Unit 7: Sorting

Prepared By:
Tanvi Patel
Asst. Prof. (CSE)

Contents

- Sorting: basics
- Selection and Heap Sort
- Insertion and Shell Sort
- Exchange: Bubble and Quick Sort
- External sort

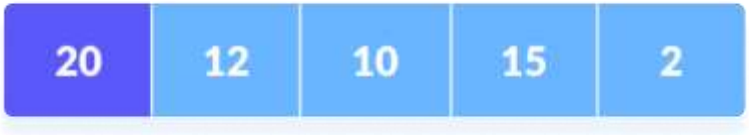
Selection sort

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

Selection sort

Working of Selection Sort

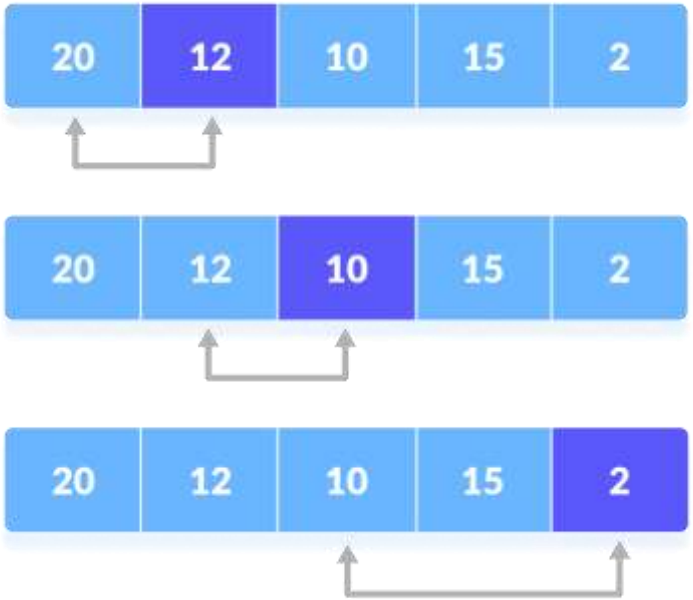
- Set the first element as minimum



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

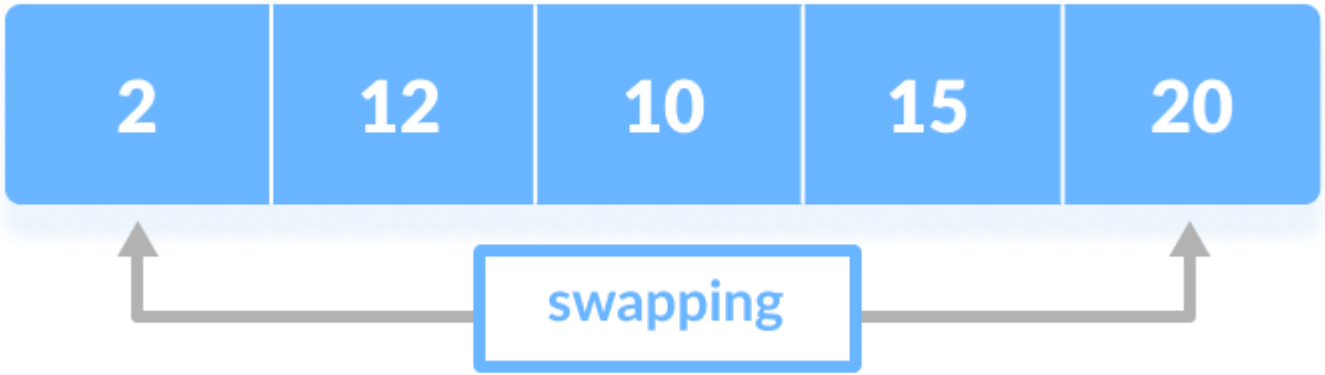
  Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

Selection sort



Selection sort

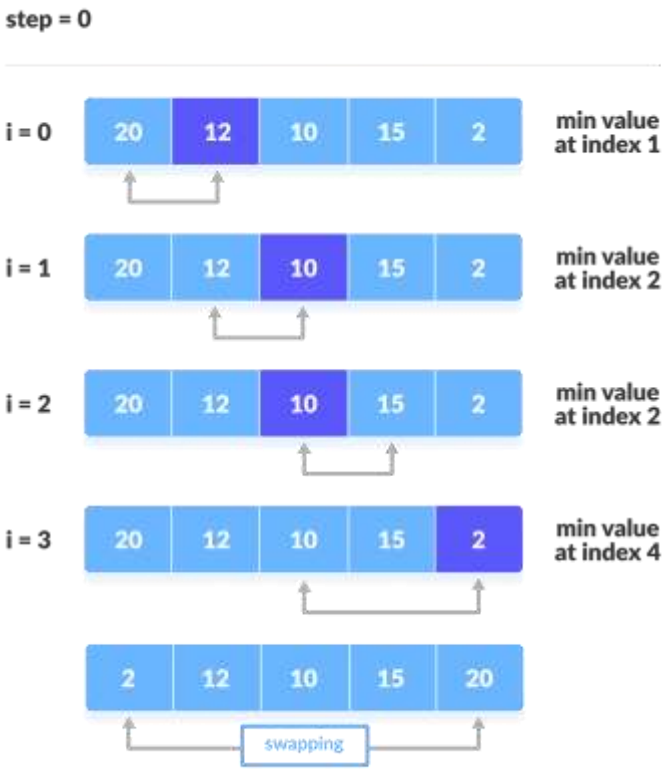
After each iteration, minimum is placed in the front of the unsorted list.



For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

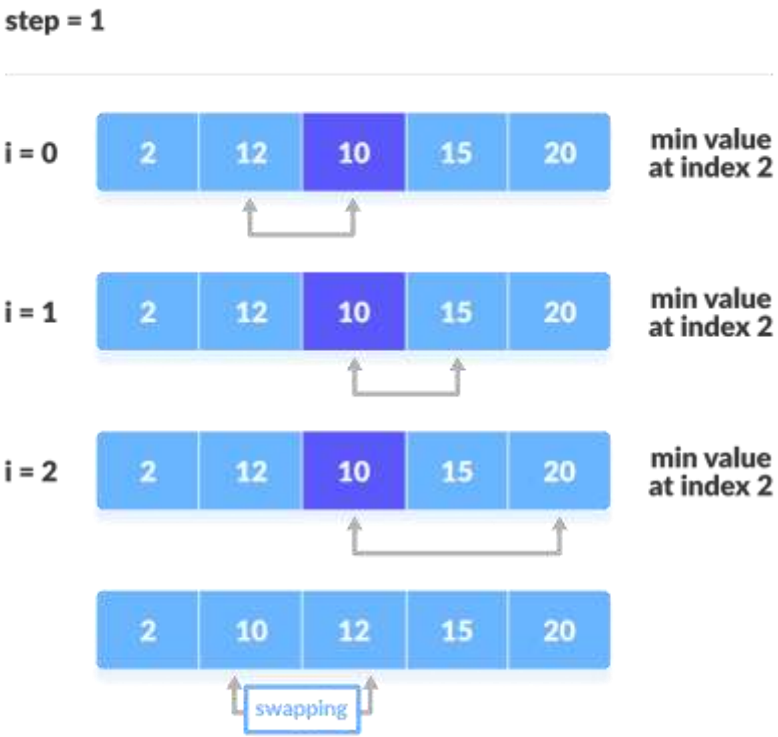
Selection sort

The first iteration



Selection sort

The second iteration



Selection sort

The third iteration

Selection sort

The fourth iteration

Selection sort

Selection Sort Complexity

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Stability	No

Selection sort

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

- Number of comparisons: $(n - 1) + (n - 2) + (n - 3) + ... + 1 = n(n - 1) / 2$ nearly equals to n^2.
- **Complexity** = $O(n^2)$

Selection sort

Time Complexities:

- **Worst Case Complexity:** $O(n^2)$
  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
- **Best Case Complexity:** $O(n^2)$
  It occurs when the array is already sorted
- **Average Case Complexity:** $O(n^2)$
  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).
- The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.

Selection sort

Selection Sort Applications

- The selection sort is used when
- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory

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.
- A similar approach is used by insertion sort.

