DATA STRUCTURES & ALGORITHMS

GAILY REY APRIL M. GUZON

Searching and Sorting Algorithms

Introduction

- In this unit, you will learn some common processes that arise in computing: searching and sorting.
- Some common processes involve finding a particular item in a given list of items called searching. Another involves ordering items in a particular way of some given data called sorting.
- The two represent a common application in computers where data must be searched and retrieved.
- An example of an everyday application of this is in a telephone directory. It stores information such as the names, addresses and phone numbers of its customers. As the amount of information to be stored and accessed becomes very large, the computer comes in handy to assist in this task. To access the stored data, searching and sorting are carried out.

TOPIC LEARNING OUTCOMES

Upon completion of this unit, you should be able to:

- describe the process of searching and sorting
- design algorithms for searching and sorting
- implement algorithms for searching and sorting

Key Terms

- **Searching:** Is the algorithmic finding of a particular item in a collection of given items
- **Sorting:** is the ordering of items in a given list
- **Insertion:** Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.
- **Deletion:** It is a process of deleting a particular element from a data structure.

Searching Algorithm

- This topic involves studying the searching process and how to write algorithms that can find particular given item from a list of given items. The activity of searching will dedicate itself to providing the answer of a presence or no presence of the searched item.
- In computer science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items.

Linear/Sequential Search

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

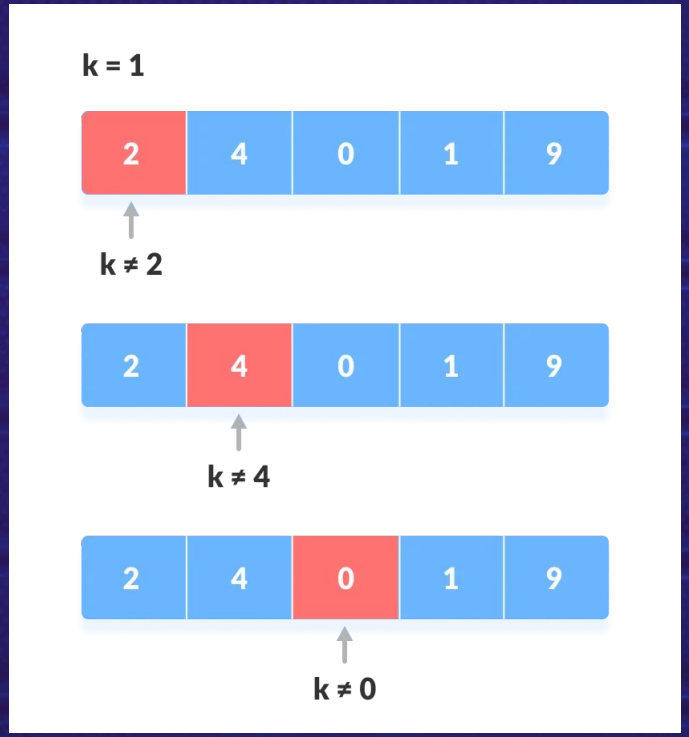
How Linear Search Works?

The following steps are followed to search for an element **k=1** in the list below.

Array to be searched for

How Linear Search Works

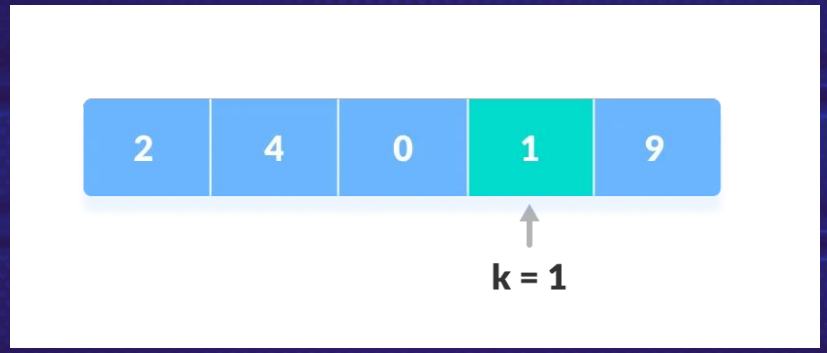
1. Start from the first element, compare with each element x.



Compare with each element

How Linear Search Works

2. If x == k, return the index.



Element Found

In this example, element x == k which is 1. Hence, k is present at index=3

3. Else, return not found

Example 1:

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

Find k = 110;

Output : 6

Element k is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

Find k = 175;

Output : -1

Element x is not found in arr[].

Linear Search Algorithm

```
LinearSearch(array, key)
  for each item in the array
    if item == value
      return its index
```

Linear Search Applications

- For searching operations in smaller arrays (<100 items).
- Linear search can be applied to both single-dimensional and multi-dimensional arrays.
- Linear search is easy to implement and effective when the array contains only a few elements.
- Linear Search is also efficient when the search is performed to fetch a single search in an unordered-List.

Binary Search

- Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array.
- Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

How Binary Search Works?

Binary Search Algorithm can be implemented in two ways which are discussed below.

1.Iterative Method

2. Recursive Method

Recursive Method

The recursive method follows the divide and conquer approach. The general steps for both methods are discussed below

1. The array in which searching is to be performed is:

Initial array

Let **x=4** be the element to be searched.

2. Set two pointers low and high at the lowest and the highest positions respectively.

Setting pointers

3. Find the middle element mid of the array: **(index of low + index of high)/2**
Using the formula above, we get (0 + 6)/2 = 3; hence index 3. Therefore, the mid = 6.

Mid element

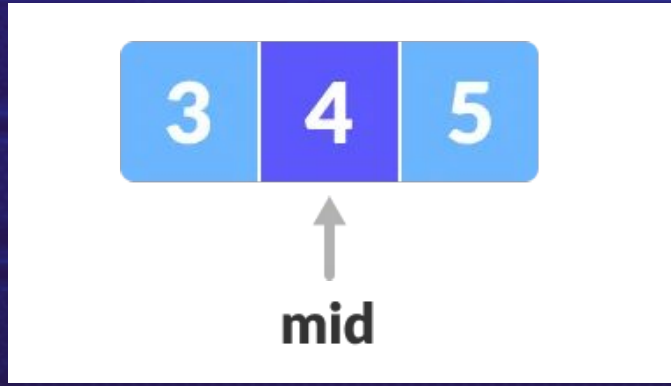
4. If x == mid, then return mid. Else, compare the element to be searched with m.
5. If x > mid, compare x by setting with the middle element of the elements on the right side of mid. This is done by setting low to low = mid + 1.

Recursive Method

6. Else, compare setting x with the middle element of the elements on the left side of mid. This is done by setting high to high = mid - 1.

Finding mid element

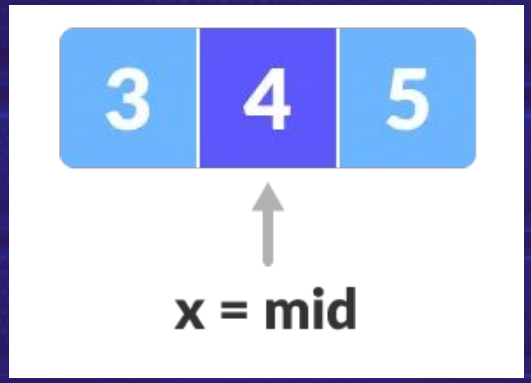
7. Repeat steps 3 to 6 until low meets high.



Mid element

Recursive Method

8. x=4 is found.



