Data Structures – CST 201
Module - 5

Syllabus

- **Sorting Techniques**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
- **Hashing**
  - Hashing Techniques
  - Collision Resolution
  - Overflow handling
  - Hashing functions: Mid square, Division, Folding, Digit Analysis

Bubble Sort

- In this method find the largest element in the array and this element is stored in the last position. Then find the second largest element and it is stored in the second last position, and so on

Bubble Sort

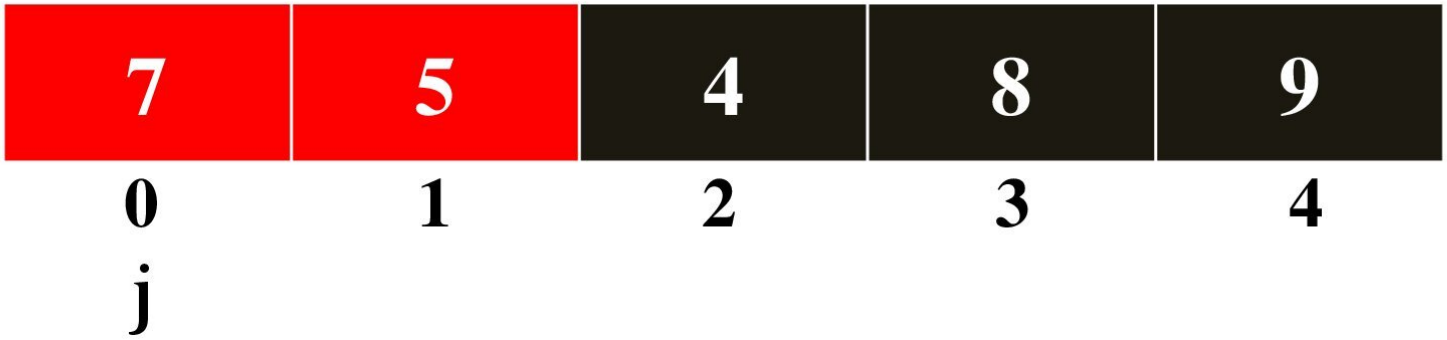
Bubble Sort

Number of Scans = 5-1 = 4

Bubble Sort

Scan-1

Bubble Sort



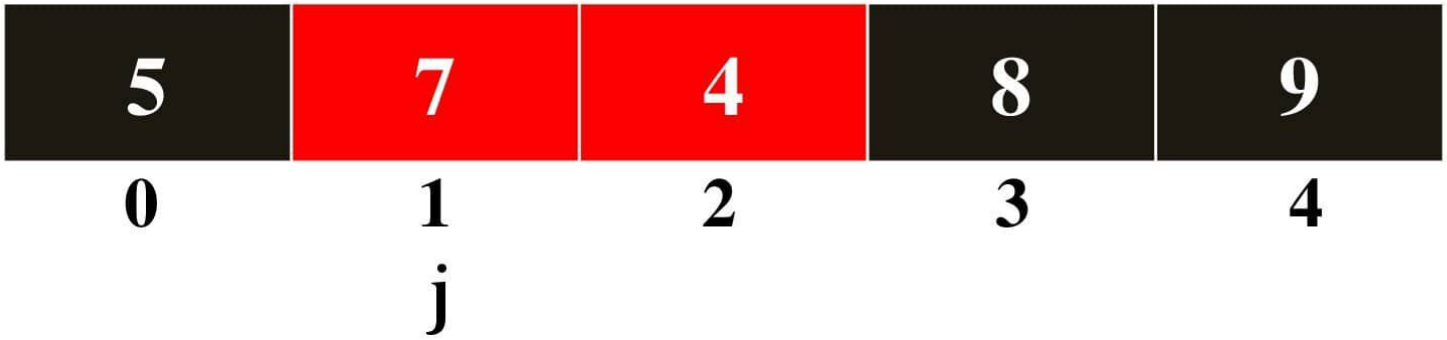
Scan-1

Swap the elements

Bubble Sort

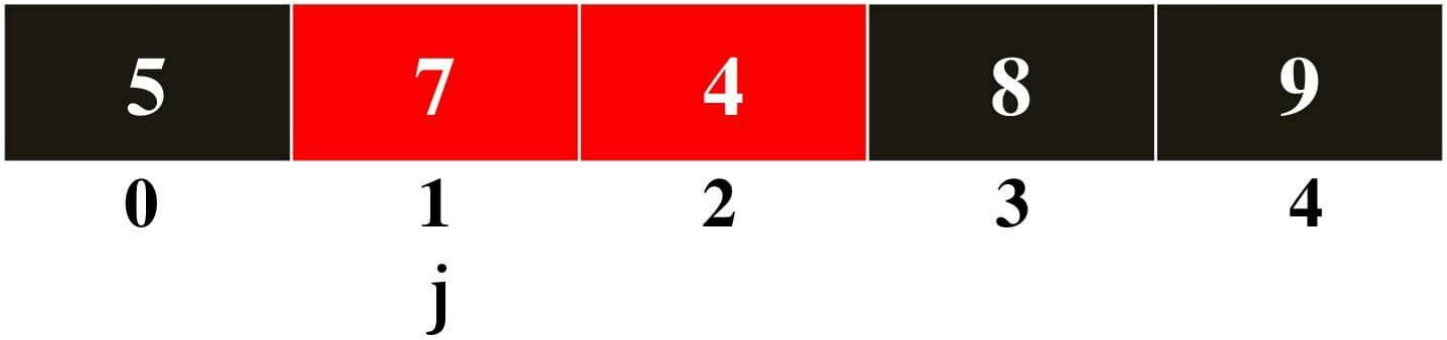
Scan-1

Bubble Sort



Scan-1

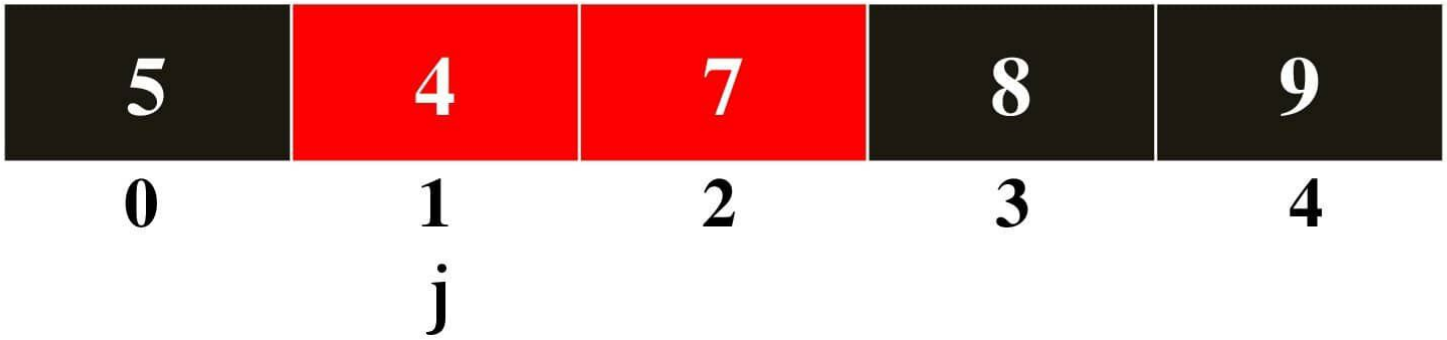
Bubble Sort



Scan-1

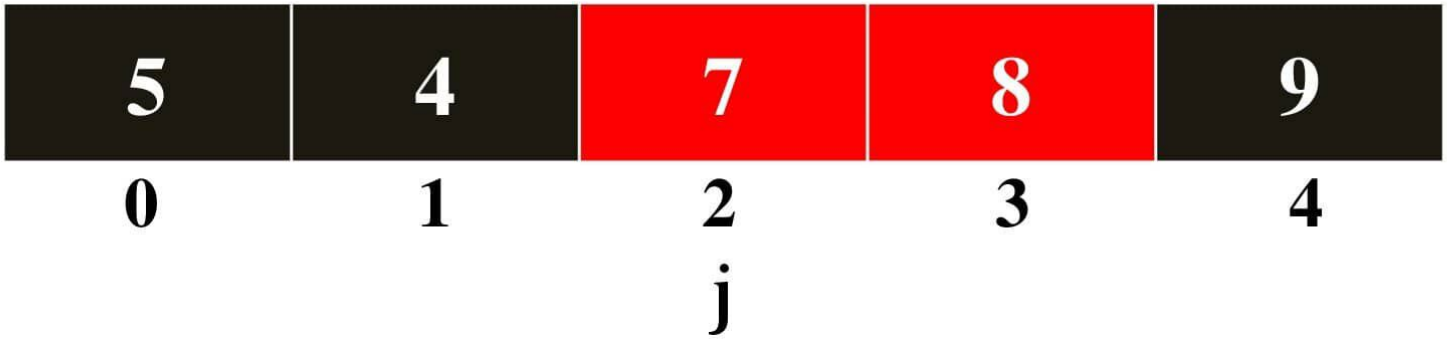
Swap the elements

Bubble Sort



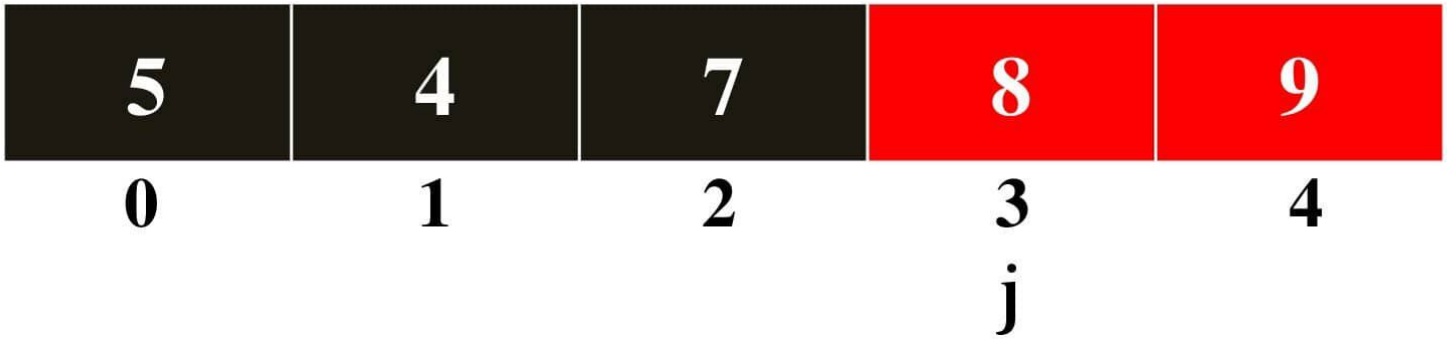
Scan-1

Bubble Sort



Scan-1

Bubble Sort



Scan-1

Bubble Sort

Scan-2

Bubble Sort

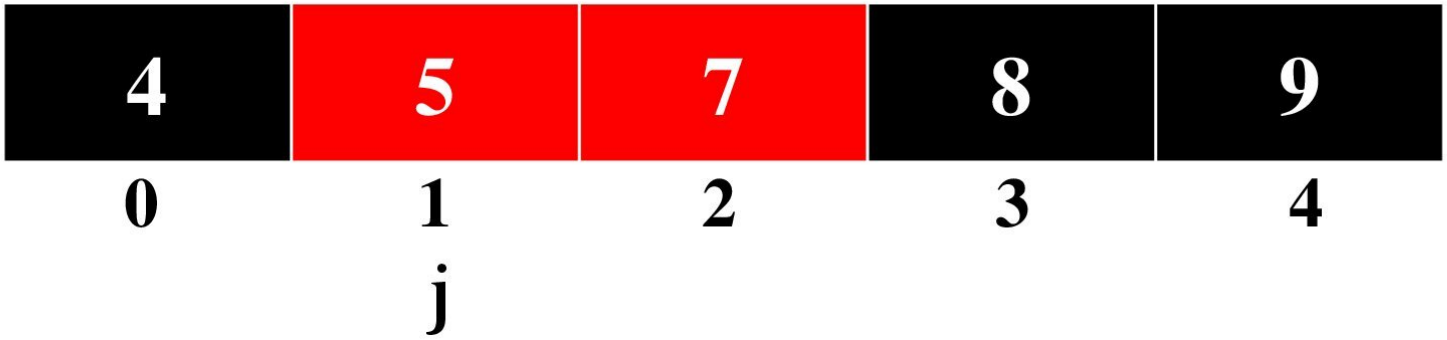
Scan-2

Swap the elements

Bubble Sort

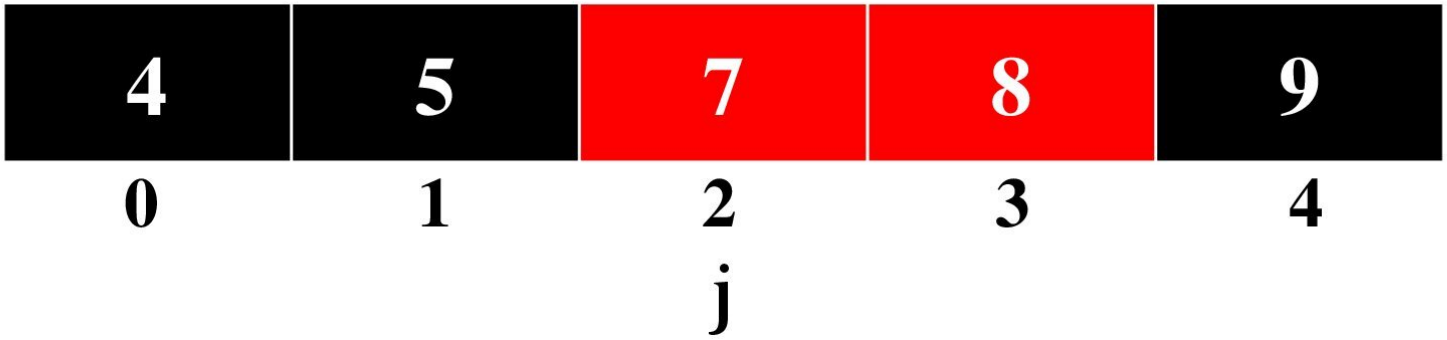
Scan-2

Bubble Sort



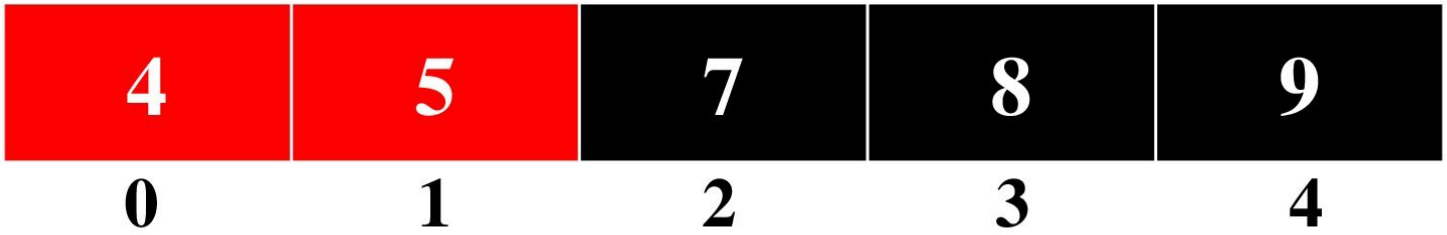
Scan-2

Bubble Sort



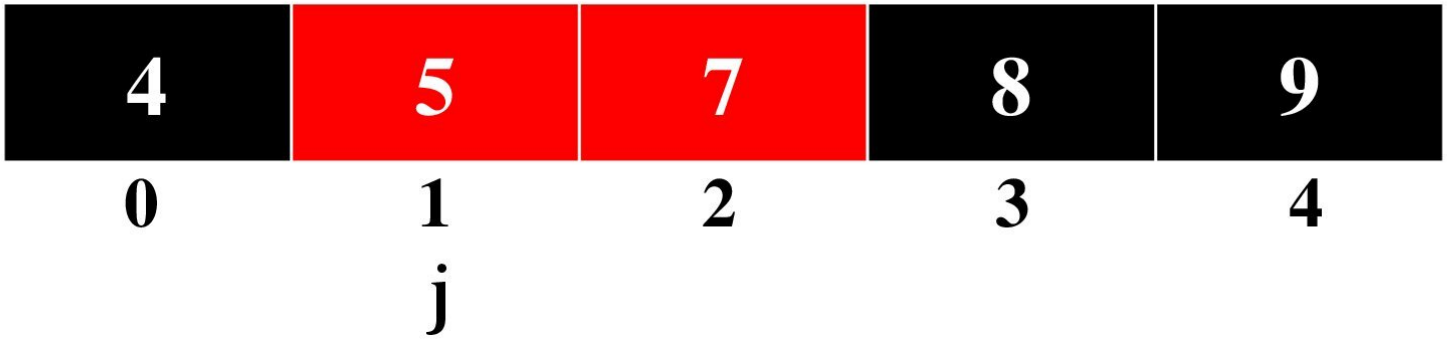
Scan-2

Bubble Sort



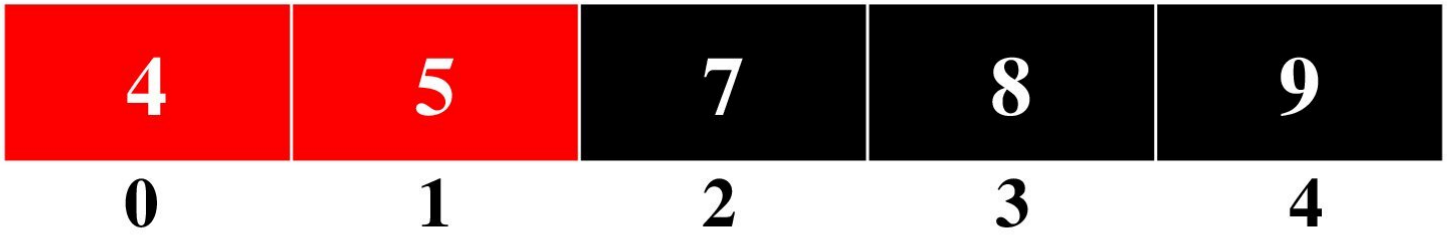
Scan-3

Bubble Sort



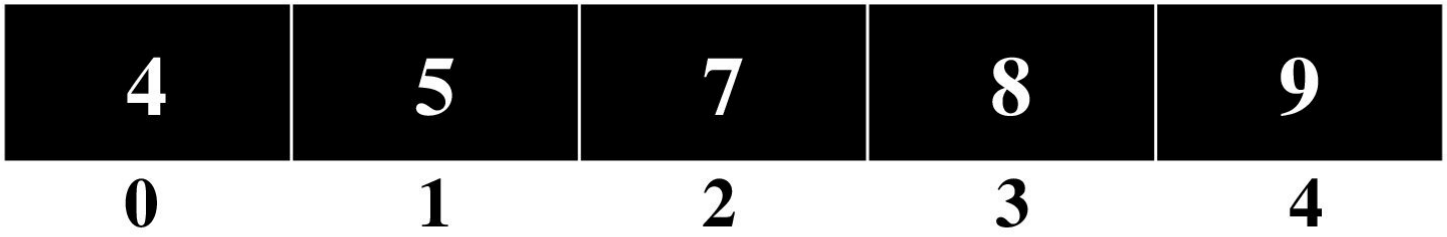
Scan-3

Bubble Sort



Scan-4

Bubble Sort



Final Sorted Array

Bubble Sort

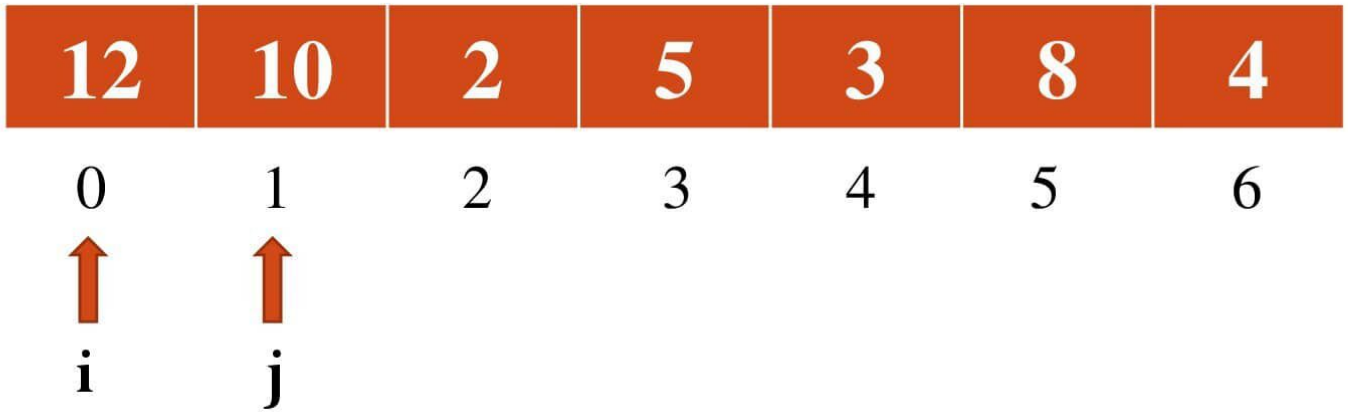
Algorithm BubbleSort(A, n)

```
{
    for i=1 to n-1 do
    {
        for j=0 to n-i-1 do
        {
                If A[j]>A[j+1] then
                        Swap A[j] and A[j+1]
        }
    }
}
```

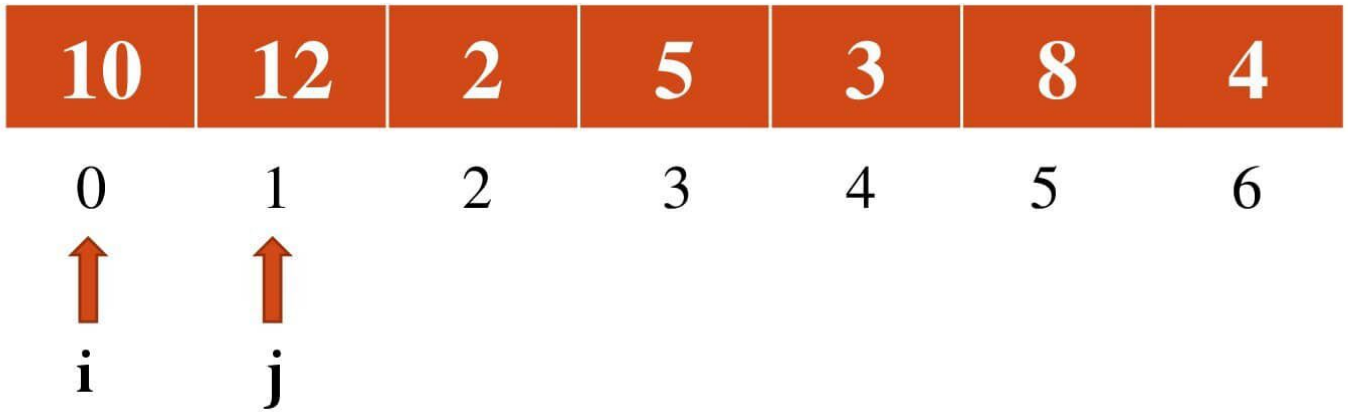
Selection Sort

- In this method find the smallest element in the array and this element is stored in the first position. Then find the second smallest element and it is stored in the second position, and so on