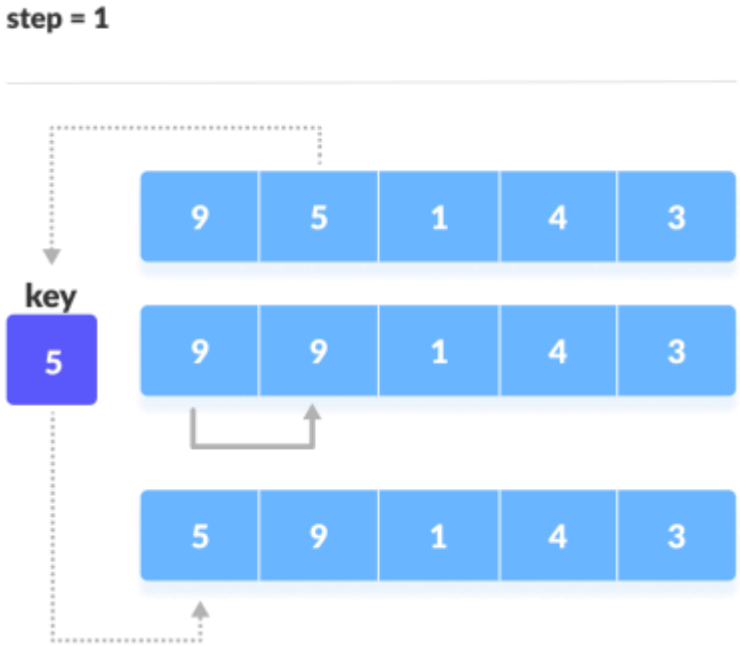
Insertion sort

- **Working of Insertion Sort**
- Suppose we need to sort the following array.

Insertion sort

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

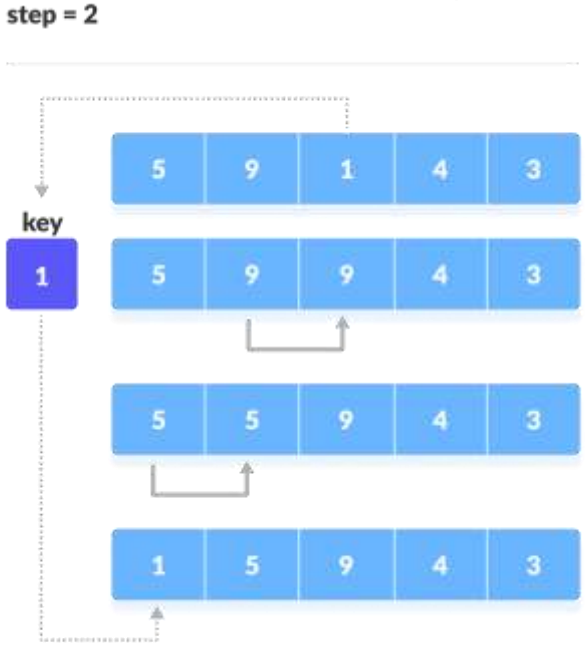
  Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



Insertion sort

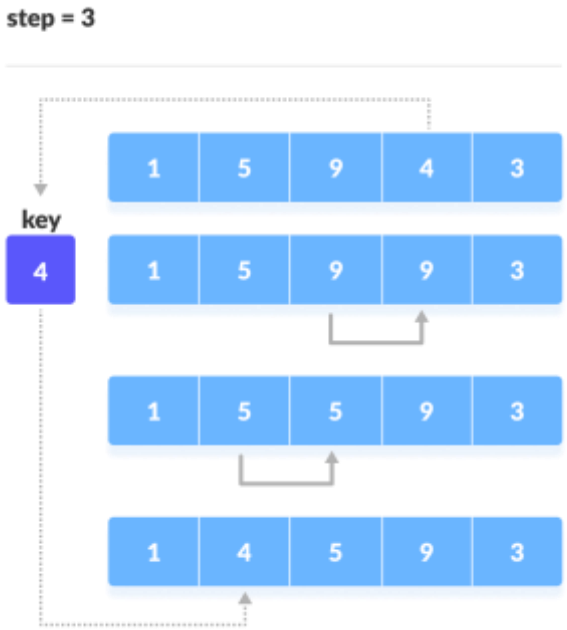
- Now, the first two elements are sorted.

  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.

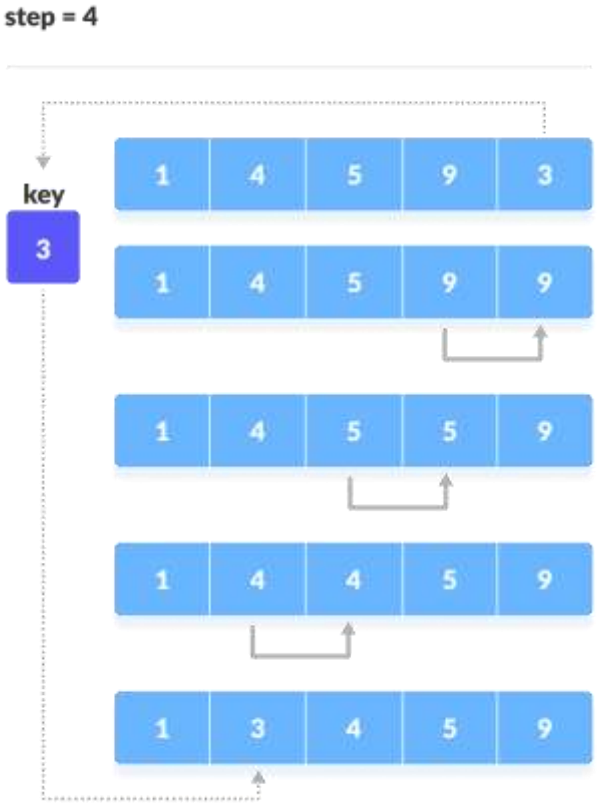


Insertion sort

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



Insertion sort



Insertion sort

- **Insertion Sort Complexity**

Time Complexity	
Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$
Stability	Yes

Insertion sort

Time Complexities

- **Worst Case Complexity:** $O(n^2)$
  Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

  Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

  Thus, the total number of comparisons = $n*(n-1) \sim n^2$
- **Best Case Complexity:** $O(n)$
  When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.
- **Average Case Complexity:** $O(n^2)$
  It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Insertion sort

Insertion Sort Applications

- The insertion sort is used when:
- the array is has a small number of elements
- there are only a few elements left to be sorted

Shell sort

- Shell sort is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted.
- The interval between the elements is reduced based on the sequence used.

Working of Shell Sort

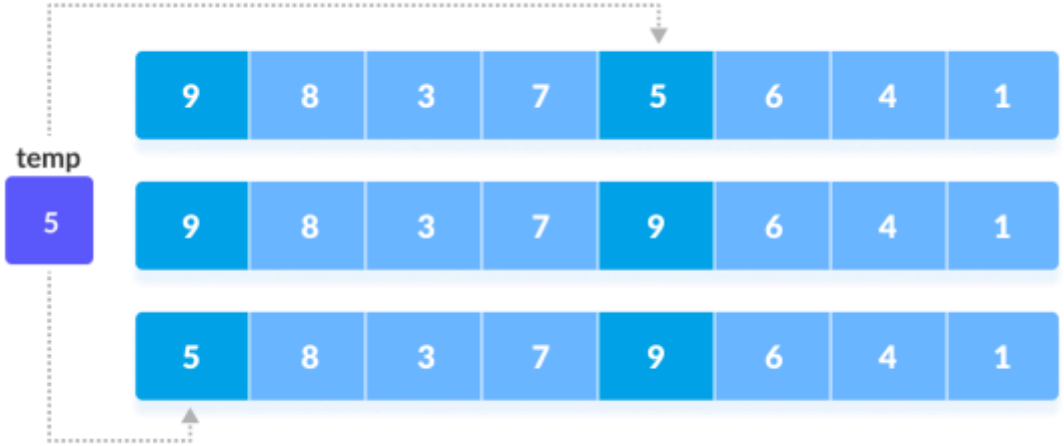
- Suppose, we need to sort the following array.

Shell sort

- We are using the shell's original sequence (N/2, N/4, ...1) as intervals in our algorithm.

