Algorithms

Sorting Algorithms

How to analyze a Sorting algorithm?

- A sorting algorithm is an algorithm made up of a **series of instructions that takes an array as input**, sometimes called a **list**, and **outputs a sorted array**.

- There are many factors to consider when choosing a sorting algorithm to use.

- All sorting algorithms share the same goal, but the way each algorithm goes about this task can vary.

- When working with any kind of algorithm, it is important to know **how fast it runs** and in **how much space** it operates—in other words, its **time complexity** and **space complexity**.

Types of sorting algorithms (1/2)

- **Comparison Sorts**

Compare elements to determine if one element should be to the left or right of another element.

Comparison sorts are usually more straightforward to implement than **integer sorts** but are limited by a lower bound of **$\Omega(n * \log n)$**.

The "on average" part here is important: there are many algorithms that run in very fast time if the input is **already sorted** or has some very particular property.

Types of sorting algorithms (2/2)

- **Integer Sorts**

Integer sorts do not make comparisons, so they are not bounded by **$\Omega(n * \log n)$**.

They determine **for each element x how many elements are less than x**. This information is used to **place each element into the correct place immediately**—no need to rearrange lists.

Space complexity

- The **running time** describes **how many operations an algorithm must carry out before it completes**.

- The **space complexity** describes **how much space must be allocated to run a particular algorithm**.
  - For example, if an algorithm takes in a list of size **n**, and for some reason makes a new list of size **n** for each element in **n**, the algorithm needs $\mathbf{n^2}$ space.

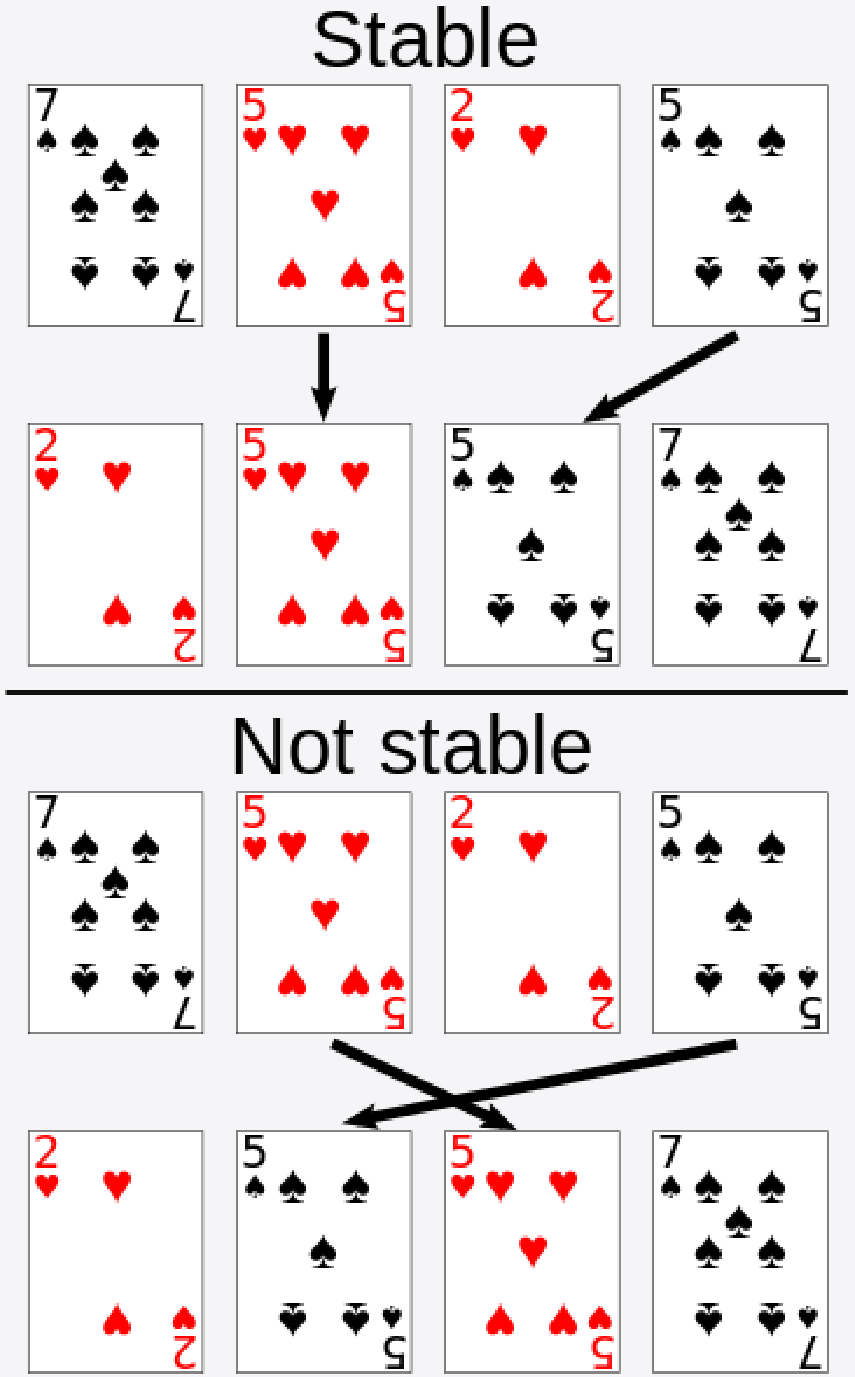
Stability

It is useful to know if a sorting algorithm is **stable**.

A sorting algorithm is **stable** if **it preserves the relative original order of elements with equal key values** (where the **key** is the value the algorithm sorts by).

Example

- When cards are sorted with a **stable sort**, the two **5's** *must remain in the same order* in the sorted output that they were originally in.
- When cards are sorted with a **non-stable sort**, the **5's** *may end up in the opposite order* in the sorted output.



Bubble Sort

A **sorting algorithm** that **compares two adjacent elements and swaps them until they are in the intended order**.

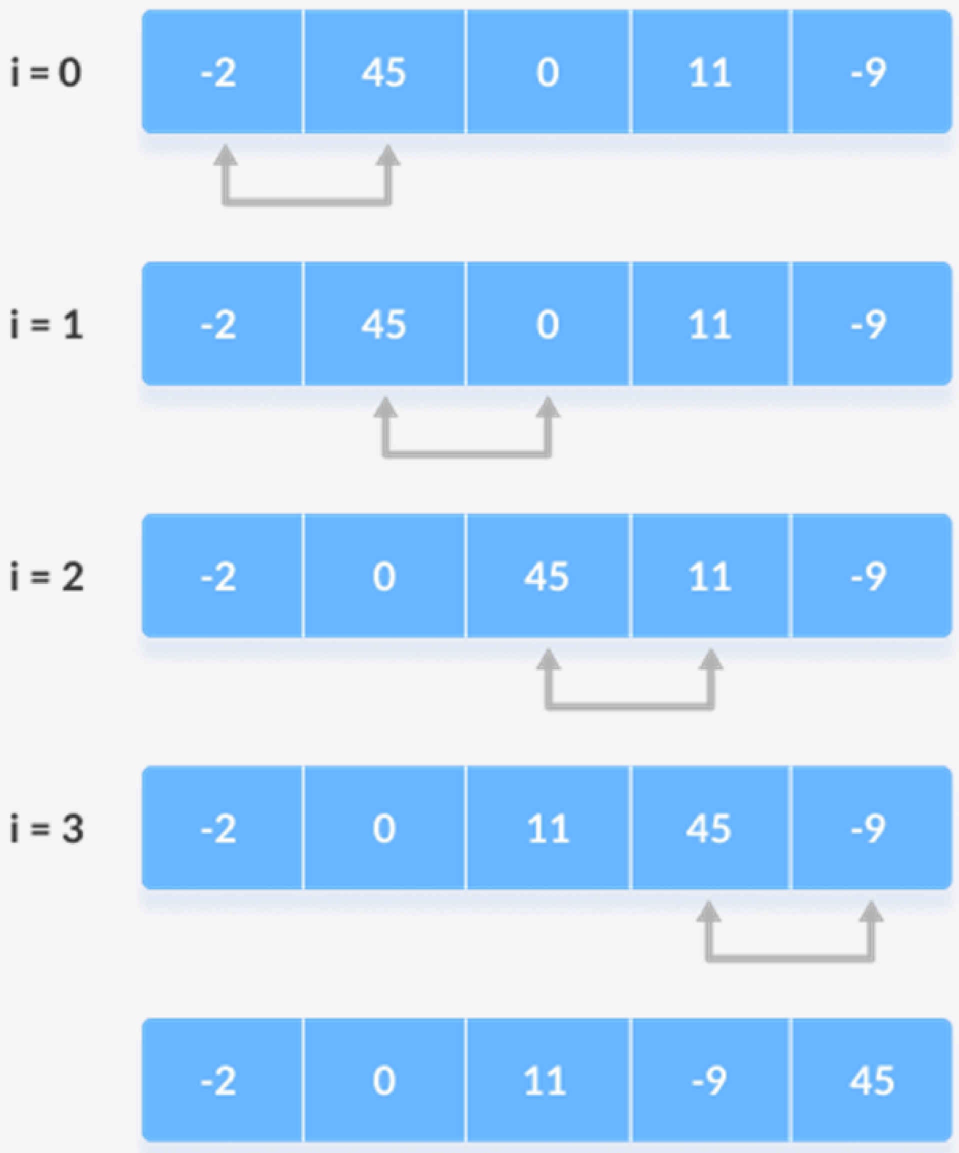
Just like the movement of air bubbles in the water that rise up to the surface, **each element of the array move to the end in each iteration**. Therefore, it is called a **bubble sort**.

Working of Bubble Sort

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.

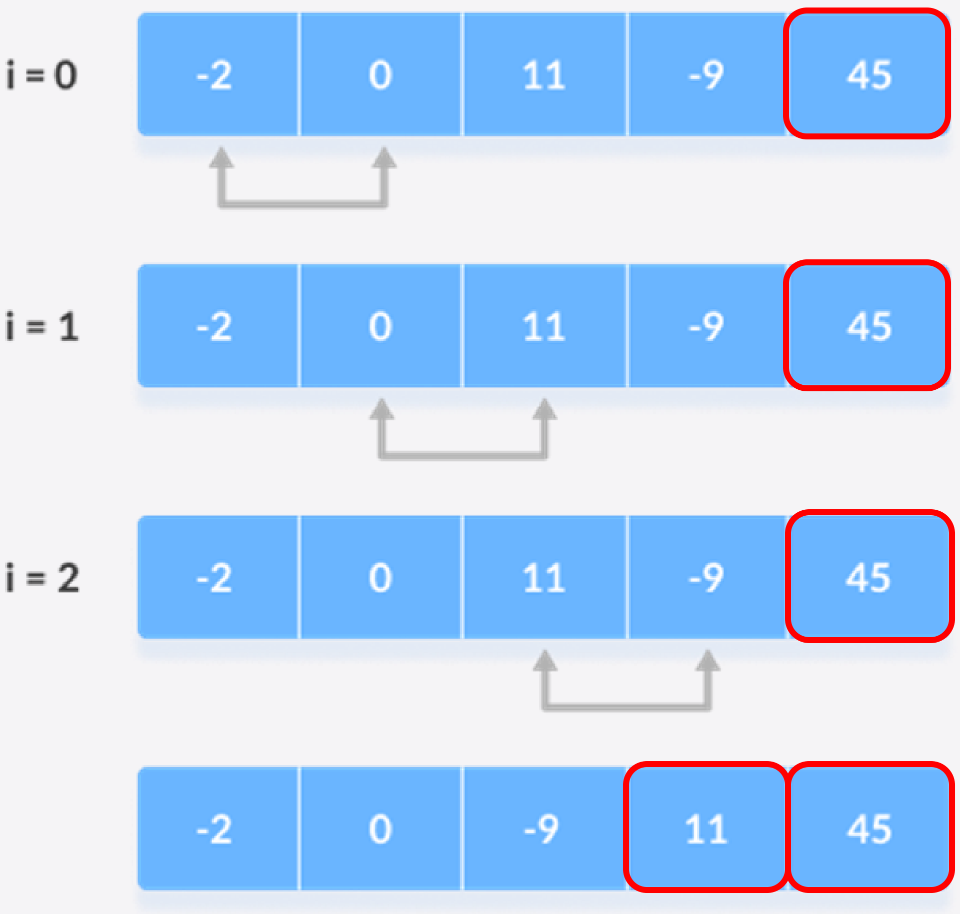
The above process goes on until the last element.



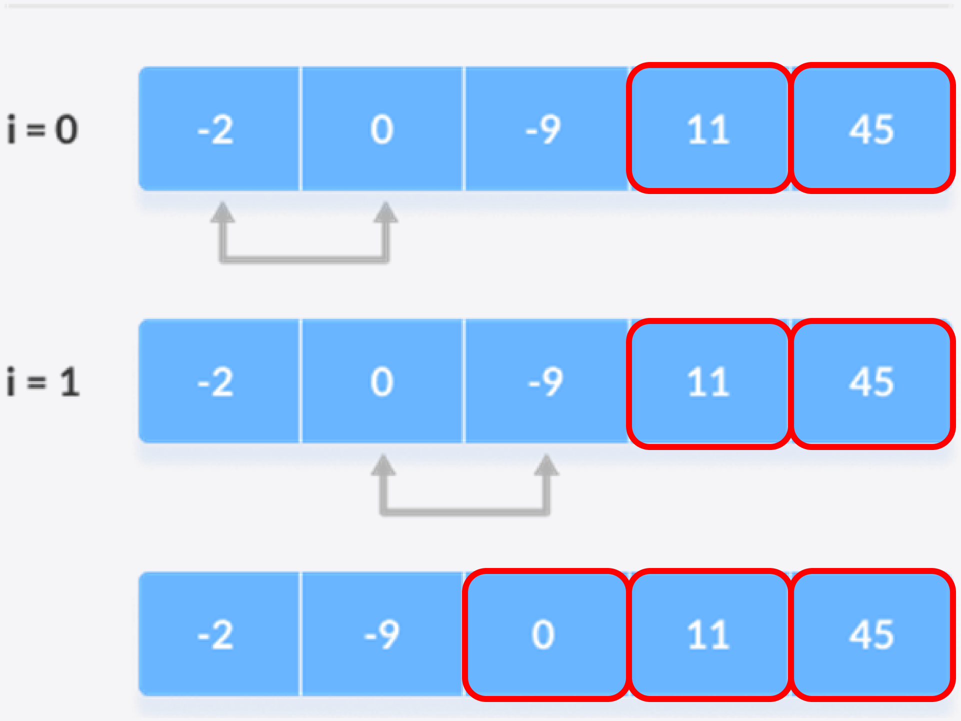
2. Remaining Iteration

- The same process goes on for the remaining iterations.