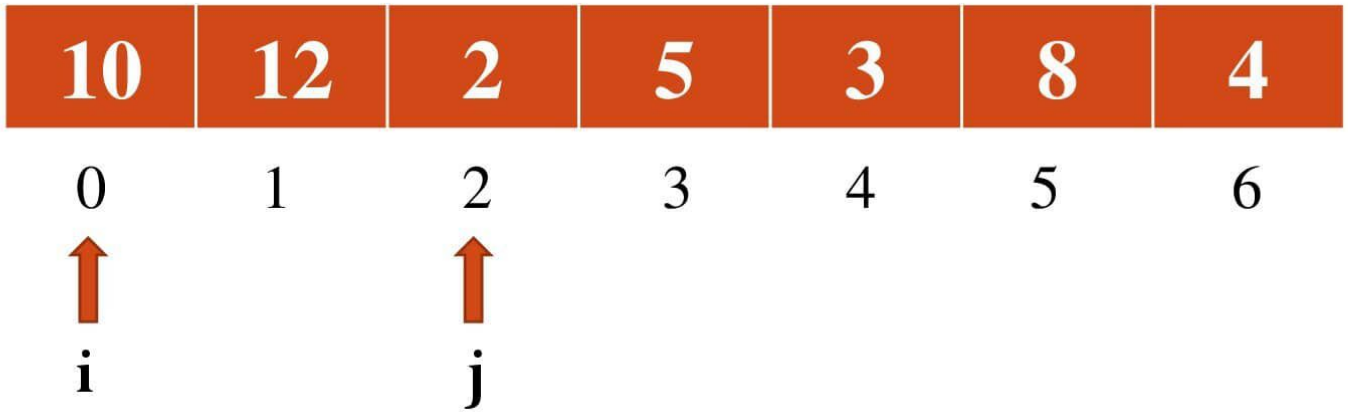
Selection Sort



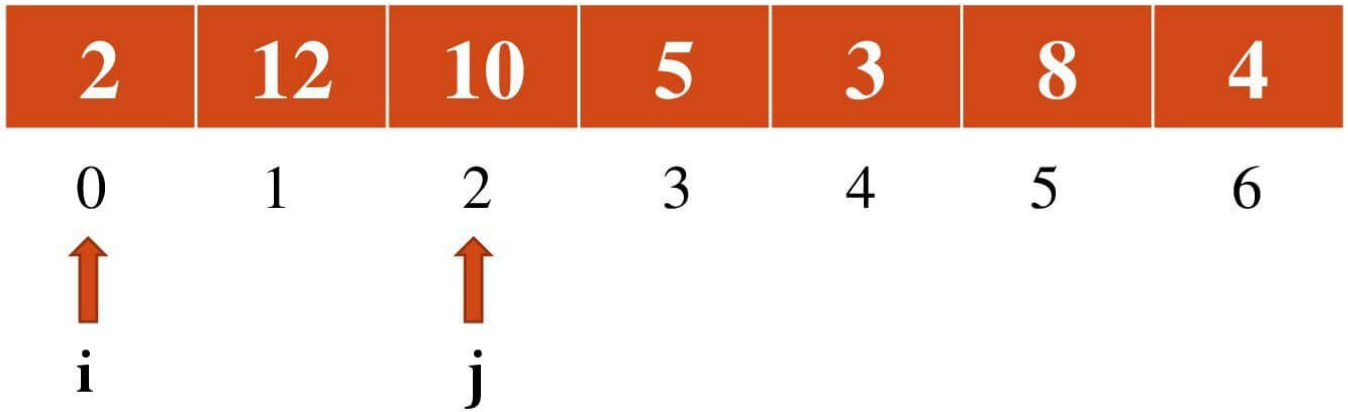
Selection Sort



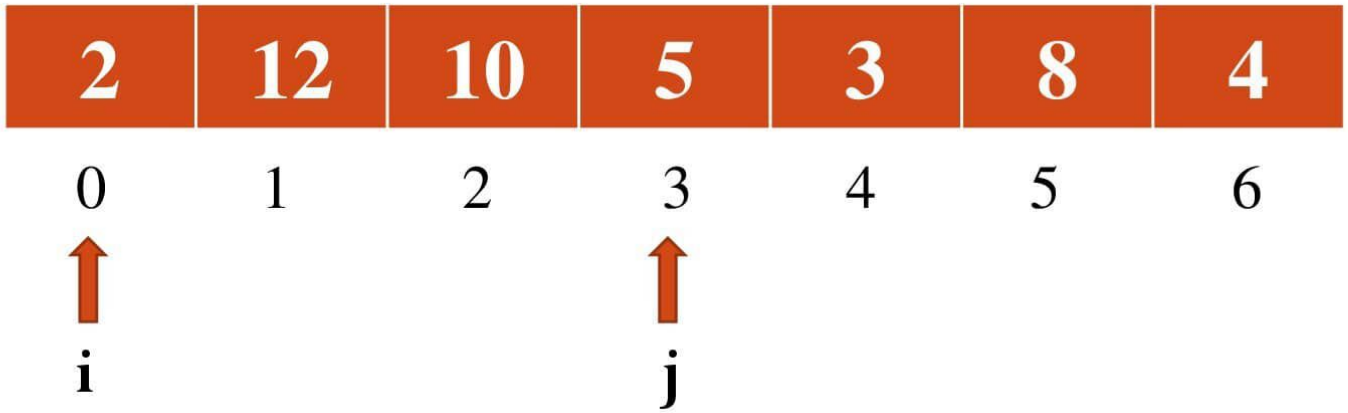
Selection Sort



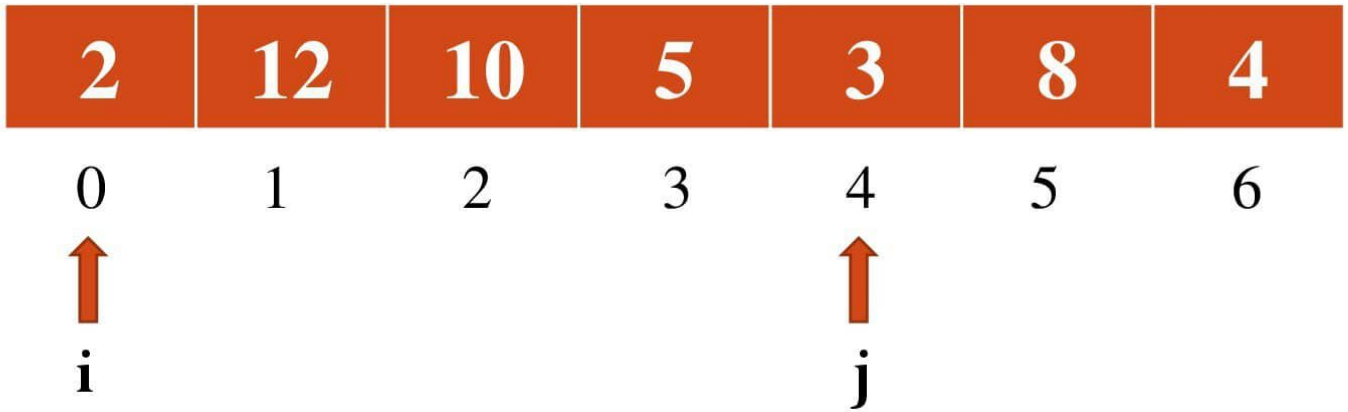
Selection Sort



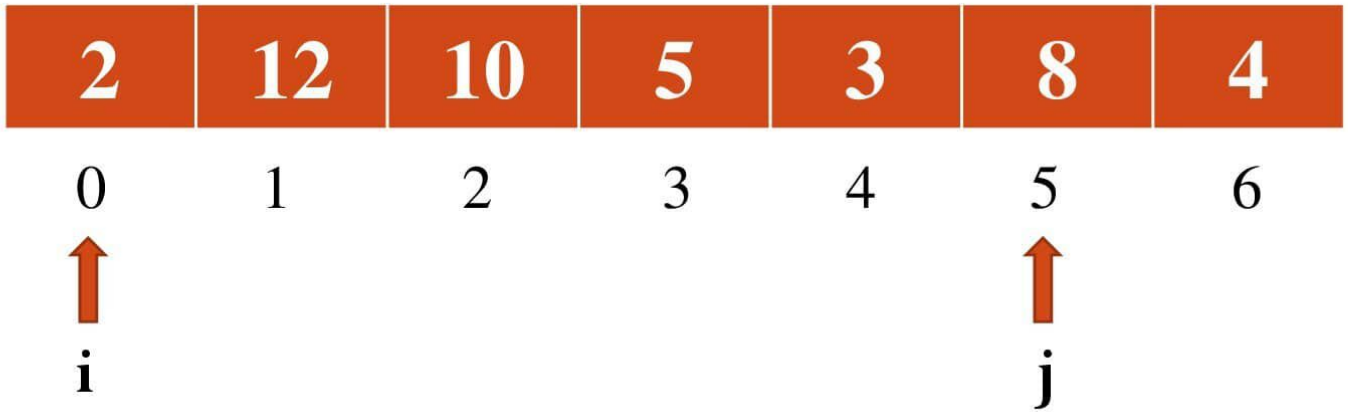
Selection Sort



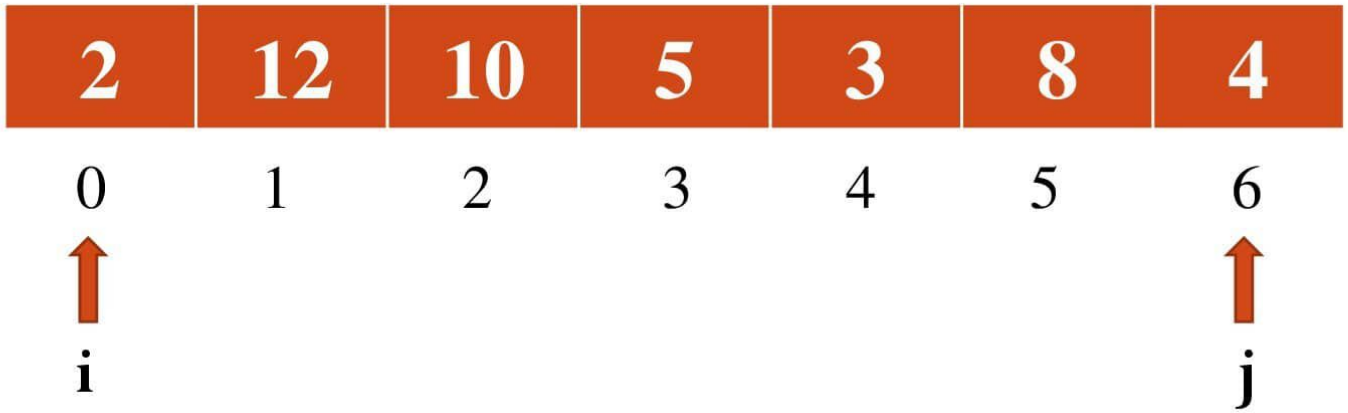
Selection Sort



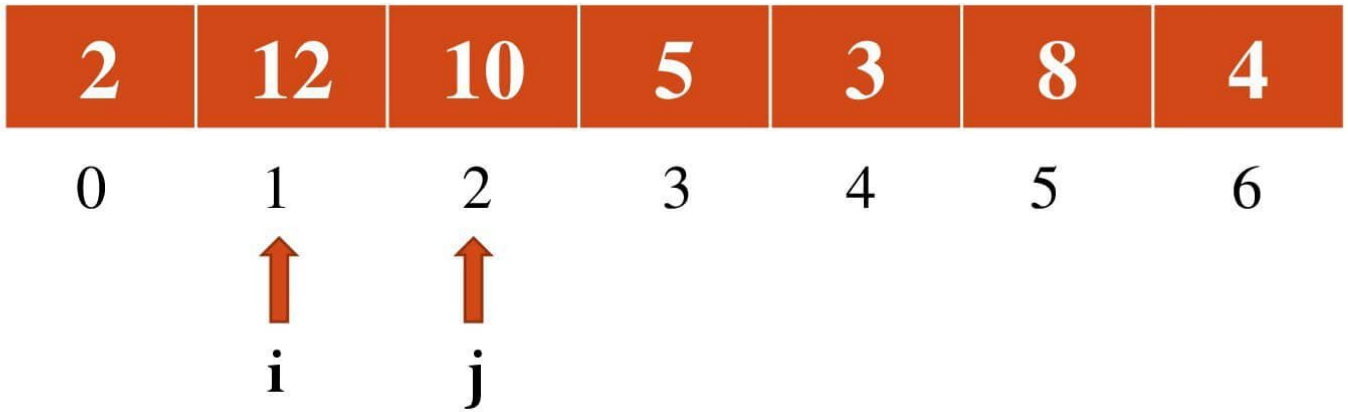
Selection Sort



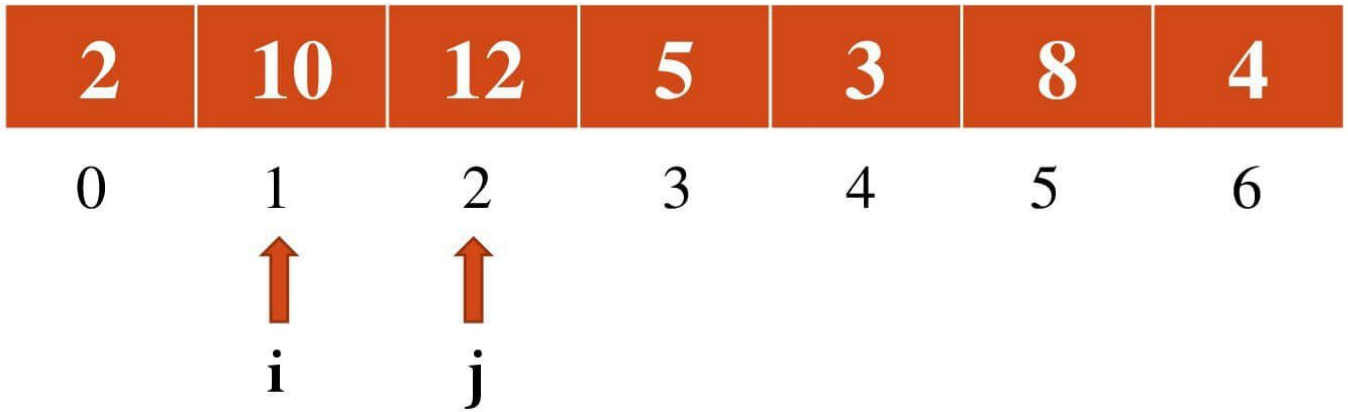
Selection Sort



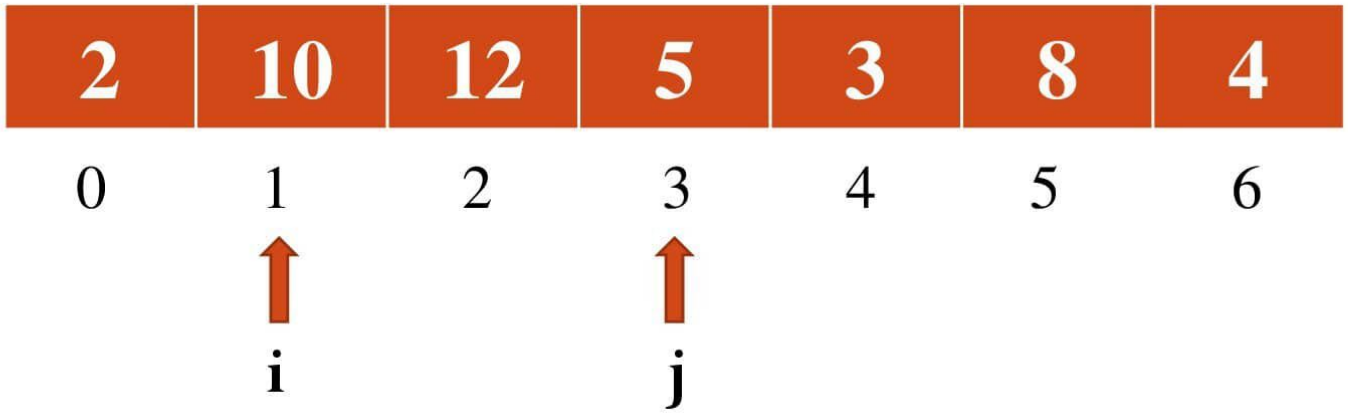
Selection Sort



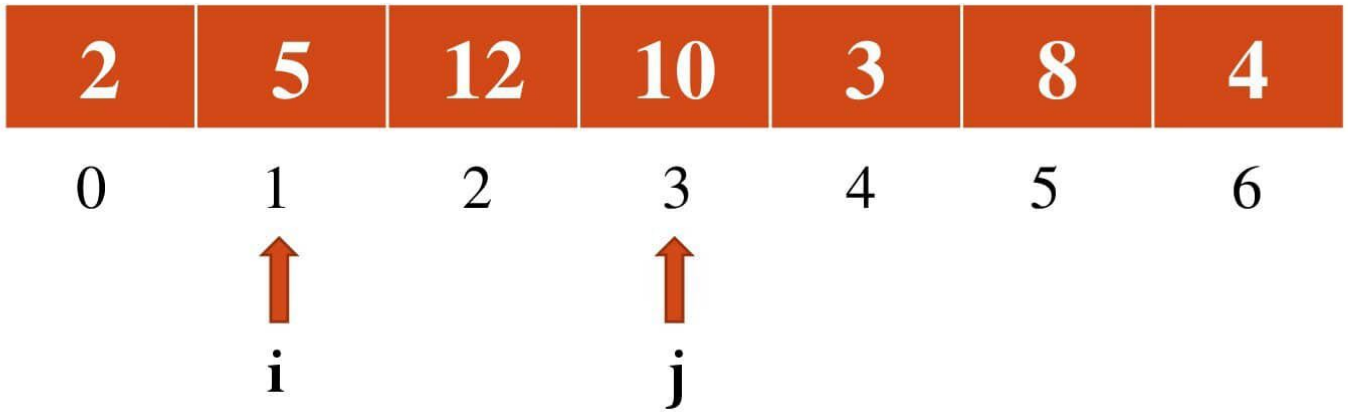
Selection Sort



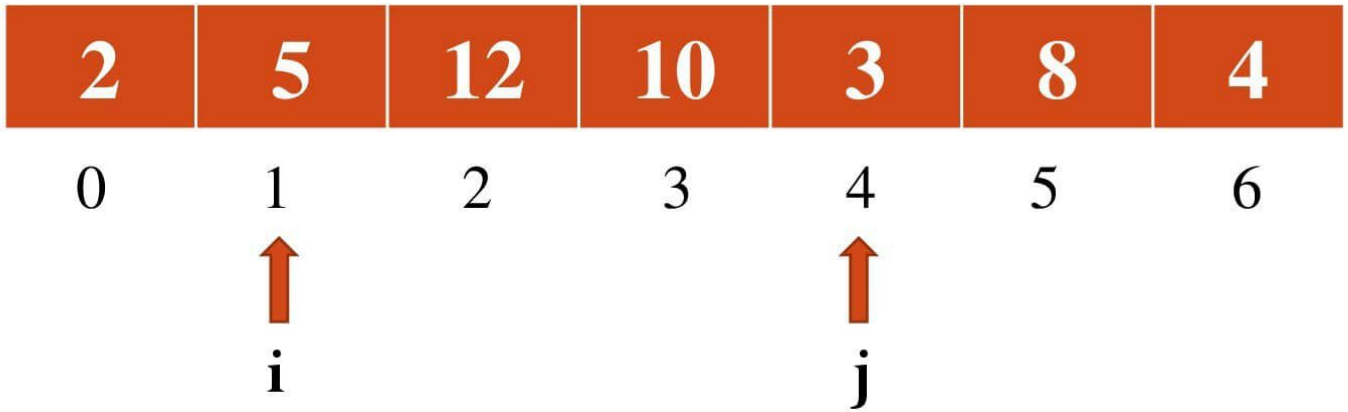
Selection Sort



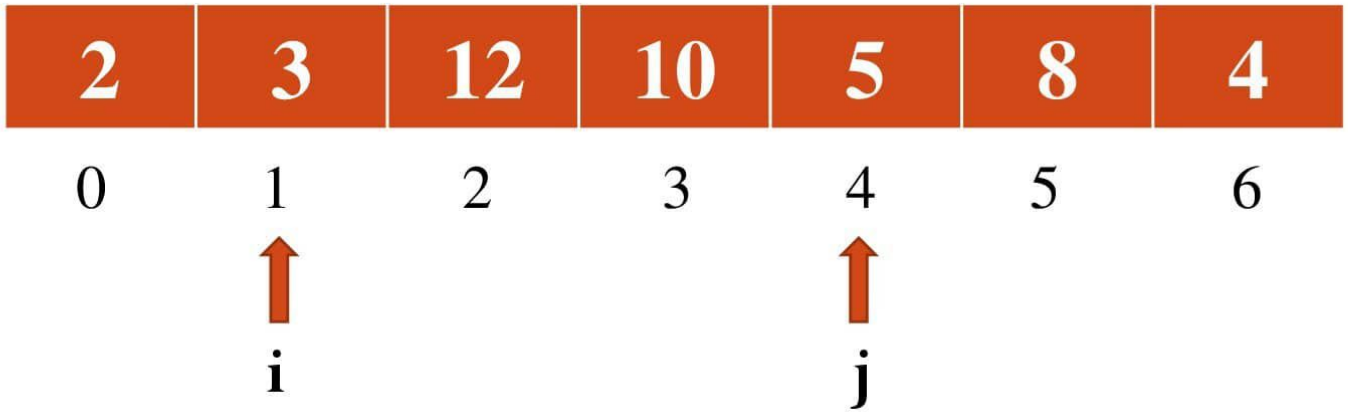
Selection Sort



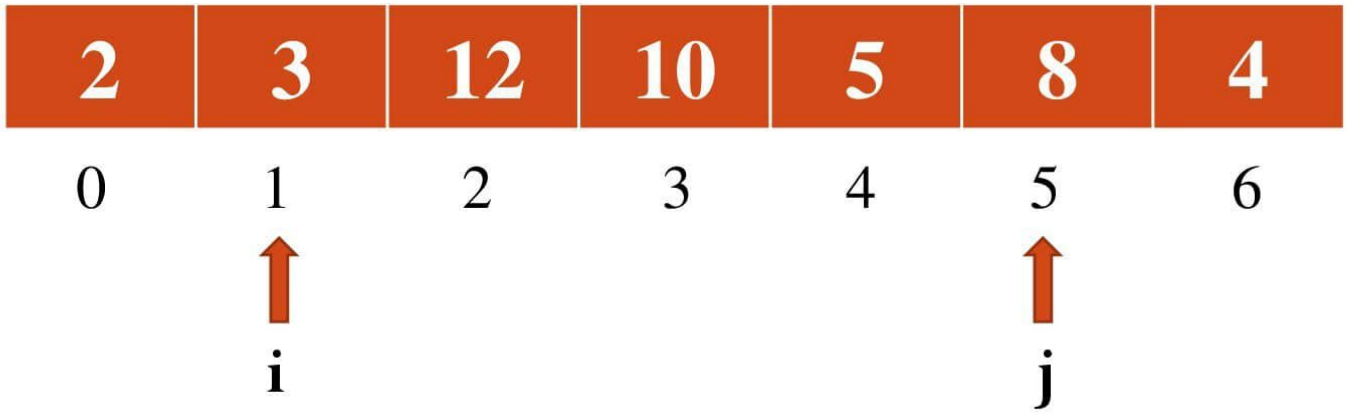
Selection Sort



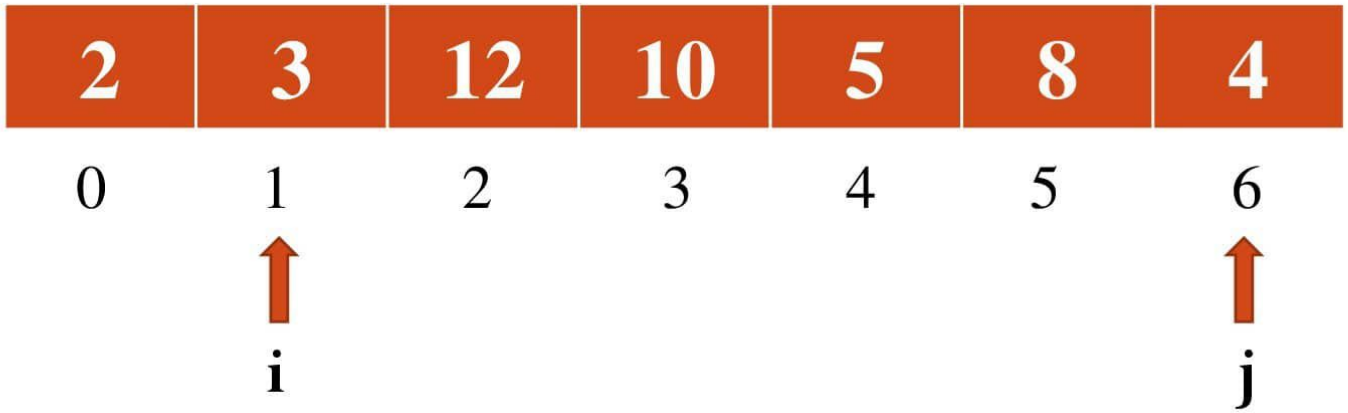
Selection Sort



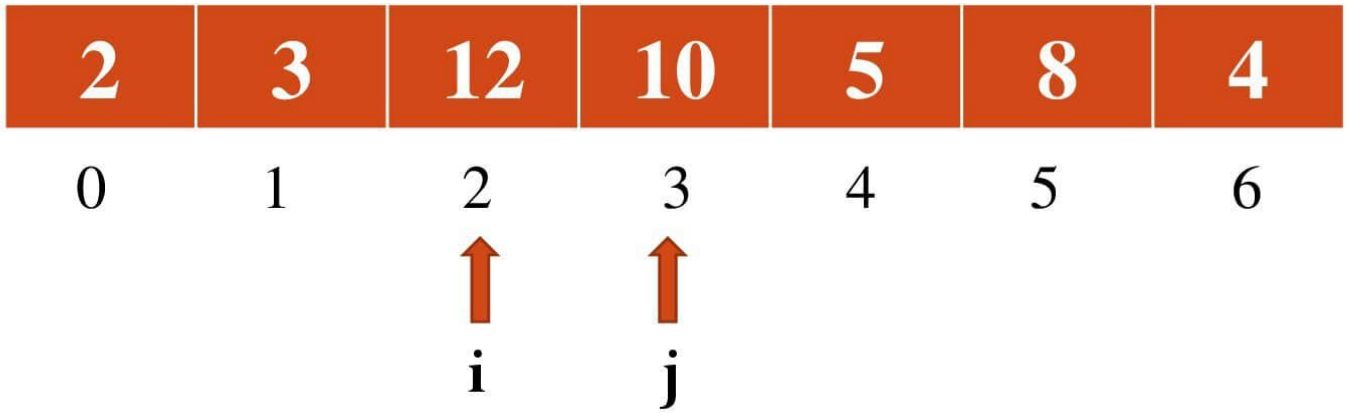
Selection Sort



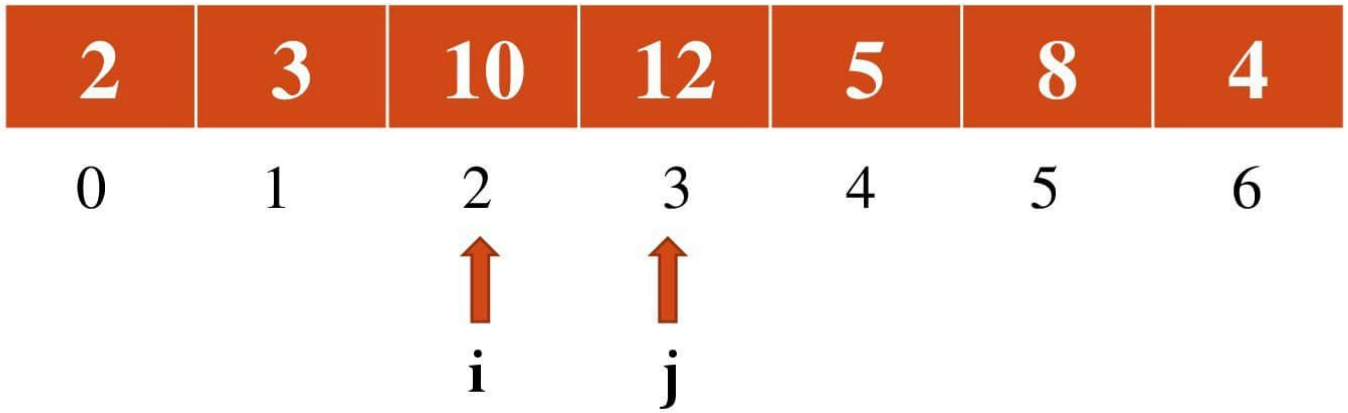
Selection Sort



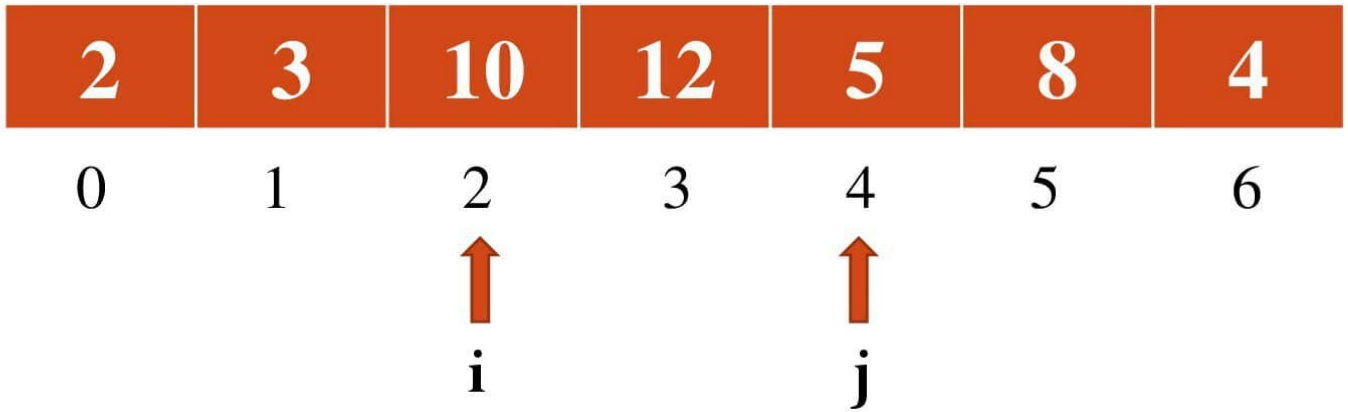
Selection Sort



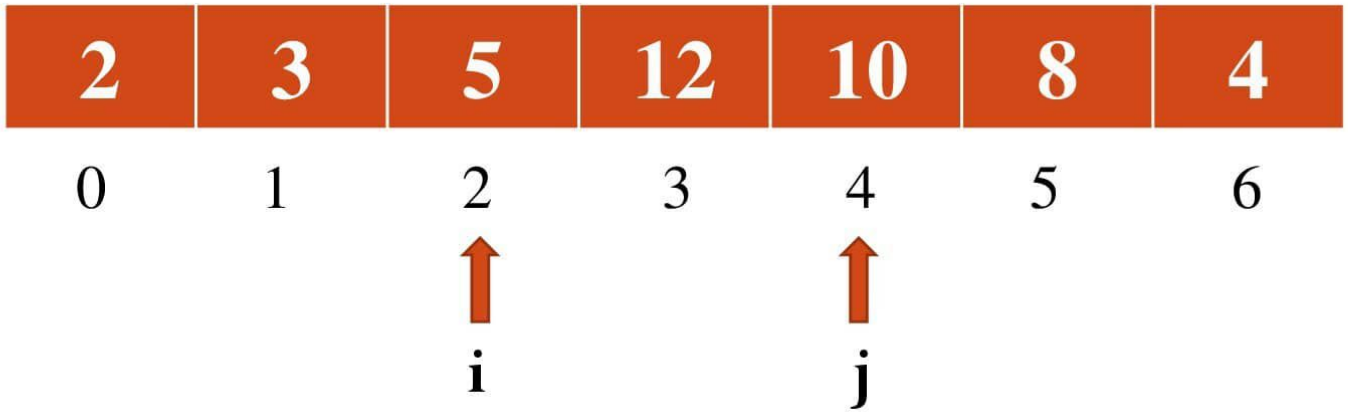
Selection Sort



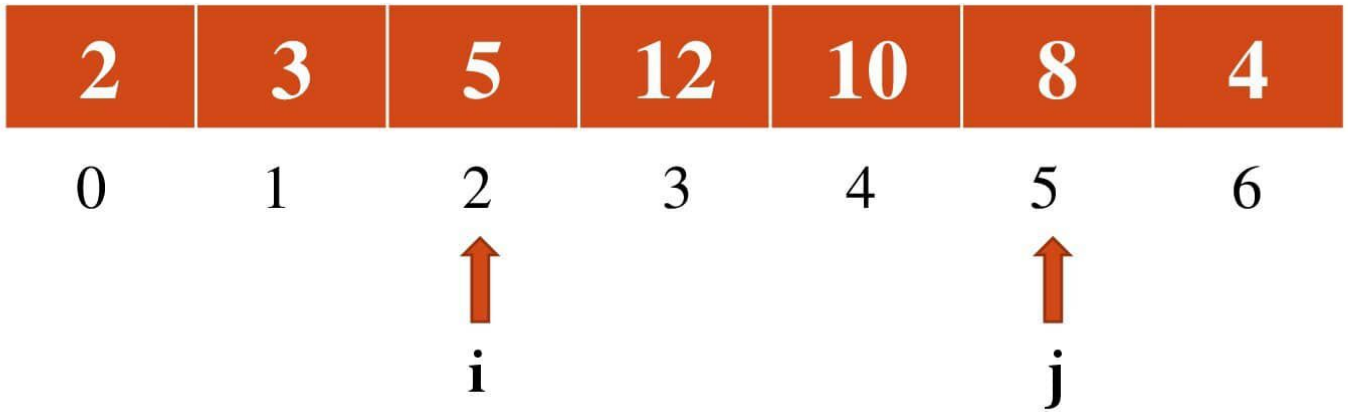
Selection Sort



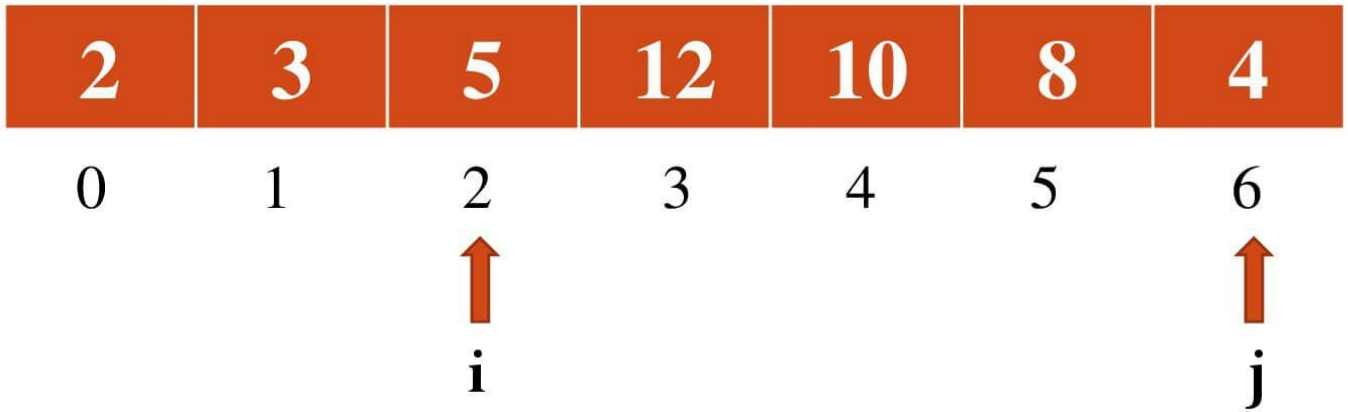
Selection Sort



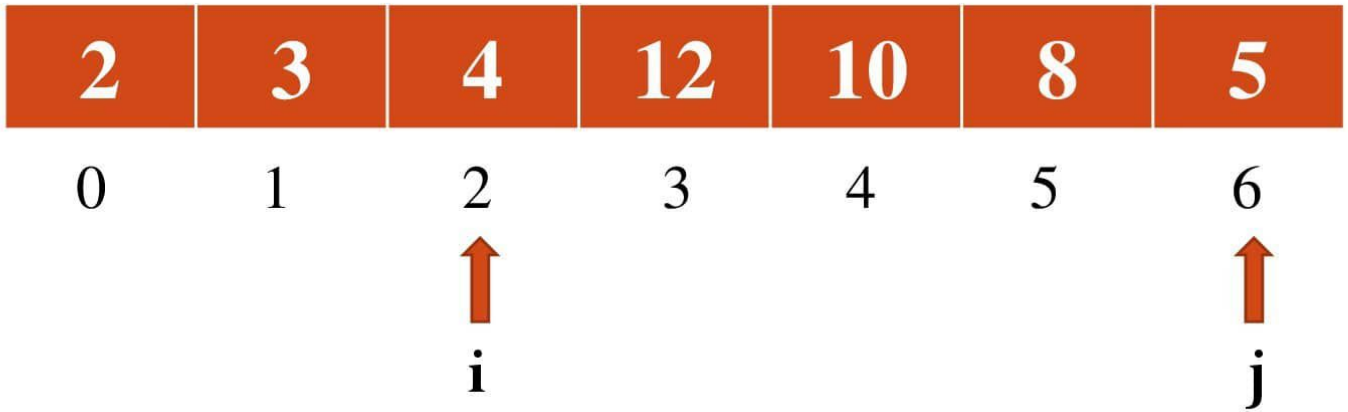
Selection Sort



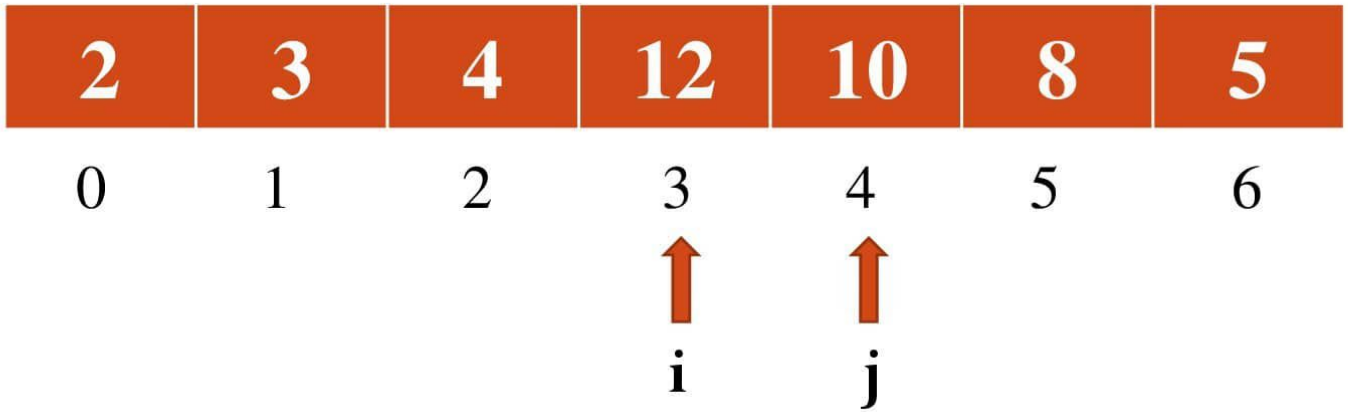
Selection Sort



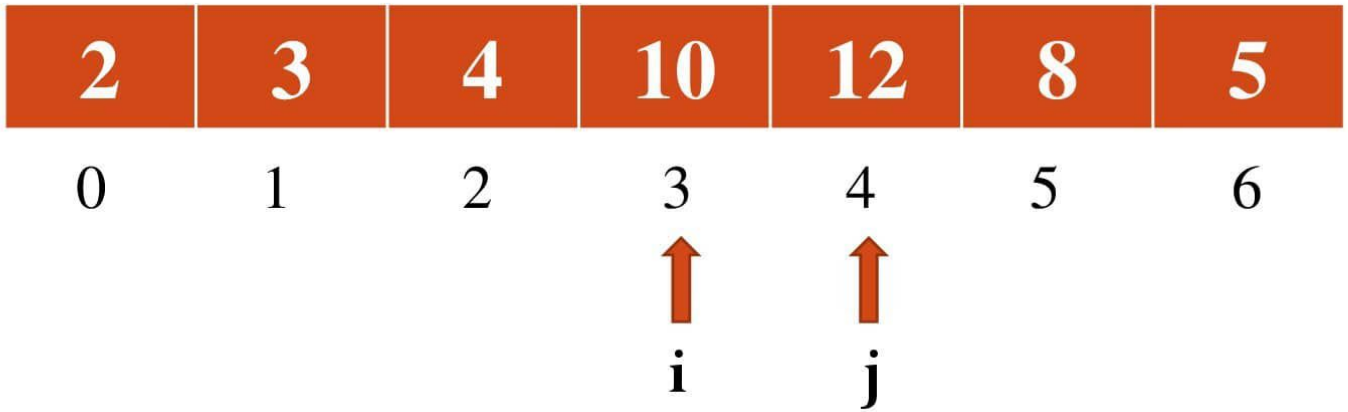
Selection Sort



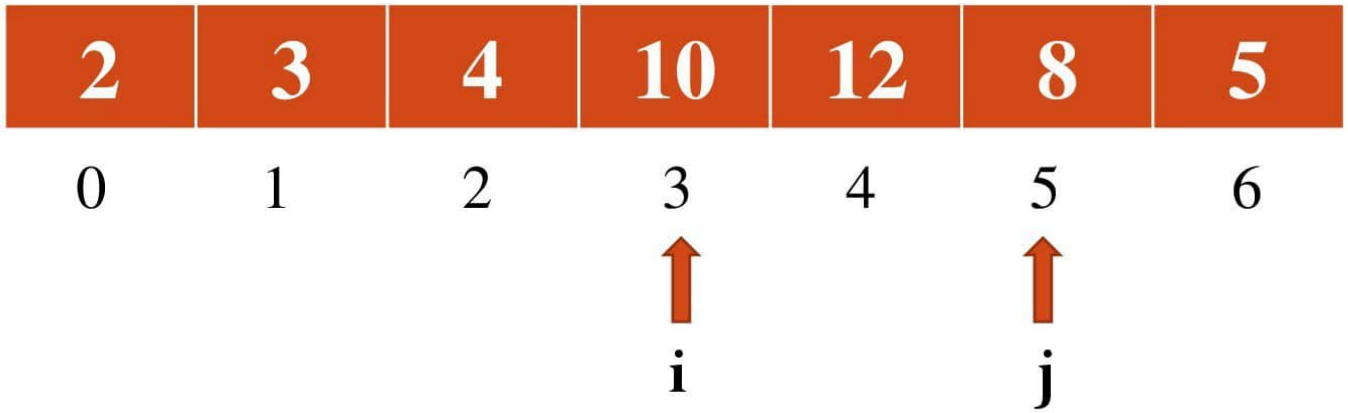
