Programming for Engineers

Lecture 6: Sorting Algorithms

Course ID: EE057IU

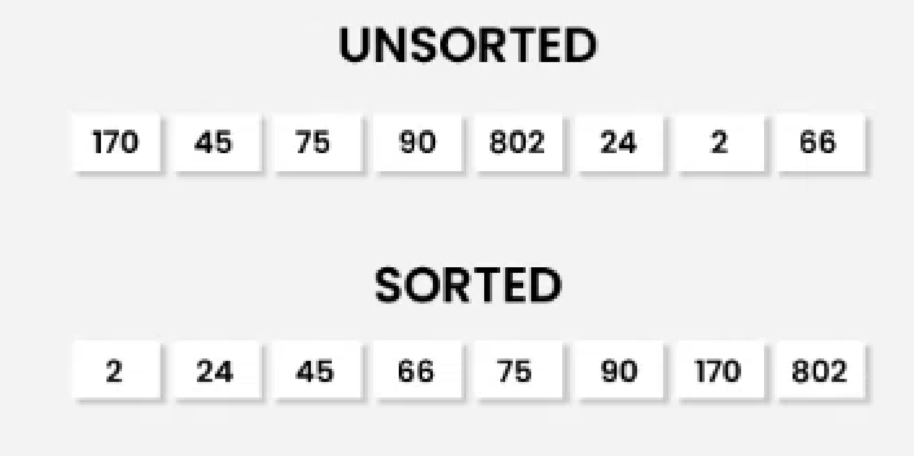
Lecture Outline

Introduction

Sorting

- refers to rearrangement of a given array or list of elements
- The **comparison operator** determines the **order** of elements (ascending or descending)

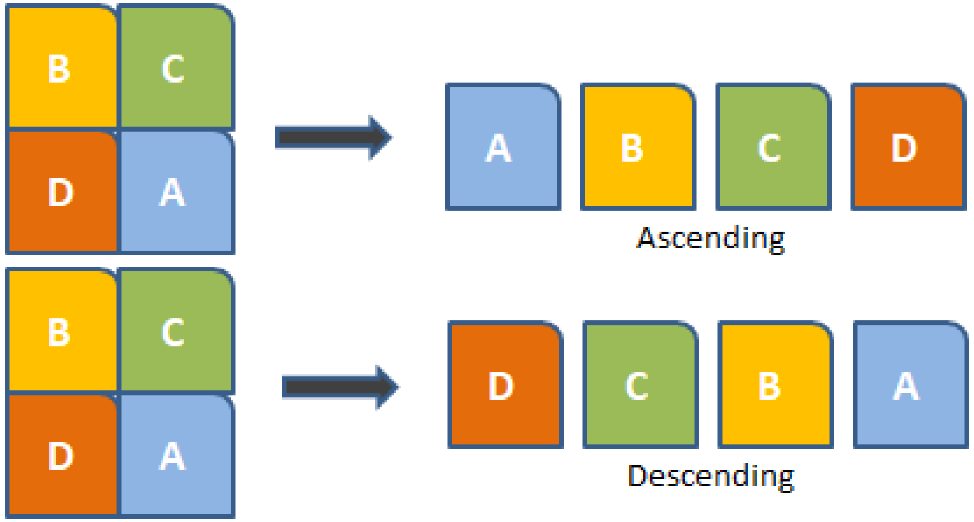
- **Why Sorting Matters**
  - **Large Data Sets**: Sorting becomes essential when dealing with large volumes of randomly arranged data.
  - **Ease of Searching**: Sorting simplifies the process of finding specific elements in the data.



Introduction

➢ Importance of Sorting Data

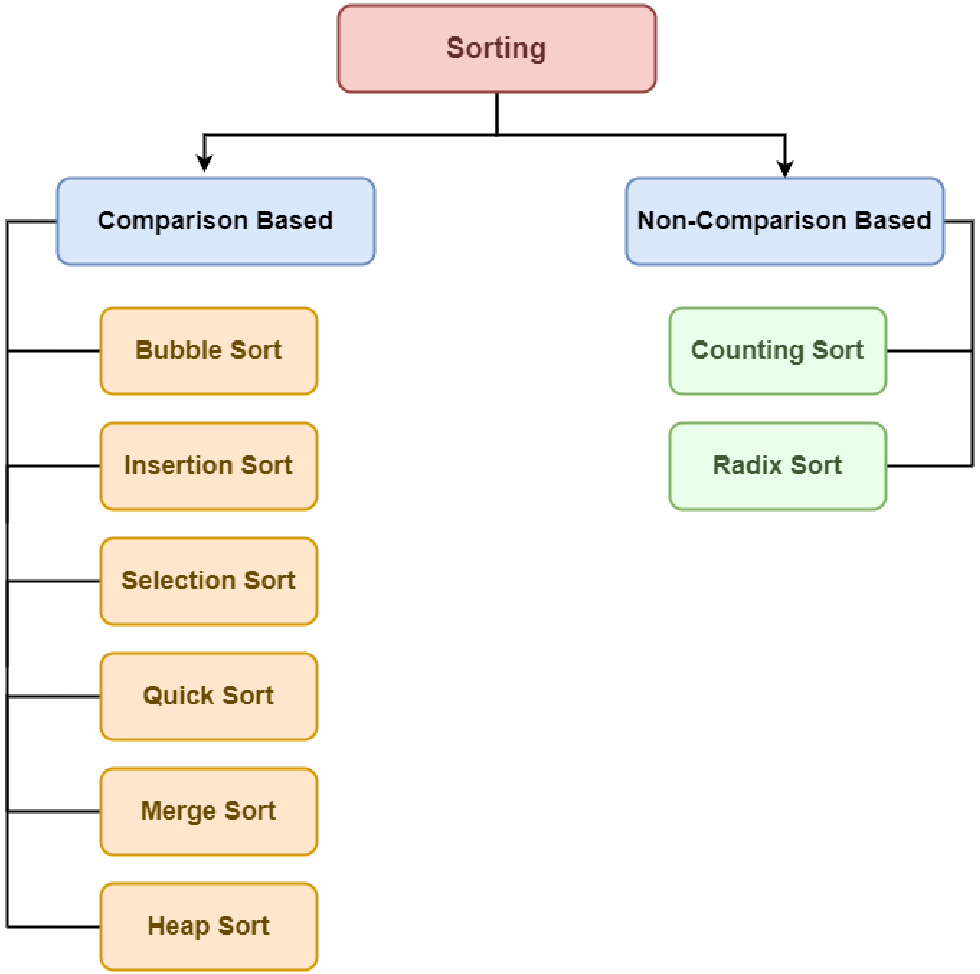
- **Organizes Data**: Places items in ascending or descending order for easier access.
- **Real-world Examples**:
  - **Banks**: Sort checks by account number to generate monthly statements.
  - **Telephone Companies**: Sort accounts by last and first names for efficient lookup.
- **Essential for Organizations**: Many organizations need to sort large volumes of data.