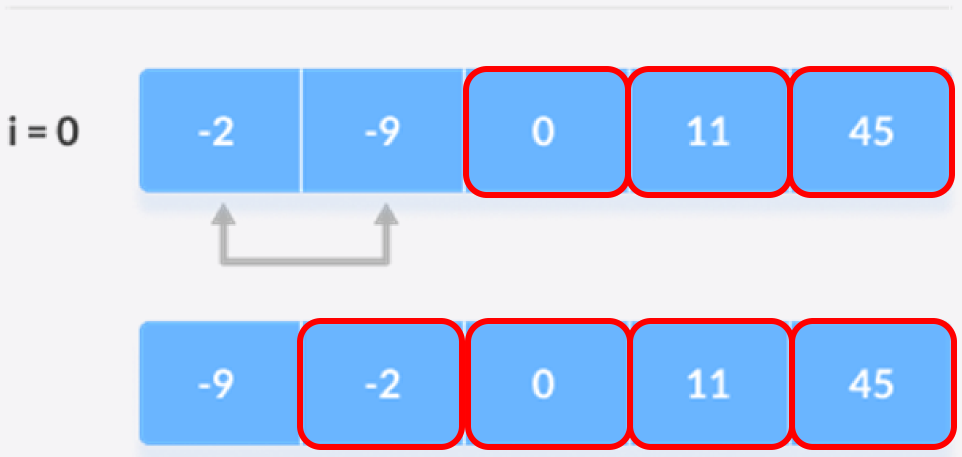
- After each iteration, **the largest element among the unsorted elements is placed at the end**.



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



- By definition, the array is sorted when all the unsorted elements are placed at their correct positions.



Bubble Sort Algorithm

```
void bubbleSort(int array[], int size)
{
    // loop to access each array element
    for (int step = 0; step < size; ++step)
    {
        // loop to compare array elements
        for (int i = 0; i < size - step; ++i)
        {
            // compare two adjacent elements
            // change > to < to sort in descending order
            if (array[i] > array[i + 1])
            {
                // swapping elements if elements
                // are not in the intended order
                swap(array[i], array[i+1]);
            }
        }
    }
}
```

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- We use an additional function, **swap()**, to swap the position of two values using the **pass-by-reference** technique.

C++ Example

Program body:

```
void bubbleSort(int array[], int size)
{
    for (int step = 0; step < size; ++step)
    {
        for (int i = 0; i < size - step; ++i)
        {
            if (array[i] > array[i + 1])
            {
                swap(array[i], array[i+1]);
            }
        }
    }
}

int main()
{
  int data[] = {-2, 45, 0, 11, -9};
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  cout << "Sorted Array in Ascending Order:\n";

  for (int i = 0; i < size; ++i)
    cout << "  " << data[i];
  cout << "\n";
}
```

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Result:

```
Sorted Array in Ascending Order:
  -9  -2  0  11  45
```

Optimized Bubble Sort Algorithm

```
void bubbleSort(int array[], int size)
{
    for (int step = 0; step < size; ++step)
    {
        for (int i = 0; i < size - step; ++i)
        {
            if (array[i] > array[i + 1])
            {
                swap(array[i], array[i+1]);
            }
        }
    }
}
```

In this version of the algorithm, all the comparisons are made **even if the array is already sorted**.

What might be the problem?

- *This increases the execution time.*

Optimized Bubble Sort Algorithm

```
void OptimizedBubbleSort(int array[], int size)
{
    for (int step = 0; step < (size-1); ++step)
    {
        // check if swapping occurs
        bool swapped = false;

        for (int i = 0; i < (size-step-1); ++i)
        {
            if (array[i] > array[i + 1])
            {
                swap(array[i], array[i+1]);

                // if swapping occurs set to True
                swapped = true;
            }
        }

        // no swapping means the array is already sorted
        // so no need of further comparison
        if (swapped == false)
            break;
    }
}
```

To solve this, we can introduce an extra variable **swapped**.

The value of swapped is set **true** if there occurs swapping of elements. Otherwise, it is set **false**.

If **swapped is false**, the **elements are already sorted** and there is no need to perform further iterations.

Bubble Sort Complexity

Time Complexity	
Best	?
Worst	?
Average	?
Space Complexity	?
Stability	?

Bubble Sort Complexity

Time Complexity	
Best	O(n)
Worst	?
Average	?
Space Complexity	?
Stability	?

If the array is already sorted, then there is no need for sorting.

Bubble Sort Complexity

Time Complexity	
Best	O(n)
Worst	O(n^2)
Average	?
Space Complexity	?
Stability	?

If the array is already sorted, then there is no need for sorting.

If we want to sort in ascending order and the array is in descending order, then the worst case occurs.

Bubble Sort Complexity

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

Bubble Sort Complexity

Time Complexity		
Best	O(n)	If the array is already sorted, then there is no need for sorting.
Worst	O(n^2)	If we want to sort in ascending order and the array is in descending order, then the worst case occurs.
Average	O(n^2)	It occurs when the elements of the array are in jumbled order (neither ascending nor descending).
Space Complexity	O(1)	
Stability	?	

Bubble Sort Complexity

Time Complexity		
Best	**O(n)**	If the array is already sorted, then there is no need for sorting.
Worst	**O(n^2)**	If we want to sort in ascending order and the array is in descending order, then the worst case occurs.
Average	**O(n^2)**	It occurs when the elements of the array are in jumbled order (neither ascending nor descending).
Space Complexity	**O(1)**	
Stability	**Yes**	

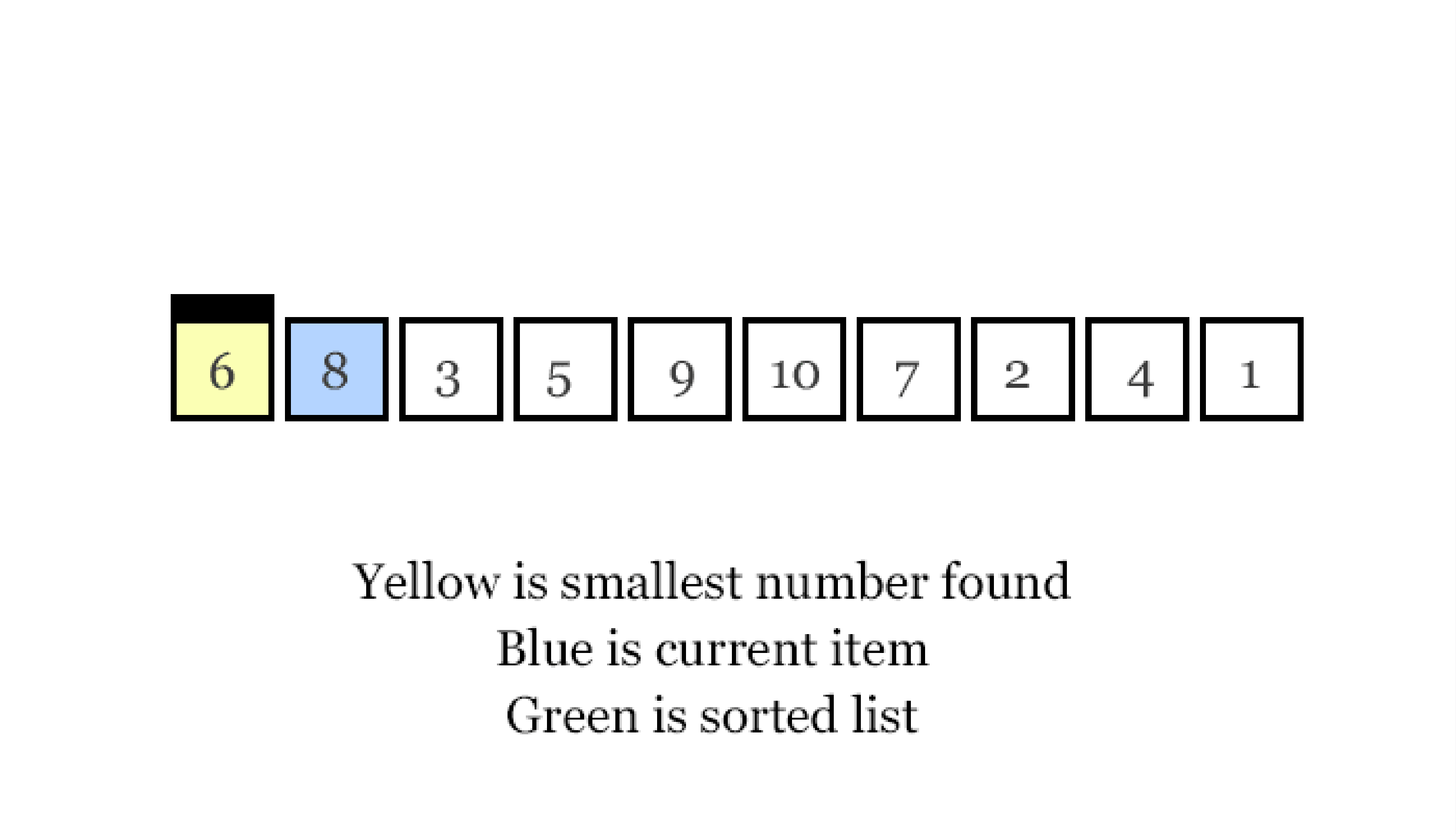
Bubble Sort Applications

Bubble sort is used if

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

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.



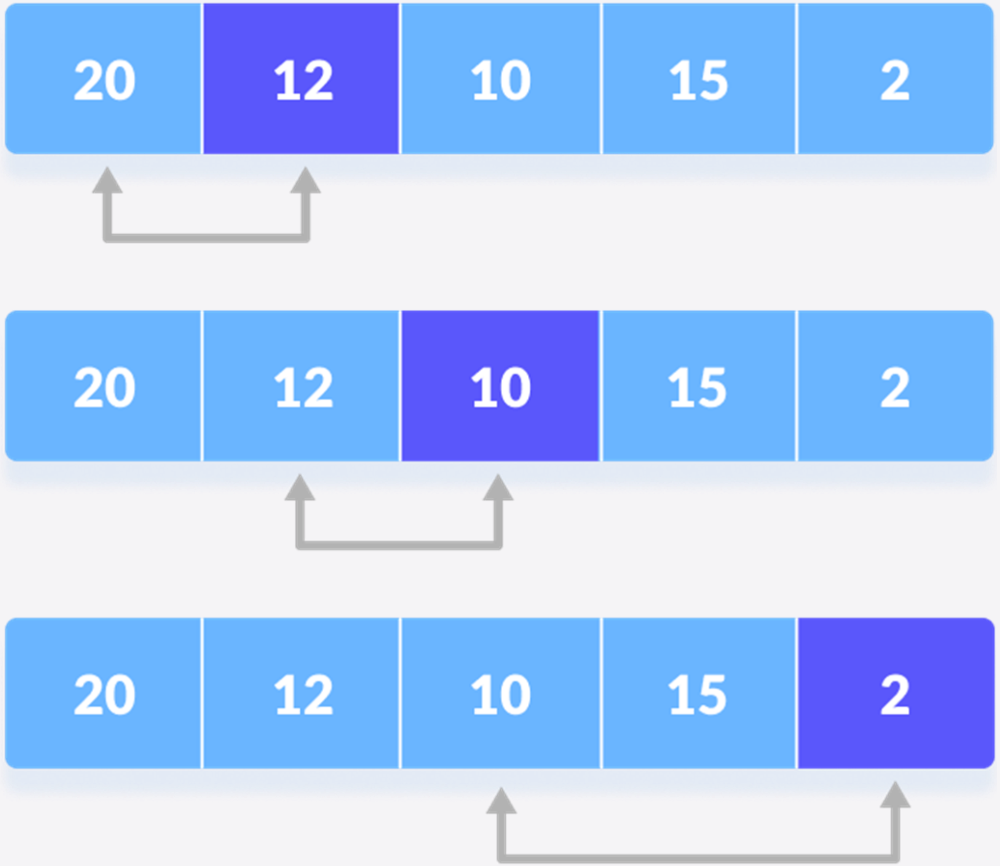
Working of Selection Sort

1. Set the first element as minimum.

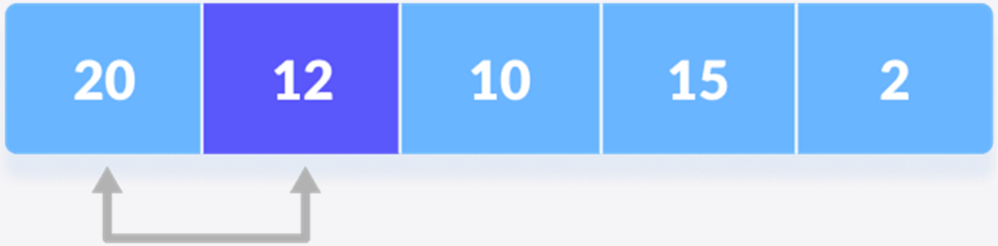
2. **Compare minimum with the second element**. If the second element is smaller than **minimum**, assign the second element as **minimum**.