Selection Sort



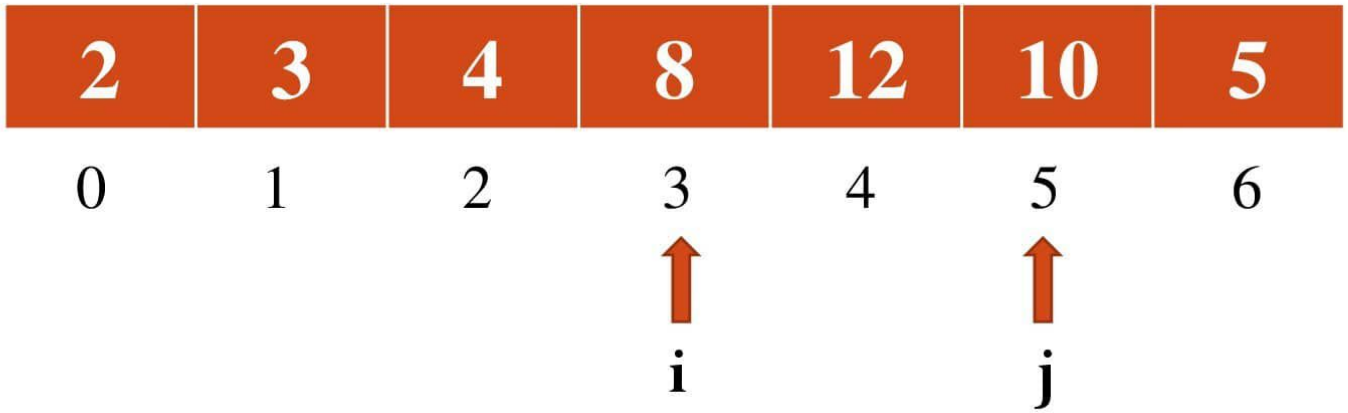
Selection Sort



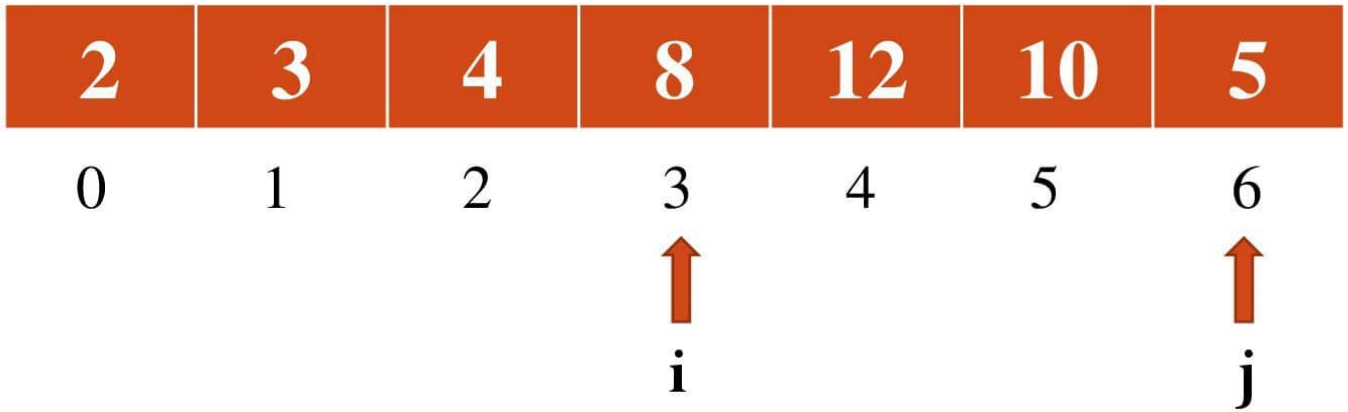
Selection Sort



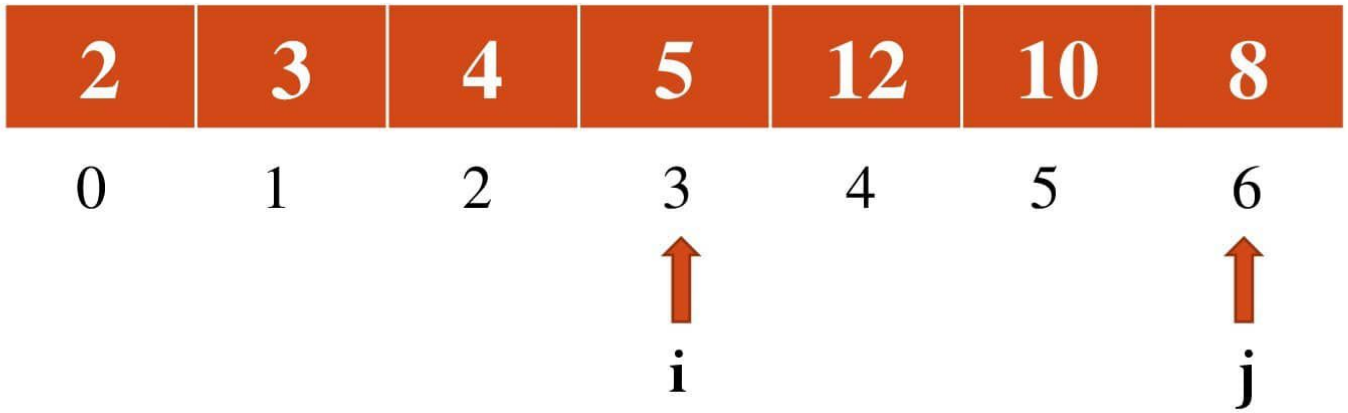
Selection Sort



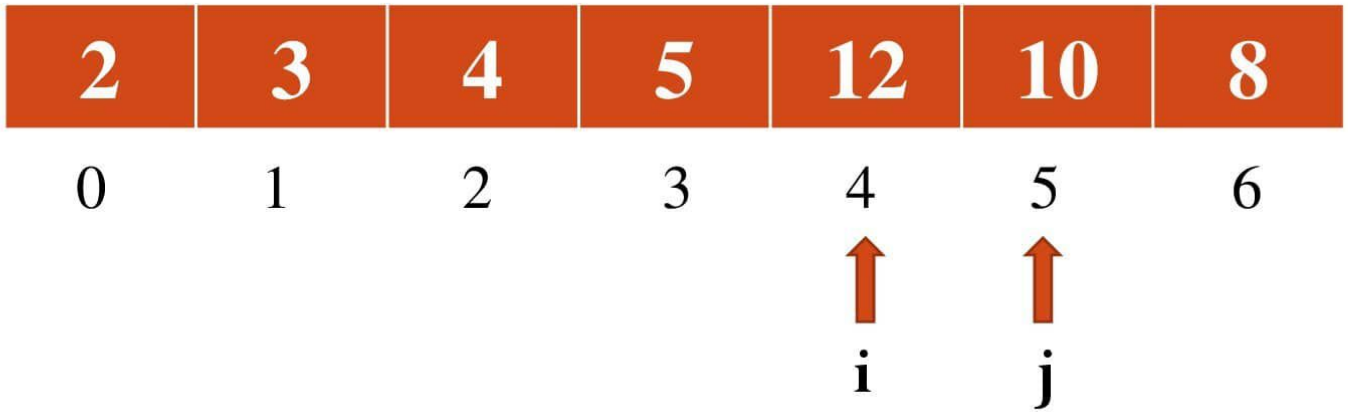
Selection Sort



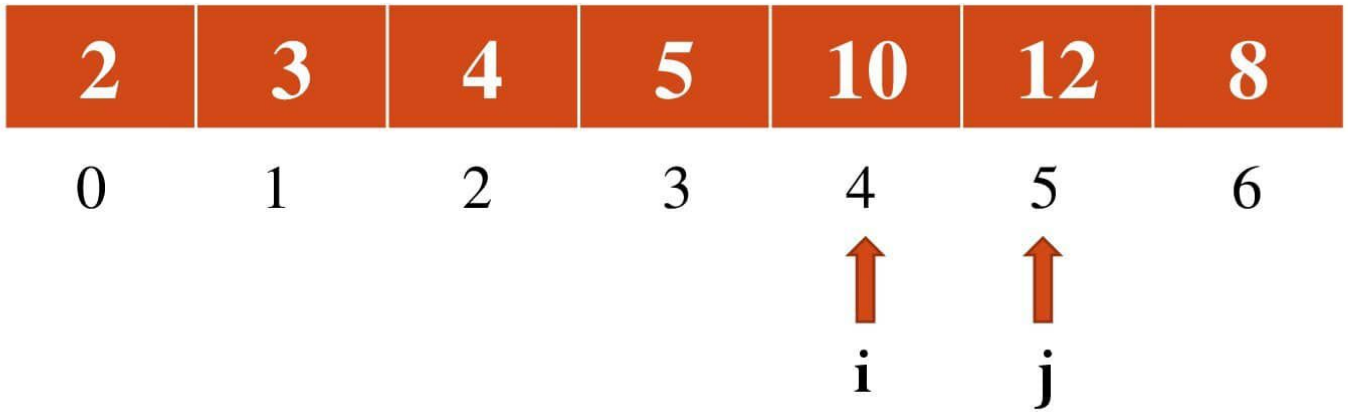
Selection Sort



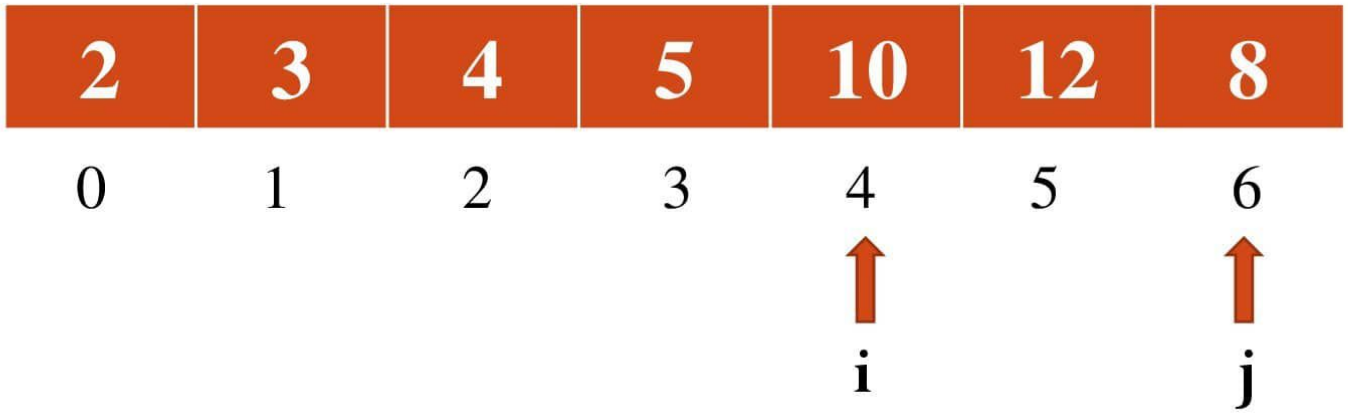
Selection Sort



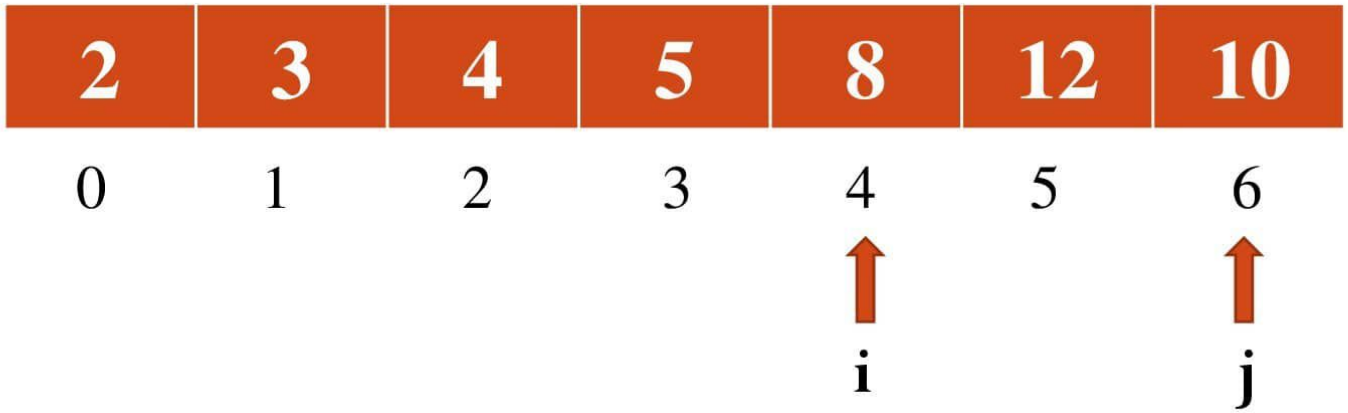
Selection Sort



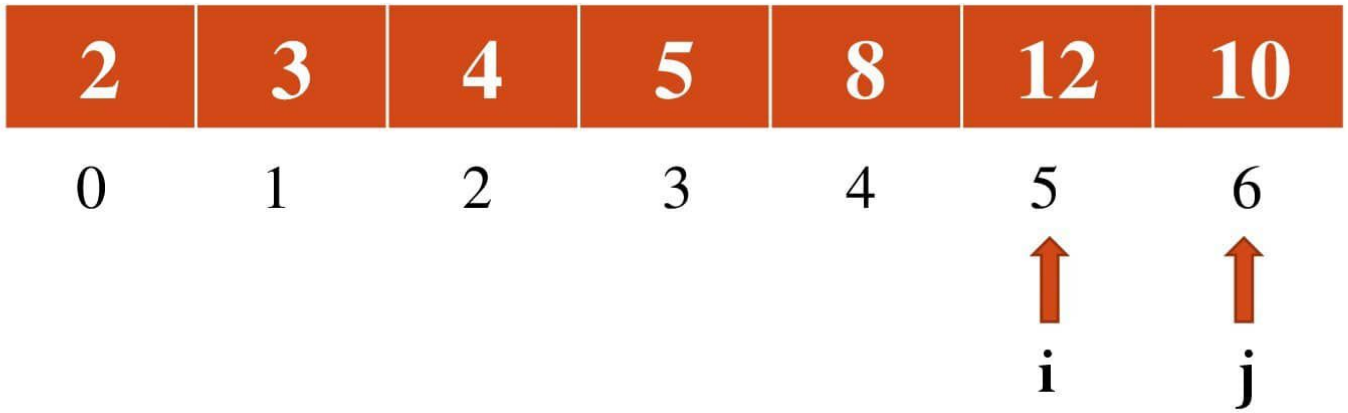
Selection Sort



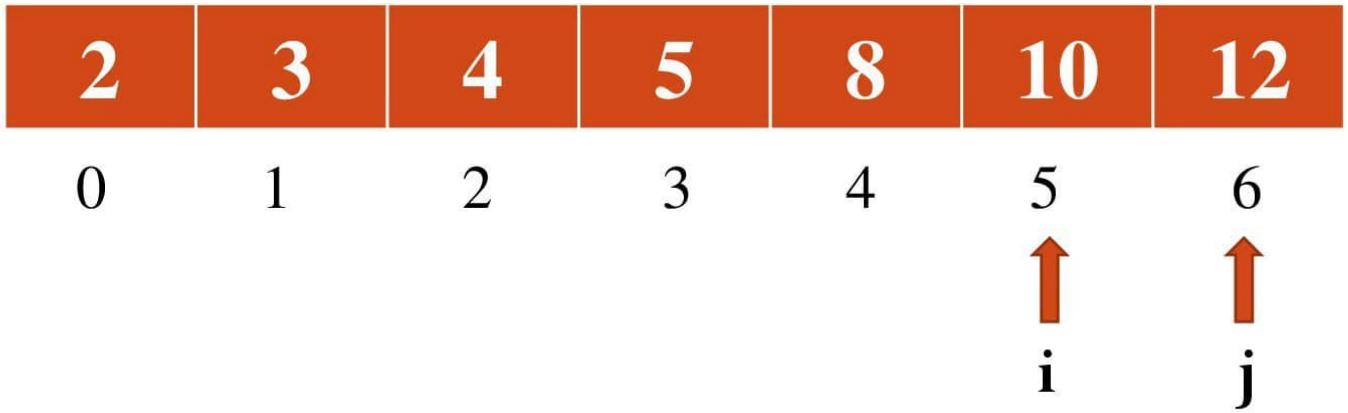
Selection Sort



Selection Sort



Selection Sort



Selection Sort

Selection Sort

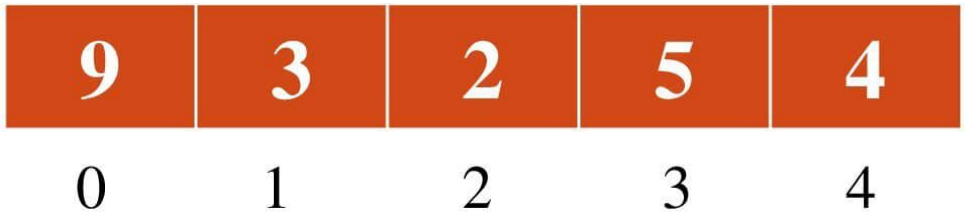
Algorithm SelectionSort(A, n)

```
{
    for i=0 to n-2 do
    {
        for j=i+1 to n-1 do
        {
                If A[i]>A[j] then
                        Swap A[i] and A[j]
        }
    }
}
```

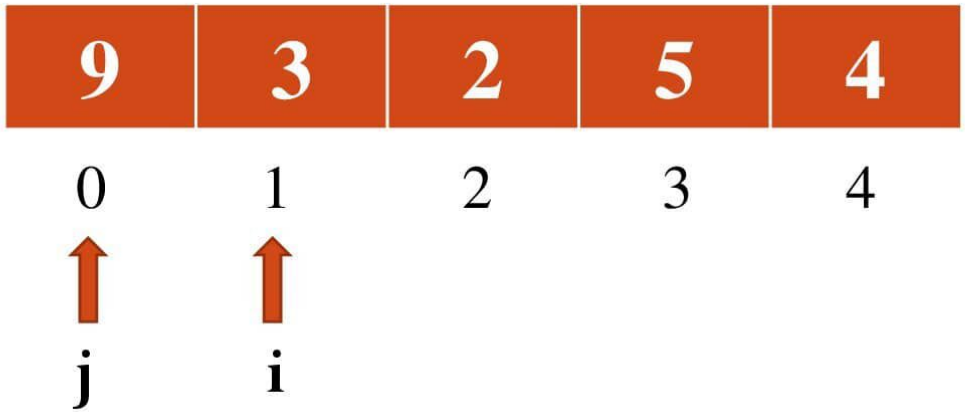
Insertion Sort

- During first iteration element in pos 1 is compared with element in pos 0 and insert it in the proper position
- During second iteration element in pos 2 is compared with elements in pos 1 and 0 and insert it in the proper position
- During third iteration element in pos 3 is compared with elements in pos 2, 1 and 0. Then insert it in the proper position
- In general in every iteration an element is compared with all elements before it and insert it in the proper position

