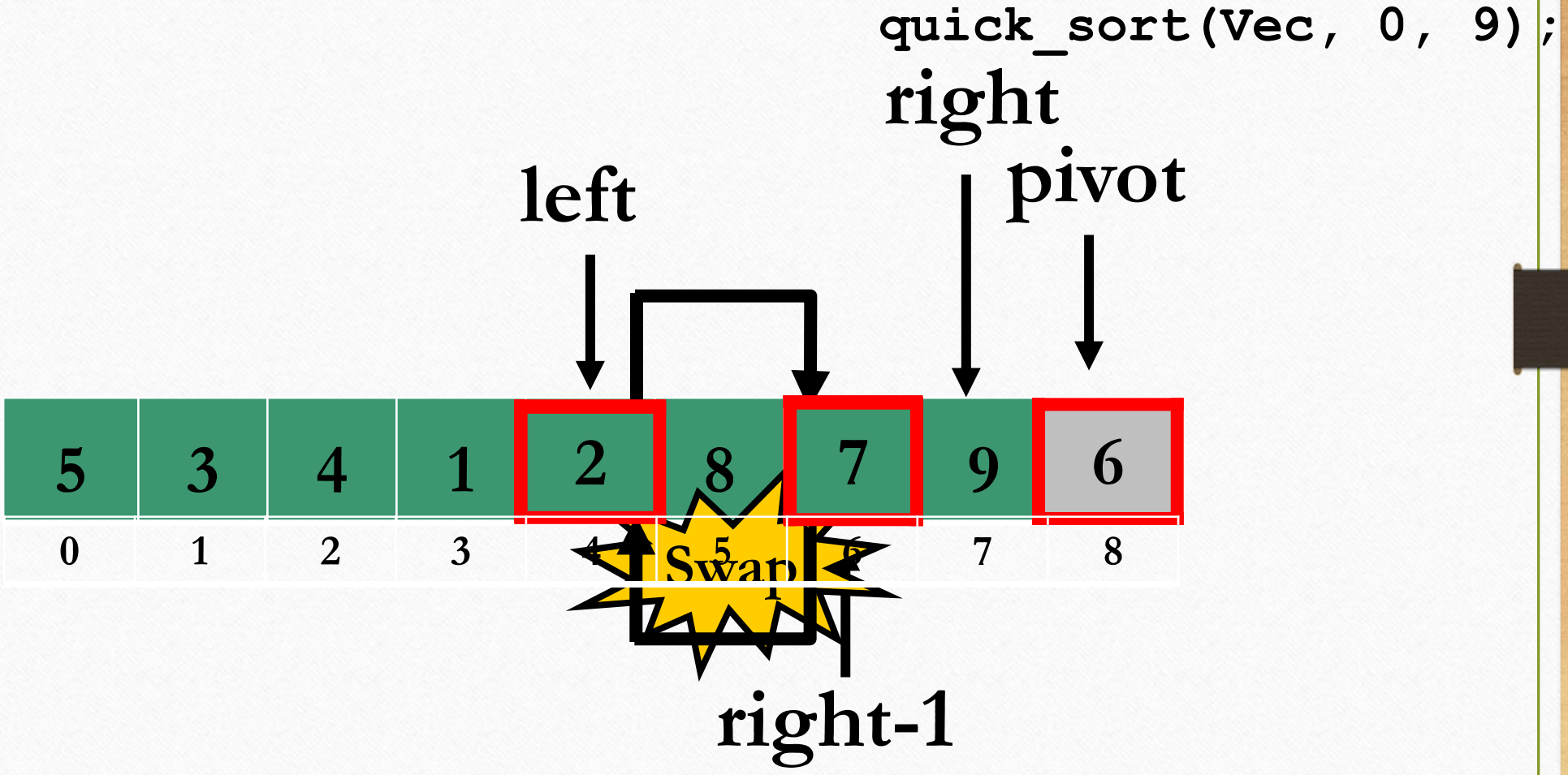
```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

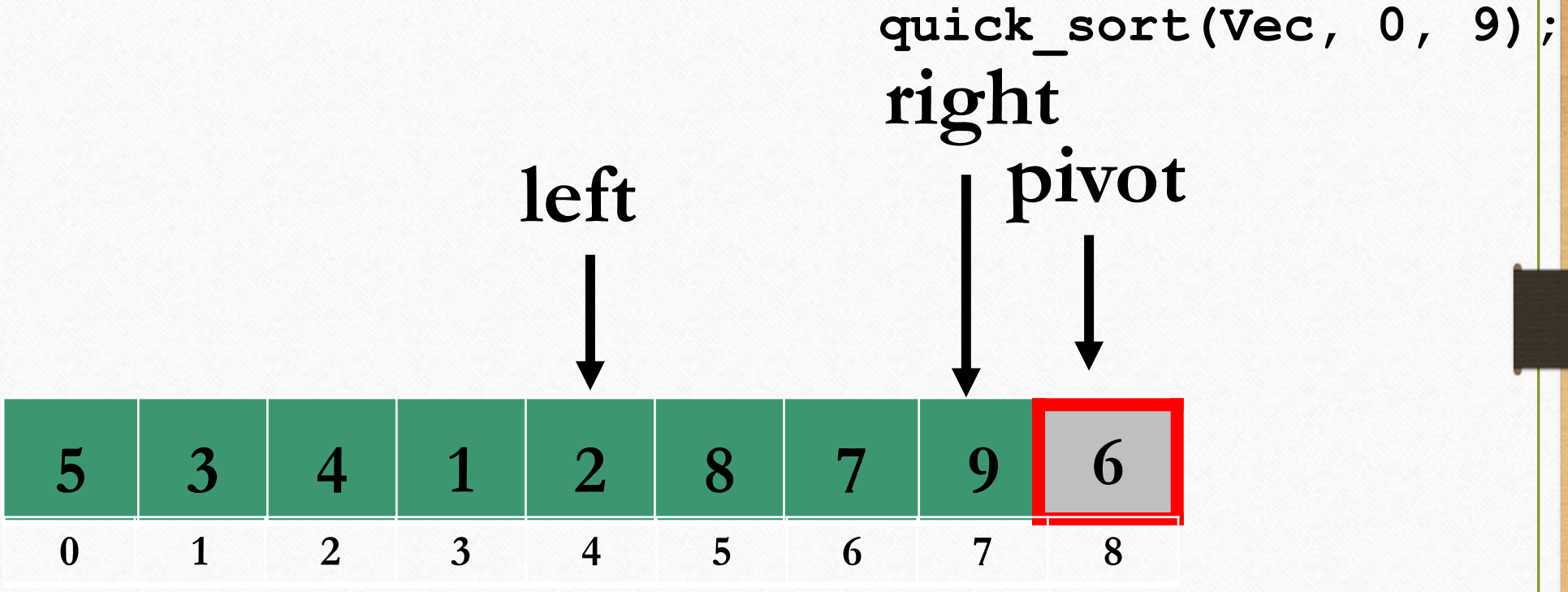
```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

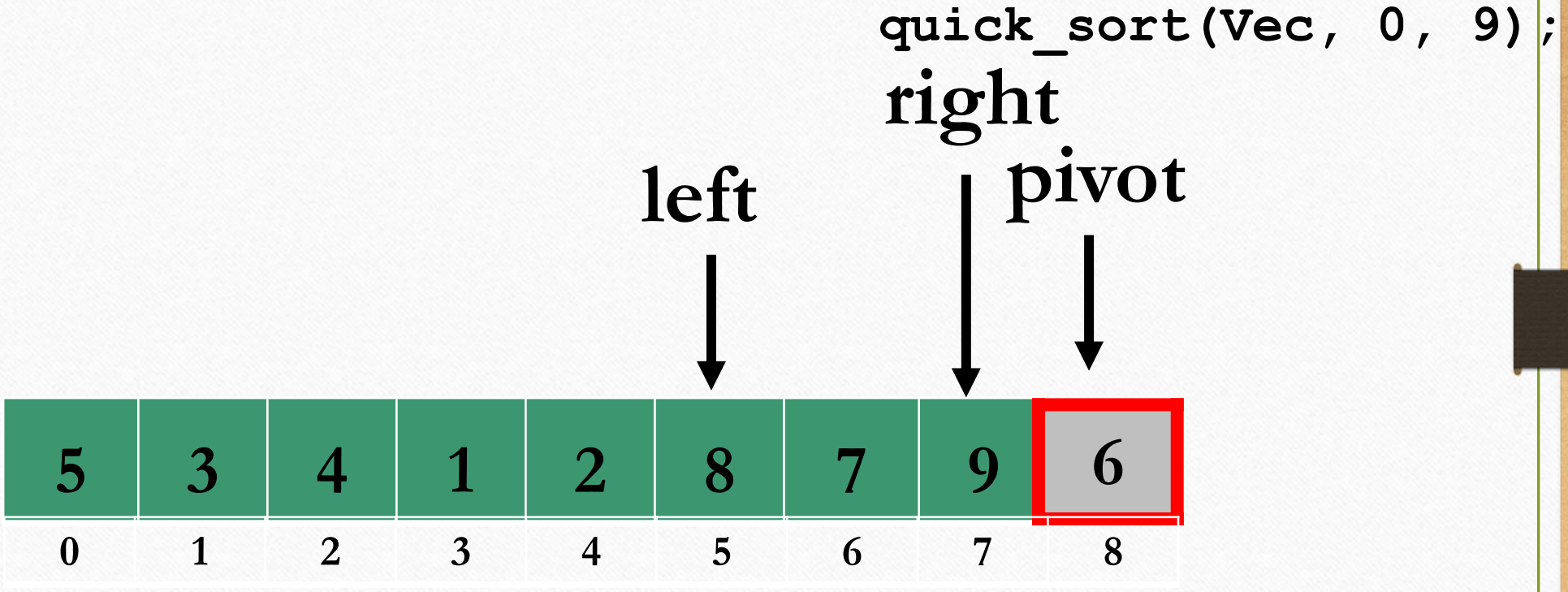
```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

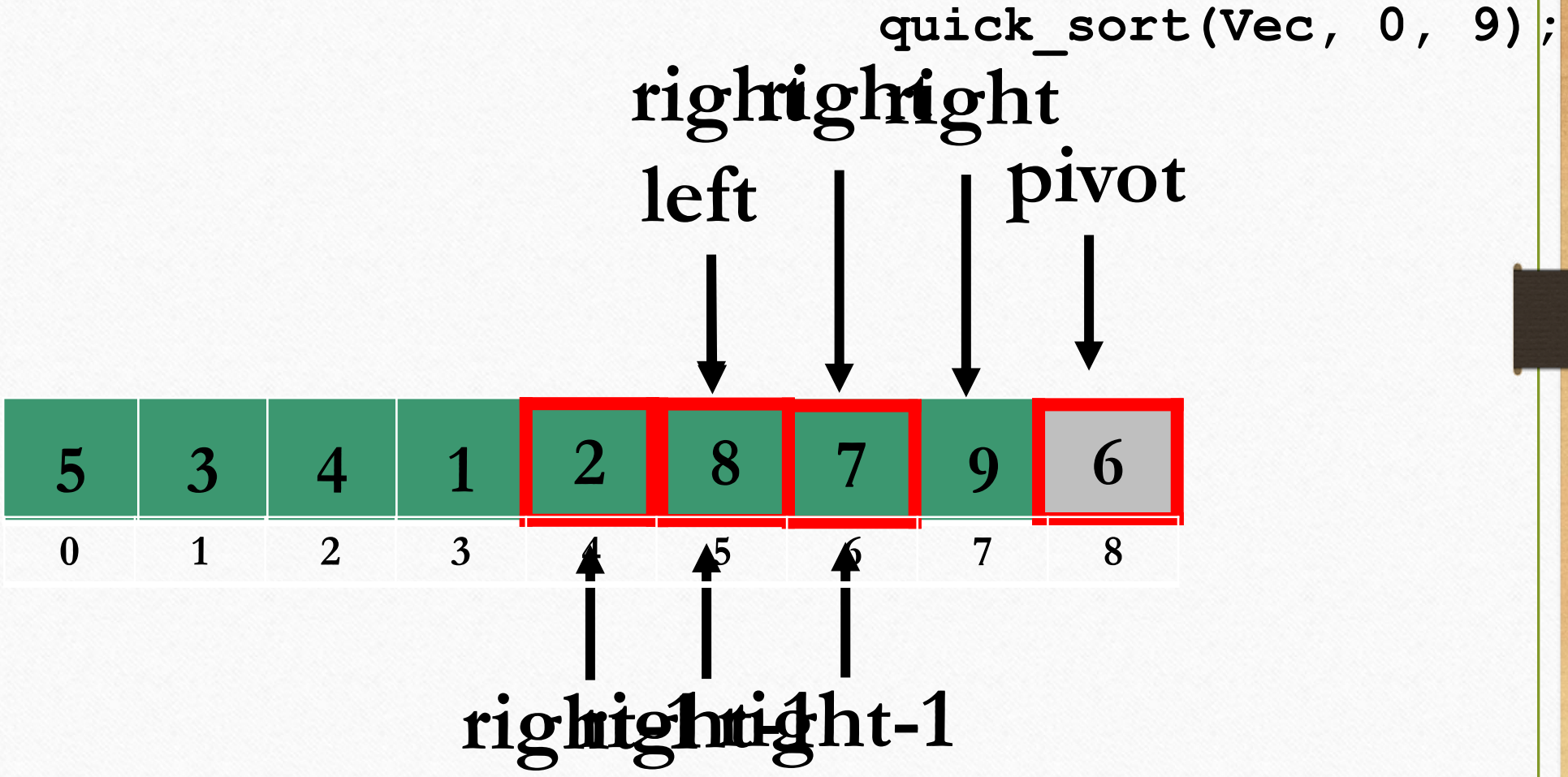
```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

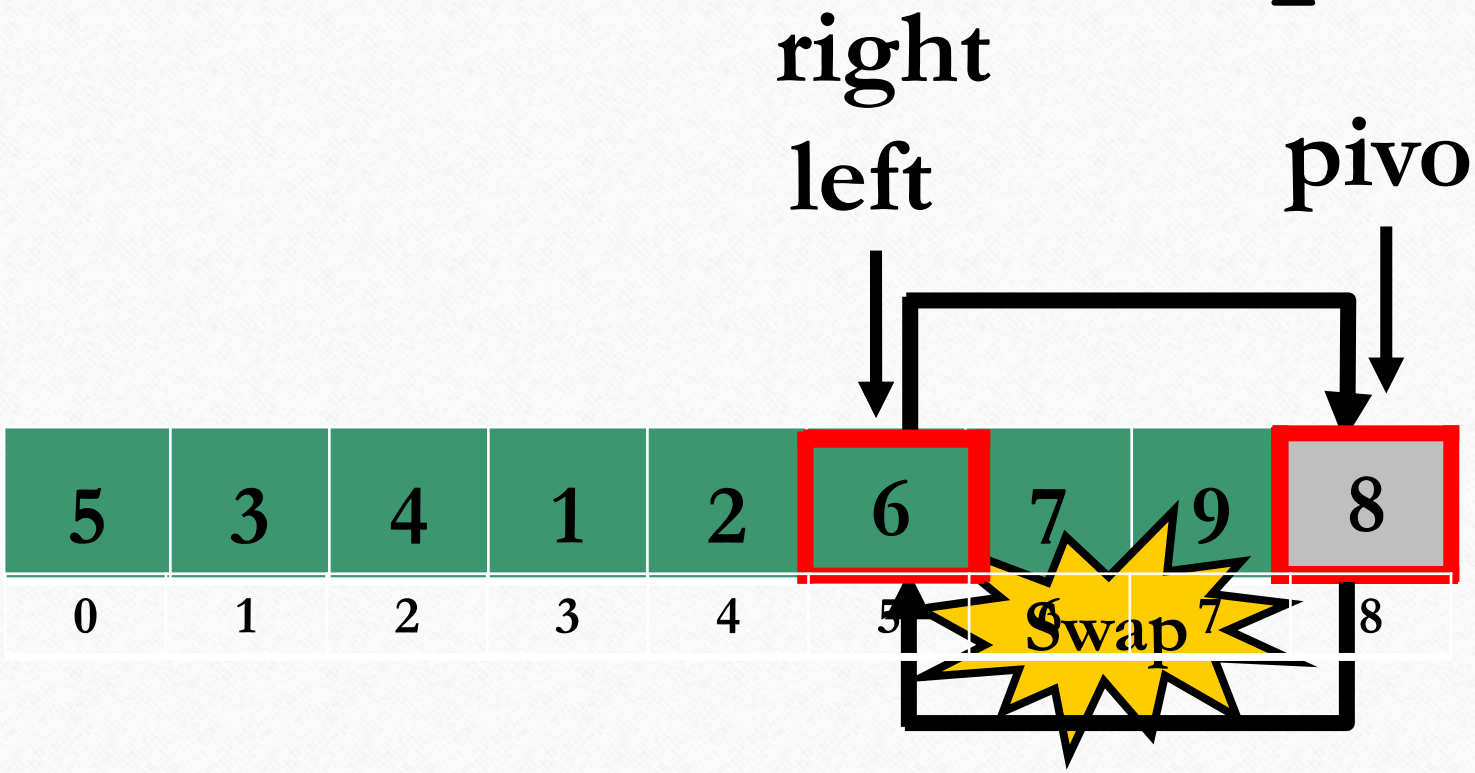


```
int pivot = right--;
 while (true) {
     while (Vec[left] < Vec[pivot]) {left++;}
     while (left < right && Vec[right - 1] >= Vec[pivot]){
             right--;}
    if (left >= right) { break;}
    swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 9);
```



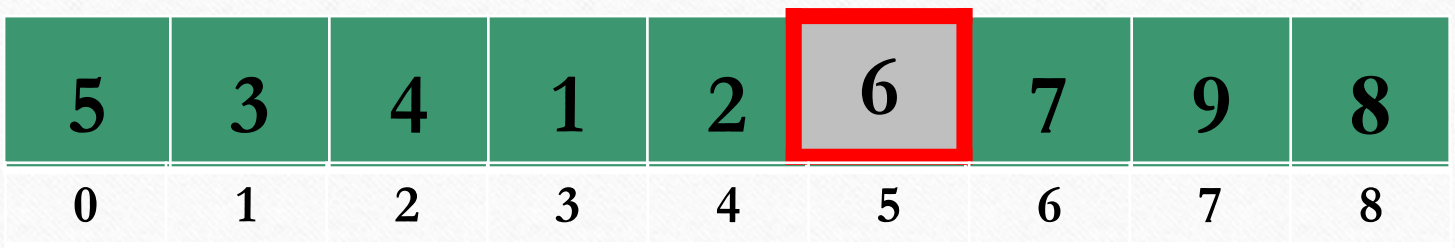
```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

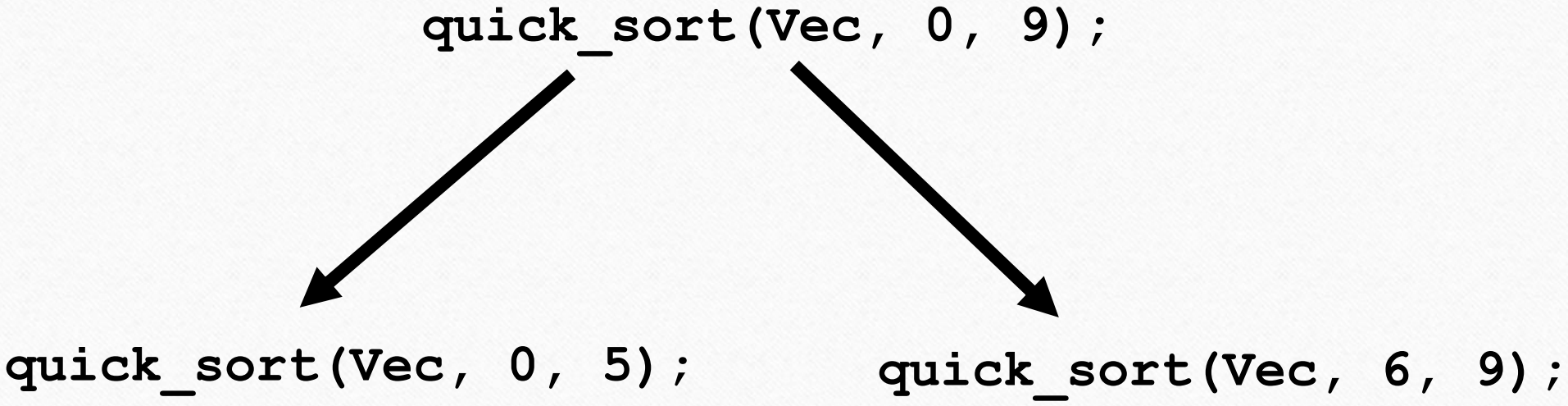
```
quick_sort(Vec, 0, 9);
```

pivot = left




```
quick_sort(Vec, 0, 5);
quick_sort(Vec, 6, 9);
```

VII. Quicksort

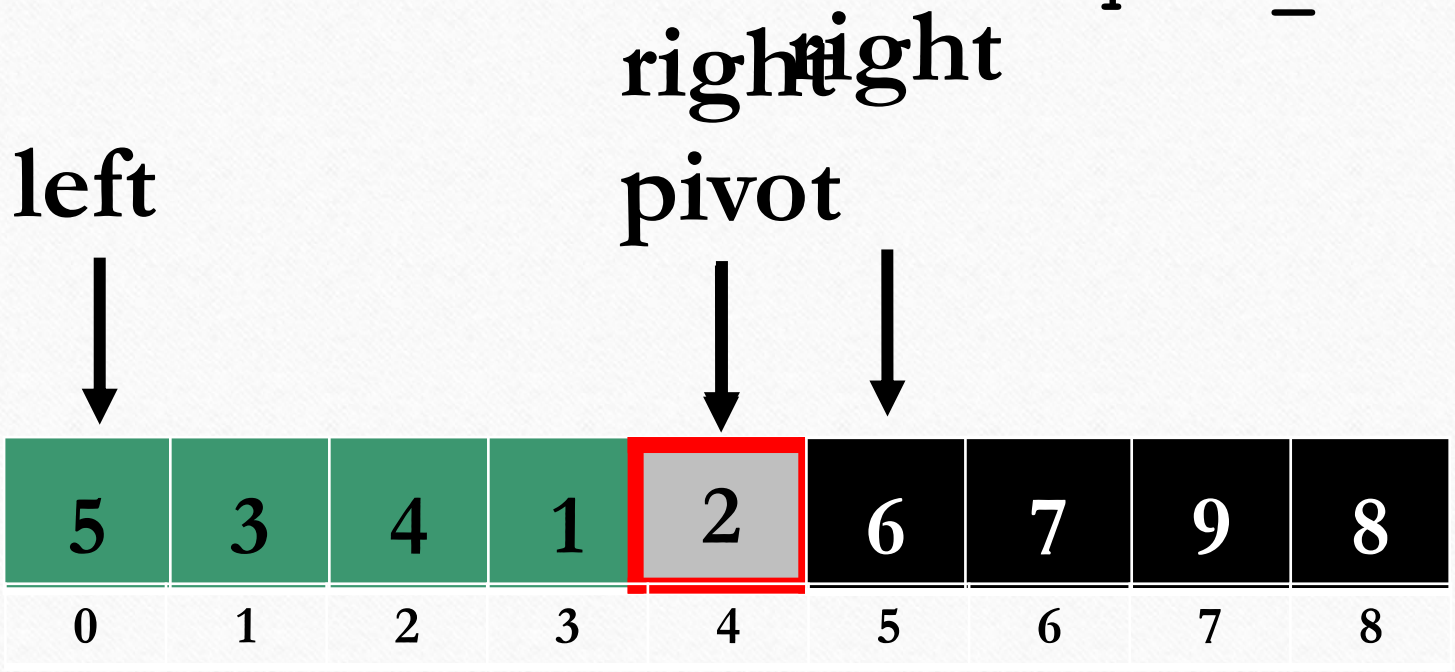


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 5);
```

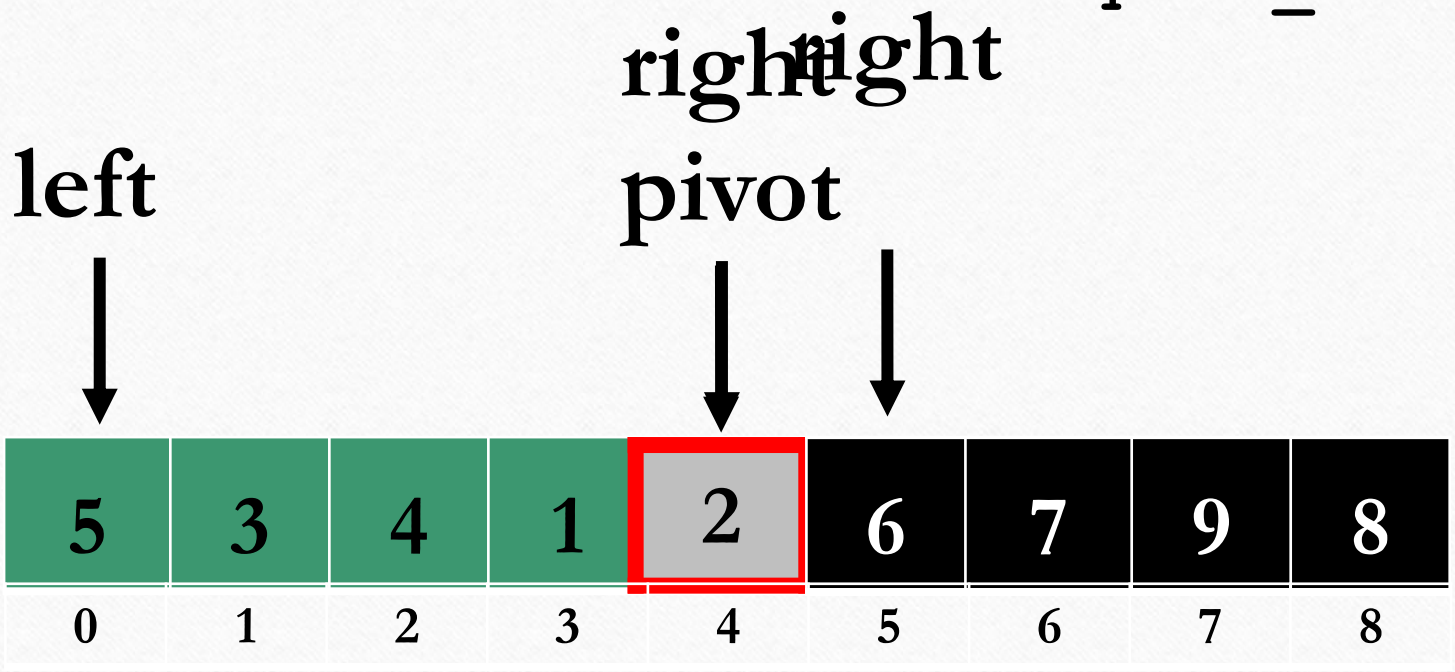


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 5);
```

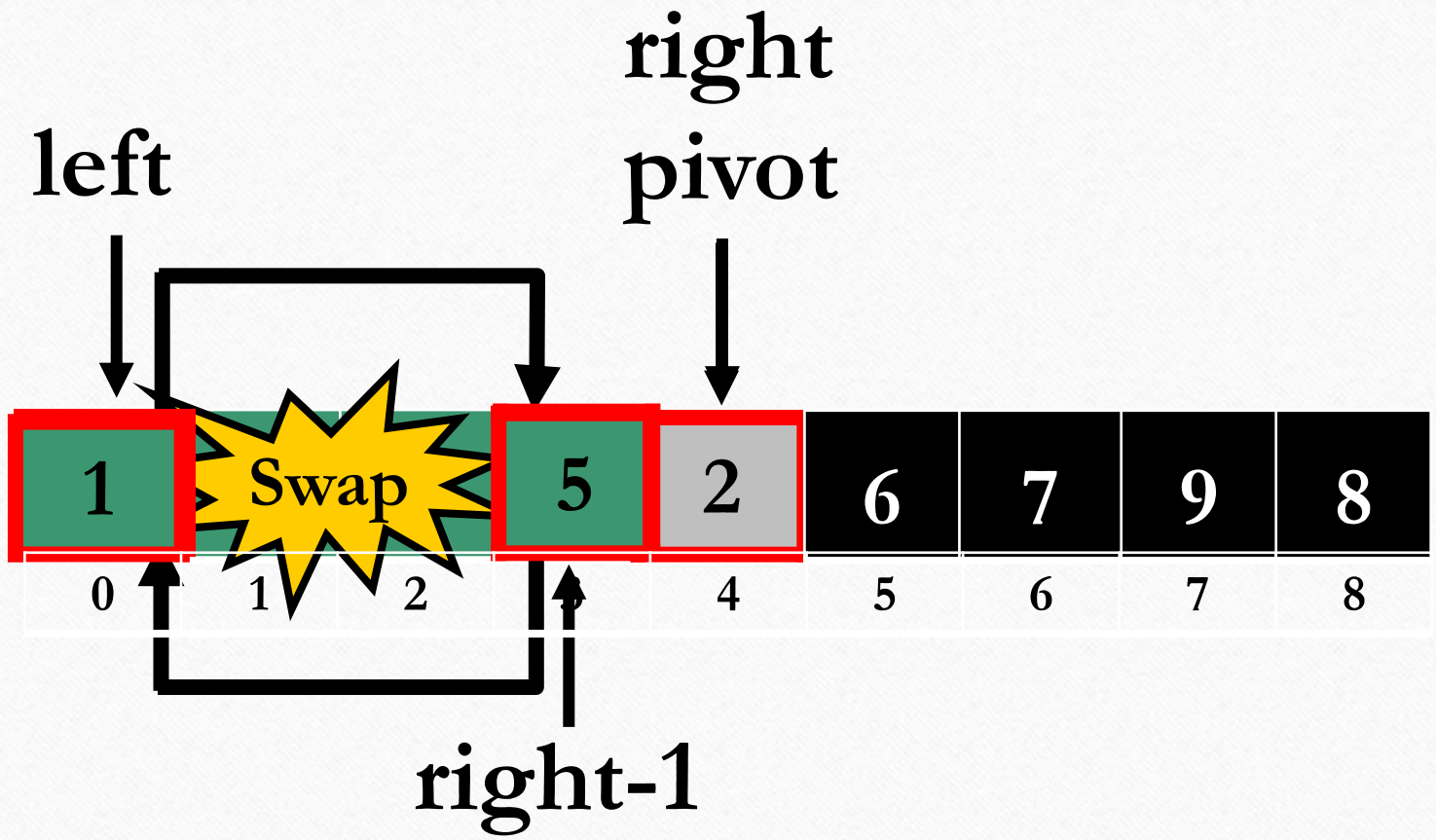


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
     while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
    swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

quick_sort(Vec, 0, 5);

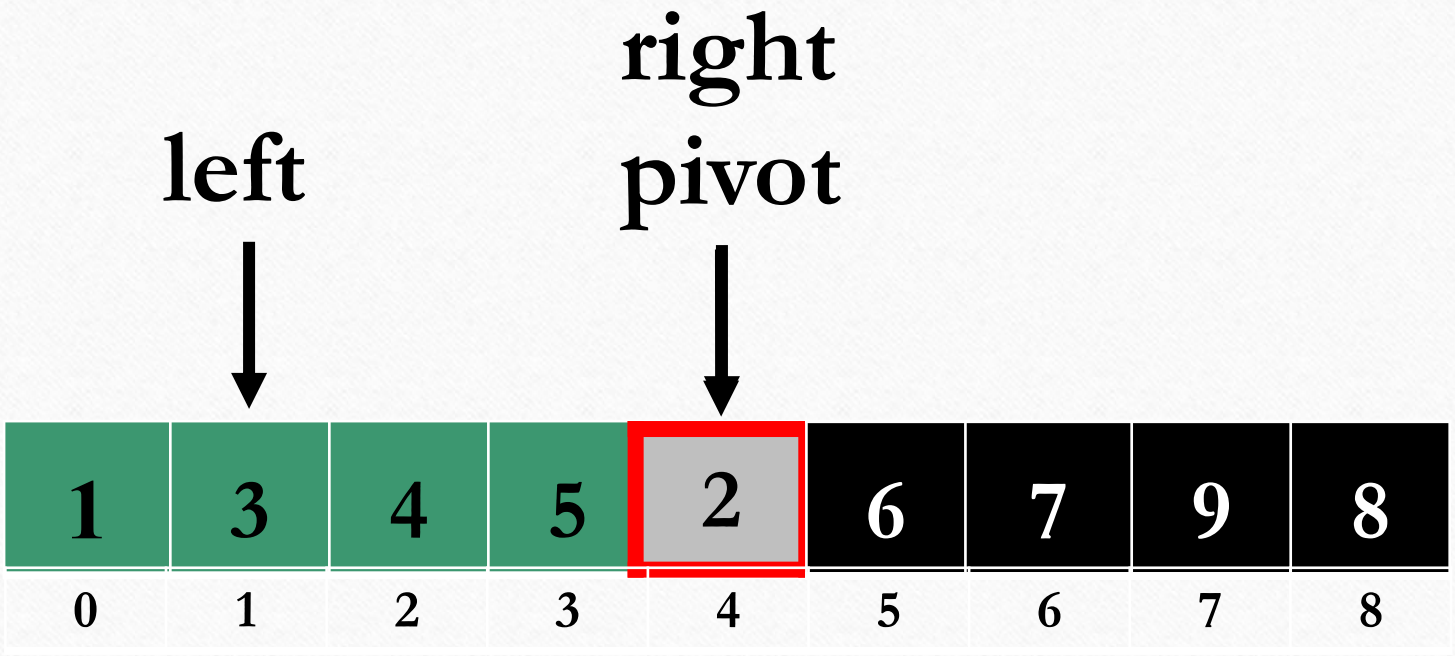


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
     while (left < right && Vec[right - 1] >= Vec[pivot]){
           right--;}
    if (left >= right) { break;}
    swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 5);
```