Compare **minimum** with the **third element**...

The process goes on until the last element.

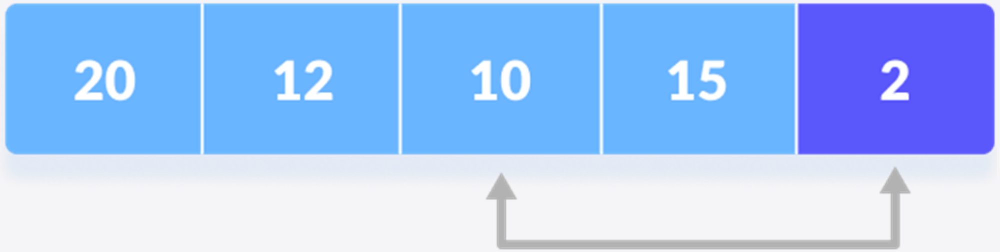


2. **Compare minimum with the second element**. If the second element is smaller than **minimum**, assign the second element as **minimum**.



Compare **minimum** with the **third element**...

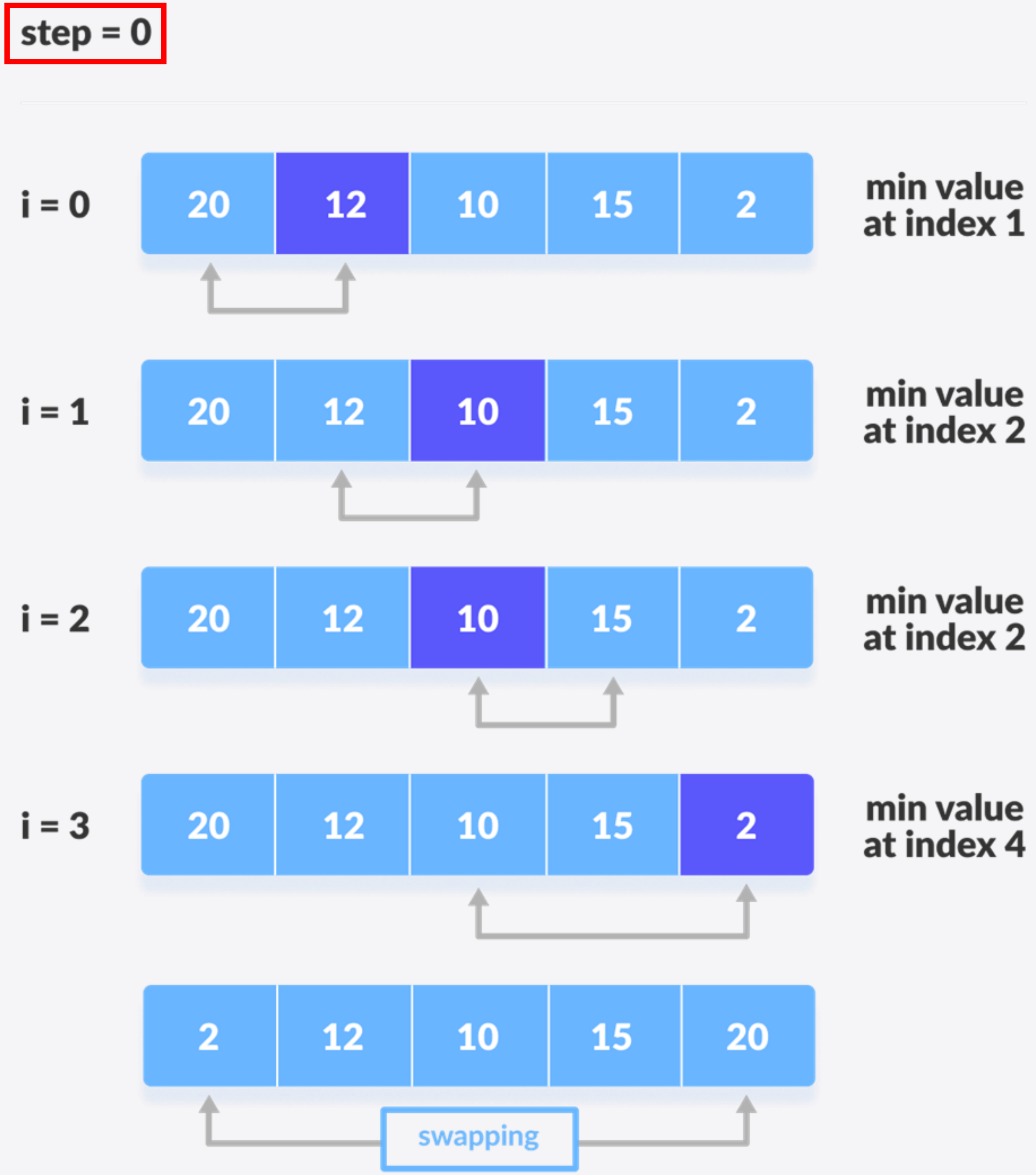
The process goes on until the last element.



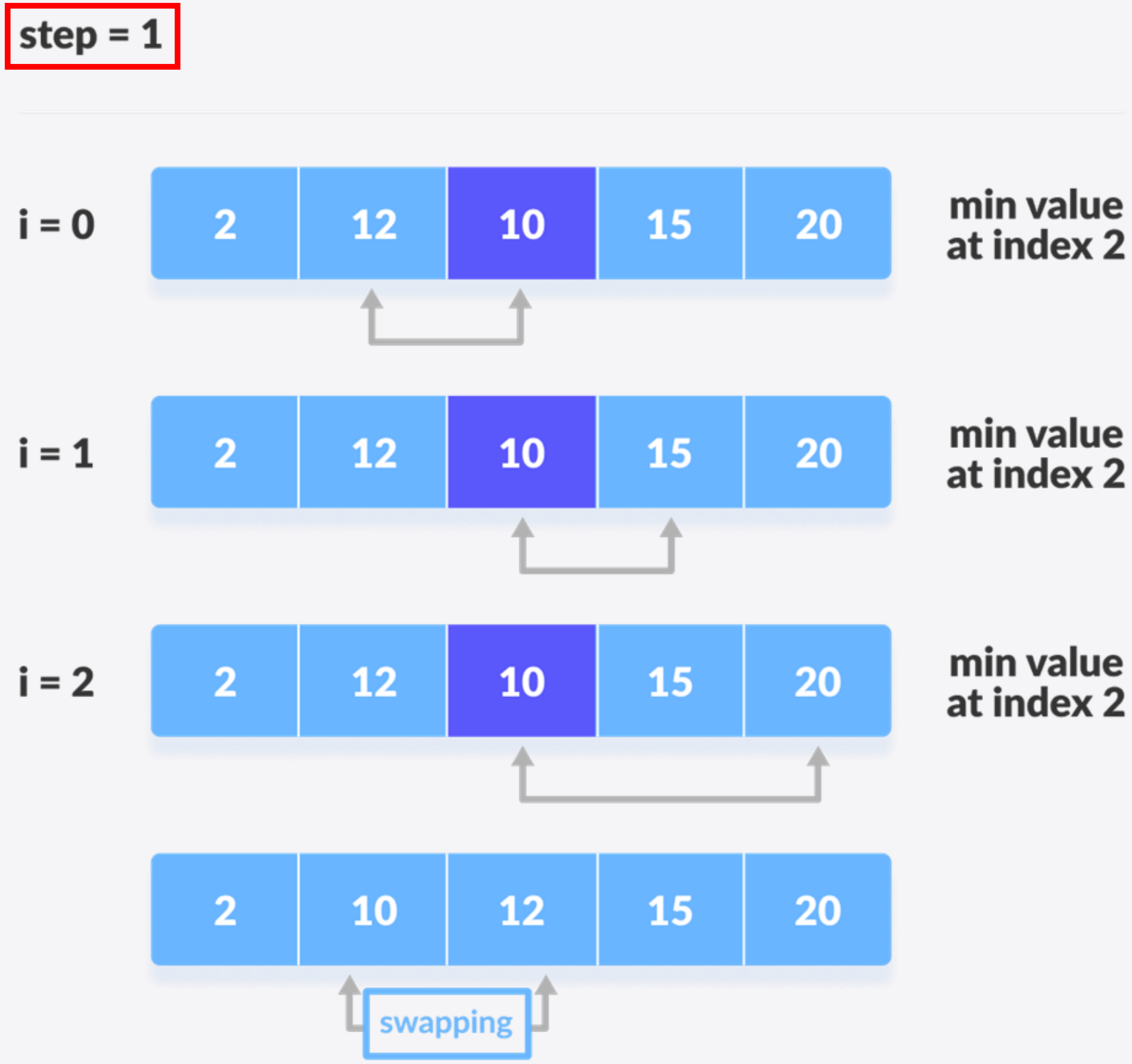
3. After each iteration, **minimum** is placed in the **front** of the **unsorted list**.

- Swap the **first** with **minimum**

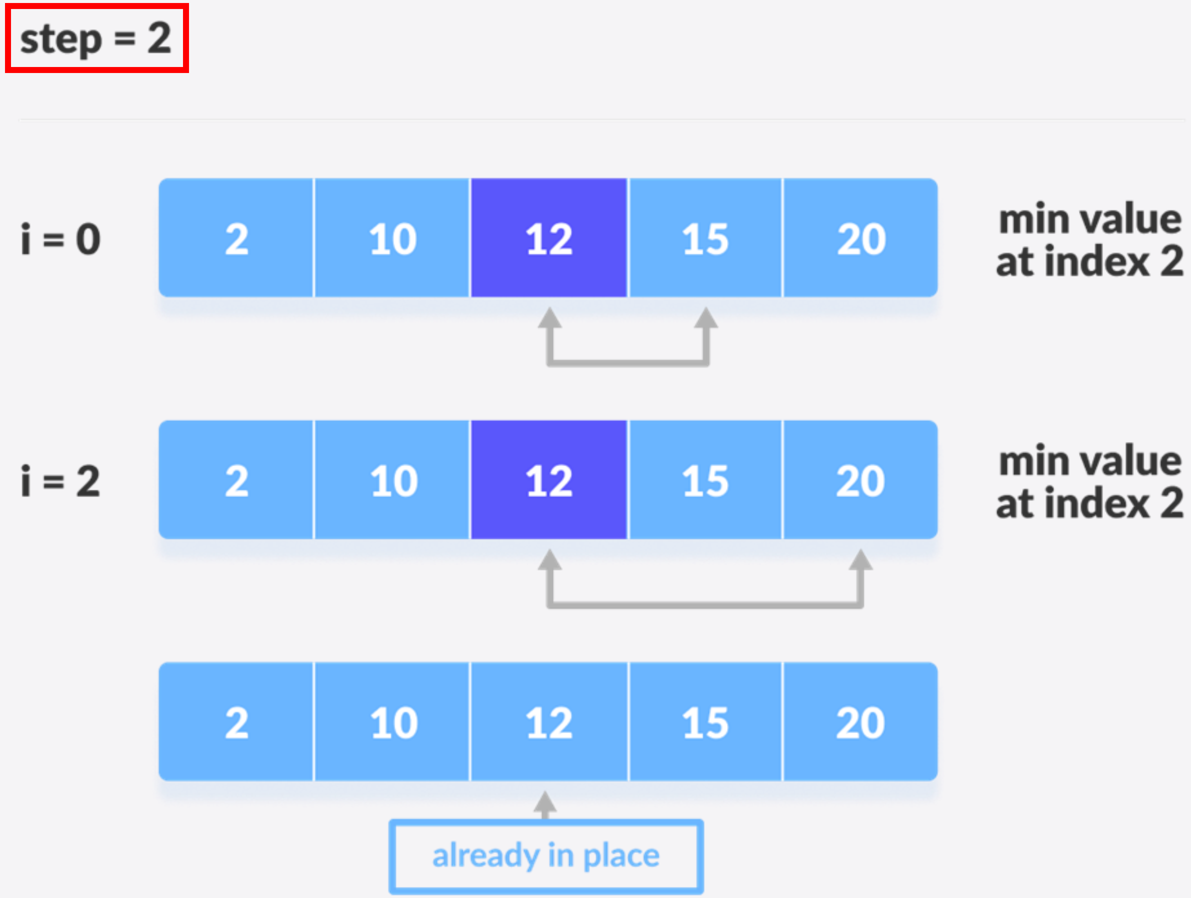
4. For each iteration, indexing starts from the first unsorted element. **Step 1** to **3** are repeated **until all the elements are placed at their correct positions**.



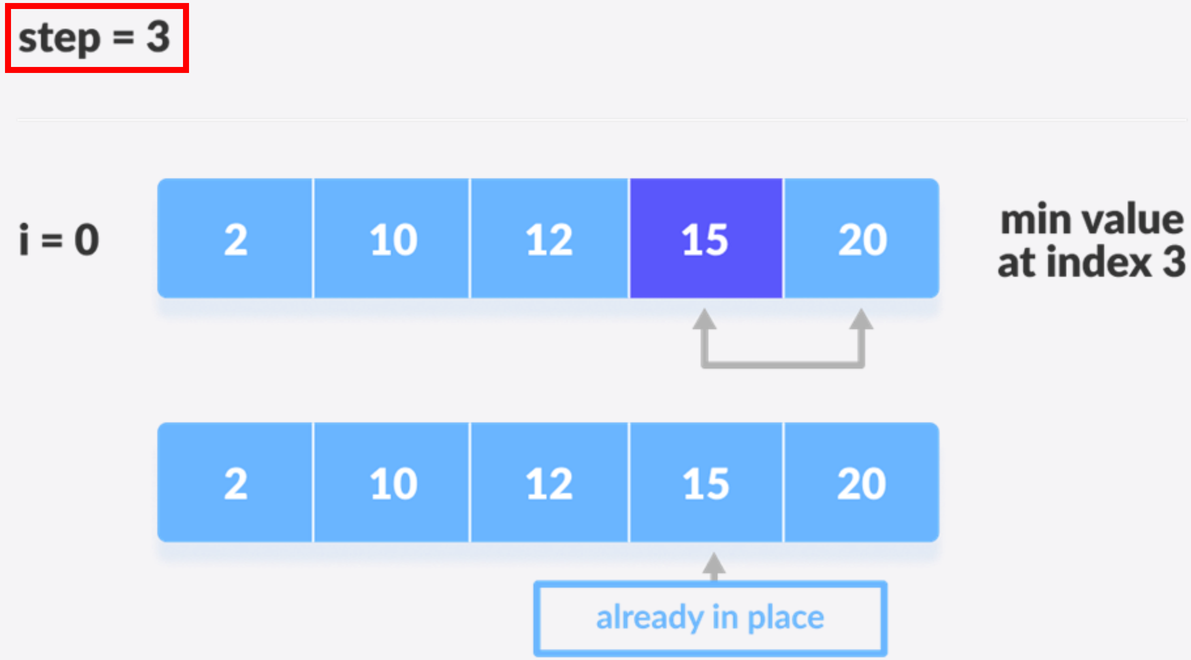
4. For each iteration, indexing starts from the first unsorted element. **Step 1** to **3** are repeated **until all the elements are placed at their correct positions**.