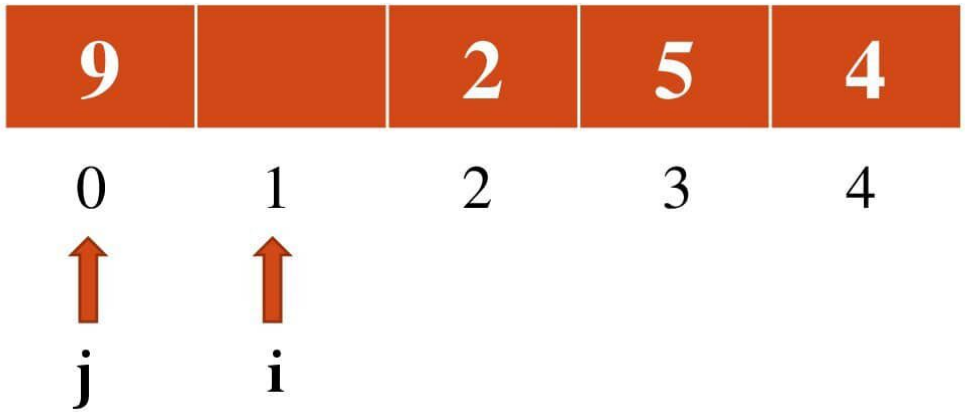
Insertion Sort



Insertion Sort

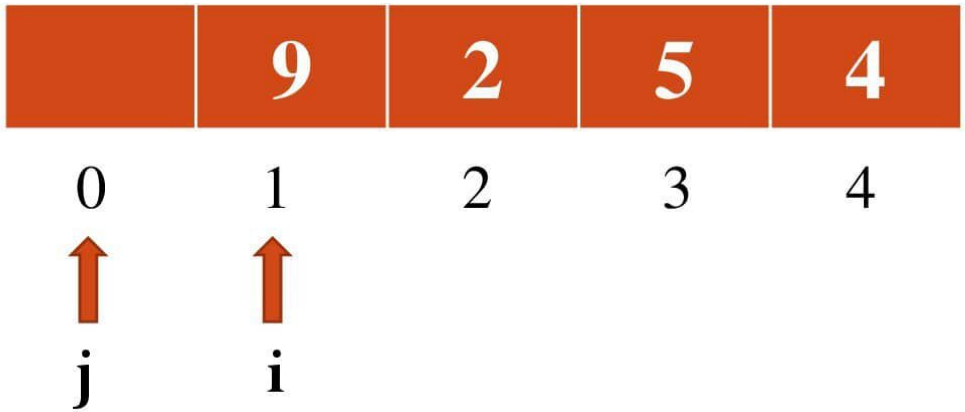


Insertion Sort



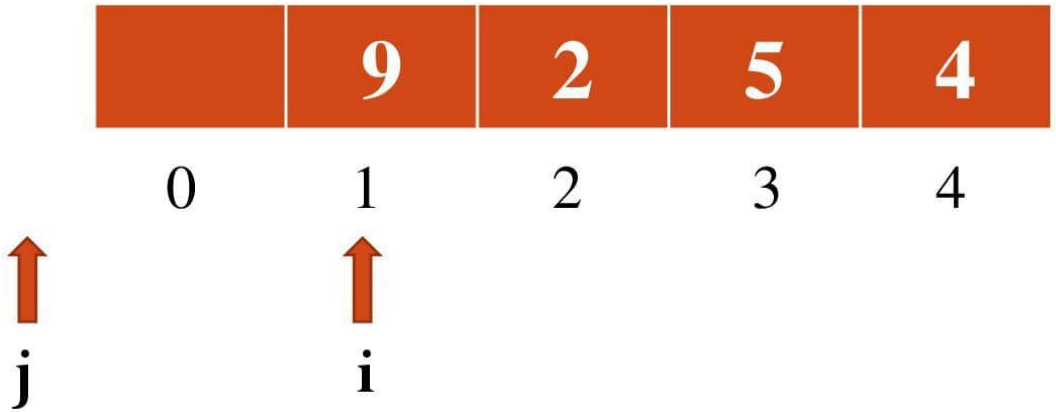
temp=3

Insertion Sort



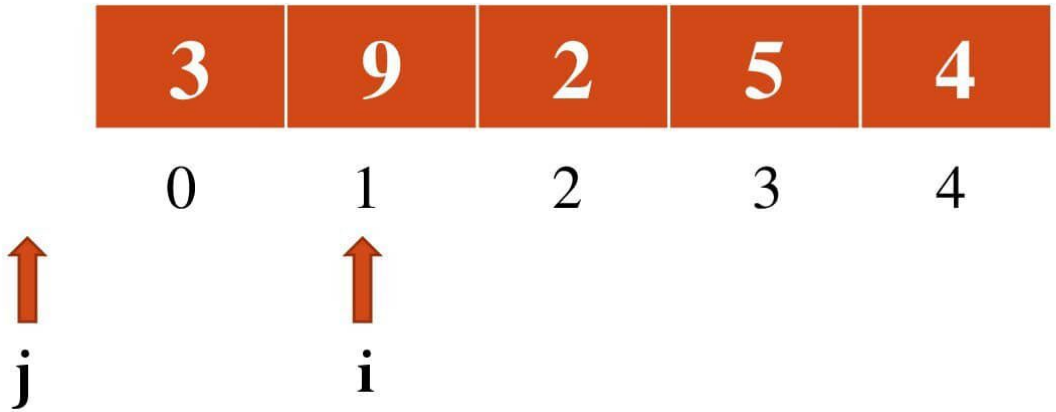
temp=3

Insertion Sort



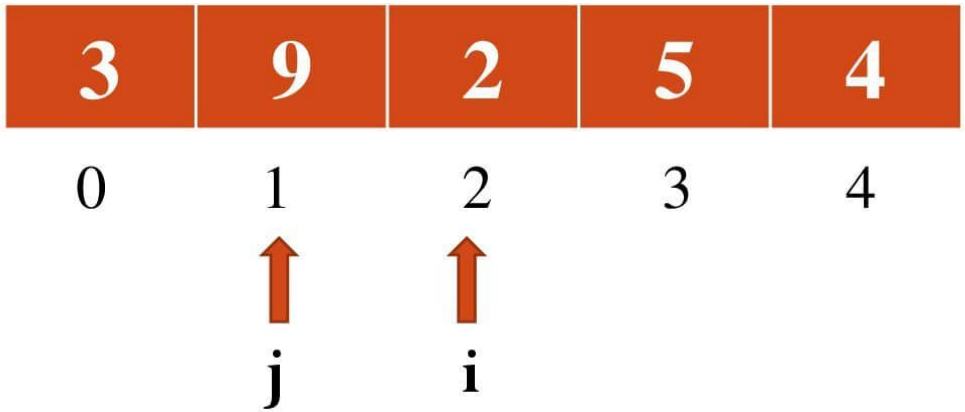
temp=3

Insertion Sort

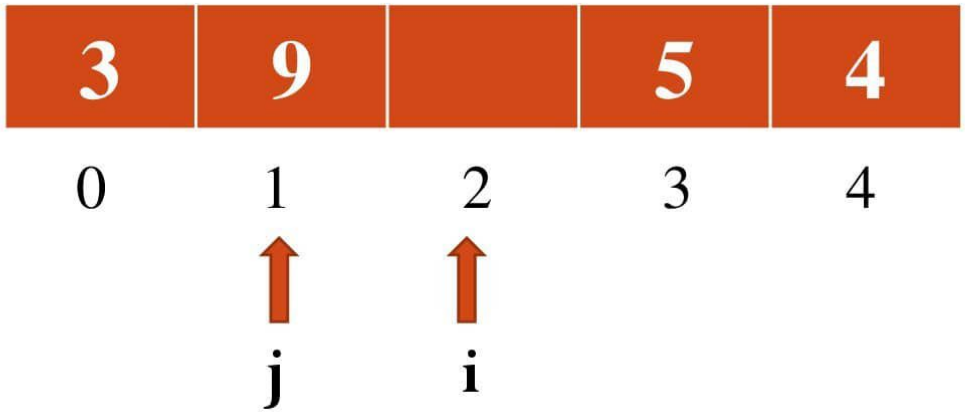


temp=3

Insertion Sort

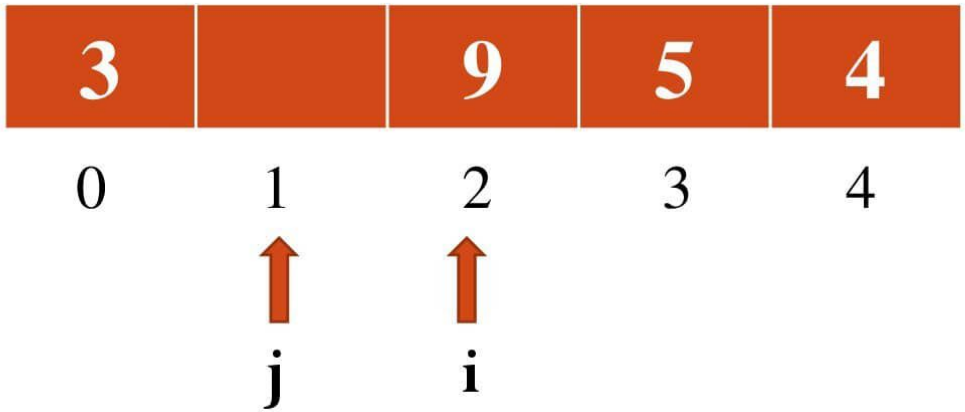


Insertion Sort



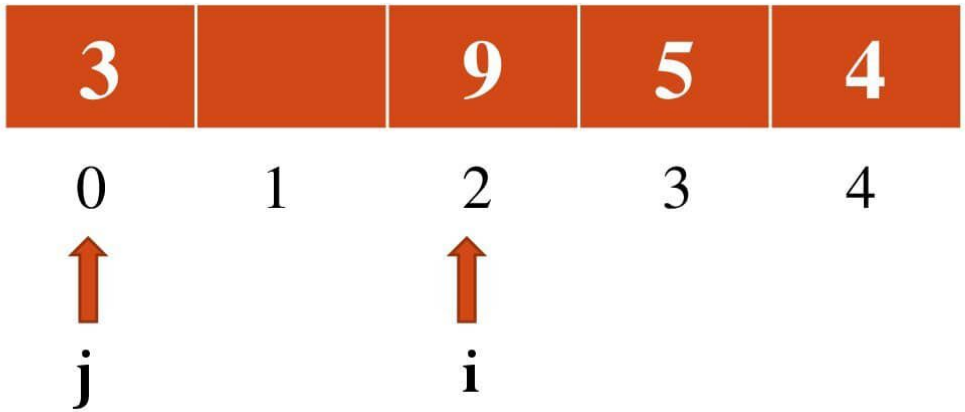
temp=2

Insertion Sort



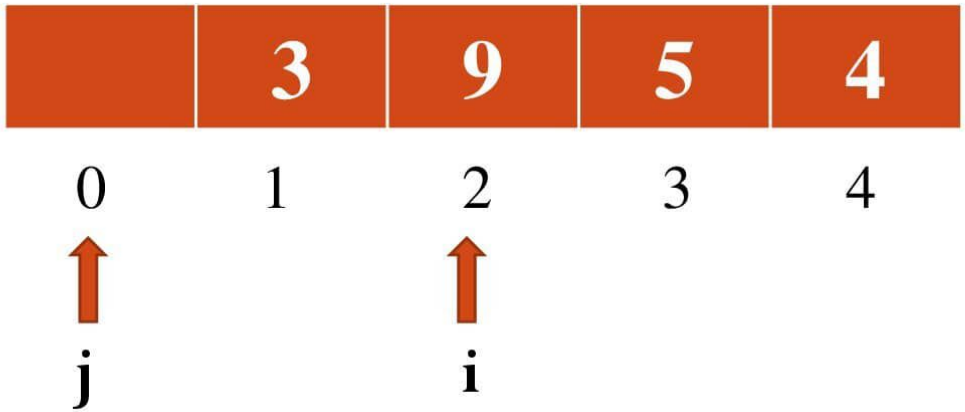
temp=2

Insertion Sort



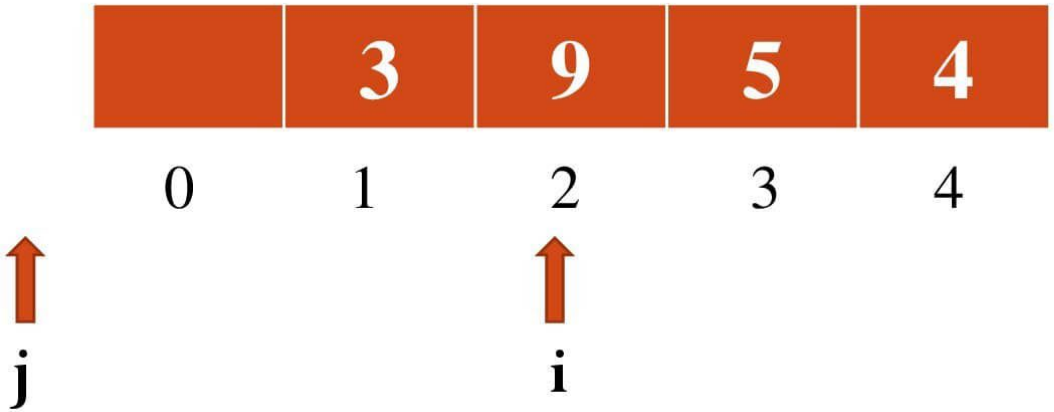
temp=2

Insertion Sort



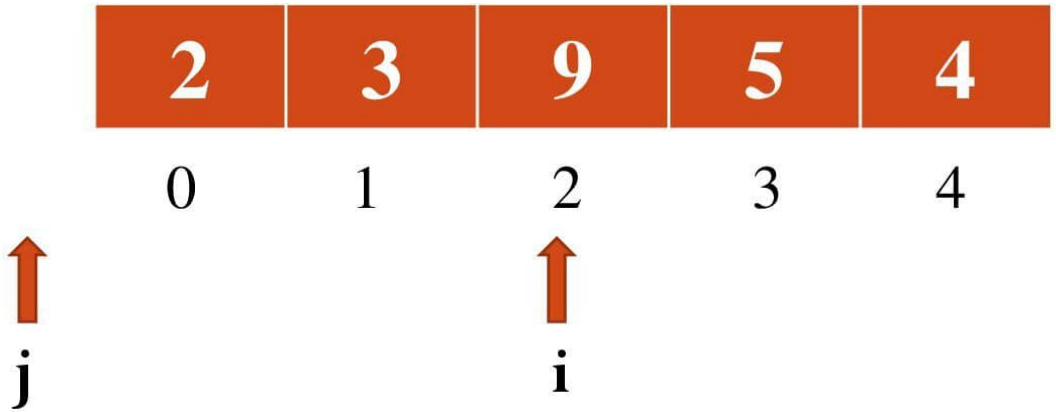
temp=2

Insertion Sort



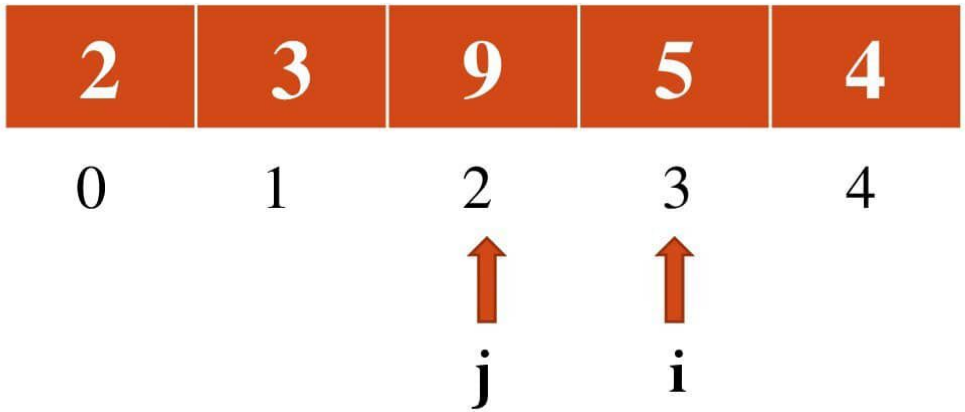
temp=2

Insertion Sort

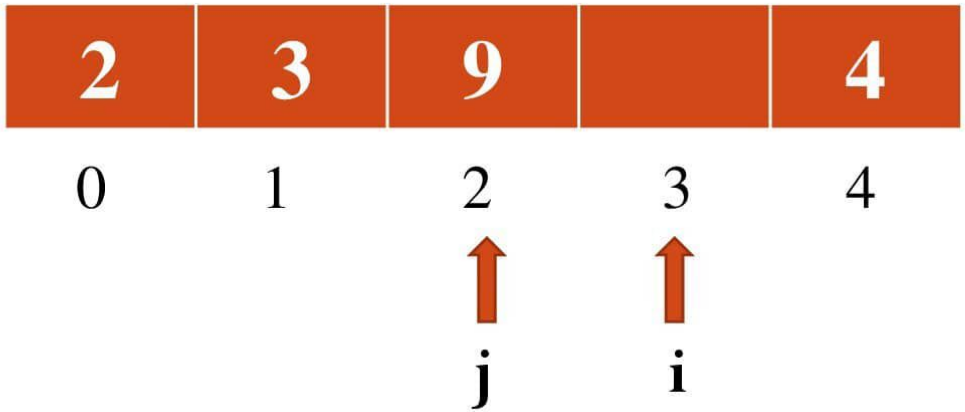


temp=2

Insertion Sort

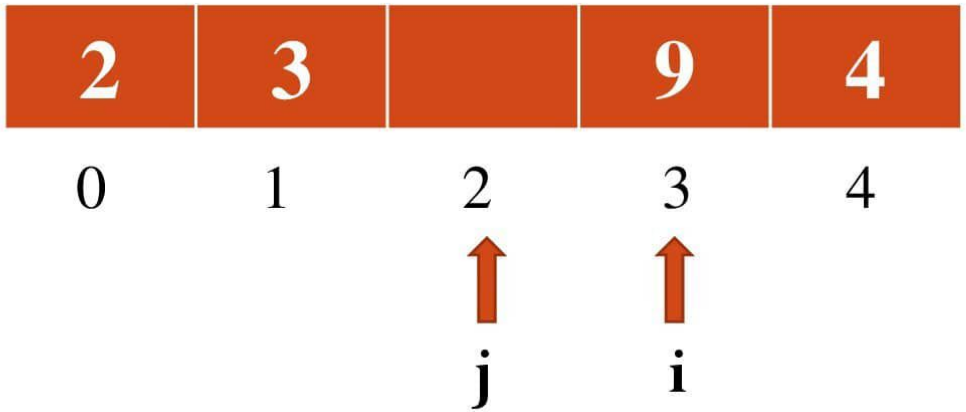


Insertion Sort



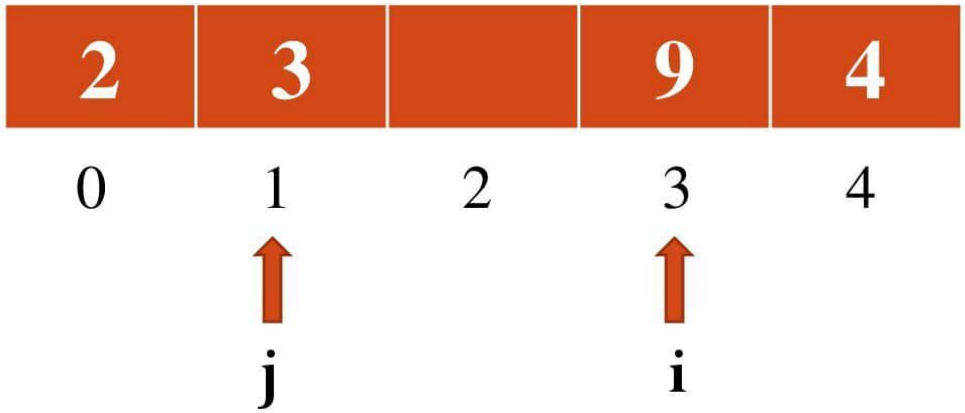
temp= 5

Insertion Sort



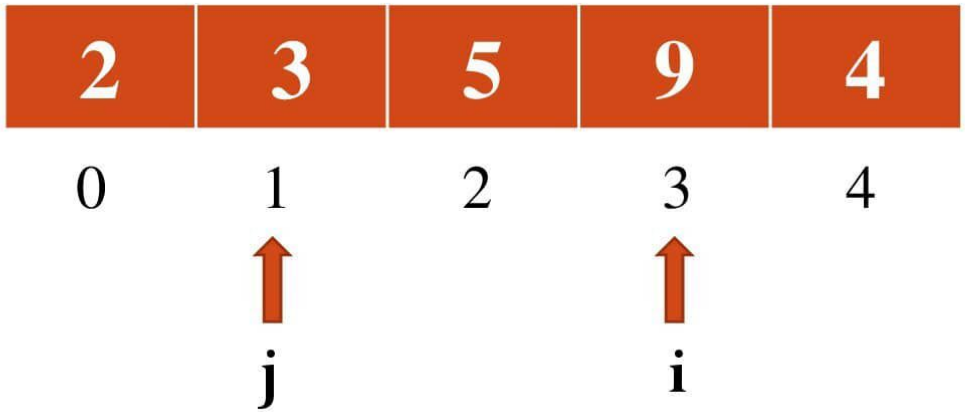
temp= 5

Insertion Sort



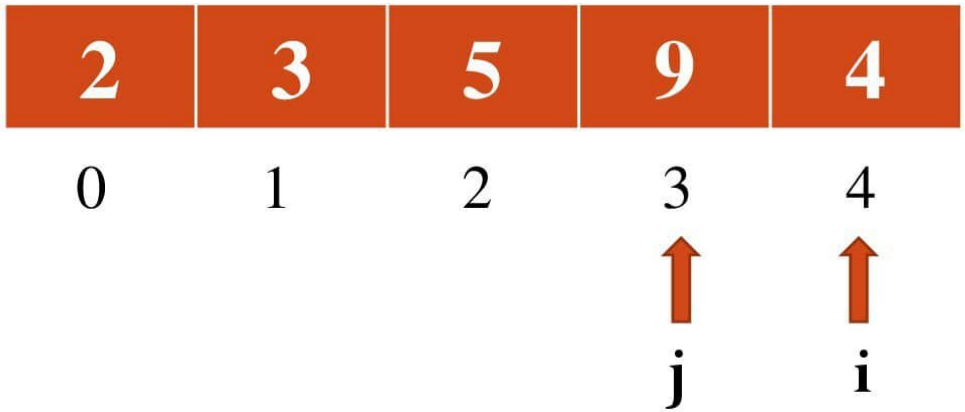
temp= 5

Insertion Sort

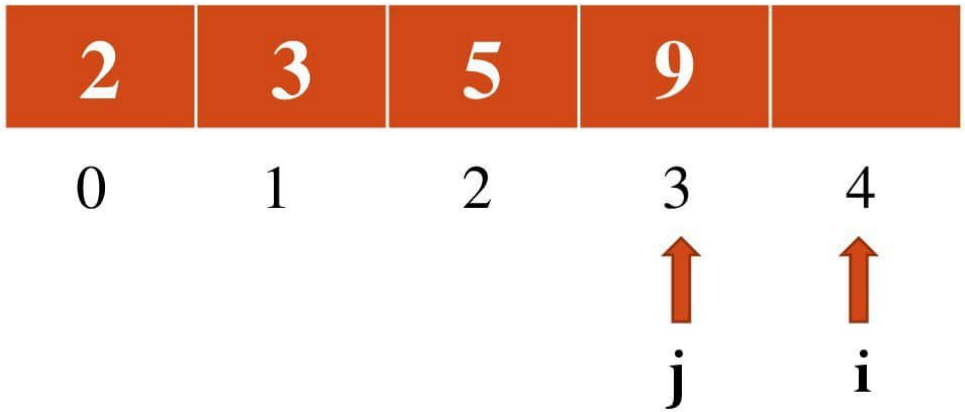


temp= 5

Insertion Sort

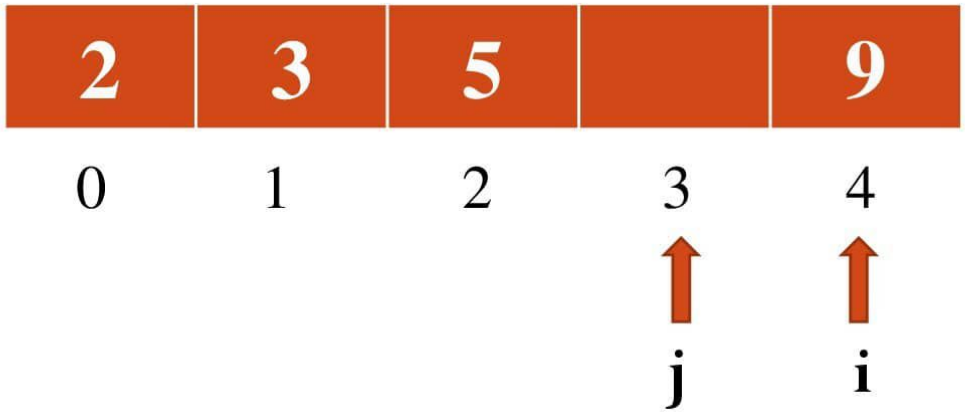


Insertion Sort



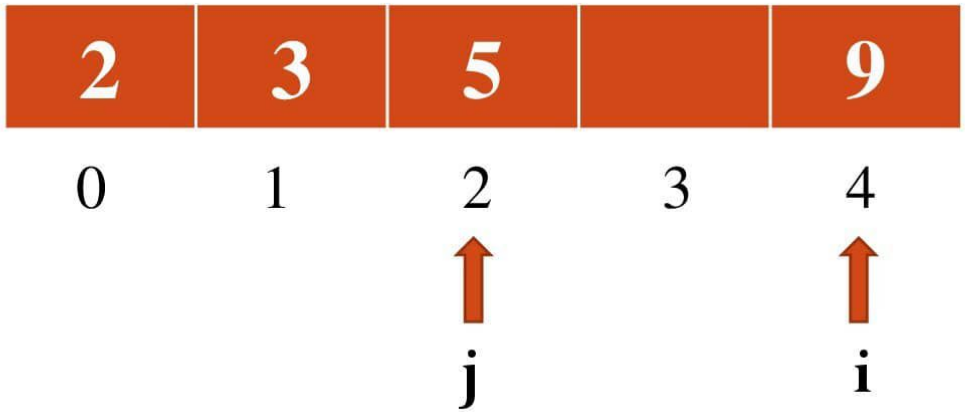
temp= 4

Insertion Sort



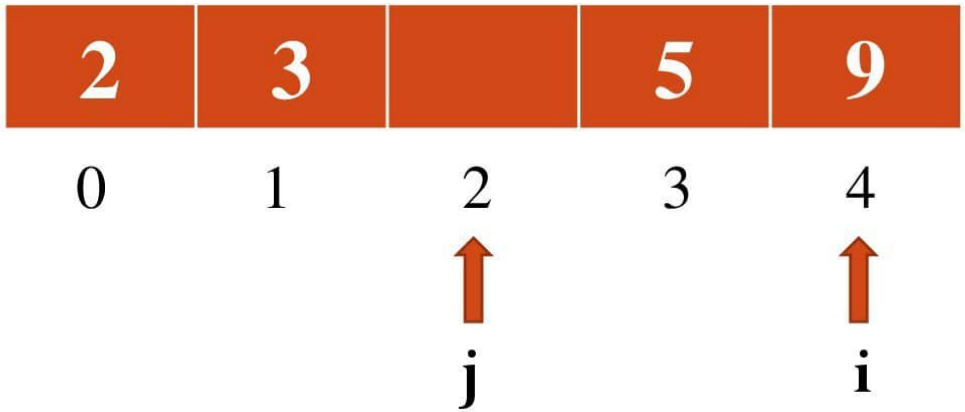
temp= 4

Insertion Sort



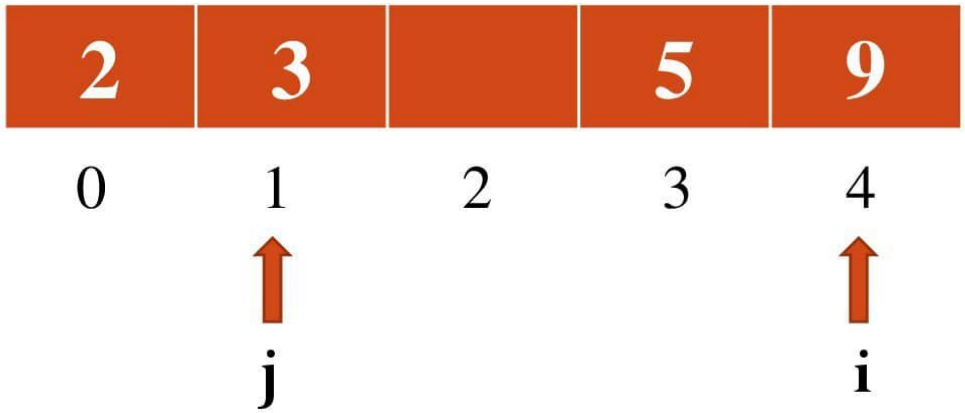
temp= 4

Insertion Sort



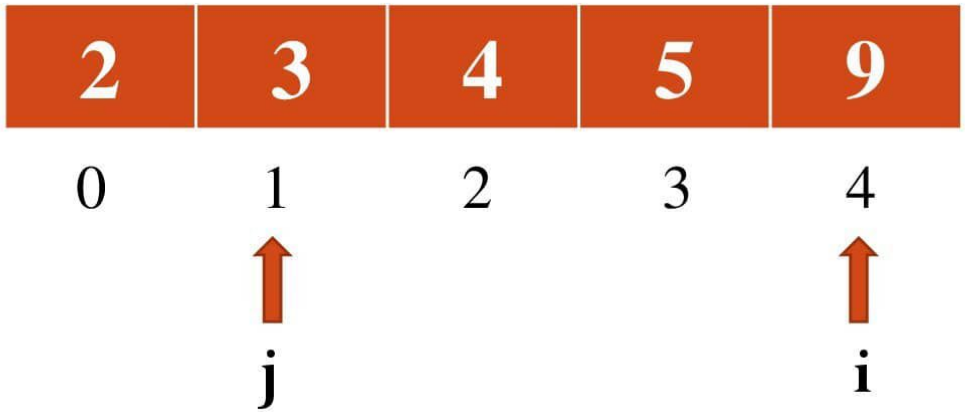
temp= 4

Insertion Sort



temp= 4

Insertion Sort



temp= 4

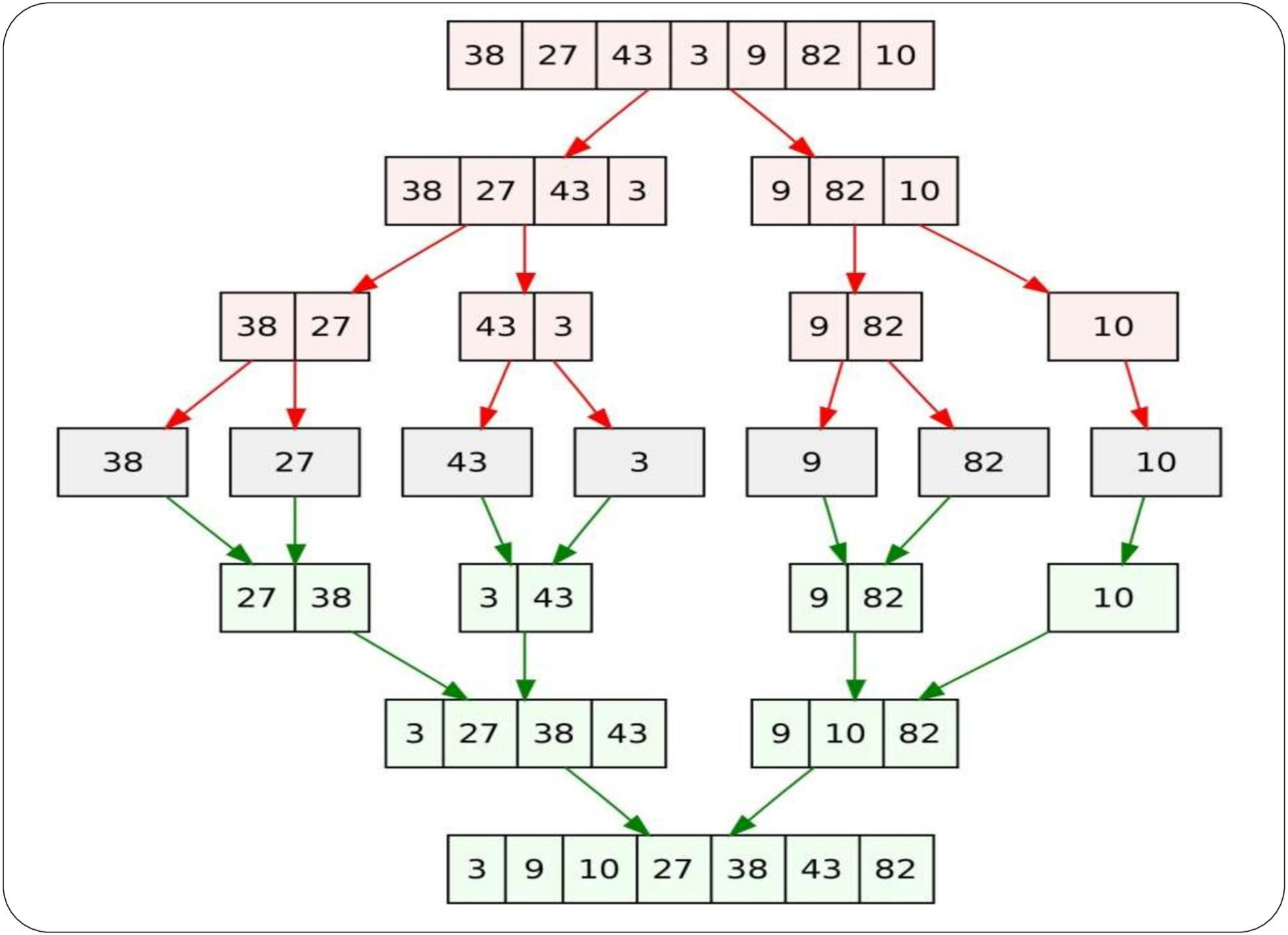
Insertion Sort

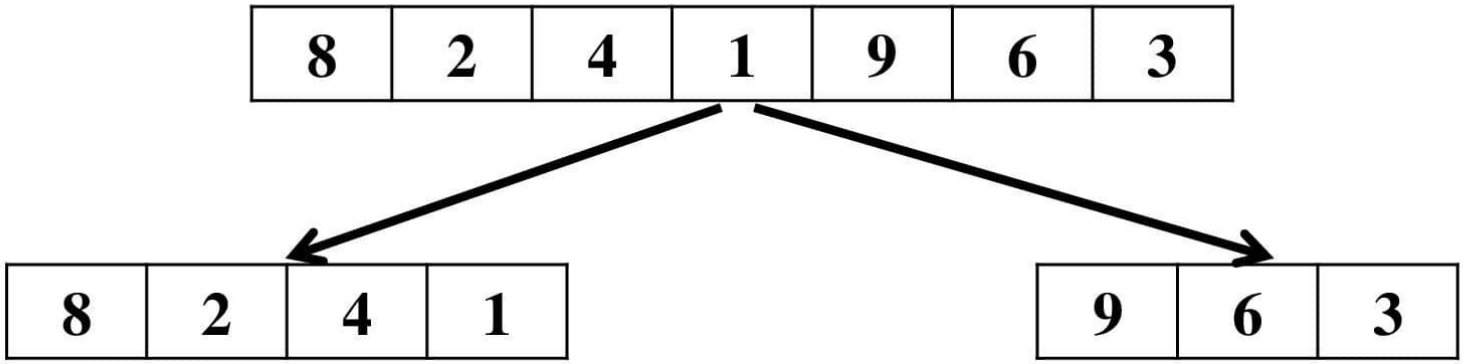
Algorithm InsertionSort(A, n)

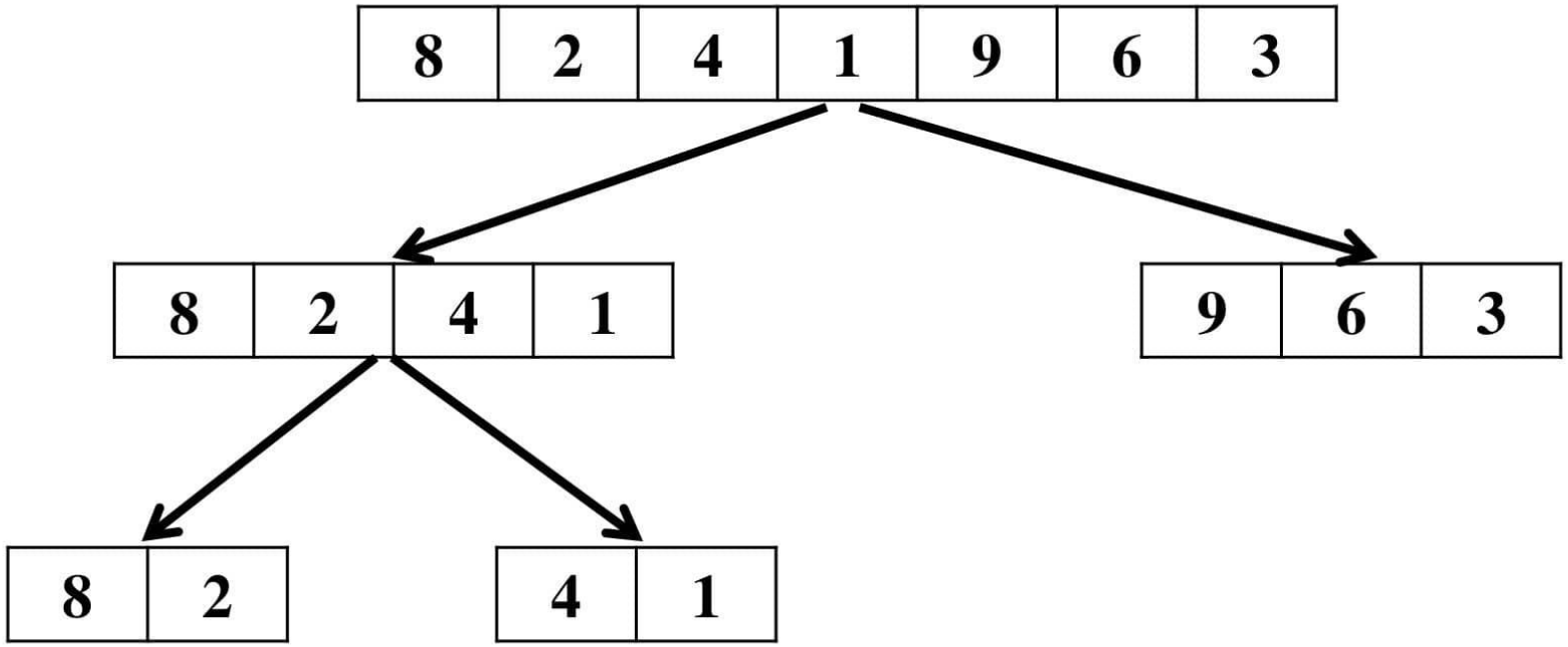
1. for i=1 to n-1 do
    1. temp=A[i]
    2. j=i-1
    3. While j >= 0 and A[j] > temp do
        1. A[j+1] = A[j]
        2. j=j-1
    4. A[j+1] = temp