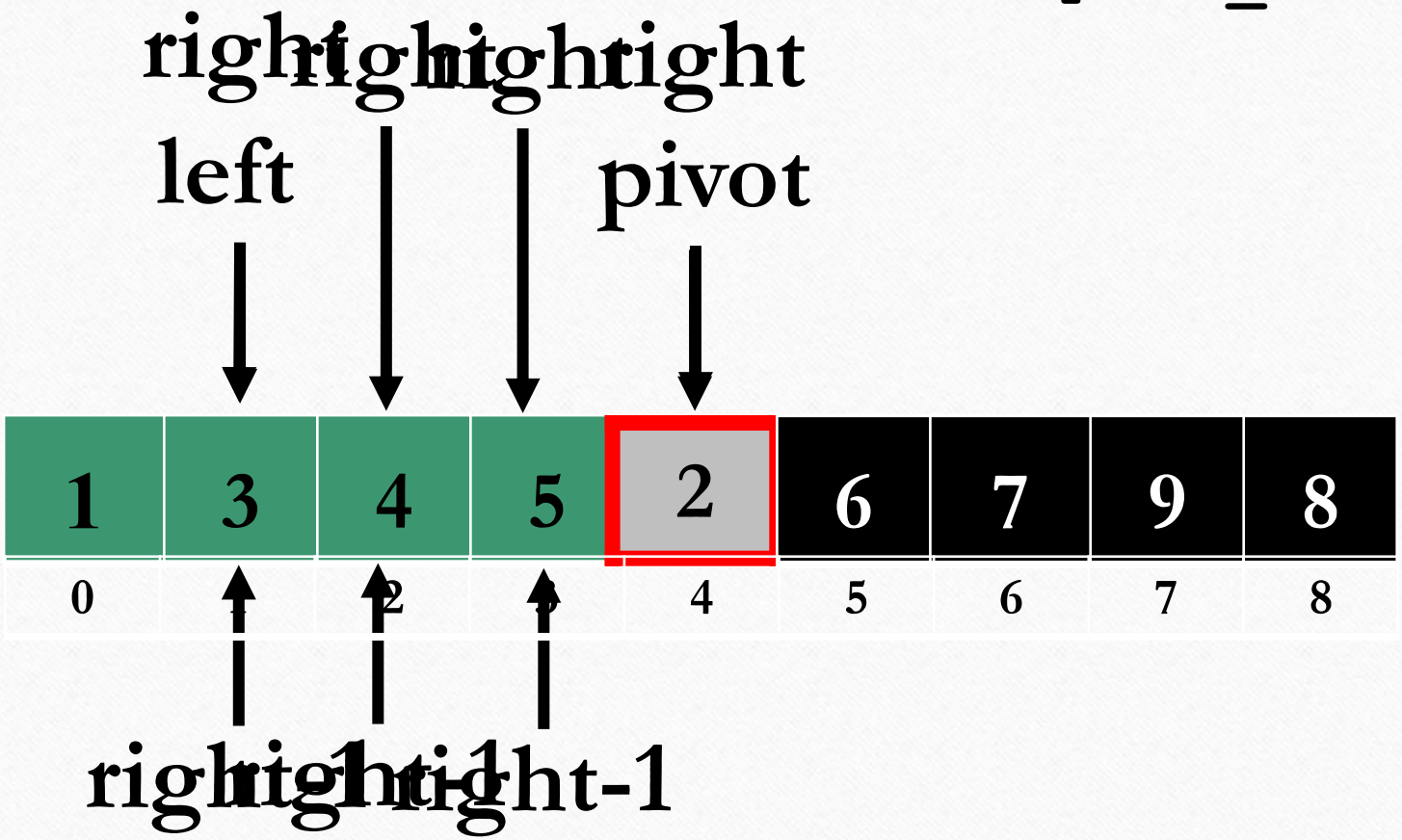


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
    swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 5);
```

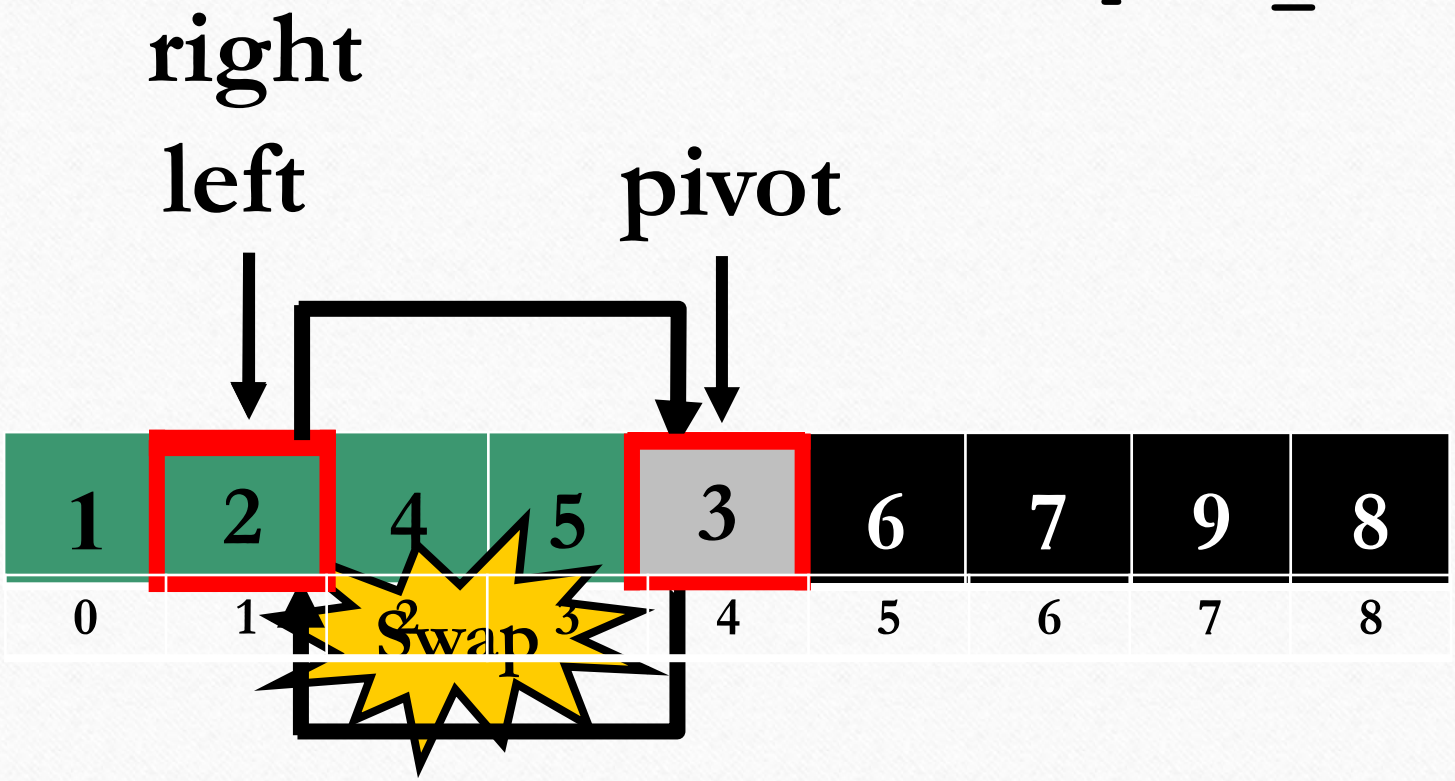


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

quick_sort(Vec, 0, 5);



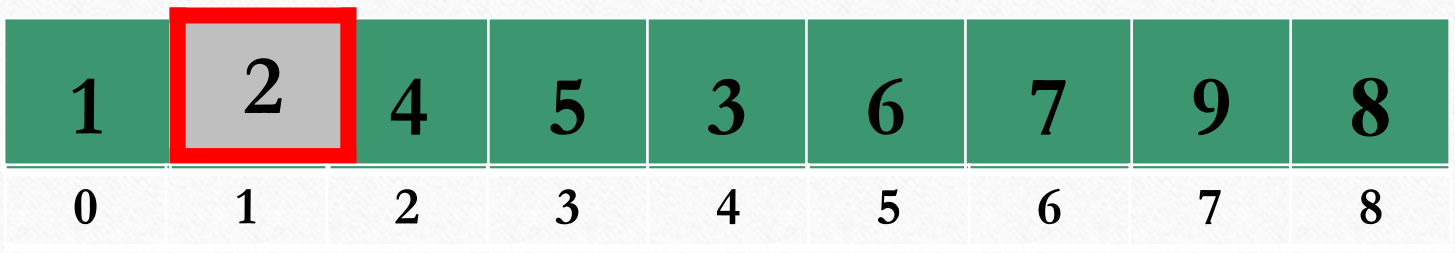
```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

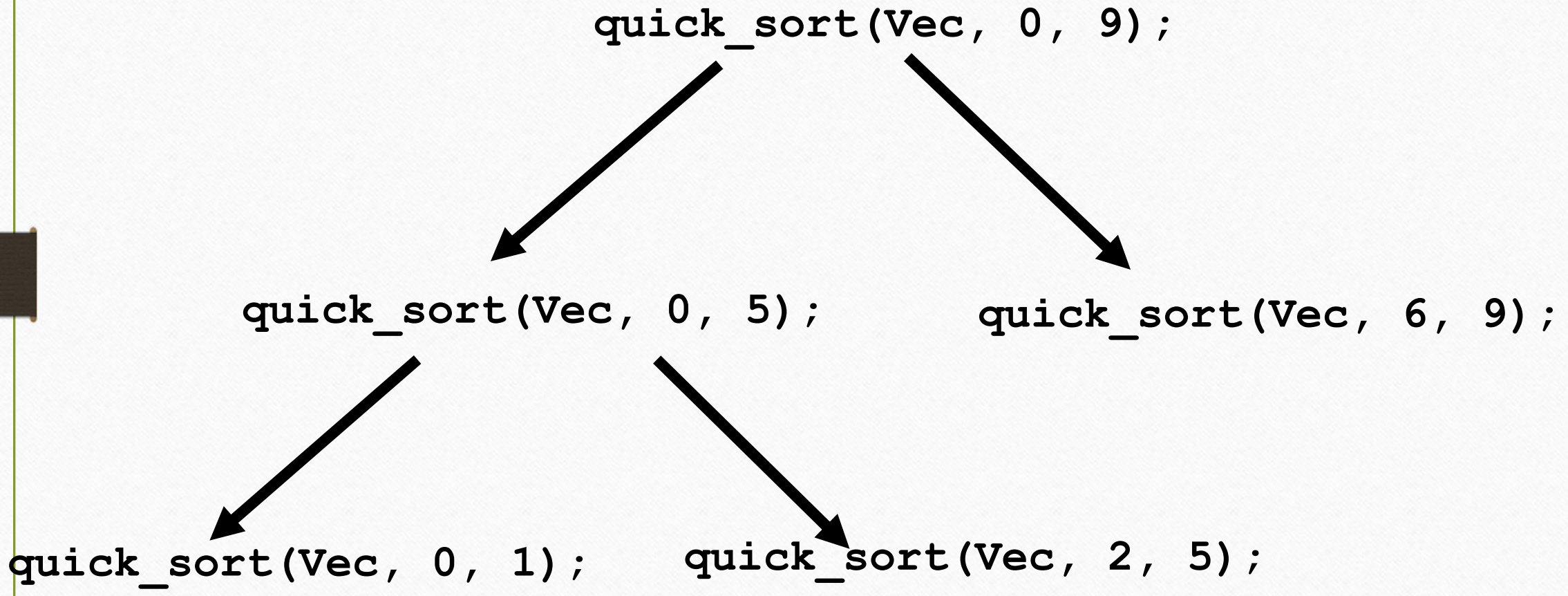
```
quick_sort(Vec, 0, 5);
```

pivot = left




```
quick_sort(Vec, 0, 1);
quick_sort(Vec, 2, 5);
```

VII. Quicksort

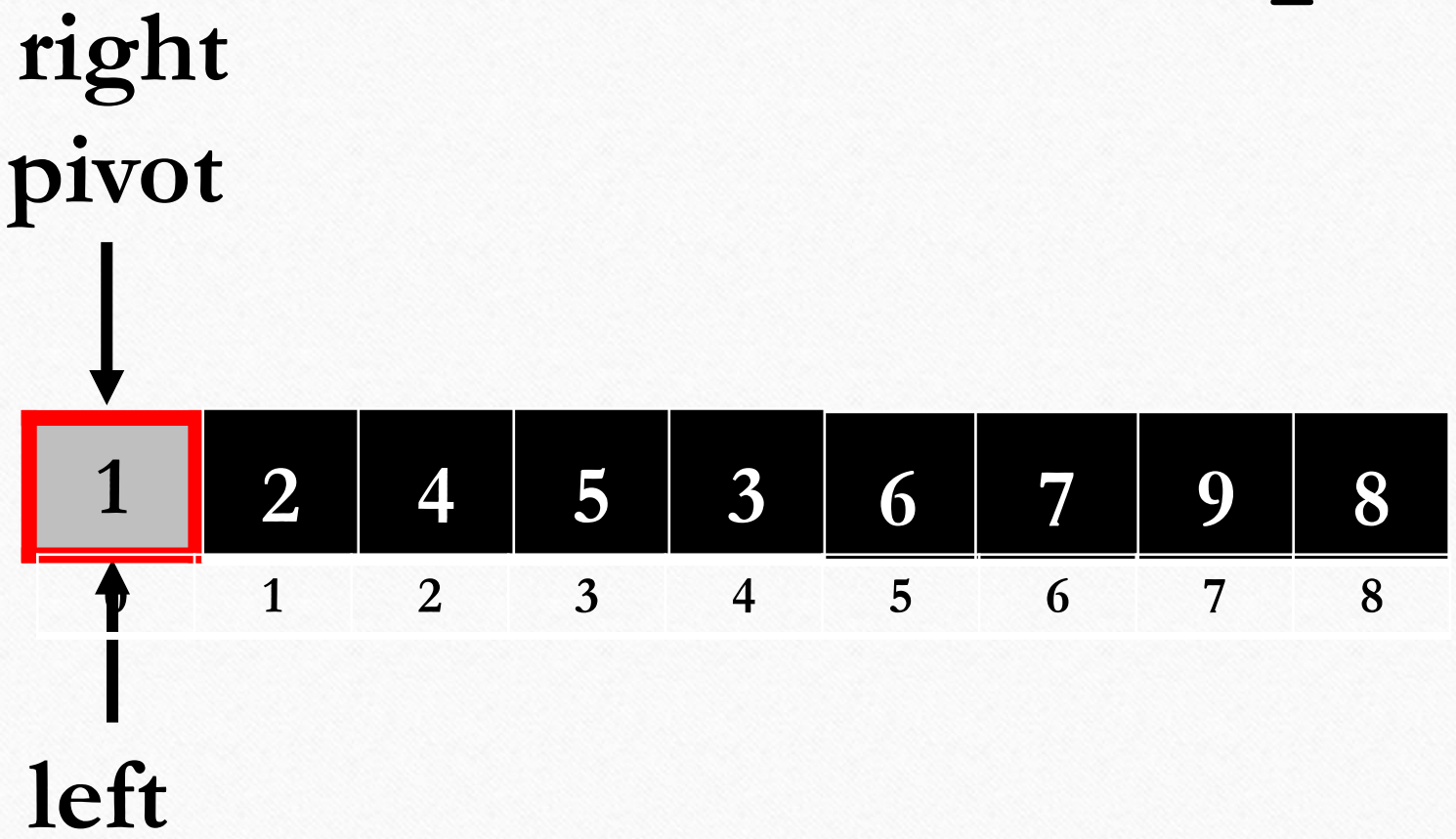


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 0, 1);
```

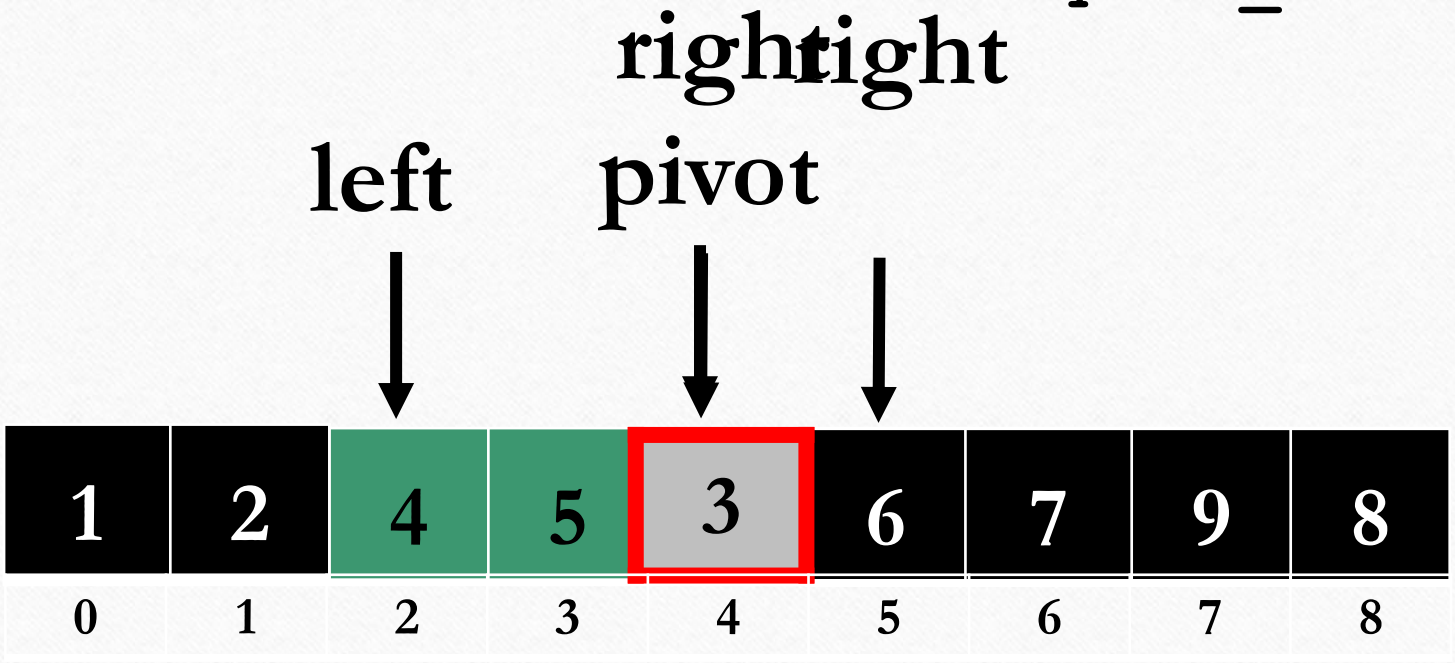


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 2, 5);
```

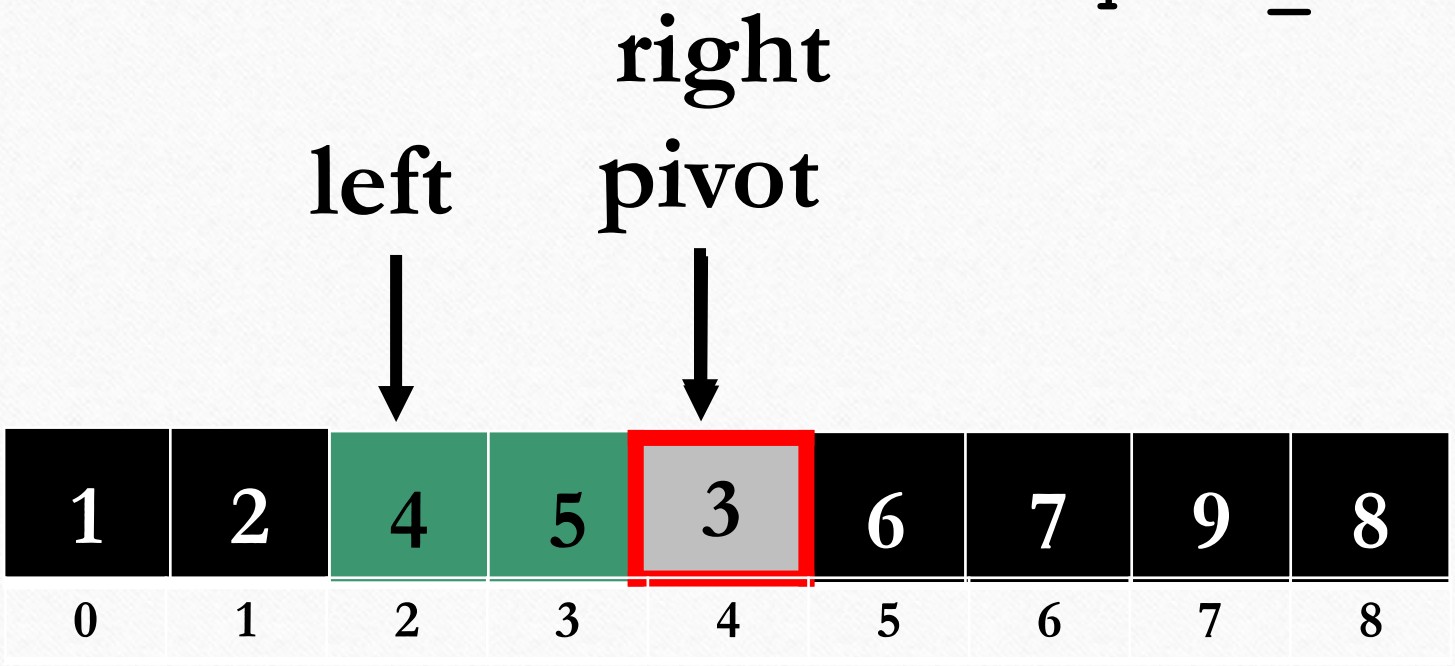


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 2, 5);
```

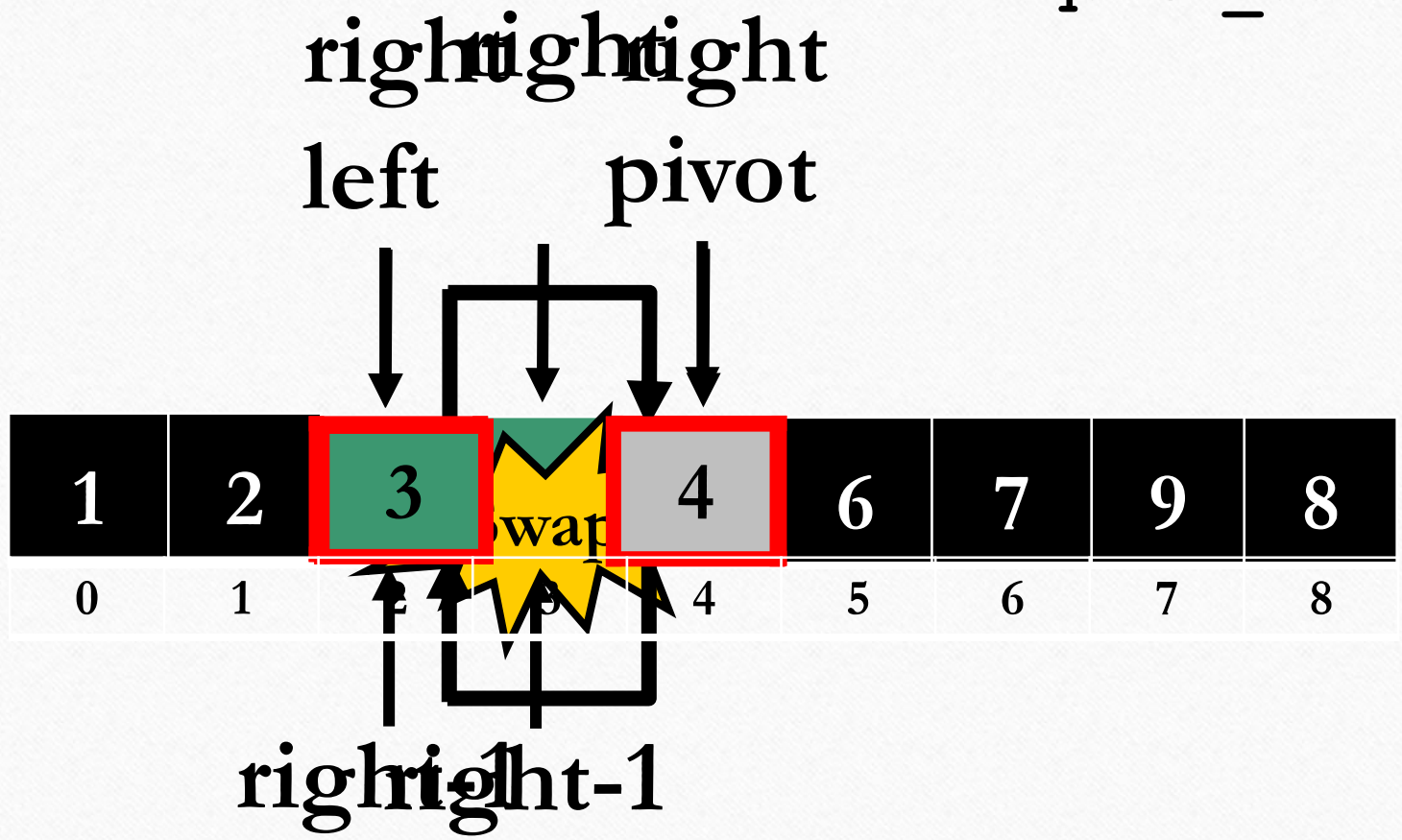


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

VII. Quicksort

```
quick_sort(Vec, 2, 5);
```



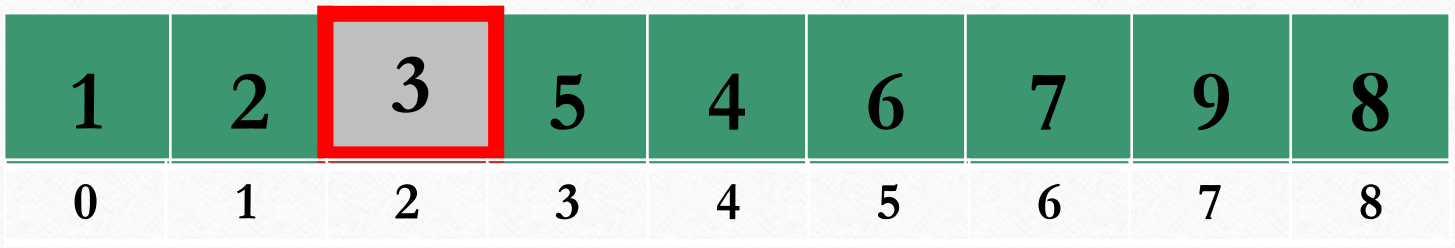
```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
            right--;}
   if (left >= right) { break;}
   swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

