Advance Sorting Algorithm(s)

Bubble sort

Insertion sort

Merge sort

Quick sort

Selection sort

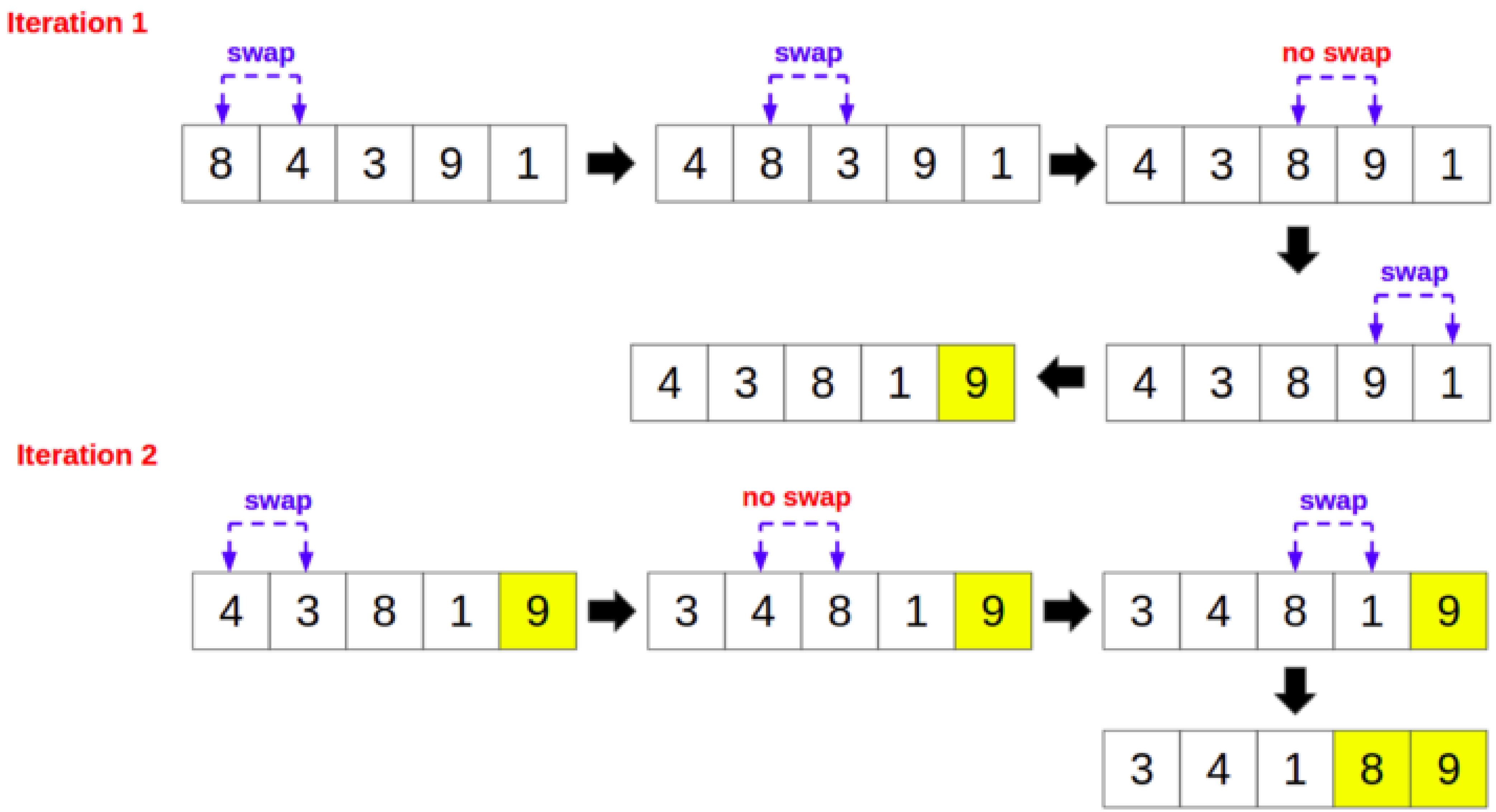
Shoaib Rauf

Bubble Sort

Algorithm 1: Bubble sort

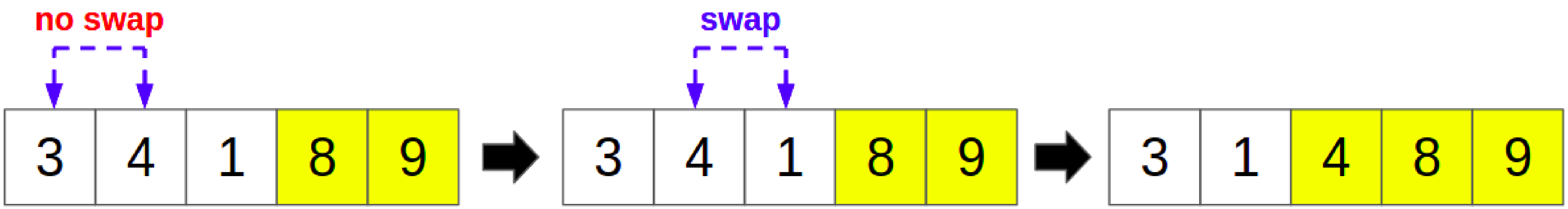
Data: Input array *A[]*
Result: Sorted *A[]*

```
int i, j, k;
N = length(A);
for j = 1 to N do
    for i = 0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
        end
    end
end
```

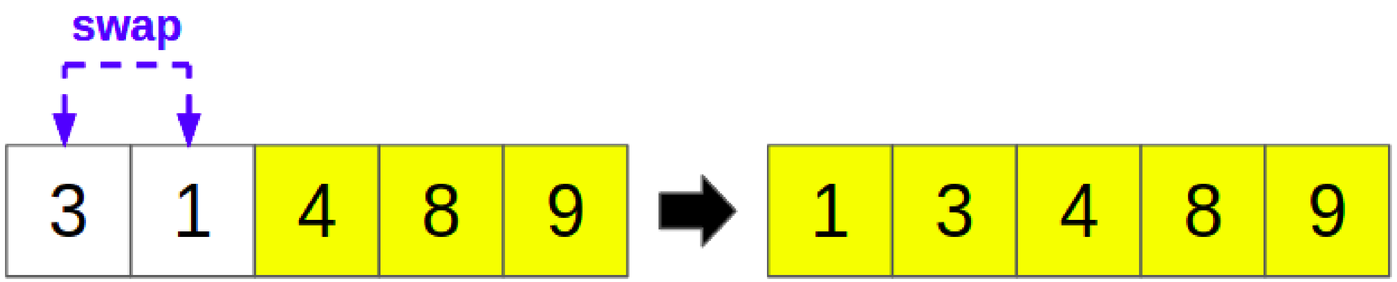


Iteration 1
swap
8 4 3 9 1
swap
4 8 3 9 1
no swap
4 3 8 9 1
swap
4 3 8 9 1
4 3 8 1 9
Iteration 2
swap
4 3 8 1 9
no swap
3 4 8 1 9
swap
3 4 8 1 9
3 4 1 8 9

Iteration 3



Iteration 4



Time Complexity (Bubble Sort)

- In traditional bubble sort algorithm:
  - The Best case and Worst case scenario, time complexity is $O(n)^2$.
  - No flag variable is used to in the outer loop to determine for no of swaps