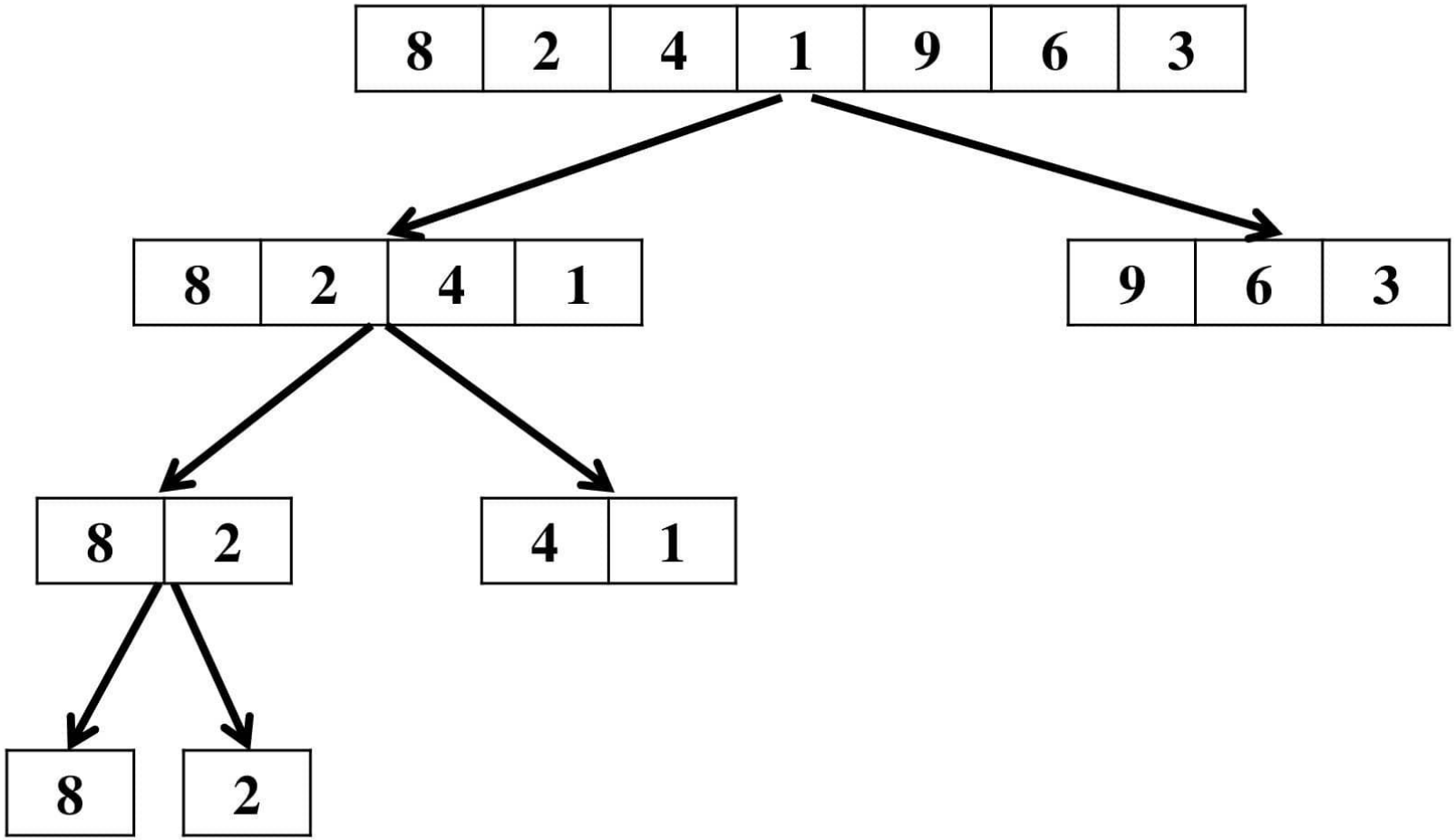
Merge Sort

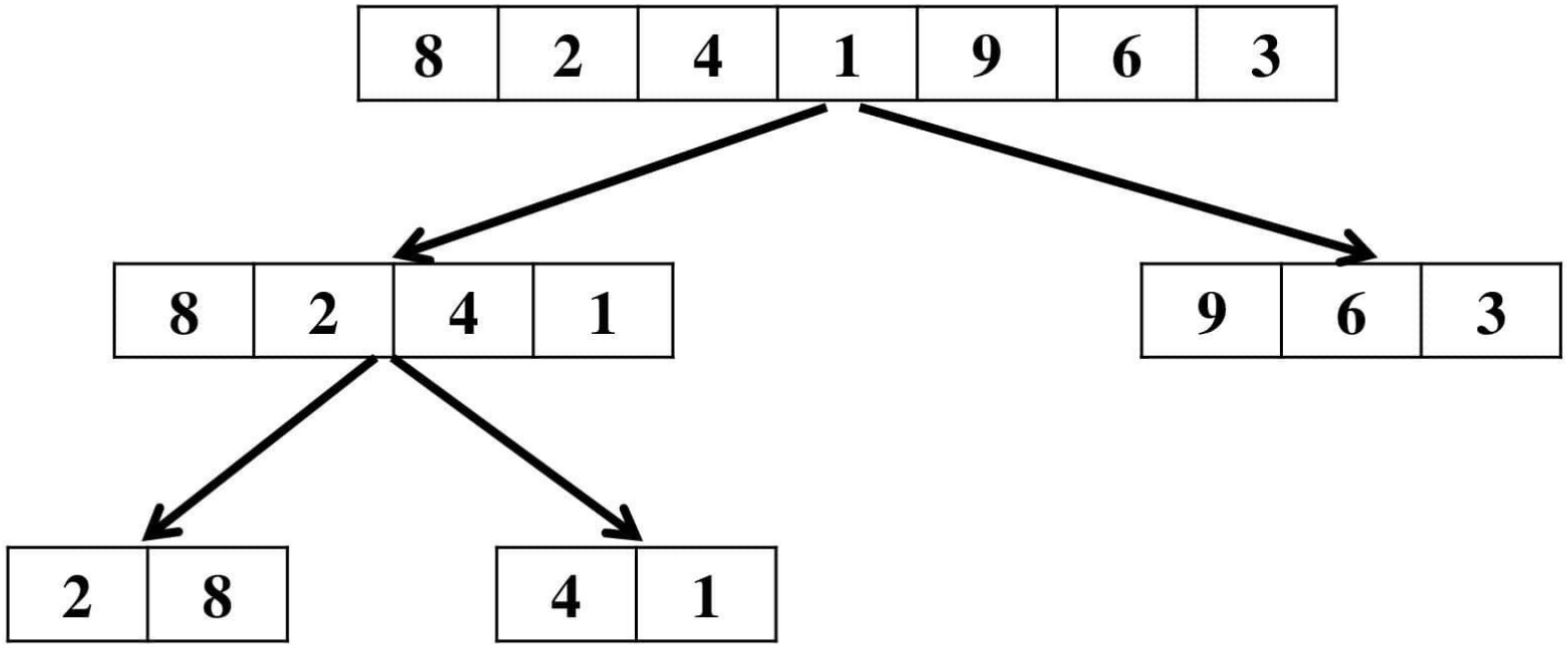
- Given a sequence of n elements a[1],……a[n]. Split this array into two sets a[l],,.a[n/2] and a[(n/2)+1],…a[n]. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n element