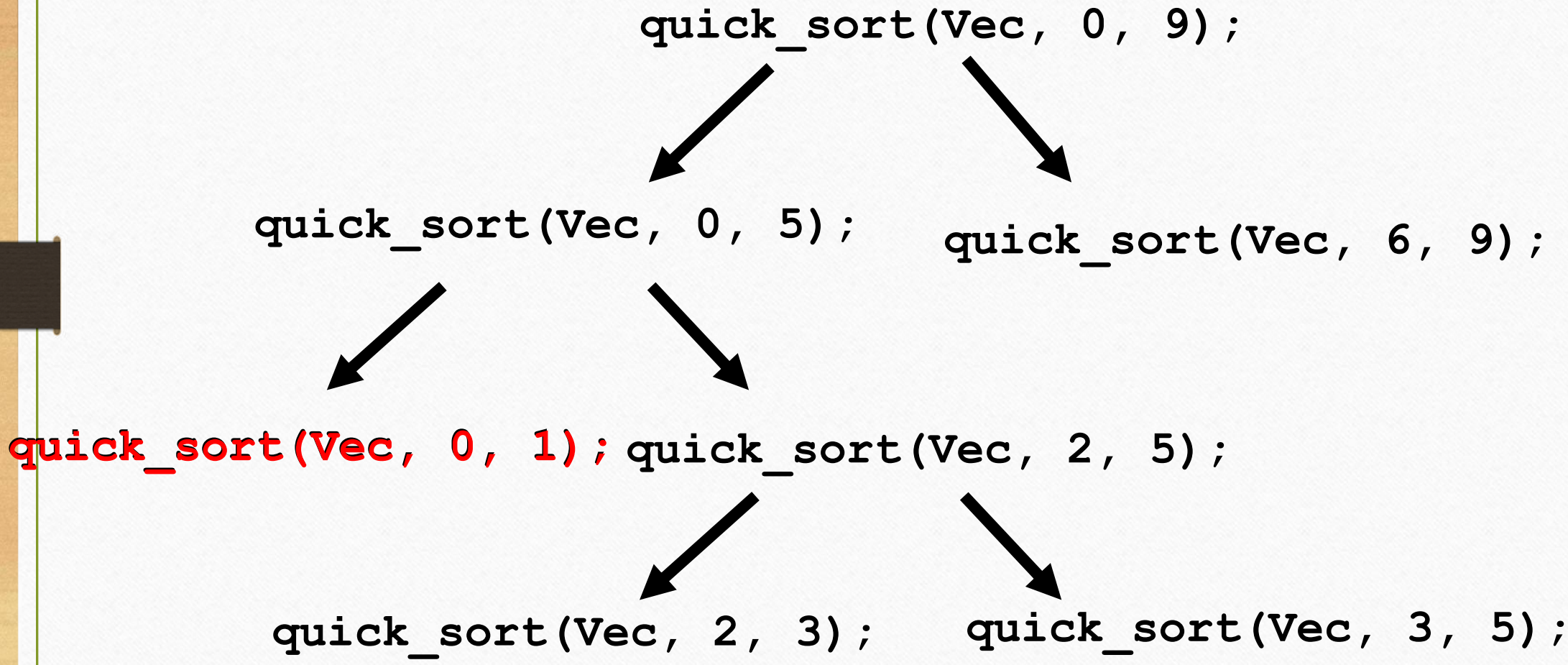
```
quick_sort(Vec, 2, 5);
```

pivot = left




```
quick_sort(Vec, 2, 3);
quick_sort(Vec, 3, 5);
```

VII. Quicksort

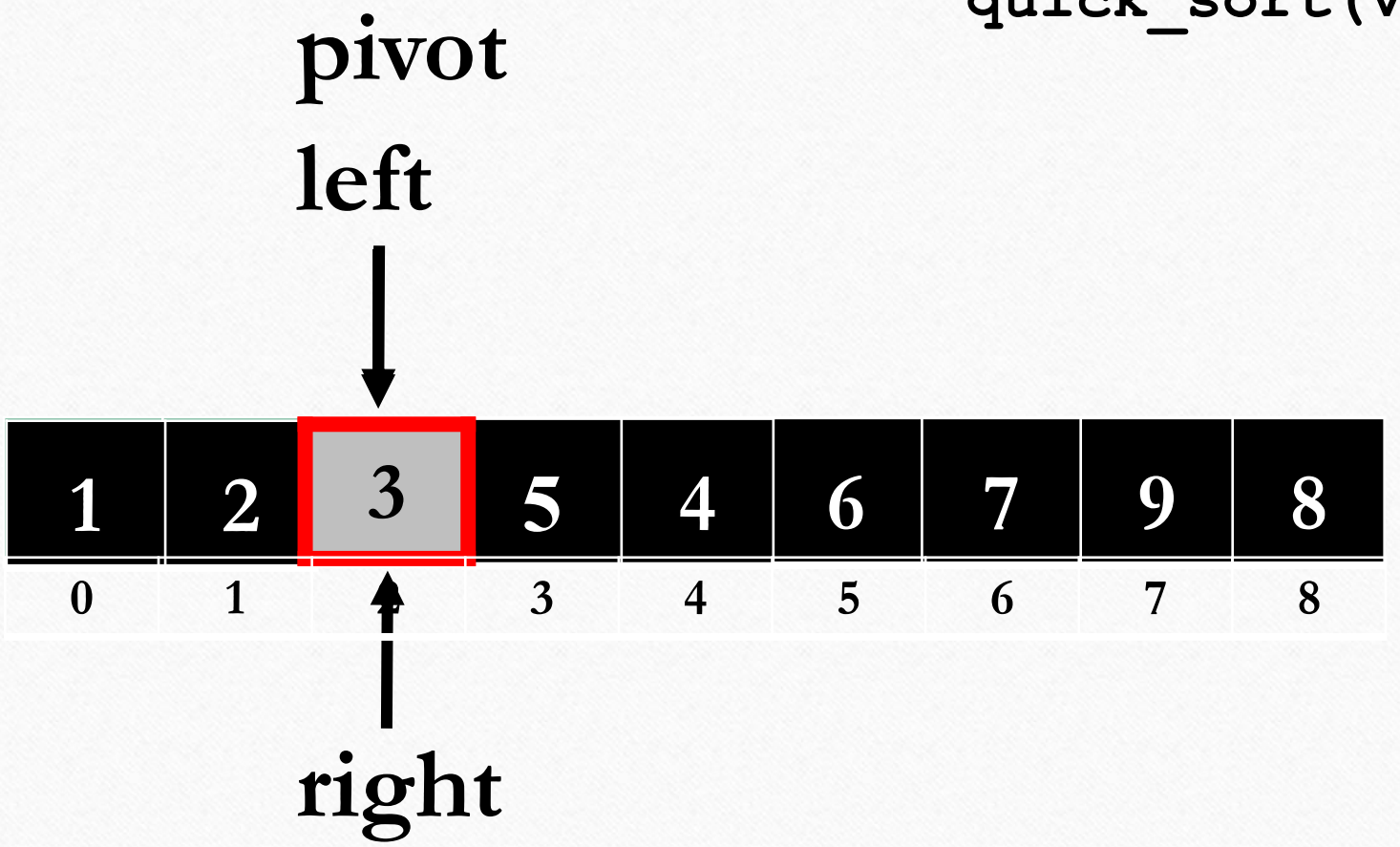


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 2, 3);
```

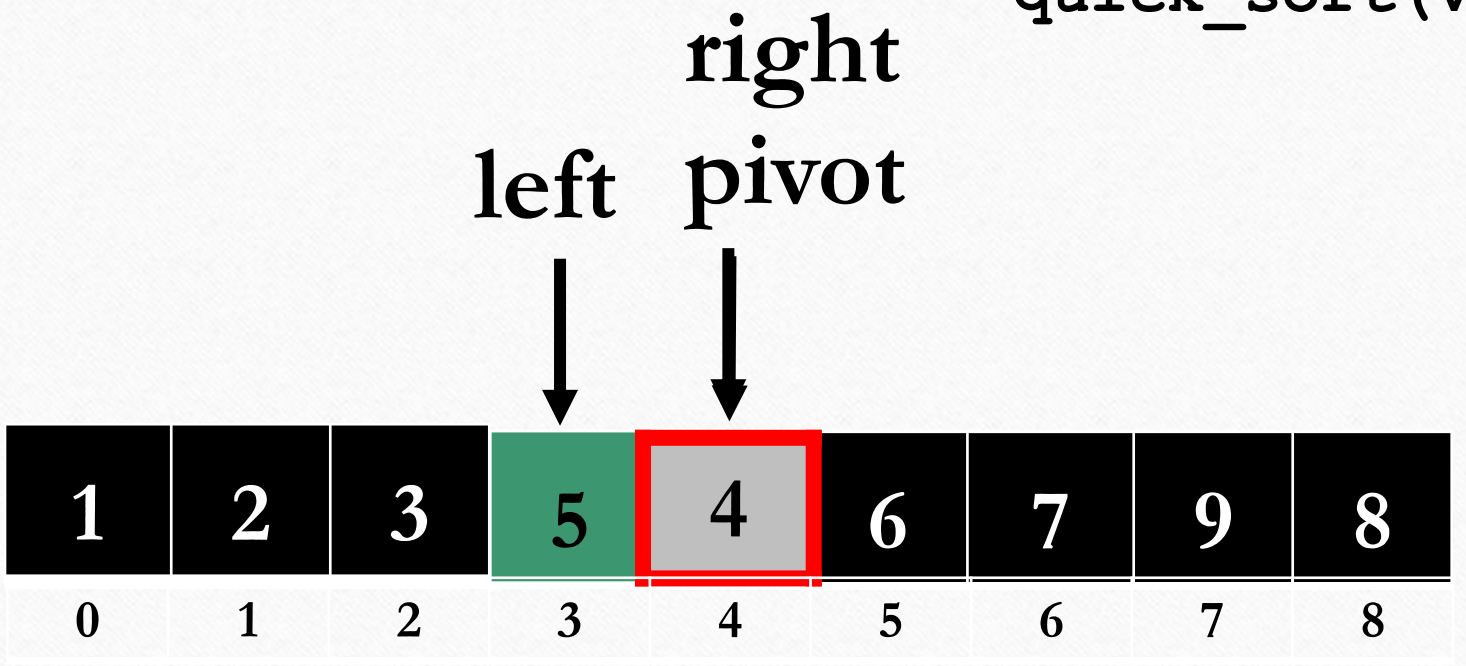


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 3, 5);
```

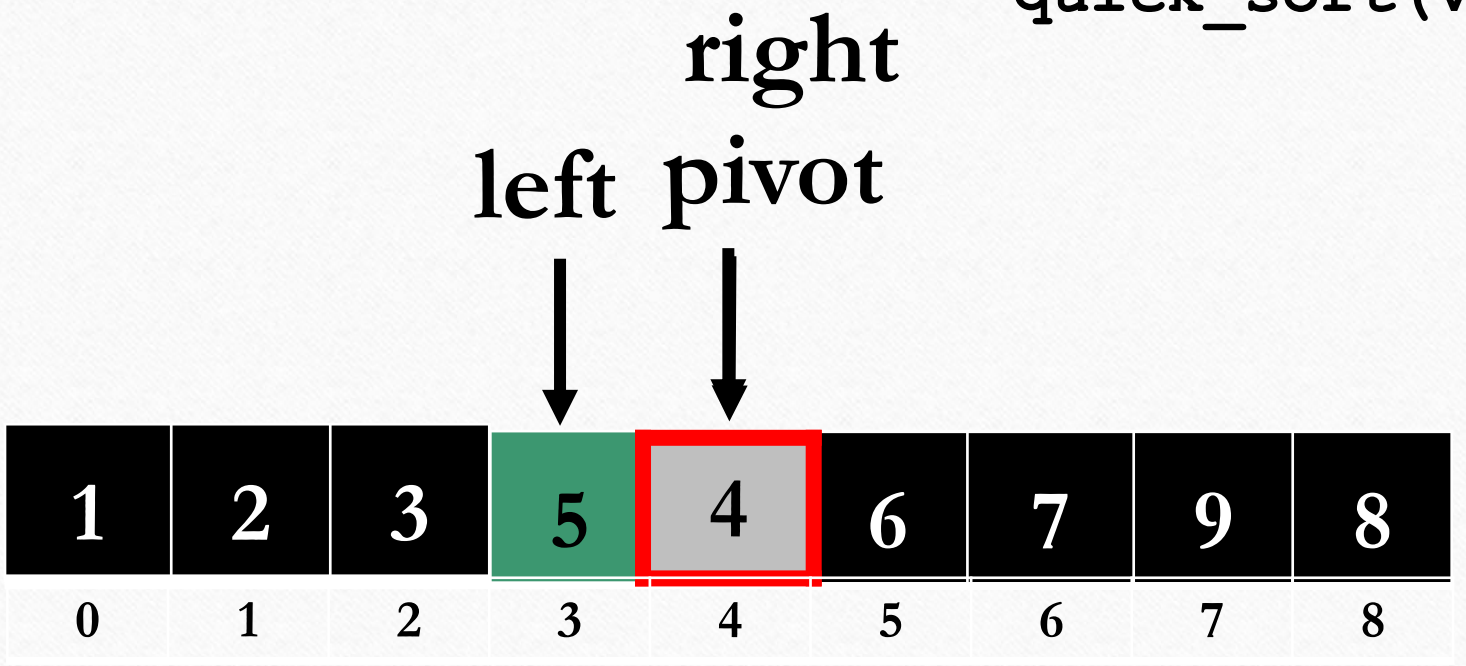


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 3, 5);
```

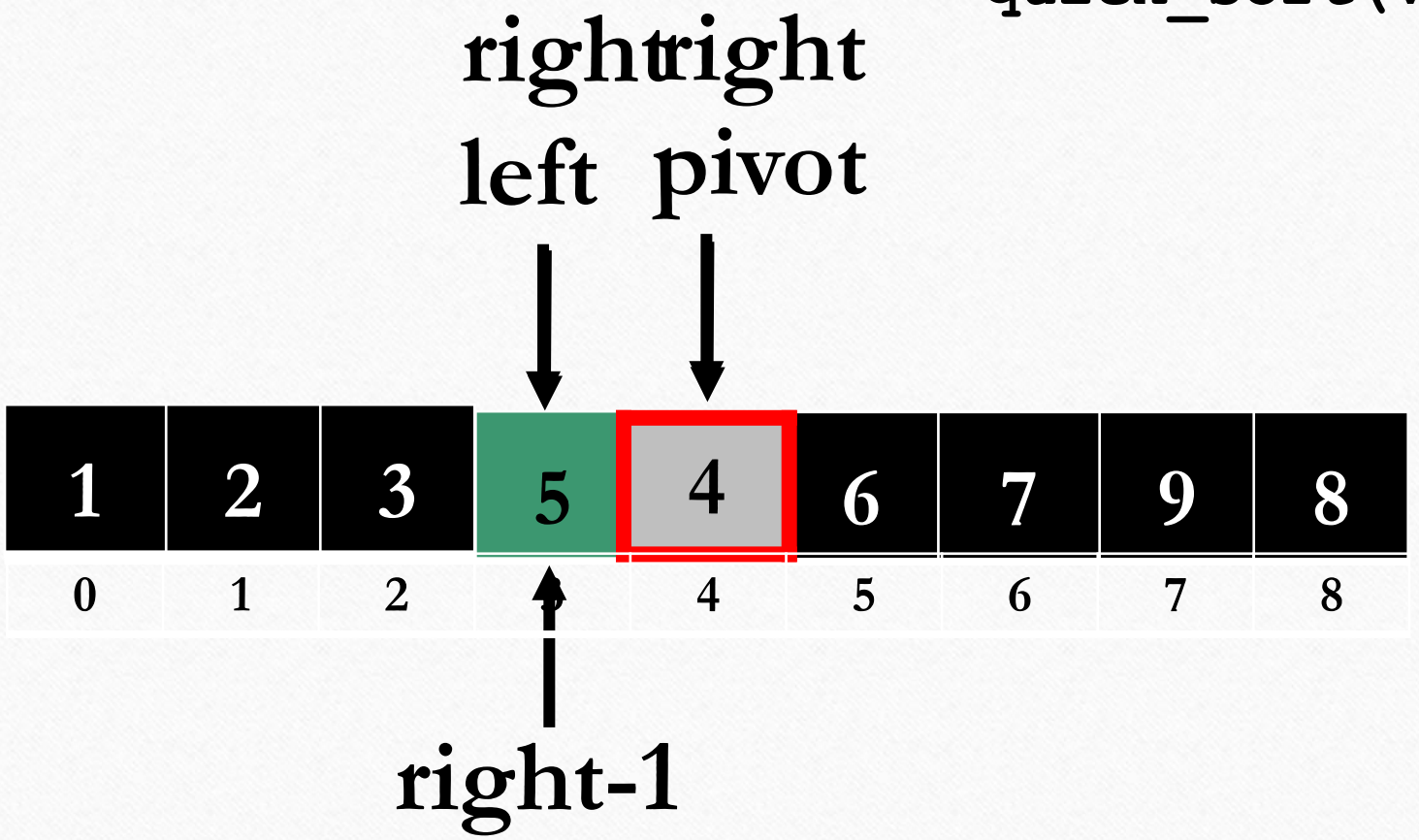


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

VII. Quicksort

```
quick_sort(Vec, 3, 5);
```

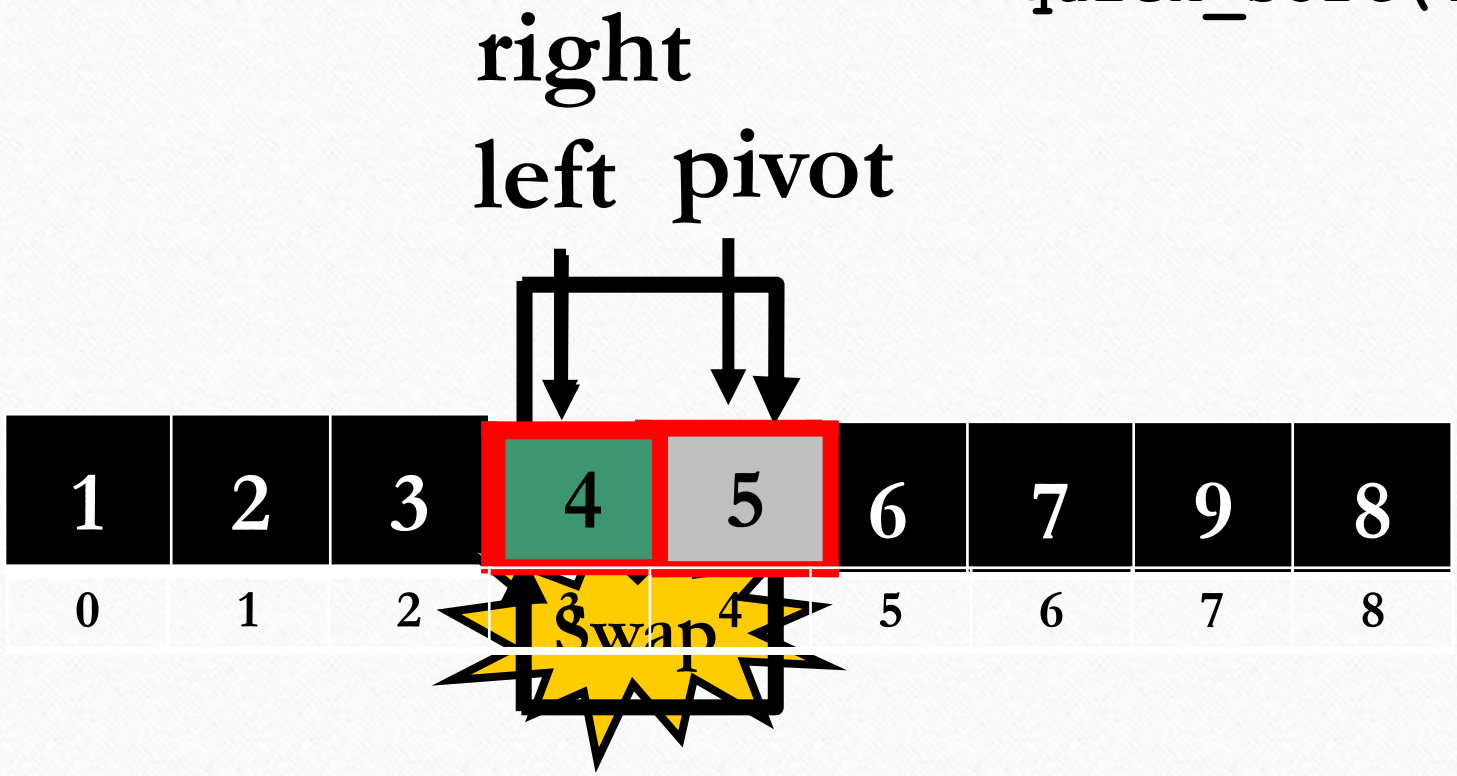


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

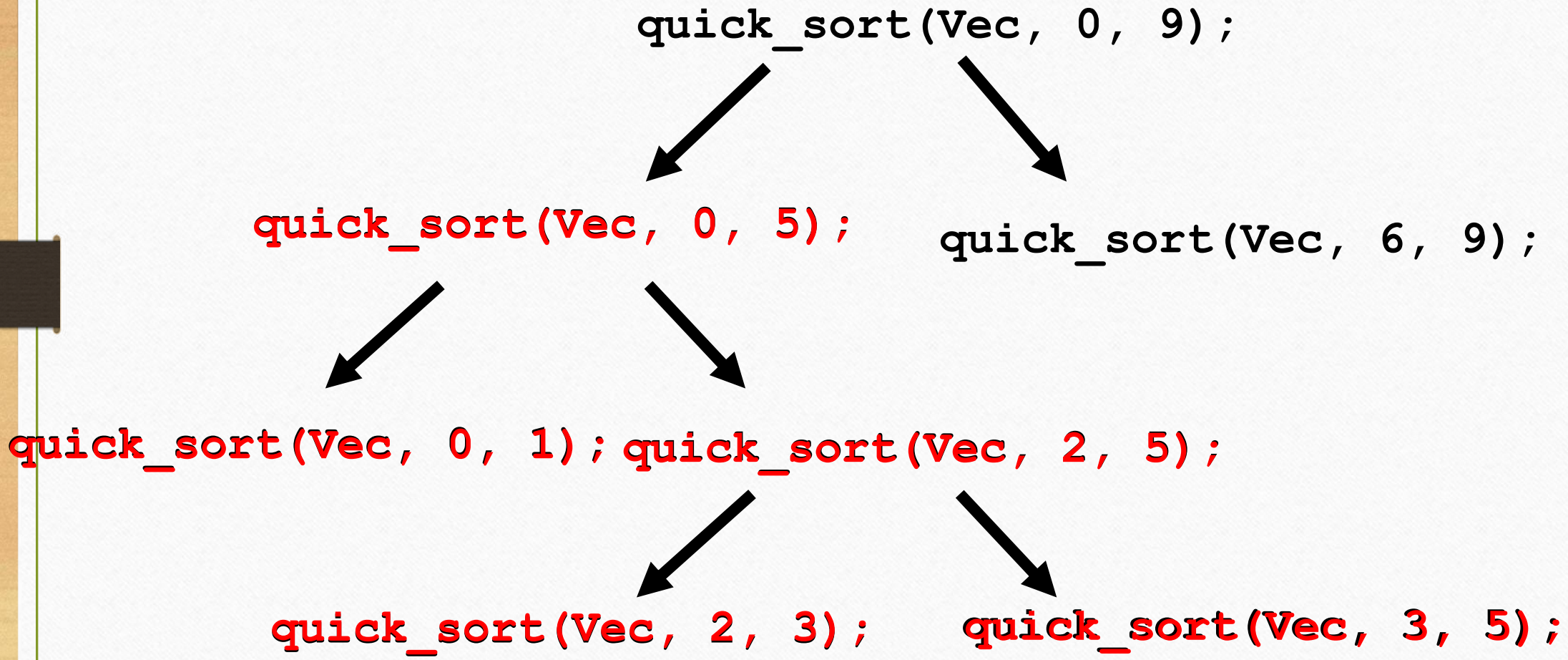
VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 3, 5);
```



VII. Quicksort

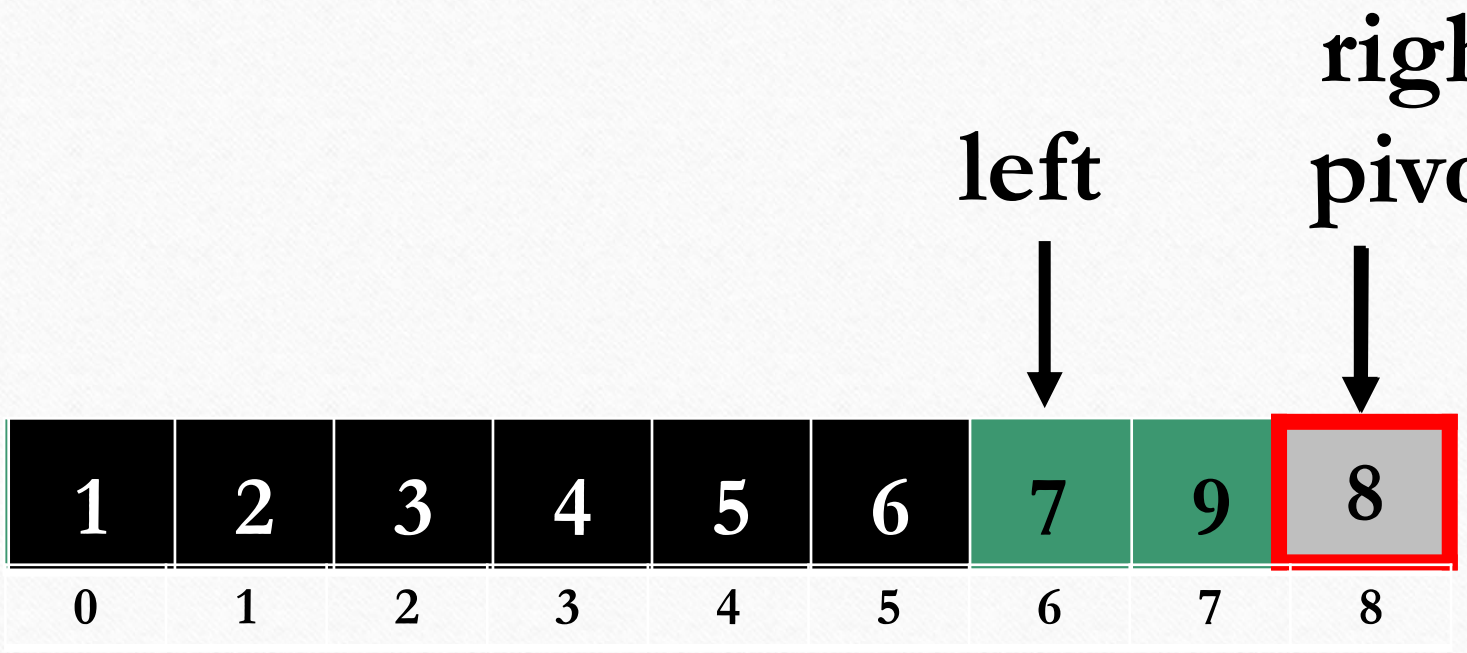


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 6, 9);
```

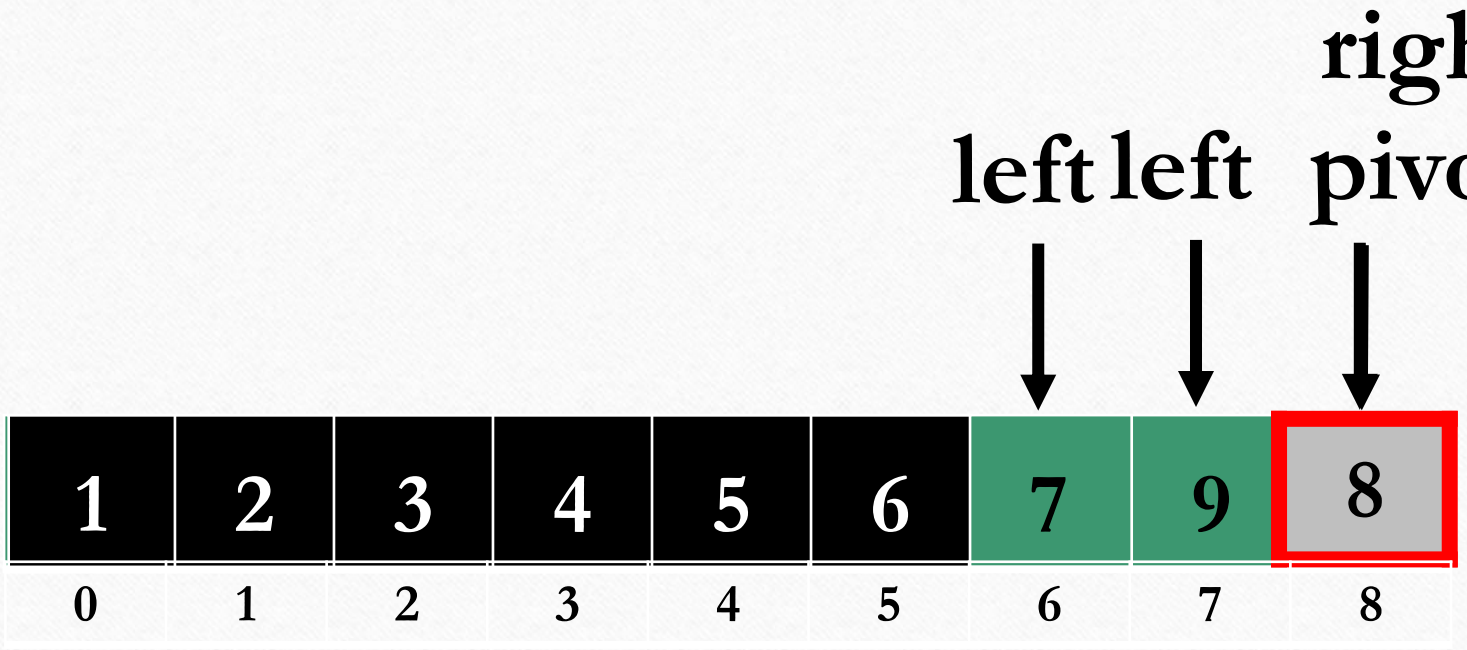


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 6, 9);
```