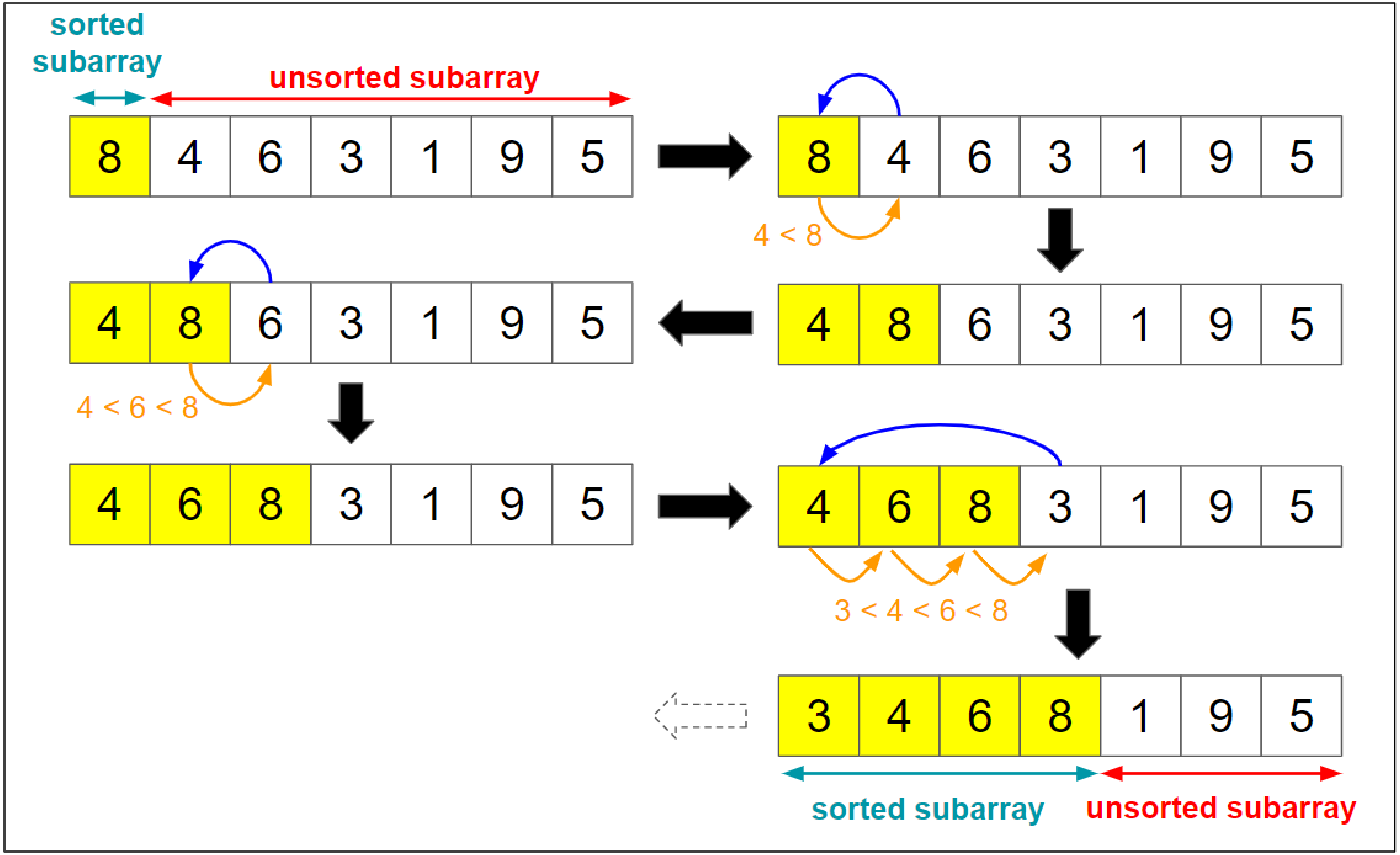
- In an optimized bubble sort algorithm:
  - If the numbers are already sorted in ascending order, the algorithm will determine in the first iteration that no number pairs need to be swapped (flag variable is used)and will then terminate immediately.

  - Best case Scenario: $O(n)$
  - Worst case Scenario: $O(n)^2$

Insertion Sort

ALGORITHM *InsertionSort*$(A[0..n-1])$

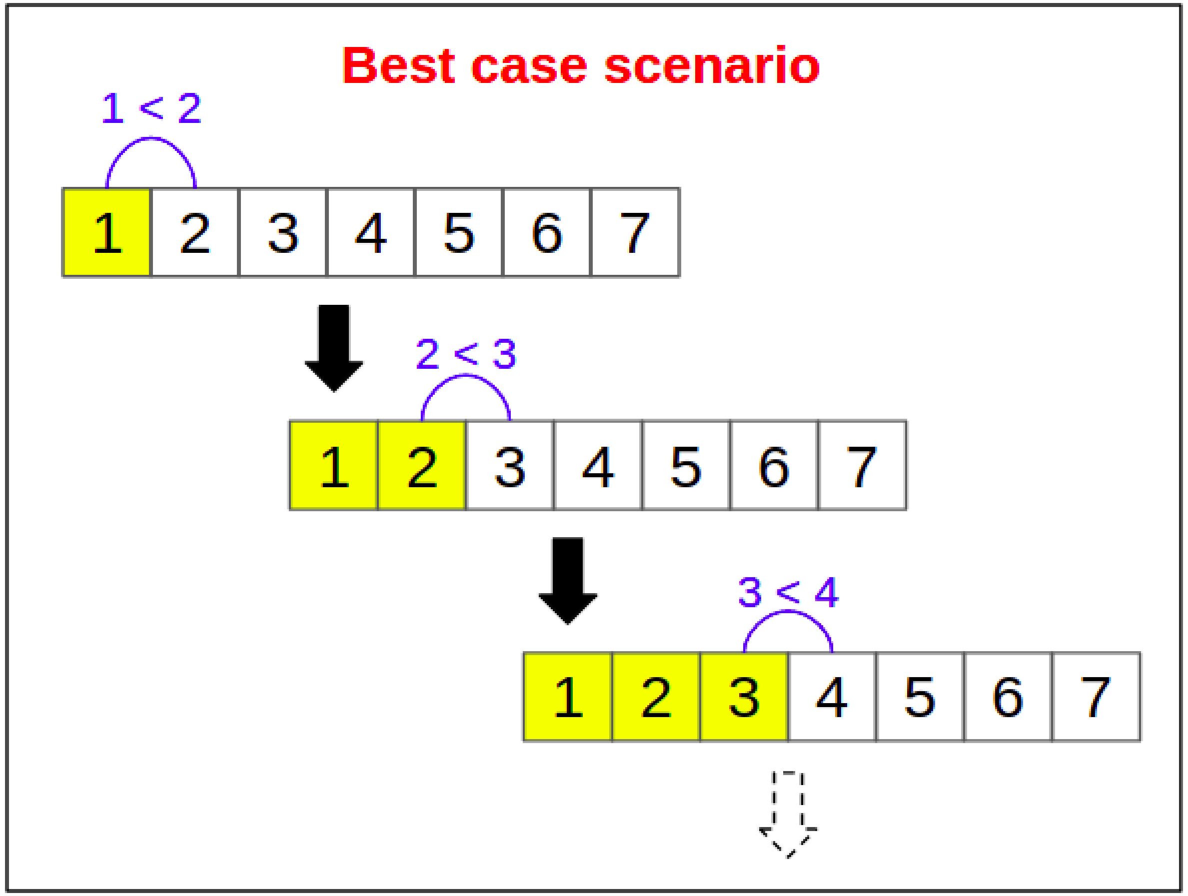
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$



sorted subarray
unsorted subarray
8 4 6 3 1 9 5
8 4 6 3 1 9 5
4 < 8
4 8 6 3 1 9 5
4 8 6 3 1 9 5
4 < 6 < 8
4 6 8 3 1 9 5
4 6 8 3 1 9 5
3 < 4 < 6 < 8
3 4 6 8 1 9 5
sorted subarray
unsorted subarray