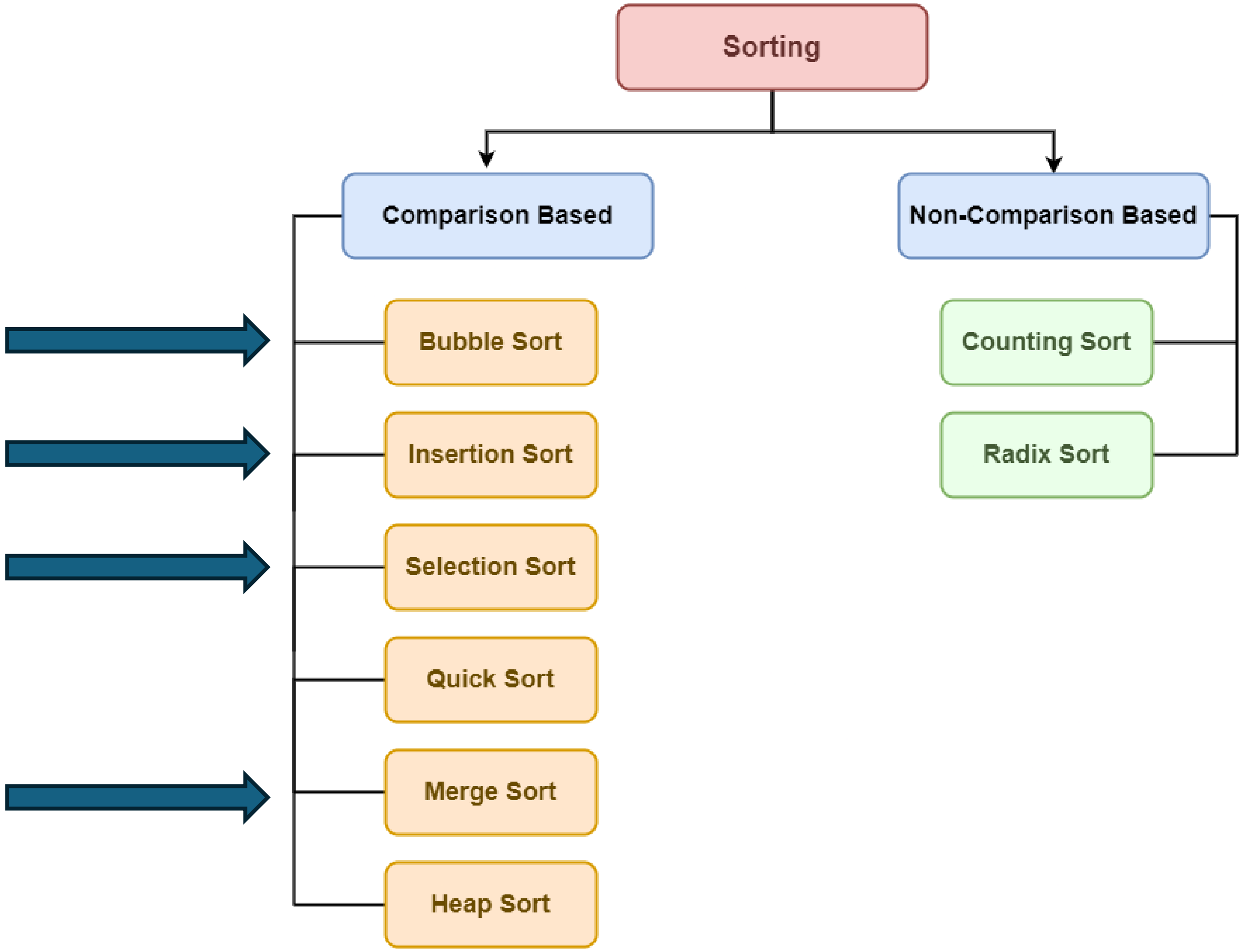
Introduction

- **A Focus on Performance**
  - **Research Significance**: Sorting is a fundamental challenge, inspiring intense study in computer science.
- **Simple vs. Complex Algorithms**
  - **Simple Algorithms:** (Chapter 6 with Bubble Sort)
    - Easy to write, test, and debug.
    - Often have lower performance.
  - **Complex Algorithms:**
    - Better performance but more complicated to implement.
    - Covered in Chapters 12 and 13 for maximum efficiency insights.

Type of Sorting Algorithms



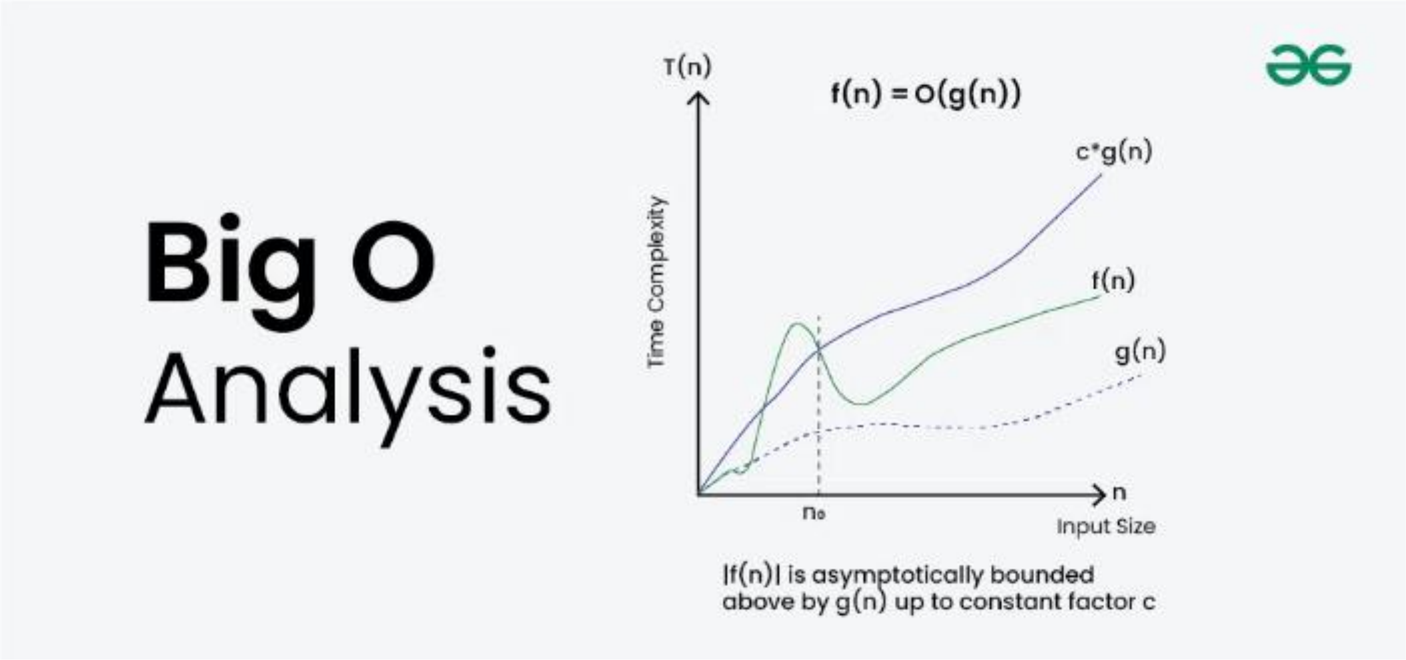
Type of Sorting Algorithms



Big O notation

➢ What is Big O notation?

- A tool used to describe the **time and space complexity** of algorithms
- Provides a **standardized way to compare algorithm efficiency** in terms of worst-case performance.



Big O notation

- **Big-O**
  - referred to as “**Order of**”
  - a way to express the **upper bound** of an algorithm’s time complexity
  - provides an **upper limit** on the time taken by an algorithm in terms of the size of the input
  - denoted as **O(f(n))**, represents the operations an algorithm performs for a problem of size **n**

Big-O notation *is used to describe the performance or complexity of an algorithm. Specifically, it describes the* ***worst-case scenario*** *in terms of* ***time*** *or* ***space complexity.***

“HOW CODE SLOWS AS DATA GROWS”

Common Big-O notations – $O(n)$ vs $O(1)$ complexity

$O(n)$: "order n"

- an algorithm that requires n-1 calculation/comparisons
- referred to as having a linear run time
- running time of an algorithm grows linearly with the size of input

```c
int main() {
    int sum = 0;
    int n=100;
    for(int i=0;i<=n;i++){
        sum += i;
    }
    printf("%d",sum);
    return 0;
}
```

n = 1000
~1000 steps

$O(1)$: "order 1"

- The run time complexity is CONSTANT

```c
int main() {
    int n = 100;
    int sum = n * (n+1)/2;
    printf("%d",sum);
return 0;
}
```

n = 1000
~3 steps

Common Big-O notations - $O(1)$

- $O(1)$ with respect to the size of the file. As the size of the file increases, it won't take any longer to get the file to your friend. The time is constant.
- $O(1)$ is pronounce “order 1”

- Example 1, accessing a value with an array index

```
int arr[5] = {1,2,3,4,5};
int value = arr[2];                    // value = 3
```

- Example 2, compare an array’s first element is equal to its second

```
int arr_1[5] = {1,2,3,4,5};
int arr_2[7] = {1,2,3,4,5,6,7};

if (arr_1[0] == arr_1[1] )
{

}
```

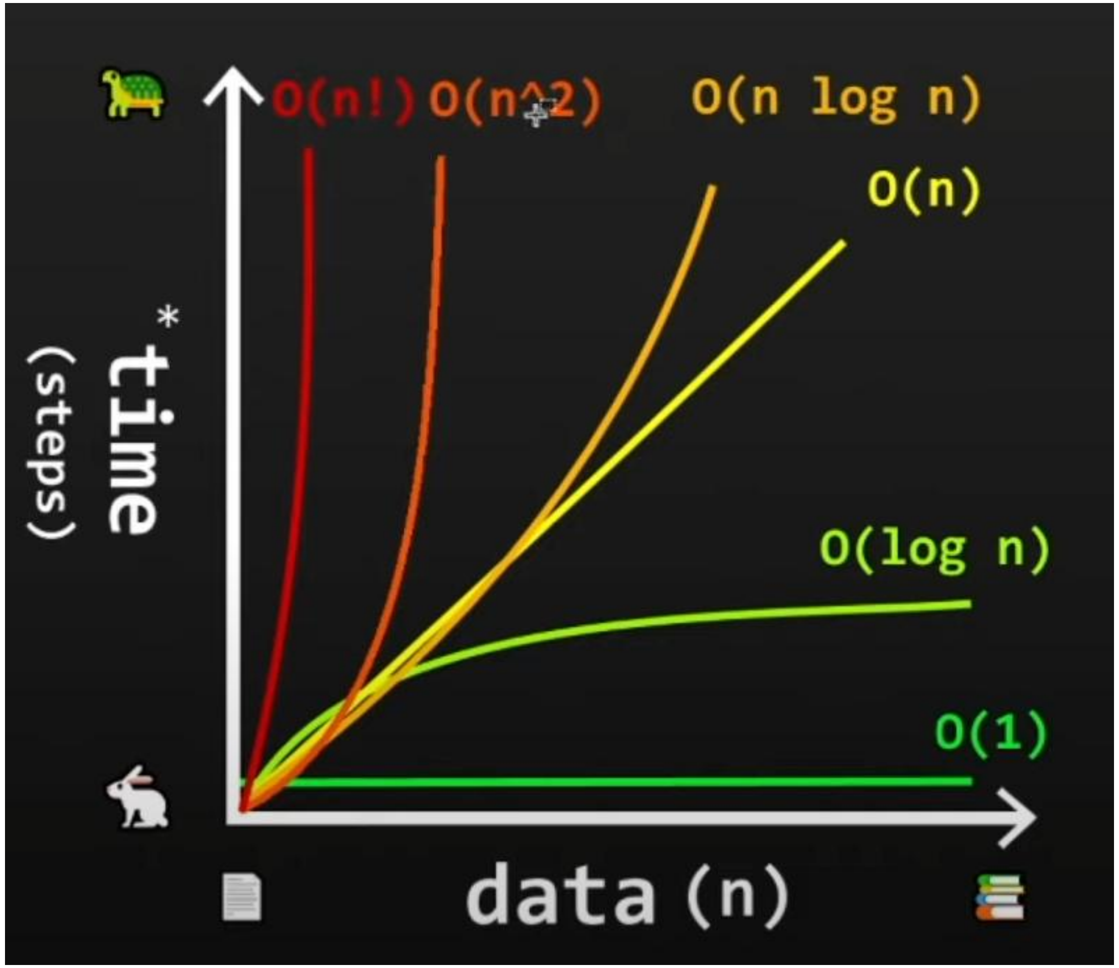
Common Big-O notations – $O(n^2)$ complexity