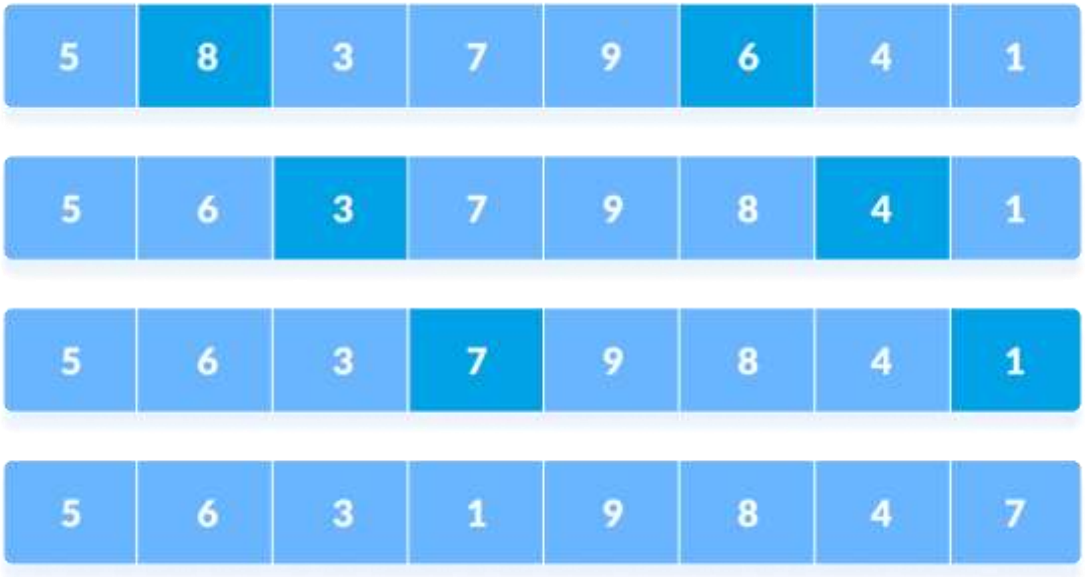
  In the first loop, if the array size is N = 8 then, the elements lying at the interval of N/2 = 4 are compared and swapped if they are not in order.The 0th element is compared with the 4th element.

- If the 0th element is greater than the 4th one then, the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position.



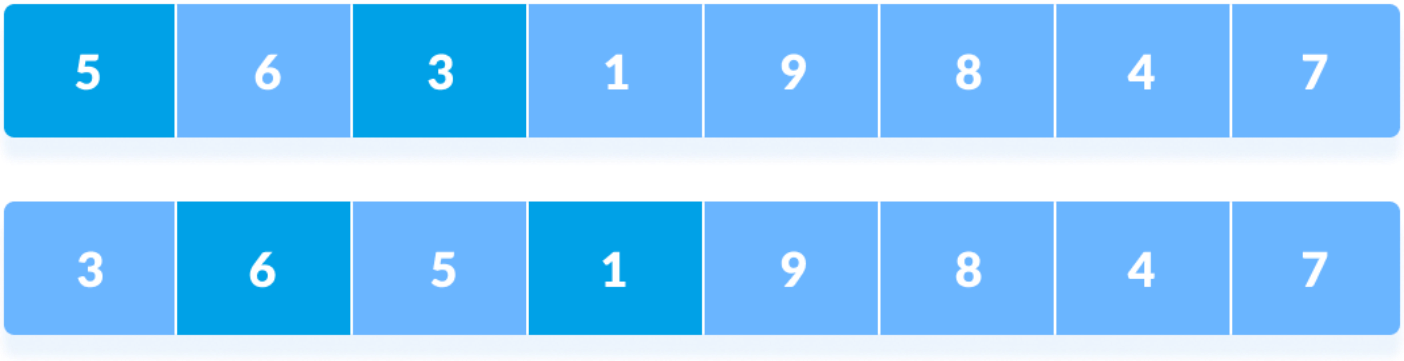
Shell sort

- This process goes on for all the remaining elements.



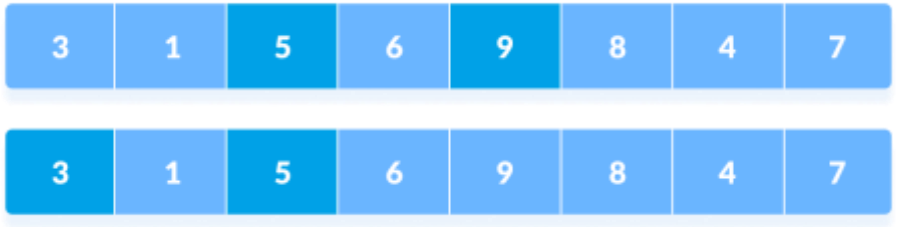
Shell sort

- In the second loop, an interval of N/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted.



Shell sort

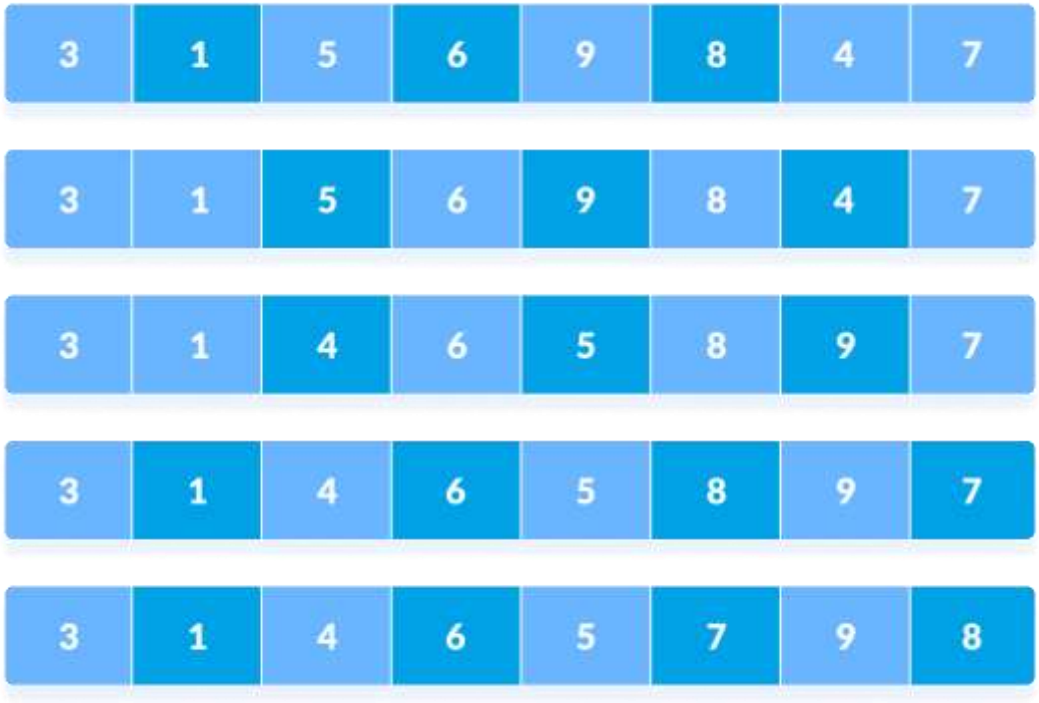
- All the elements in the array lying at the current interval are compared.



- The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position are also compared. All the elements in the array lying at the current interval are compared.

