Lecture 16

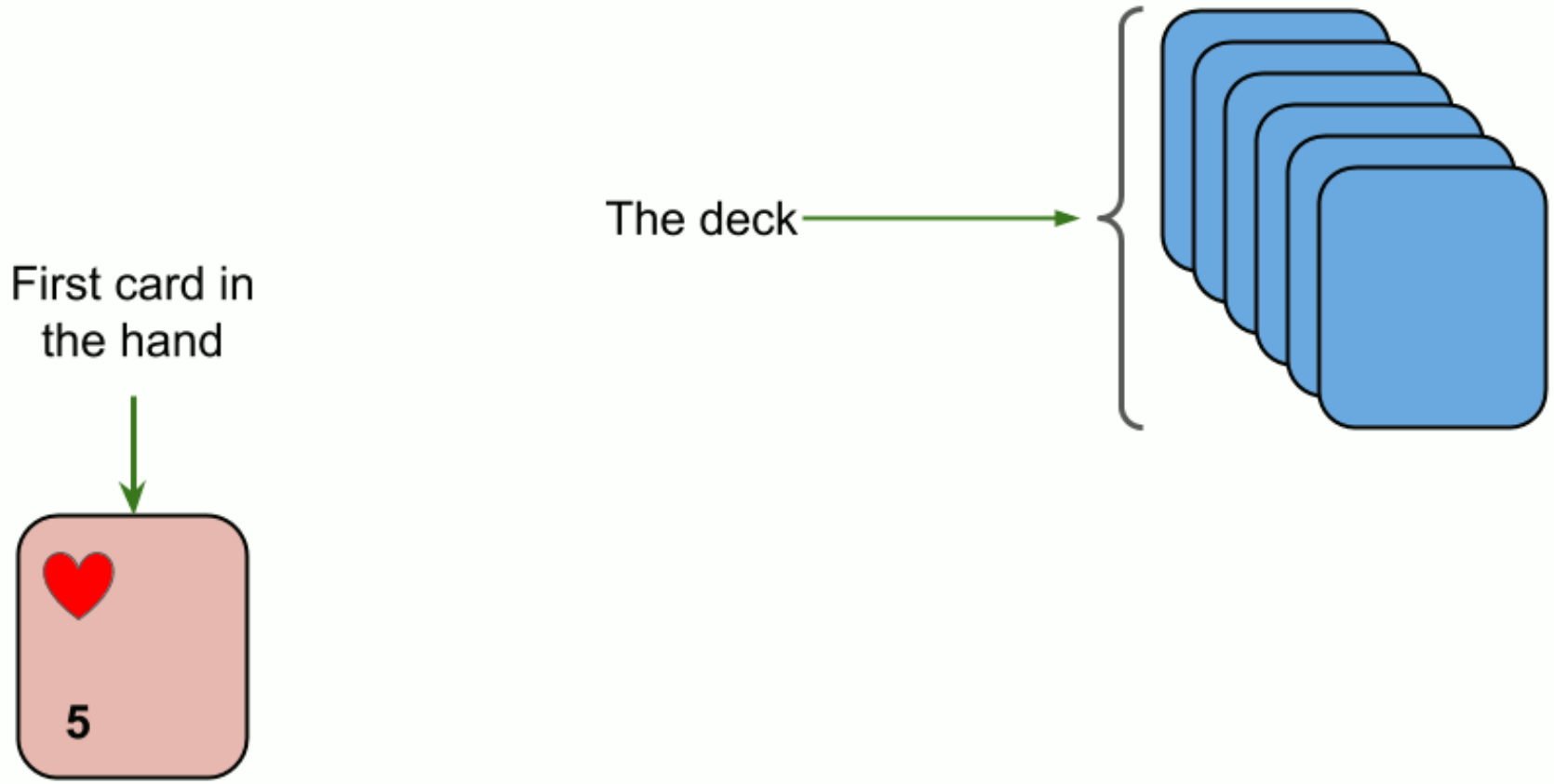
M M Imran

Insertion Sort

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

- 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 in their right place.