$\boldsymbol{O(n^2)}$

- referred as quadratic run time
- running time of an algorithm is proportional to the square of the input size

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

Big O notation



Source: Bro Code

$O(1)$ = constant time

- random access of an element in an array

$O(\log n)$ = logarithmic time

-binary search

$O(n)$ = linear time

-looping through elements in an array

$O(n \log n)$ = Quasilinear time

-quick sort

-merge sort

-heap sort

$O(n\text{^}2)$ = Quadratic time

- insertion sort
- selection sort
- bubble sort

$O(n!)$ = factorial time

Common Big-O notations – $O(log\ n)$ complexity

$\boldsymbol{O(log\ n)}$

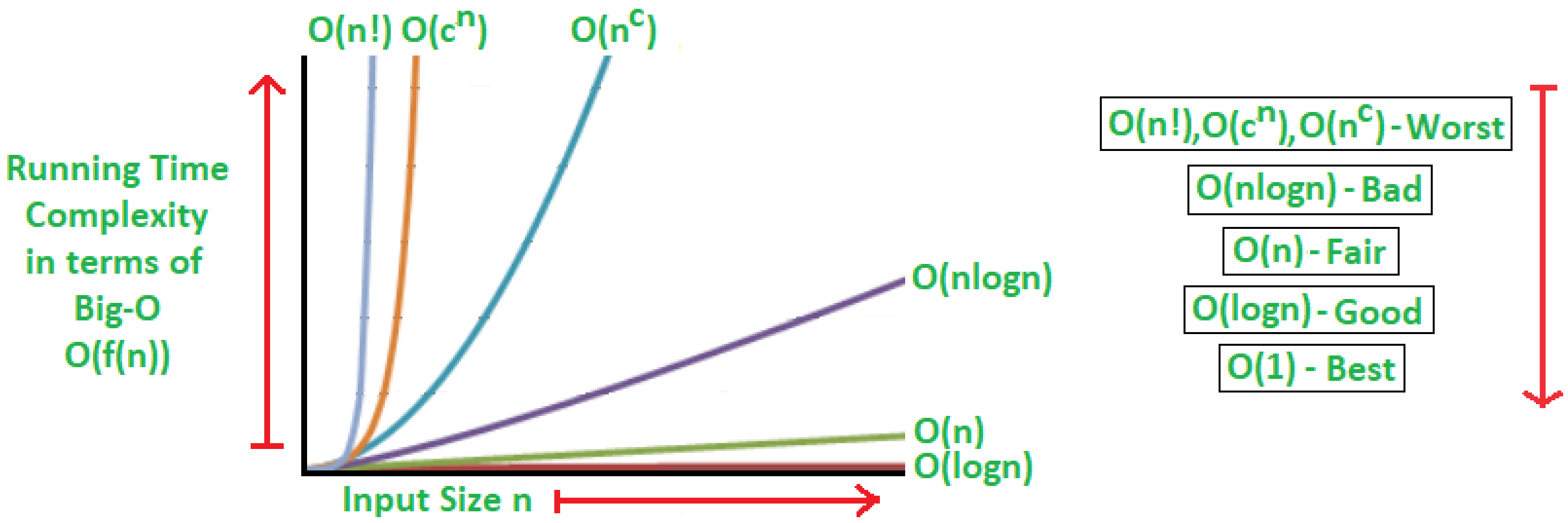
- running time of an algorithm is proportional to the logarithm of the input size

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```

Common Big-O notations

Other Big-O Complexity

- Cubic Time Complexity: Big $O(n^3)$ Complexity
- Polynomial Time Complexity: Big $O(n^k)$ Complexity
- Exponential Time Complexity: Big $O(2^n)$ Complexity
- Factorial Time Complexity: Big $O(n!)$ Complexity



Sorting Algorithms – $Bubble\ Sort$

Bubble Sort | Runtime $\boldsymbol{O(n^2)}$

- We start at the beginning of the array and swap the first two elements if the first is greater than the second
- We go to the next pair, continuously making sweeps of the array until it is sorted
- The smaller items slowly "bubble" up to the beginning of the list

Sorting Algorithms – *Bubble Sort*

Consider the following array arr[] = {5, 1, 4, 2, 8}

First Pass: *Bubble sort starts with very first two elements, comparing them to check which one is greater.*
*(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and **swaps** since 5 > 1.*
*(1 **5 4** 2 8) –> (1 **4 5** 2 8), **Swap** since 5 > 4*
*(1 4 **5 2** 8) –> (1 4 **2 5** 8), **Swap** since 5 > 2*
*(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.*

Second Pass: *Now, during second iteration it should look like this:*

*(**1 4** 2 5 8) –> (**1 4** 2 5 8)*
*(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2*
*(1 2 **4 5** 8) –> (1 2 **4 5** 8)*
*(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

***Third Pass**: Now, the array is already sorted, but our algorithm does not know if it is completed.*
The algorithm needs one whole pass without any swap to know it is sorted.

*(**1 2** 4 5 8) –> (**1 2** 4 5 8)*
*(1 **2 4** 5 8) –> (1 **2 4** 5 8)*
*(1 2 **4 5** 8) –> (1 2 **4 5** 8)*
*(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

Sorting Algorithms – *Bubble Sort*

```c
#include <stdio.h>
#define SIZE 10

// function main begins program execution
int main(void) {
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   puts("Data items in original order");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```c
   // bubble sort
   // loop to control number of passes
   for (int pass = 1; pass < SIZE; ++pass) {
      // loop to control number of comparisons per pass
      for (size_t i = 0; i < SIZE - 1; ++i) {
         // compare adjacent elements and swap them if first
         // element is greater than second element
         if (a[i] > a[i + 1]) {
            int hold = a[i];
            a[i] = a[i + 1];
            a[i + 1] = hold;
         }
      }
   }
```

```c
   // output sorted array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

   puts("");
}
```