4. 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**.



4. 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 Algorithm

```
void selectionSort(int array[], int size)
{
    for (int step = 0; step < size - 1; step++)
    {
        int min_idx = step;

        for (int i = step + 1; i < size; i++)
        {
            // To sort in descending order, change > to < in this line.
            // Select the minimum element in each loop.
            if (array[i] < array[min_idx])
                min_idx = i;
        }

        // put min at the correct position
        swap(array[min_idx], array[step]);
    }
}
```

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- We use an additional function, **swap()**, to swap the position of two values using the **pass-by-reference** technique.

Selection Sort Complexity

Time Complexity	
Best	?
Worst	?
Average	?
Space Complexity	?
Stability	?

Selection Sort Complexity

Time Complexity	
Best	**O(n^2)** It occurs when the array is already sorted
Worst	?
Average	?
Space Complexity	?
Stability	?

Selection Sort Complexity

Time Complexity	
Best	**O(n^2)** It occurs when the array is already sorted
Worst	**O(n^2)** If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
Average	?
Space Complexity	?
Stability	?

Selection Sort Complexity

Time Complexity	
Best	O(n^2)
Worst	O(n^2)
Average	O(n^2)
Space Complexity	?
Stability	?

The time complexity of the selection sort is the same in all cases.
You must 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.

Best: It occurs when the array is already sorted

Worst: If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

Average: It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

Selection Sort Complexity

Time Complexity		
Best	**O(n^2)**	It occurs when the array is already sorted
Worst	**O(n^2)**	If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
Average	**O(n^2)**	It occurs when the elements of the array are in jumbled order (neither ascending nor descending).
Space Complexity	**O(1)**	
Stability	**?**	

Selection Sort Complexity

This can be shown by seeing how the algorithm handles the array:
A = [5, 7, 5, 2]

Time Complexity		
Best	**O(n^2)**	It occurs when the array is already sorted
Worst	**O(n^2)**	If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
Average	**O(n^2)**	It occurs when the elements of the array are in jumbled order (neither ascending nor descending).
Space Complexity	**O(1)**	
Stability	**No**	

Selection Sort Applications

The selection sort is used when

- complexity does not matter
- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is $O(n)$ as compared to $O(n^2)$ of bubble sort)

Insertion Sort

Insertion sort places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand:

We assume that the first card is already sorted, then we select an unsorted random card. If the card is greater than the card in hand, it is placed on the right, otherwise, to the left.

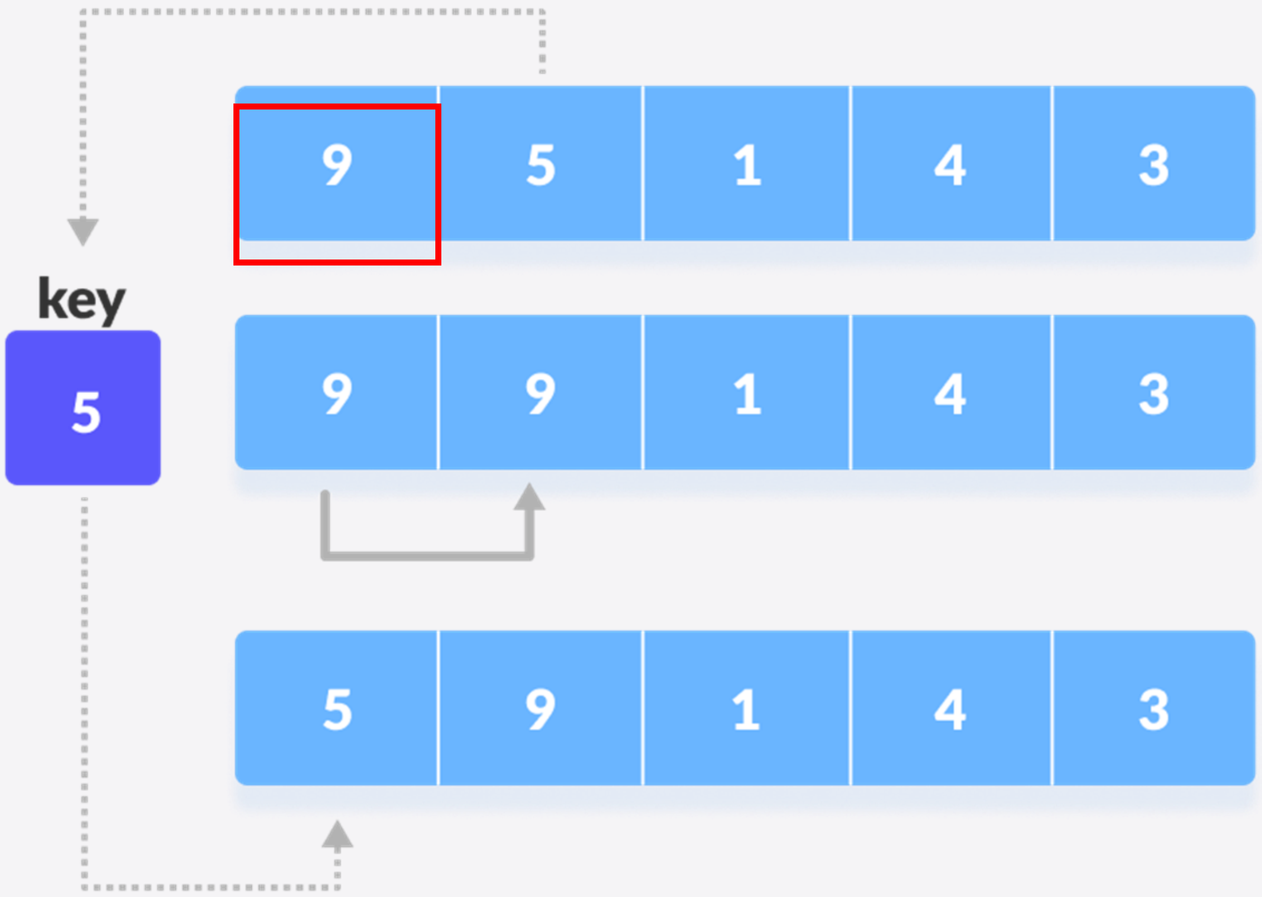
Working of Insertion Sort

Suppose we need to sort the following array.

The first element in the array is assumed to be sorted.

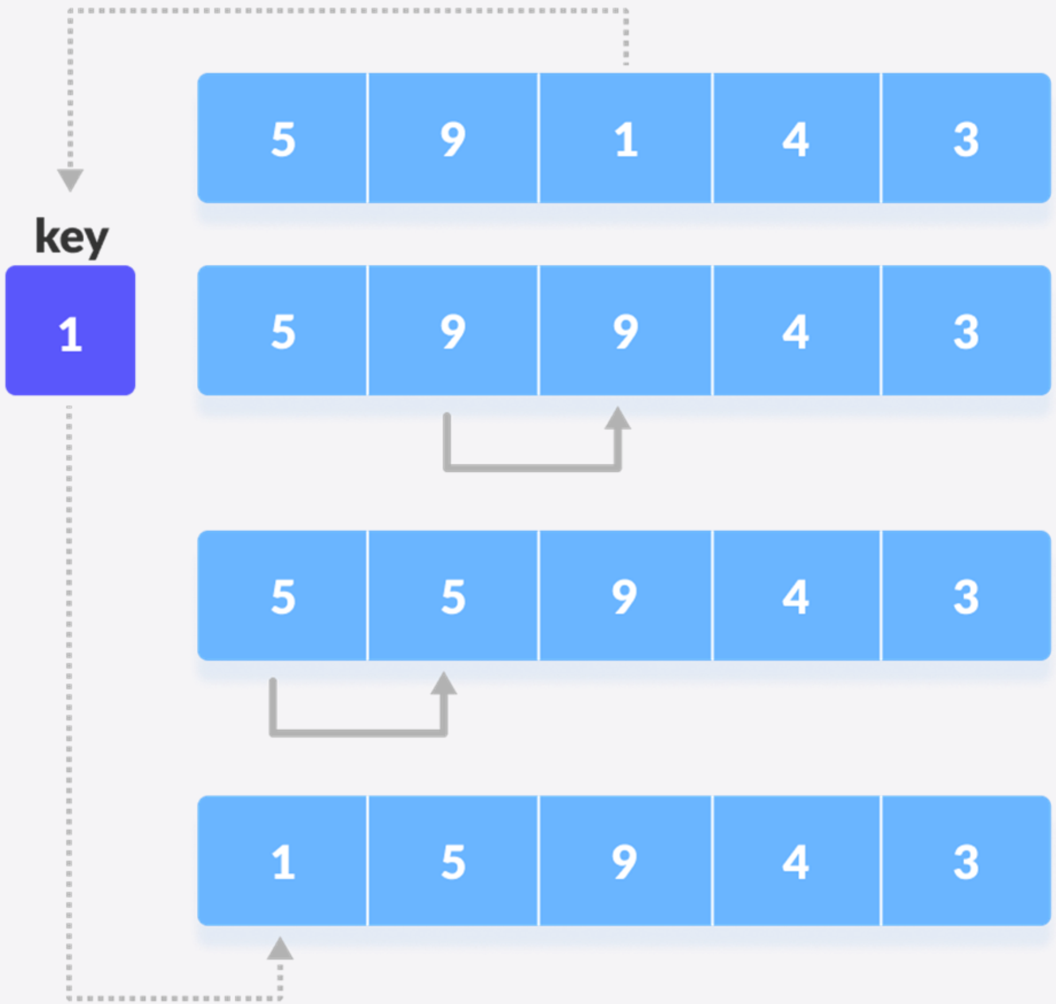
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in **key**.

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

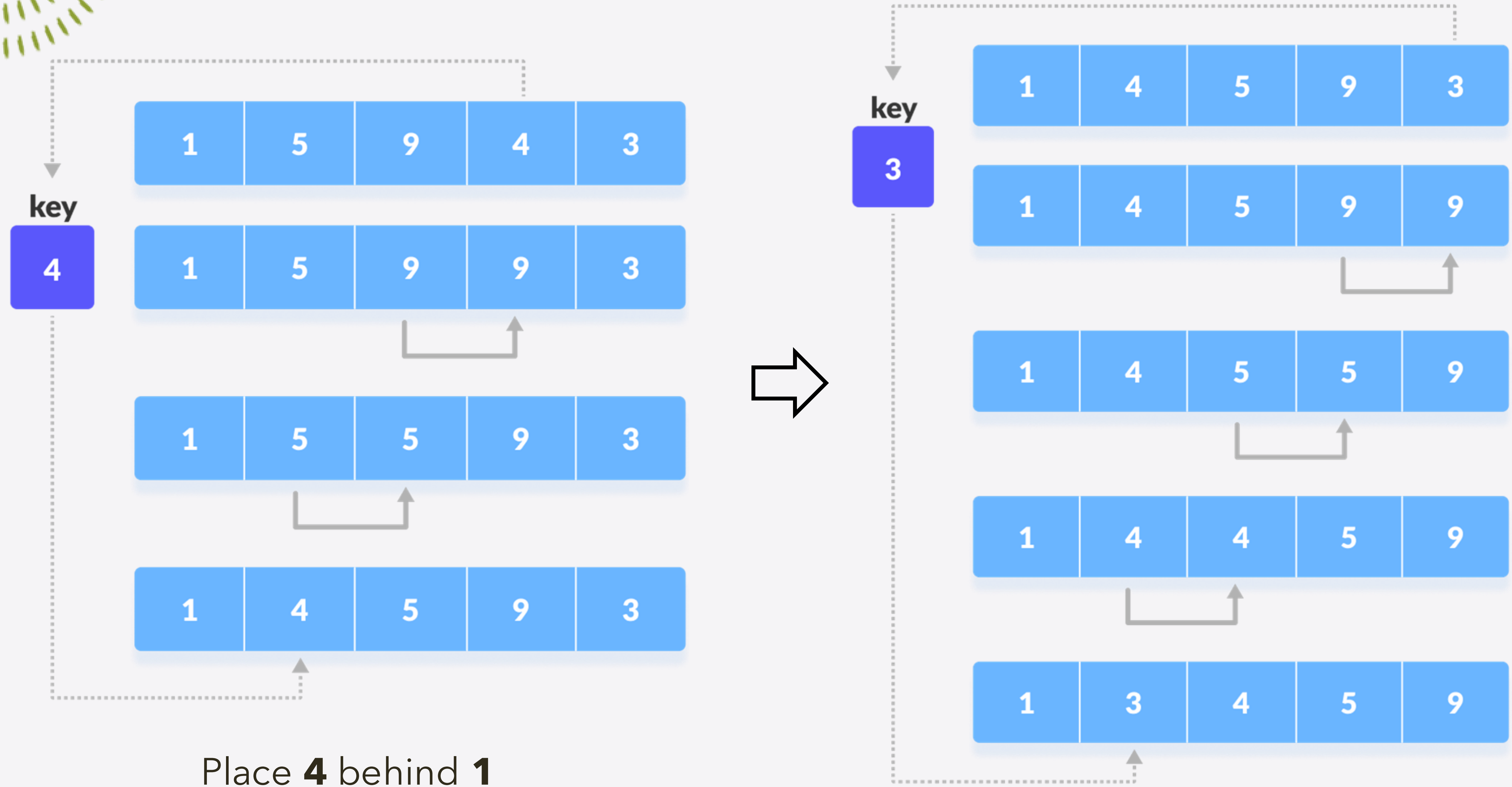


2. Now, the first two elements are sorted.

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



Place **4** behind **1**

Place **3** behind **1** and the array is sorted

Insertion Sort Algorithm

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
        sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

cost	*times*

Insertion Sort Complexity

Time Complexity	
Best	?
Worst	?
Average	?
Space Complexity	?
Stability	?