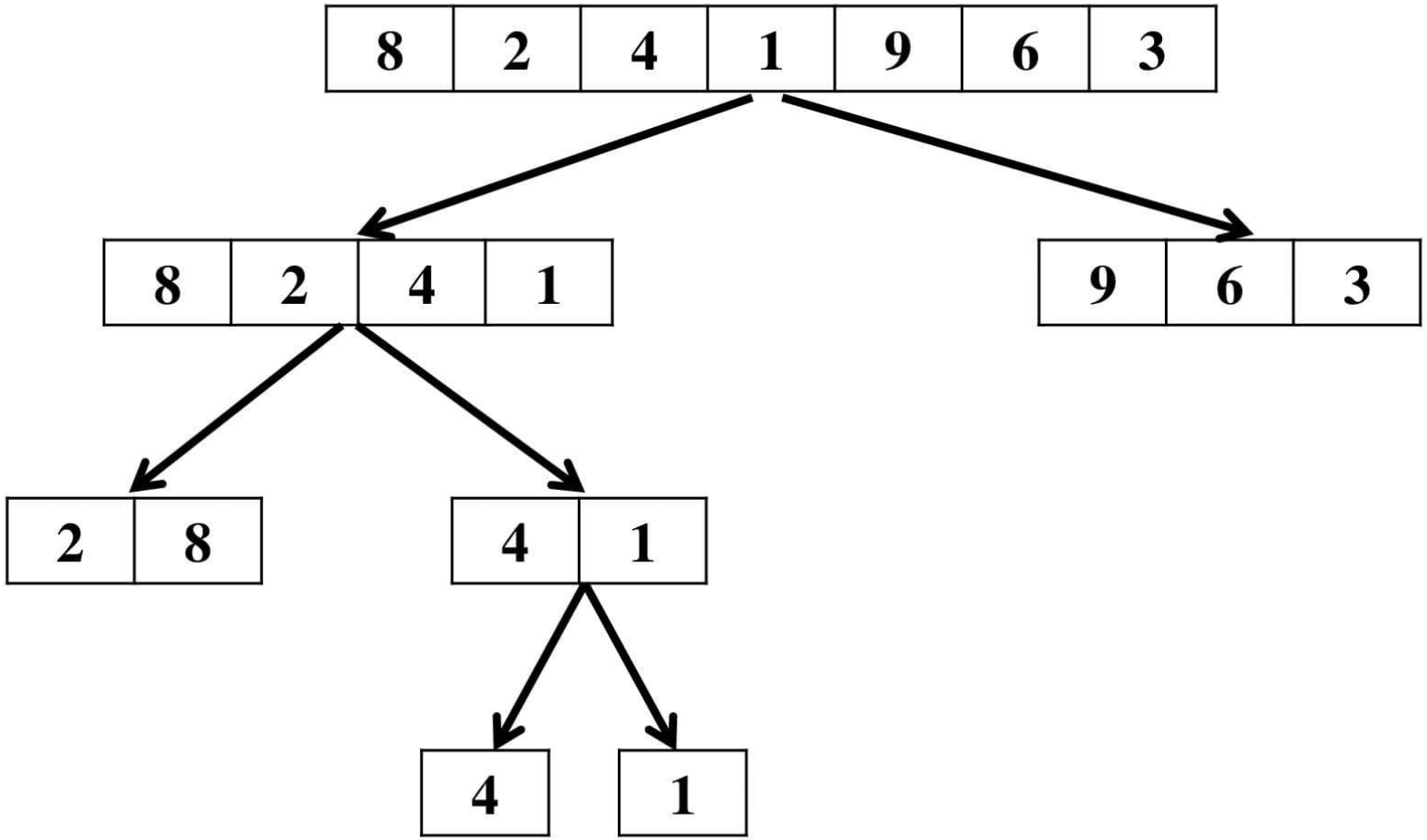

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

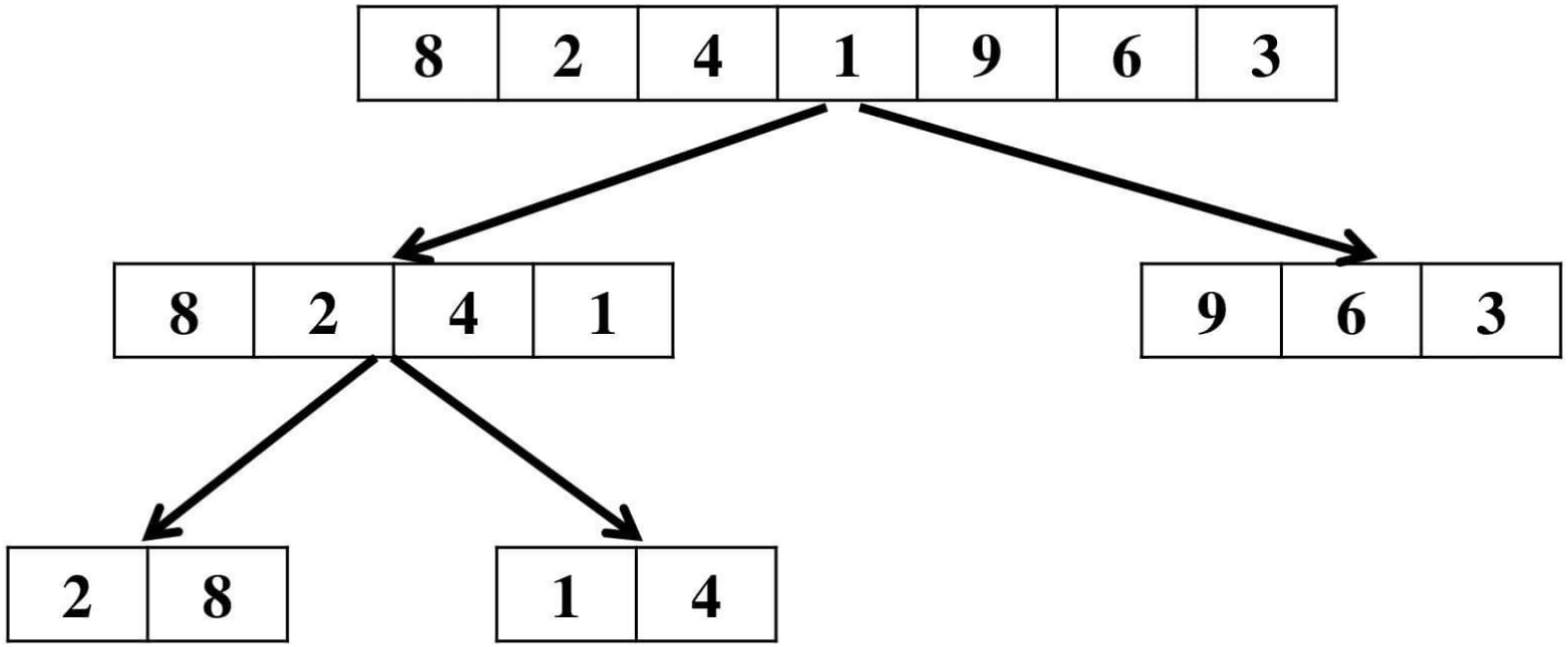

8 2 4 1 9 6 3
8 2 4 1
9 6 3

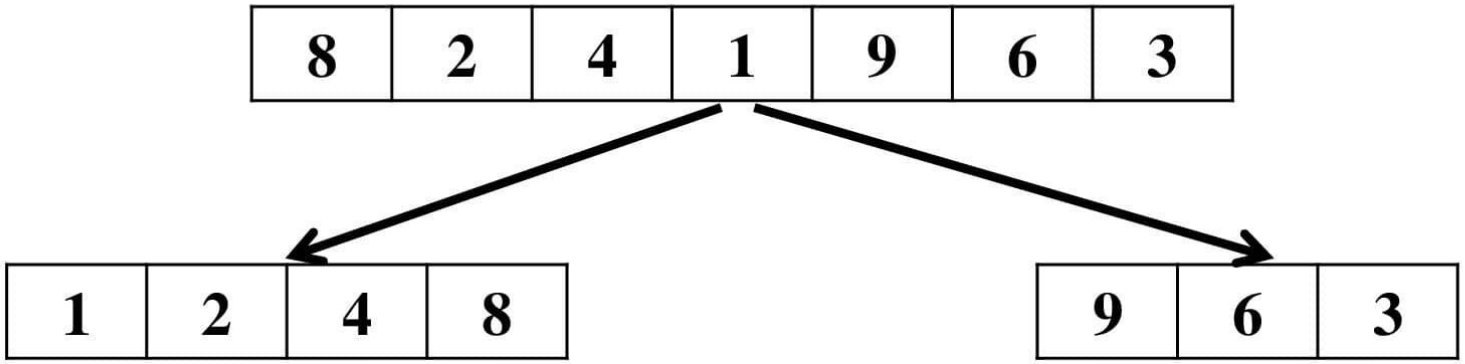

8 2 4 1 9 6 3
8 2 4 1
9 6 3
8 2
4 1

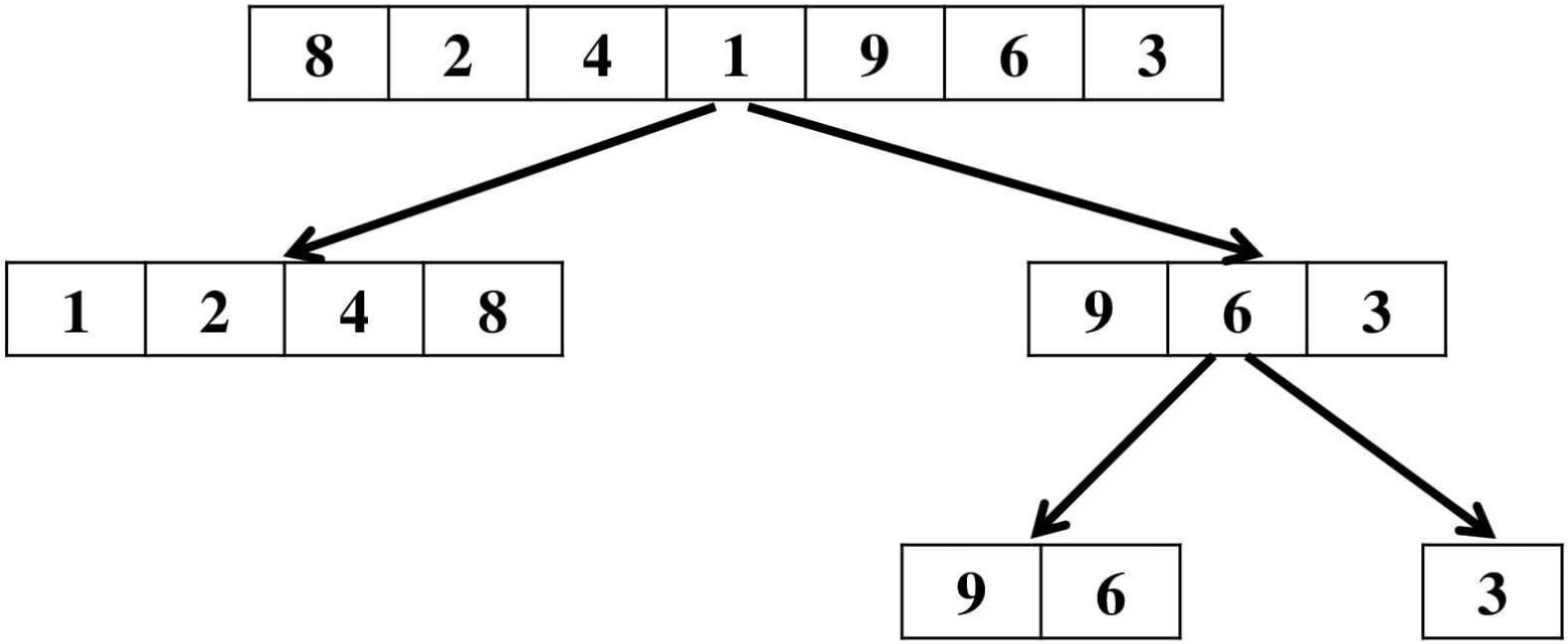

8 2 4 1 9 6 3
8 2 4 1
9 6 3
8 2
4 1
8
2

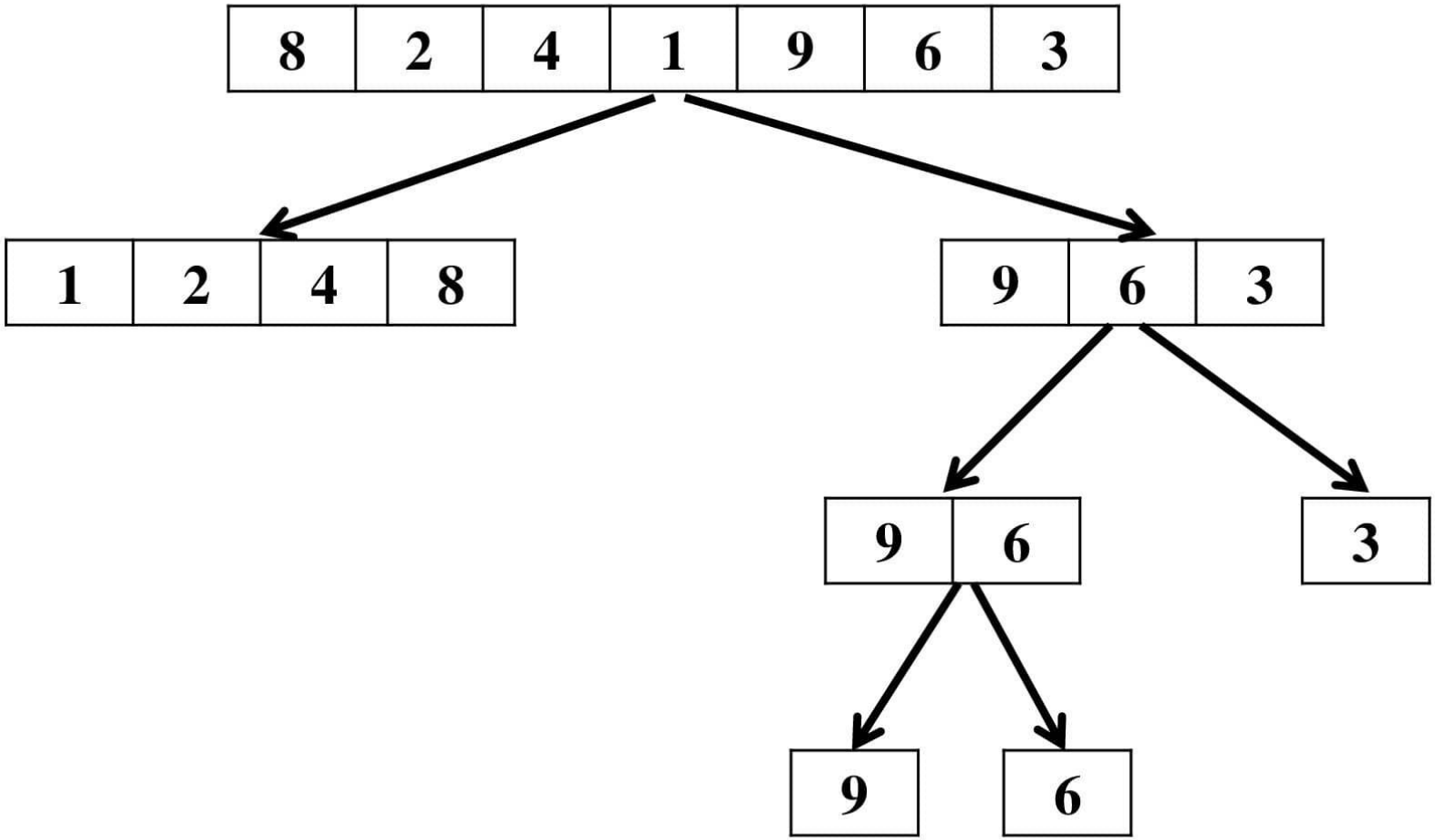

8 2 4 1 9 6 3
8 2 4 1
9 6 3
2 8
4 1

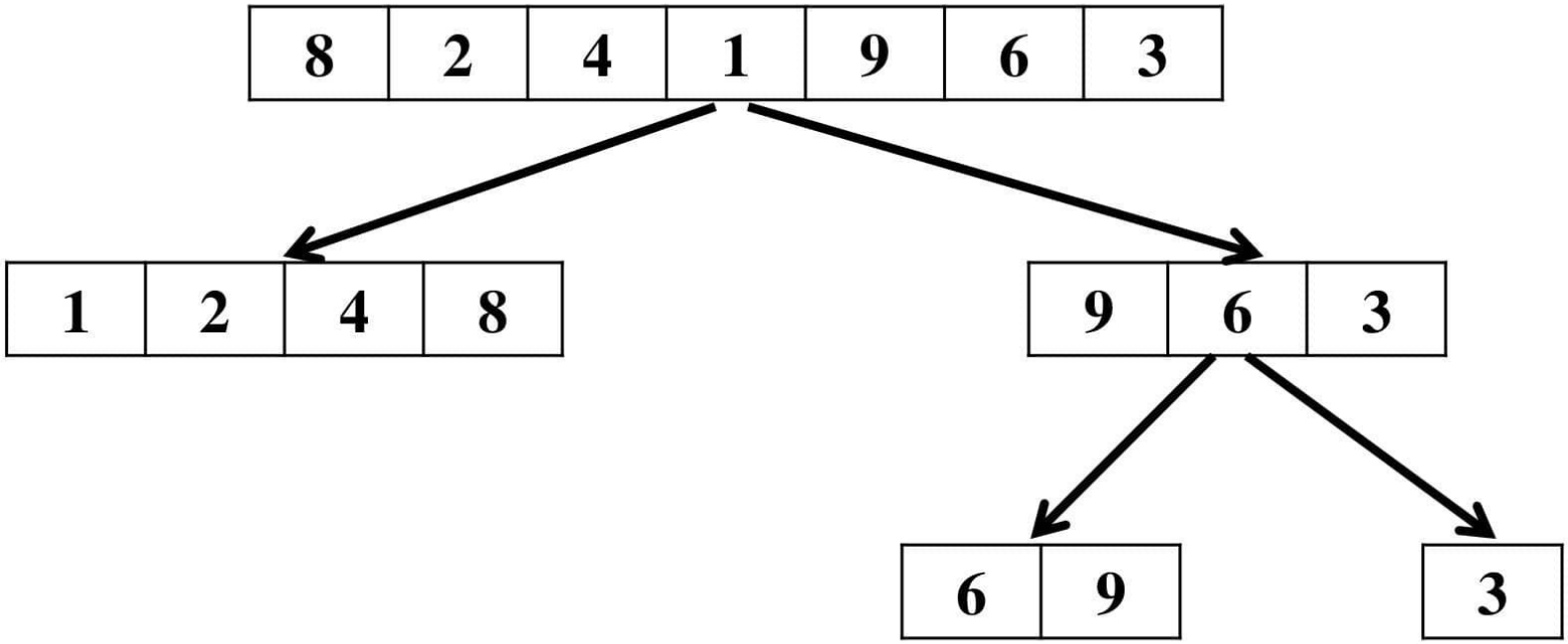

8 2 4 1 9 6 3
8 2 4 1
9 6 3
2 8
4 1
4
1

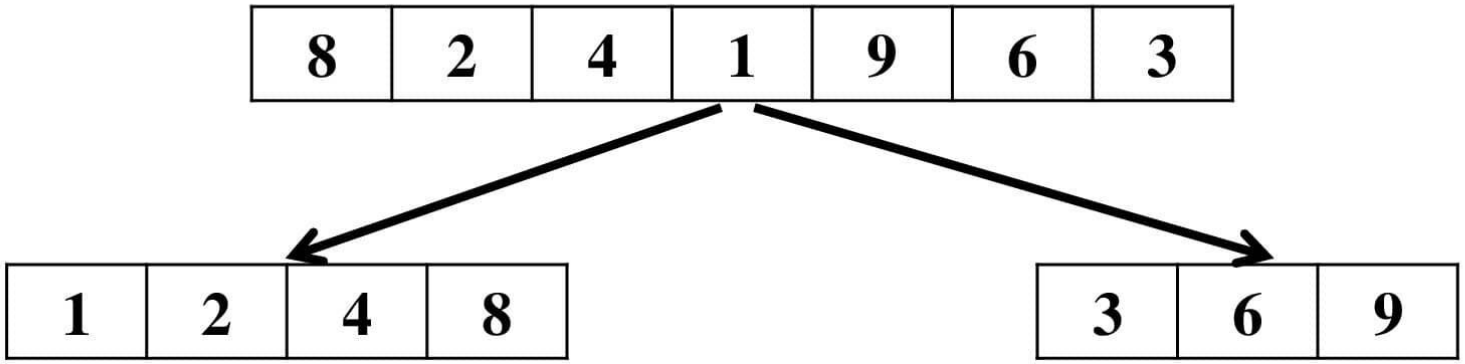

8 2 4 1 9 6 3
8 2 4 1
9 6 3
2 8
1 4


8 2 4 1 9 6 3
1 2 4 8
9 6 3


8 2 4 1 9 6 3
1 2 4 8
9 6 3
9 6
3


8 2 4 1 9 6 3
1 2 4 8
9 6 3
9 6
3
9
6


8 2 4 1 9 6 3
1 2 4 8
9 6 3
6 9
3


8 2 4 1 9 6 3
1 2 4 8
3 6 9

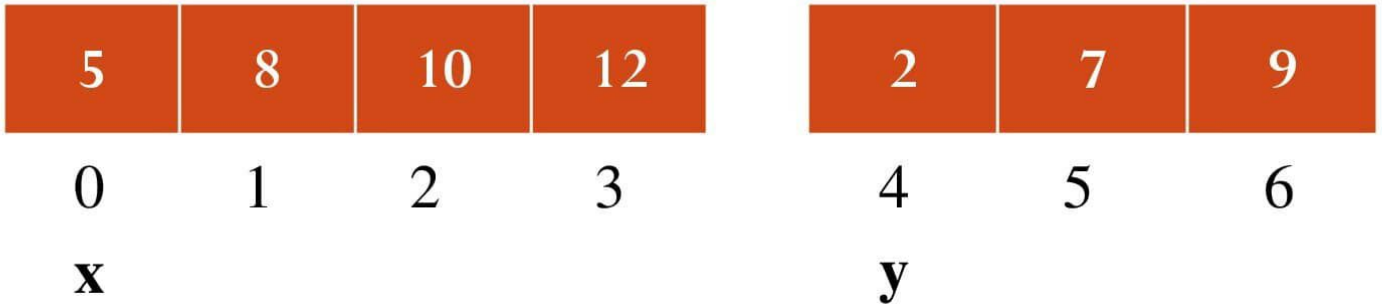
1	2	3	4	6	8	9

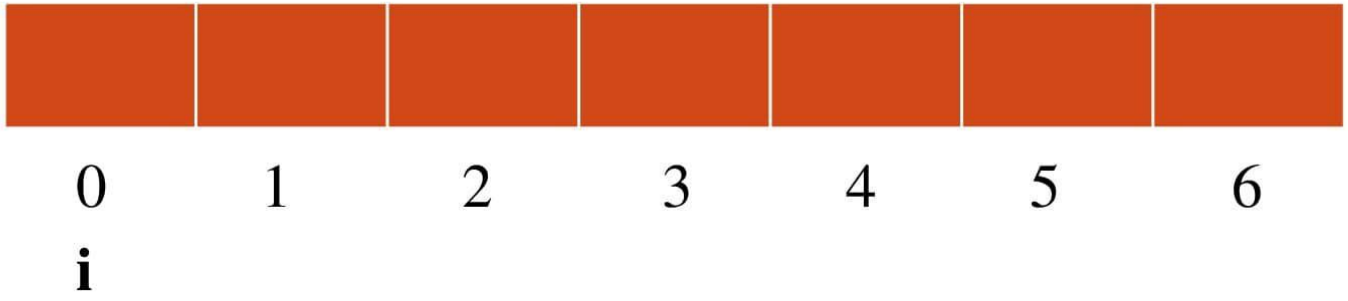
Merge Sort

```
Algorithm MergeSort(low, high)
{
        mid = (low + high )/2;
        MergeSort(low, mid);
        MergeSort(mid+1, high);
        Merge(low, mid, high);
}
```