First card in
the hand
5
The deck

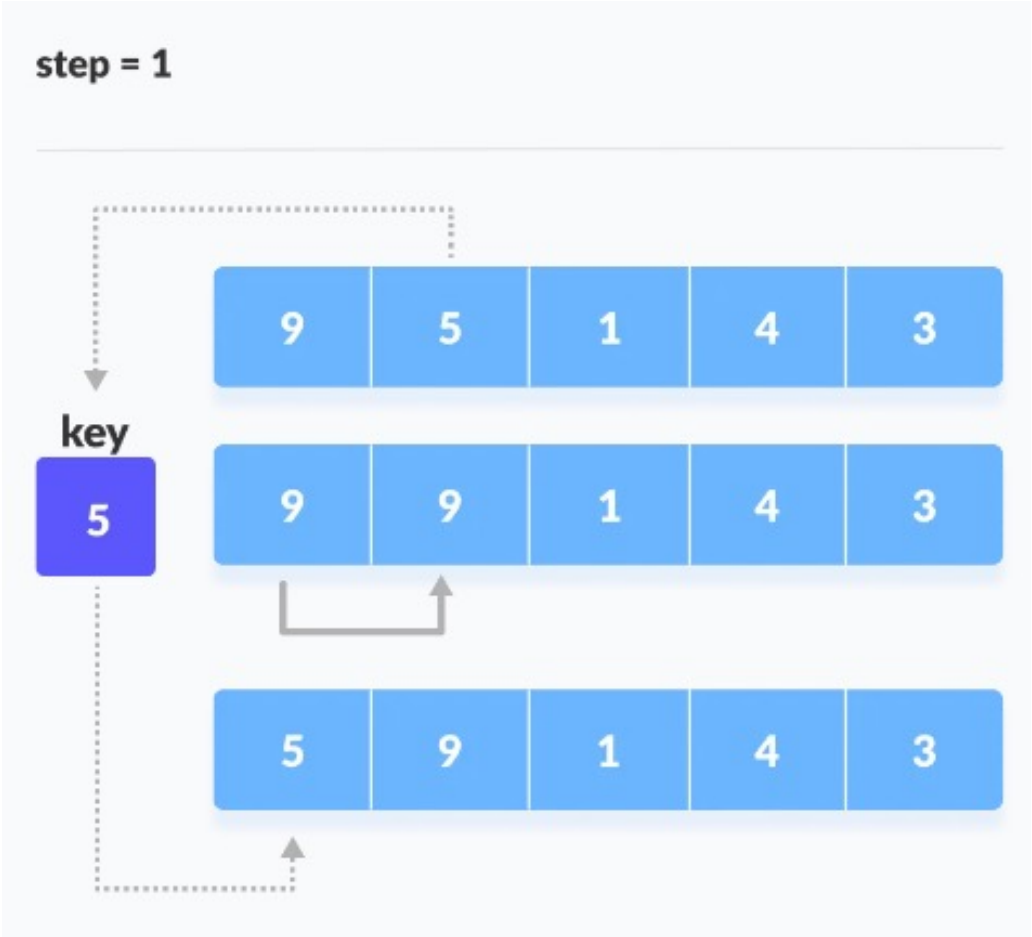
Steps

Let’s say we want to sort this array [9, 5, 1, 4, 3]

1. Assume, the first element in the array is 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 key is placed in front of the first element.