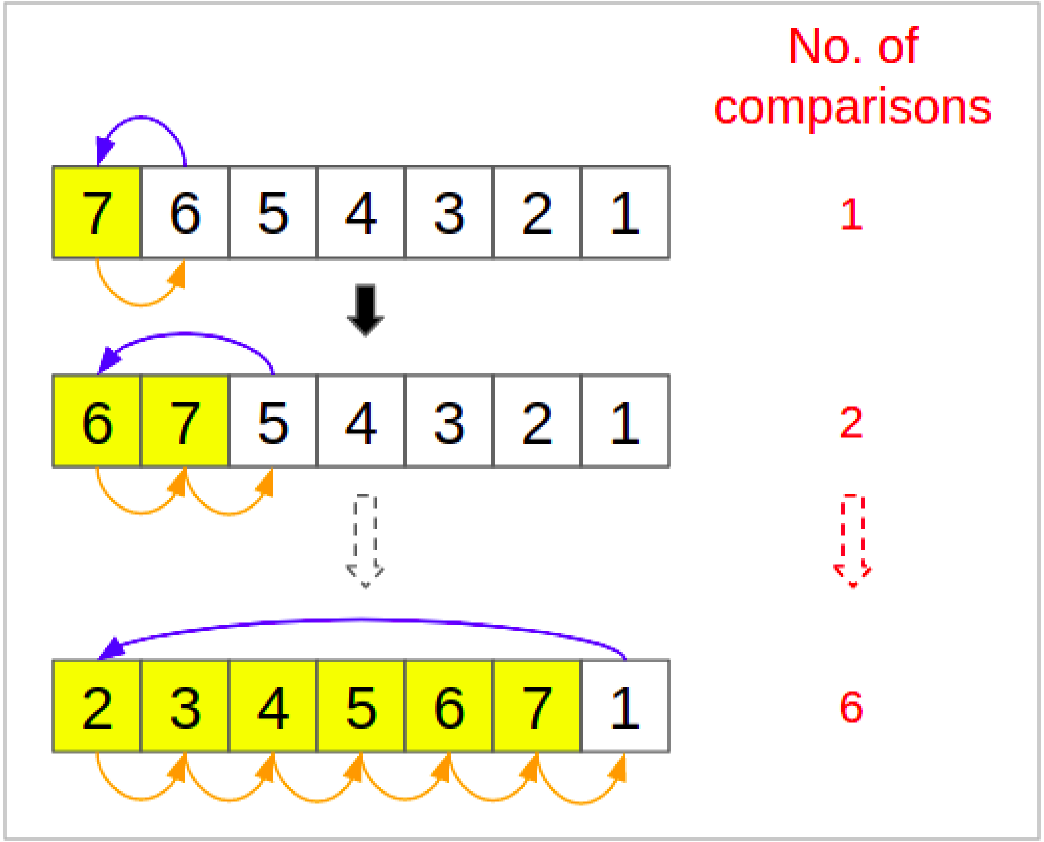
Time Complexity (Insertion Sort)

- The **best-case** time complexity of insertion sort is **O(n)**. When the array is already sorted (which is the best case), insertion sort has to perform only one comparison in each iteration



Time Complexity (Insertion Sort)

- The **worst-case complexity is O(n^2)**. When the array is sorted in reverse order (which is the worst case), we have to perform *i* number of comparisons in the i^{th} iteration



Merge Sort

- Merge sort algorithm uses the “**divide and conquer**” strategy wherein we divide the problem into subproblems and solve those subproblems individually.
- These subproblems are then combined or merged together to form a unified solution.

Worst case time complexity	O(n*log n)
Best case time complexity	O(n*log n)

Merge Sort: Pseudocode

Declare an array Arr of length N

If N=1, Arr is already sorted

If N>1,

Left = 0, right = N-1

Find middle = (left + right)/2

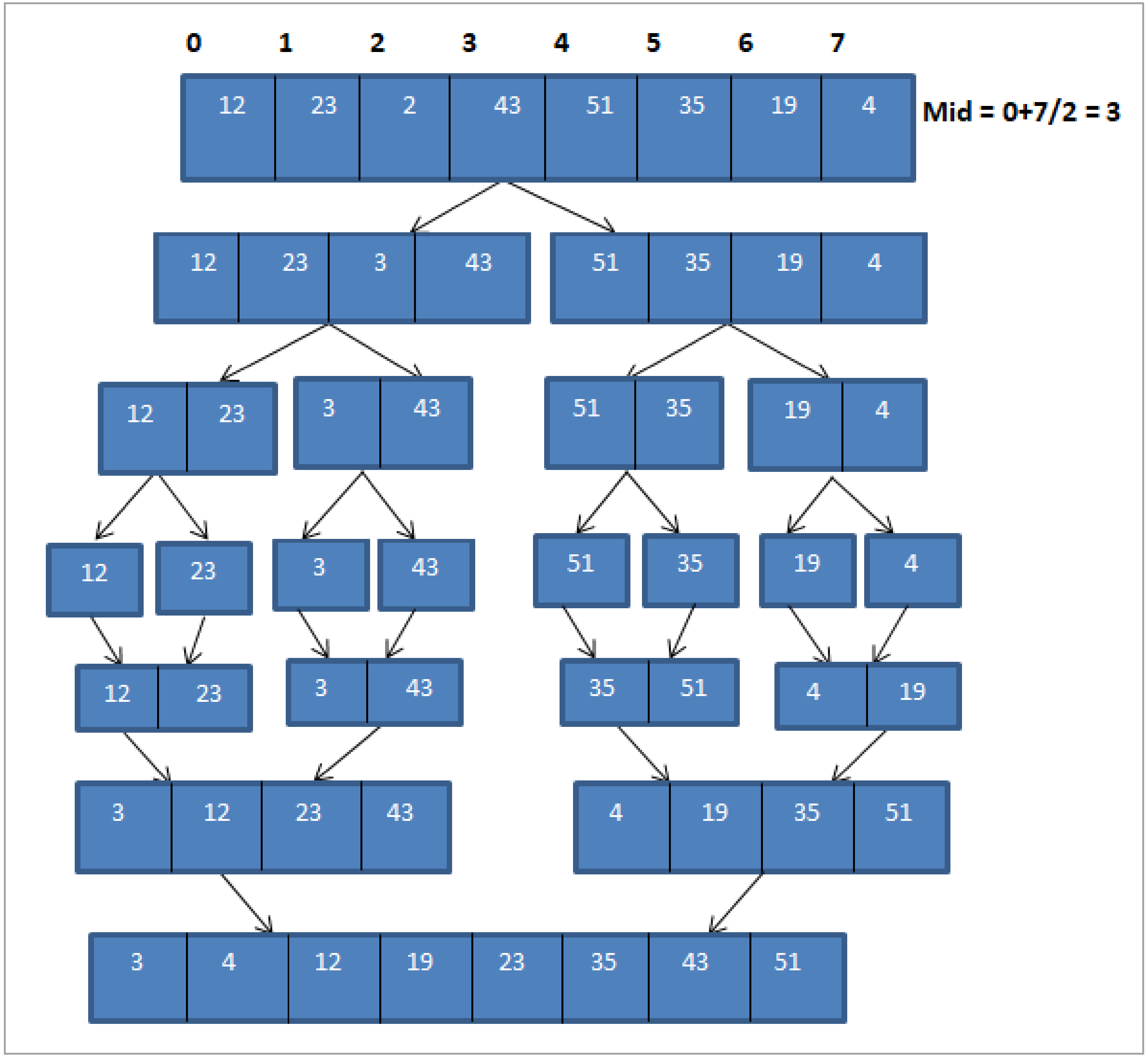
Call merge_sort(Arr,left,middle) =>sort first half recursively

Call merge_sort(Arr,middle+1,right) => sort second half recursively

Call merge(Arr, left, middle, right) to merge sorted arrays in above steps.