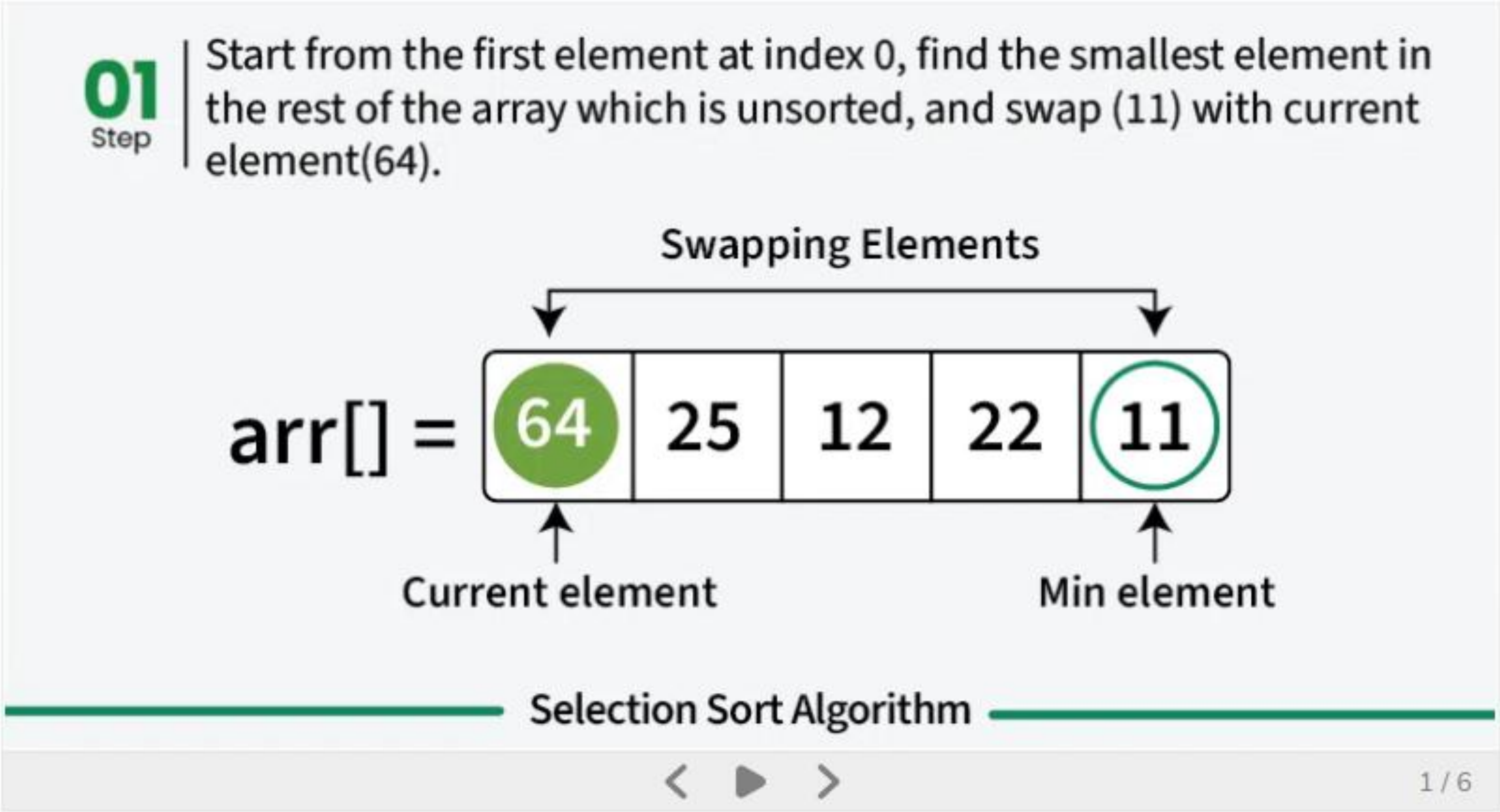
Sorting Algorithms – $Selection\ Sort$

Selection Sort | Runtime $O(n^2)$

- Find the smallest element using a linear scan and move it to the front
- Find the second smallest and move it, again doing a linear scan
- Continue doing this until all the elements are in place