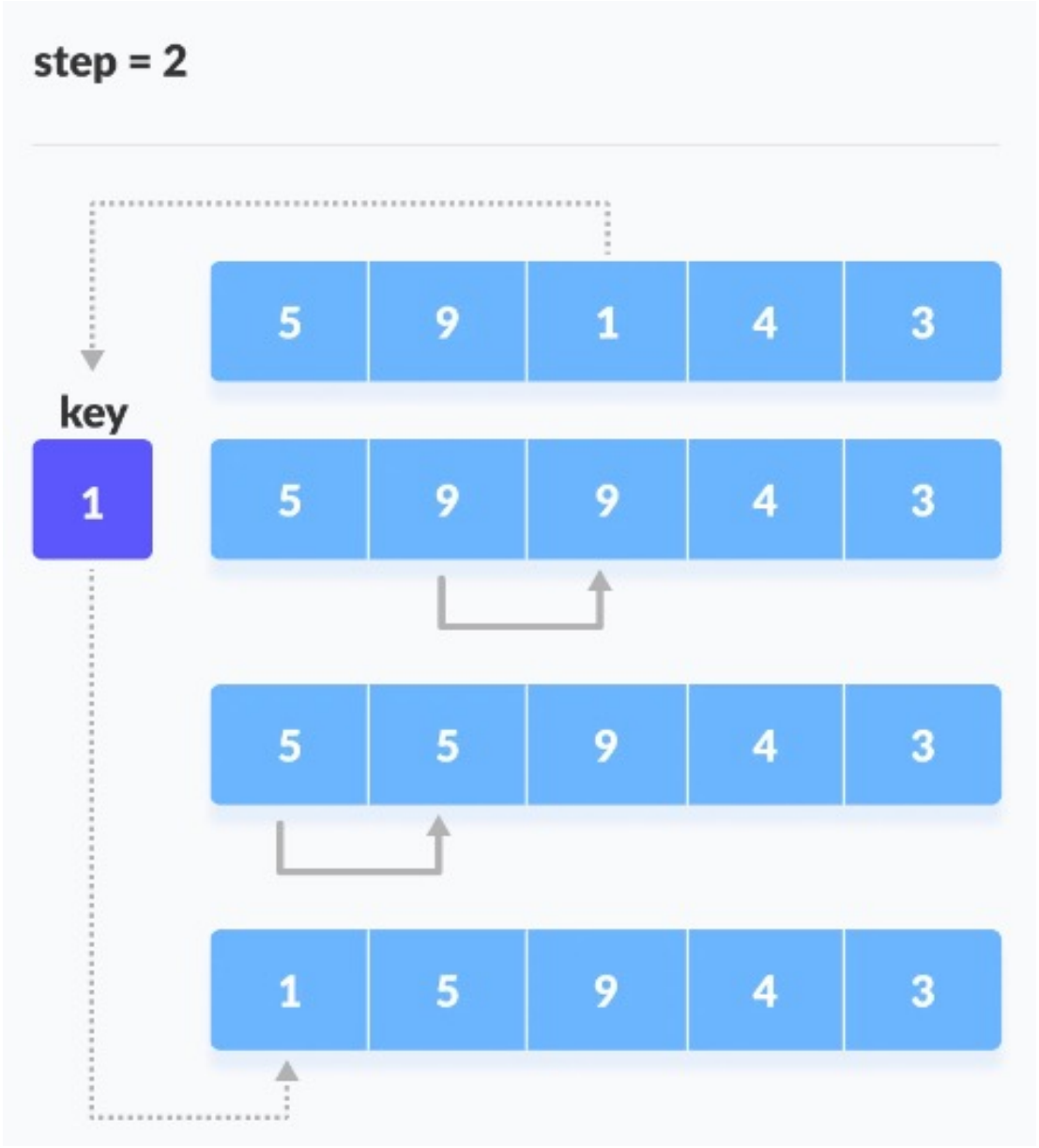


2. Now, the first two elements are sorted.

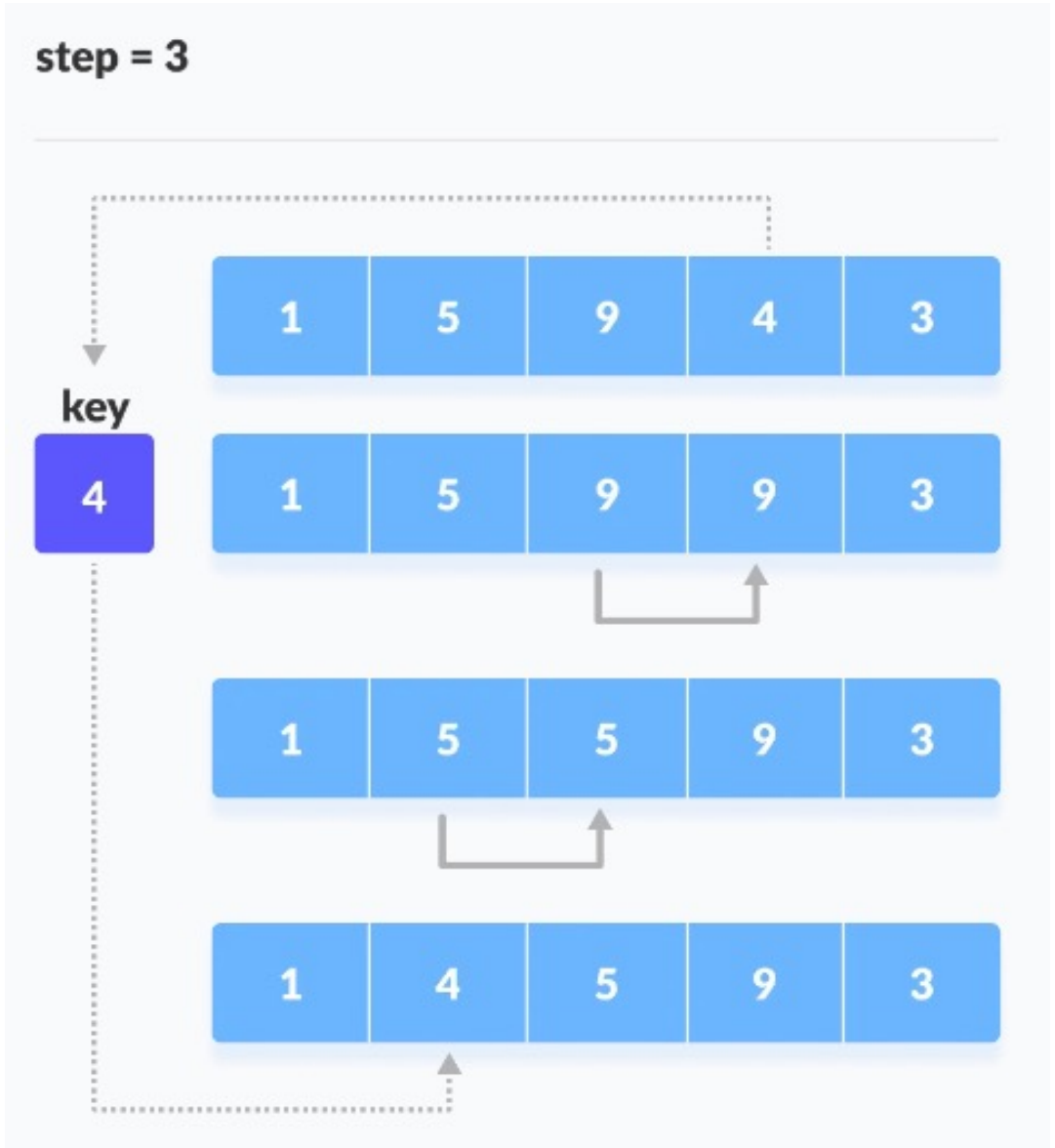