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

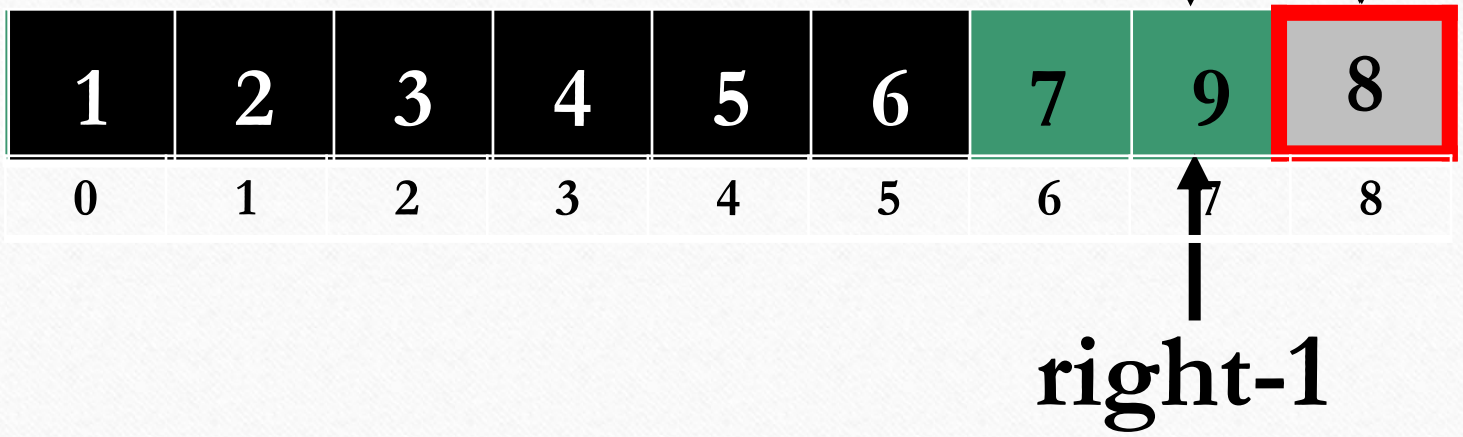
VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

quick_sort(Vec, 6, 9);

rightright

left pivot



```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

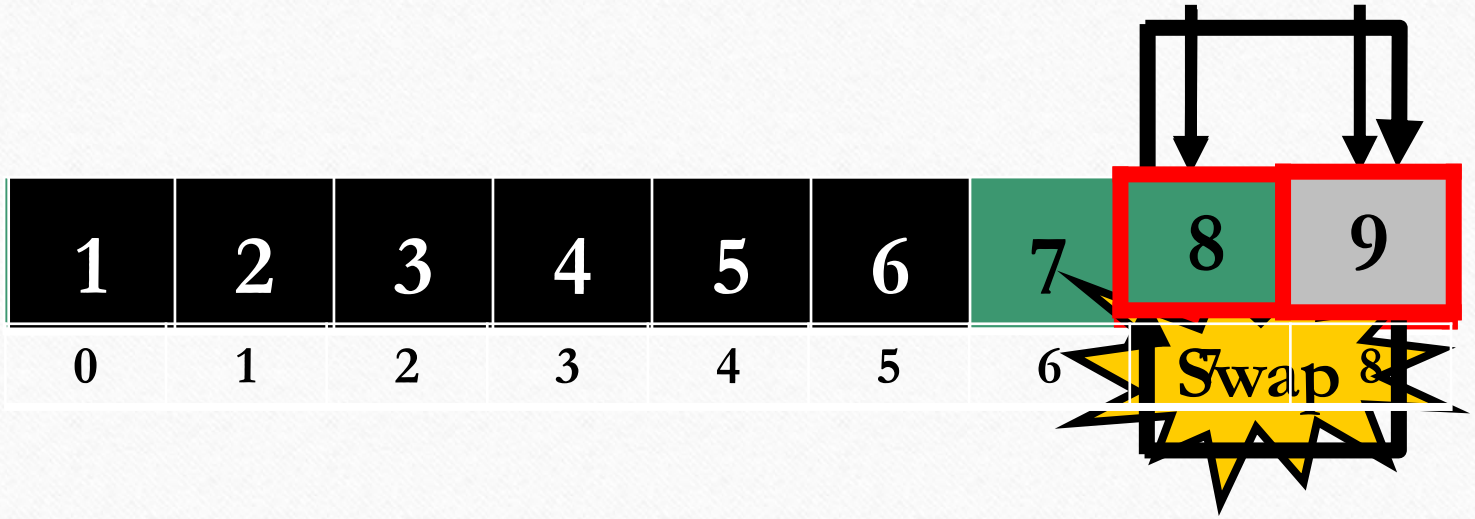
VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

quick_sort(Vec, 6, 9);

right

left pivot



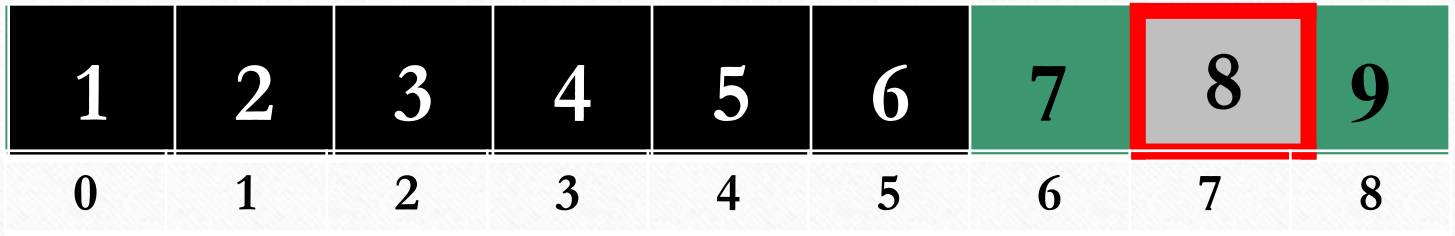
```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

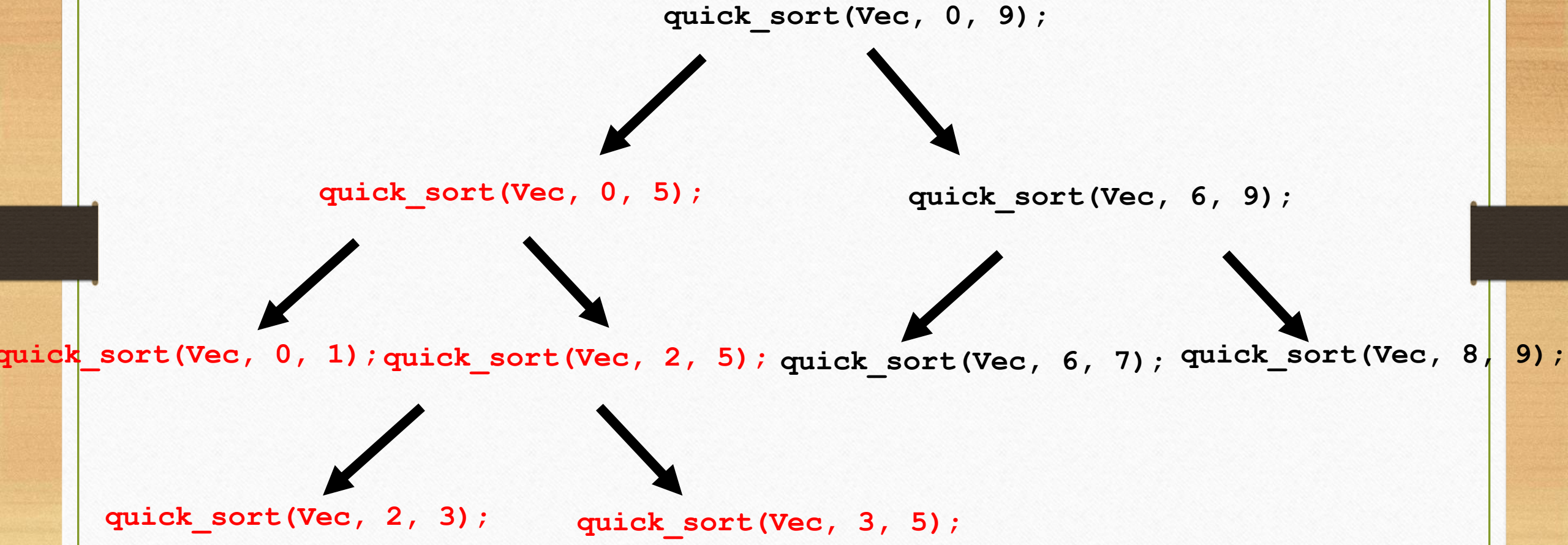
```
quick_sort(Vec, 6, 9);
```

left=pivot




```
quick_sort(Vec, 6, 7);
quick_sort(Vec, 8, 9);
```

VII. Quicksort

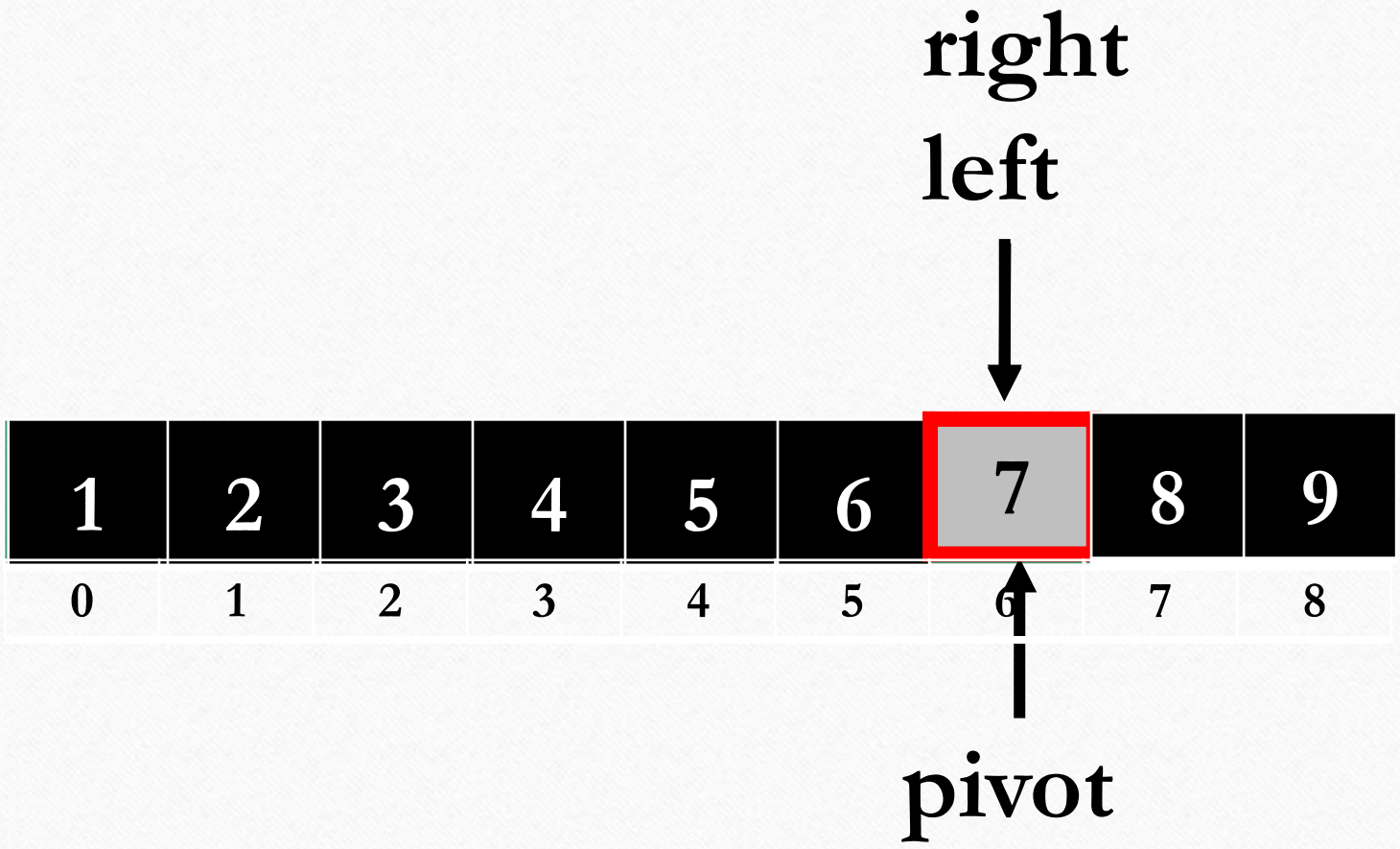


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

```
quick_sort(Vec, 6, 7);
```

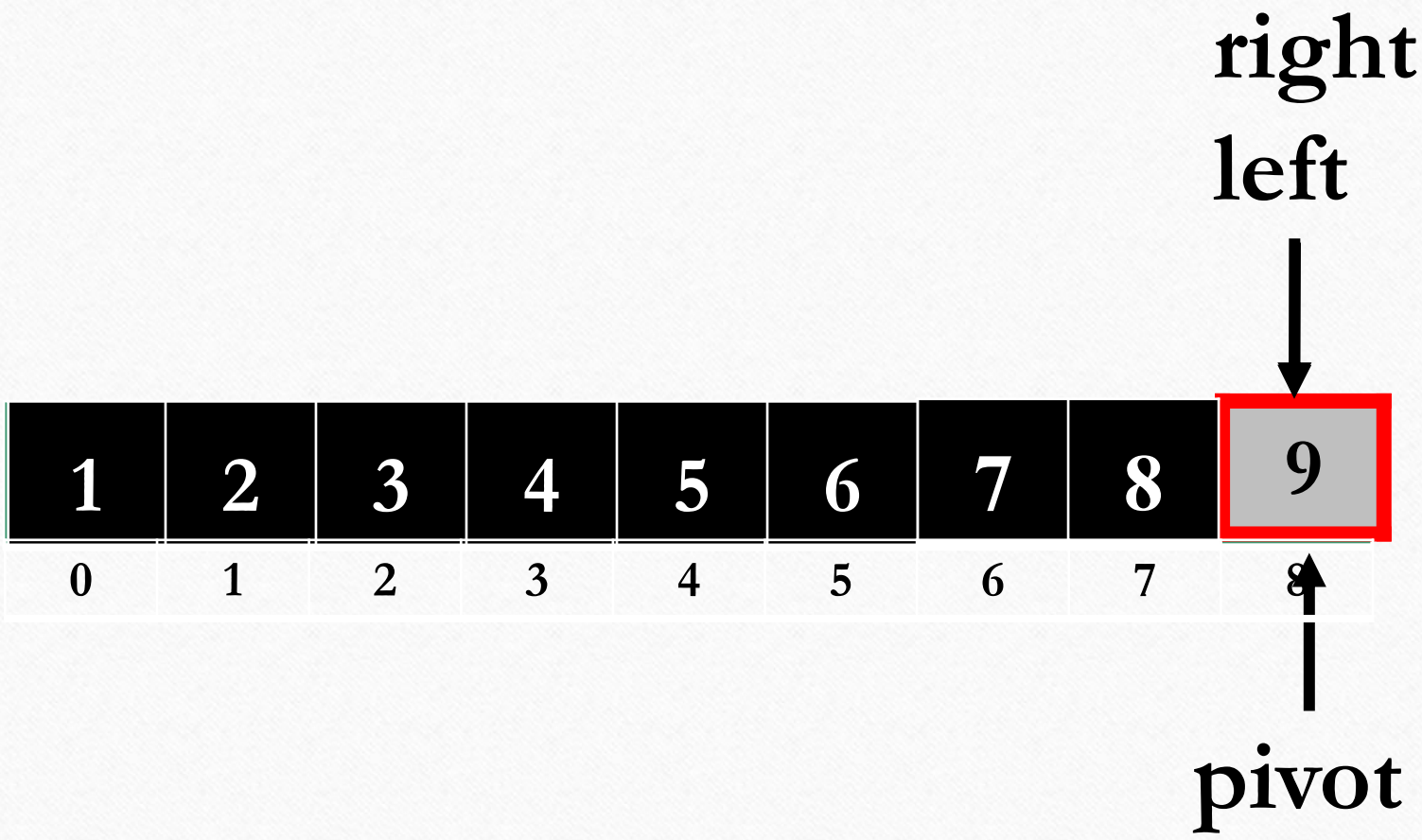


```
int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
       swap(Vec[left], Vec[right - 1]);
 }
swap(Vec[left], Vec[pivot]);
return left;
```

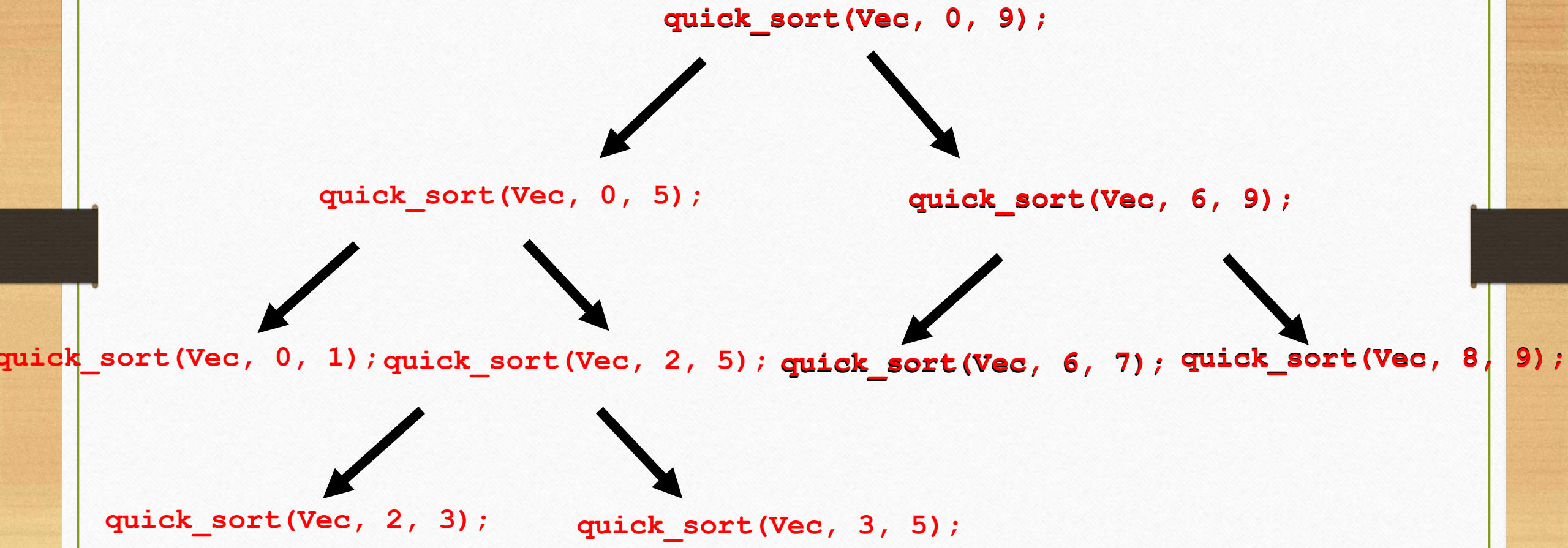
VII. Quicksort

```
if (left + 1 >= right) {
return;
}
int pivot = partition(Vec, left, right);
quick_sort(Vec, left, pivot);
quick_sort(Vec, pivot + 1, right);
```

quick_sort(Vec, 8, 9);



VII. Quicksort



VII. Quicksort

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

End of QuickSort

VII. Quicksort

Time Complexity of Quicksort

```
void quick_sort(int Vec[], int left, int right)
{
      if (left + 1 >= right) {
         return;
      } // if
      int pivot = partition(Vec, left, right);
      quick_sort(Vec, left, pivot);
      quick_sort(Vec, pivot + 1, right);
   } // quick_sort()
```

VII. Quicksort

Time Complexity of Quicksort

```
int partition(int Vec[], int left, int right) {
 int pivot = right--;
 while (true) {
    while (Vec[left] < Vec[pivot]) {left++;}
    while (left < right && Vec[right - 1] >= Vec[pivot]){
          right--;}
    if (left >= right) { break;}
    swap(Vec[left], Vec[right - 1]);
 }
 swap(Vec[left], Vec[pivot]);
 return left;
} // partition()
```

$\mathcal{O}(1)$

$\mathcal{O}(n)$

$\mathcal{O}(n)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(n)$

VII. Quicksort

Time Complexity of Quicksort

- Quicksort is a **recursive** algorithm.
- We call the ***partition()*** function, which ensures that the **pivot value is placed in its correct sorted position,** and that all elements to the left are smaller and all elements to the right are larger.
- This partitioning step takes $\boldsymbol{O(n)}$ time, since a single array pass is conducted when traversing using the left and right indices.
- Then, we **recursively call quicksort** on the elements **to the left and right of the pivot.**

VII. Quicksort

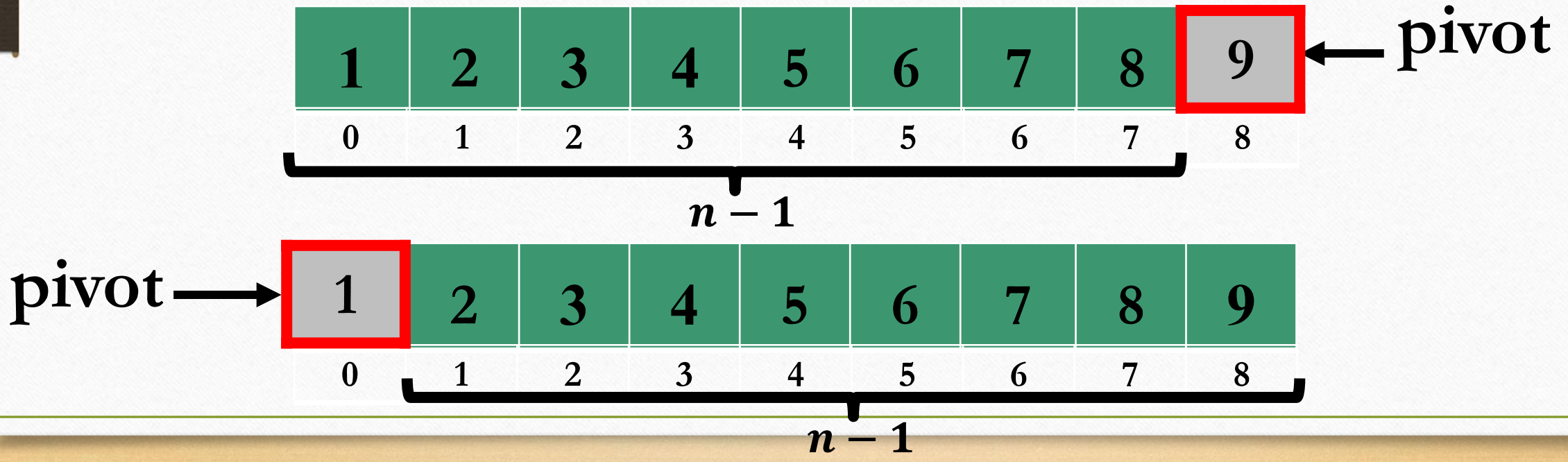
Time Complexity of Quicksort

- **What is the input size of these recursive calls?**
- **This actually depends on the value of the pivot!**
- In the worst case, either the **smallest or largest element is chosen as the pivot at every step**.
- When this happens, the pivot always leaves one side empty and the other side with all the remaining elements.

VII. Quicksort

Time Complexity of Quicksort

- For instance, if **1 (or 9)** were chosen as the pivot, every other element would be partitioned to the right (left) of the pivot:



VII. Quicksort

Time Complexity of Quicksort

- When this happens, one of the recursive calls would have an input size of $\boldsymbol{0}$, and the other would have an input size of $\boldsymbol{n-1}$.
- If either the smallest or largest element is chosen as the pivot every time, the recurrence relation for quicksort becomes

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ T(n-1) + n & \text{if } n > 1 \end{cases}$$

VII. Quicksort

Time Complexity of Quicksort

- Here, the $\boldsymbol{T(n-1)}$ comes from the recursive quicksort call with input size $\boldsymbol{n-1}$, and the $\boldsymbol{n}$ comes from the partitioning step.
- Using Iterative Substitution Method, we can see that the worst-case time complexity of quicksort is $\boldsymbol{O(n^2)}$.

VIII. Merge Sort

- The **basic idea of merge sorting** is to **divide the list** to be sorted into two halves (equal halves).
- The two halves are then recursively sorted and merged using a ***merge()*** function.

VIII. Merge Sort

- An implementation of merge sort is shown in the code below:

```
void merge_sort(int Vec[], int left, int right) {
   if (right - left < 2) {
      return;
   } // if
   int mid = left + (right - left) / 2;
   merge_sort(Vec, left, mid);
   merge_sort(Vec, mid, right);
   merge(Vec, left, mid, right);
} // merge_sort()
```

VIII. Merge Sort

- On line ***merge(Vec, left, mid, right);*** , we merge two sorted arrays together. This merging step can be done in $\boldsymbol{O(n)}$ time using the following process:

1) Keep track of two indices, $\boldsymbol{i}$ and $\boldsymbol{j}$, that refer to the first element of each of the sorted halves.
2) Initialize a separate vector of size "$\boldsymbol{size = right - left}$; " to store the merged ***output***.

VIII. Merge Sort

3) **If Vec[i] <= Vec[j]** copy **Vec[i]** to the output vector and increment ***i***.

Otherwise, copy **Vec[j]** to the output vector and increment ***j***.

4) Repeat until the entire output vector is filled. If one sorted half reaches the end before the other, simply append all of the remaining elements in the unfinished half into the output vector.

VIII. Merge Sort

```
void merge(int Vec[], int left, int mid, int right) {
    int size = right - left;
    int output[size]; //init output vector
    for (int i = left, j = mid, k = 0; k < size; k++) {
        if (i == mid) {output[k] = Vec[j++];}
        else
            if (j == right) {output[k] = Vec[i++];}
            else {
                if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
                else {output[k] = Vec[j++];}
            }
    } // for
    for (int i = left; i < right; i++){
        Vec[i] = output[i-left];
    }
} // merge()
```

if first half is complete

if second half is complete

General Case

copies data sorted in output[] to Vec[].

VIII. Merge Sort

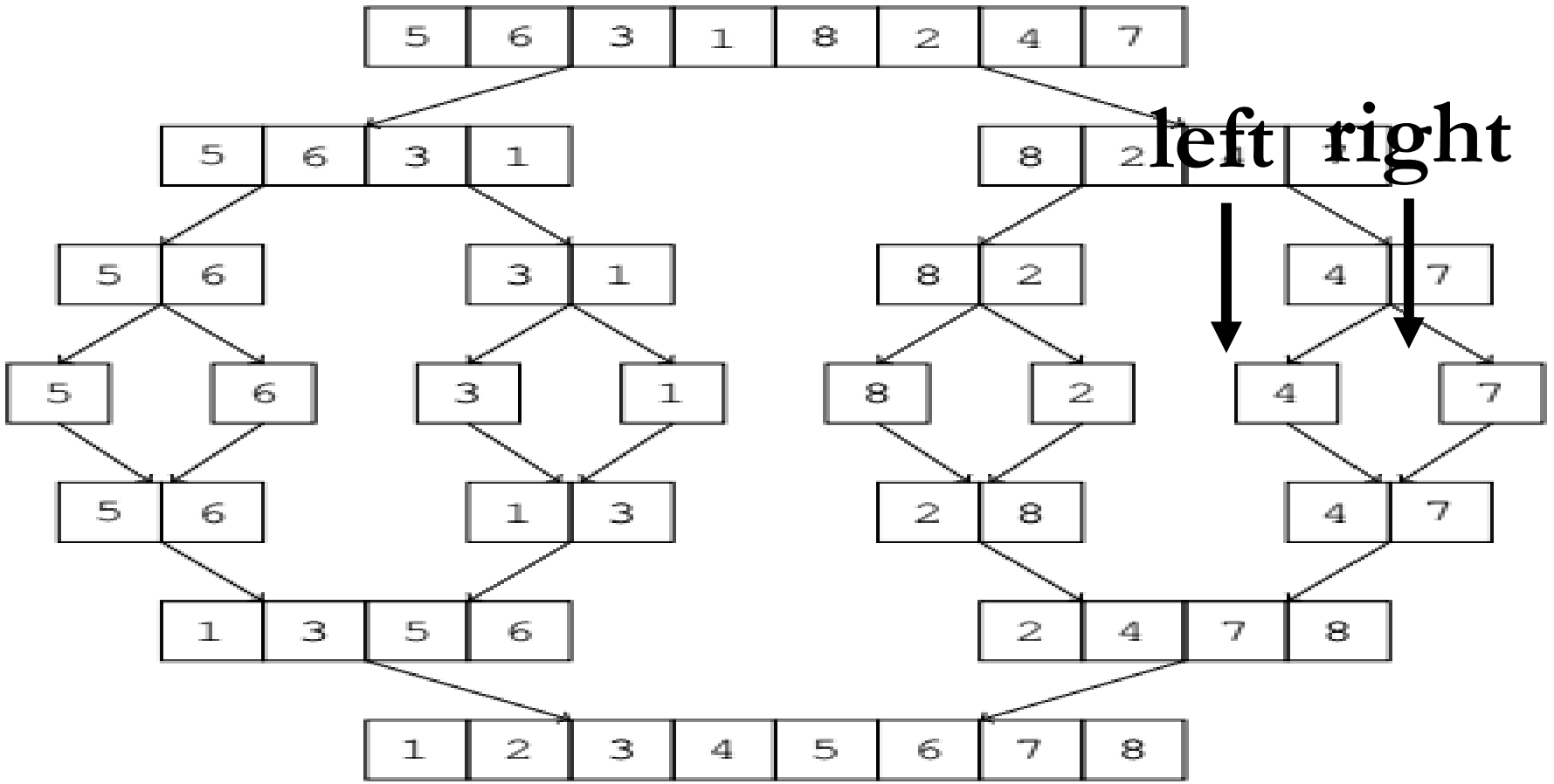
Example:

Let's look at what this code does using an example. Consider the following unsorted array:

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

The following diagram illustrates the entire merge sort process on our initial array.

VIII. Merge Sort



VIII. Merge Sort

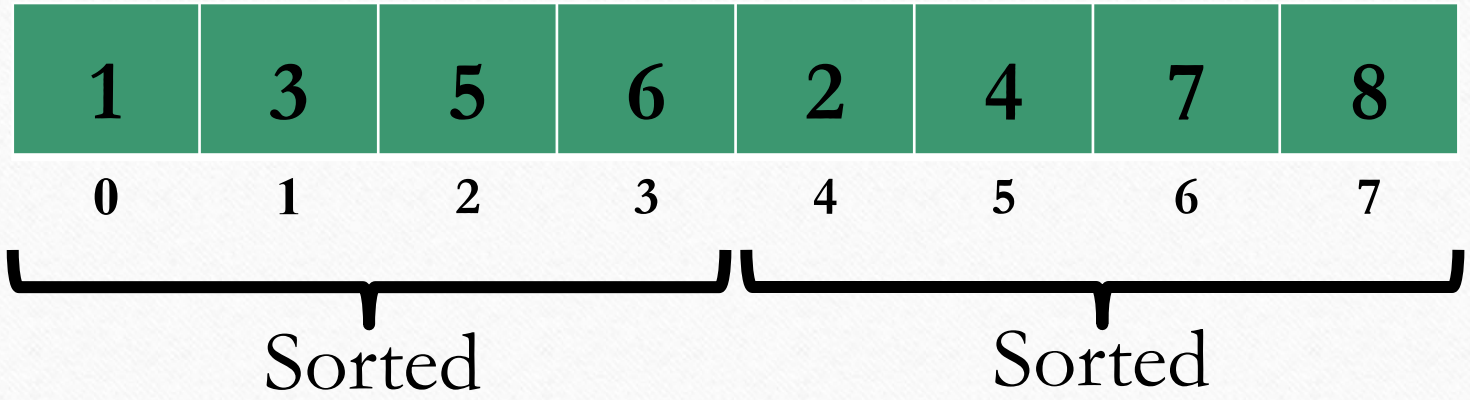
- Consider the ***merge()*** function on the following unsorted array:

$$[5, 6, 3, 1, 8, 2, 4, 7].$$

- During the merge sort process, we first recursively merge sort the two halves of the array, $[5, 6, 3, 1]$ and $[8, 2, 4, 7]$.
- This sorts the two halves of the array.