Insertion Sort Complexity

Time Complexity	
Best	O(n)
Worst	?
Average	?
Space Complexity	?
Stability	?

If the array is sorted, the outer loop runs for n times whereas the inner loop does not run at all

Insertion Sort Complexity

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

Time Complexity	
Best	O(n)
Worst	O(n^2)
Average	?
Space Complexity	?
Stability	?

If the array is sorted, the outer loop runs for n times whereas the inner loop does not run at all

If an array is in ascending order, and you want to sort it in descending order.

Insertion Sort Complexity

Time Complexity		
Best	**O(n)**	If the array is sorted, the outer loop runs for n times whereas the inner loop does not run at all
Worst	**O(n^2)**	If an array is in ascending order, and you want to sort it in descending order.
Average	**O(n^2)**	It occurs when the elements of an array are in jumbled order (neither ascending nor descending).
Space Complexity	?	
Stability	?	

Insertion Sort Complexity

Time Complexity	
Best	**O(n)**
Worst	**O(n^2)**
Average	**O(n^2)**
Space Complexity	**O(1)**
Stability	**?**

If the array is sorted, the outer loop runs for n times whereas the inner loop does not run at all

If an array is in ascending order, and you want to sort it in descending order.

It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Insertion Sort Complexity

Time Complexity		
Best	**O(n)**	If the array is sorted, the outer loop runs for n times whereas the inner loop does not run at all
Worst	**O(n^2)**	If an array is in ascending order, and you want to sort it in descending order.
Average	**O(n^2)**	It occurs when the elements of an array are in jumbled order (neither ascending nor descending).
Space Complexity	**O(1)**	
Stability	**Yes**	

Insertion Sort Applications

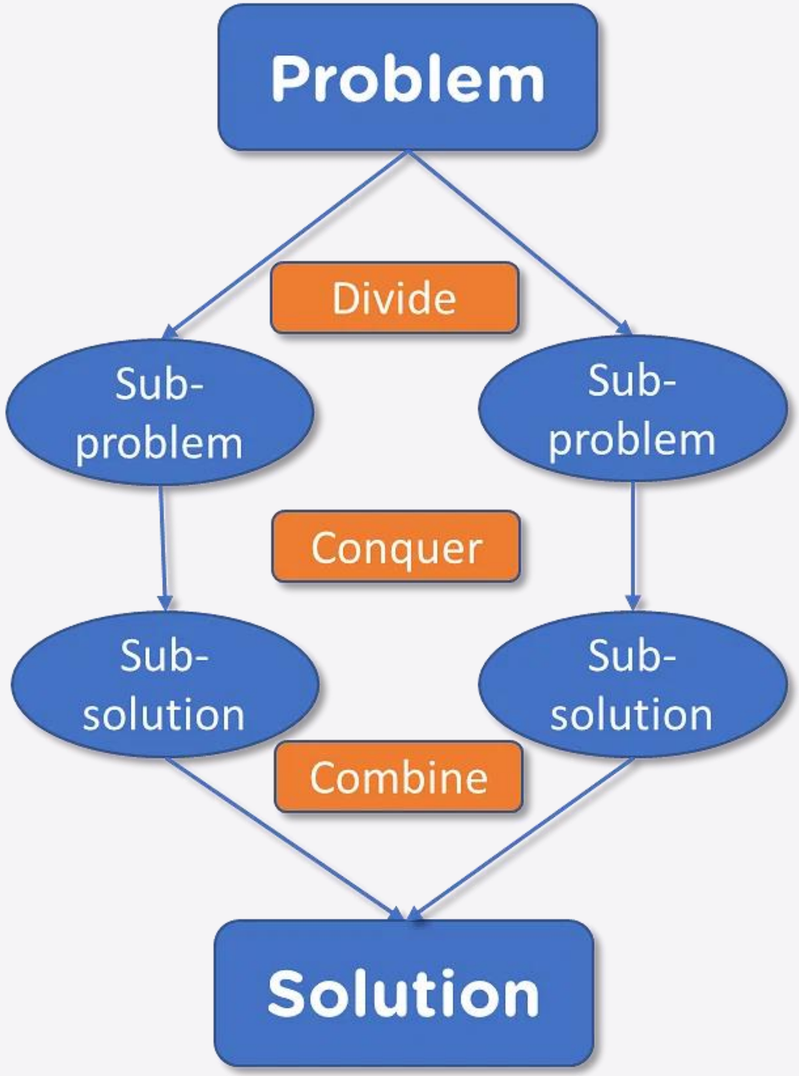
The insertion sort is used when:

- **The array has a small number of elements**
- **There are only a few elements left to be sorted**

Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of **Divide and Conquer Algorithm**.

Merge Sort is one of the most popular sorting algorithms that is based on the principle of **Divide and Conquer Algorithm**.



We divide a problem into subproblems.

When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an **array A**.

We must sort an array starting at index **p** and ending at index **r**, denoted as **A[p...r]**.

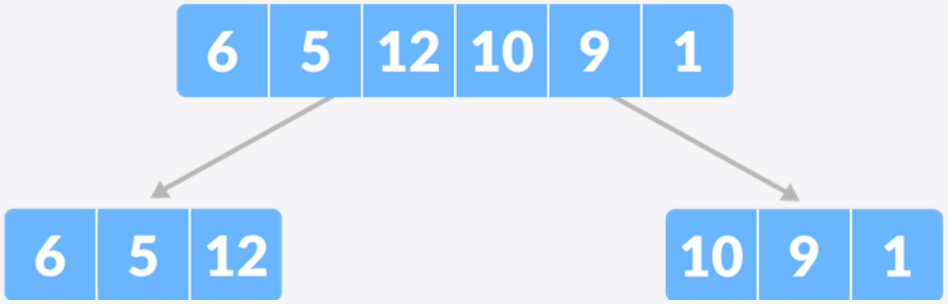
Merge Sort example

6	5	12	10	9	1

Suppose we had to sort an **array A**.

We must sort an array starting at index **p** and ending at index **r**, denoted as **A[p...r]**.

Merge Sort example



Divide

we can split the subarray **A[p...r]** into two arrays **A[p...q]** and **A[q+1...r]**, where $p<q<r$.

Suppose we had to sort an **array A**.

We must sort an array starting at index **p** and ending at index **r**, denoted as **A[p...r]**.

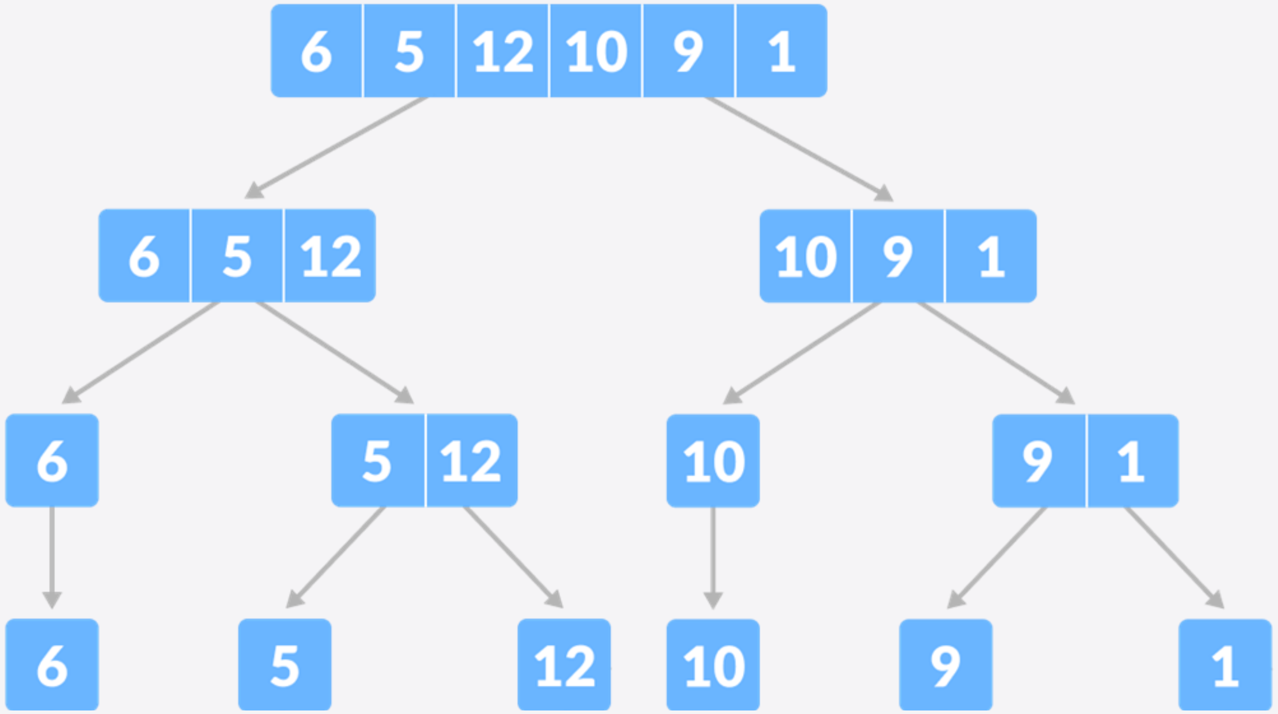
Divide

we can split the subarray **A[p...r]** into two arrays **A[p...q]** and **A[q+1...r]**, where $p<q<r$.

Conquer

We try to sort both **subarrays A[p...r]** and **A[q+1...r]**. If we haven't yet reached the ***base case***, we again **divide** and try to sort them.

Merge Sort example



Suppose we had to sort an **array A**.

We must sort an array starting at index **p** and ending at index **r**, denoted as **A[p...r]**.

Divide

we can split the subarray **A[p...r]** into two arrays **A[p...q]** and **A[q+1...r]**, where $p<q<r$.

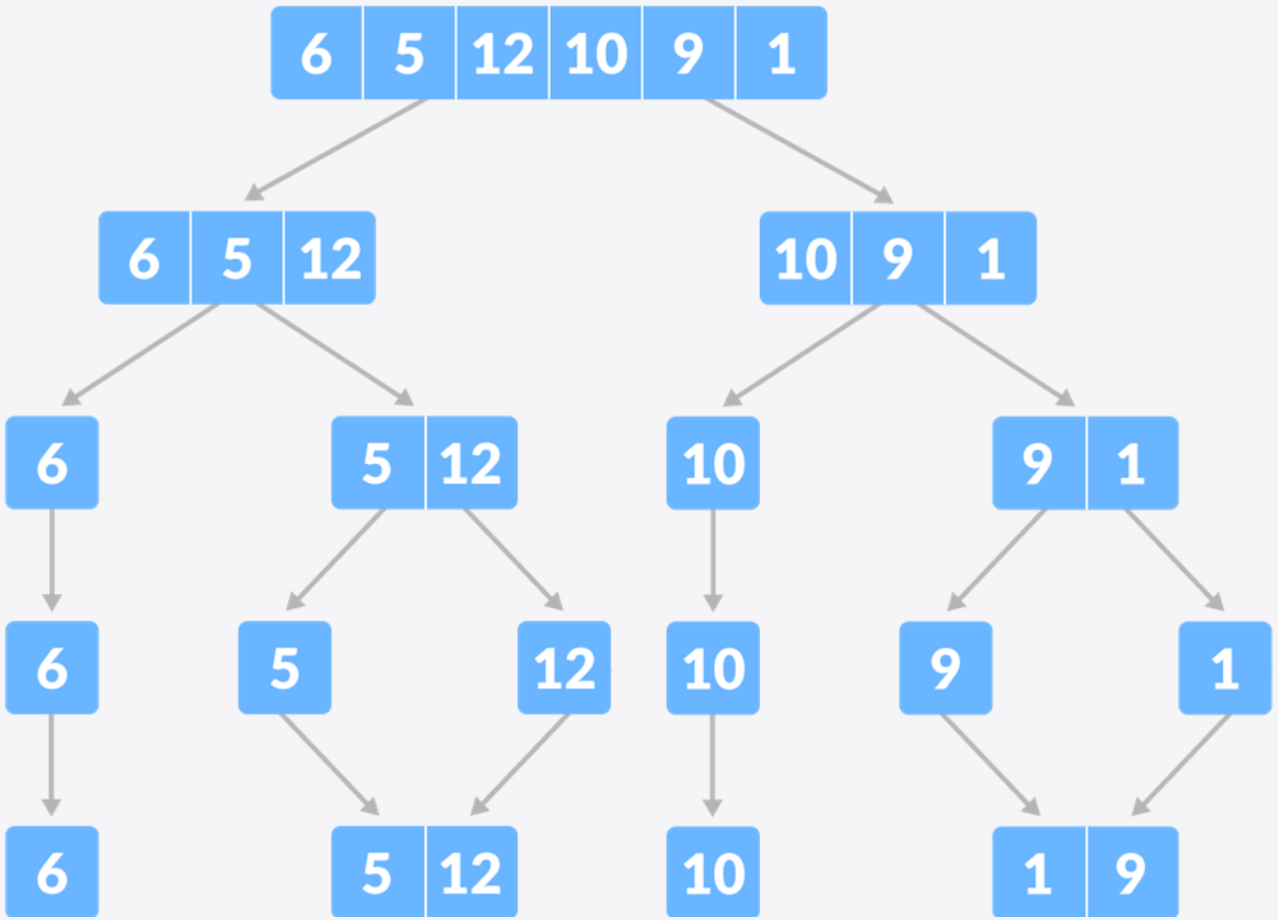
Conquer

We try to sort both **subarrays A[p...r]** and **A[q+1...r]**. If we haven't yet reached the ***base case***, we again **divide** and try to sort them.