Take the third element and compare it with the elements on the left of it.

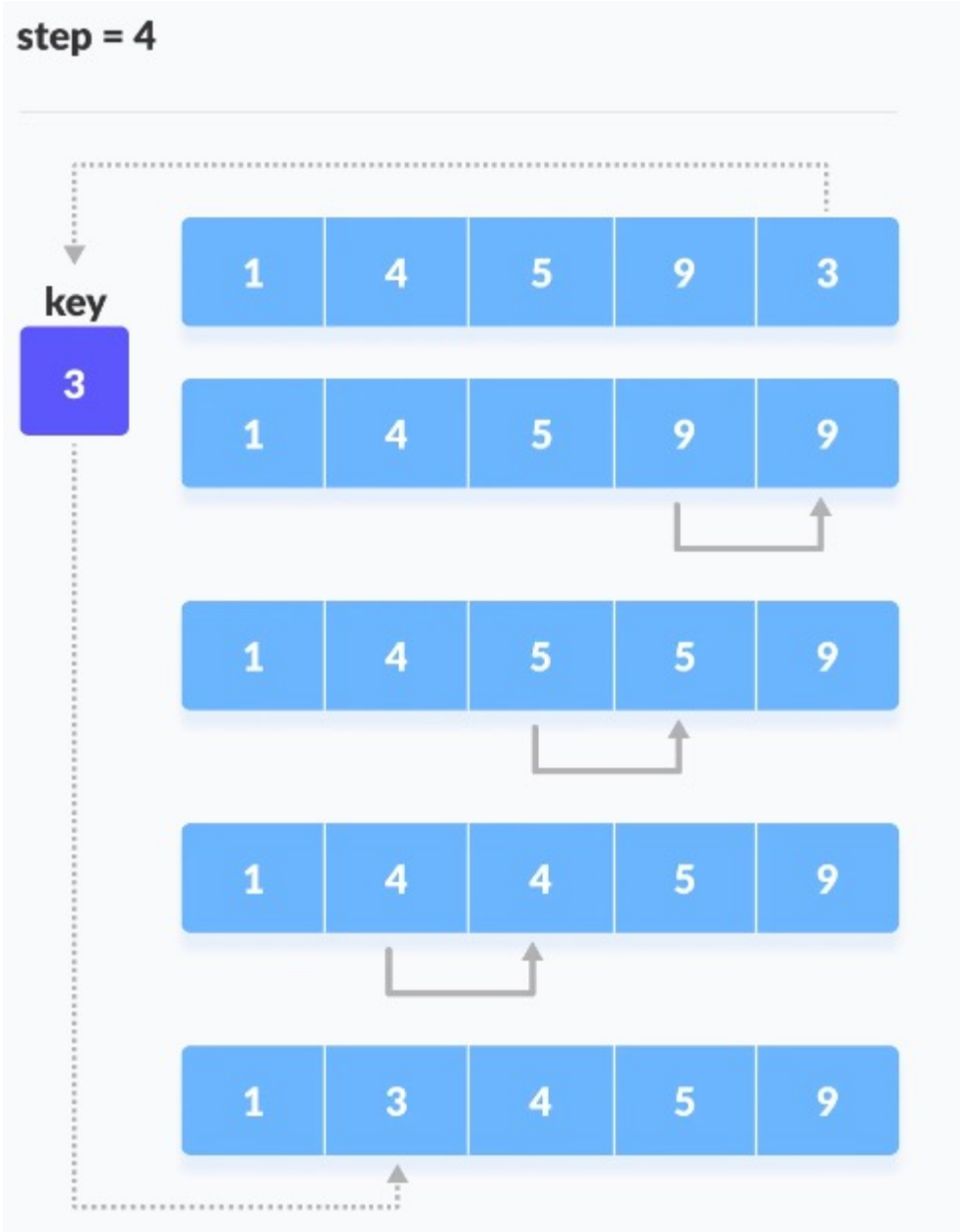
Placed it just behind the element smaller than it.