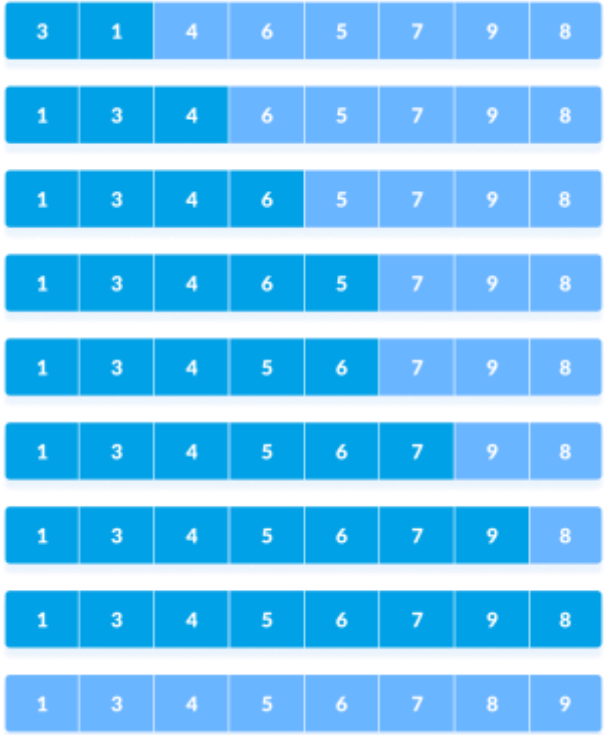
Shell sort

- The same process goes on for remaining elements.



Shell sort

- Finally, when the interval is N/8 = 8/8 =1 then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.



Shell sort

- Start with gap = n/2 and sort the sub arrays
- Keep reducing gap by n/2 in and keep on sorting subarrays.
- Last iteration should have gap=1. At this point it is same as insertion sort.

Shell sort

- **Shell Sort Complexity**
- Shell sort is an unstable sorting algorithm because this algorithm does not examine the elements lying in between the intervals.

Time Complexity	
Best	O(nlog n)
Worst	$O(n^2)$
Average	O(nlog n)
Stability	No

Shell sort

Time Complexity

- **Worst Case Complexity**: less than or equal to $O(n^2)$
- **Best Case Complexity**: O(n*log n)
  When the array is already sorted, the total number of comparisons for each interval (or increment) is equal to the size of the array.
- **Average Case Complexity**: O(n*log n)
- The complexity depends on the interval chosen. The above complexities differ for different increment sequences chosen. Best increment sequence is unknown.

Bubble sort

- **Bubble sort** is a sorting algorithm that compares two adjacent elements and swaps them if they are not in the intended order.

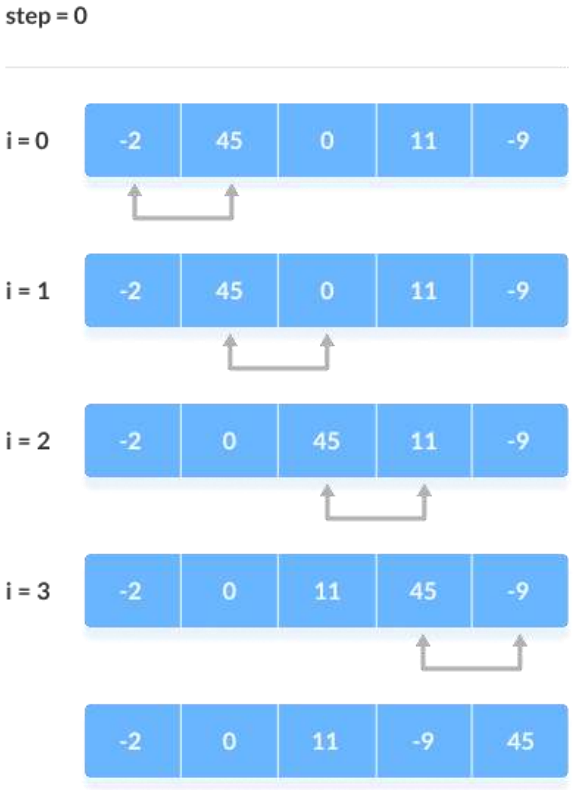
Working of Bubble Sort

- Suppose we are trying to sort the elements in **ascending order**.

1. First Iteration (Compare and Swap)

- Starting from the first index, compare the first and the second elements.
- If the first element is greater than the second element, they are swapped.
- Now, compare the second and the third elements. Swap them if they are not in order.
- The above process goes on until the last element.

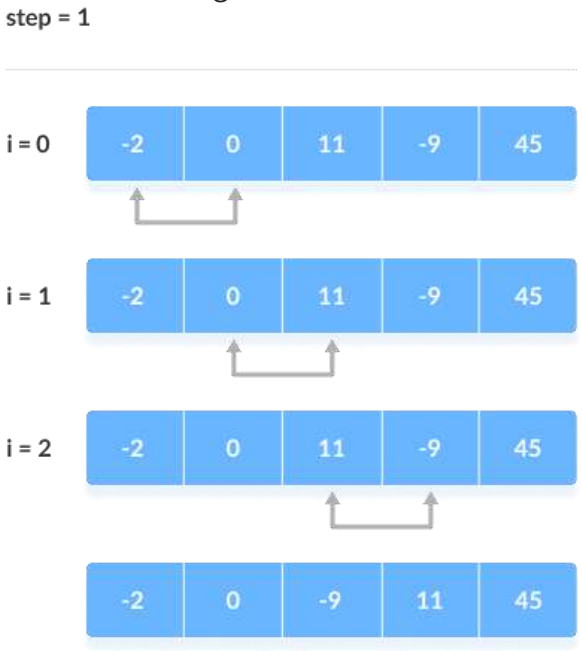
Bubble sort



Bubble sort

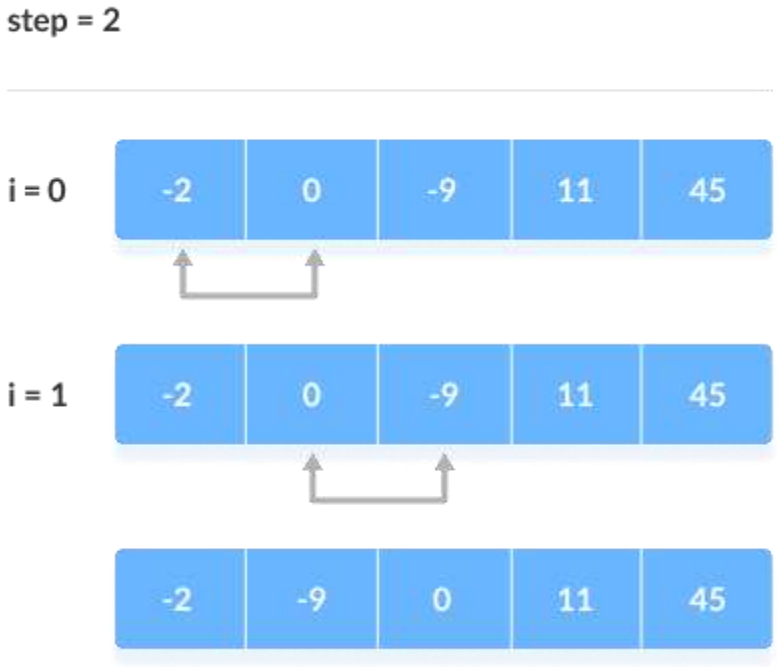
2. Remaining Iteration

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



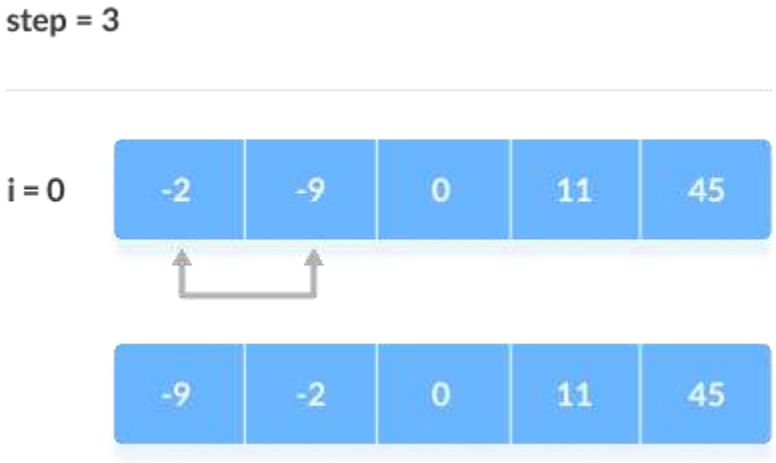
Bubble sort

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



Bubble sort

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



Bubble sort

Bubble Sort Complexity

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

Bubble sort

Complexity in Detail

- Bubble Sort compares the adjacent elements.