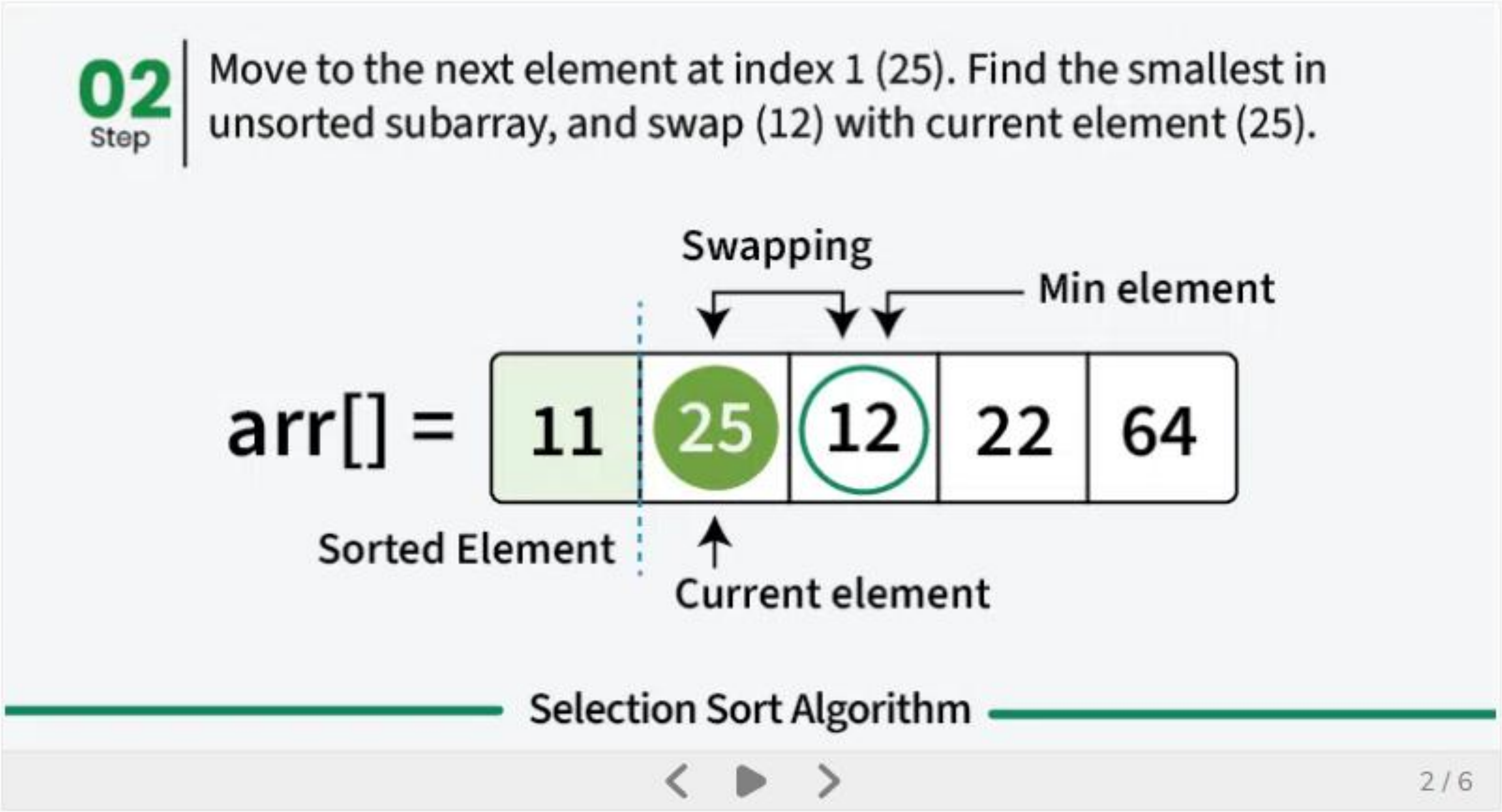
Sorting Algorithms – *Selection Sort*

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}



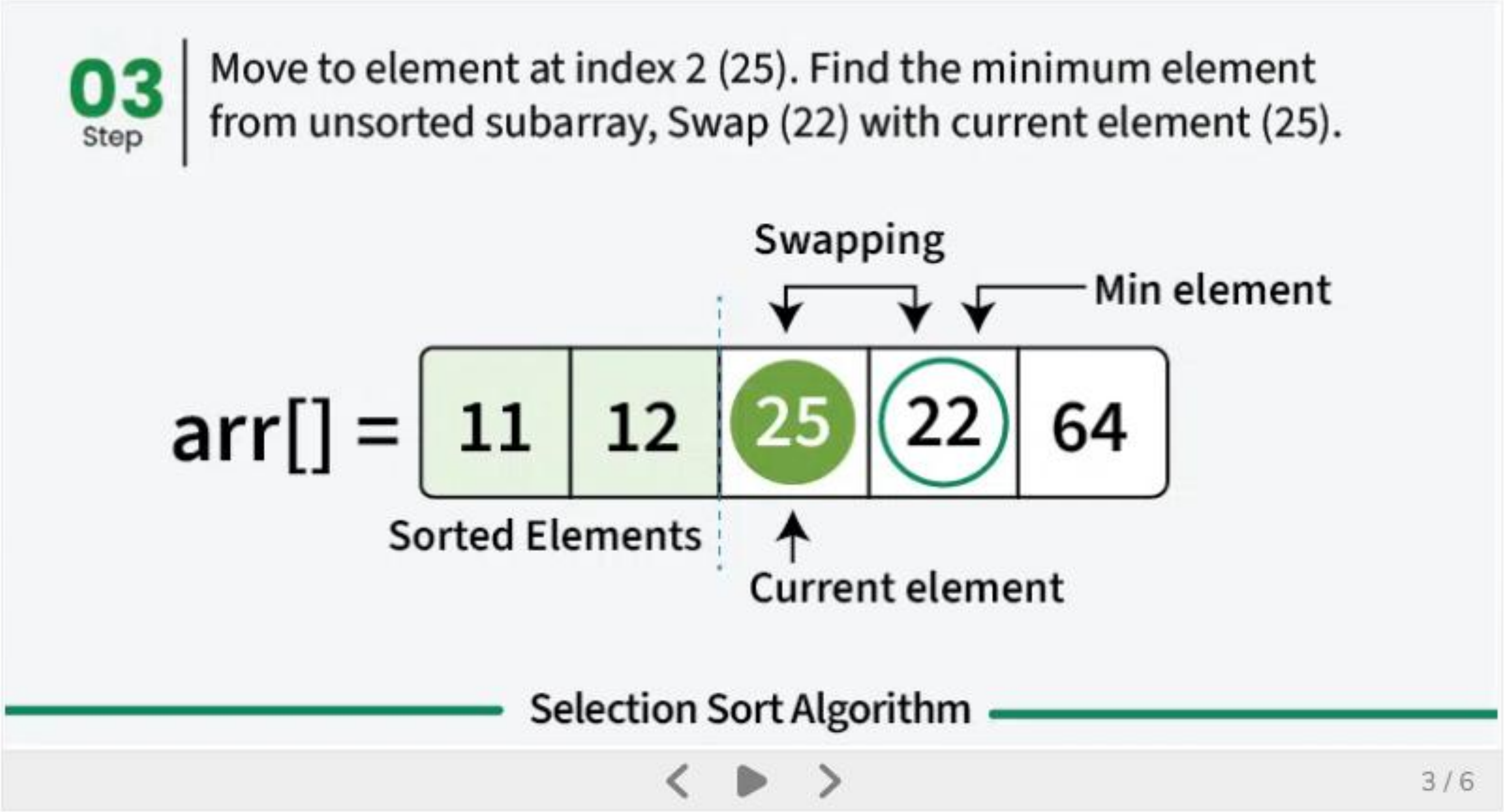
Sorting Algorithms – *Selection Sort*

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}



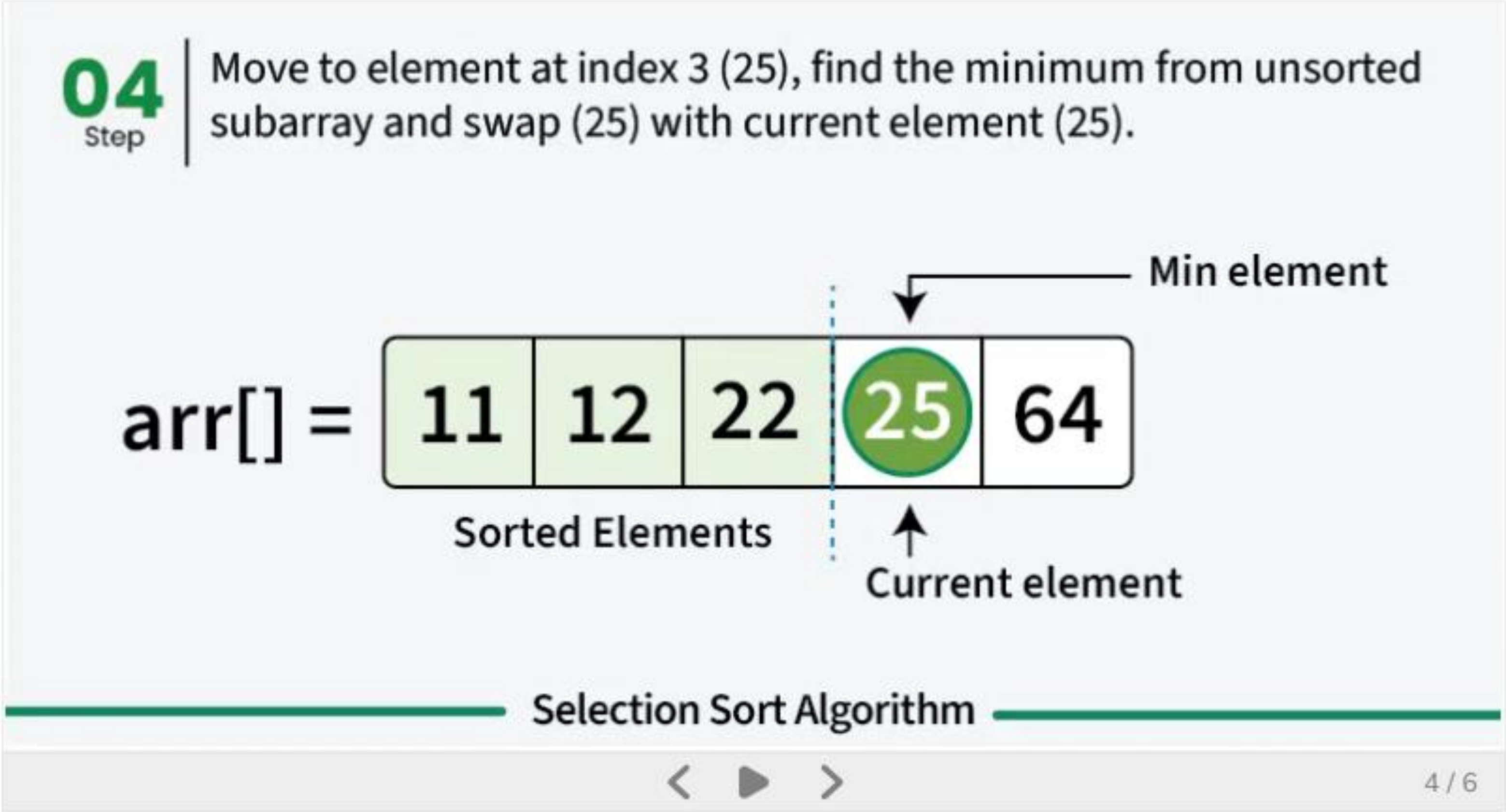
Sorting Algorithms – *Selection Sort*

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}



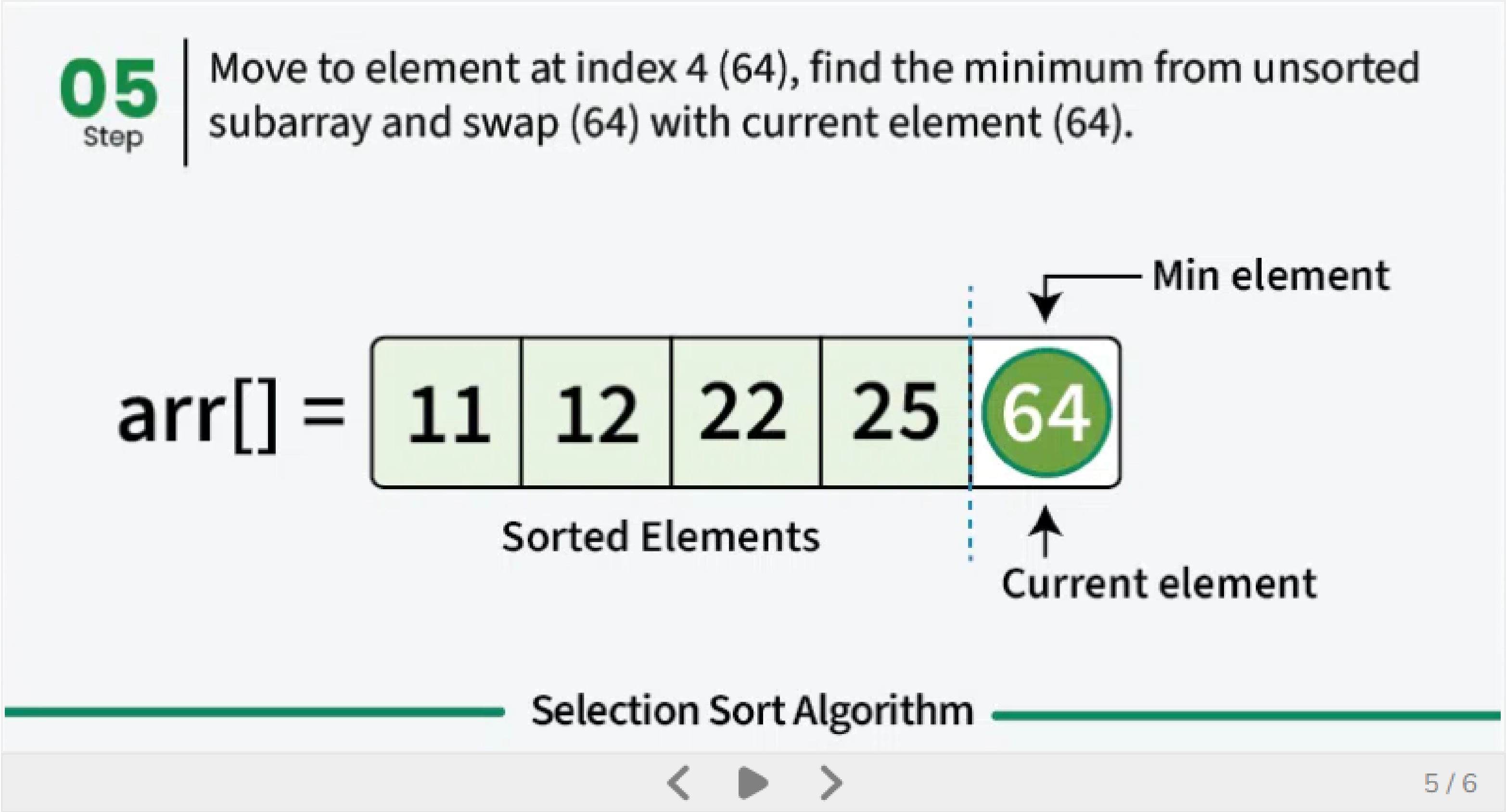
Sorting Algorithms – *Selection Sort*

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}



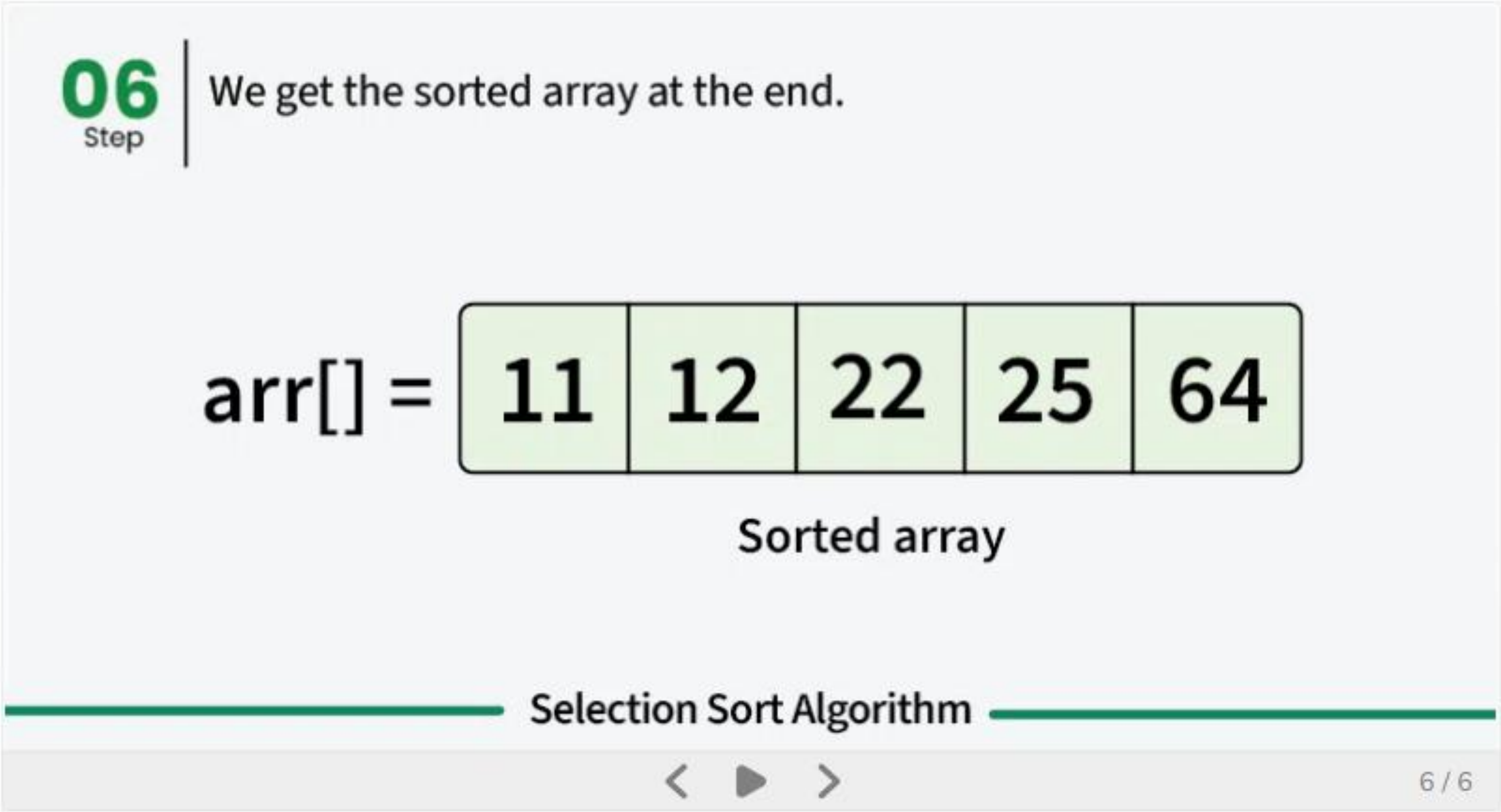
Sorting Algorithms – *Selection Sort*

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}



Sorting Algorithms – *Selection Sort*

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}



Sorting Algorithms – *Selection Sort*

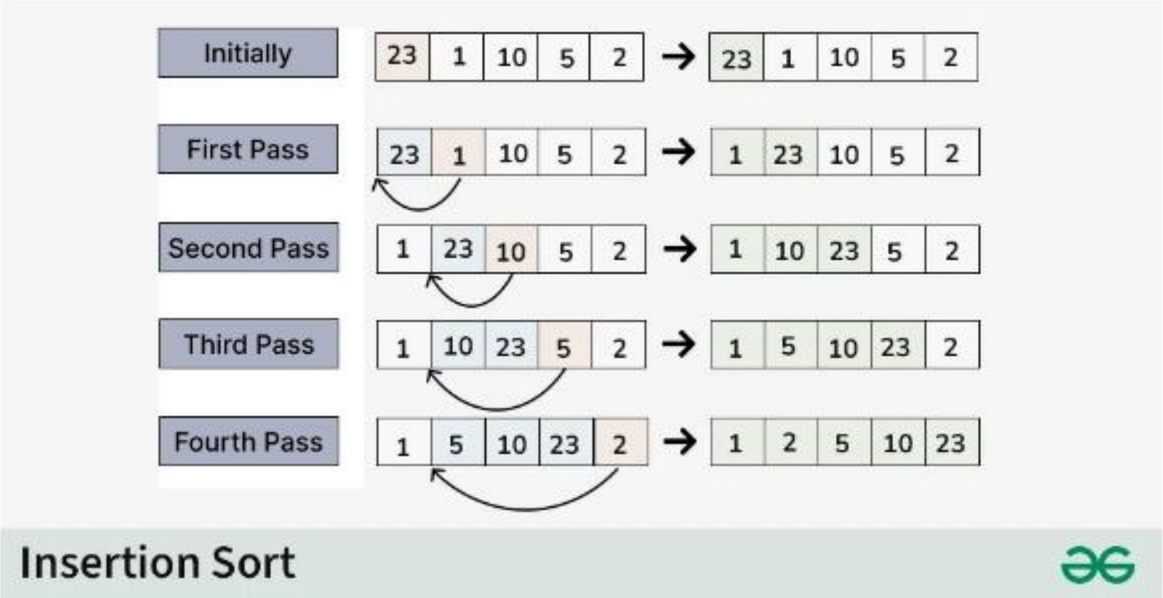
```c
// C program for implementation of selection sort
#include <stdio.h>

void printArray(int arr[], int n);
void selectionSort(int arr[], int n);