Combine

When the **conquer step** reaches the ***base case***, we get two sorted subarrays **A[p...q]** and **A[q+1...r]** to combine and get a sorted array **A[p...r]**.

Merge Sort example



Suppose we had to sort an **array A**.

We must sort an array starting at index **p** and ending at index **r**, denoted as **A[p…r]**.

Divide

we can split the subarray **A[p…r]** into two arrays **A[p…q]** and **A[q+1…r]**, where $p<q<r$.

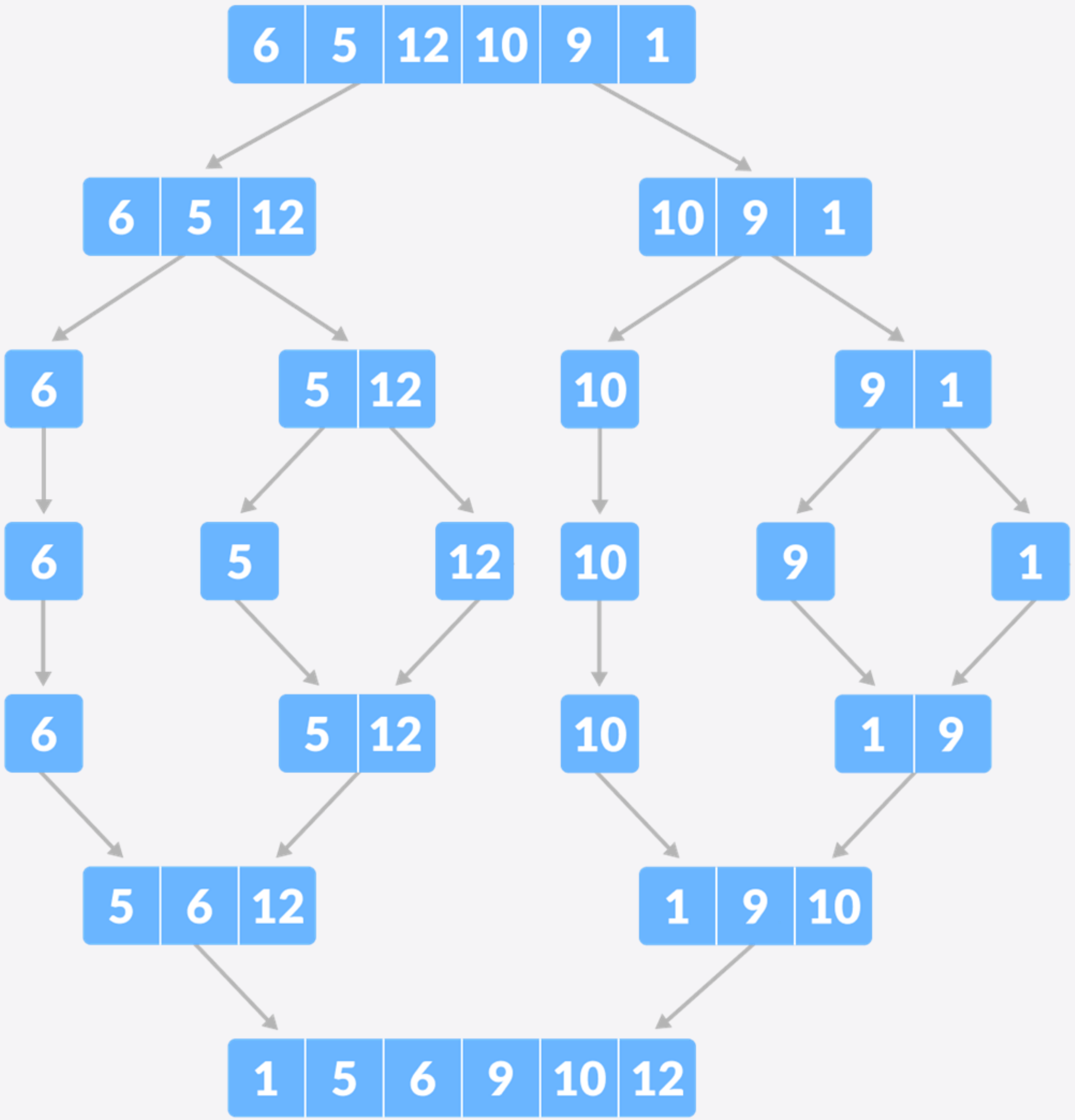
Conquer

We try to sort both **subarrays A[p…r]** and **A[q+1…r]**. If we haven't yet reached the ***base case***, we again **divide** and try to sort them.

Combine

When the **conquer step** reaches the ***base case***, we get two sorted subarrays **A[p…q]** and **A[q+1…r]** to combine and get a sorted array **A[p…r]**.

Merge Sort example



Merge Sort Algorithm (1/3)

The algorithm recursively divides the array into halves until we reach the **base case** of array with **1 element**.

Then, the **merge function** takes the sorted sub-arrays and merges them to gradually sort the entire array.

MERGE-SORT(A, p, r)

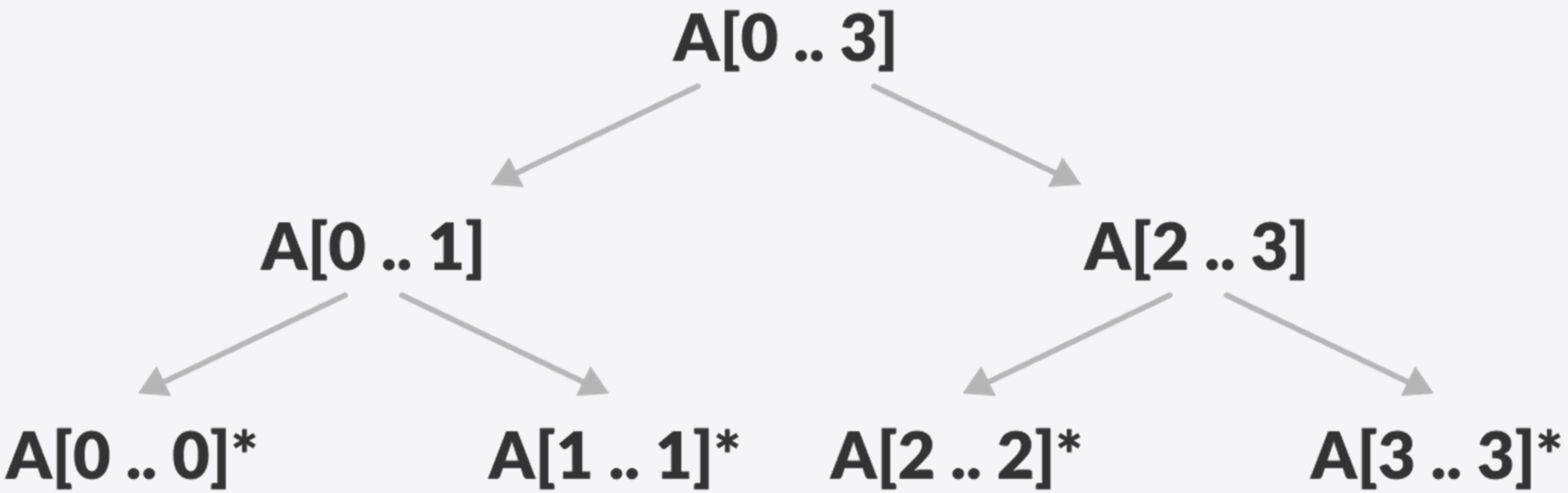
if $p < r$

  $q = \lfloor (p + r)/2 \rfloor$

  MERGE-SORT(A, p, q)

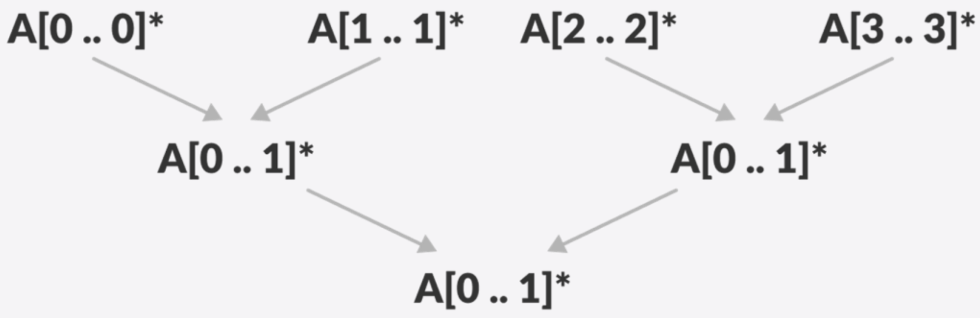
  MERGE-SORT$(A, q + 1, r)$

  MERGE(A, p, q, r)



Merge Sort Algorithm (2/3)

Then, the **merge function** takes the sorted sub-arrays and merges them to gradually sort the entire array.




```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
      L[i] = A[p + i - 1]
for j = 1 to n2
      R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
      if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
      else A[k] = R[j]
            j = j + 1
```

Create
L ← A[p...q]
M ← A[q+1...r]

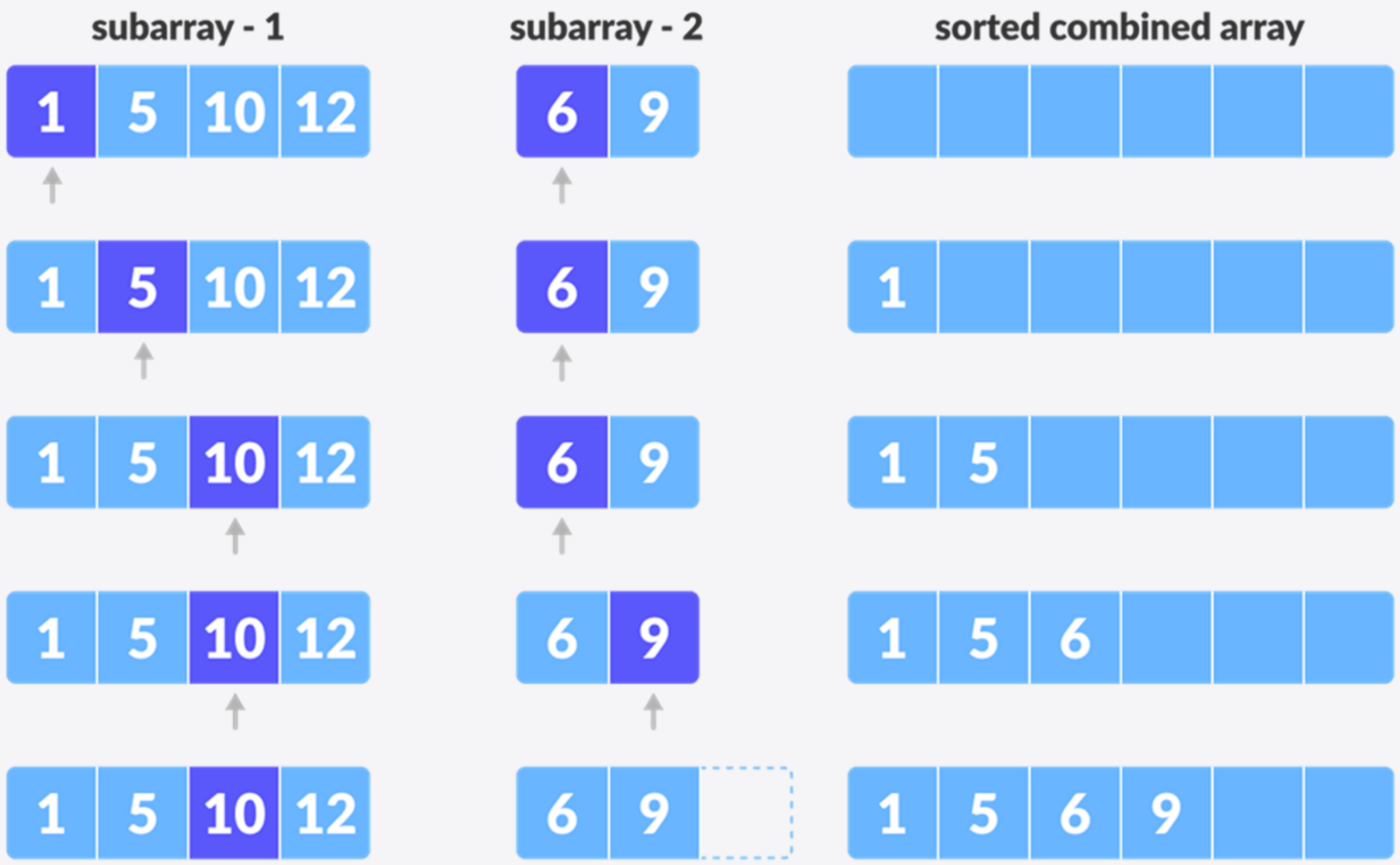
sub-arrays L & R Index

Fill the array A with the sorted sub-array values

Merge Sort Algorithm (3/3)

The algorithm maintains **three pointers**, one for each of the **two temporal arrays** and one for maintaining the current index of the **final sorted array**.