Cycle	Number of Comparisons
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
.....	.....
last	1

- Hence, the number of comparisons is
- (n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2nearly equals to n^2
- Hence, **Complexity:** $O(n^2)$
- Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is n*n = n^2

Bubble sort

Time Complexities

- **Worst Case Complexity:** $O(n^2)$
  If we want to sort in ascending order and the array is in descending order then the worst case occurs.
- **Best Case Complexity:** $O(n)$
  If the array is already sorted, then there is no need for sorting.
- **Average Case Complexity:** $O(n^2)$
  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

Bubble Sort Applications

Bubble sort is used if

- complexity does not matter
- short and simple code is preferred

Quick sort

- Quicksort is a sorting algorithm based on the **divide and conquer approach** where
- An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

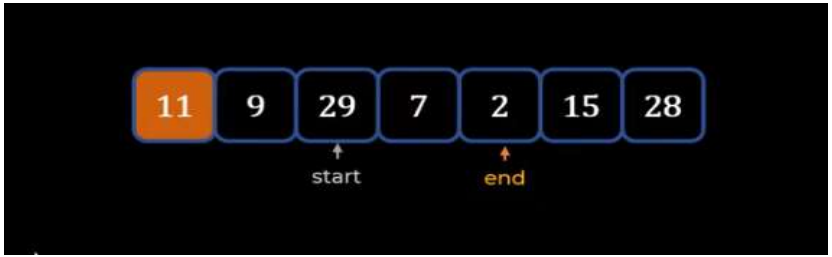
  While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
- The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
- At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Quick sort

- Two types of Partition schemes
- Hoare Partition
- Lomuto Partition

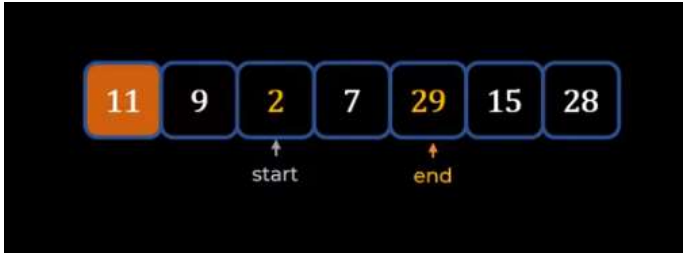
Quick sort

- Hoare Partition:
- Move the start pointer till we get an element greater than pivot value.
- Move the end pointer till we get an element less than pivot value.



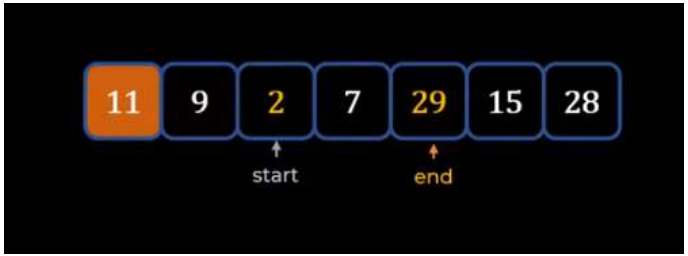
Quick sort

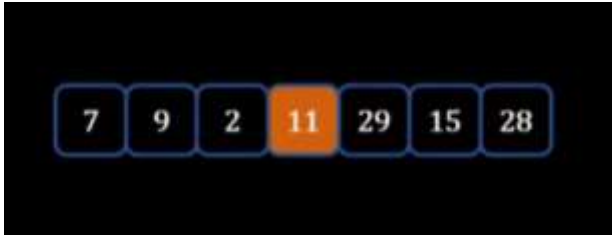
- Hoare Partition:
- Move the start pointer till we get an element greater than pivot value.
- Move the end pointer till we get an element less than pivot value.
- Now, Swap the start and end pointer



Quick sort

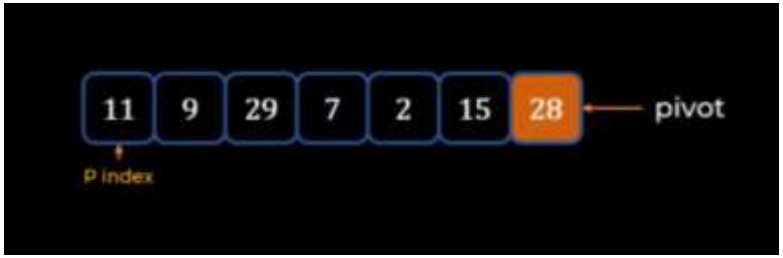
- Again repeat the same process. We stop this procedure whenever end crosses start. So, now swap end and pivot value.
- Now do the same procedure for left partition and right partition. So, then your whole list will be sorted.





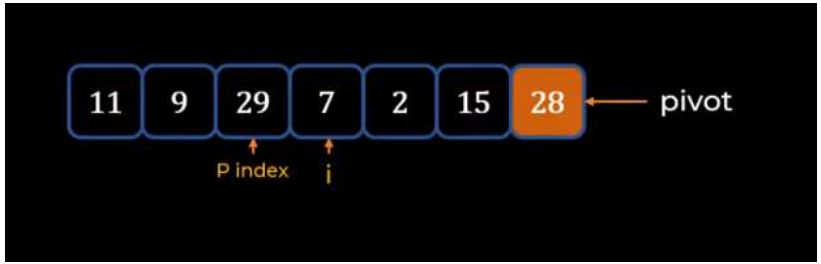
Quick sort

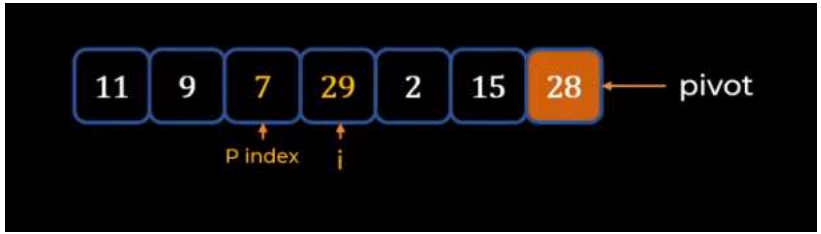
Lomuto Partition: Here we decide the end element as pivot element. We marked the first element as partition index. Keep on moving the P index till the element is greater than the pivot value.



Quick sort

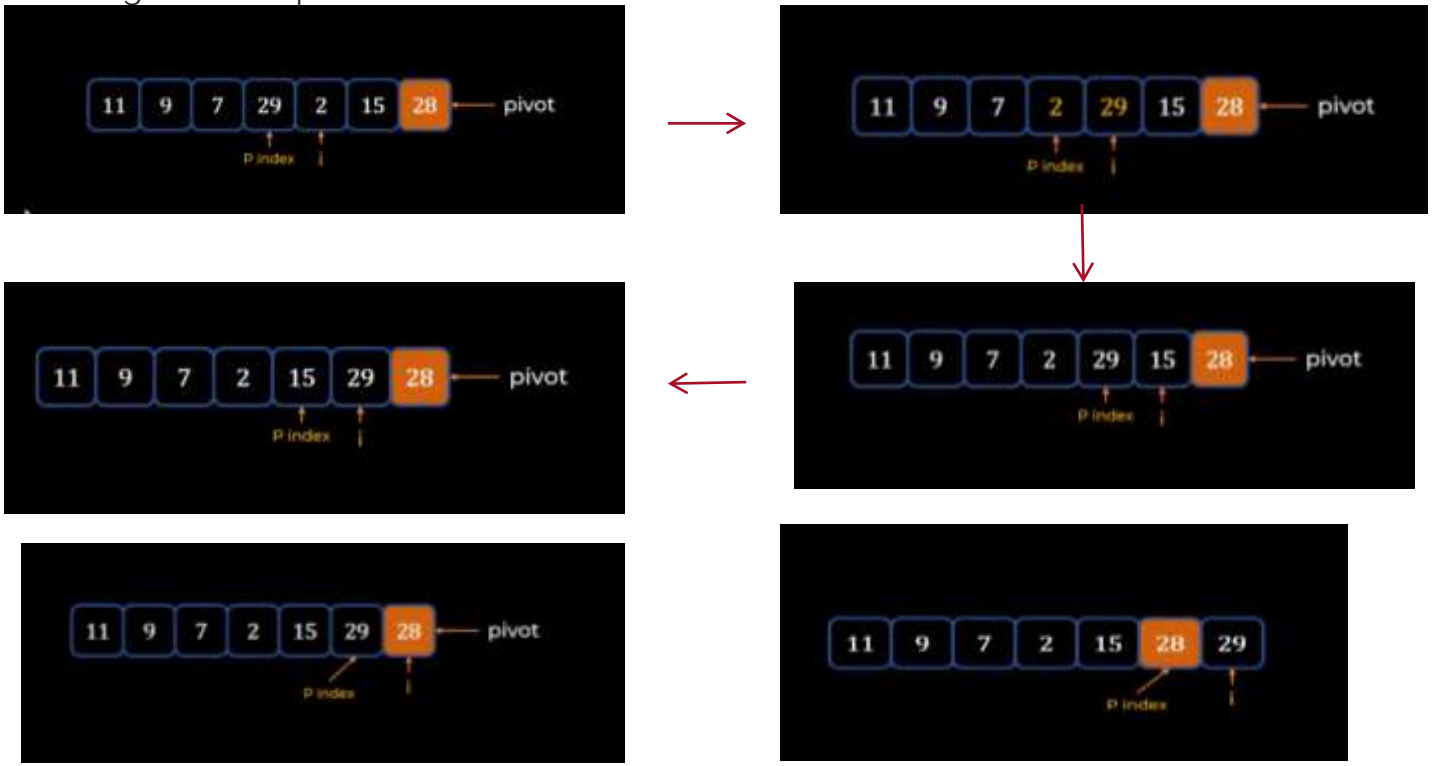
Now we take another counter i which starts at p index and keep on moving till we find an element less than pivot value. Then swap i and p index.