If there is no element smaller than it, then place it at the beginning of the array.



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





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
```

```cpp
void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;

    // Compare key with each element on the left of it
    // until an element smaller than it is found.
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

int main() {
  int data[] = {9, 5, 1, 4, 3};
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  cout << "Sorted array in ascending order:\n";
  for (int i = 0; i < size; i++) {
    cout << data[i] << " ";
  }
}
```

```
Sorted array in ascending order:
1 3 4 5 9
```

Insertion Sort Complexity

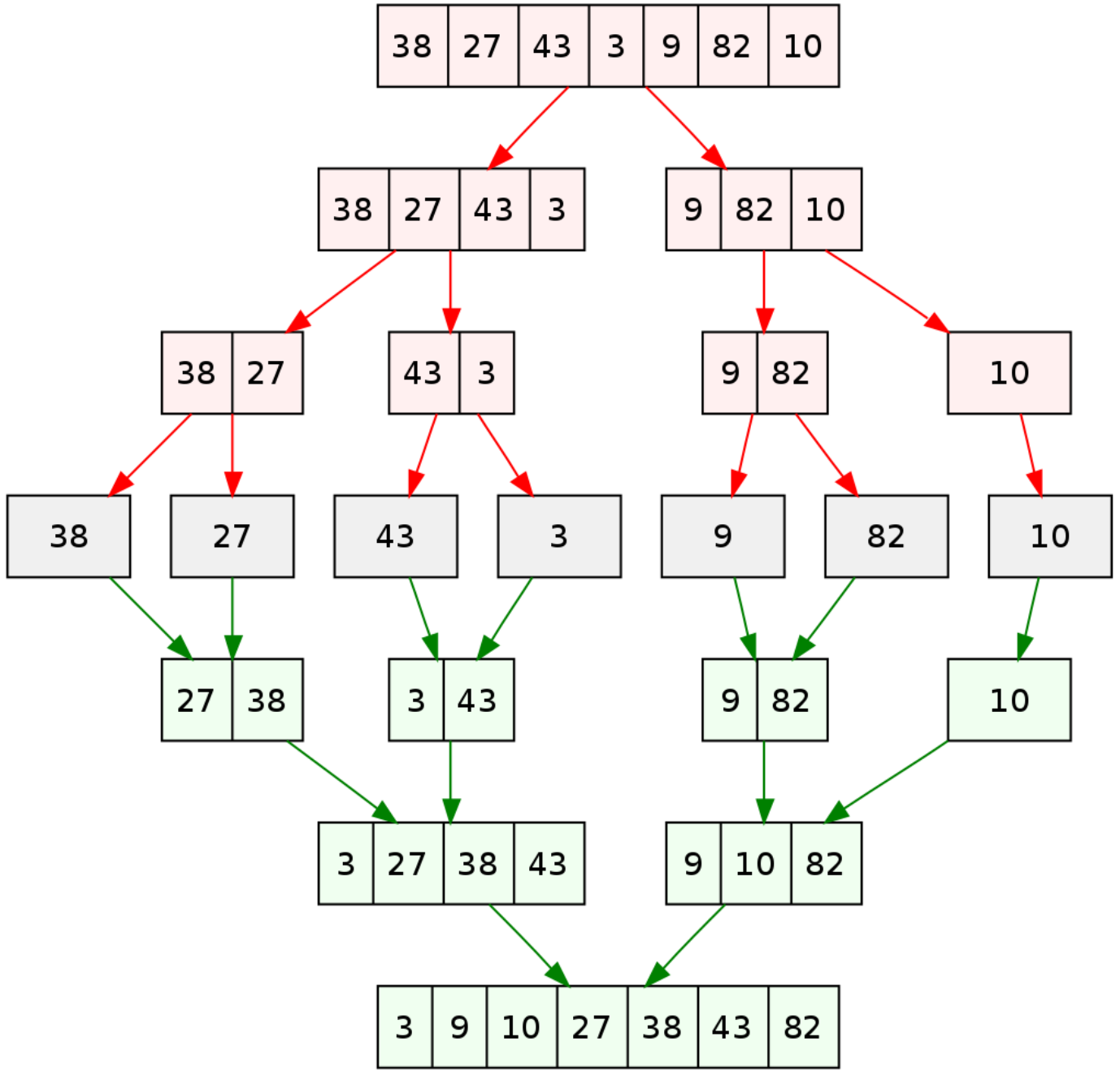
Time Complexity	
Best	$O(n)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	$O(1)$

Question

Why the Best-Case Complexity is O(n)?

Divide and Conquer

- Divide and Conquer is an algorithmic paradigm.
- It’s used to solve problems by continually dividing the problem into smaller parts until a part is easy enough to solve (conquer) on its own.
- The solutions to the solved parts are then combined to give the solution for the original problem.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Divide and Conquer

This technique can be divided into the following three parts:

1. **Divide**: This involves dividing the problem into smaller sub-problems.
2. **Conquer**: Solve sub-problems by calling recursively until solved.
3. **Combine**: Combine the sub-problems to get the final solution of the whole problem.