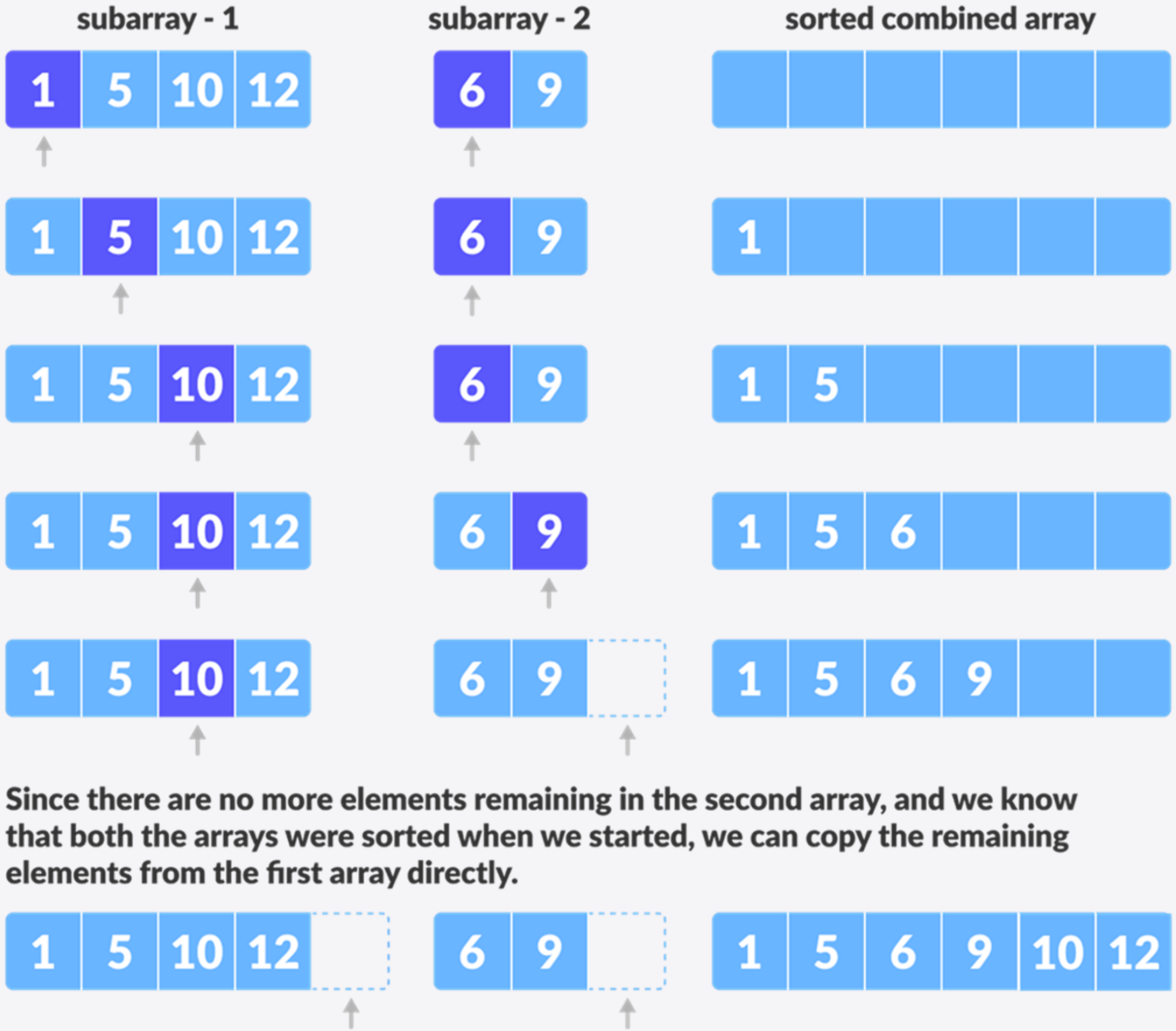
Compares de elements in the two sub-arrays to obtain the final / sub - solution.



Merge Sort Algorithm (3/3)

The algorithm maintains **three pointers**, one for each of the **two temporal arrays** and one for maintaining the current index of the **final sorted array**.

Compares de elements in the two sub-arrays to obtain the final / sub - solution.



Merge Sort Complexity

Time Complexity	
Best	?
Worst	?
Average	?
Space Complexity	?
Stability	?

Merge Sort Complexity

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

Merge Sort Complexity

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

Merge Sort Complexity

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

Merge Sort Complexity

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

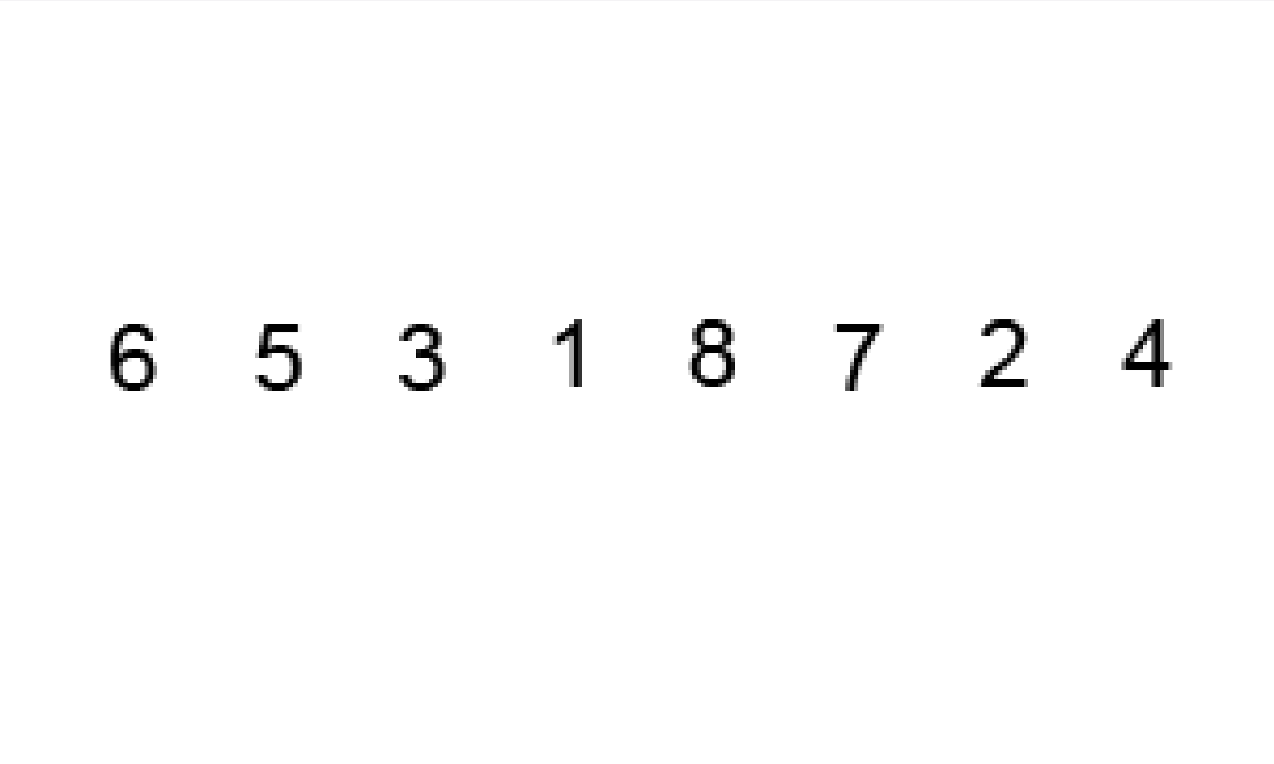
Merge Sort Complexity

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

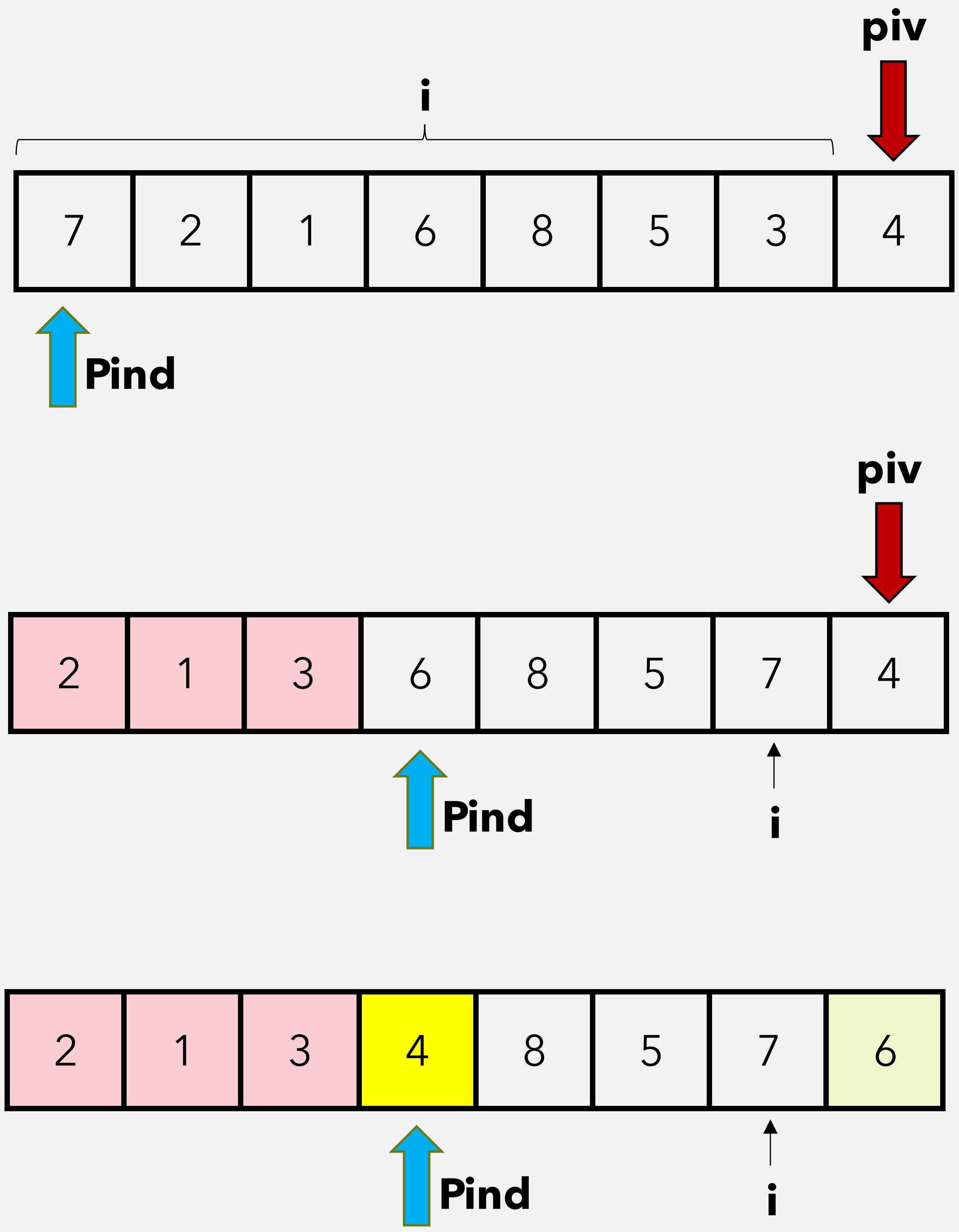
QuickSort

Quicksort is a **comparison-based algorithm** that uses **divide-and-conquer** to sort an array.