Quick sort

Continue doing the same procedure



Quick sort

Working of Quicksort Algorithm

- **1. Select the Pivot Element**
- There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

Quick sort

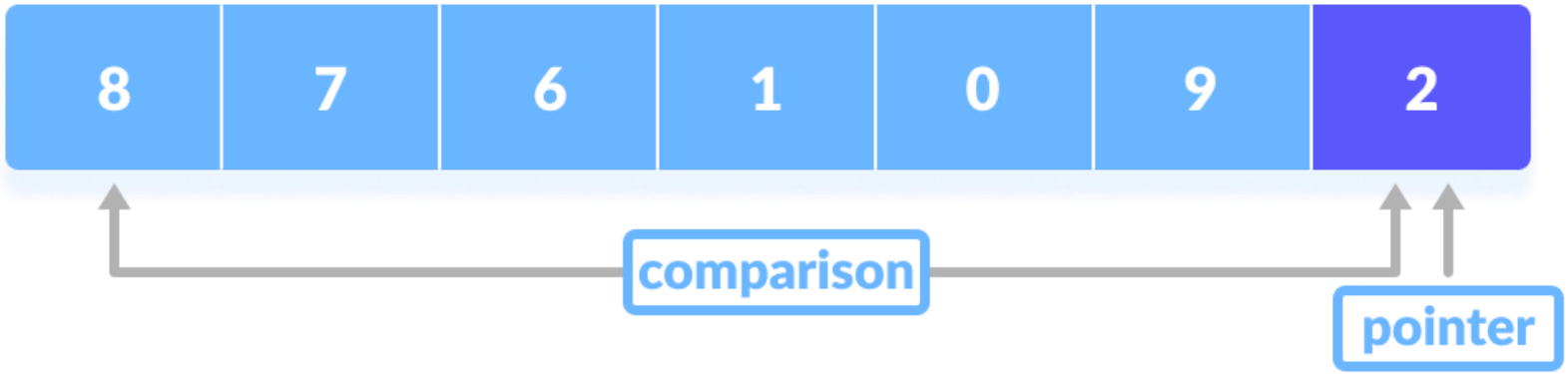
2. Rearrange the Array

- Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.
-

Quick sort

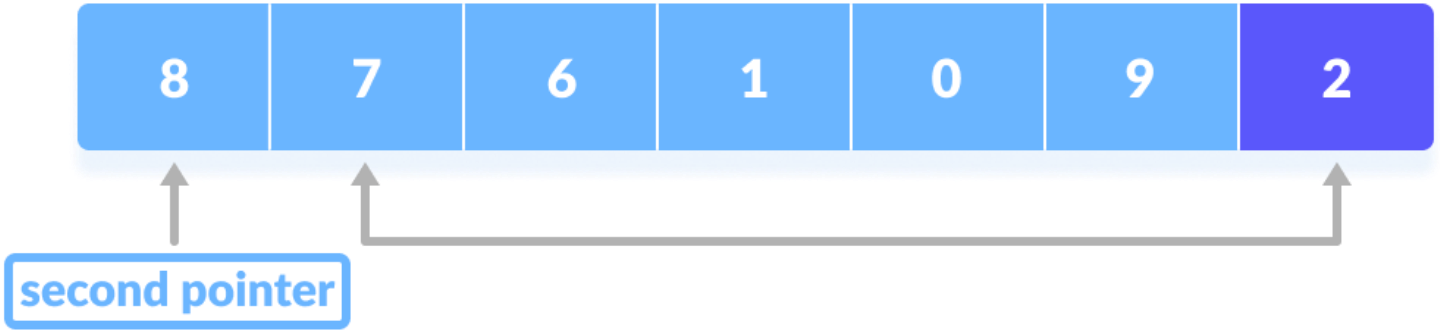
Here's how we rearrange the array:

- A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



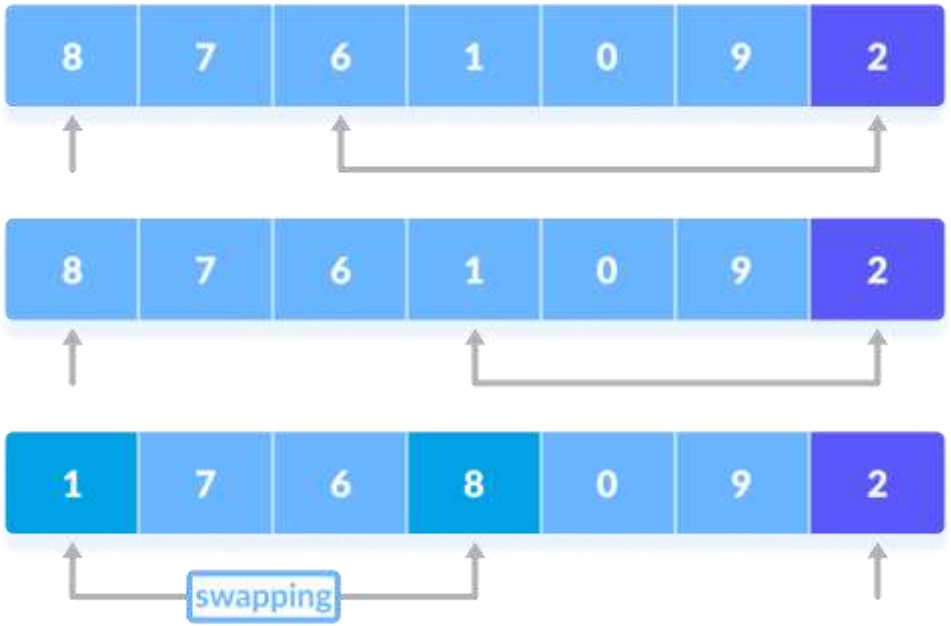
Quick sort

If the element is greater than the pivot element, a second pointer is set for that element.