Application

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix multiplication
- Karatsuba Algorithm

Merge Sort Algorithm

- The Merge Sort algorithm is a sorting algorithm that is considered as an example of the divide and conquer strategy.

- In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

- We can think of it as a recursive algorithm that continuously splits the array in half until it cannot be further divided.

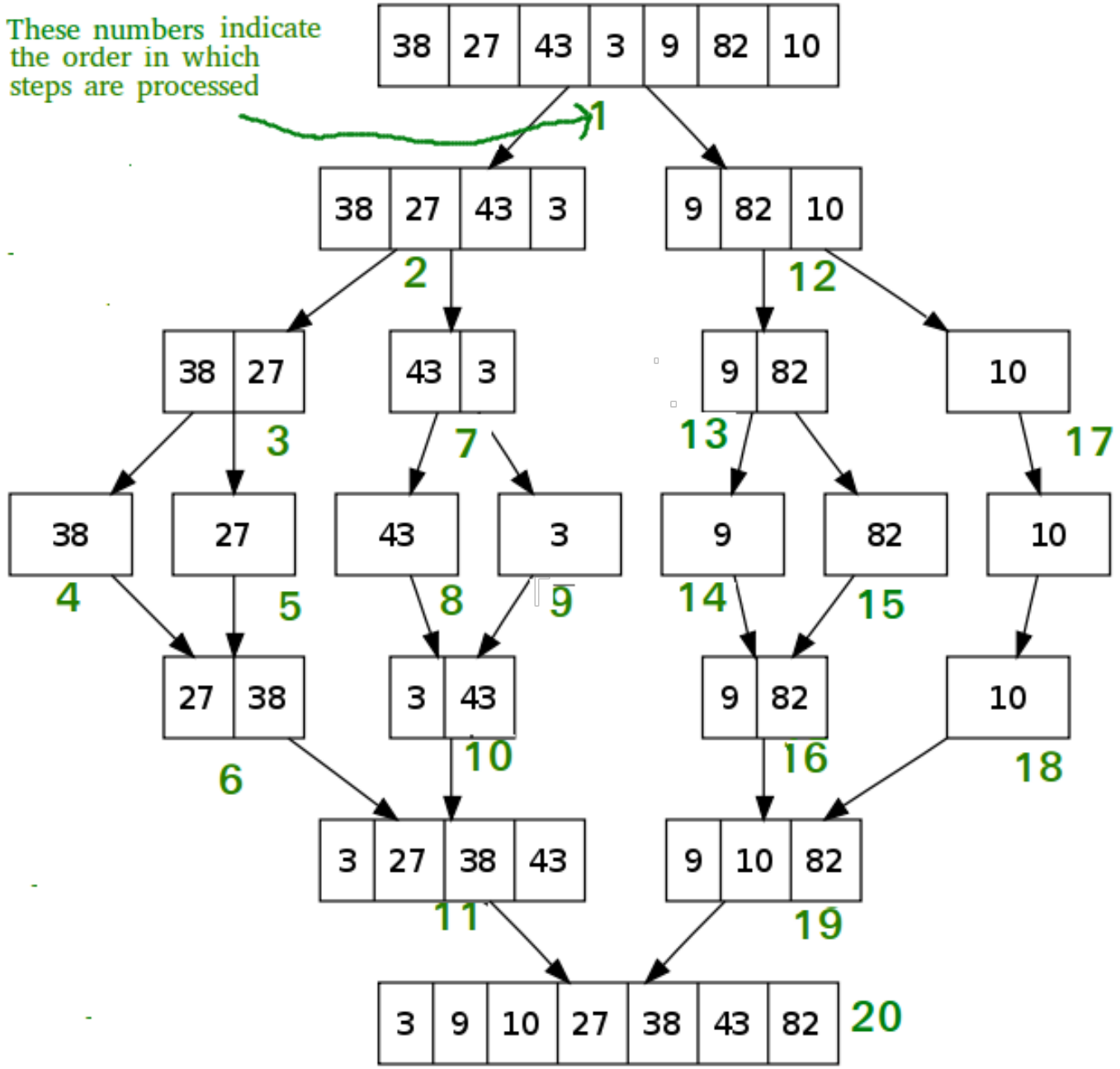
Merge Sort Algorithm

- If the array has multiple elements, we split the array into halves and recursively invoke the merge sort on each of the halves.

- If the array becomes empty or has only one element left, the dividing will stop, i.e. it is the **base case** to stop the recursion.

- Finally, when both the halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

6 5 3 1 8 7 2 4


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Algorithm

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

Steps

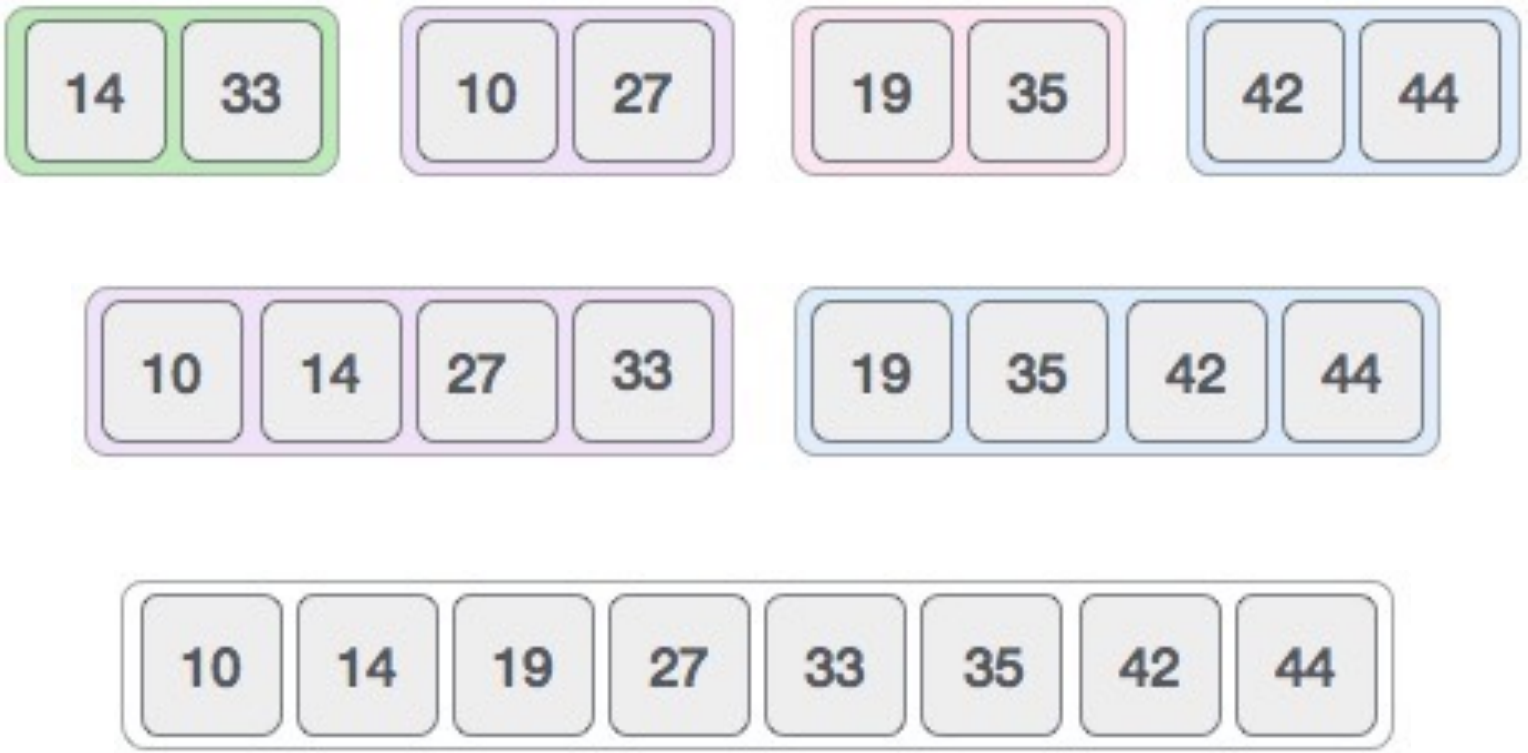
Let’s consider this unsorted array.

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

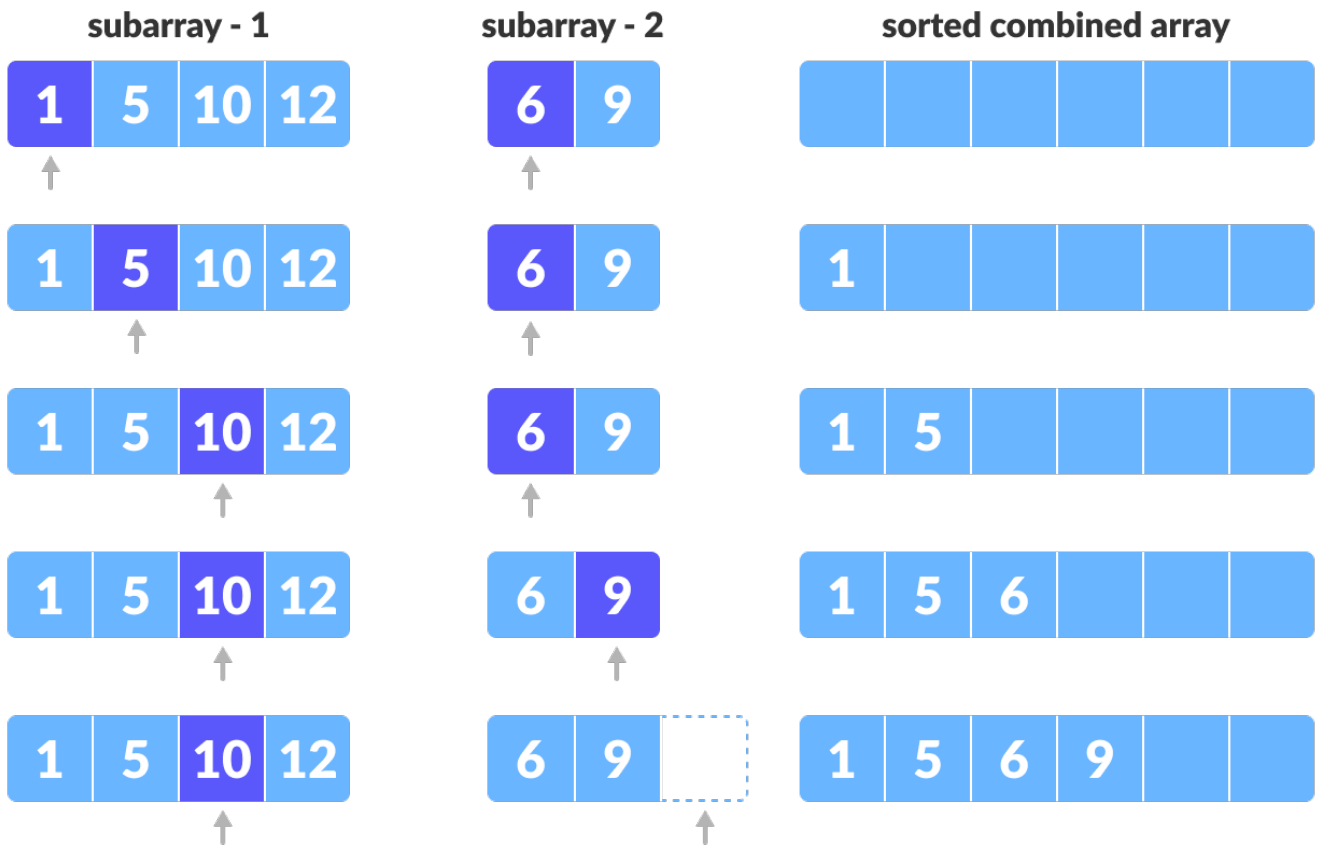
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. We first compare the element for each list and then combine them into another list in a sorted manner.



How merge is done



Since there are no more elements remaining in the second array, and we know that both the arrays were sorted when we started, we can copy the remaining elements from the first array directly.

Merge Sort Pseudocode