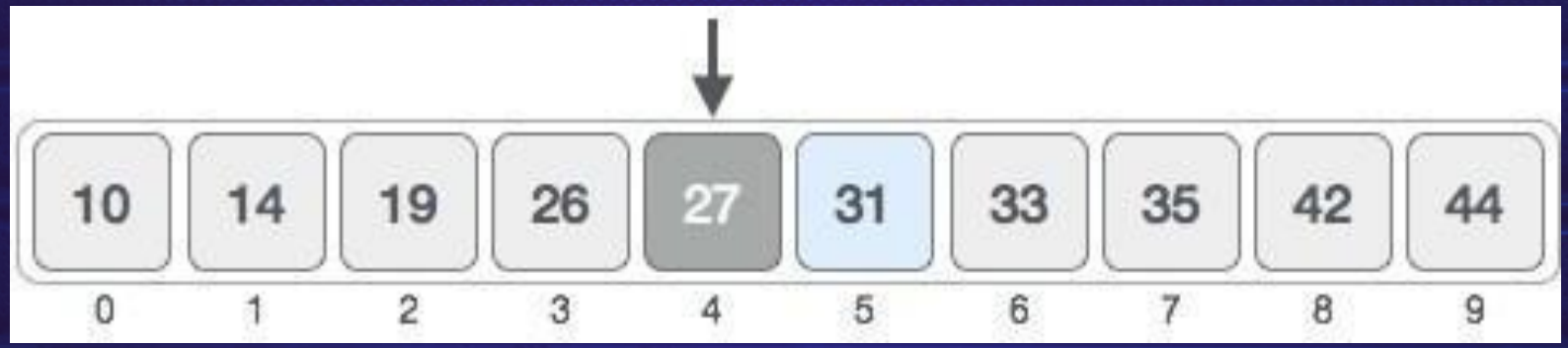
Found

Example: Search the location of value 31 using binary search.

Using the formula to find mid, we get 0 + 9/ 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.

Example: Search the location of value 31 using binary search.

- Now we compare the value stored at location 4, with the value being searched, i.e., 31.
- The value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, we also know that the target value must be in the upper portion of the array.



Example: Search the location of value 31 using binary search.

- Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

- The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Example: Search the location of value 31 using binary search.

- We compare the value stored at location 5 with our target value. We find that it is a match.

- We conclude that the target value 31 is stored at location 5.

Binary Search Algorithm

ITERATION METHOD

```
do until the pointers low and high meet each other.
    mid = (low + high)/2
    if (x == arr[mid])
        return mid
    else if (x > A[mid]) // x is on the right side
        low = mid + 1
    else                          // x is on the left side
        high = mid - 1
```