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

```c
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {

        // Assume the current position holds
        // the minimum element
        int min_idx = i;

        // Iterate through the unsorted portion
        // to find the actual minimum
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {

                // Update min_idx if a smaller element is found
                min_idx = j;
            }
        }

        // Move minimum element to its
        // correct position
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Sorting Algorithms – *Insertion Sort*

Insertion Sort | Runtime $O(n^2)$

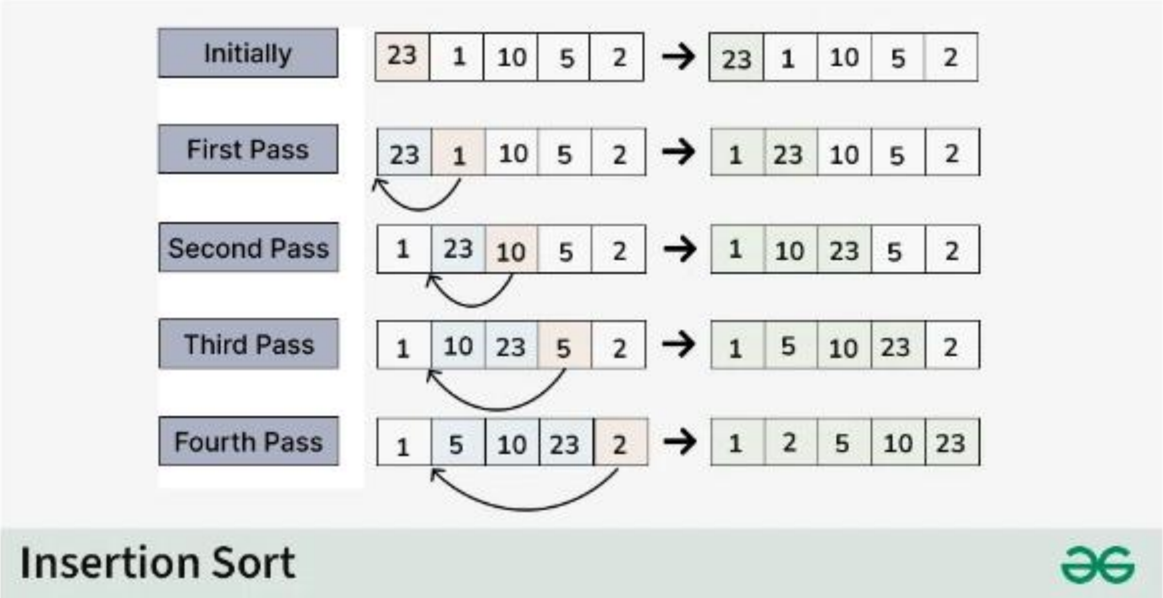
- simple but inefficient sorting algorithm
- iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list
- It is like sorting playing cards in your hands



Sorting Algorithms – *Insertion Sort*

Insertion Sort | Runtime $O(n^2)$

- We start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.