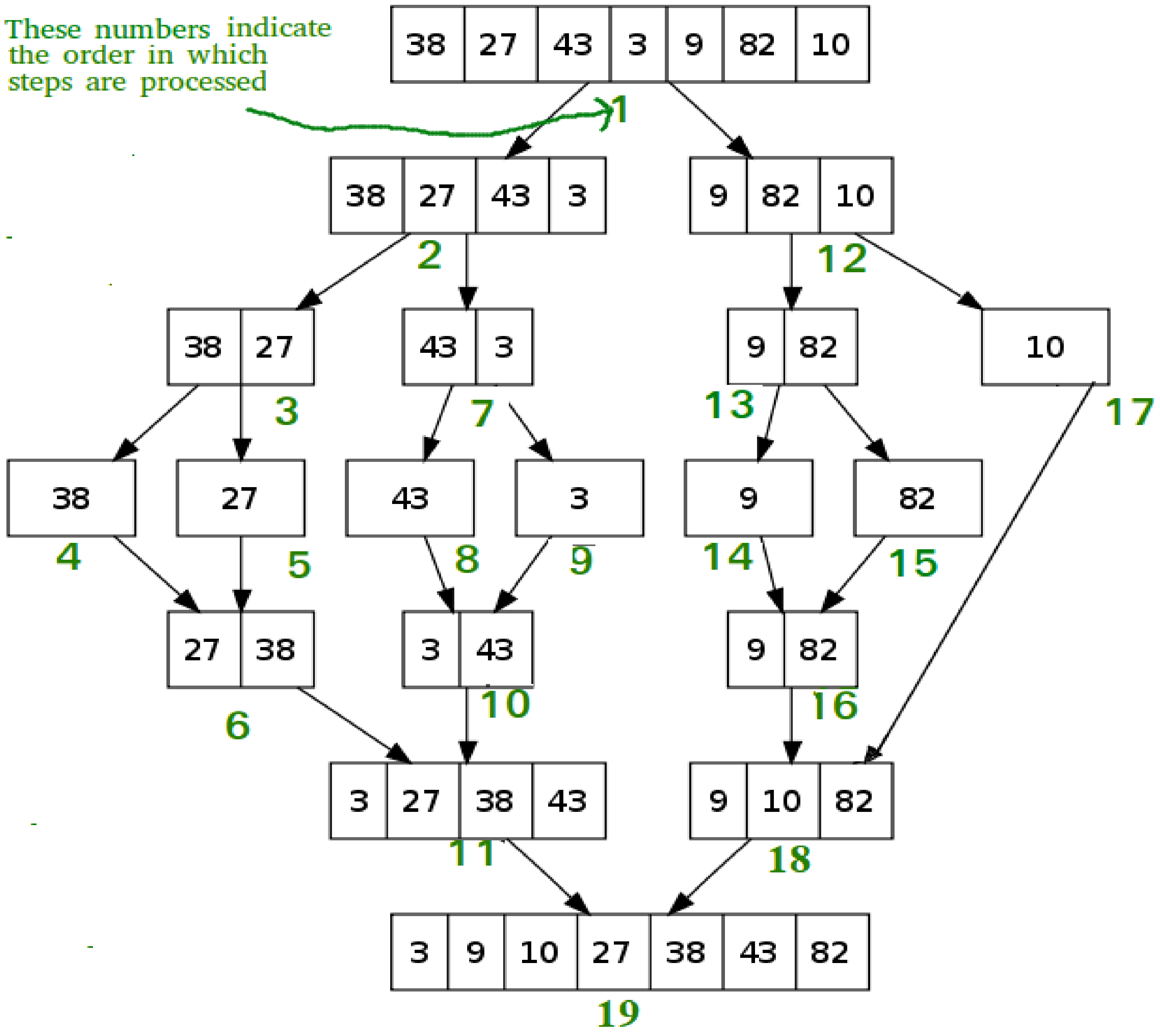
Exit

Merge Sort



0 1 2 3 4 5 6 7
12 23 2 43 51 35 19 4
Mid = 0+7/2 = 3
12 23 3 43
51 35 19 4
12 23
3 43
51 35
19 4
12
23
3
43
51
35
19
4
12 23
3 43
35 51
4 19
3 12 23 43
4 19 35 51
3 4 12 19 23 35 43 51

Pass	Unsorted list	divide	Sorted list
1	{12, 23,2,43,51,35,19,4 }	{12,23,2,43} {51,35,19,4}	{}
2	{12,23,2,43} {51,35,19,4}	{12,23}{2,43} {51,35}{19,4}	{}
3	{12,23}{2,43} {51,35}{19,4}	{12,23} {2,43} {35,51}{4,19}	{12,23} {2,43} {35,51}{4,19}
4	{12,23} {2,43} {35,51}{4,19}	{2,12,23,43} {4,19,35,51}	{2,12,23,43} {4,19,35,51}
5	{2,12,23,43} {4,19,35,51}	{2,4,12,19,23,35,43,51}	{2,4,12,19,23,35,43,51}
6	{}	{}	{2,4,12,19,23,35,43,51}



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
4
5
8
9
14
15
27 38
3 43
9 82
6
10
16
3 27 38 43
9 10 82
11
18
3 9 10 27 38 43 82
19

Quick Sort

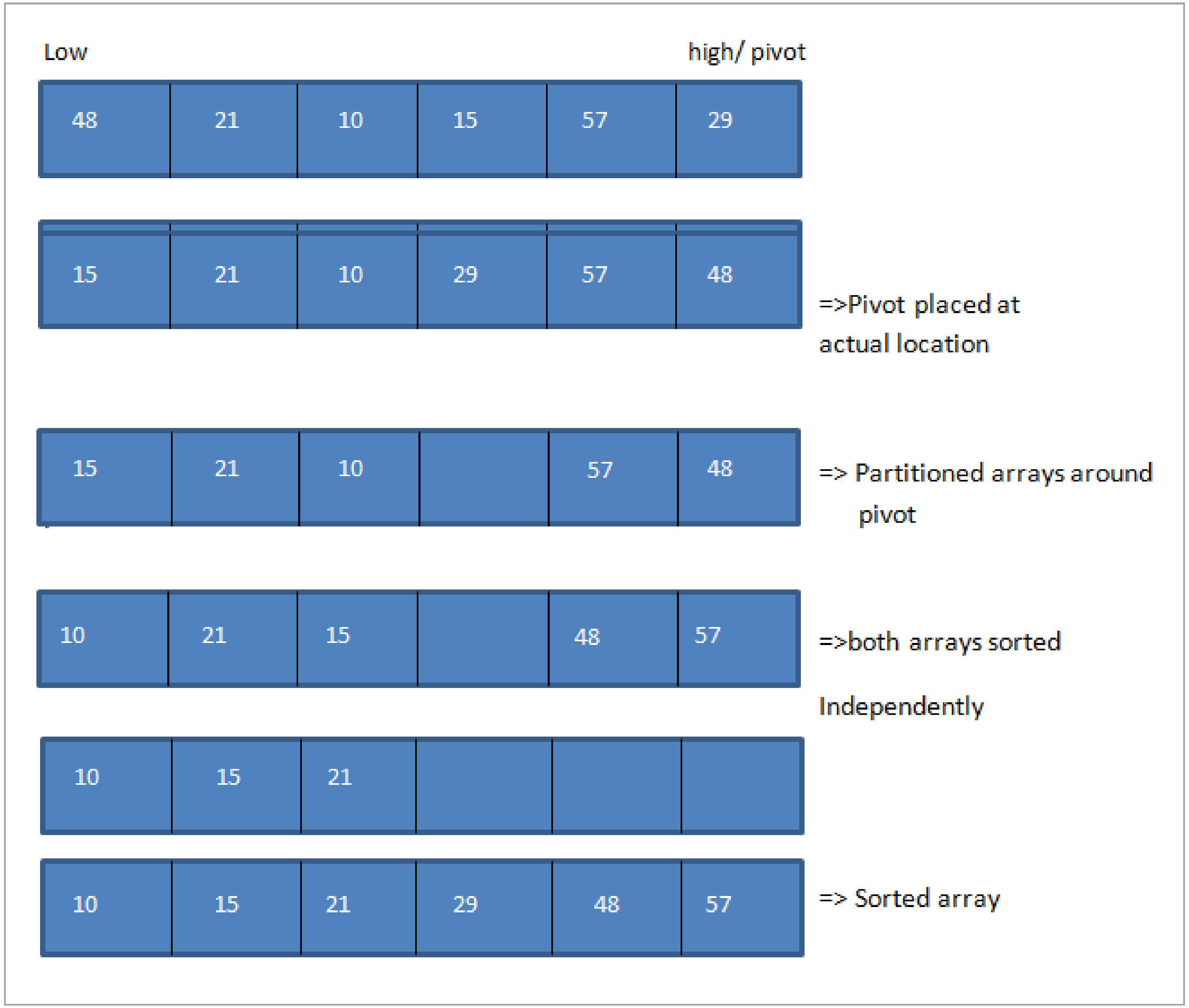
- Quicksort works efficiently as well as faster even for larger arrays or lists.
- Quicksort is a widely used sorting algorithm which selects a specific element called “pivot” and partitions the array or list to be sorted into two parts based on this pivot s0 that the elements lesser than the pivot are to the left of the list and the elements greater than the pivot are to the right of the list.
- Thus the list is partitioned into two subsist. The subsists may not be necessary for the same size. Then Quicksort calls itself recursively to sort these two subsists.