```
MergeSort(A, p, r):
    if p >= r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

Merge Sort Complexity

Time Complexity	
Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(n)

Searching Algorithms

- Linear Search
- Binary search

Linear Search

- Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found.

- It is the simplest searching algorithm.

Linear Search Algorithm

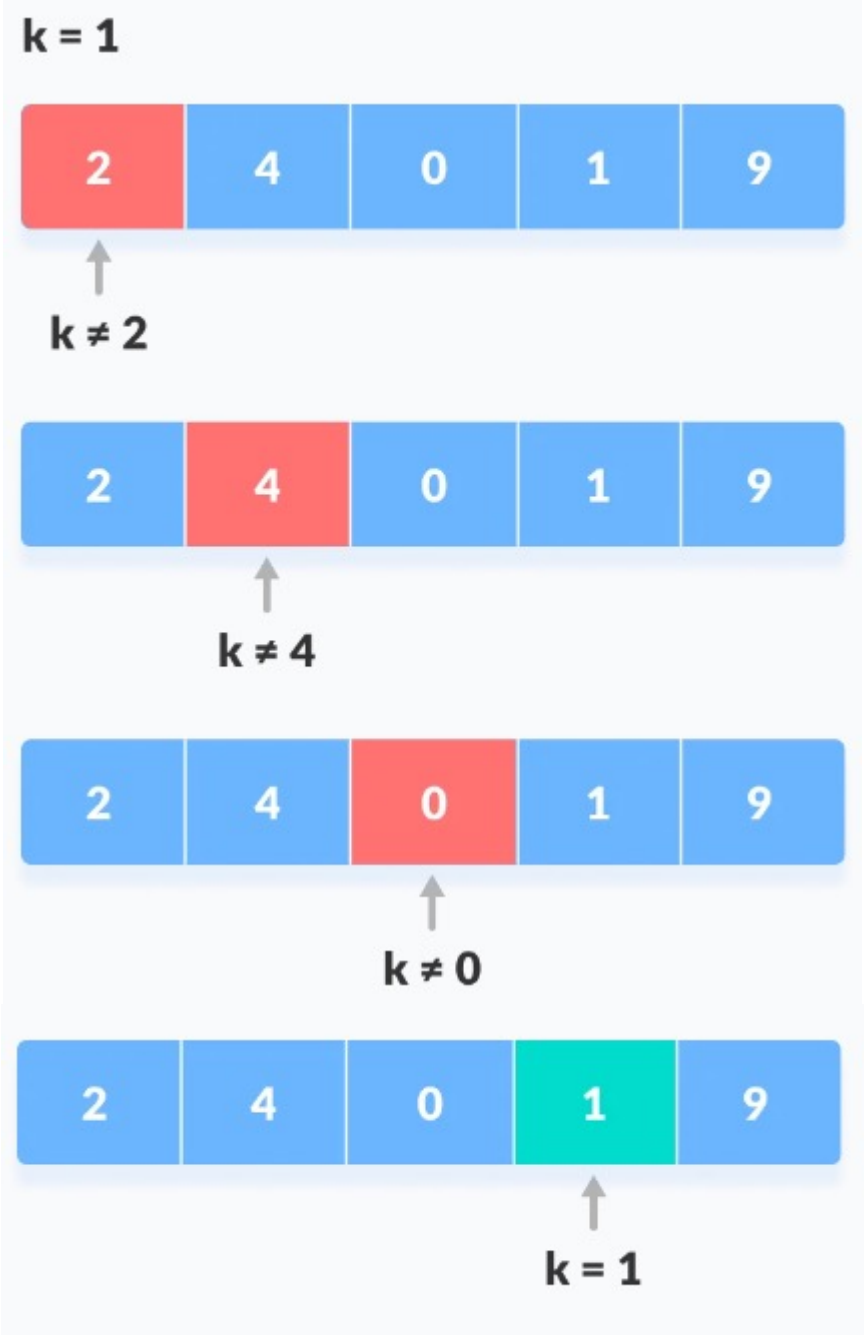
```
LinearSearch(array, key)
        for each item in the array
                if item == value
                        return its index
```

Steps

Let’s consider this array [2, 4, 0, 1, 9]

We want to search for 1. Let’s denote it with k = 1

Start from the first element, compare k with each element



```cpp
int search(int array[], int n, int x) {
  // Going through array sequencially
  for (int i = 0; i < n; i++)
    if (array[i] == x)
      return i;

  return -1;
}

int main() {
  int array[] = {2, 4, 0, 1, 9};
  int x = 1;
  int n = sizeof(array) / sizeof(array[0]);

  int result = search(array, n, x);

  if (result == -1)
      cout << "Element not found";
  else
      cout << "Element found at index: " << result;
}
```

```
Element found at index: 3
```

Linear Search Complexities

Time Complexity: O(n)

Space Complexity: O(1)

Binary Search

- Binary search algorithm works on the principle of divide and conquer.