VIII. Merge Sort

- After this step, the contents of the array are as follows:



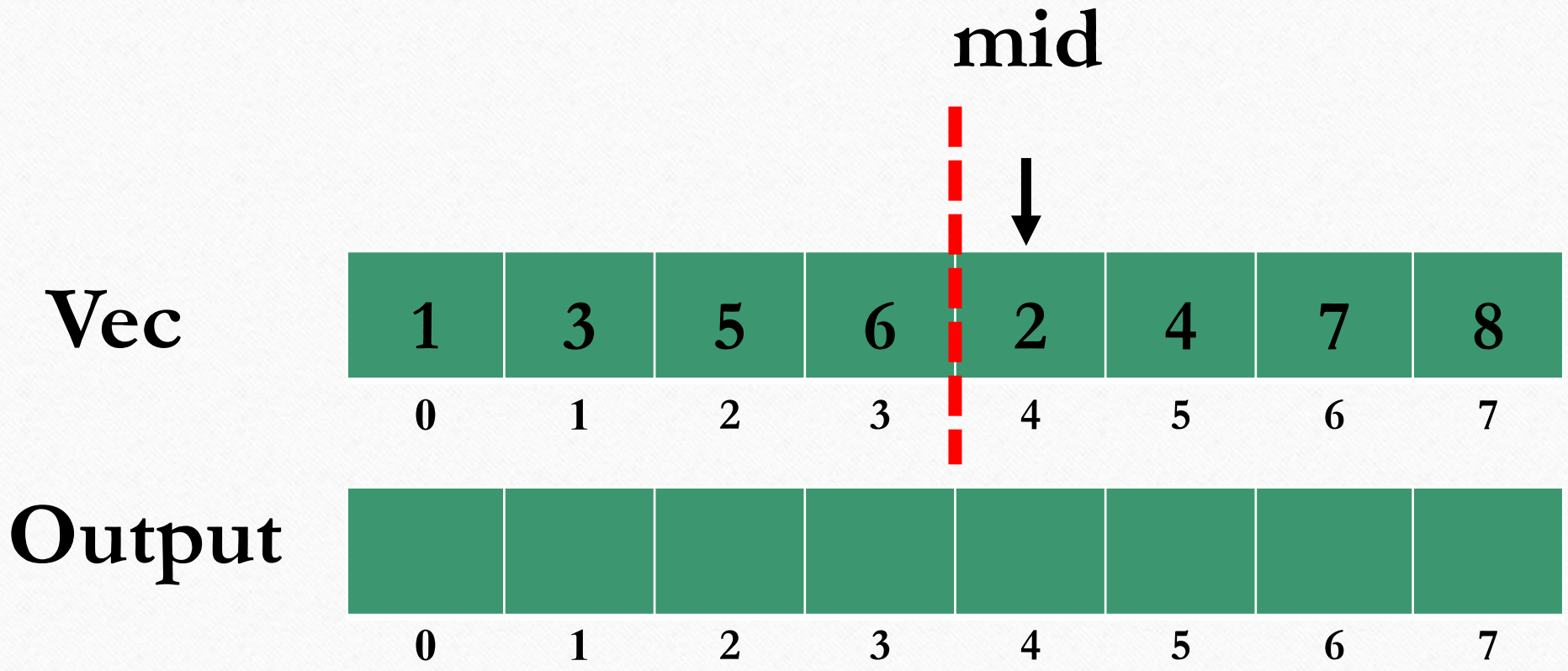
VIII. Merge Sort

- To merge the two sorted halves, we will first initialize an empty vector to store the sorted output and initialize indices ***i*** and ***j*** to refer to the first elements of the two halves.
- The variable ***k*** represents the next open slot in the output vector.

```
merge(Vec, 0, 4, 8)
```

VIII. Merge Sort

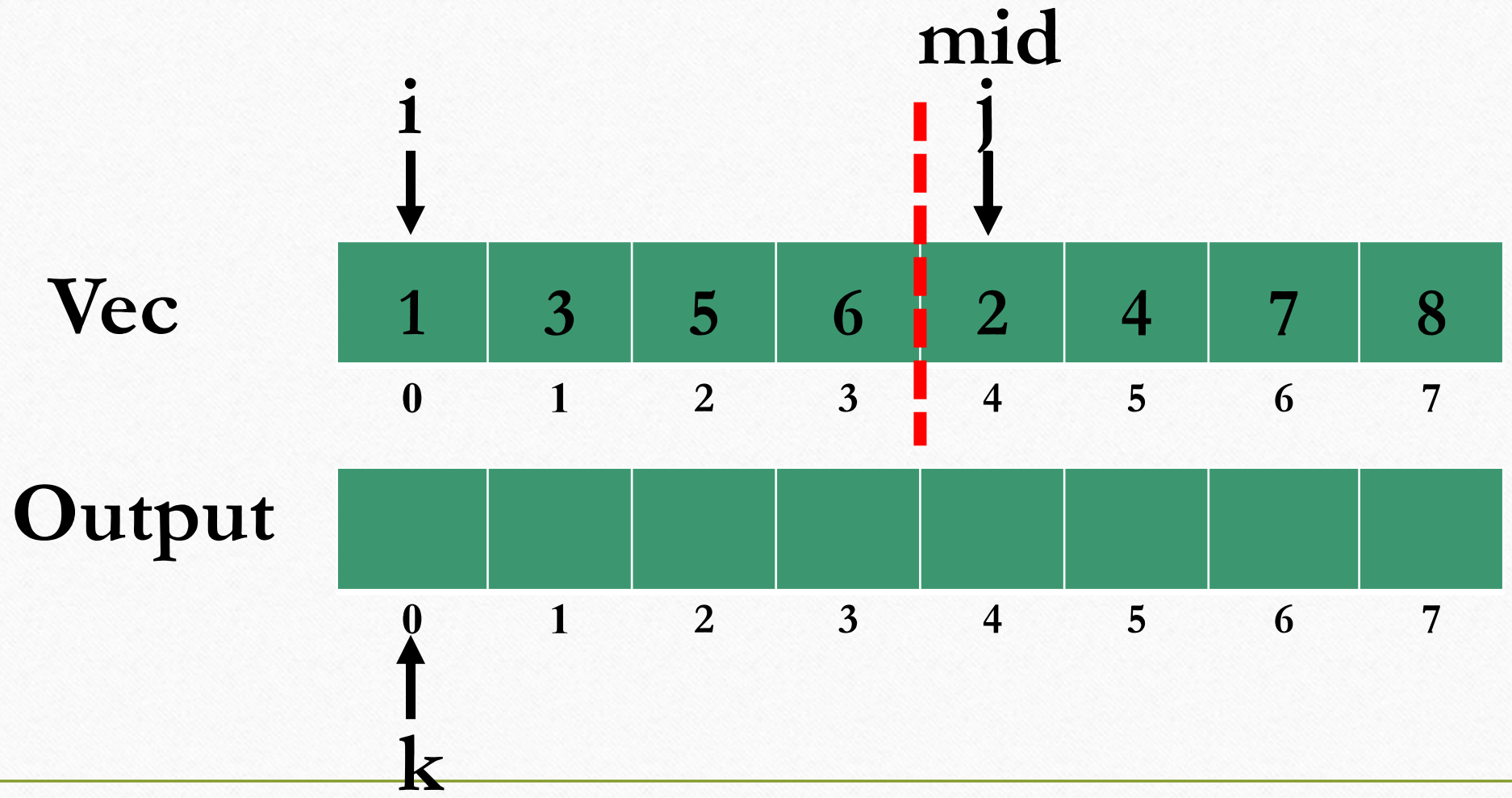
```
int size = right - left;
int output[size];
```



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left, j = mid, k = 0; k < size; k++)
```

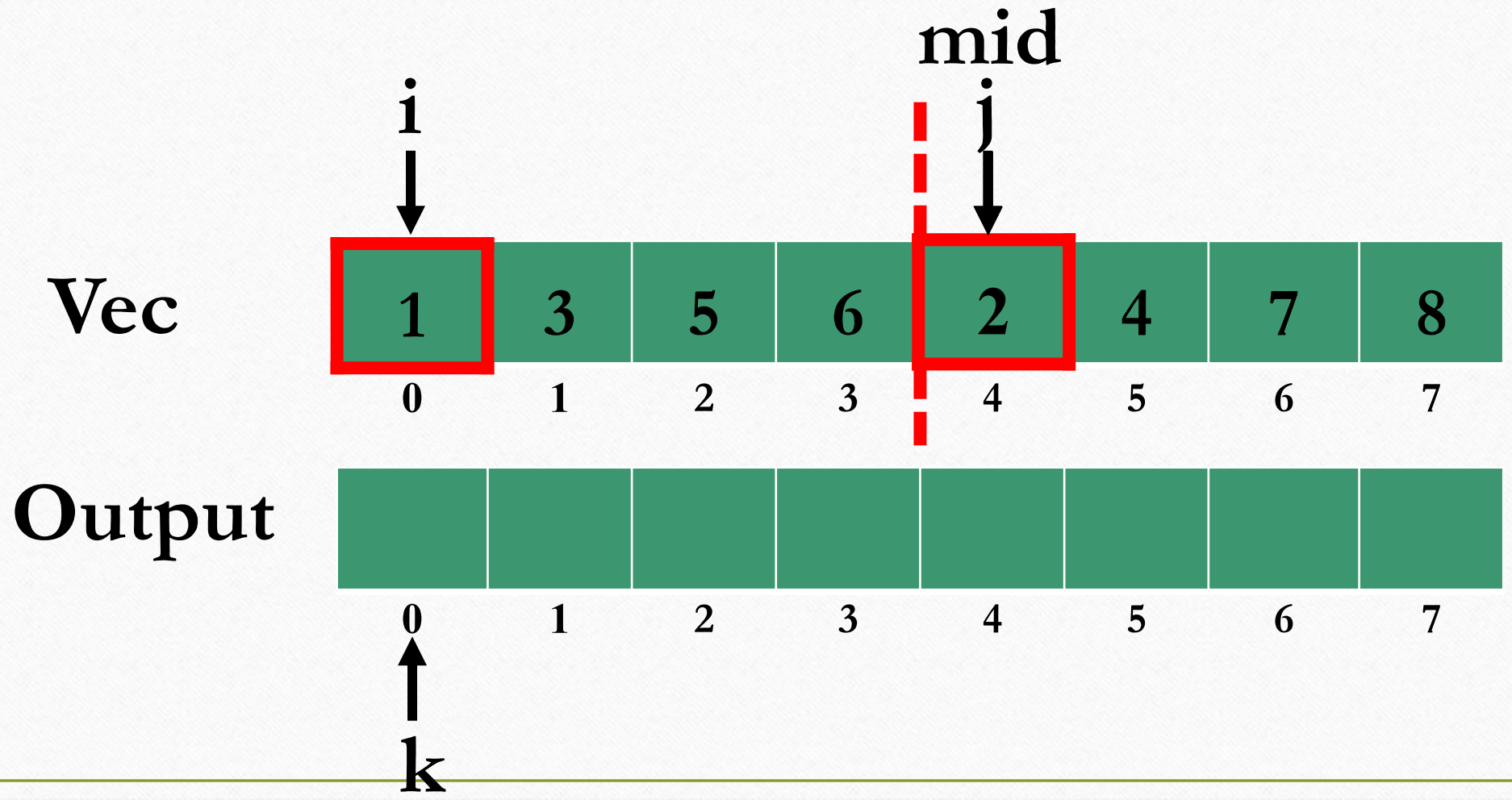
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
else {output[k] = Vec[j++];}
```

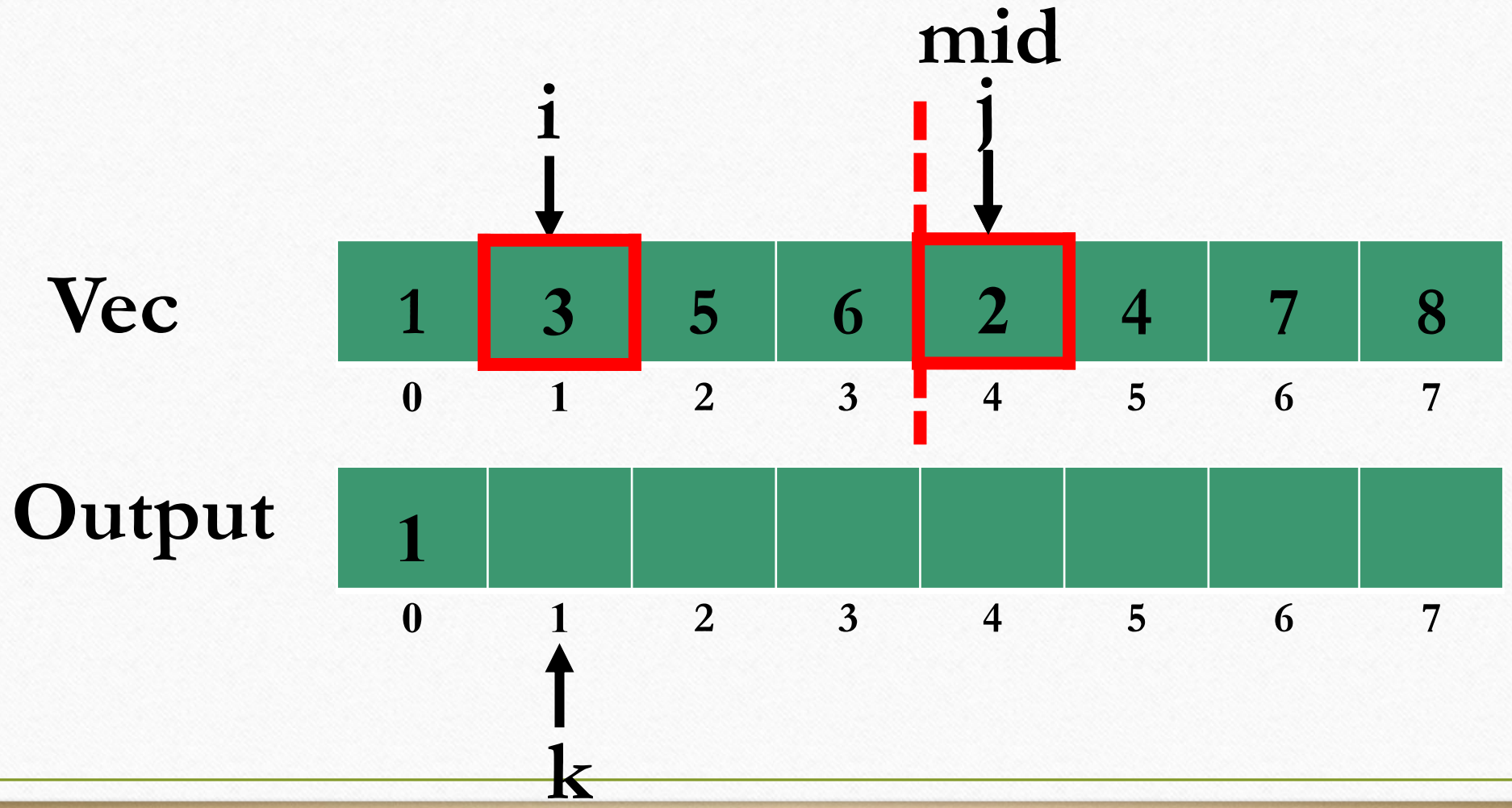
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
else {output[k] = Vec[j++];}
```

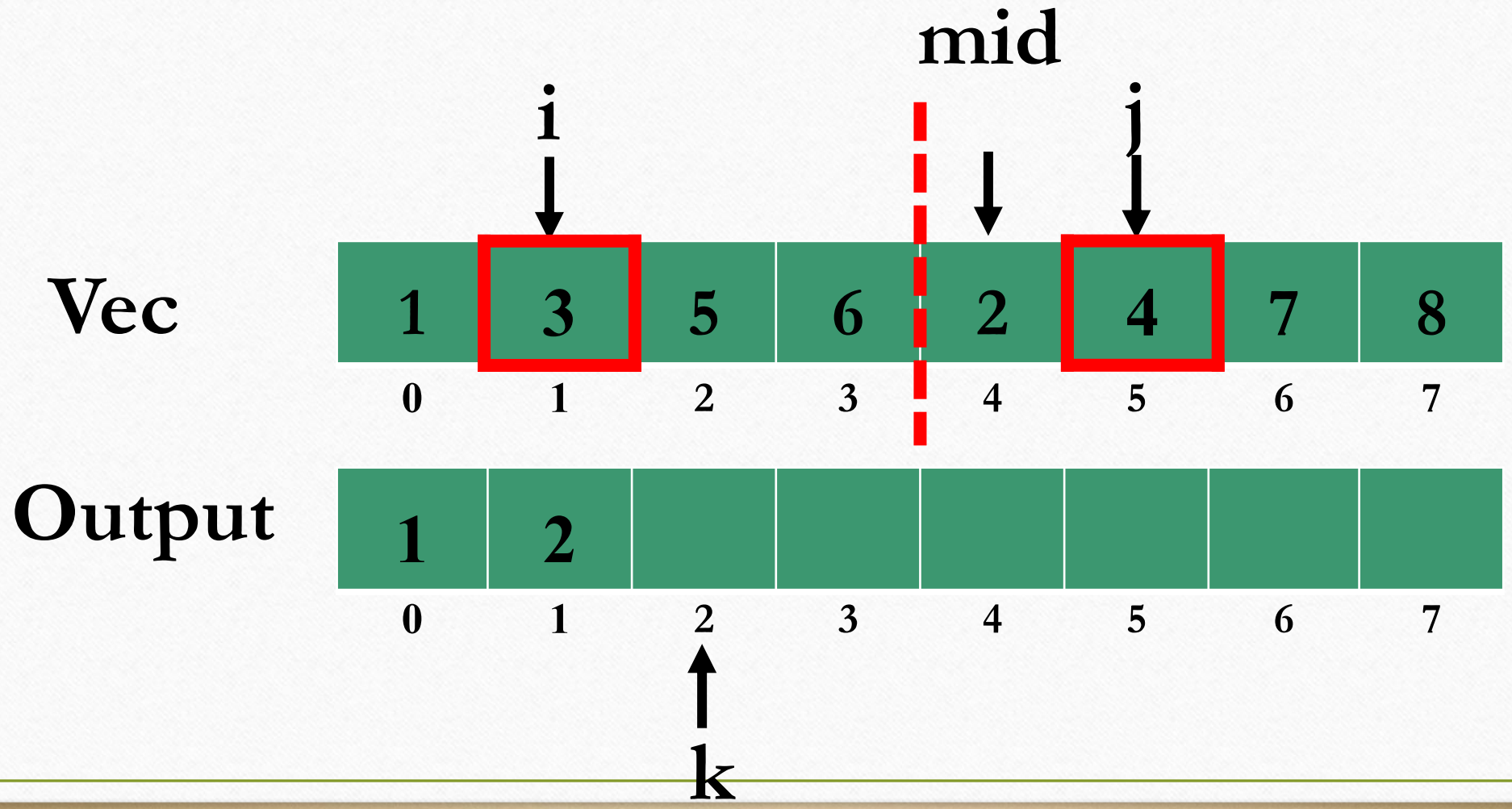
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
else {output[k] = Vec[j++];}
```

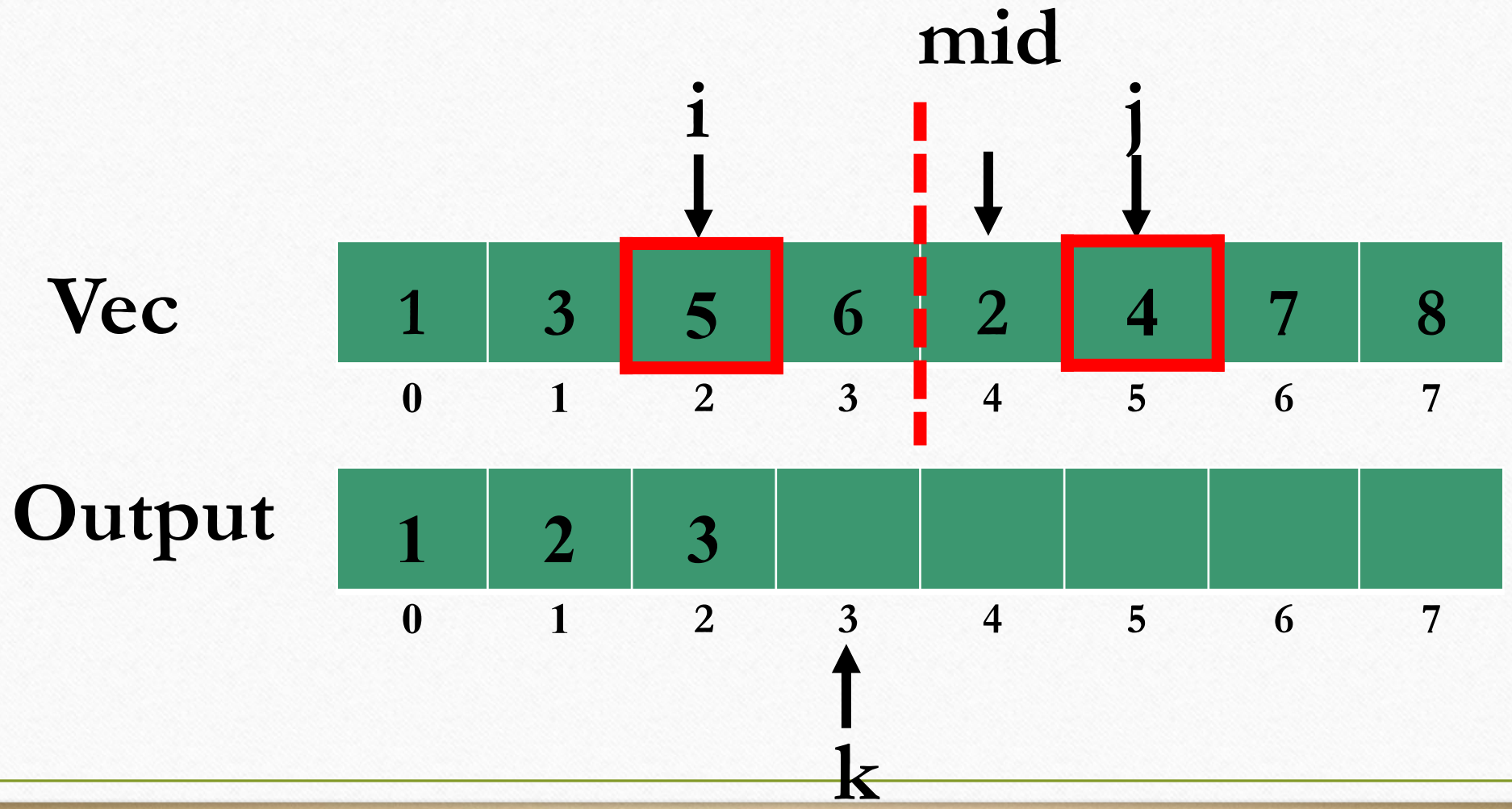
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
else {output[k] = Vec[j++];}
```

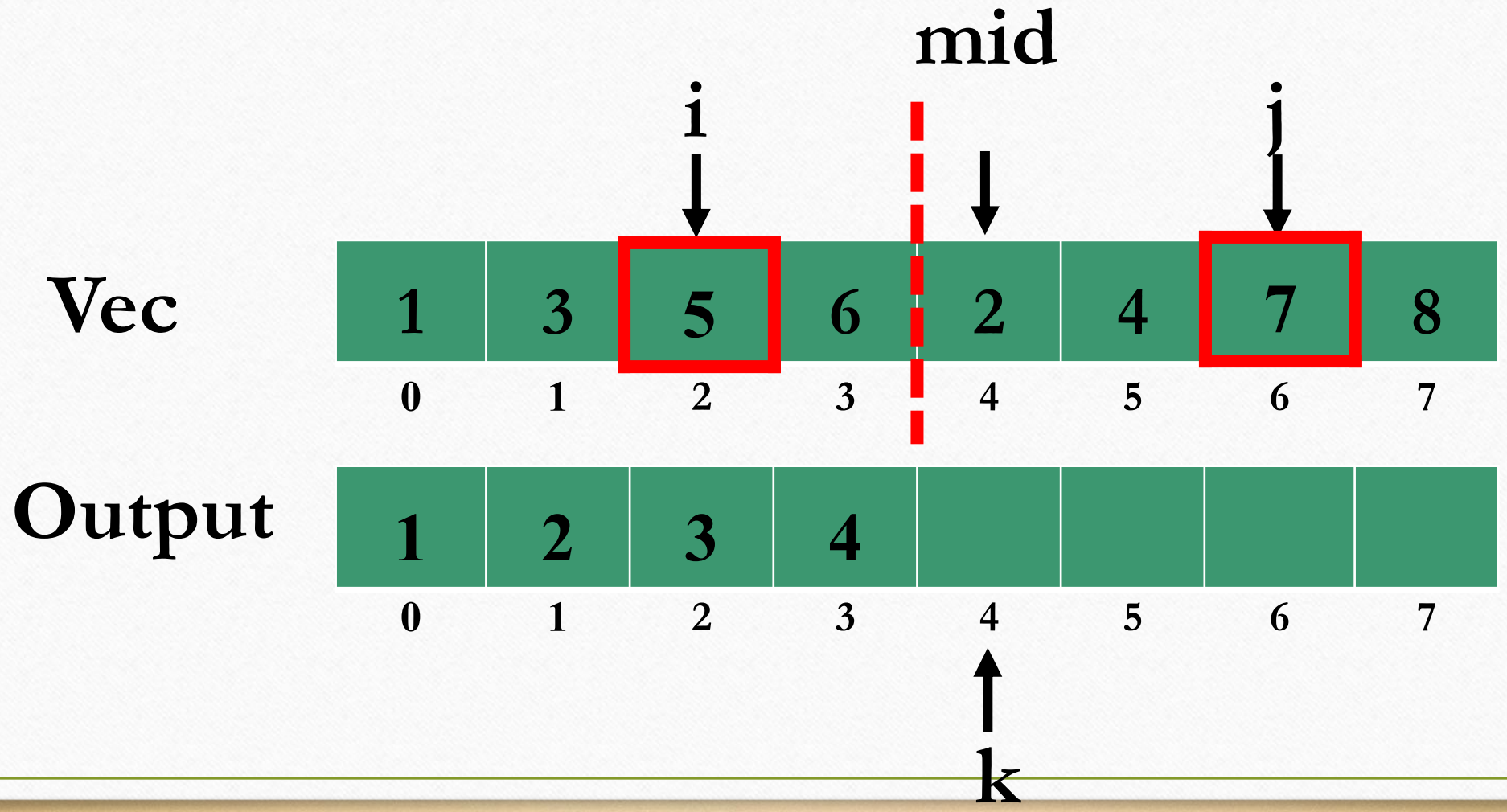
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
else {output[k] = Vec[j++];}
```

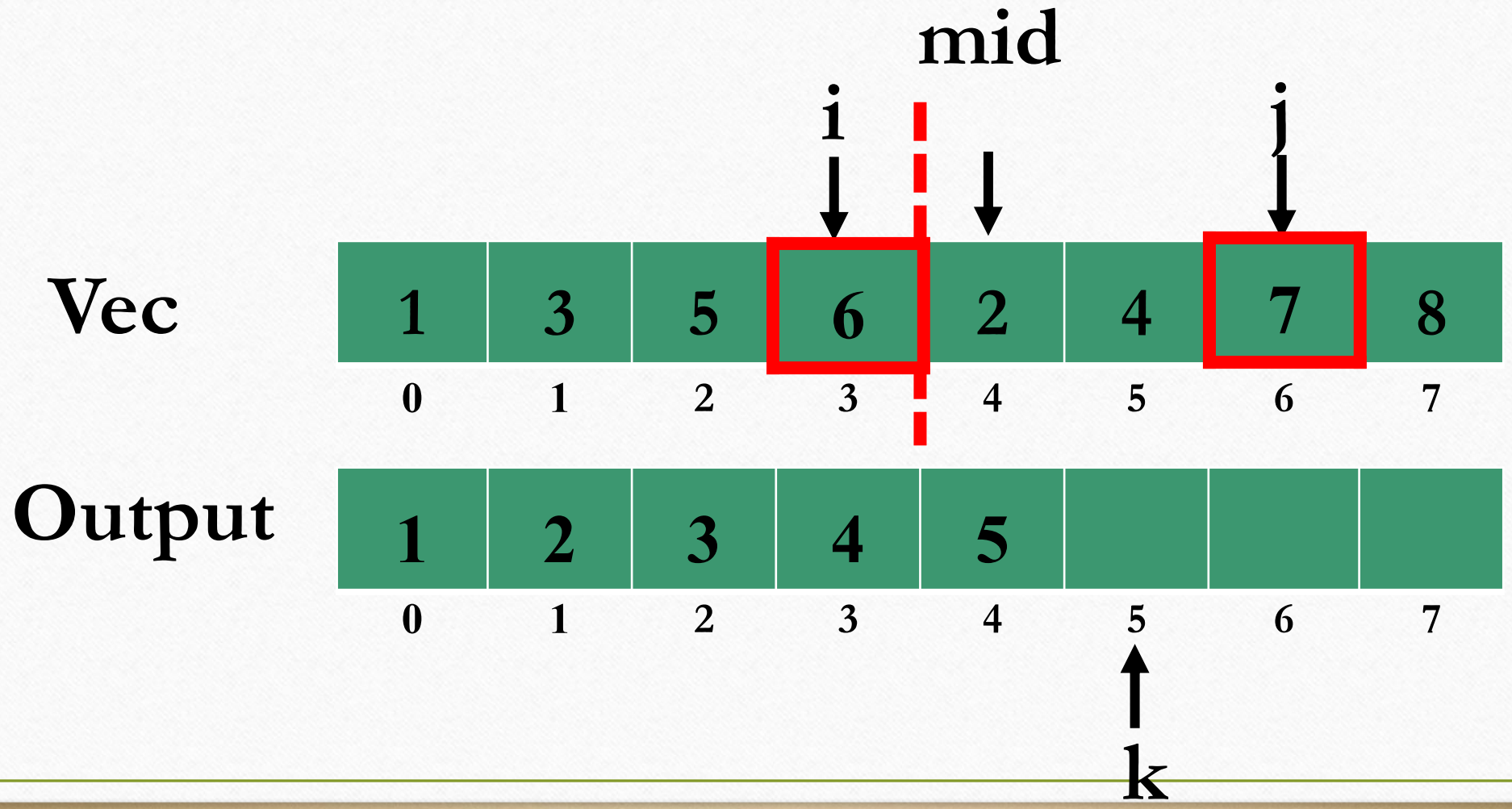
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
else {output[k] = Vec[j++];}
```

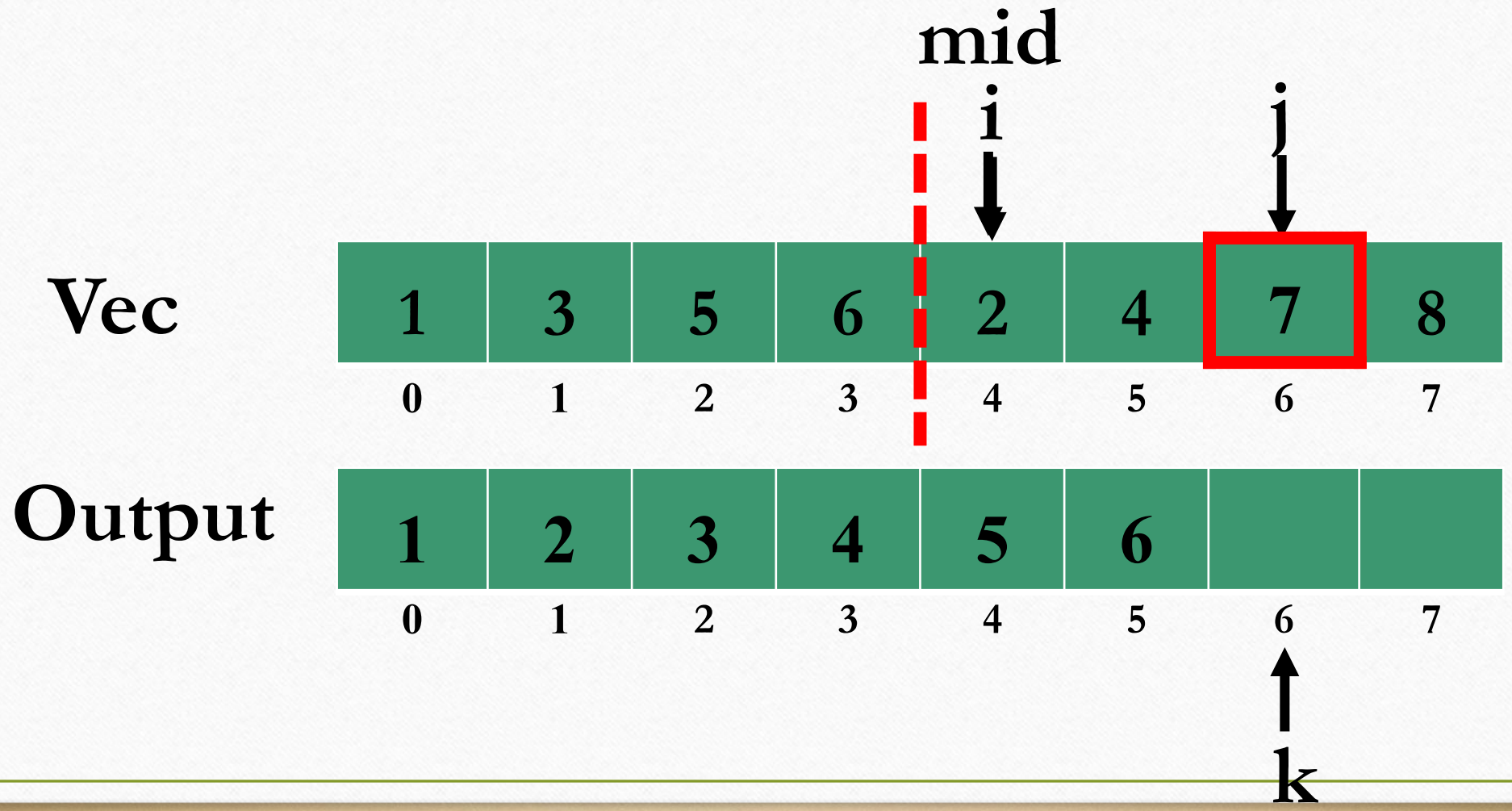
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (i == mid) {output[k] = Vec[j++];}
```