Worst case time complexity	O(n 2)
Best case time complexity	O(n*log n)

Quick Sort: Pseudocode

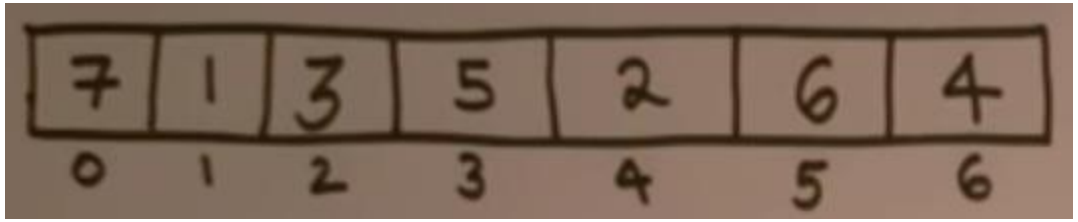
```
quicksort(A, low, high)
begin
Declare array A[N] to be sorted
    low = 1st element; high = last element; pivot
if(low < high)
    begin
    pivot = partition (A,low,high);
    quicksort(A,low,pivot-1)
    quicksort(A,pivot+1,high)
    End
end
```

```
// Sorts an array arr[low..high] using randomized quick sort
randomQuickSort(array[], low, high)
    array – array to be sorted
    low – lowest element in array
    high – highest element in array
begin
    1. If low >= high, then EXIT.
    //select central pivot
    2. While pivot 'pi' is not a Central Pivot.
         (i)   Choose uniformly at random a number from [low..high].
                Let pi be the randomly picked number.
         (ii)  Count elements in array[low..high] that are smaller
            than array[pi]. Let this count be a_low.
         (iii) Count elements in array[low..high] that are greater
              than array[pi]. Let this count be a_high.
         (iv)  Let n = (high-low+1). If a_low >= n/4 and
            a_high >= n/4, then pi is a central pivot.
//partition the array
3. Partition array[low..high] around the pivot pi.
4. // sort first half
   randomQuickSort(array, low, a_low-1)
5. // sort second half
   randomQuickSort(array, high-a_high+1, high)
end procedure
```

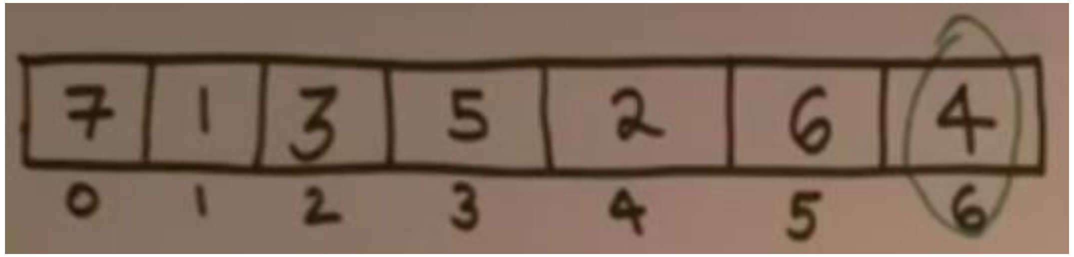

Low
high/ pivot
48
21
10
15
57
29
15
21
10
29
57
48
=>Pivot placed at actual location
15
21
10
57
48
=> Partitioned arrays around pivot
10
21
15
48
57
=>both arrays sorted
Independently
10
15
21
10
15
21
29
48
57
=> Sorted array