Binary Search Algorithm

RECURSION METHOD

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x < data[mid]         // x is on the right side
            return binarySearch(arr, x, mid + 1, high)
        else                                 // x is on the right side
            return binarySearch(arr, x, low, mid - 1)
```

Binary Search Applications

1. Find an element in a sorted array
2. Applications of Binary Search beyond arrays
    a. To find if n is a square of an integer
    b. Find the first value greater than or equal to x in a given array of sorted integers
    c. Find the frequency of a given target value in an array of integers
    d. Find the peak of an array which increases and then decreases
    e. A sorted array is rotated n times. Search for a target value in the array.
3. Real life applications of Binary Search
    a. Dictionary
    b. Debugging a linear piece of code
    c. Figuring out resource requirements for a large system
    d. Find values in sorted collection
    e. Semiconductor test programs
    f. Numerical solutions to an equation

EXAMPLE

Suppose you have the following sorted list **[1, 5, 6, 8, 11, 12, 14, 15, 17, 18]**. Search for the index/location of value **8** using Binary Search.

QUIZ

Suppose you have the following sorted list **[6, 12, 16, 18, 21, 22, 24, 25, 27, 28, 32]**. Search for the index/location of value **28** using Binary Search.

Sorting

Sorting Algorithm Introduction

This topic involves studying the sorting process and how to write algorithms that can order particular items in a list to be in given order. The sorting activity ensures items in a given list are arranged in a desired order say ascending or descending. The section involves studying insertion sort, shell sort, quicksort algorithms.

The Sorting Algorithm

- A sorting algorithm is defined as an algorithm that puts elements of a list in a certain order. The most- used order is the alphabetical order, also known as **the lexicographical order**.
- Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists.
- Examples of sorting algorithms includes; insertion sort, shell sort and quick sort.
- Insertion sort is the simplest method, and does not require any additional storage.
- Shell sort is a simple modification that improves performance significantly.
- Quicksort is the most efficient and popular method, and is the method of choice for large arrays.

Insertion Sort Algorithm

- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted then, we select an unsorted card.
- If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.
- A similar approach is used by insertion sort.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

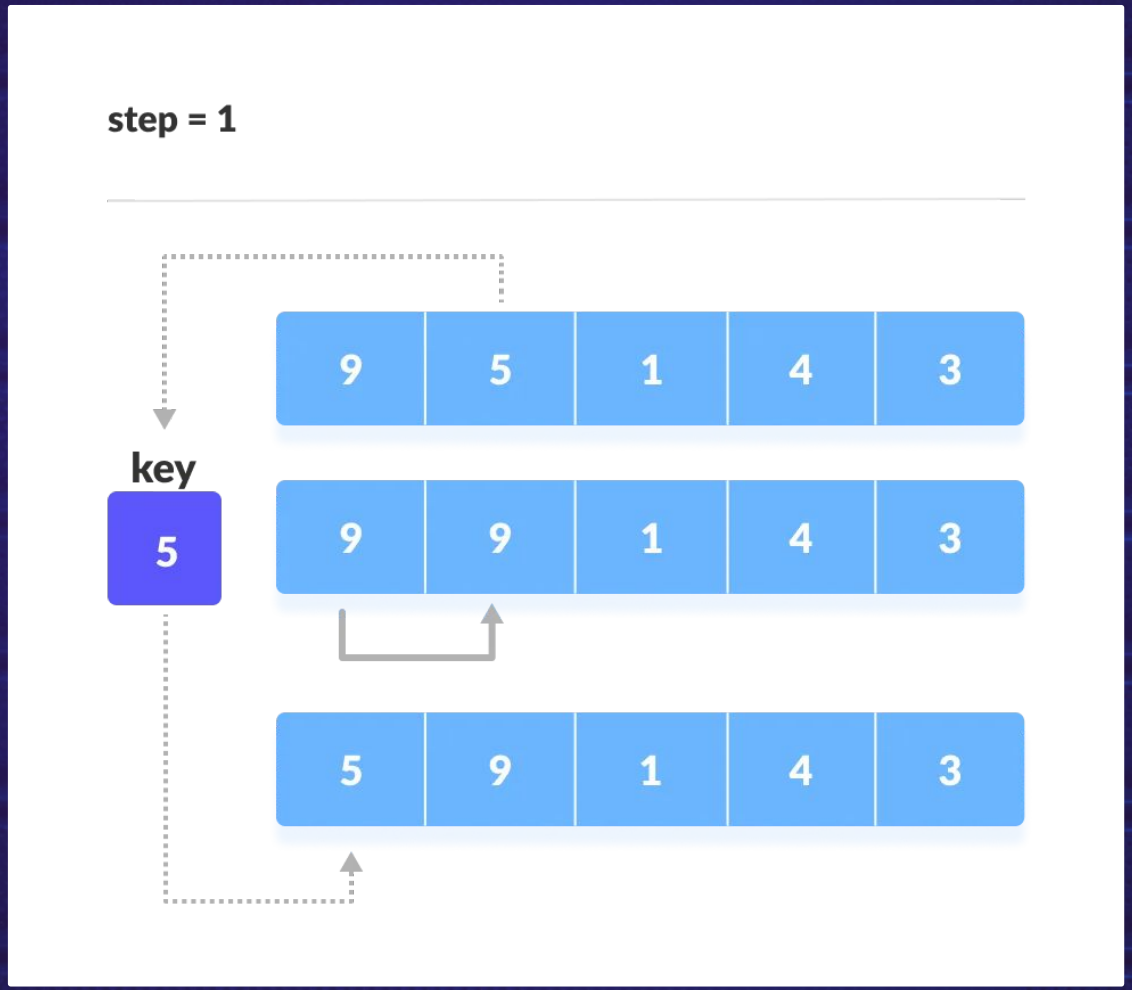
How Insertion Sort Works?