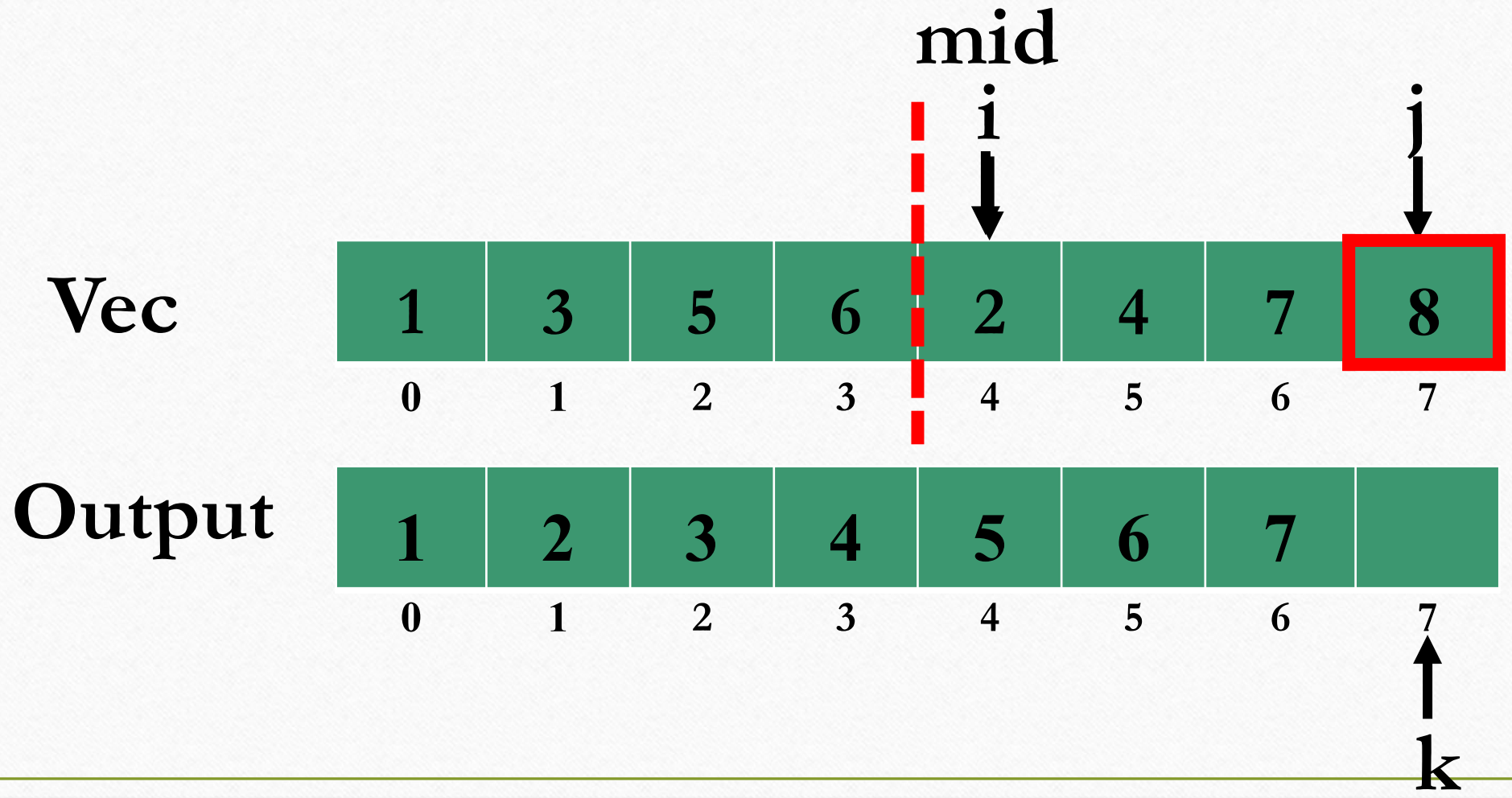
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (i == mid) {output[k] = Vec[j++];}
```

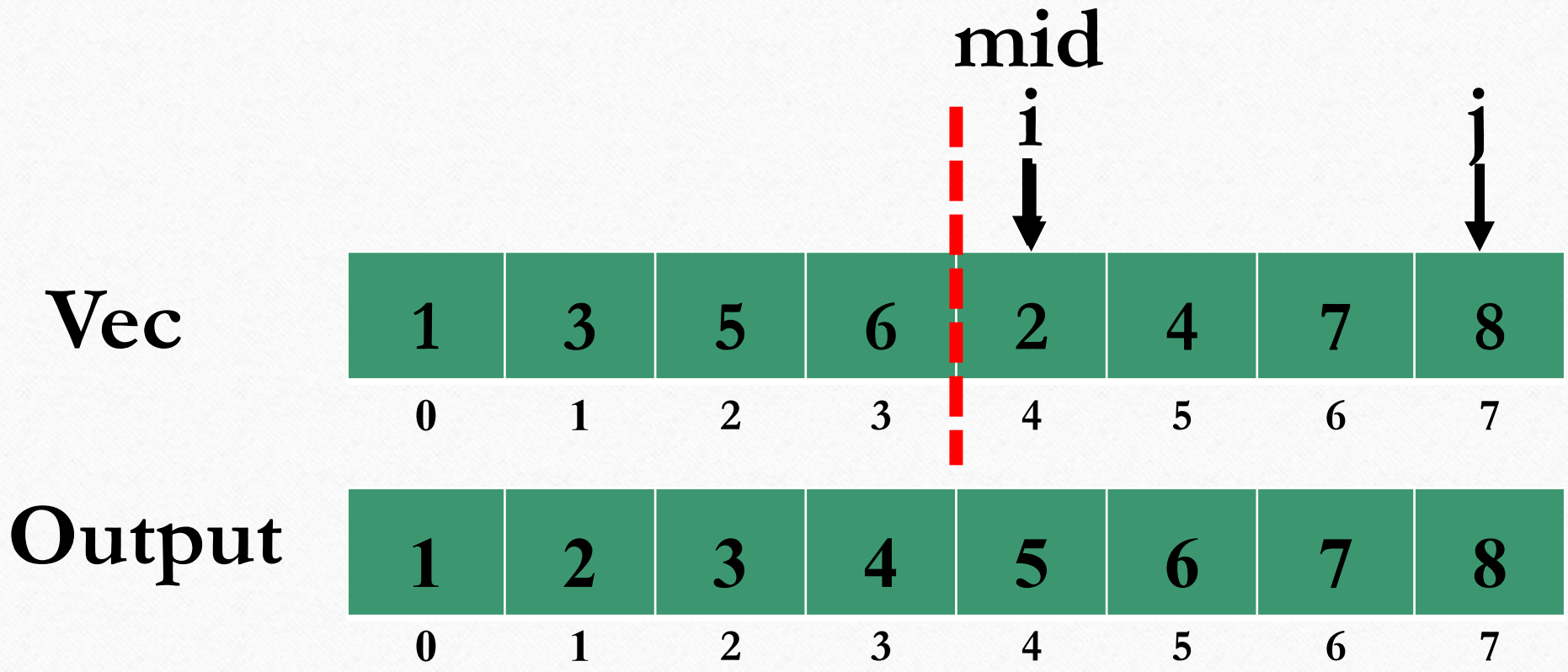
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
if (i == mid) {output[k] = Vec[j++];}
```

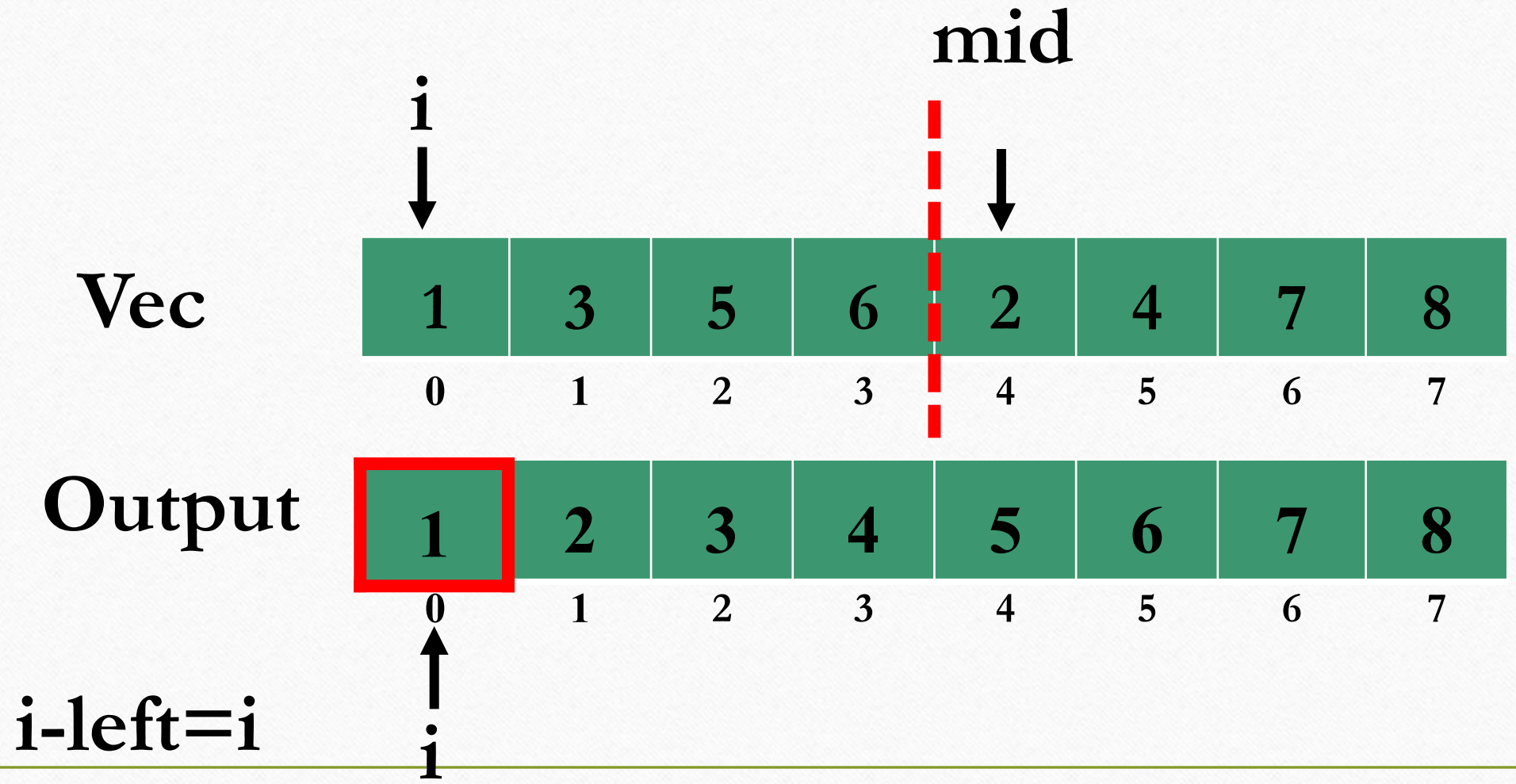
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

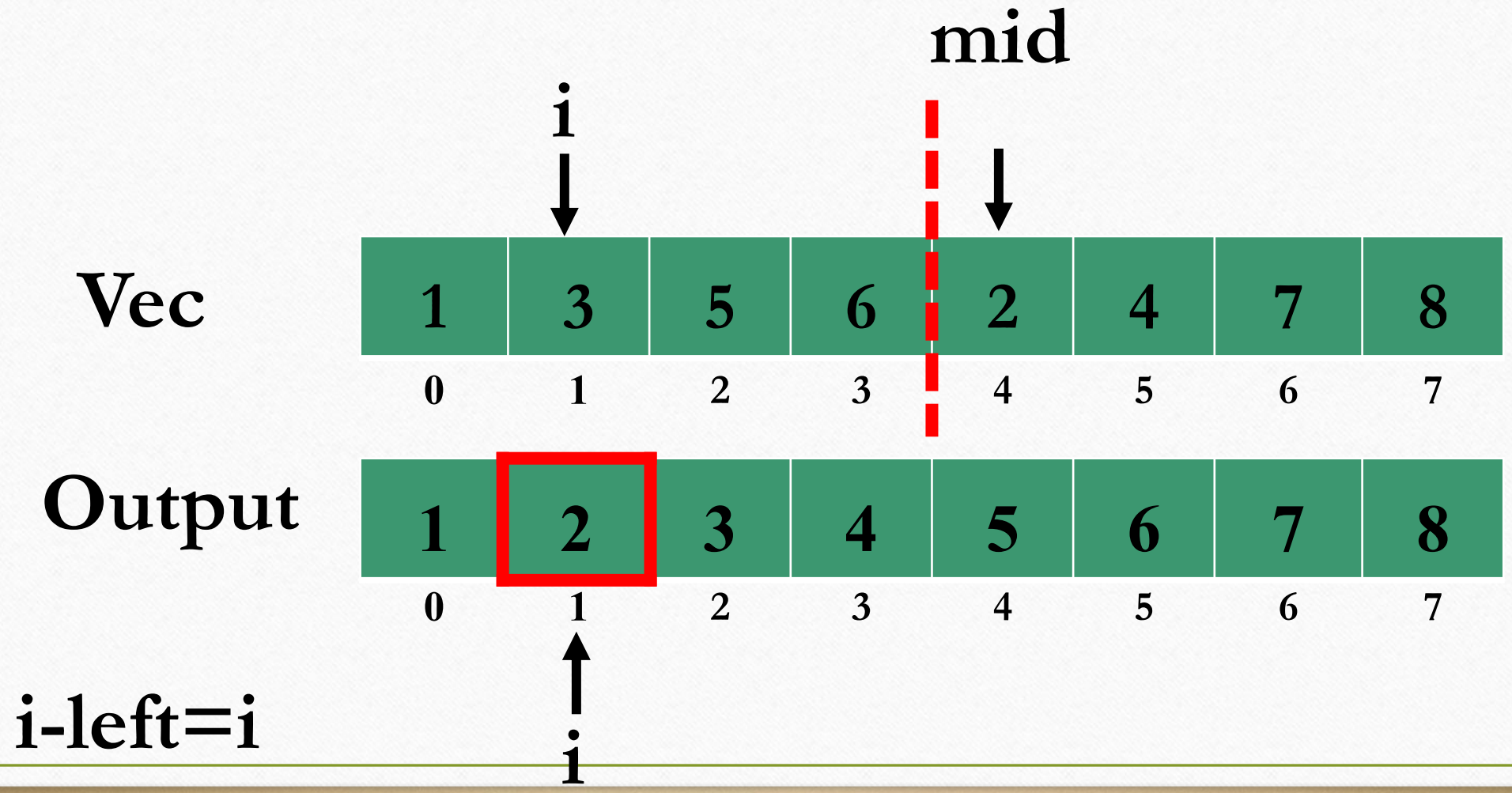
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

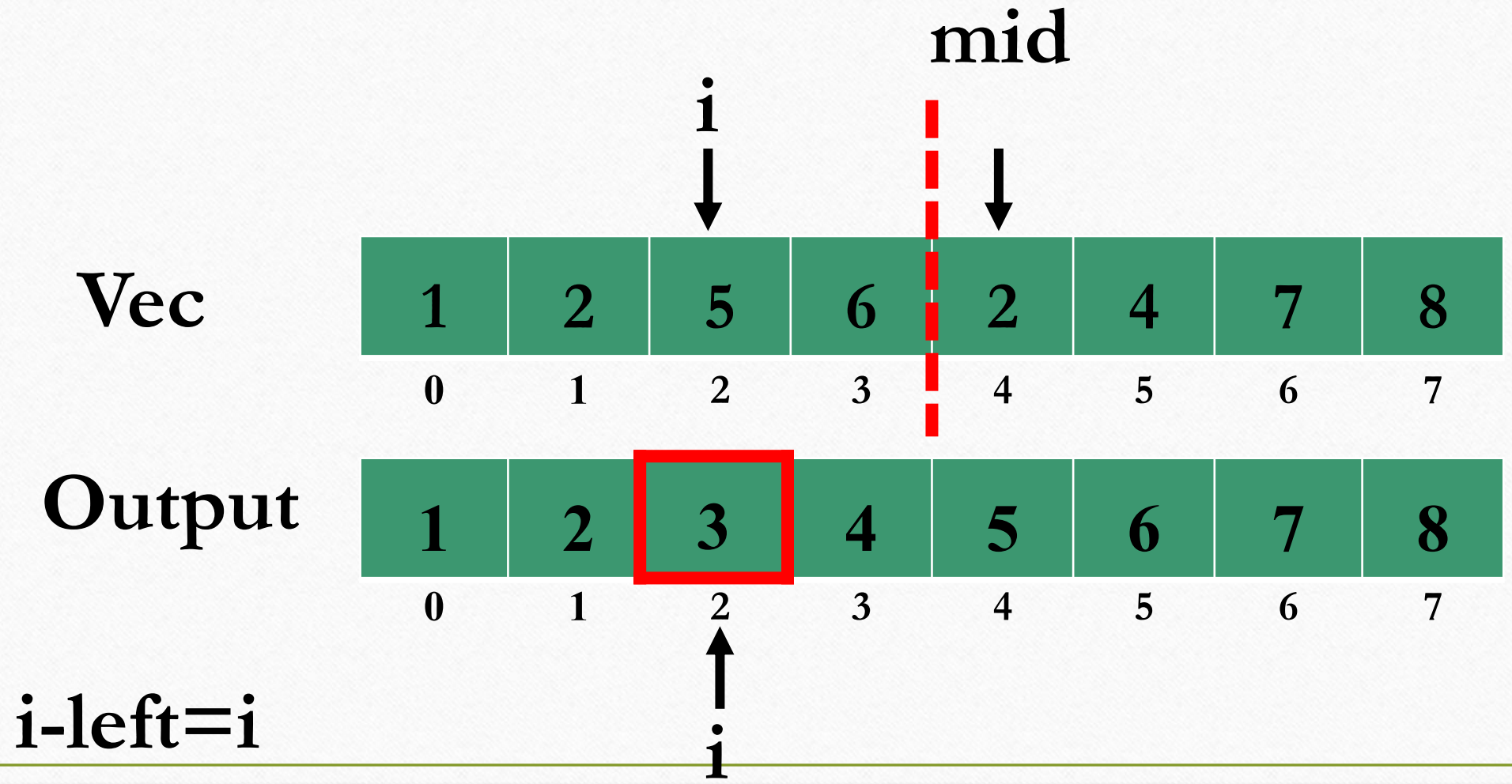
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

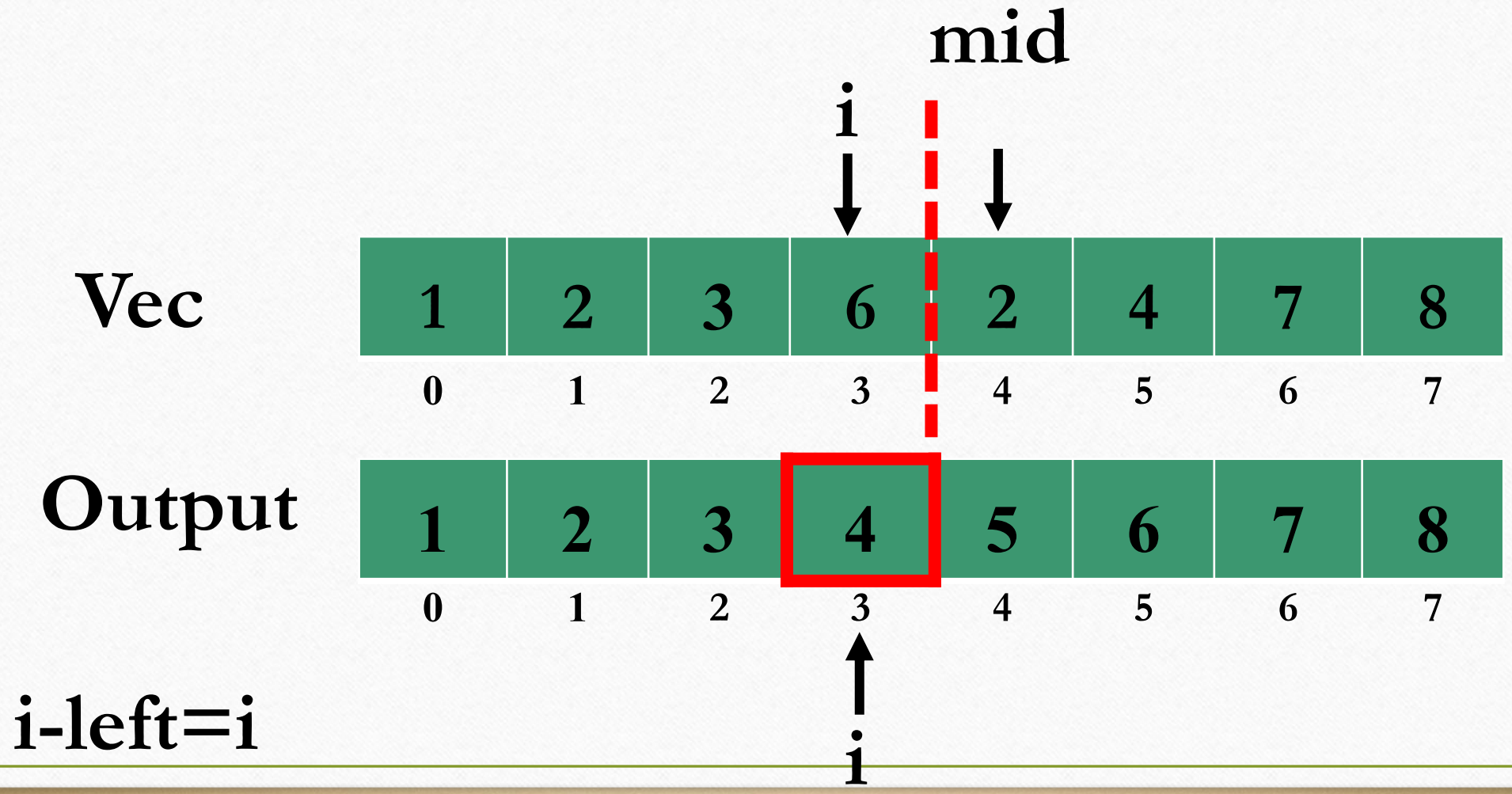
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

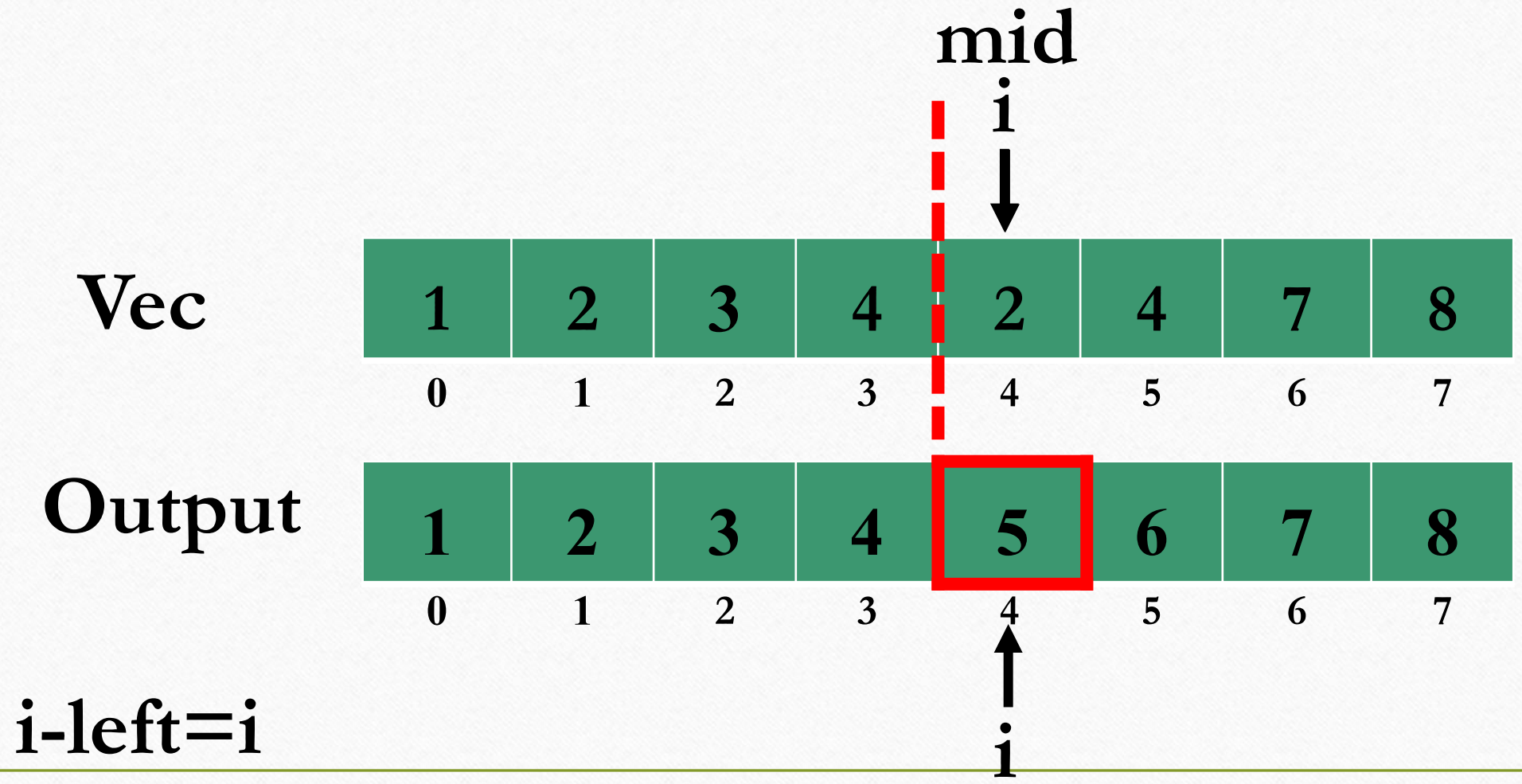
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

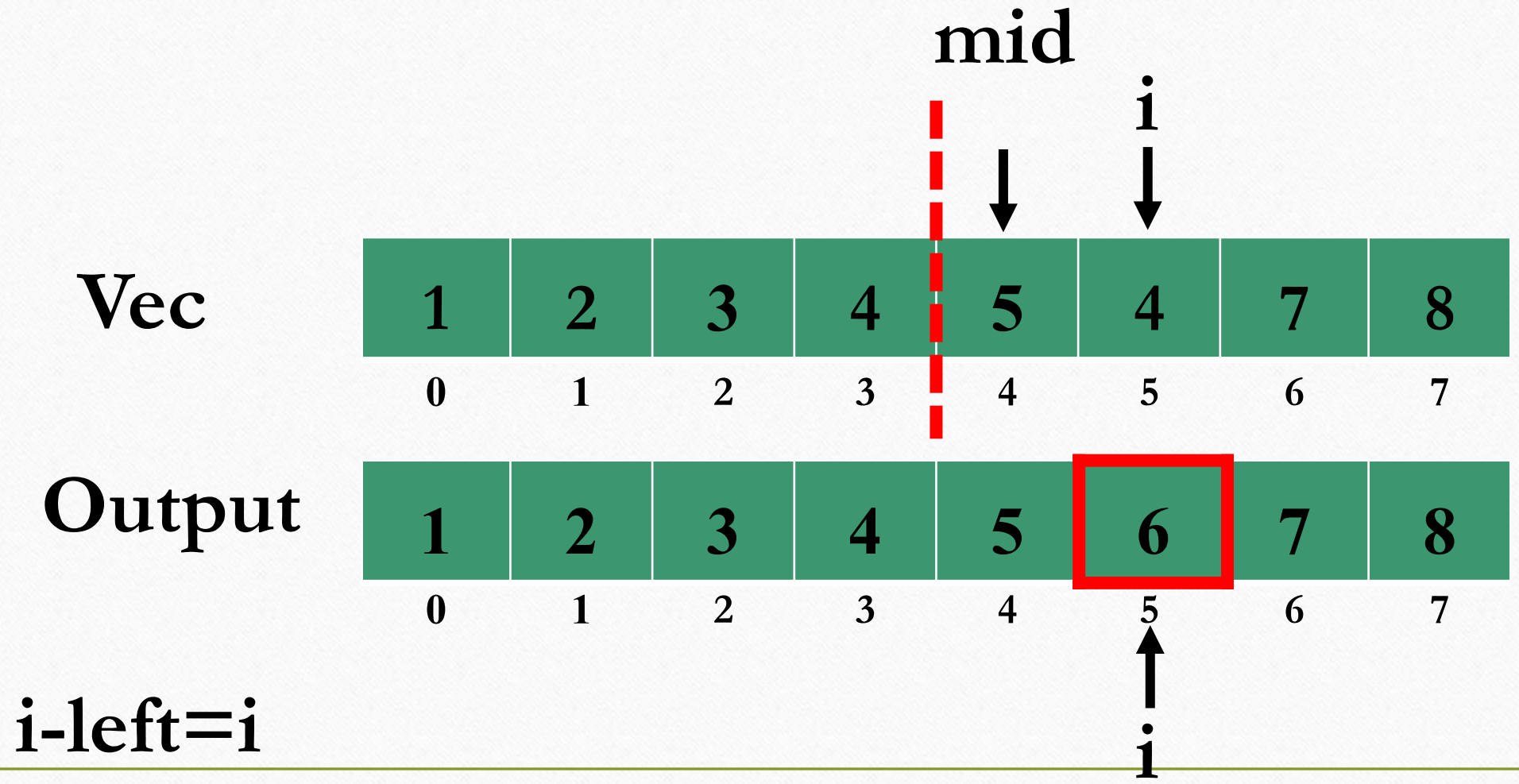
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