Selection of Pivot

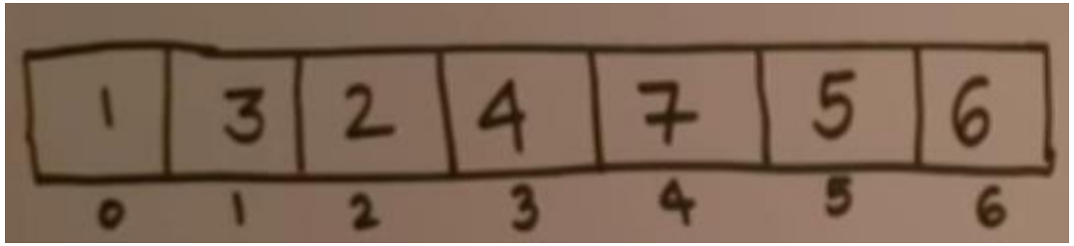
- Unsorted array



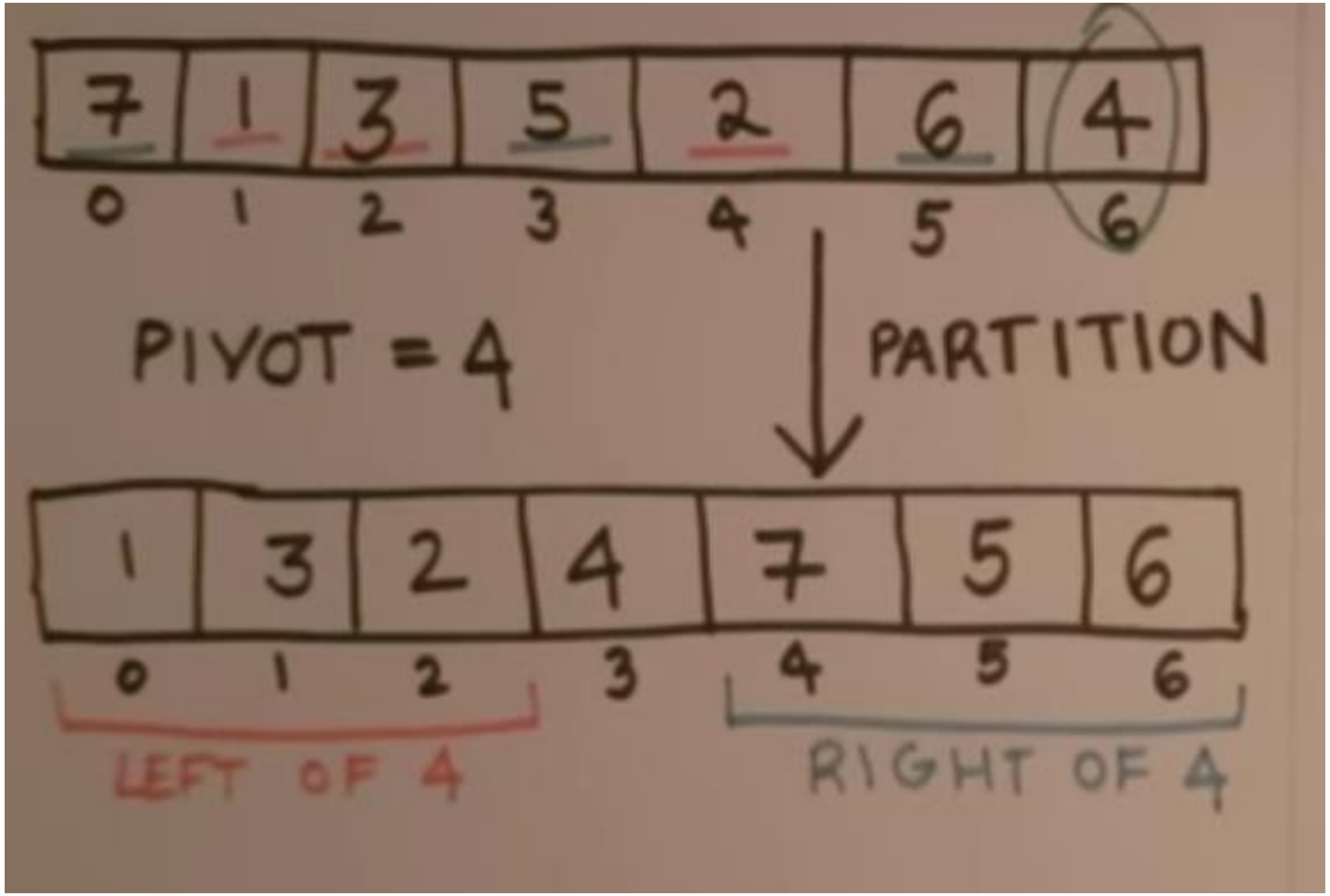
- Selection of pivot



- partition of array

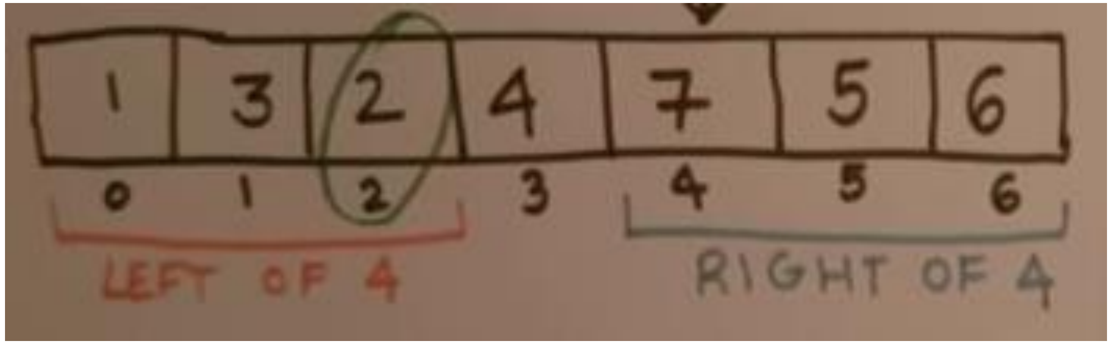


Partition of Array

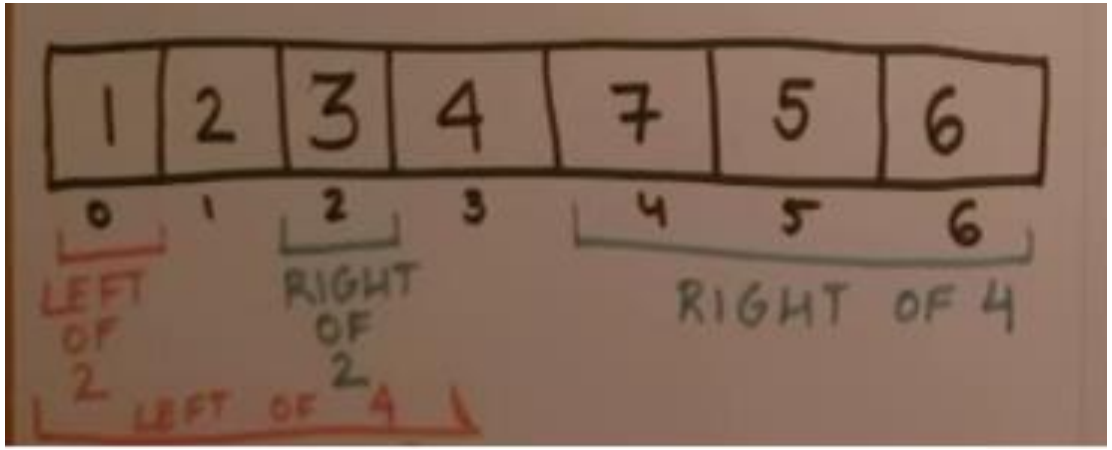


Sorting the Left part of array or Pivot

- Selection of pivot

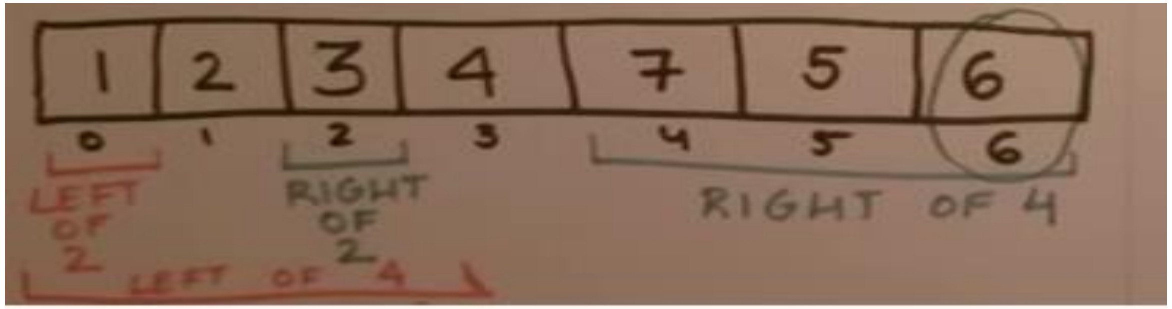


- partition of array

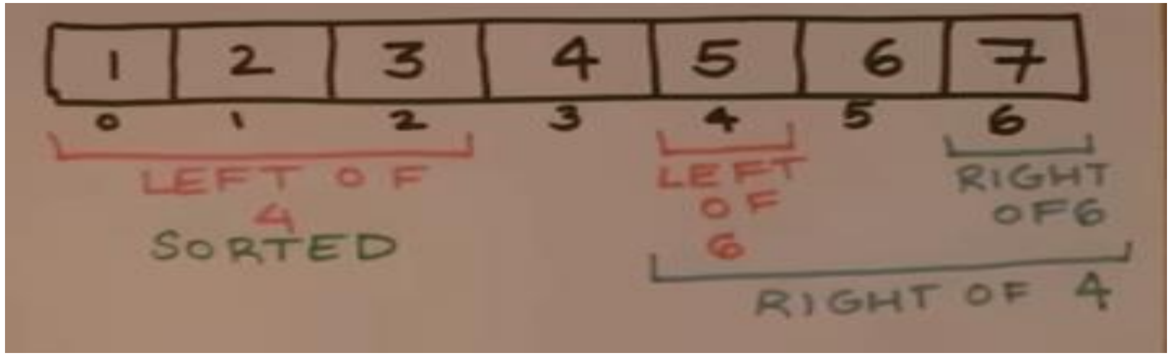


Sorting the Right part of array or Pivot

- Selection of pivot

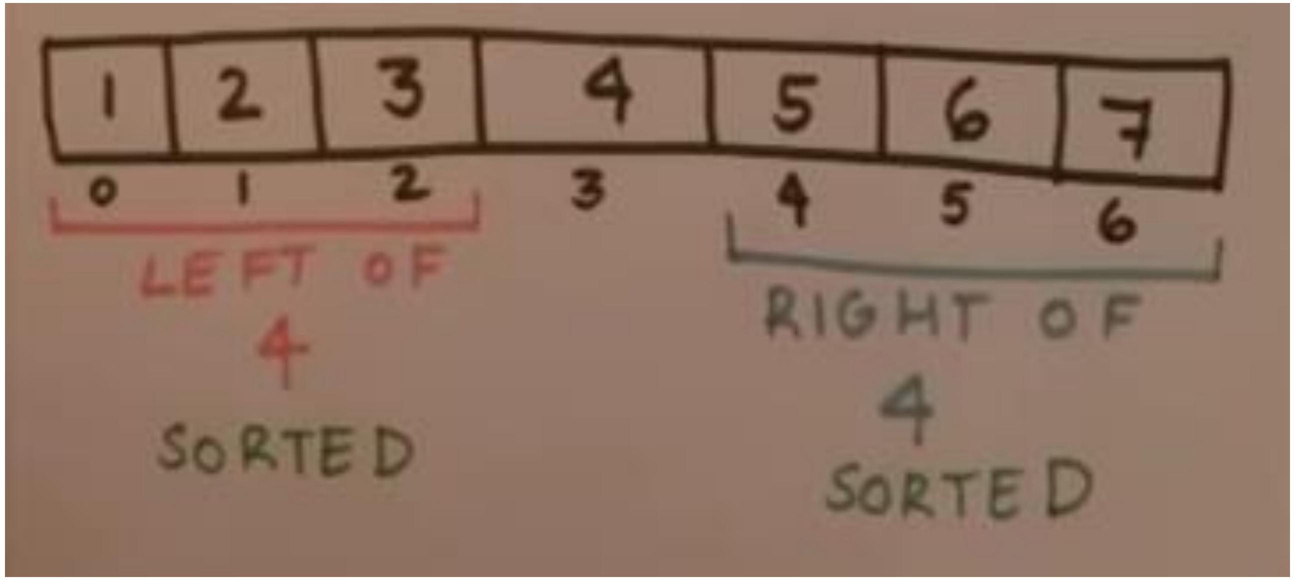


- partition of array



Quick sort

- Sorted Array



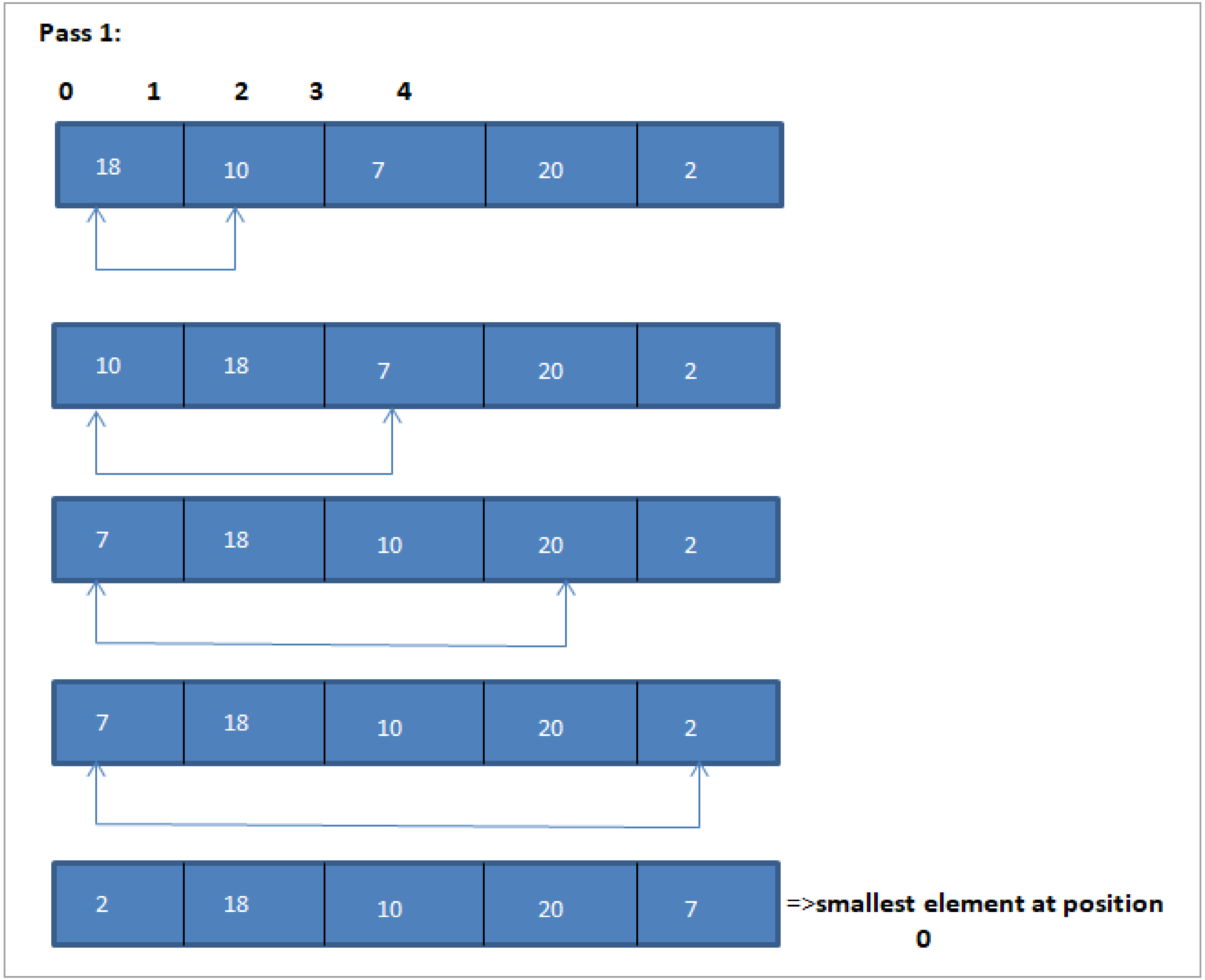
Selection sort

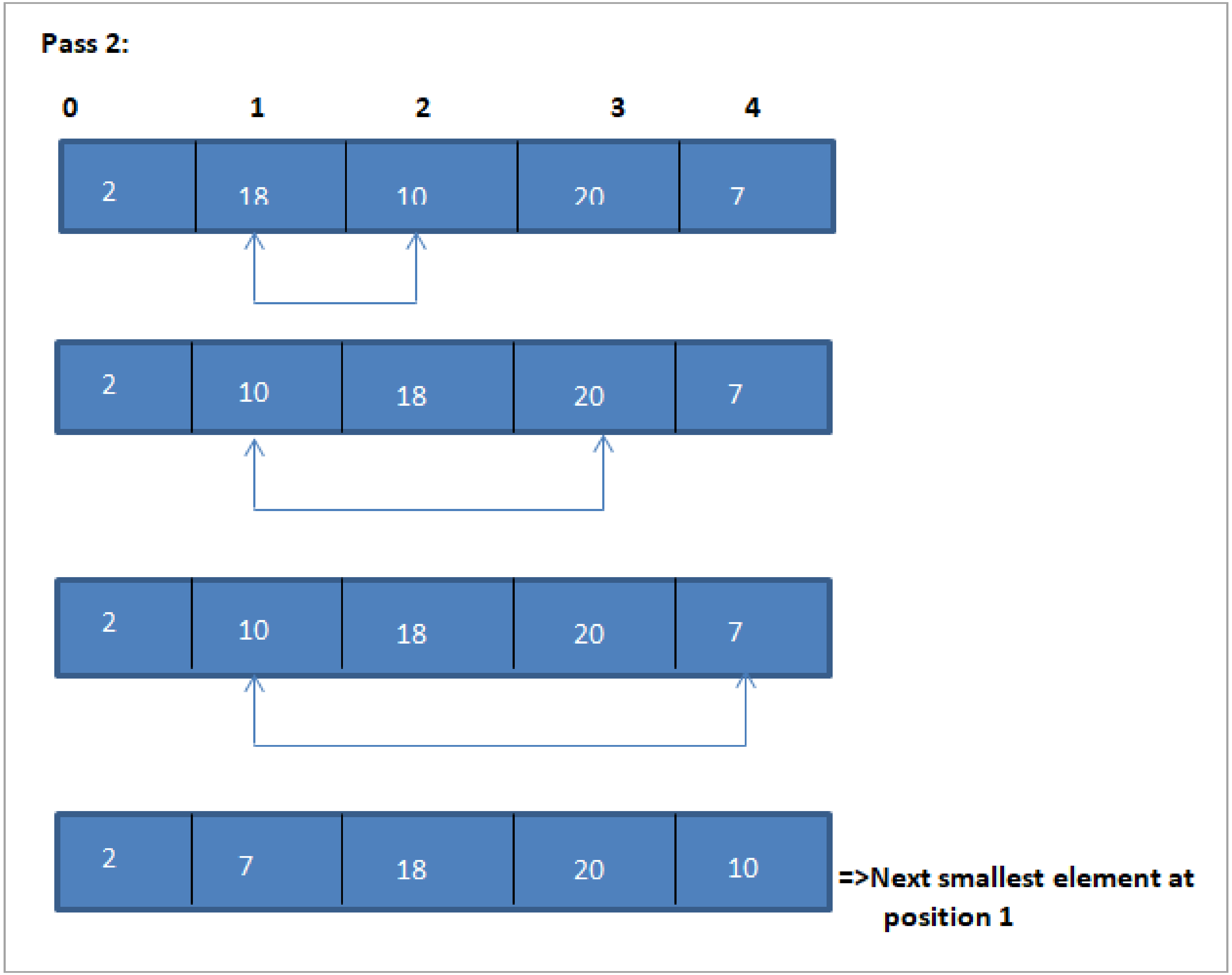