Suppose we need to sort the following array.

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

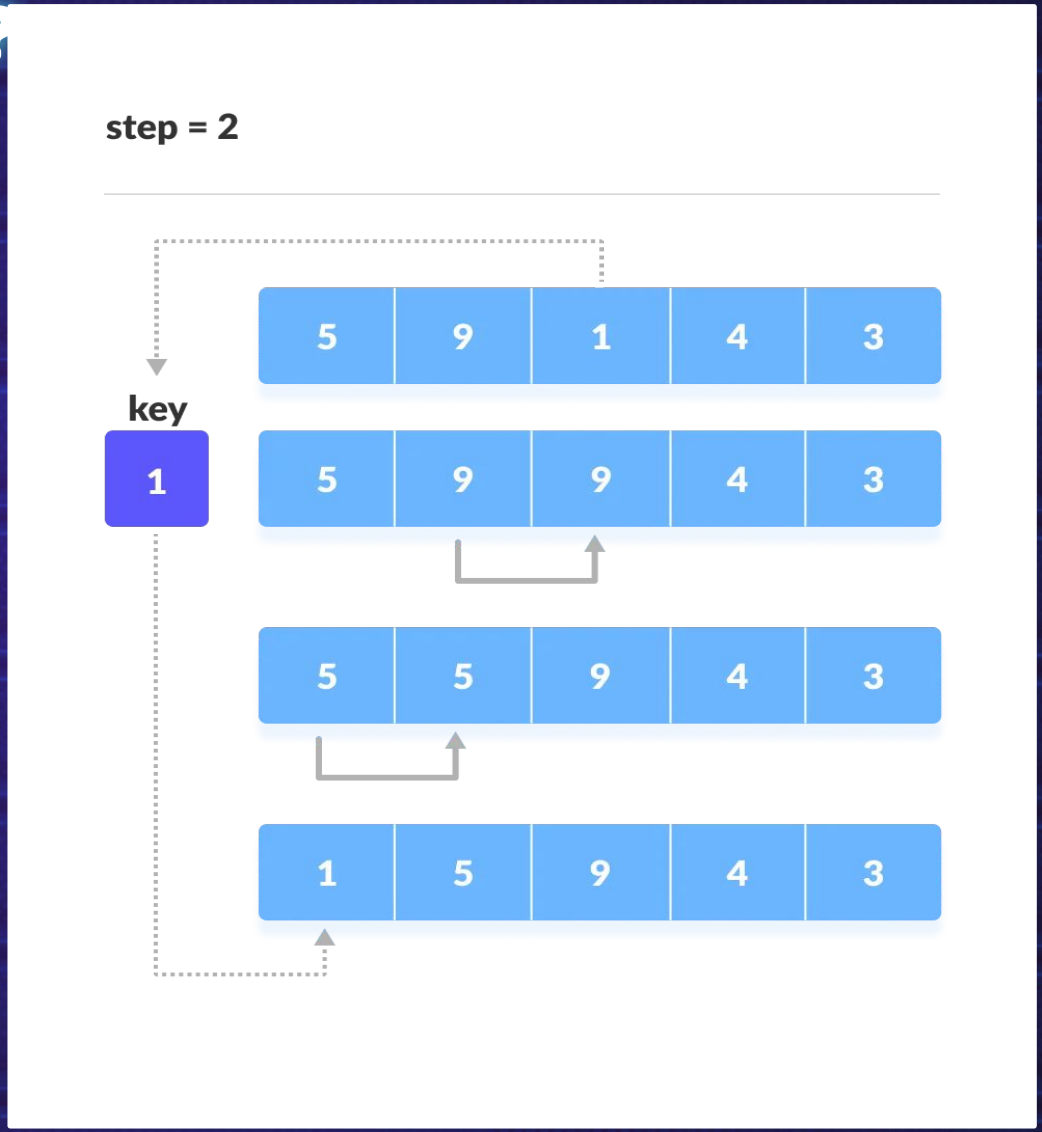
Compare key with the first element. If the first element is greater than key, then is placed in front of the first element.

How Insertion Sort Works?



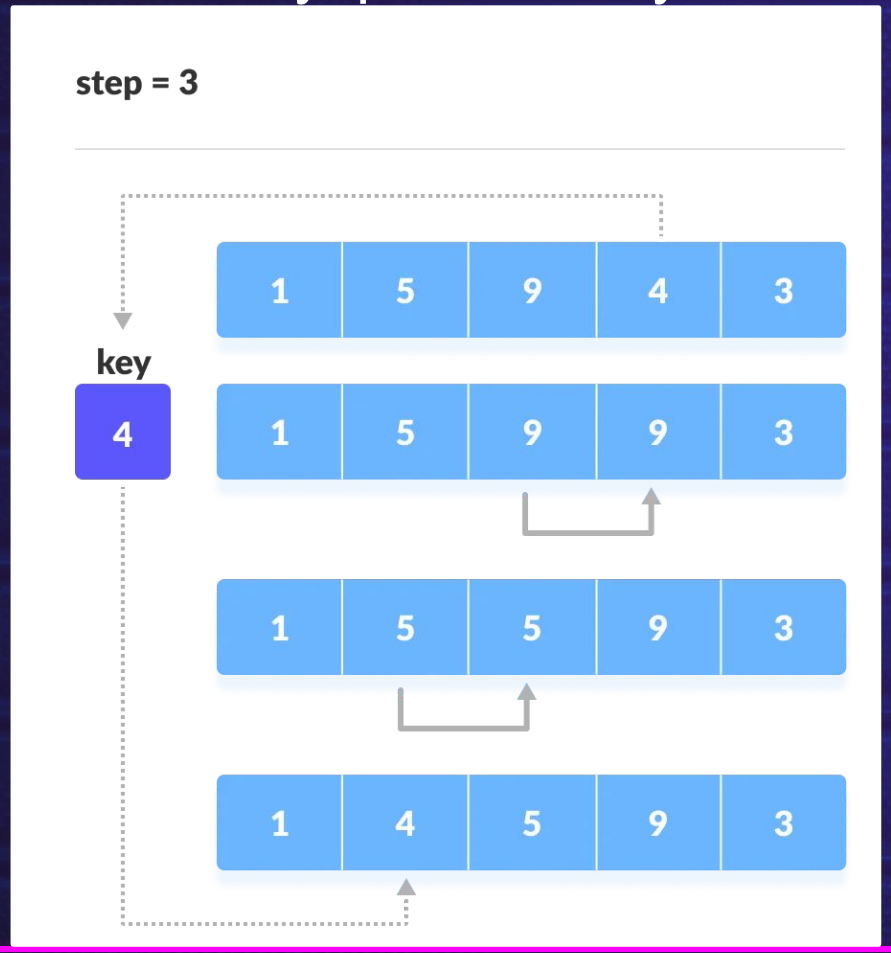
How Insertion Sort Works?

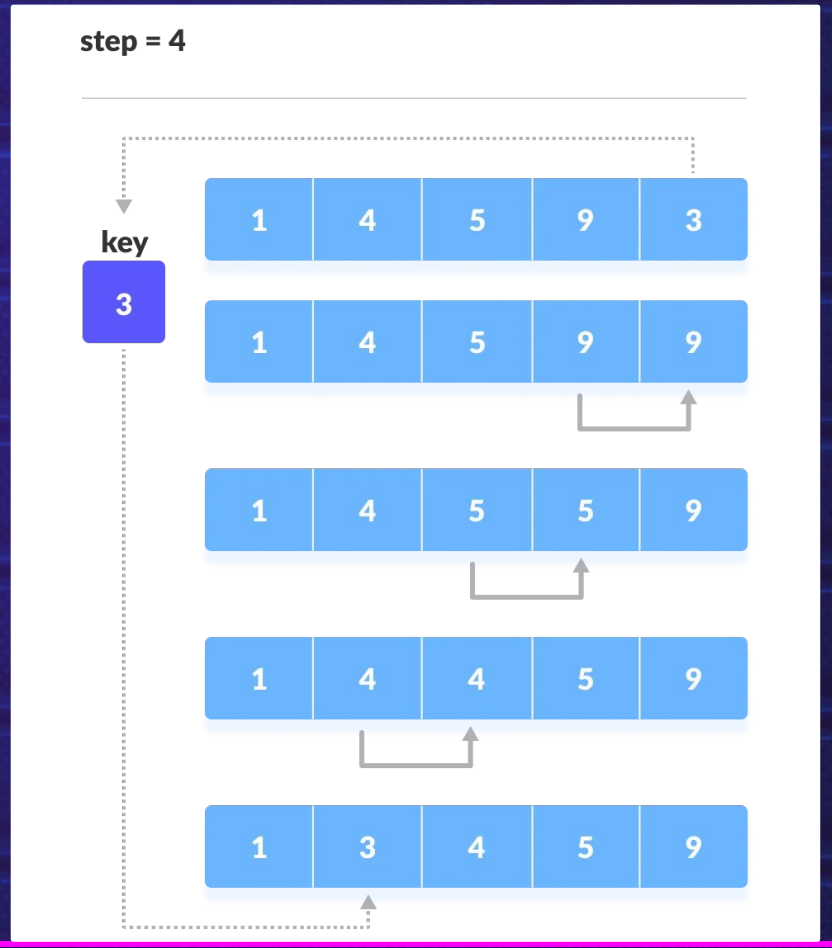
2. Now, the first two elements are sorted.
Take the third element and compare it with the elements on its left. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



How Insertion Sort Works?

3. Similarly, place every unsorted element at its correct position.





Another Example: Sort the given elements [12, 11, 13, 5, 6]

1. Let us loop for i = 1 (second element of the array) to 4 (last element of the array) i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
[11, 12, 13, 5, 6]

2. i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
[11, 12, 13, 5, 6]

3. i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
[5, 11, 12, 13, 6]

4. i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
[5, 6, 11, 12, 13]

Insertion Sort Algorithm

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

Insertion Sort Applications

The insertion sort is used when:

- If the cost of comparisons exceeds the cost of swaps, as is the case for example with string keys stored by reference or with human interaction, then using binary insertion sort may yield better performance.