Sorting Algorithms – *Insertion Sort*

Lets consider the following array as an example: arr[] = *{23, 1, 10, 5, 2}*

- ***Initial:***
  - *Current element is **23***
  - *The first element in the array is assumed to be sorted.*
  - *The sorted part until **0th** index is : **[23]***
- ***First Pass:***
  - *Compare **1** with **23** (current element with the sorted part).*
  - *Since **1** is smaller, insert **1** before **23** .*
  - *The sorted part until **1st** index is: **[1, 23]***
- ***Second Pass:***
  - *Compare **10** with **1** and **23** (current element with the sorted part).*
  - *Since **10** is greater than **1** and smaller than **23** , insert **10** between **1** and **23** .*
  - *The sorted part until **2nd** index is: **[1, 10, 23]***

Sorting Algorithms – *Insertion Sort*

Lets consider the following array as an example: arr[] = *{23, 1, 10, 5, 2}*

- ***Third Pass:***
  - *Compare **5** with **1** , **10** , and **23** (current element with the sorted part).*
  - *Since **5** is greater than **1** and smaller than **10** , insert **5** between **1** and **10***
  - *The sorted part until **3rd** index is : **[1, 5, 10, 23]***
- ***Fourth Pass:***
  - *Compare **2** with **1, 5, 10** , and **23** (current element with the sorted part).*
  - *Since **2** is greater than **1** and smaller than **5** insert **2** between **1** and **5** .*
  - *The sorted part until **4th** index is: **[1, 2, 5, 10, 23]***
- ***Final Array:***
  - *The sorted array is: **[1, 2, 5, 10, 23]***

Sorting Algorithms – *Insertion Sort*

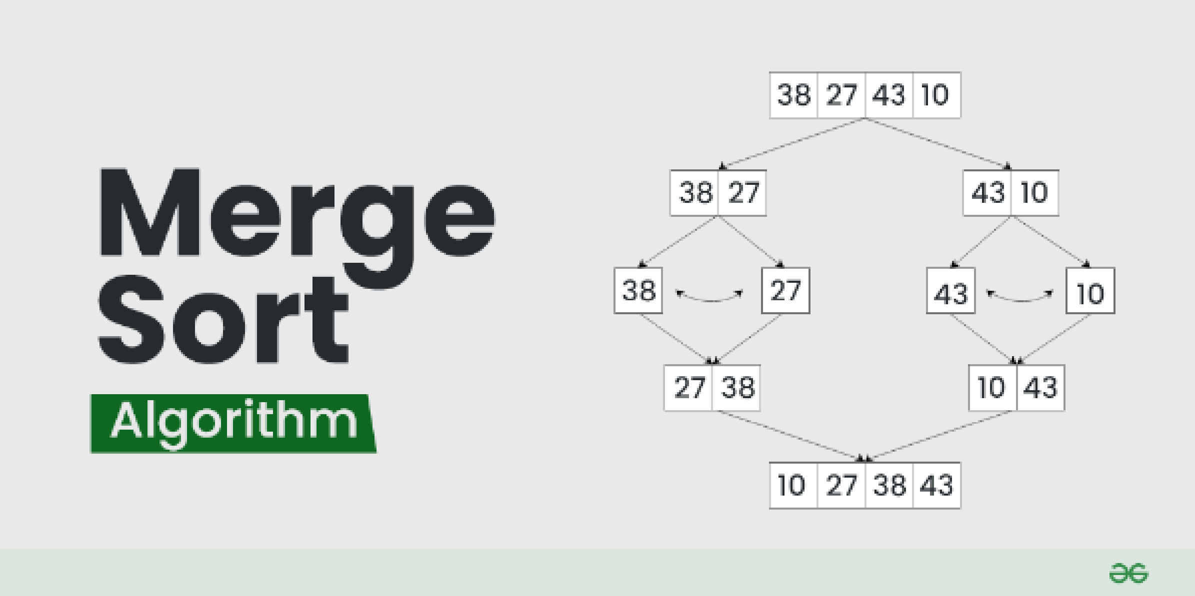
```c
// C program for implementation of Insertion Sort
#include <stdio.h>

void insertionSort(int arr[], int n);
void printArray(int arr[], int n);

int main()
{
    int arr[] = {123, 1, 10, 5, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