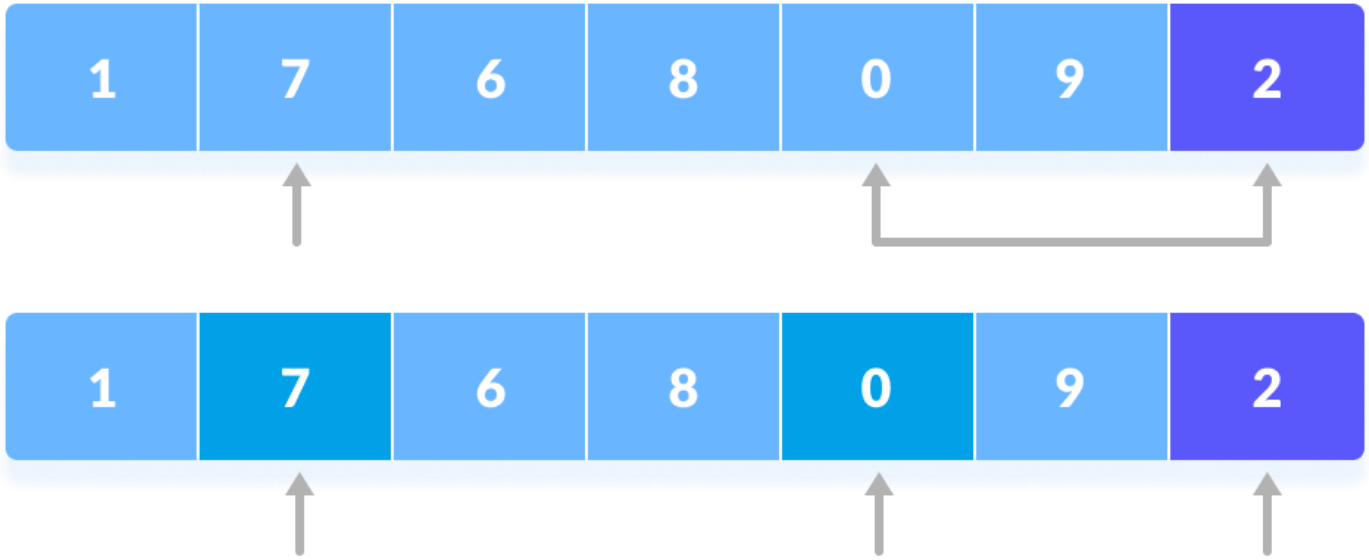
Quick sort

Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



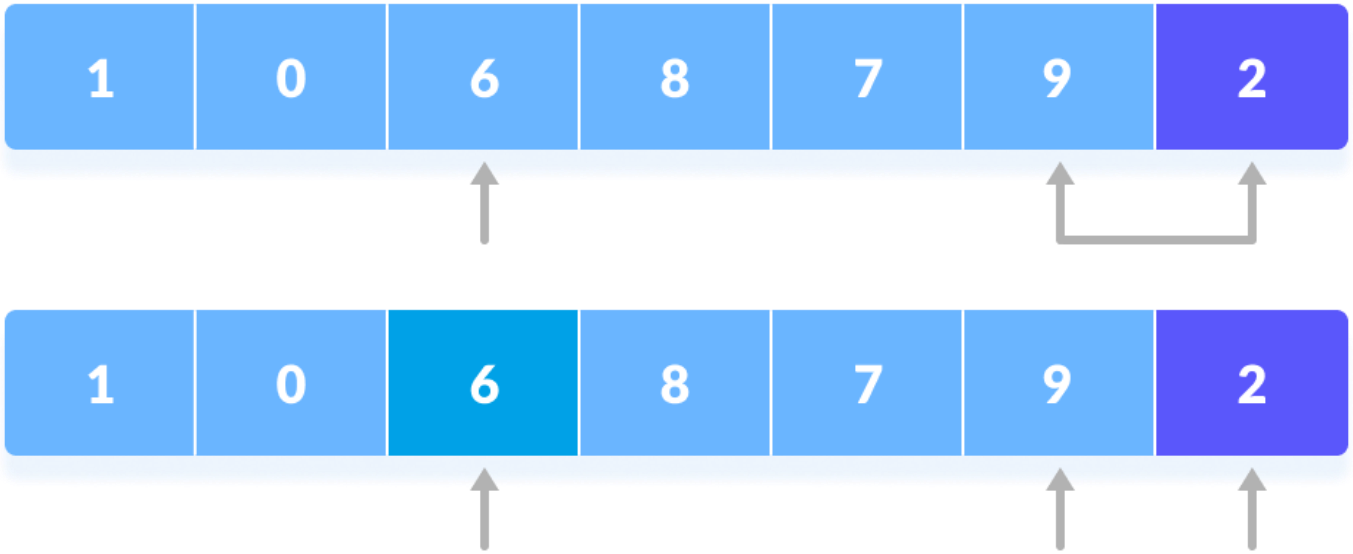
Quick sort

Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



Quick sort

The process goes on until the second last element is reached.



Quick sort

Finally, the pivot element is swapped with the second pointer.

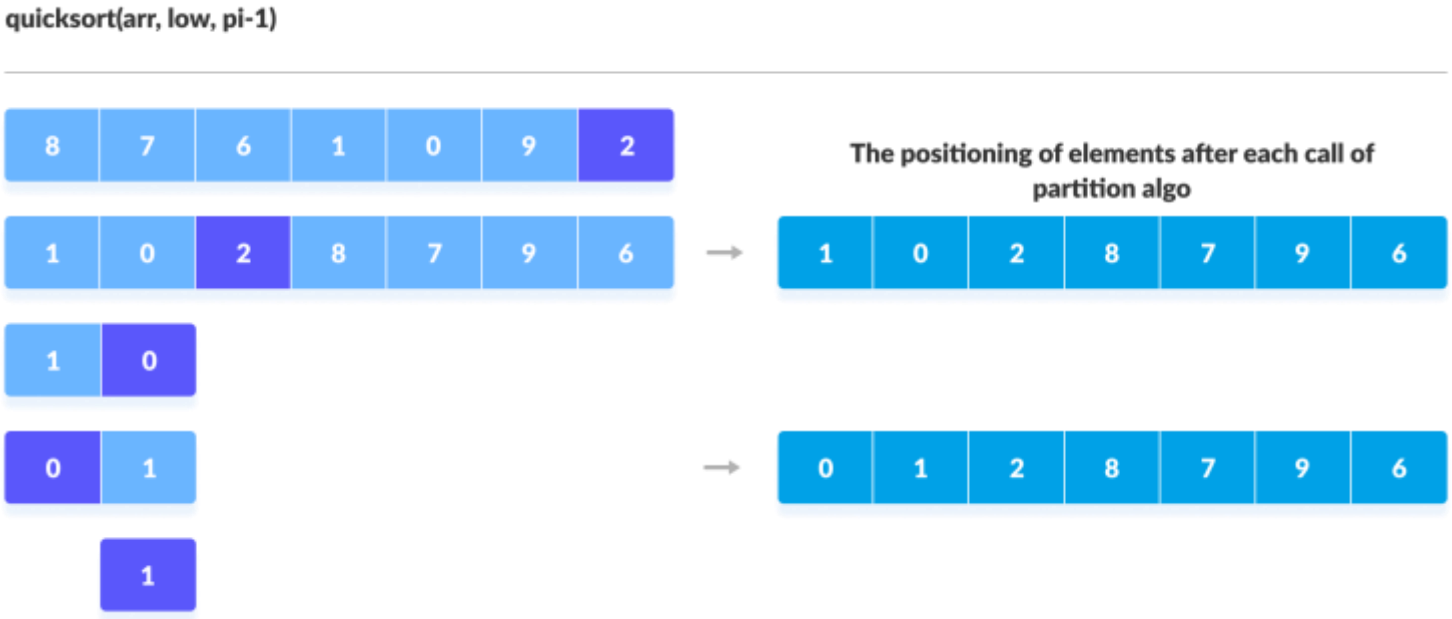
Quick sort

3. Divide Subarrays

- Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.
- The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted. quicksort(arr, pi, high)

Quick sort

You can understand the working of quicksort algorithm with the help of the illustrations below.



Quick sort

You can understand the working of quicksort algorithm with the help of the illustrations below.

Quick sort

Quicksort Complexity

Time Complexity	
Best	O(n*log n)
Worst	$O(n^2)$
Average	O(n*log n)
Stability	No

Quick sort

Time Complexities

- **Worst Case Complexity [Big-O]**: $O(n^2)$
  It occurs when the pivot element picked is either the greatest or the smallest element.