- A variant named binary merge sort uses a binary insertion sort to sort groups of 32 elements, followed by a final sort using merge sort.

Bubble Sort Algorithm

Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

How Bubble Sort Works?

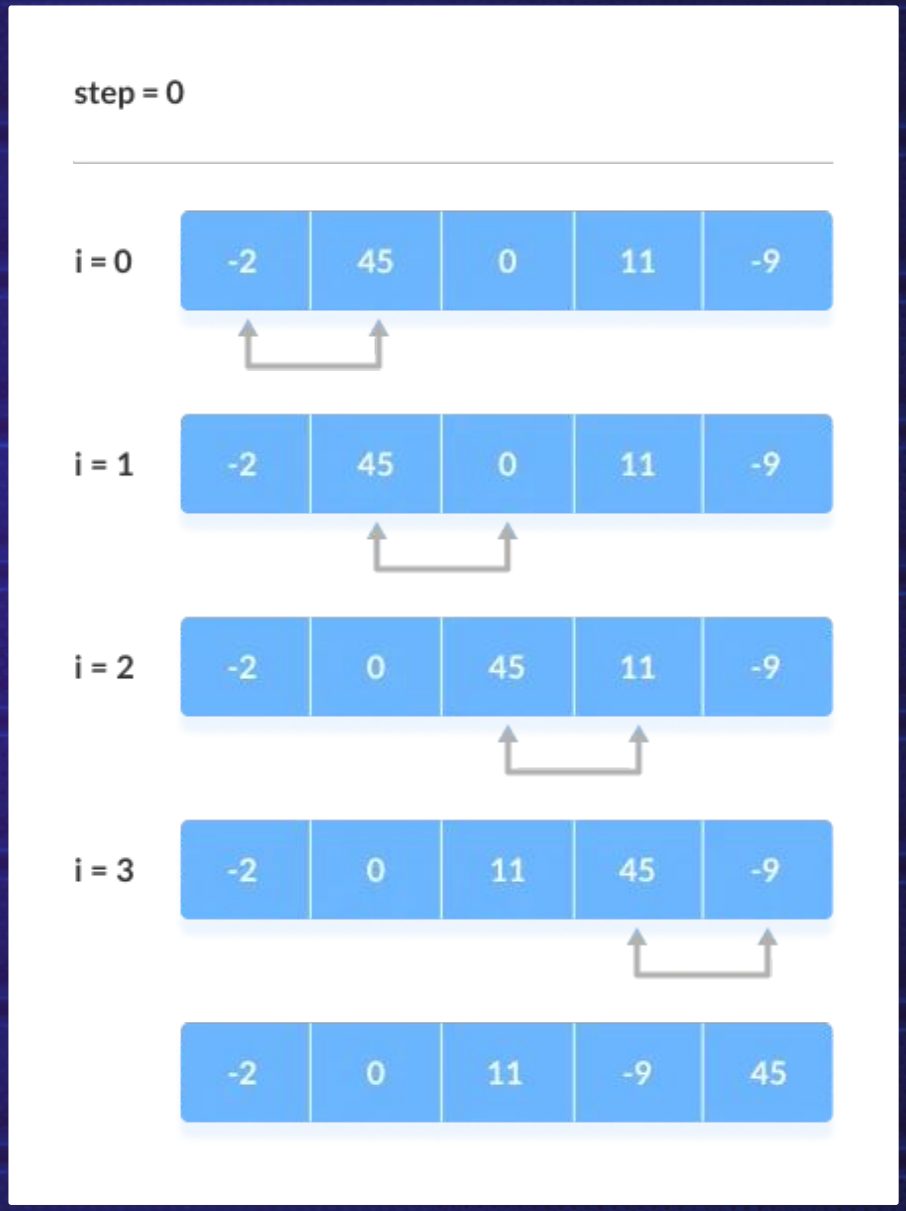
1. First Iteration (Compare and Swap)

i. Starting from the first index, compare the first and the second elements. I

ii. If the first element is greater than the second element, they are swapped.

iii. Now, compare the second and the third elements. Swap them if they are not in order.

iv. The above process goes on until the last element.



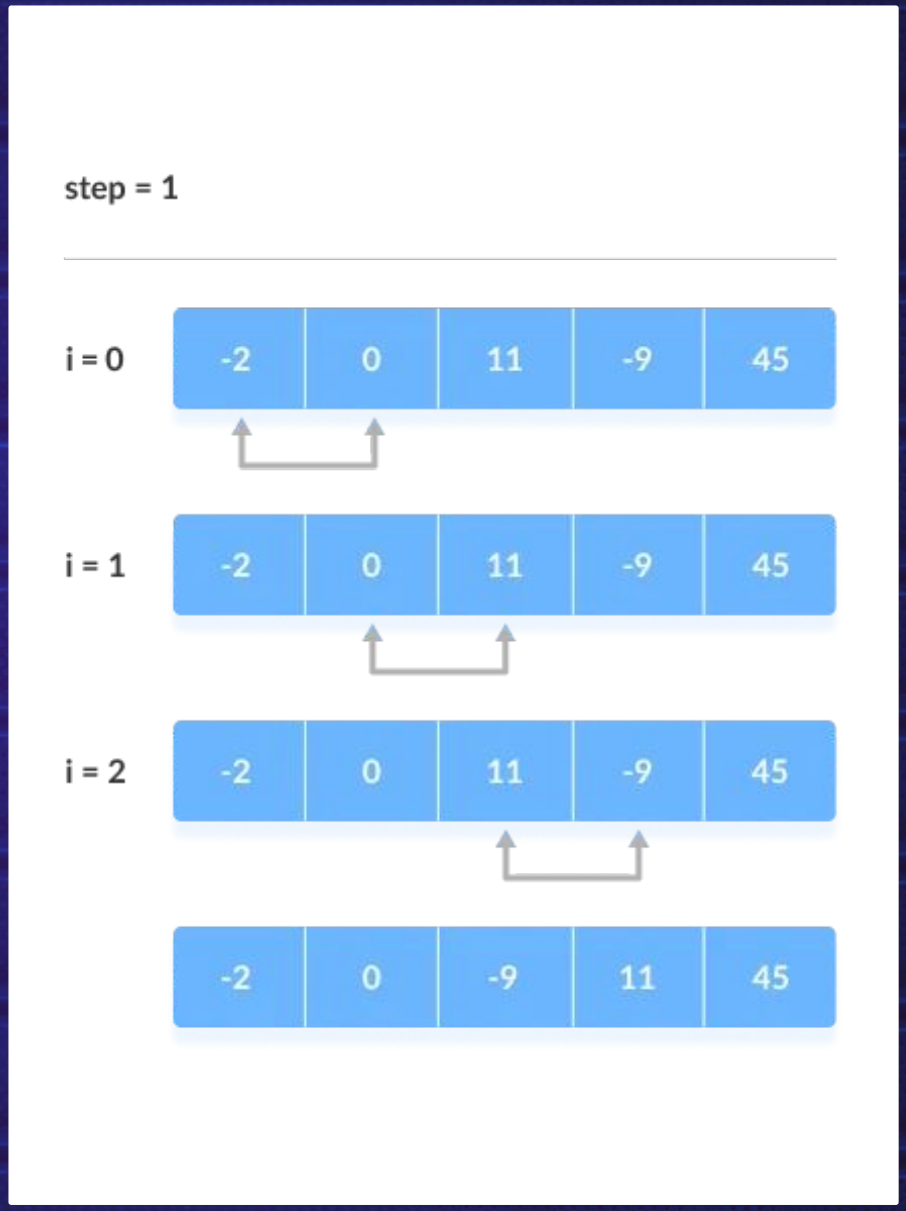
How Bubble Sort Works?

Remaining Iteration

i. The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.

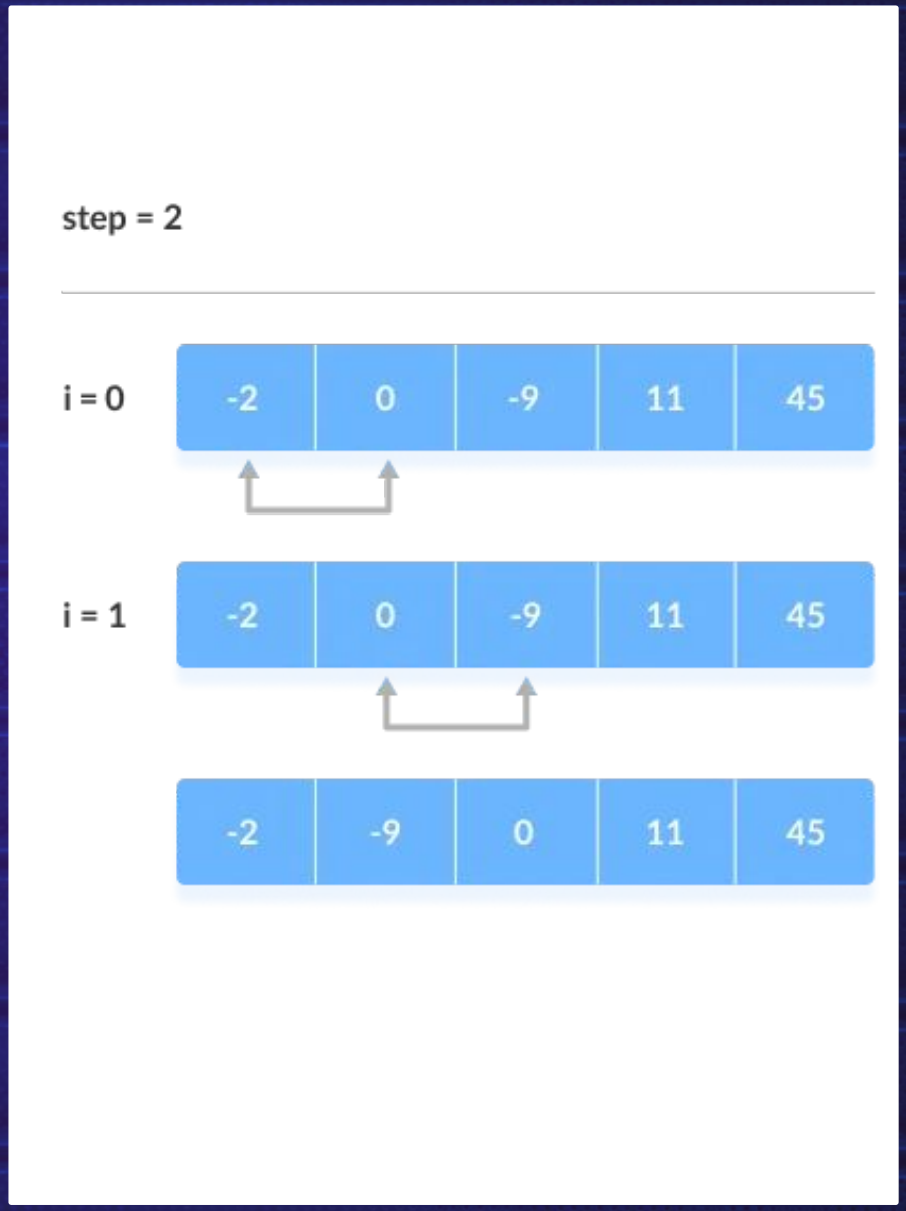
ii. In each iteration, the comparison takes place up to the last unsorted element.

iii. The array is sorted when all the unsorted elements are placed at their correct positions.



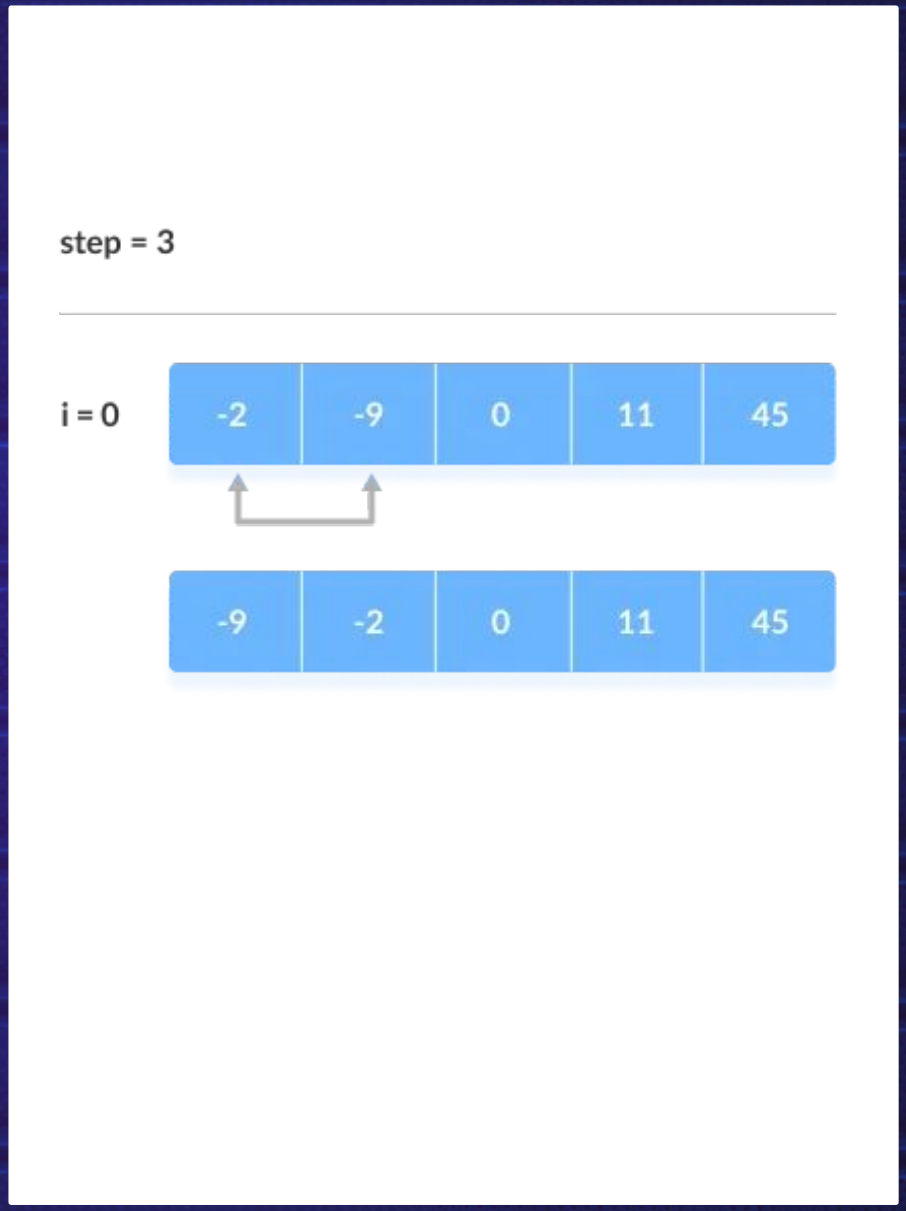
How Bubble Sort Works?

i.In each iteration, the comparison takes place up to the last unsorted element.



How Bubble Sort Works?

The array is sorted when all the unsorted elements are placed at their correct positions.



Bubble Sort Algorithm