```c
/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}
```

Sorting Algorithms – *Merge Sort*

Merge Sort | Runtime $O(n\ log(n))$

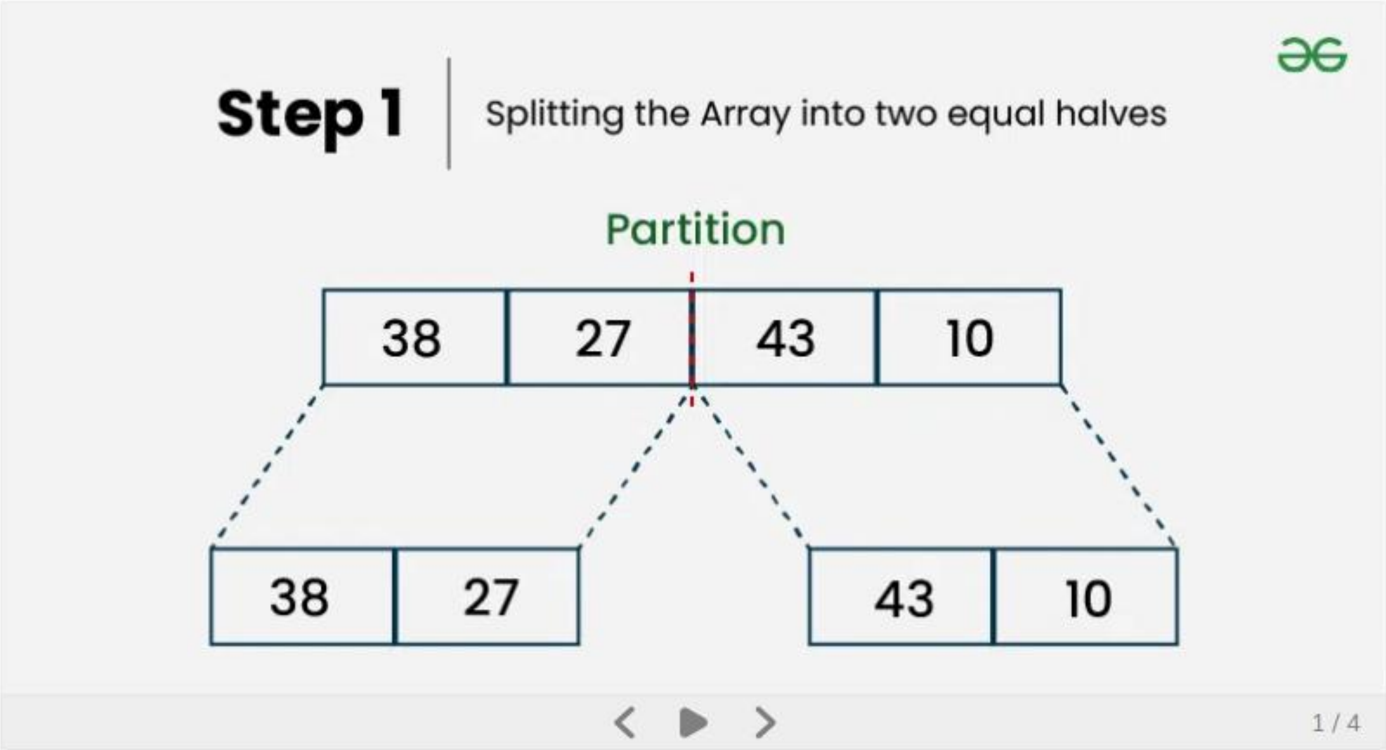
- Efficient but conceptually more complex than selection sort and insertion sort
- Merge sort divides the array in half, sorts each of those halves, and then merges them back together
- Each of those halves has the same sorting algorithm applied to it



Sorting Algorithms – *Merge Sort*

Let's sort the array or list [38, 27, 43, 10] using Merge Sort

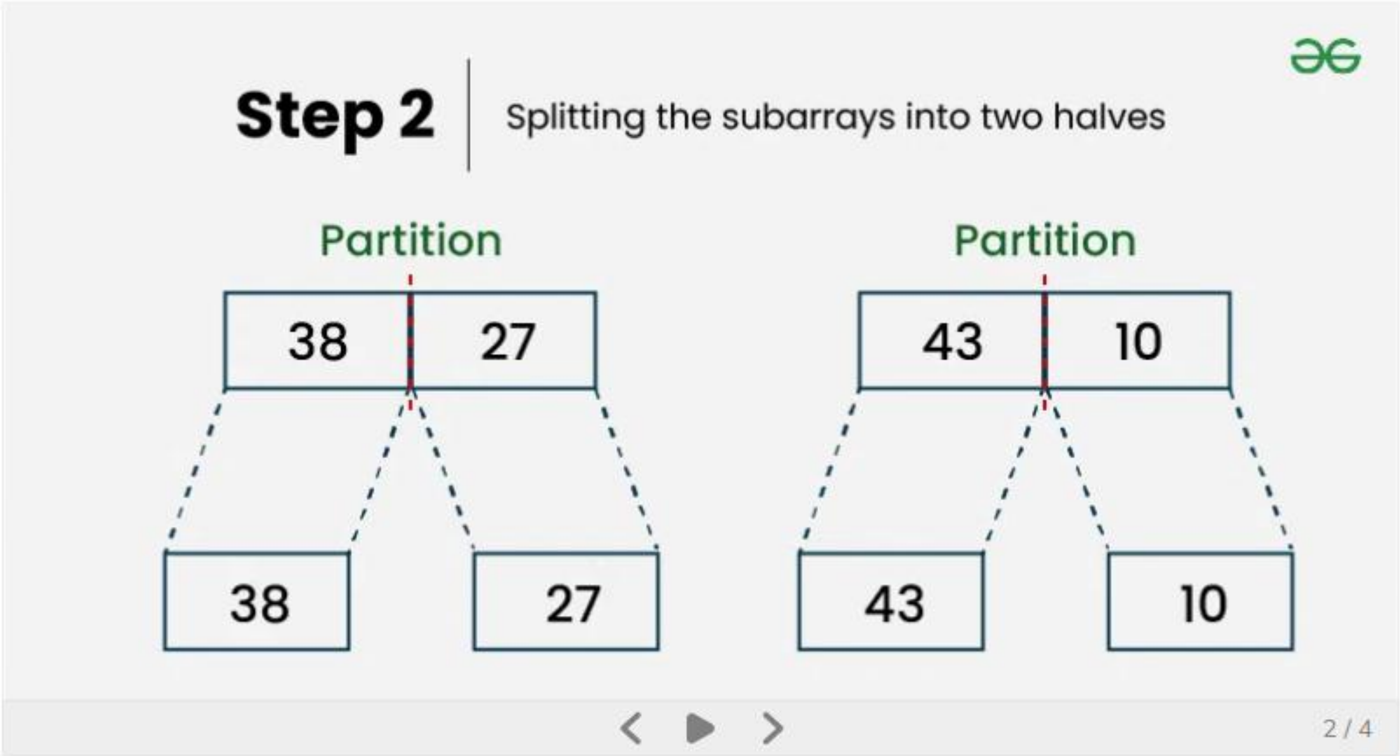
- ***Divide:***
  - ***38, 27, 43, 10]*** *is divided into* ***[38, 27*** *] and* ***[43, 10]*** *.*
  - ***[38, 27]*** *is divided into* ***[38]*** *and* ***[27]*** *.*
  - ***[43, 10]*** *is divided into* ***[43]*** *and* ***[10]*** *.*



Sorting Algorithms – *Merge Sort*

Let's sort the array or list [38, 27, 43, 10] using Merge Sort

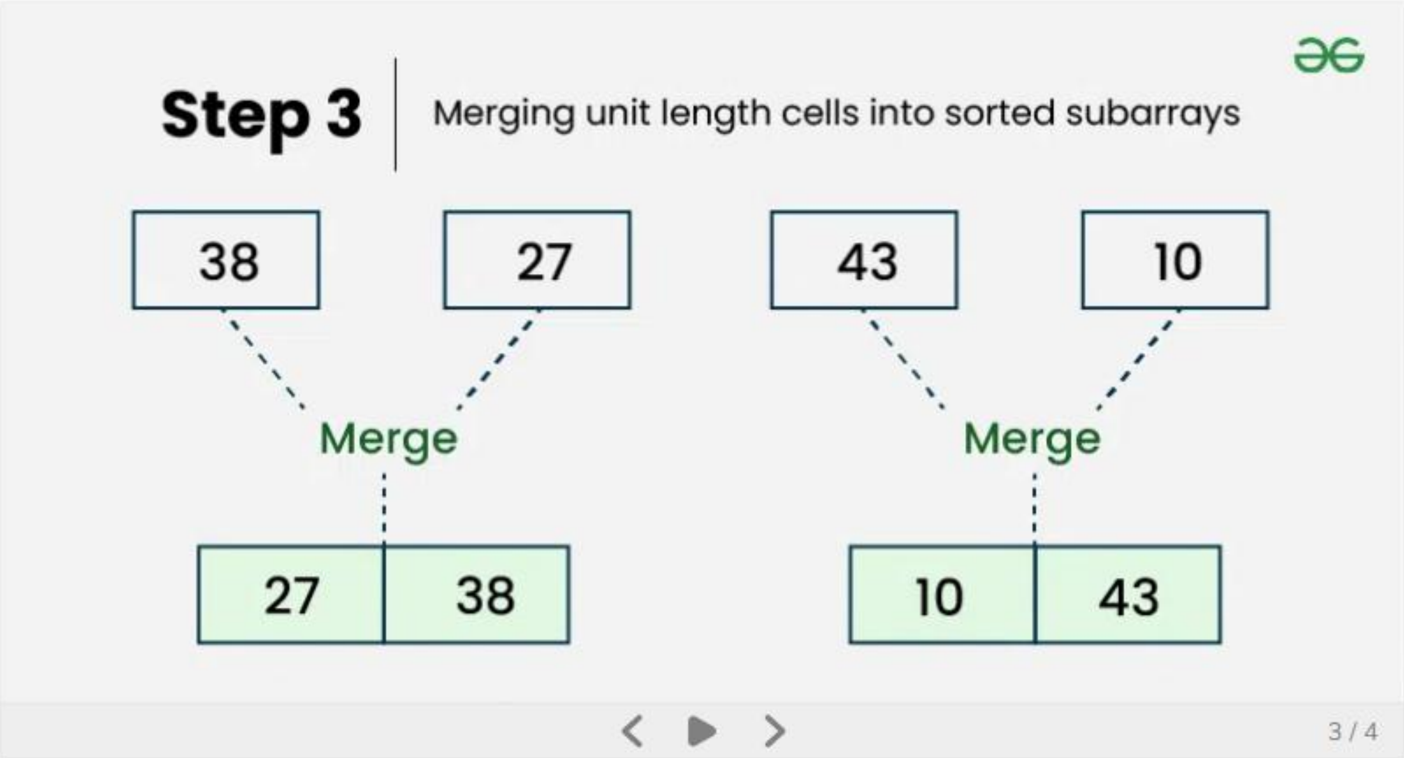
- ***Conquer:***
  - *[38] is already sorted.*
  - *[27] is already sorted.*
  - *[43] is already sorted.*
  - *[10] is already sorted.*



Sorting Algorithms – *Merge Sort*

Let's sort the array or list [38, 27, 43, 10] using Merge Sort

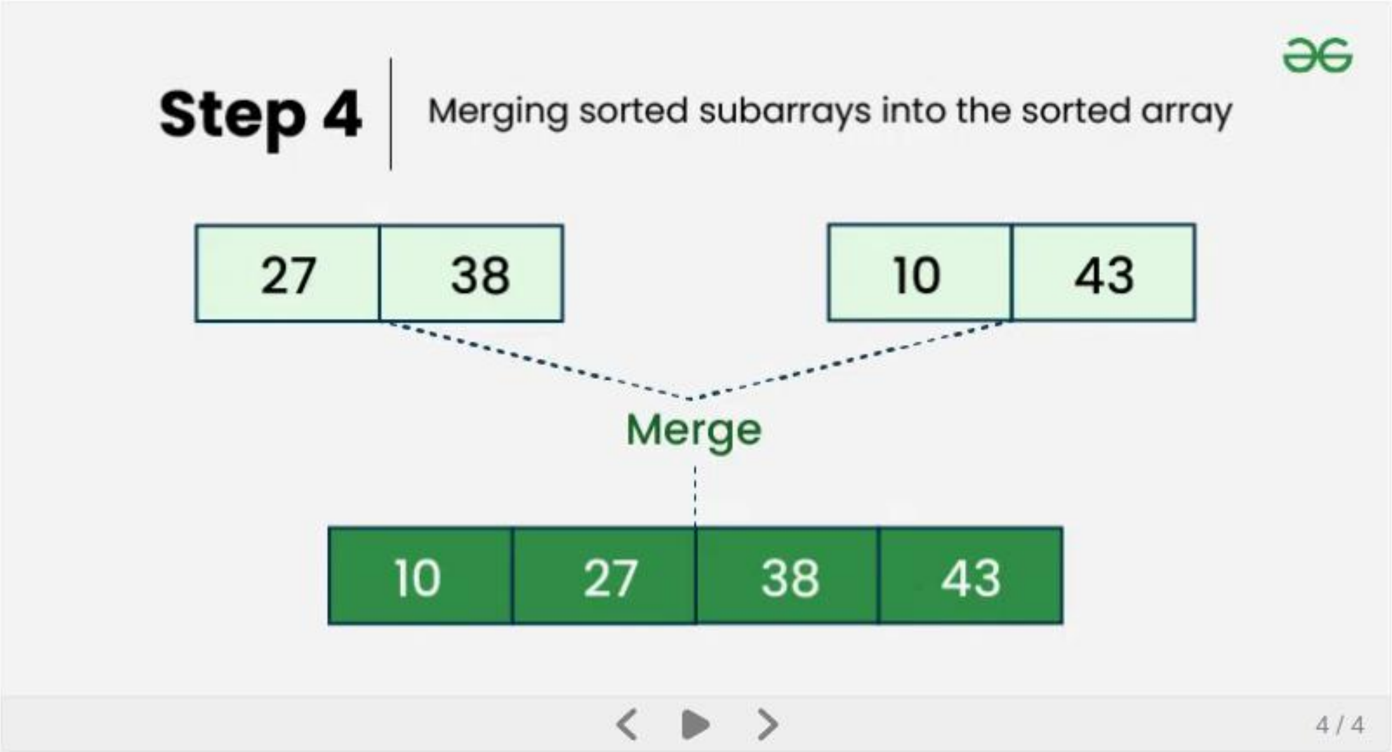
- ***Merge:***
  - *Merge **[38]** and **[27]** to get **[27, 38]** .*
  - *Merge **[43]** and **[10]** to get **[10,43]** .*
  - *Merge **[27, 38]** and **[10,43]** to get the final sorted list **[10, 27, 38, 43]***



Sorting Algorithms – *Merge Sort*

Let's sort the array or list [38, 27, 43, 10] using Merge Sort

- *Therefore, the sorted list is* ***[10, 27, 38, 43]***



Sorting Algorithms – $Merge\ Sort$

```c
// C program for the implementation of merge sort
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]

void mergeSort(int arr[], int left, int right)
void merge(int arr[], int left, int mid, int right)

int main() {
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

      // Sorting arr using mergesort
    mergeSort(arr, 0, n - 1);

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}

// The subarray to be sorted is in the index range [left-right]
void mergeSort(int arr[], int left, int right) {
    if (left < right) {

        // Calculate the midpoint
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```

```c
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int leftArr[n1], rightArr[n2];

    // Copy data to temporary arrays
    for (i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        rightArr[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[left..right]
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        }
        else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of leftArr[], if any
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    // Copy the remaining elements of rightArr[], if any
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```