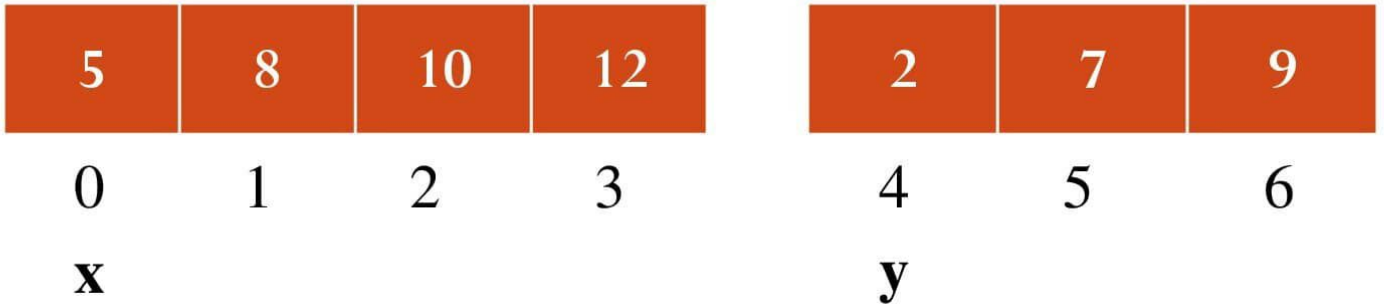
Merge

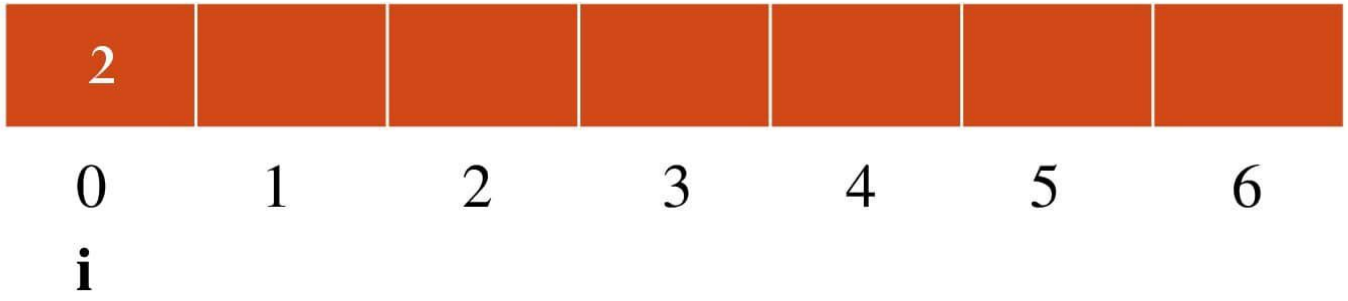
Merge



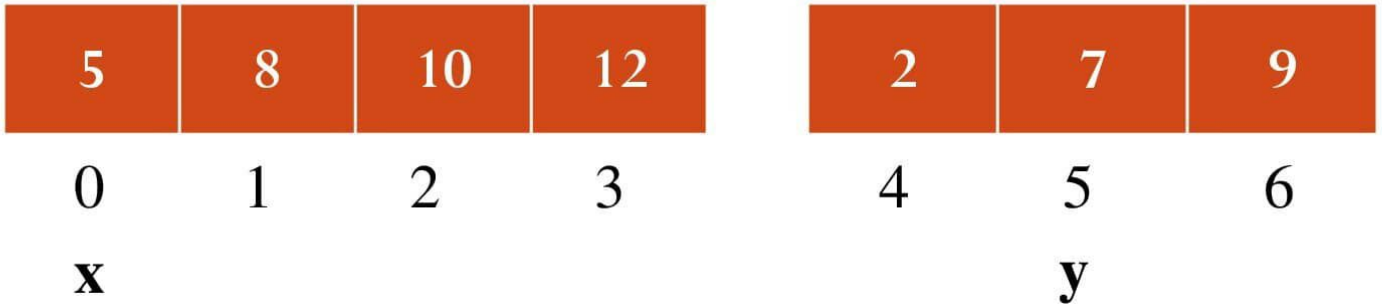


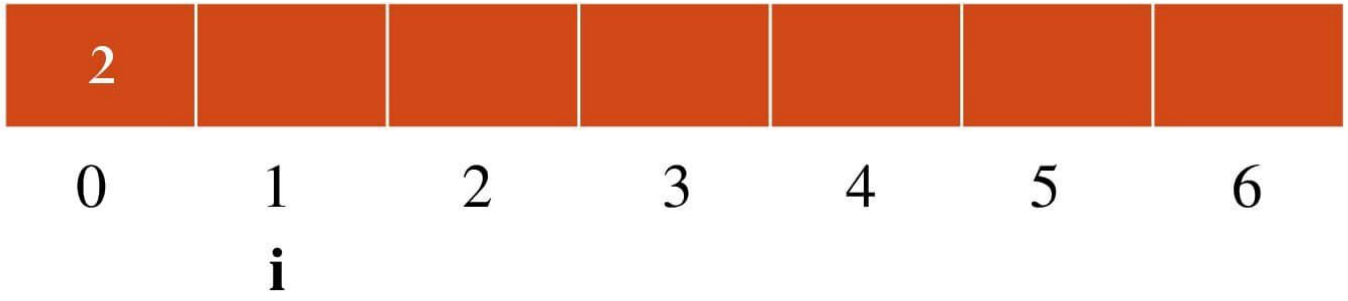
Merge



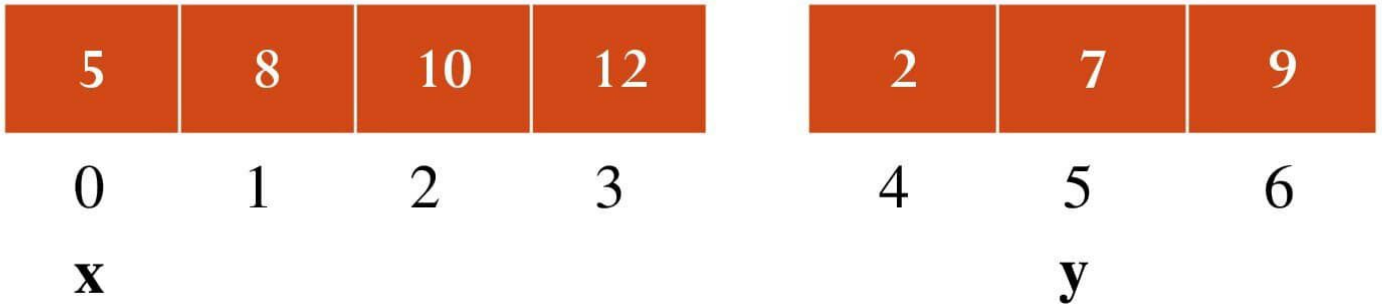


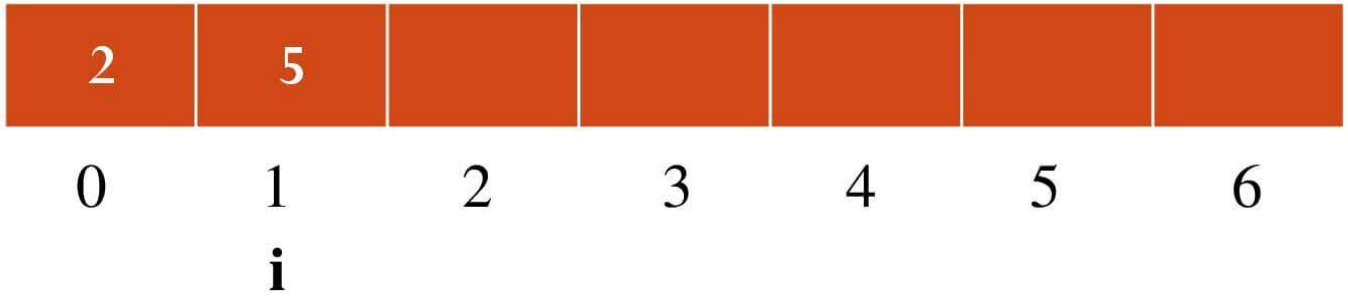
Merge



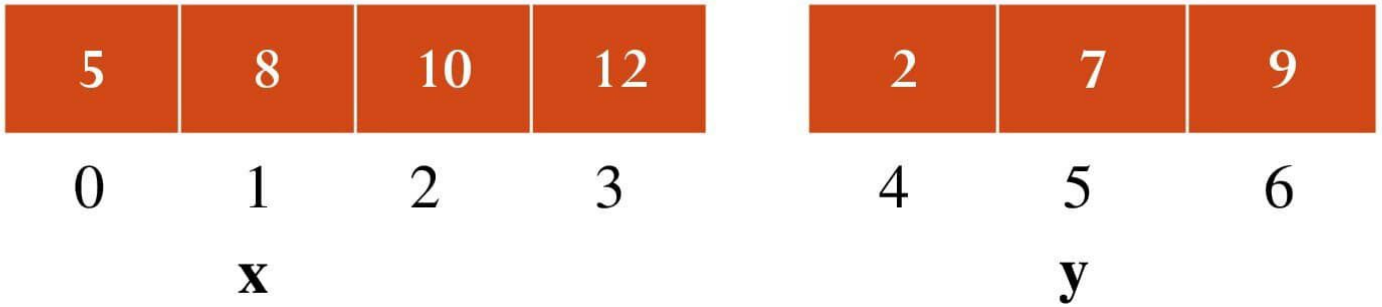


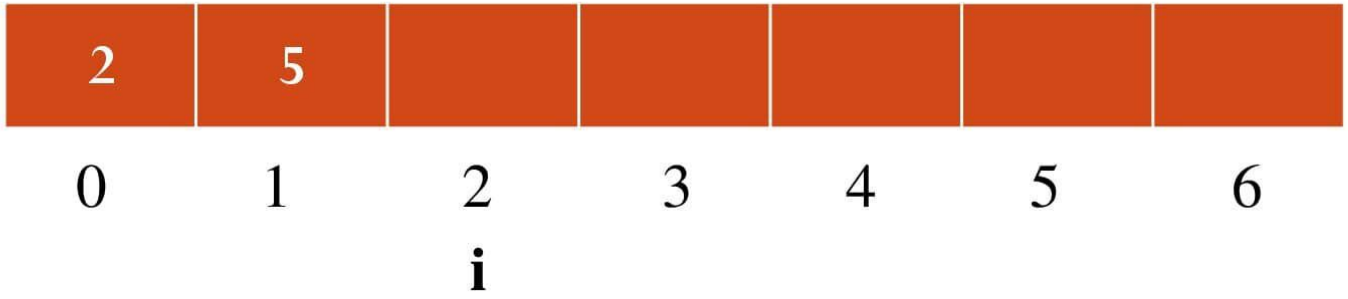
Merge



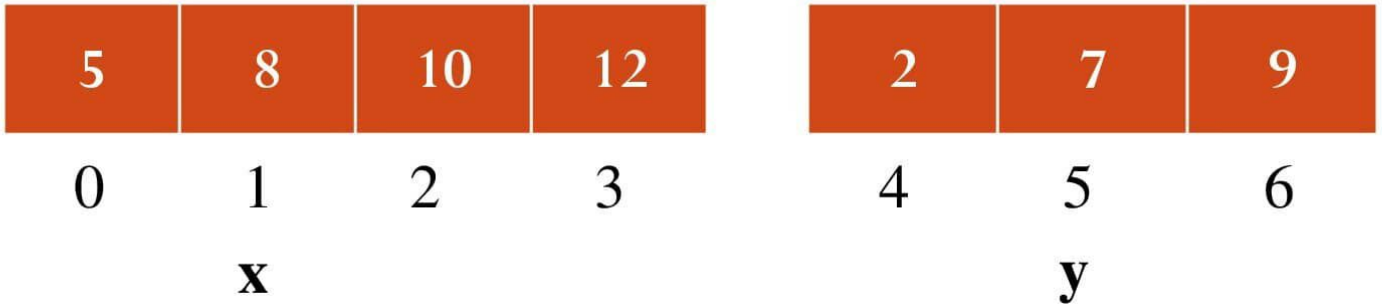


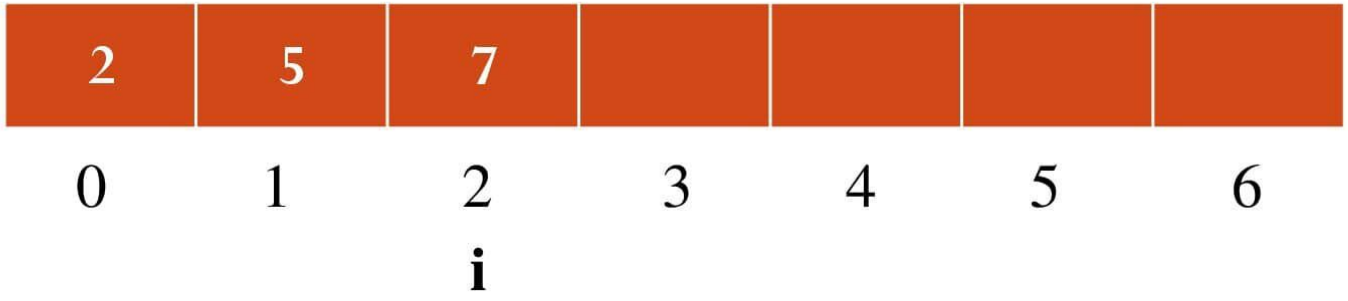
Merge



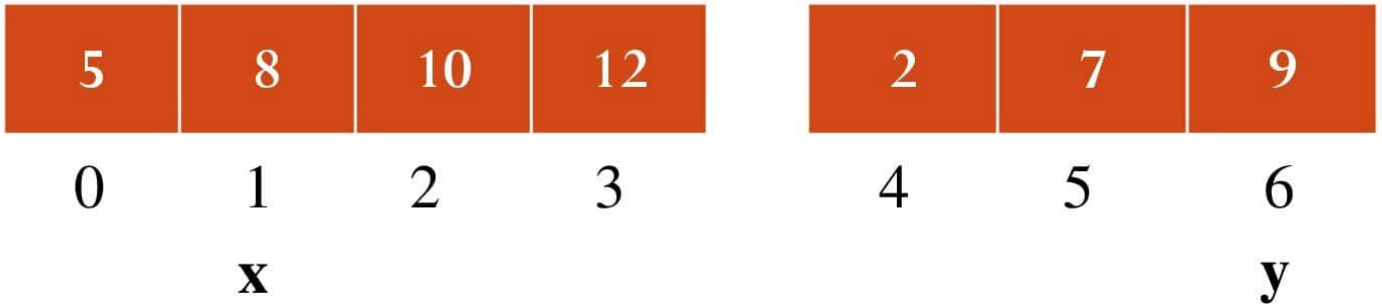


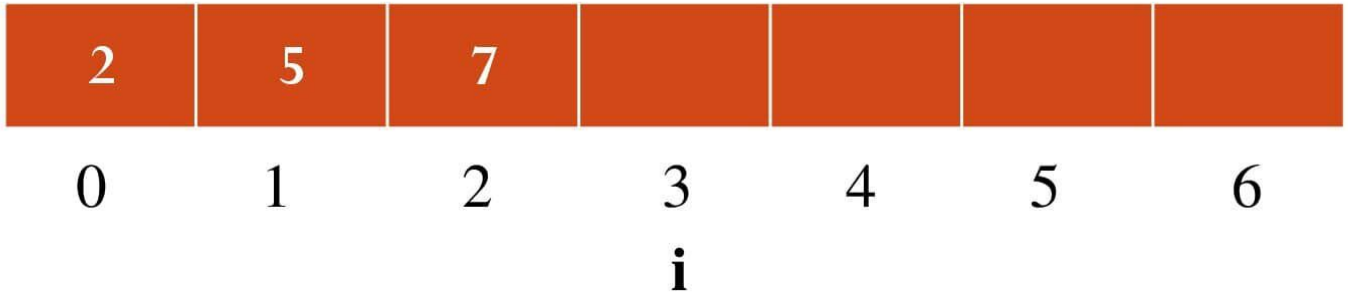
Merge



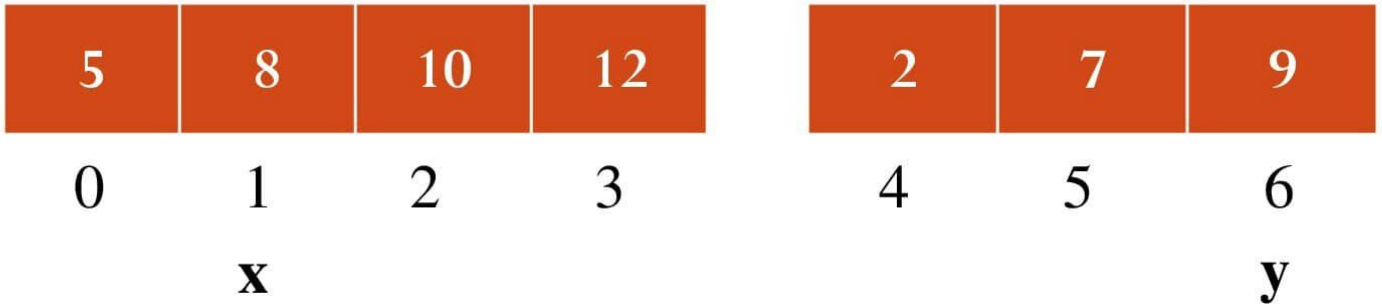


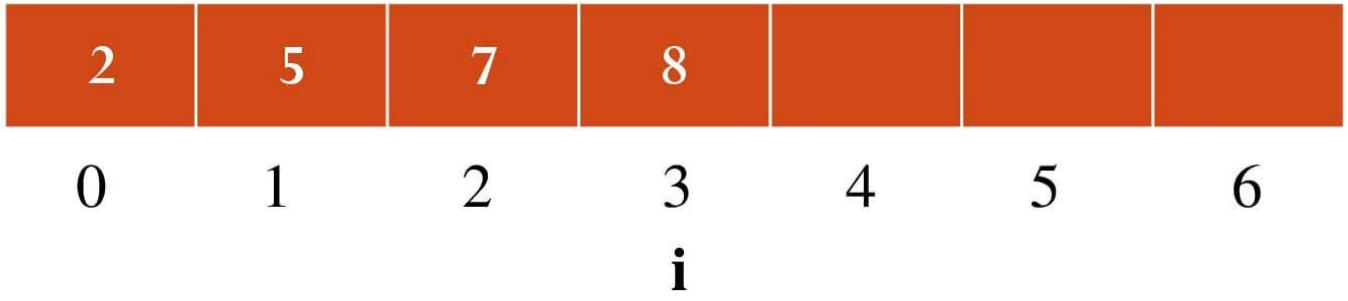
Merge



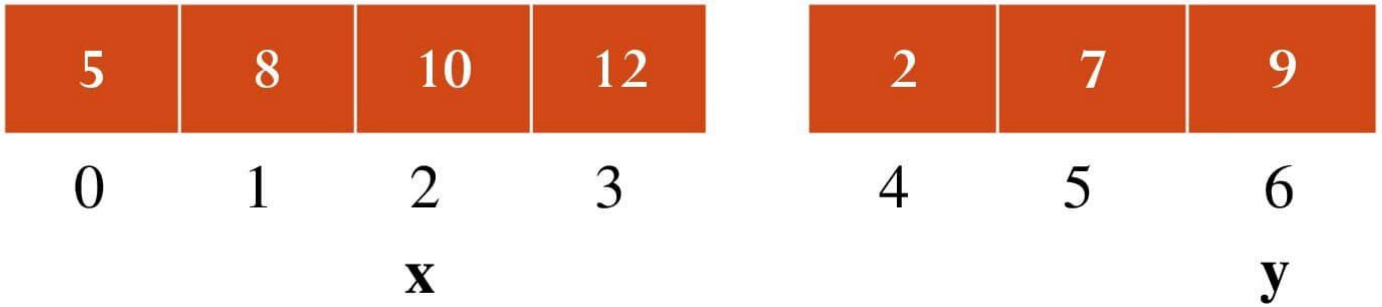


Merge

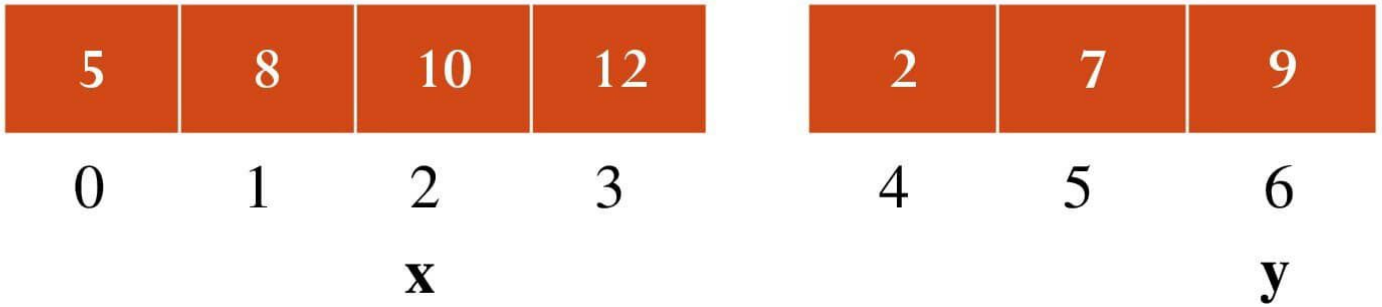




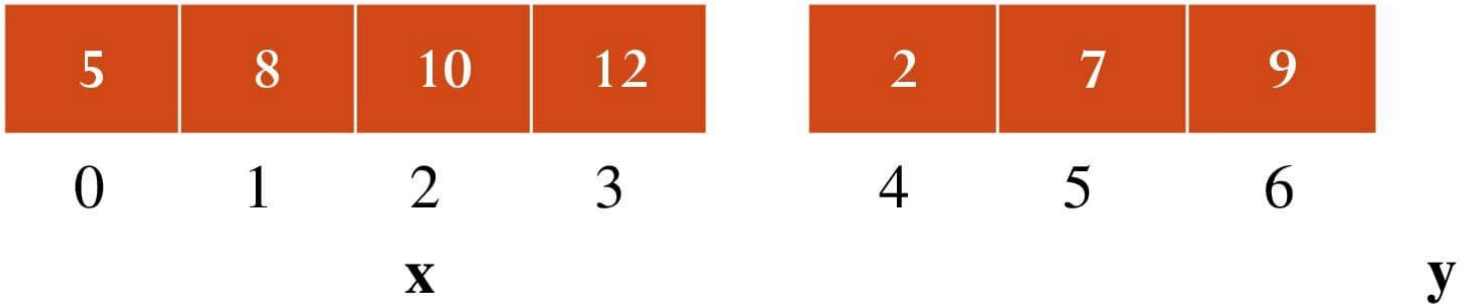
Merge



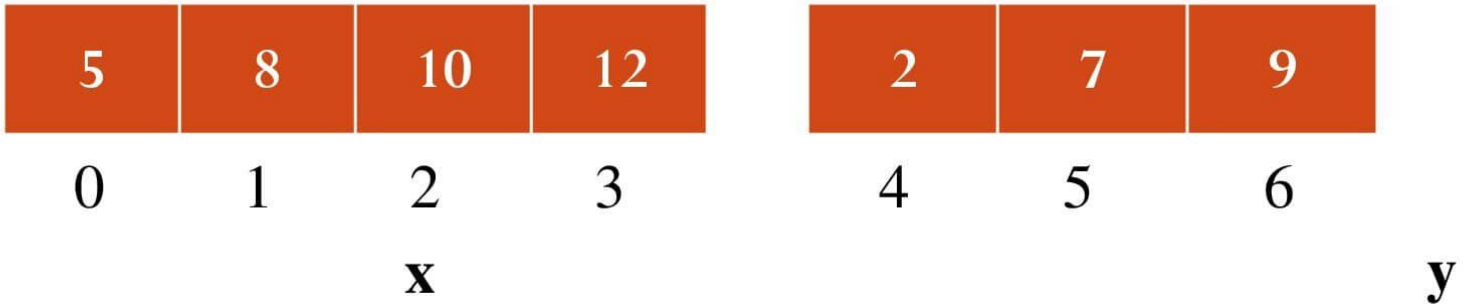
Merge



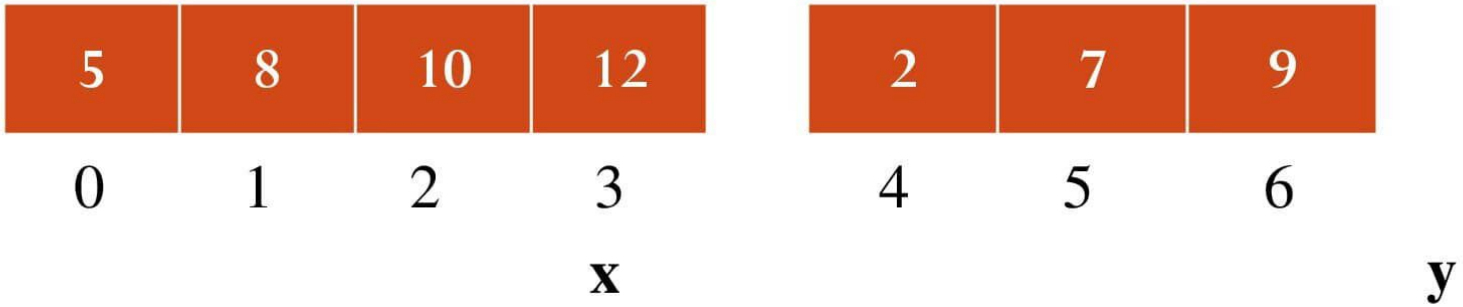
Merge



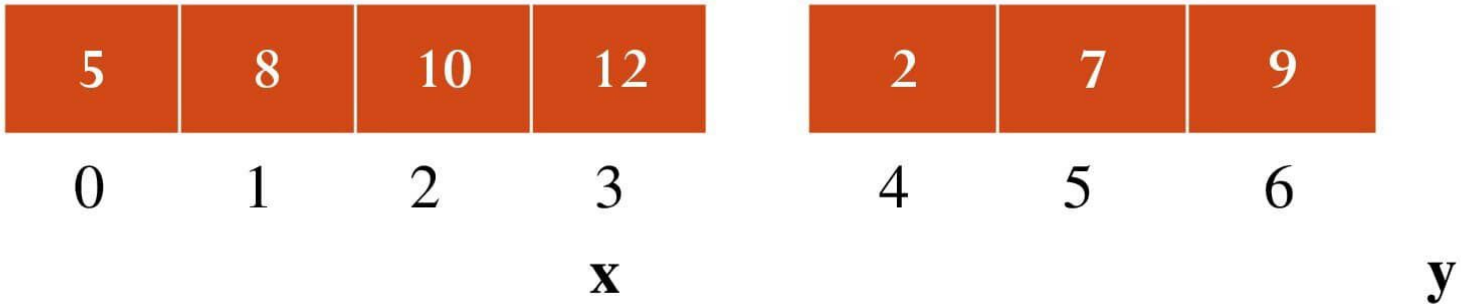
Merge



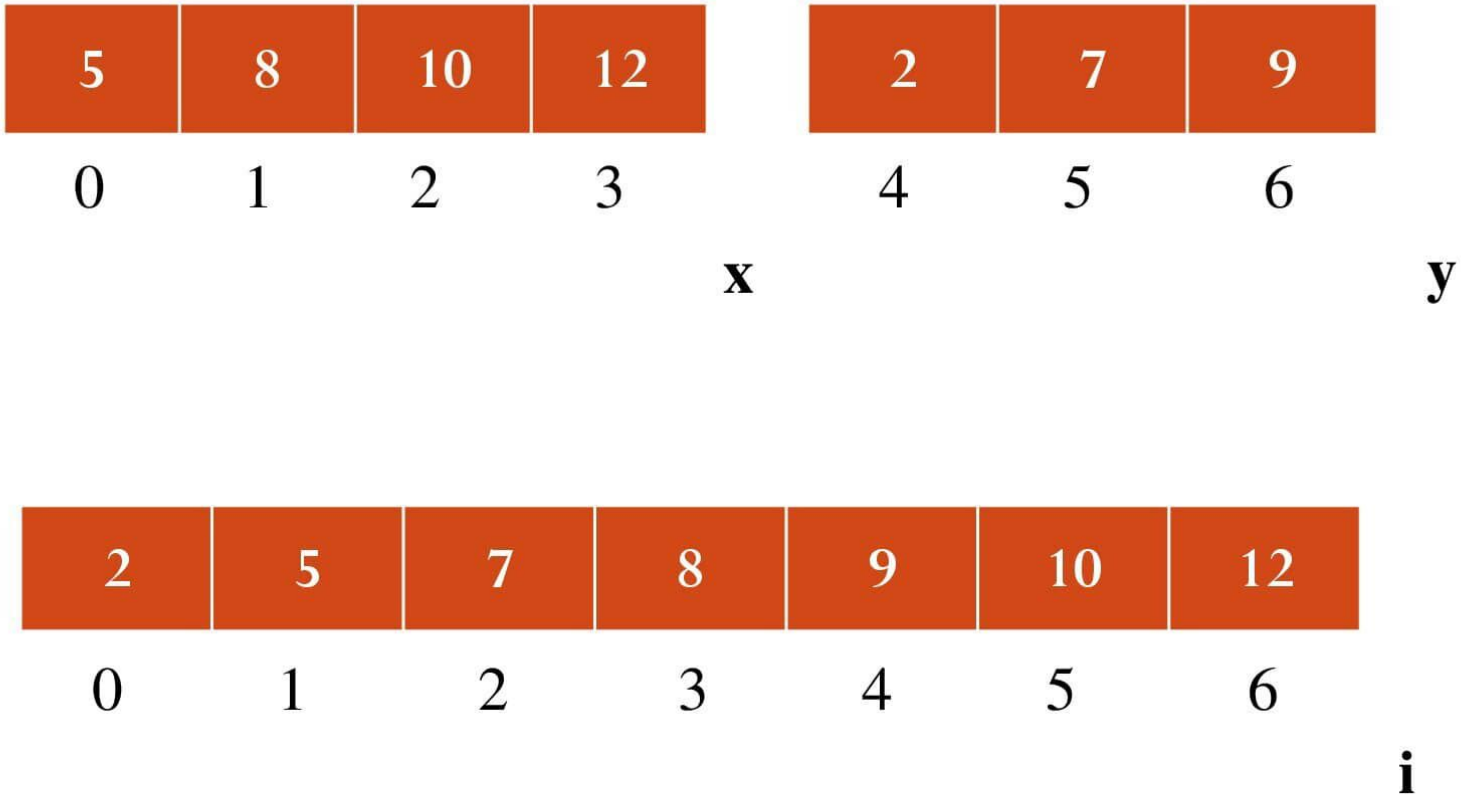
Merge



Merge



Merge



```
Algorithm Merge(low, mid, high)
{       i= low; x= low;  y= mid + 1;
        while((x ≤ mid) and (y ≤ high)) do
        {   if ( a[x] ≤ a[y] ) then