```
bubbleSort(array)
 for i <- 1 to indexOfLastUnsortedElement-1
   if leftElement > rightElement
     swap leftElement and rightElement
end bubbleSort
```

Bubble Sort Applications

1. Bubble sort is a sorting algorithm used to sort the elements in an ascending order.
2. It uses less storage space
3. Bubble sort can be beneficial to sort the unsorted elements in a specific order.
4. It can be used to sort the students on basis of their height in a line.
5. To create a stack , pile up the elements on basis of their weight .

Selection Sort Algorithm

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

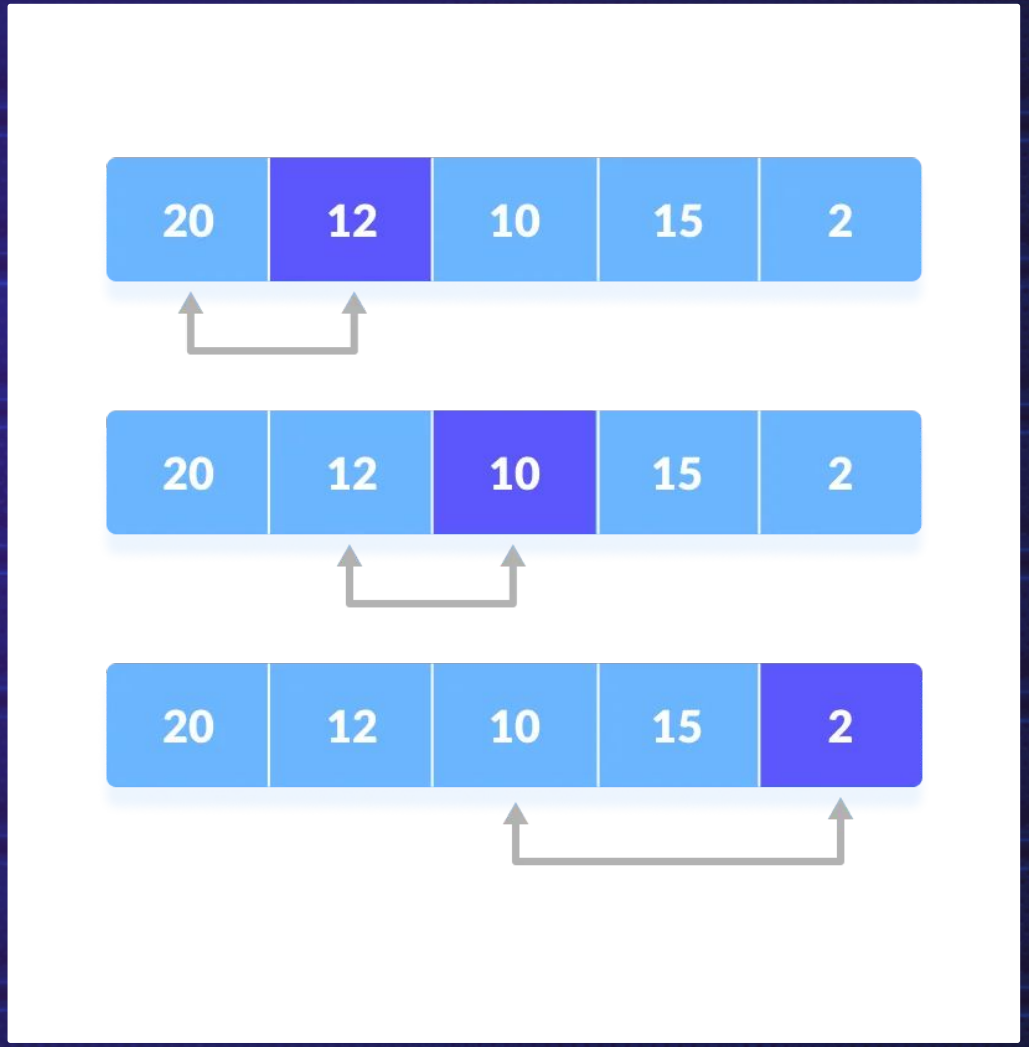
How Selection Sort Works?

1. Set the first element as minimum.

2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

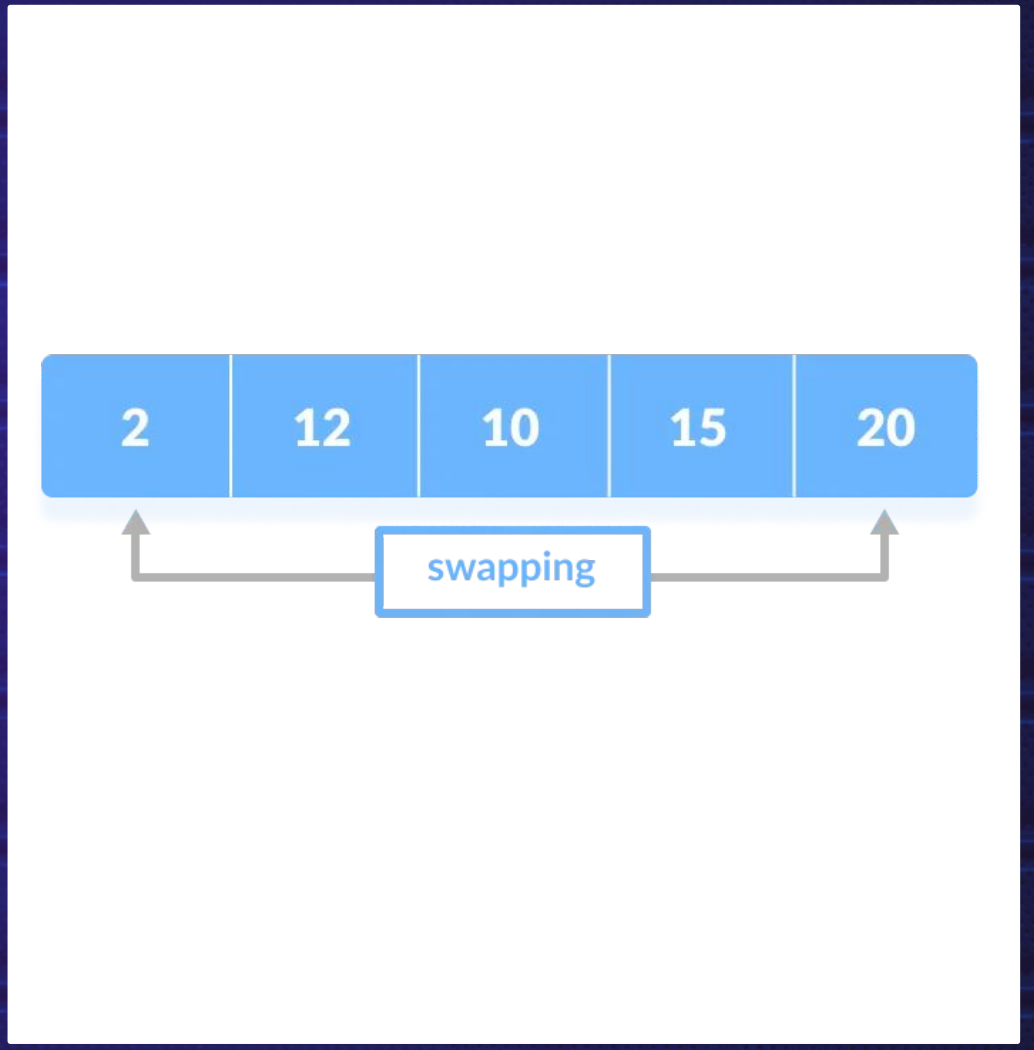
How Selection Sort Works?

3. Compare minimum with the third element. Again, if the third element is smaller, assign minimum to the third element otherwise do nothing. The process goes on until the last element.



How Selection Sort Works?

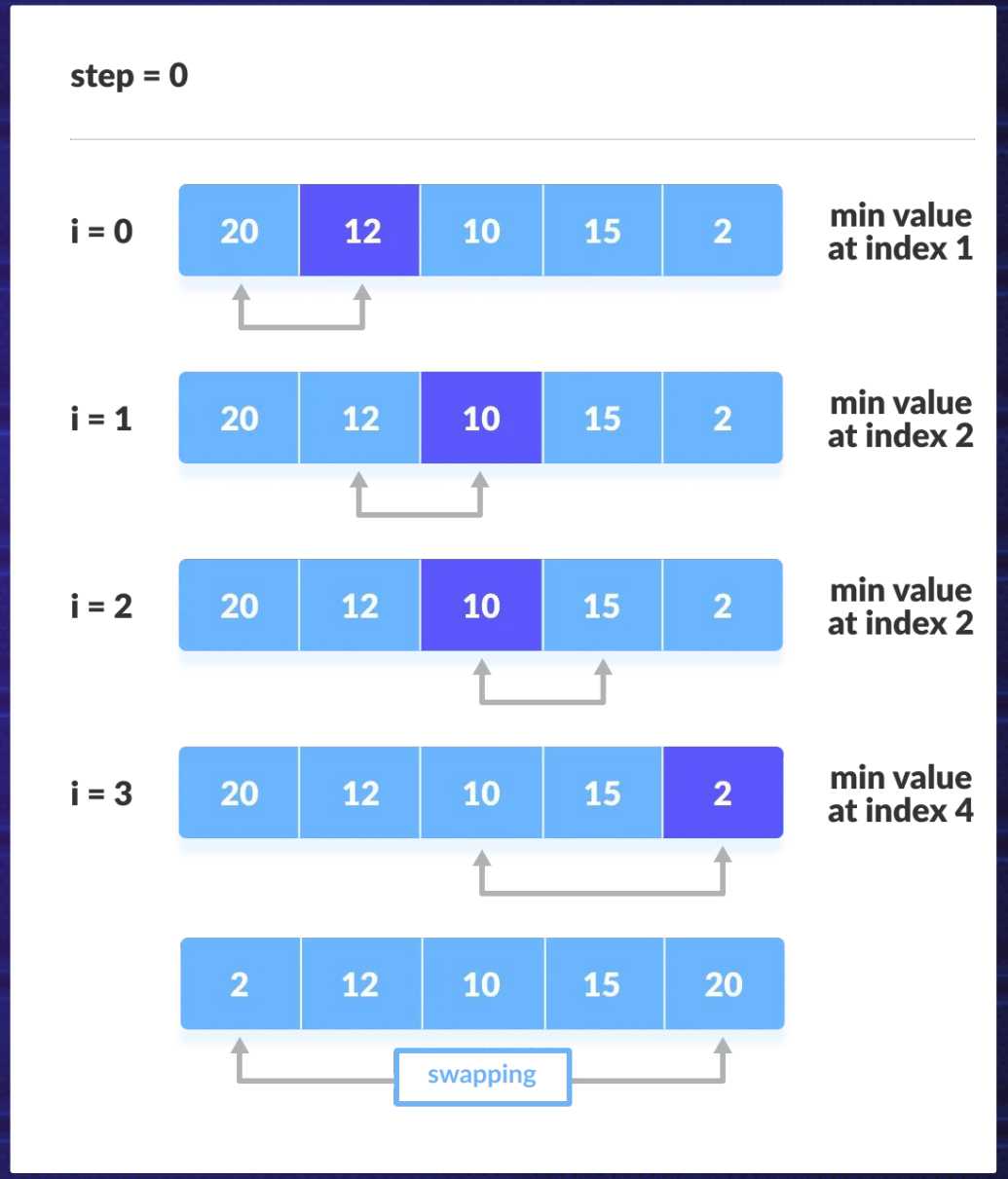
4. After each iteration, minimum is placed in the front of the unsorted list.



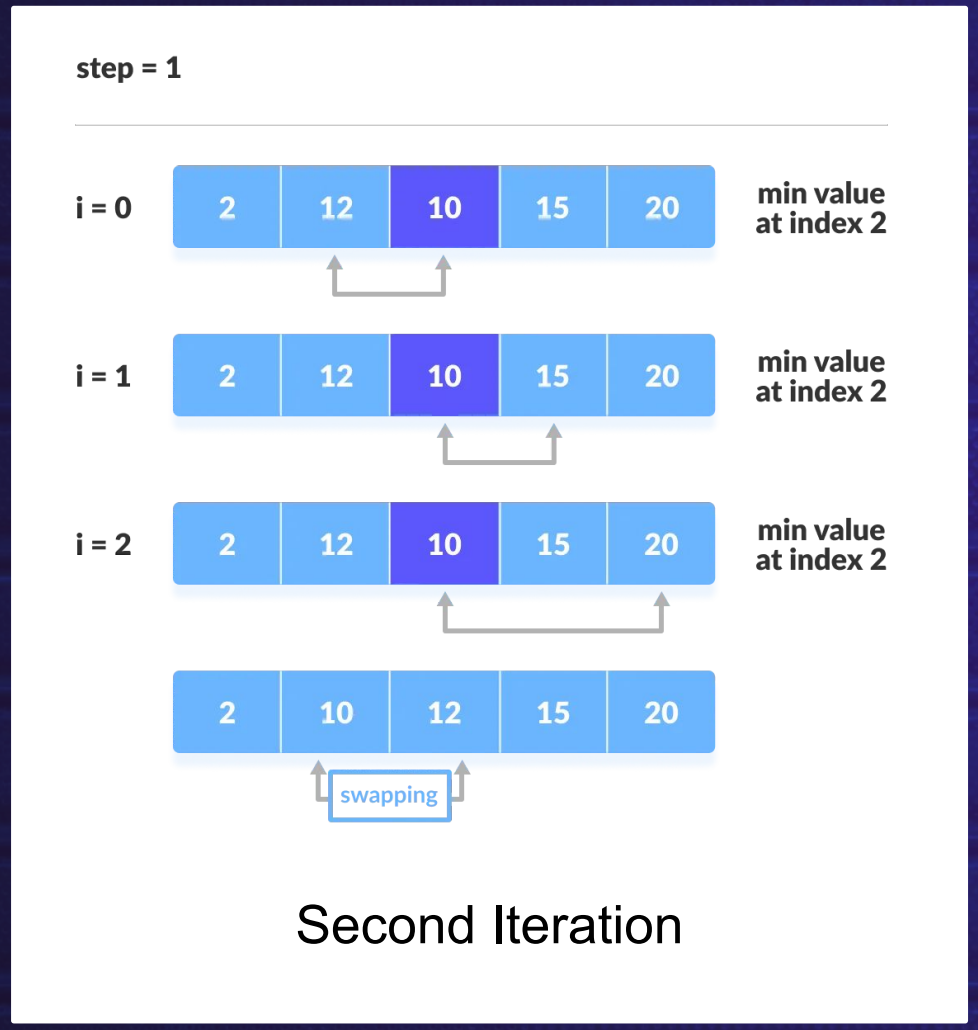