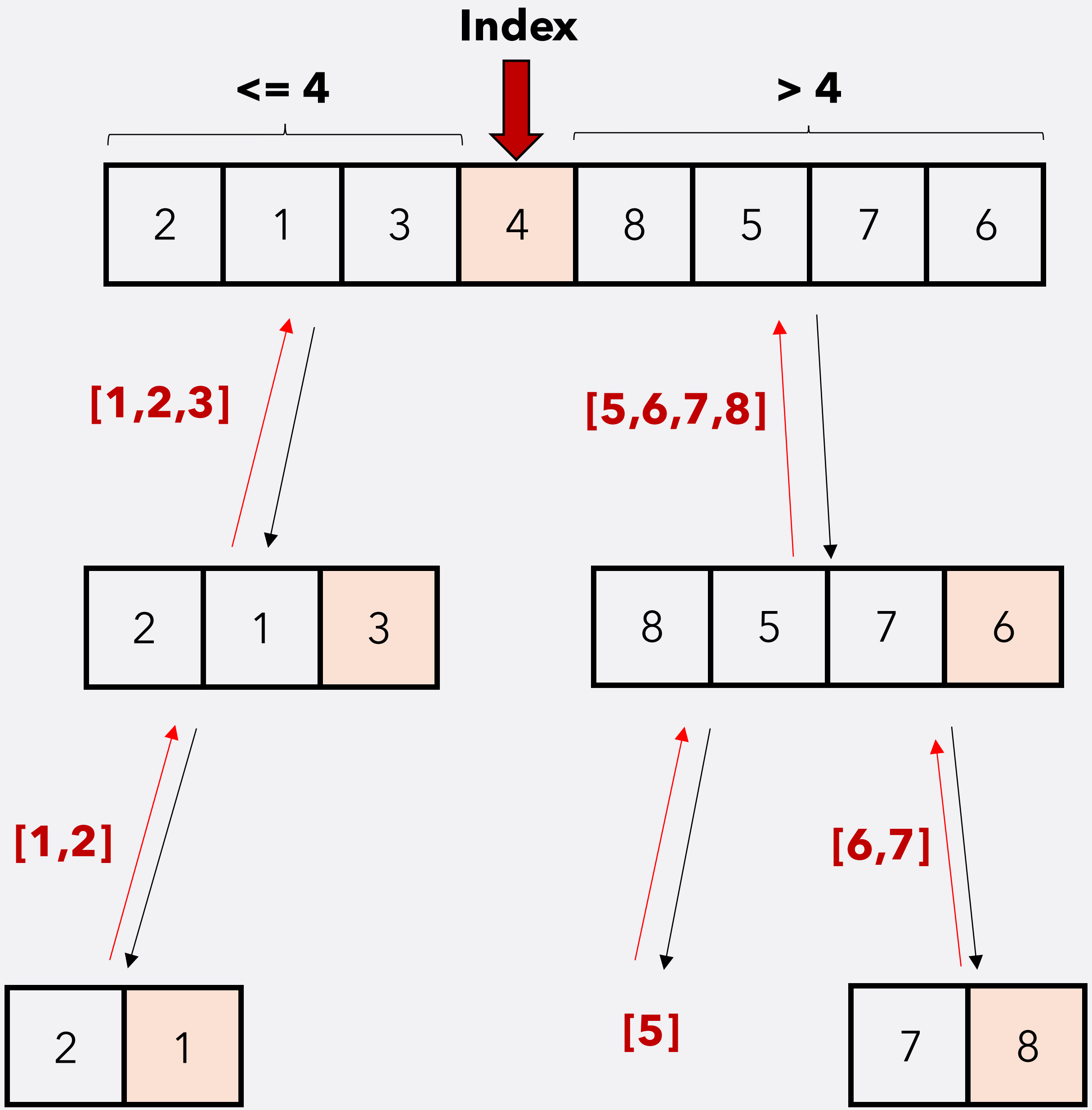
The algorithm picks a **pivot** element, **A[q]**, and then rearranges the array into two subarrays **A[p … q-1]**, such that all elements are **less** than **A[q]**, and **A[q+1…r]**, such that all elements are **greater** than or equal to **A[q]**.



```
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```

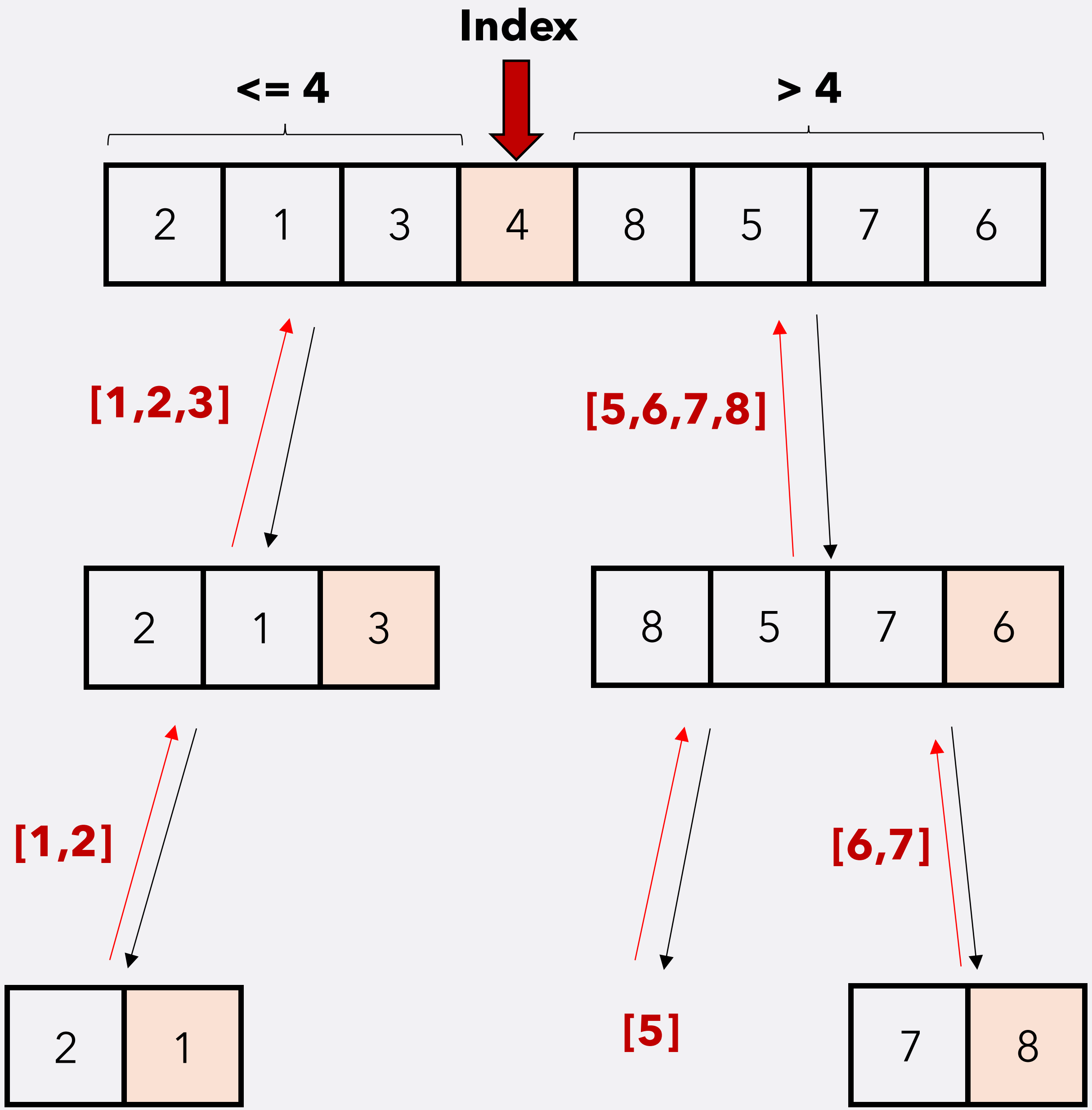


```
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```



Time analysis (best)

```
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```



$$T(n) = O(n) + 2*T(n/2)$$
$$= \Omega(n*\log n)$$

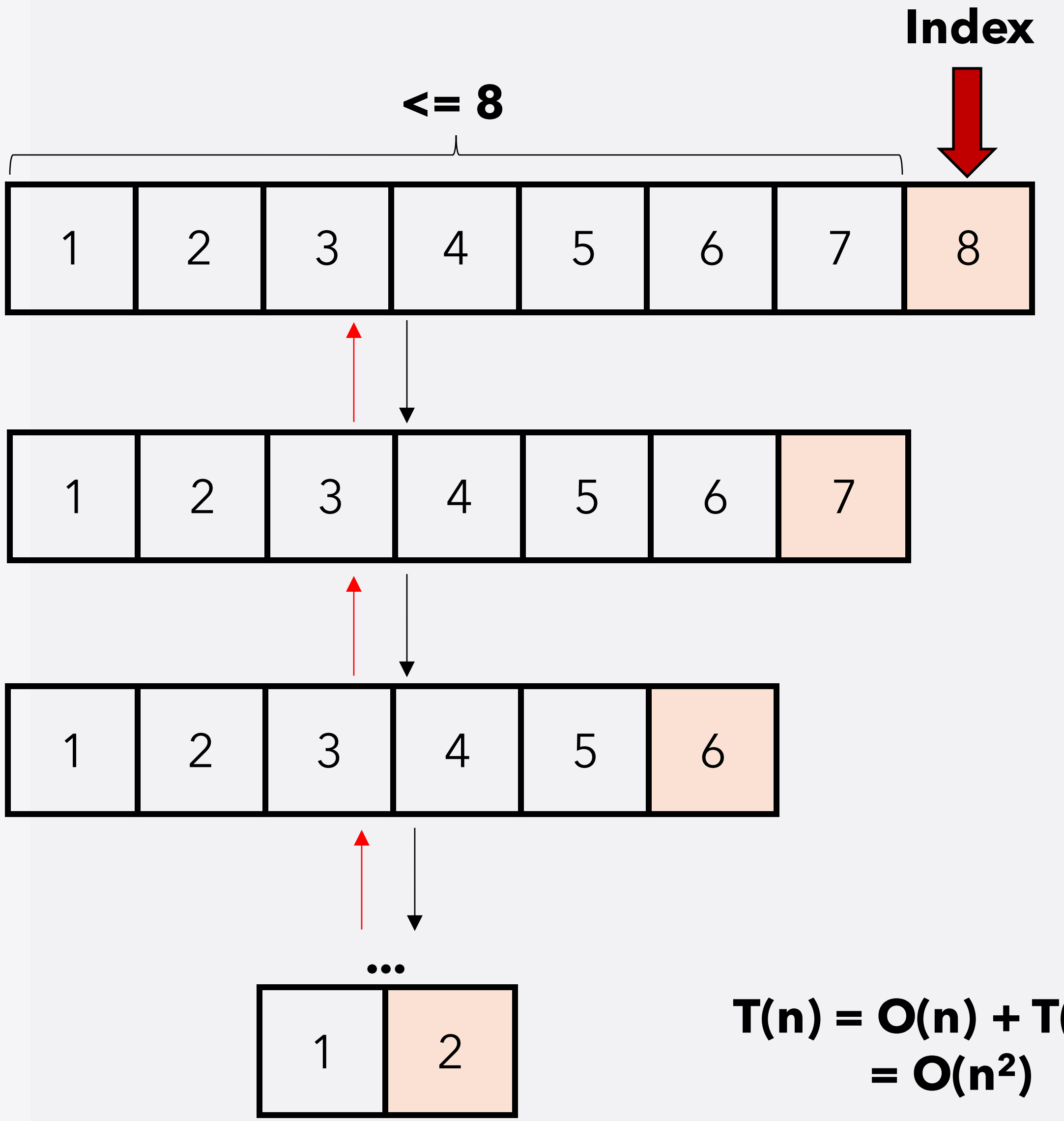
Quicksort Complexity

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

It occurs when the pivot element is always the middle element or near to the middle element.

Time analysis (worst)

```
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```



$$T(n) = O(n) + T(n-1) = O(n^2)$$

Is Quicksort a stable sort?

Quicksort Complexity

A 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 random pivots.

Time Complexity	
Best	**O(n*log n)**
Worst	**O(n^2)**
Average	**?**
Space Complexity	**?**
Stability	**?**

It occurs when the pivot element is always the middle element or near to the middle element.

It occurs when the pivot element picked is either the greatest or the smallest element.

Quicksort Complexity

Time Complexity		
Best	O(n*log n)	It occurs when the pivot element is always the middle element or near to the middle element.
Worst	O(n^2)	It occurs when the pivot element picked is either the greatest or the smallest element.
Average	O(n*log n)	It occurs when the above conditions do not occur.
Space Complexity	?	
Stability	?	

Quicksort Complexity

Time Complexity		
Best	O(n*log n)	It occurs when the pivot element is always the middle element or near to the middle element.
Worst	O(n^2)	It occurs when the pivot element picked is either the greatest or the smallest element.
Average	O(n*log n)	It occurs when the above conditions do not occur.
Space Complexity	O(log n)	
Stability	?	

Quicksort Complexity

Time Complexity		
Best	**O(n*log n)**	**It occurs when the pivot element is always the middle element or near to the middle element.**
Worst	**O(n^2)**	**It occurs when the pivot element picked is either the greatest or the smallest element.**
Average	**O(n*log n)**	**It occurs when the above conditions do not occur.**
Space Complexity	**O(log n)**	
Stability	**No**	

Quicksort Applications

Quicksort is used when:

- the programming language is good for recursion
- time complexity matters
- space complexity matters

Summary

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Act 1.3
- Comprehensive Activity of Basic Concepts and Fundamental Algorithms (Competency Evidence)

What do I have to do?

In teams of three, make an application that:

- Open the input file **"Bitacora.txt"**, which contains a server access log, with each entry in the format:

Month Day Time IP_address:Port_number Access_error_message

```
Oct 9 10:32:24 423.2.230.77:6166 Failed password for illegal user guest
Aug 28 23:07:49 897.53.984.6:6710 Failed password for root
Aug 4 03:18:56 960.96.3.29:5268 Failed password for admin
Jun 20 13:39:21 118.15.416.57:4486 Failed password for illegal user guest
...
```

- Sort the information by **date (Month/Day/Time)** with **preferred sort-algorithm**. compare **>month**, **>day**, **>time**.
- Ask the user for the information search in that range of dates (**start date**, **end date**).
- Display the entries corresponding to those dates. **All entries within this date range, sorted in ascending order**.
- Store the sorting result in a file. **Txt file**.

How to read a Txt

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
using namespace std;

int main()
{
    system("cls");

    vector<string> v;

    //Make sure you have the CPP file and the TXT file in the same folder.
    ifstream InputFile("bitacora.txt");

    if (!InputFile)
    {
        cerr << "Error opening the input file." << endl;
        return 0;
    }

    // Read and store each entry in the vector
    string entry;
    while(InputFile)
    {
        getline(InputFile, entry);
        v.push_back(entry);
    }

    // Closes the file currently associated
    InputFile.close();

    // Print the first 5 entries
    for(int i=0; i<5; i++)
    {
        cout << v[i] << endl;
    }
    cout << endl;

    return 0;
}
```

```
Oct 9 10:32:24 423.2.230.77:6166 Failed password for illegal user guest
Aug 28 23:07:49 897.53.984.6:6710 Failed password for root
Aug 4 03:18:56 960.96.3.29:5268 Failed password for admin
Jun 20 13:39:21 118.15.416.57:4486 Failed password for illegal user guest
Jun 2 18:37:41 108.57.27.85:5491 Failed password for illegal user guest
```

How to split strings

```
int main()
{
    system("cls");

    string s= "Oct 9 10:32:24 423.2.230.77:6166 Failed password for illegal user guest";

    cout << "Split bitacora entry:\n";

    string sa, sb, sc;
    char ch;

    stringstream SS(s);
    SS >> sa >> ch >> sb >> sc;

    cout << sa << endl;
    cout << ch << endl;
    cout << sb << endl;
    cout << sc << endl;

    cout << "\n\nSplit IP address:\n";

    int a,b,c,d;

    SS.clear(); //reset
    SS.str(sc); //new value
    SS >> a >> ch >> b >> ch >> c >> ch >> d;

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;

    return 0;
}
```

```
Split bitacora entry:
Oct
9
10:32:24
423.2.230.77:6166


Split IP address:
423
2
230
77
```