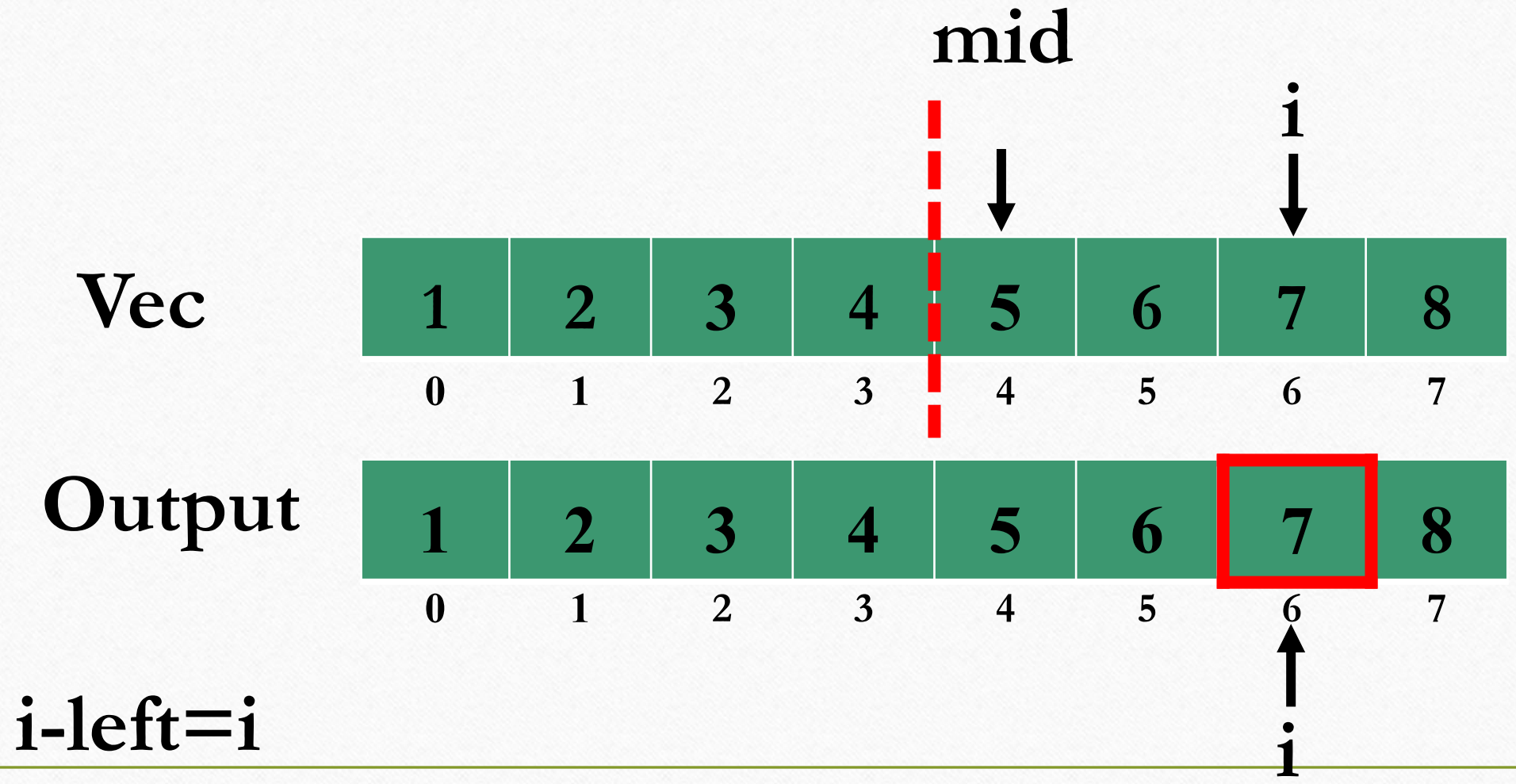
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

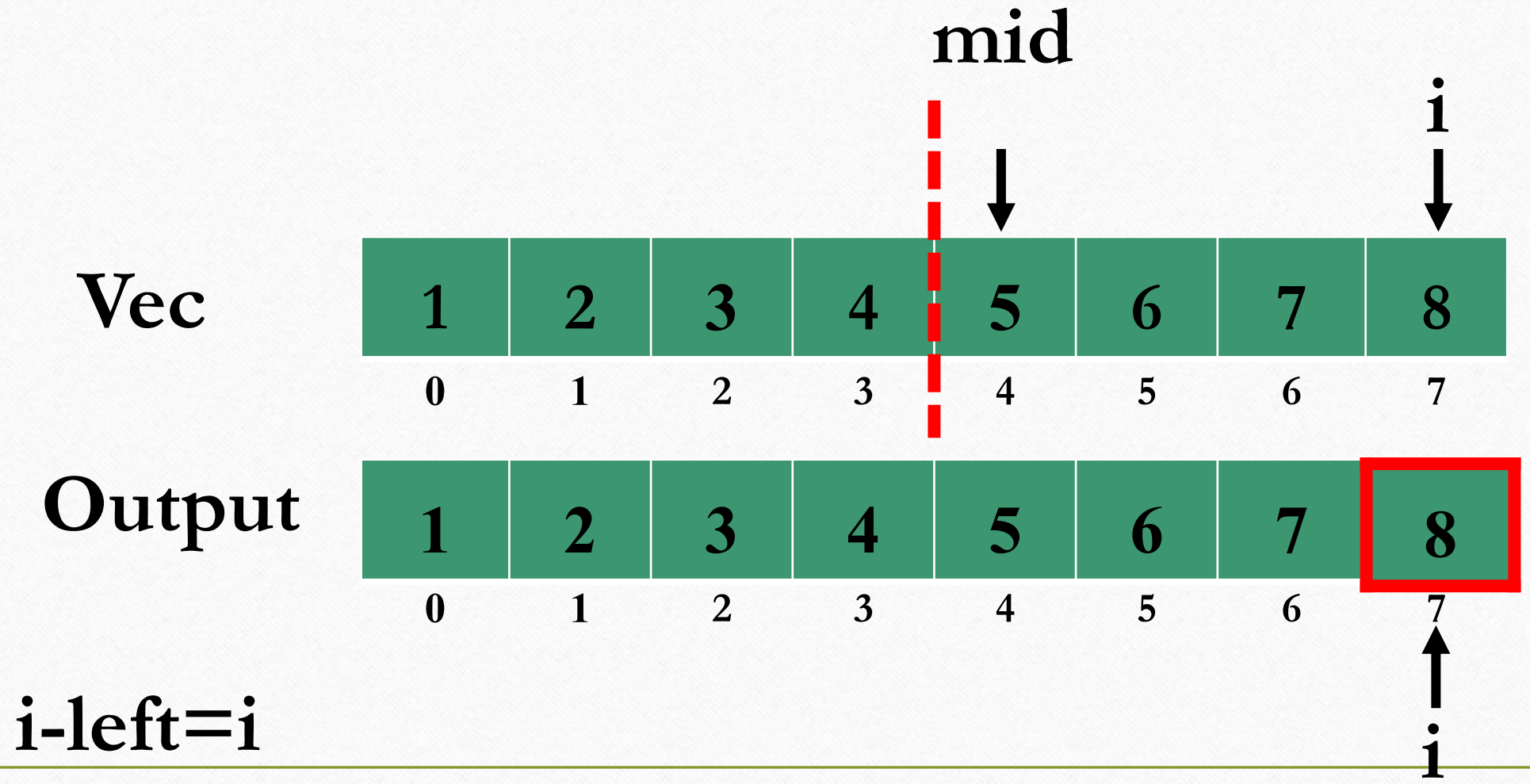
VIII. Merge Sort



```
merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

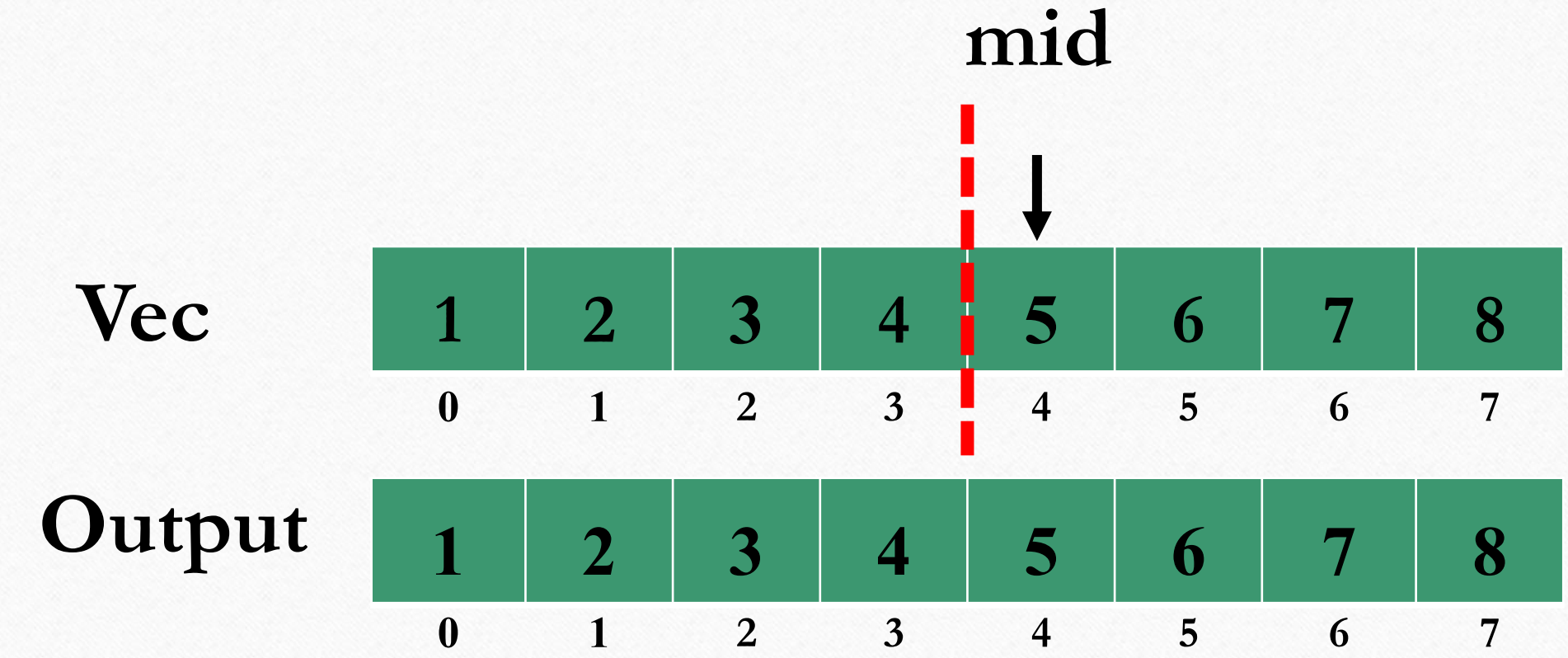
VIII. Merge Sort



```
void merge(Vec, 0, 4, 8)
```

```
for (int i = left; i < right; i++){
    Vec[i] = output[i-left];
}
```

VIII. Merge Sort



i-left=i

VIII. Merge Sort

Time Complexity of Merge Sort

```
void merge_sort(int Vec[], int left, int right) {
   if (right - left < 2) {
      return;
   } // if
   int mid = left + (right - left) / 2;
   merge_sort(Vec, left, mid);
   merge_sort(Vec, mid, right);
   merge(Vec, left, mid, right);
} // merge_sort()
```

VIII. Merge Sort

```
void merge(int Vec[], int left, int mid, int right) {
    int size = right - left;
    int output[size]; //init output vector
    for (int i = left, j = mid, k = 0; k < size; k++) {
        if (i == mid) {output[k] = Vec[j++];}
        else
            if (j == right) {output[k] = Vec[i++];}
            else {
                if (Vec[i] <= Vec[j]) {output[k] = Vec[i++];}
                else {output[k] = Vec[j++];}
            }
    } // for
    for (int i = left; i < right; i++){
        Vec[i] = output[i-left];
    }
} // merge()
```

$\mathcal{O}(1)$
$\mathcal{O}(1)$
$\mathcal{O}(1)$
$\mathcal{O}(1)$
$\mathcal{O}(1)$
$\mathcal{O}(1)$
$\mathcal{O}(n)$
$\mathcal{O}(n)$
$\mathcal{O}(1)$
$\mathcal{O}(n)$

VIII. Merge Sort

Time Complexity of Merge Sort

- What is the time complexity of merge sort?
- If we look at the ***merge()*** function implemented above, we can see that the algorithm does a single pass through all the elements in its input vector range.
- Thus, the merging operation takes $\boldsymbol{O(n)}$ time.

VIII. Merge Sort

Time Complexity of Merge Sort

- We can now express the merge sort function as a recurrence relation.
- Since merge sort makes two recursive calls with input size $\mathbf{n/2}$ and calls ***merge()*** (which takes linear time), the merge sort process can be expressed using the following recurrence relation:

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ 2T(n/2) + n & if\ n > 1 \end{cases}$$

- Using Master's Theorem gives us a complexity of $O(n \log n)$.

VIII. Merge Sort

Space Complexity of Merge Sort

- Additionally, since merge sort initializes an output vector to store the contents of the sorted output, the auxiliary space required by the algorithm is $\boldsymbol{\mathcal{O}(n)}$, as the size of the output vector is linear on the number of elements that need to be sorted.

VIII. Merge Sort

Iterative version of Merge sort

- The merge sort implementation we have presented is based on a **top-down approach**, since it uses recursion to break the input array **into halves that can be sorted individually**.
- Nevertheless, there exists an alternative **method to implement merge sort iteratively,** devoid of recursive invocations.

VIII. Merge Sort

Iterative version of Merge sort

- The following is an implementation of this bottom-up merge sort:

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
       merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
} // merge_sort()
```

VIII. Merge Sort

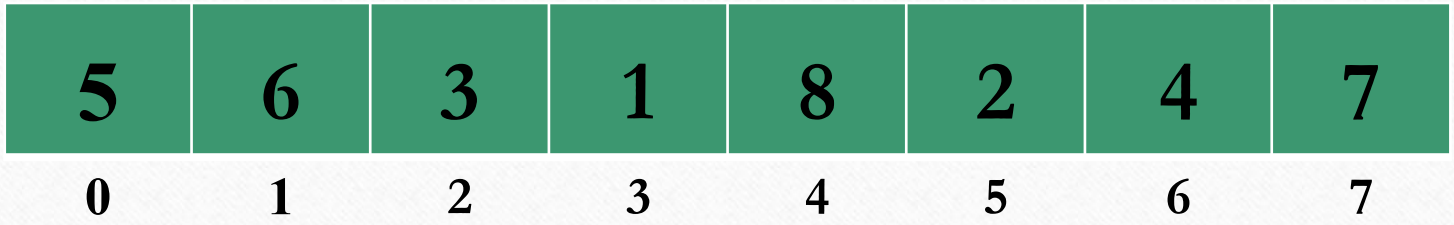
Example:

- This variation of merge sort uses **nested for loops to simulate** the behavior of the recursive calls.
- First, it merges groups of two elements, then it merges groups of four elements, then it merges groups of eight elements, and so on until the entire array is sorted.
- To illustrate this, consider the unsorted array [**5, 6, 3, 1, 8, 2, 4, 7**].

VIII. Merge Sort

Example:

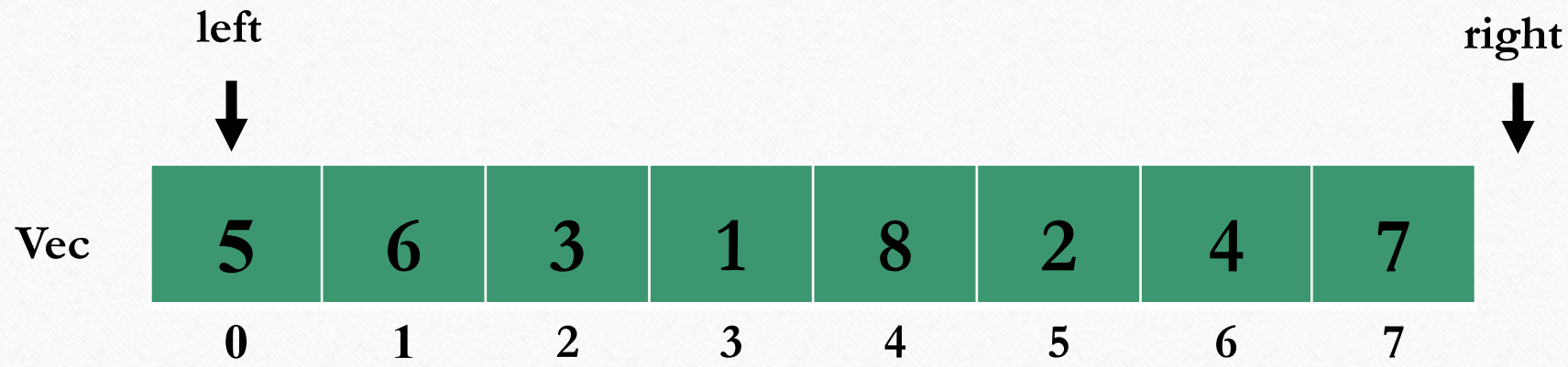
- To illustrate this, consider the unsorted array **[5, 6, 3, 1, 8, 2, 4, 7]**.



VIII. Merge Sort

merge_sort(Vec, 0, 8)

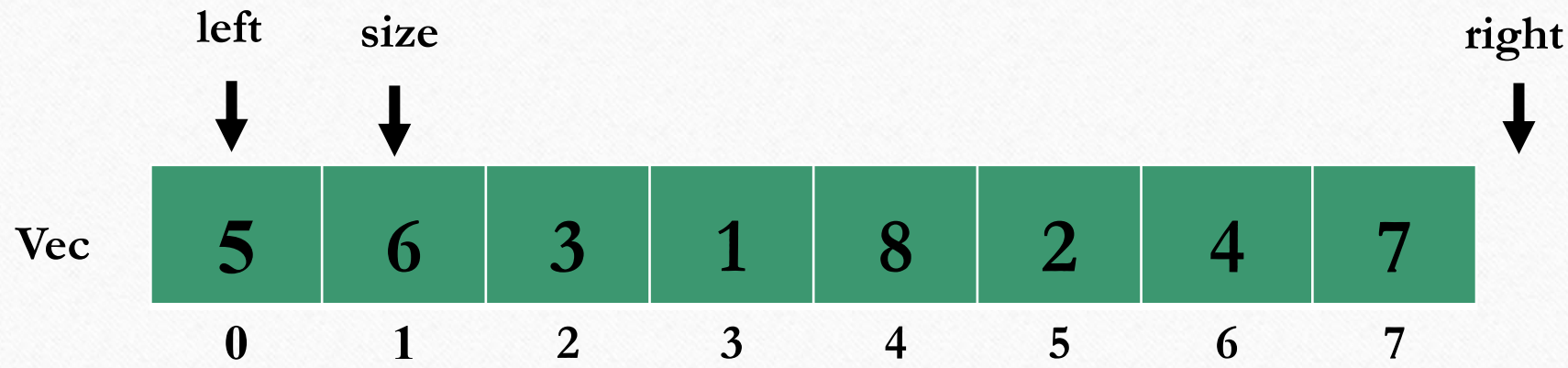
```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```



VIII. Merge Sort

merge_sort(Vec, 0, 8)

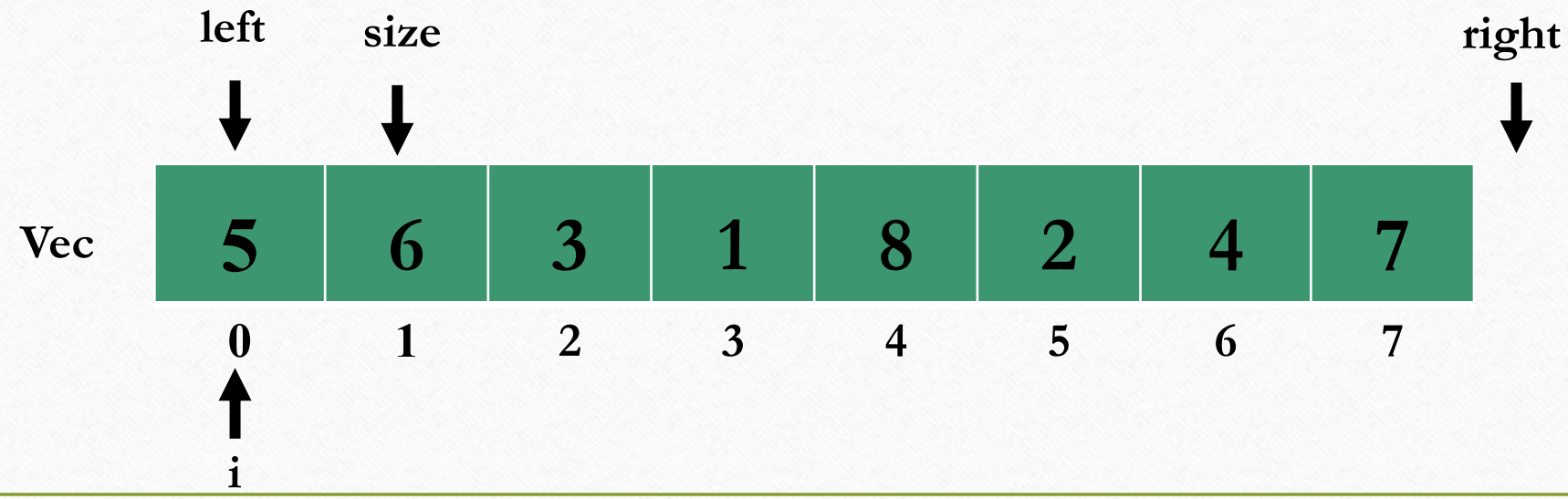
```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```



VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

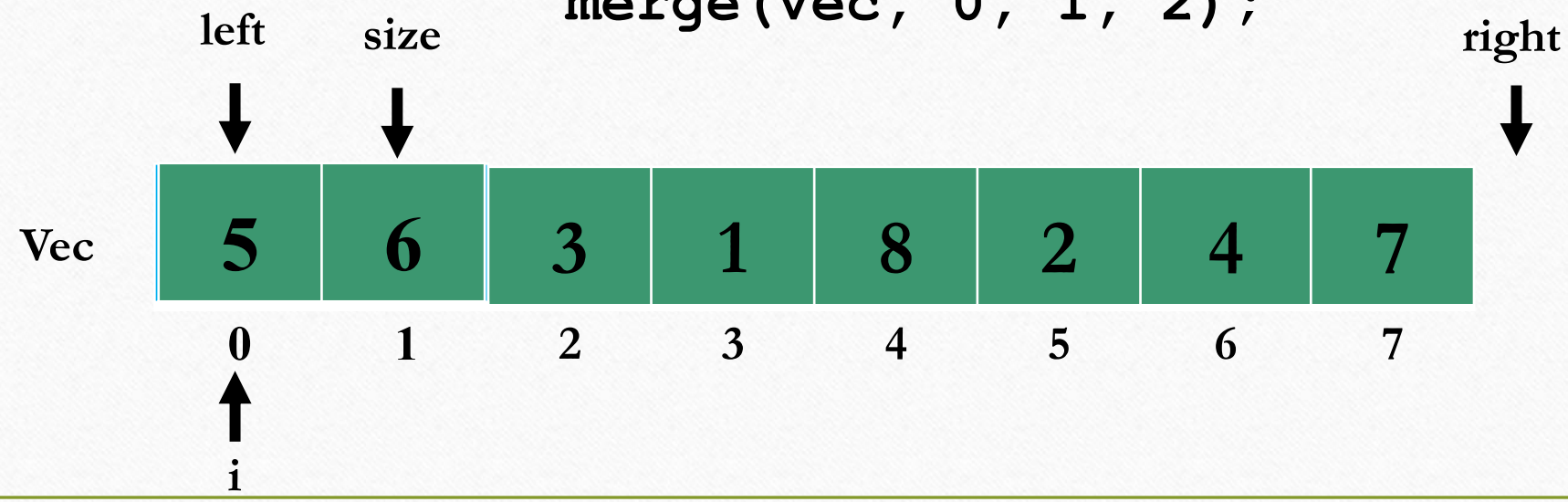


VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

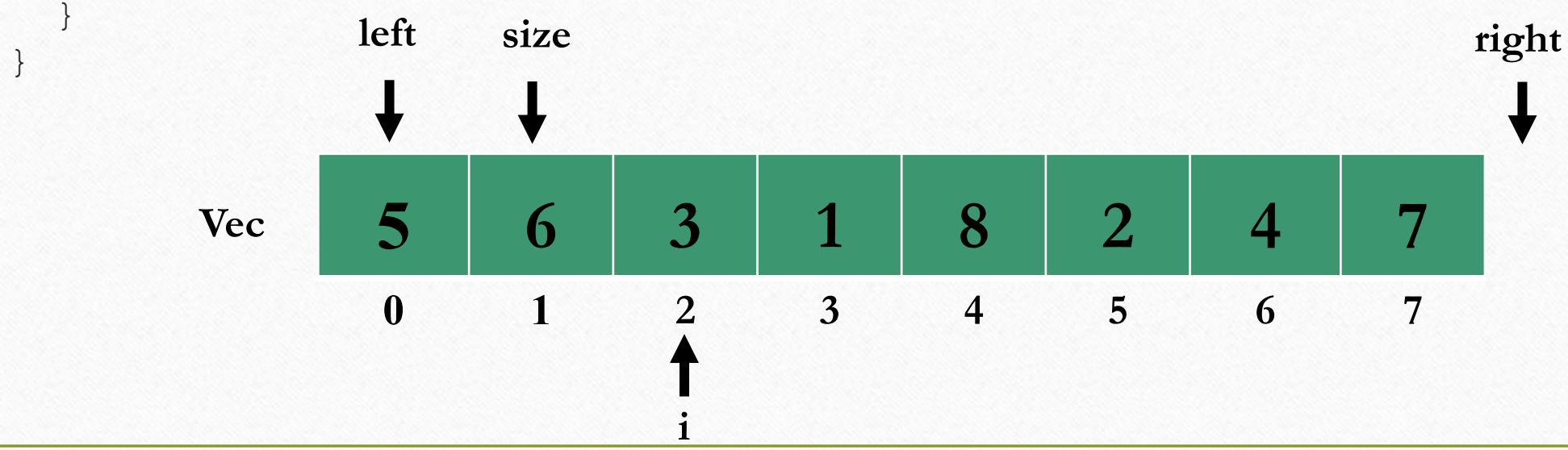
merge(Vec, 0, 1, 2);



VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

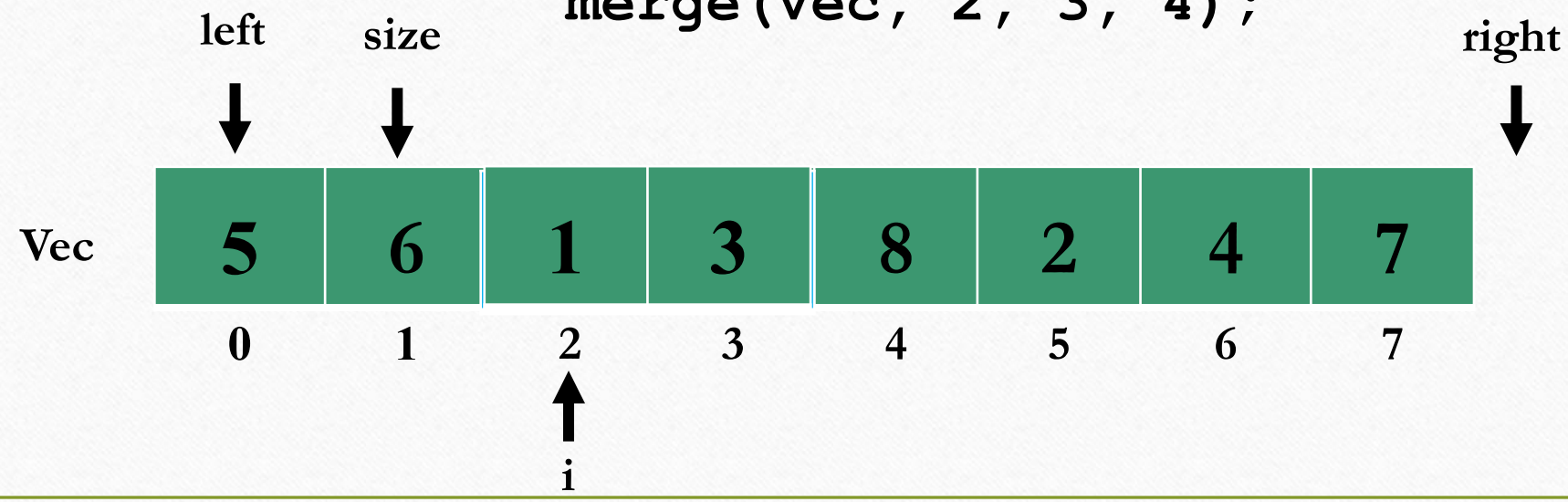


VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

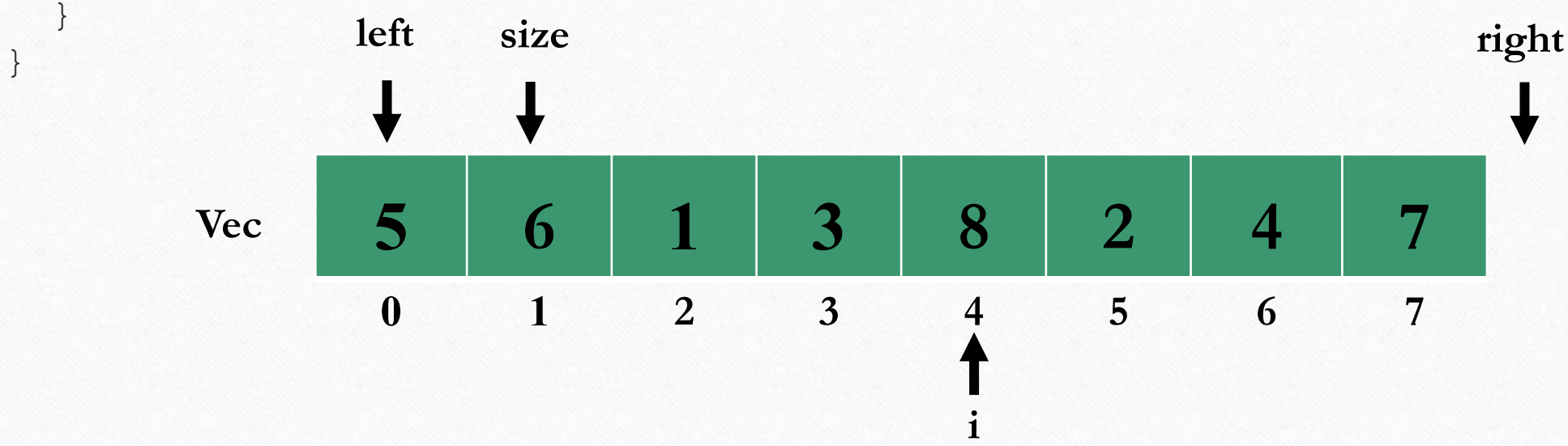
merge(Vec, 2, 3, 4);



VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

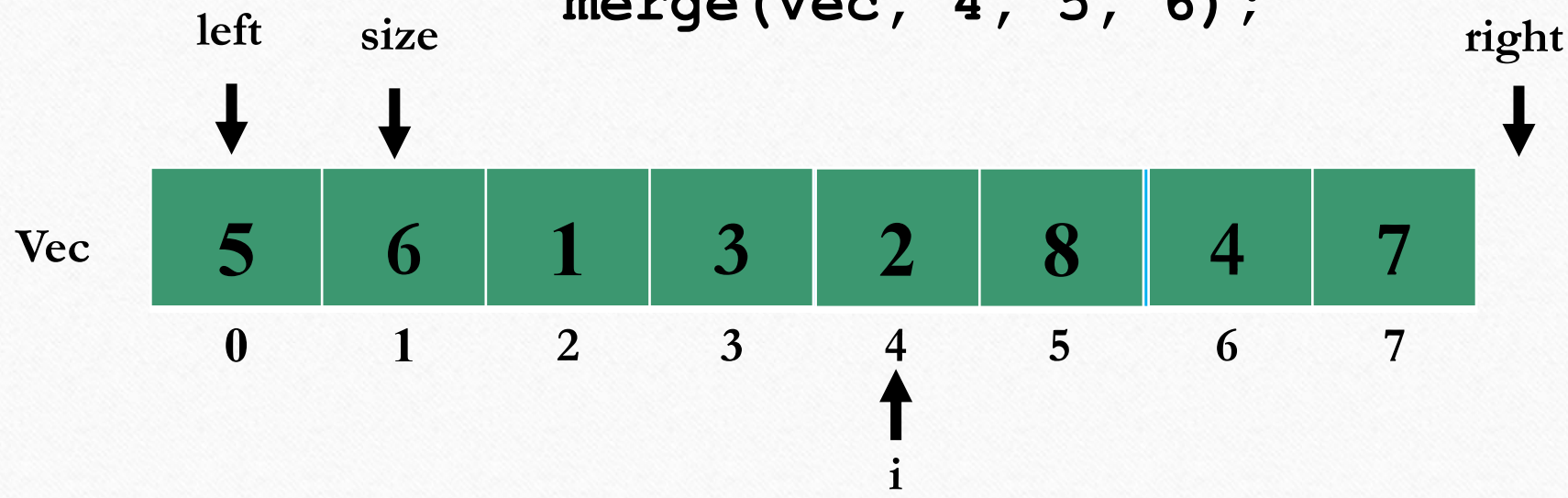


VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

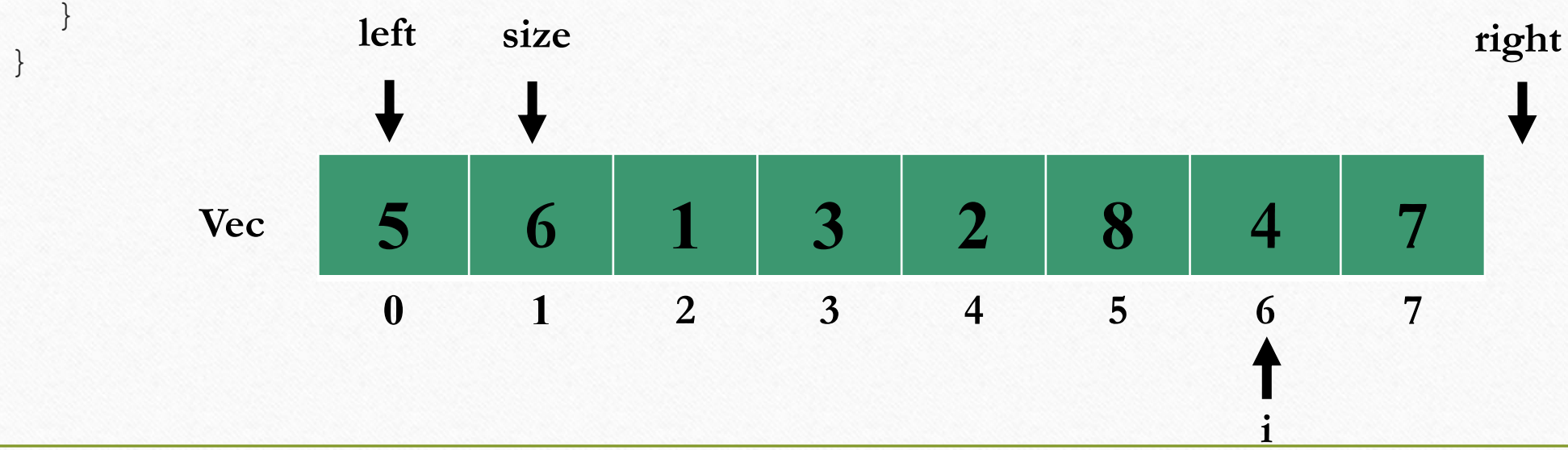
merge(Vec, 4, 5, 6);



VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

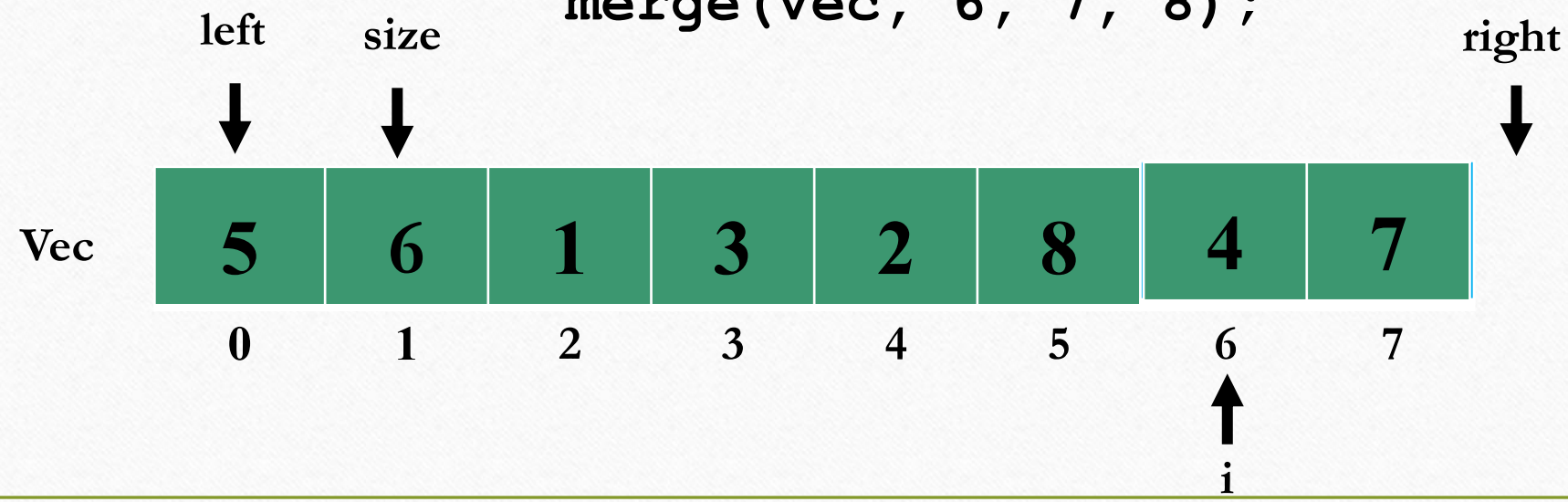


VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

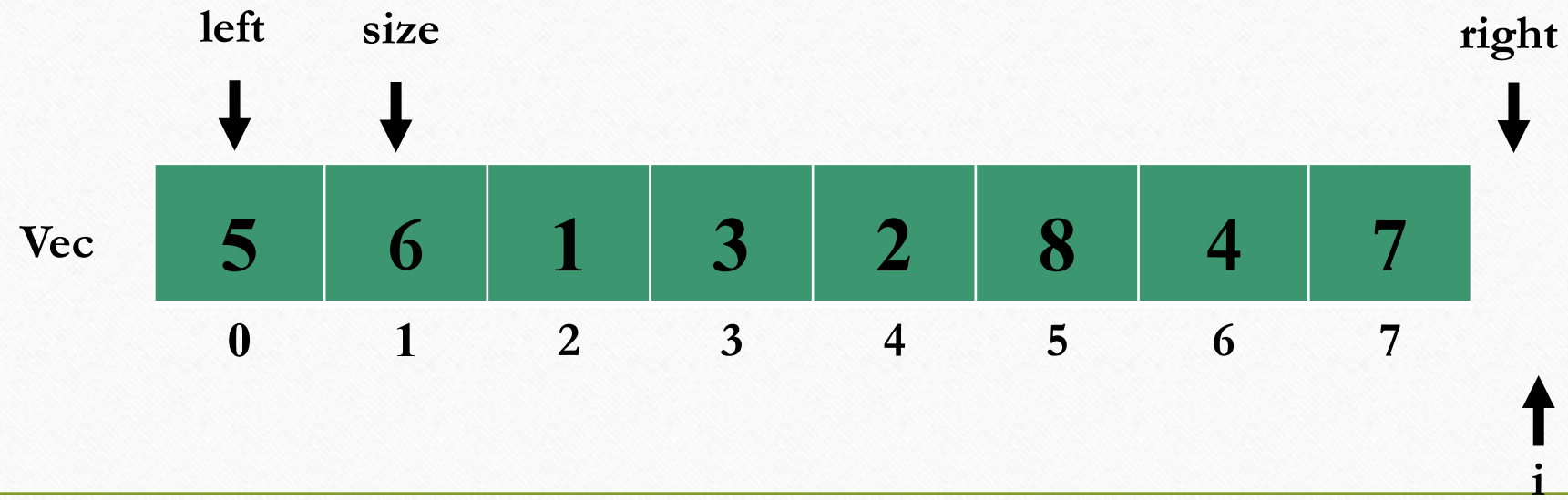
merge(Vec, 6, 7, 8);



VIII. Merge Sort

merge_sort(Vec, 0, 8)

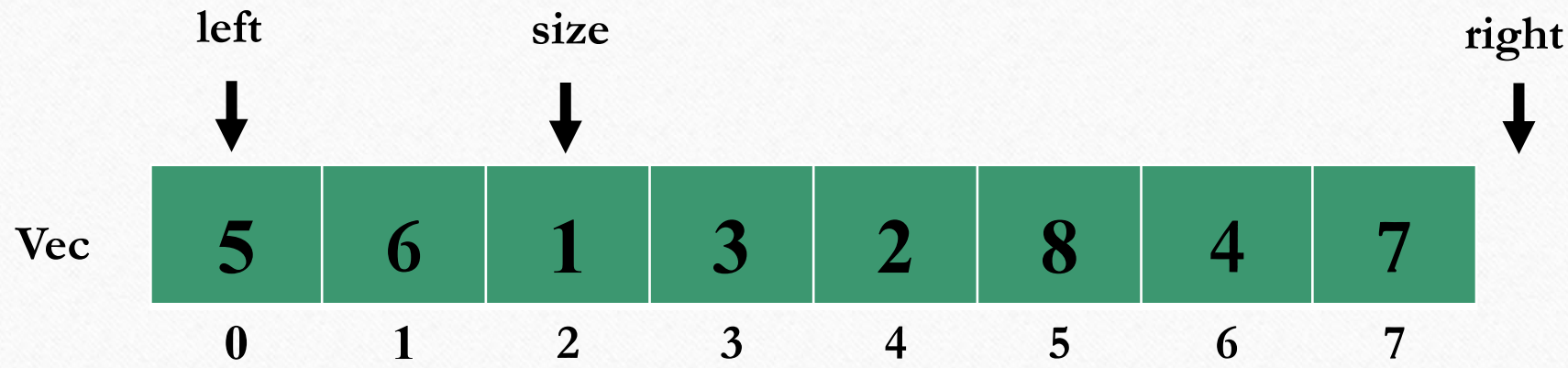
```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```



VIII. Merge Sort

merge_sort(Vec, 0, 8)

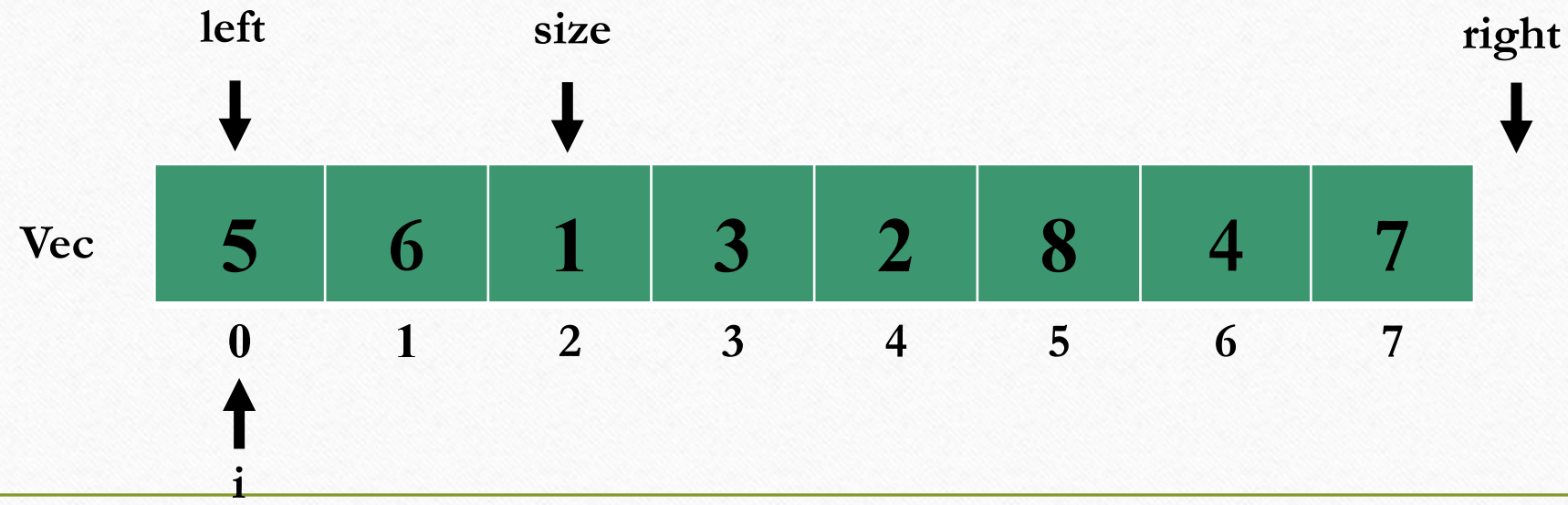
```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```



VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

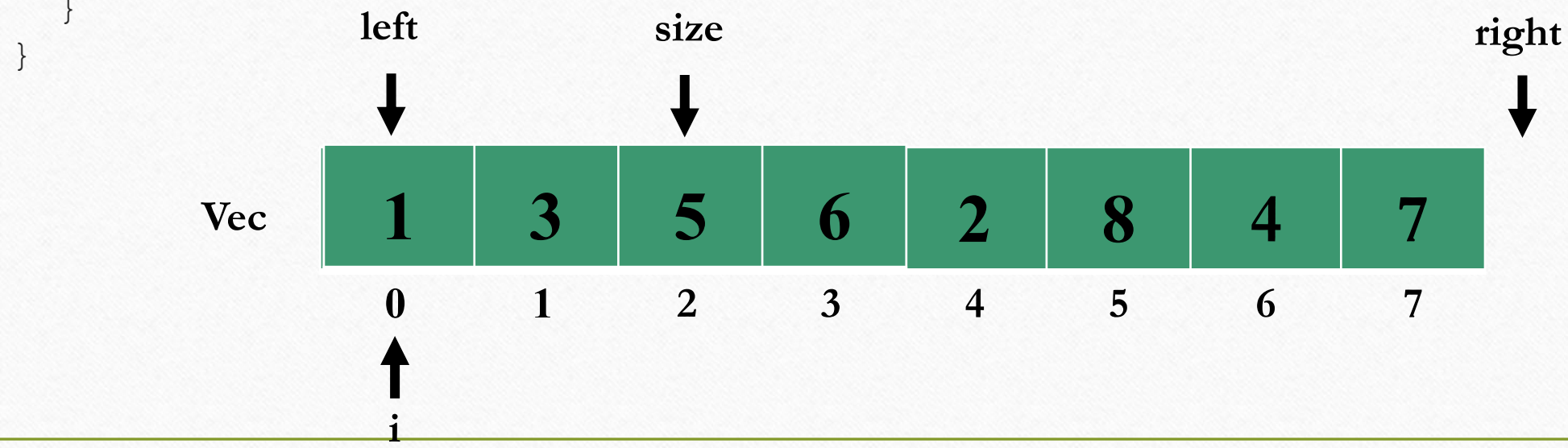


VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

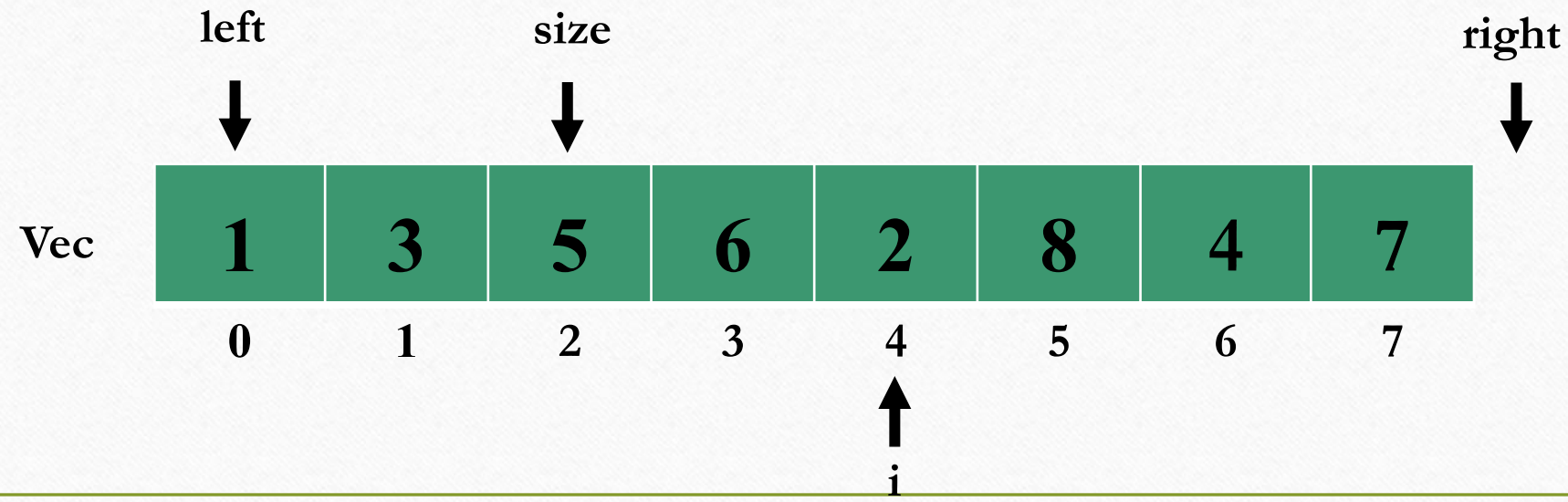
merge(Vec, 0, 2, 4);



VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

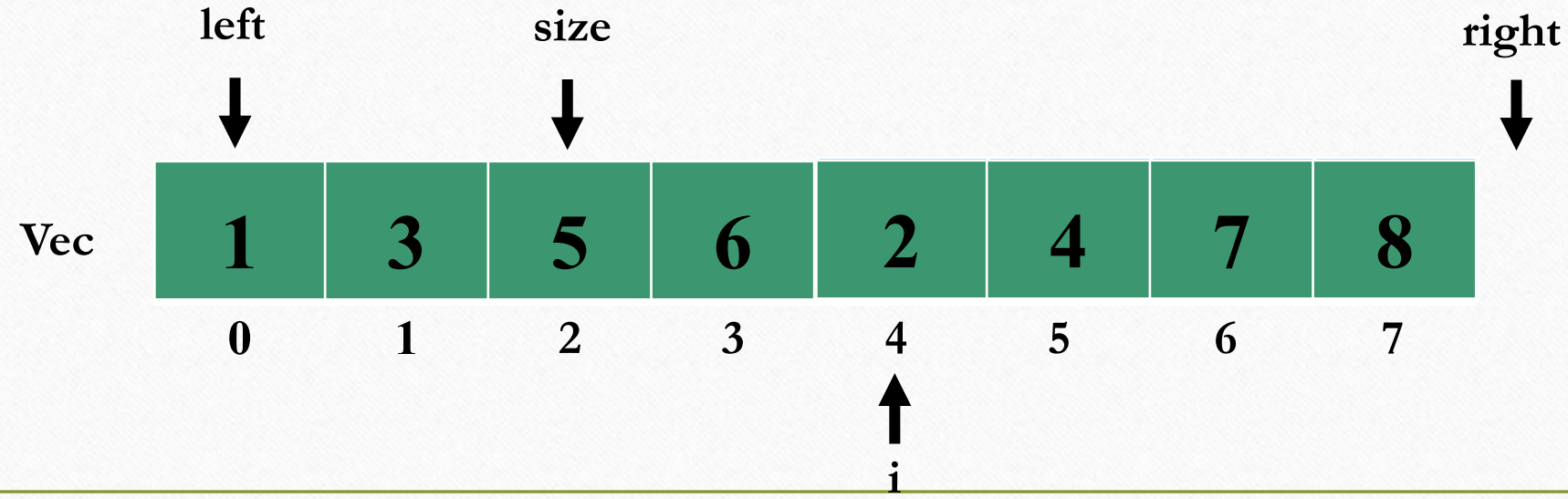


VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

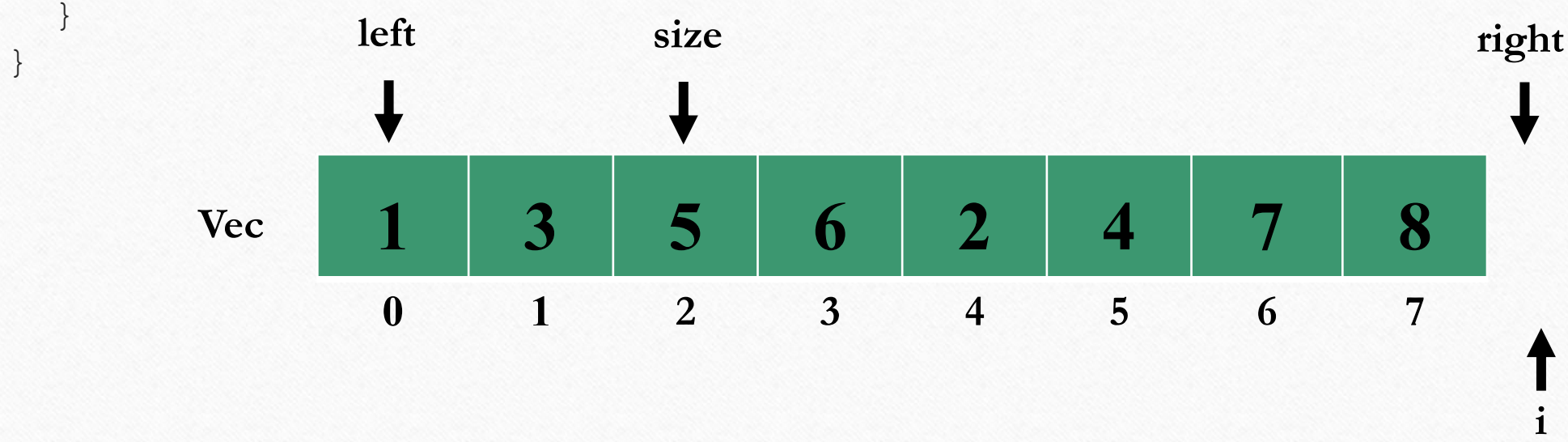
merge(Vec, 4, 6, 8);



VIII. Merge Sort

merge_sort(Vec, 0, 8)

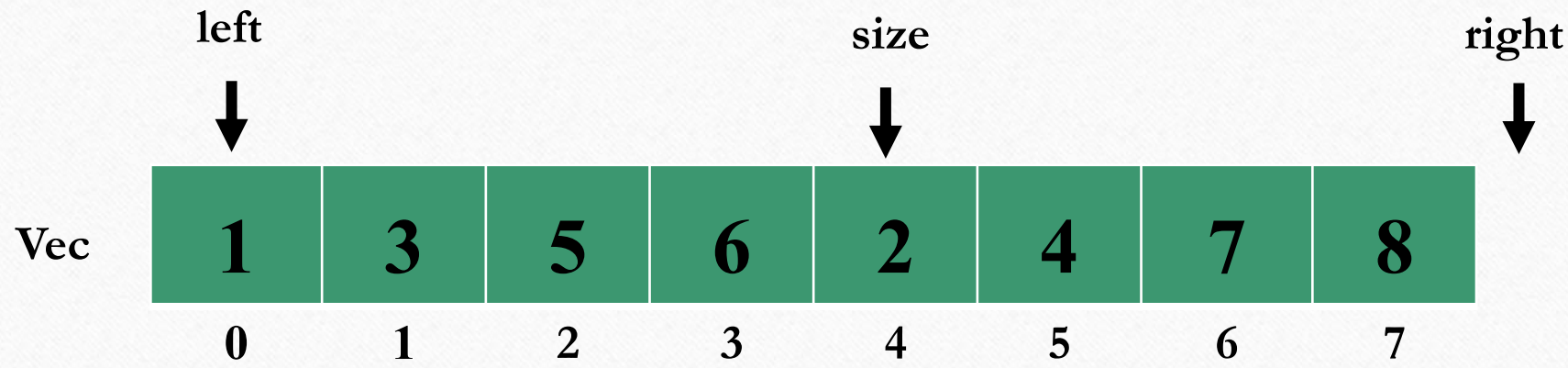
```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```



VIII. Merge Sort

merge_sort(Vec, 0, 8)

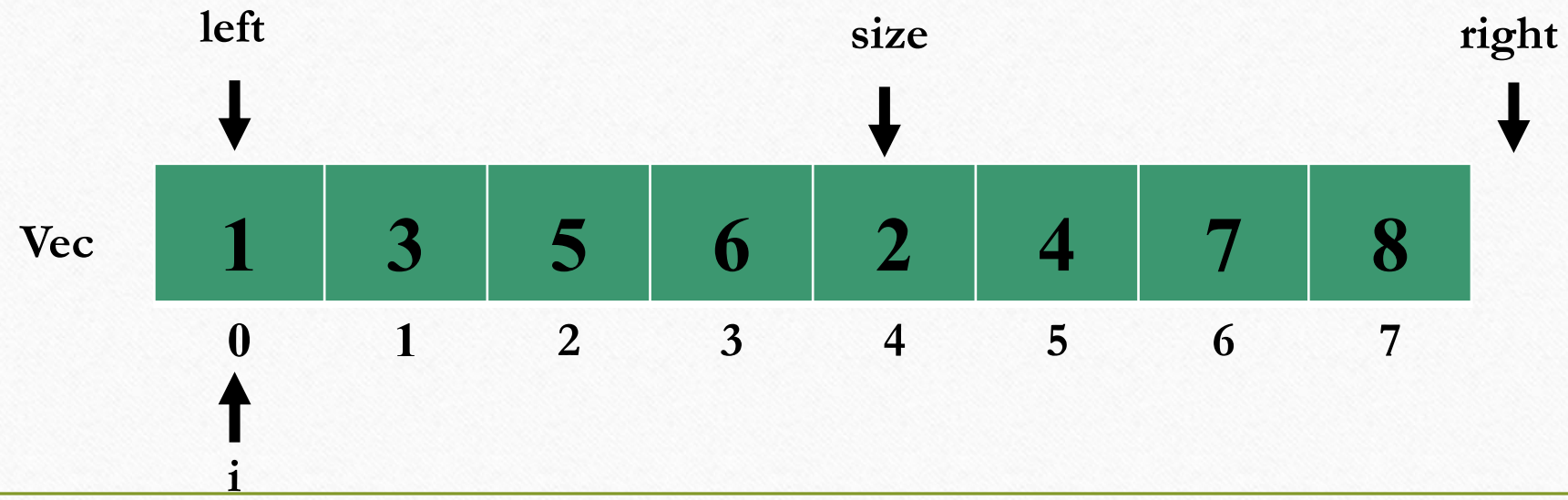
```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```



VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

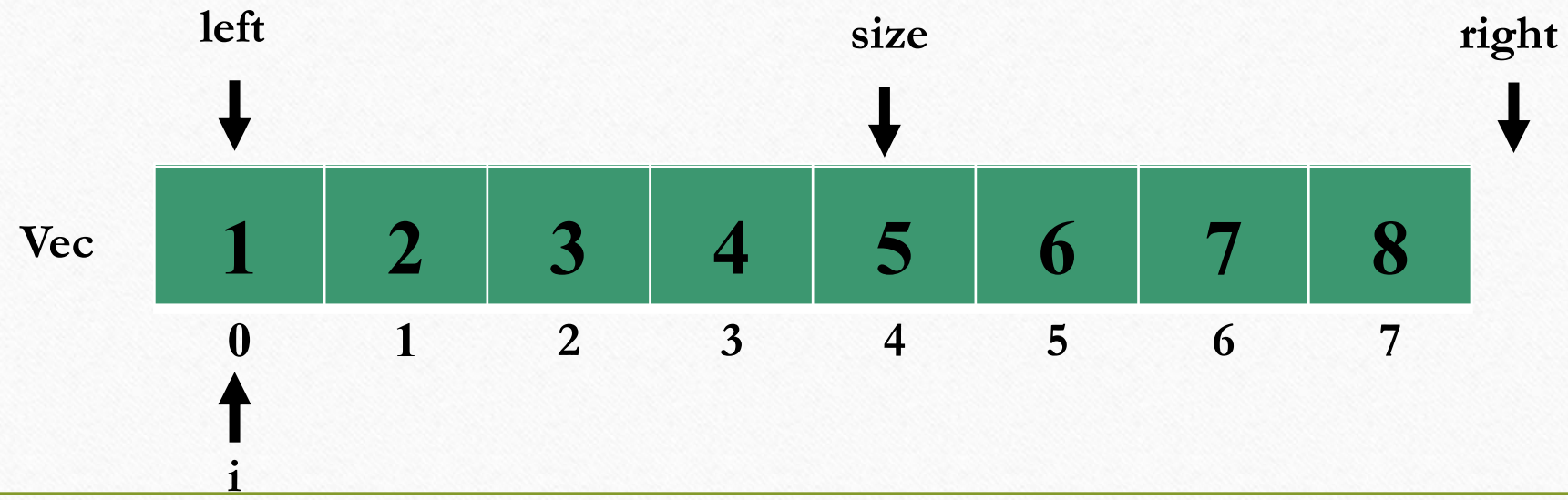


VIII. Merge Sort

merge_sort(Vec, 0, 8)

```
void merge_sort(int Vec[], int left, int right) {
  for (int size = 1; size <= right - left; size *= 2) {
    for (int i = left; i <= right - size; i += 2 * size) {
      merge(Vec, i, i + size, min(i + 2 * size, right));
    }
  }
}
```

merge(Vec, 0, 4, 8);



IX. Analysis of Comparison Sorts

All of the sorting algorithms we have discussed so far are **comparison-based** sorting algorithms. These sorting algorithms sort a list (array) by comparing elements with each other using a **comparison operator**.

Algorithm	Best Time	Average Time	Worst Time	Auxiliary Space
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

Questions ?