How Selection Sort Works?

5. For each iteration, indexing starts from the first unsorted element. Steps 1 to 3 are repeated until all the elements are placed correctly.

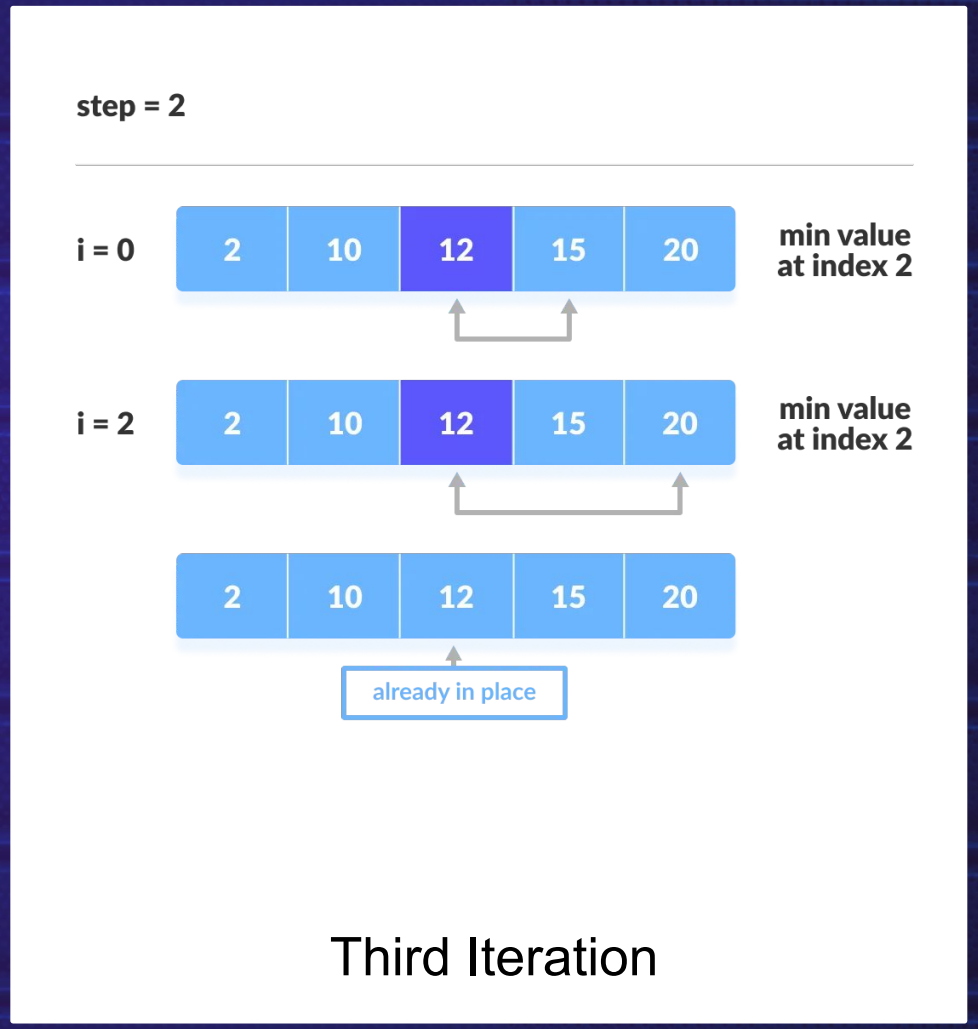
The first iteration



How Selection Sort Works?

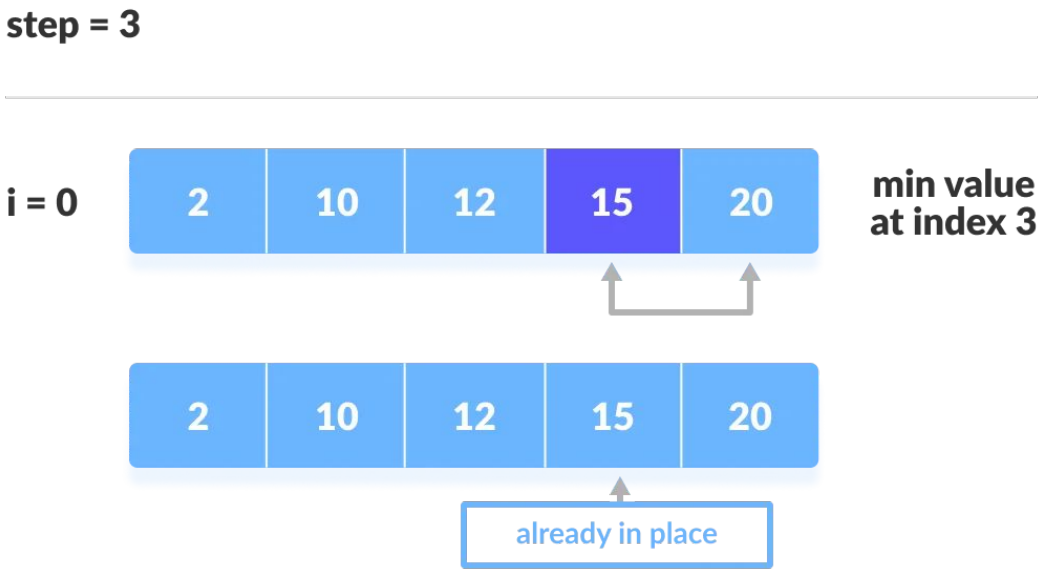


Second Iteration



Third Iteration

How Selection Sort Works?



Fourth Iteration

Another Example

```
arr[] = 64 25 12 22 11
//  Find  the  minimum element  in  arr[0...4]
// and place it at beginning
11 25 12 22 64

//  Find  the  minimum  element  in  arr[1...4]
//  and  place  it  at  beginning  of  arr[1...4]
11 12 25 22 64

//  Find  the  minimum  element  in  arr[2...4]
//  and  place  it  at  beginning  of  arr[2...4]
11 12 22 25 64

//  Find  the  minimum  element  in  arr[3...4]
//  and  place  it  at  beginning  of  arr[3...4]
11 12 22 25 64
```

Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the
minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

Selection Sort Applications

1. when a small dataset needs to be sorted
2. in algorithms where swapping costs do not matter.
3. in scenarios where there is a need to check all elements.
4. In situations where the cost of writing in flash memory matters.