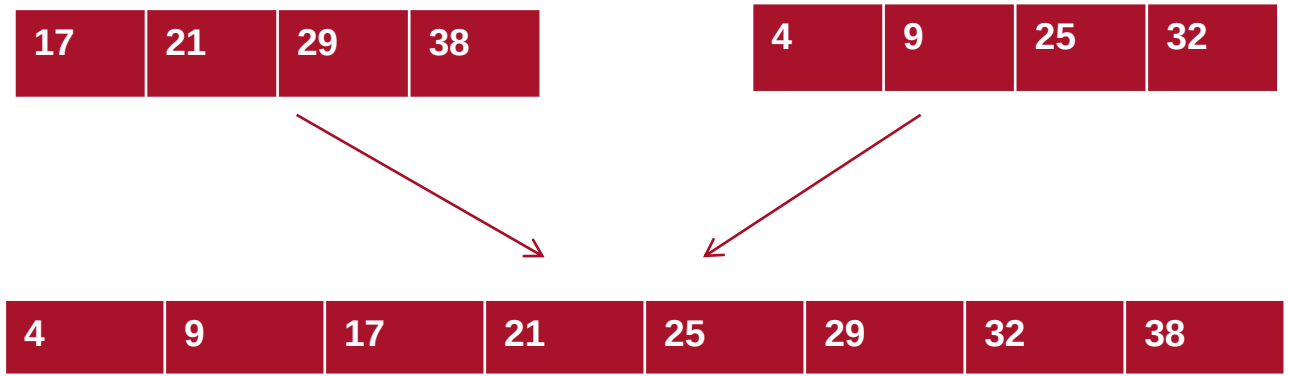
  This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.

  However, the quicksort algorithm has better performance for scattered pivots.

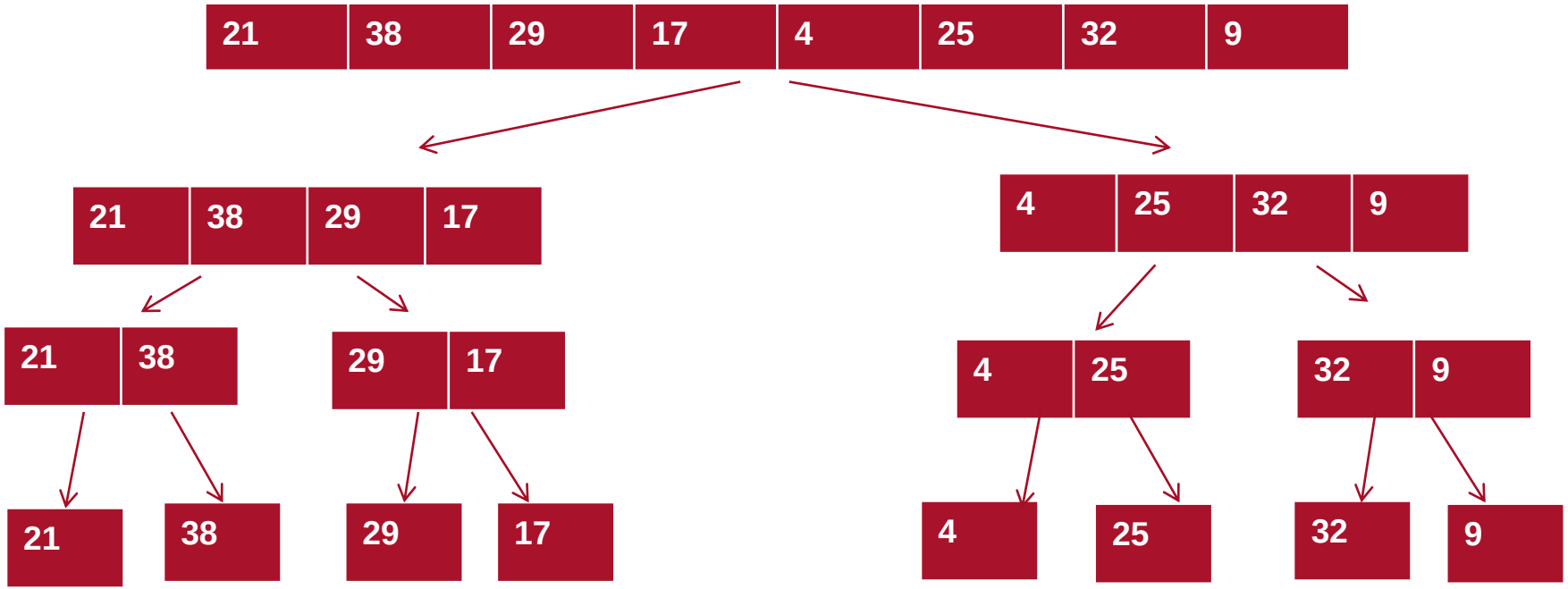
- **Best Case Complexity [Big-omega]**: O(n*log n)
  It occurs when the pivot element is always the middle element or near to the middle element.

- **Average Case Complexity [Big-theta]**: O(n*log n)
  It occurs when the above conditions do not occur.

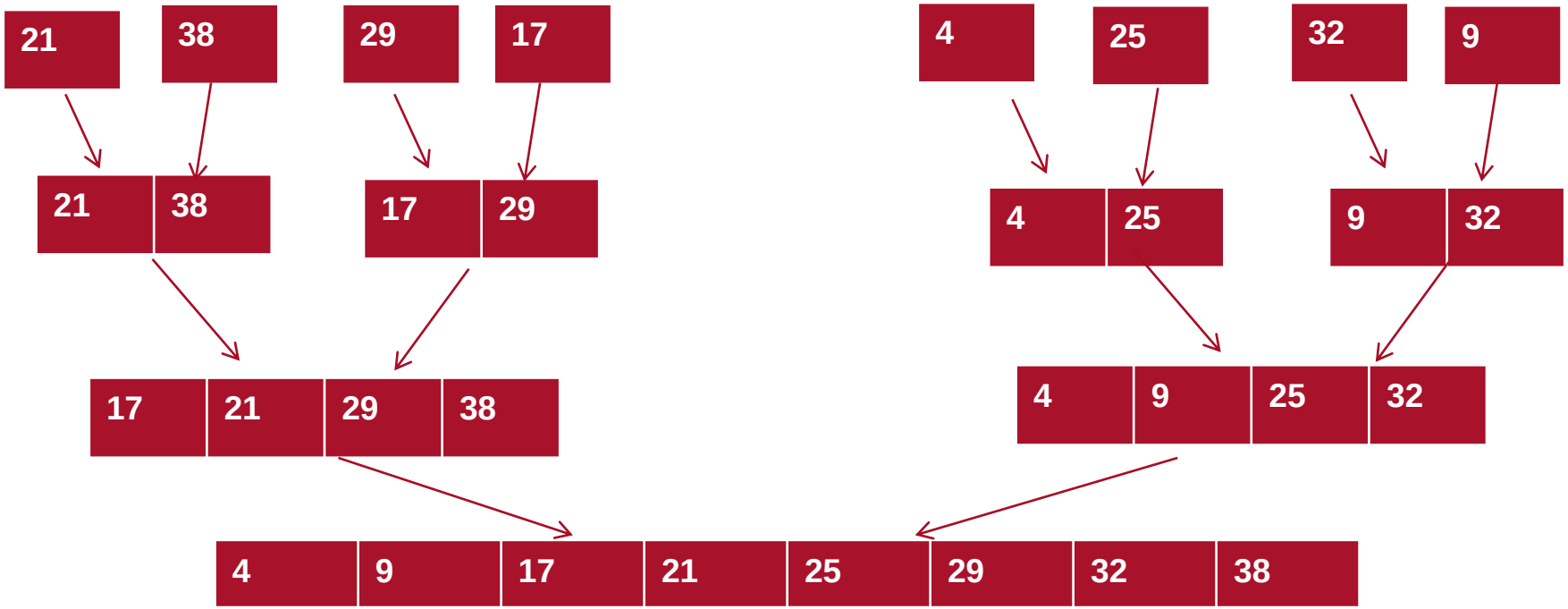
Merge sort



Merge sort


21 38 29 17 4 25 32 9
21 38 29 17
4 25 32 9
21 38
29 17
4 25
32 9
21
38
29
17
4
25
32
9

Merge sort


21
38
29
17
4
25
32
9
21
38
17
29
4
25
9
32
17
21
29
38
4
9
25
32
4
9
17
21
25
29
32
38

Thank You