- Input array should be in the sorted form.

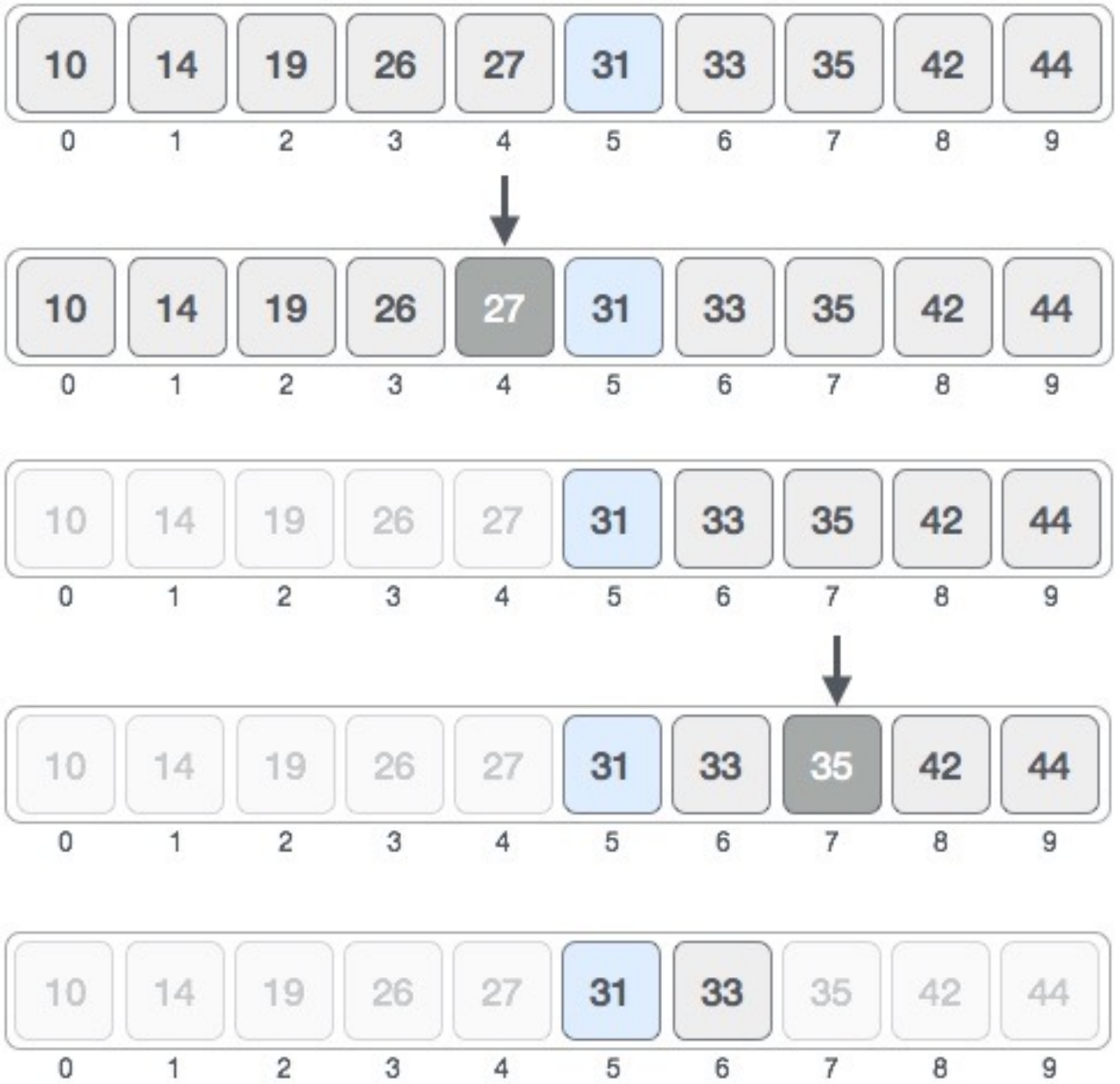
Binary Search Algorithm

1. Begin with the **mid element** of the whole array as a search key.
2. If the value of the search key is equal to the item then return an index of the search key.
3. Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
4. Otherwise, narrow it to the upper half.
5. Repeatedly check from the **second point** until the value is found or the interval is empty.

$$mid = low + (high - low) / 2$$

Steps

Search for 31



Steps


10 14 19 26 27 31 33 35 42 44
0 1 2 3 4 5 6 7 8 9
10 14 19 26 27 31 33 35 42 44
0 1 2 3 4 5 6 7 8 9

Binary Search Algorithm

Binary Search Algorithm can be implemented in two ways:

1. Iterative Method
2. Recursive Method

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 > arr[mid]) // x is on the right side
        low = mid + 1
    else                   // x is on the left side
        high = mid - 1
```

Recursive 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 > arr[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)
```

Iterative Method

```c
int binarySearch(int array[], int x, int low, int high) {
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int x = 4;
  int n = sizeof(array) / sizeof(array[0]);
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

Recursive Method

```c
int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;

    // If found at mid, then return it
    if (array[mid] == x)
      return mid;

    // Search the left half
    if (array[mid] > x)
      return binarySearch(array, x, low, mid - 1);

    // Search the right half
    return binarySearch(array, x, mid + 1, high);
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int x = 4;
  int n = sizeof(array) / sizeof(array[0]);
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

Binary Search Complexity

Time Complexities

Best case complexity: O(1)

Average case complexity: O(log n)

Worst case complexity: O(log n)

Space Complexity: O(1).