- Selection sort is quite a straightforward sorting technique as the technique only involves finding the smallest element in every pass and placing it in the correct position.
- Selection sort works efficiently when the list to be sorted is of small size but its performance is affected badly as the list to be sorted grows in size.
- Hence we can say that selection sort is not advisable for larger lists of data.

Worst case time complexity	O(n 2)
Best case time complexity	O(n 2)

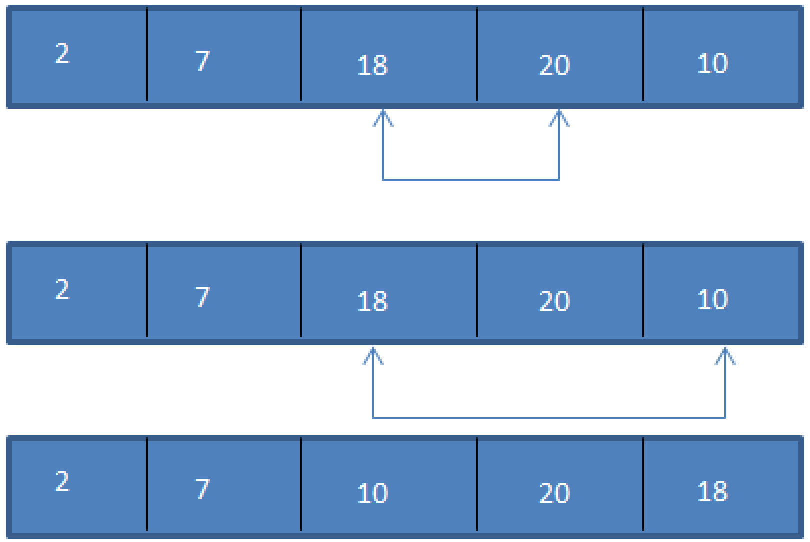
Pseudocode For Selection Sort

```
Procedure selection_sort(array,N)
    array – array of items to be sorted
    N – size of array
begin
    for I = 1 to N-1
    begin
        set min  = i
        for j = i+1 to N
        begin
            if array[j] < array[min] then
                min = j;
            end if
        end for
        //swap the minimum element with current element
        if minIndex != I then
            swap array[min[] and array[i]
        end if
    end for
end procedure
```


Pass 1:
0 1 2 3 4
18 10 7 20 2
10 18 7 20 2
7 18 10 20 2
7 18 10 20 2
2 18 10 20 7
=>smallest element at position 0

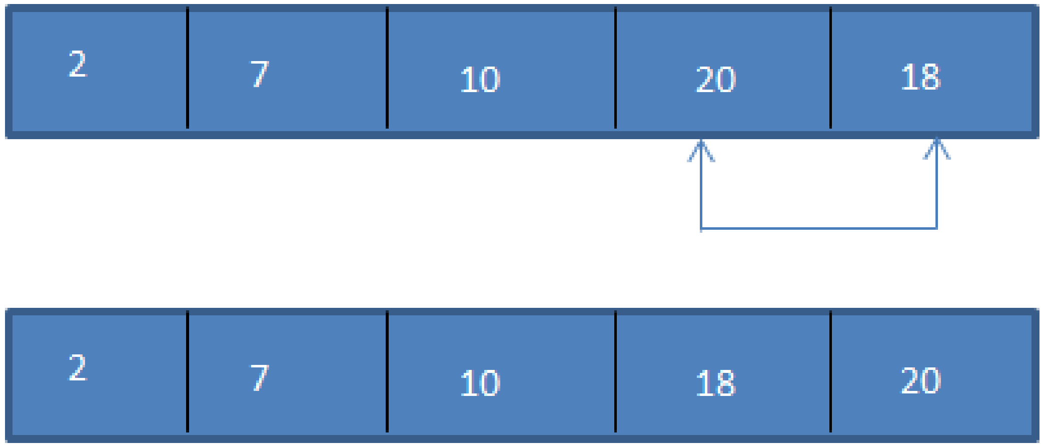

Pass 2:
0
1
2
3
4
2
18
10
20
7
2
10
18
20
7
2
10
18
20
7
2
7
18
20
10
=>Next smallest element at position 1

Pass 3:



=>Next smallest element at position 2

Pass 4:



=>Sorted array

Unsorted list	Least element	Sorted list
{18,10,7,20,2}	2	{}
{18,10,7,20}	7	{2}
{18,10,20}	10	{2,7}
{18,20}	18	{2,7,10)
{20}	20	{2,7,10,18}
{}		{2,7,10,18,20}