            {       b[i] = a[x];
                    x = x+1;
            }
            else
            {       b[i] = a[y];
                    y = y+1;
            }
            i=i+1;
        }
        if( x ≤ mid) then
        {   for k=x to mid do
            {       b[i] = a[k];
                    i =i+1;
            }
        }
        else
        {   for k=y to high do
            {       b[i] = a[k];
                    i =i+1;
            }
        }
        for k= low to high do
            a[k] = b[k];
}
```

Quick Sort

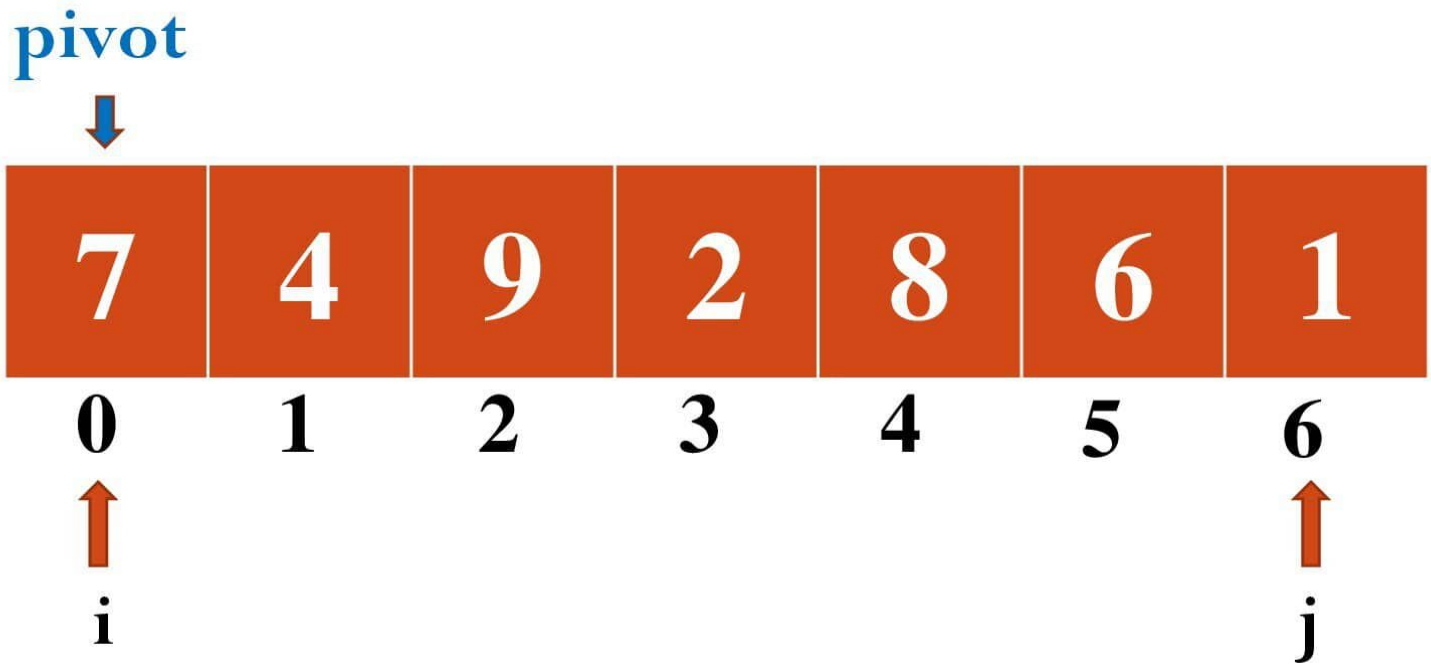
1. Let **A** be an array with n elements.
2. Select an element 'x' from any position, called a **pivot**, from the list.
3. Partition **A** into two, so that x is placed in position j such that
    1. Elements from position 0 to j-1 is less than x
    2. Elements from position j+1 to n is greater than x

    This is called a partition operation

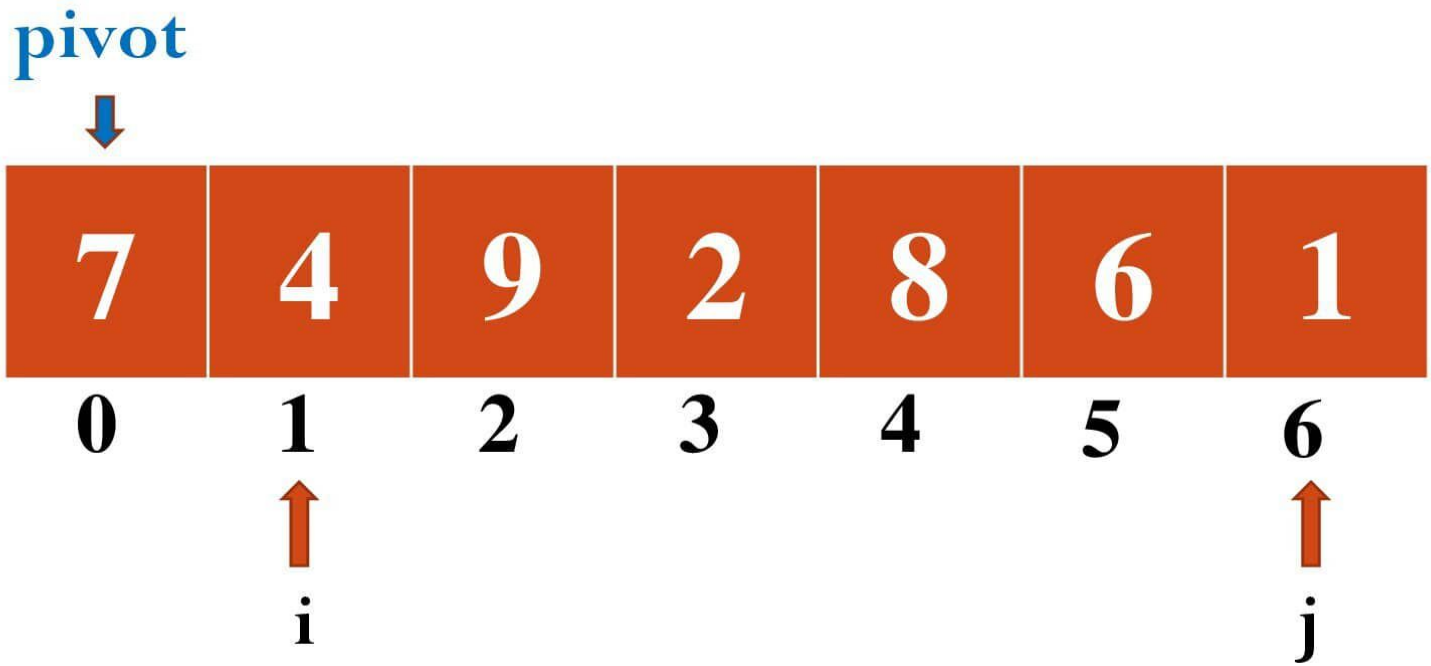
    After this partitioning, the pivot is in its final position
4. Recursively sort the sub-list of lesser elements and the sub-list of greater elements

Quick Sort

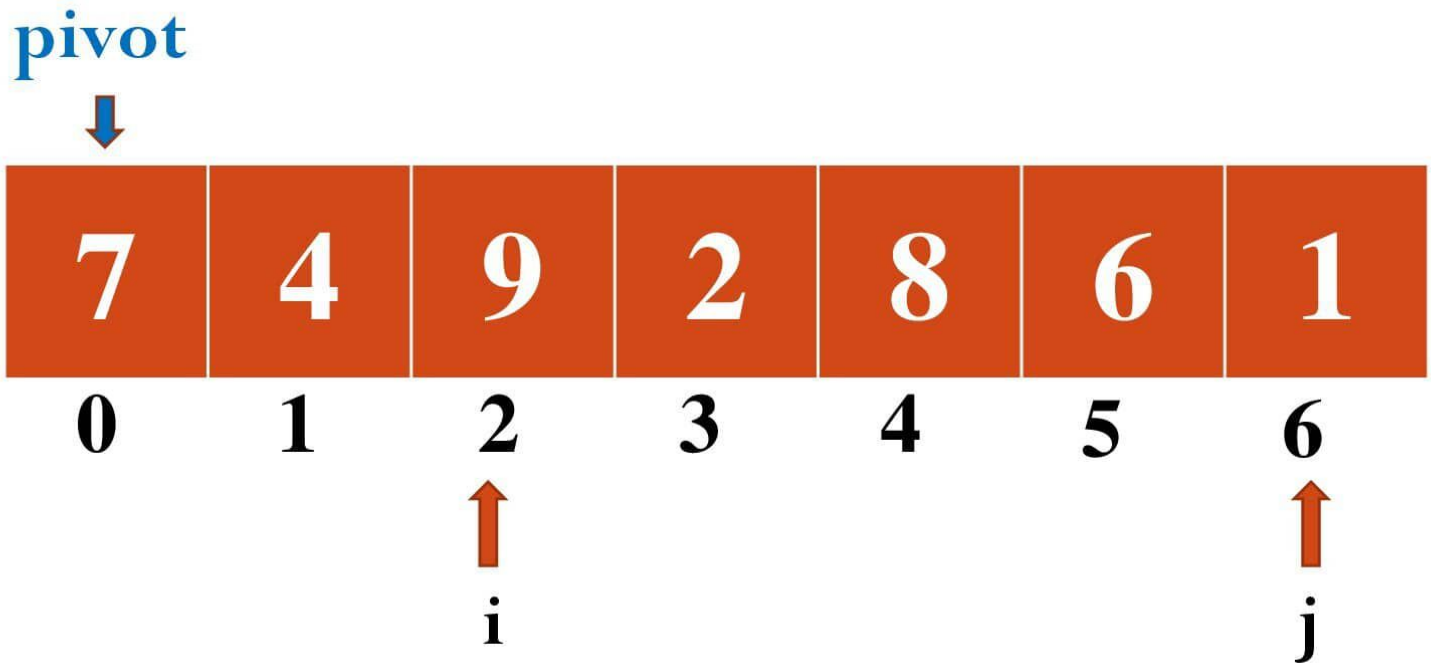
Quick Sort



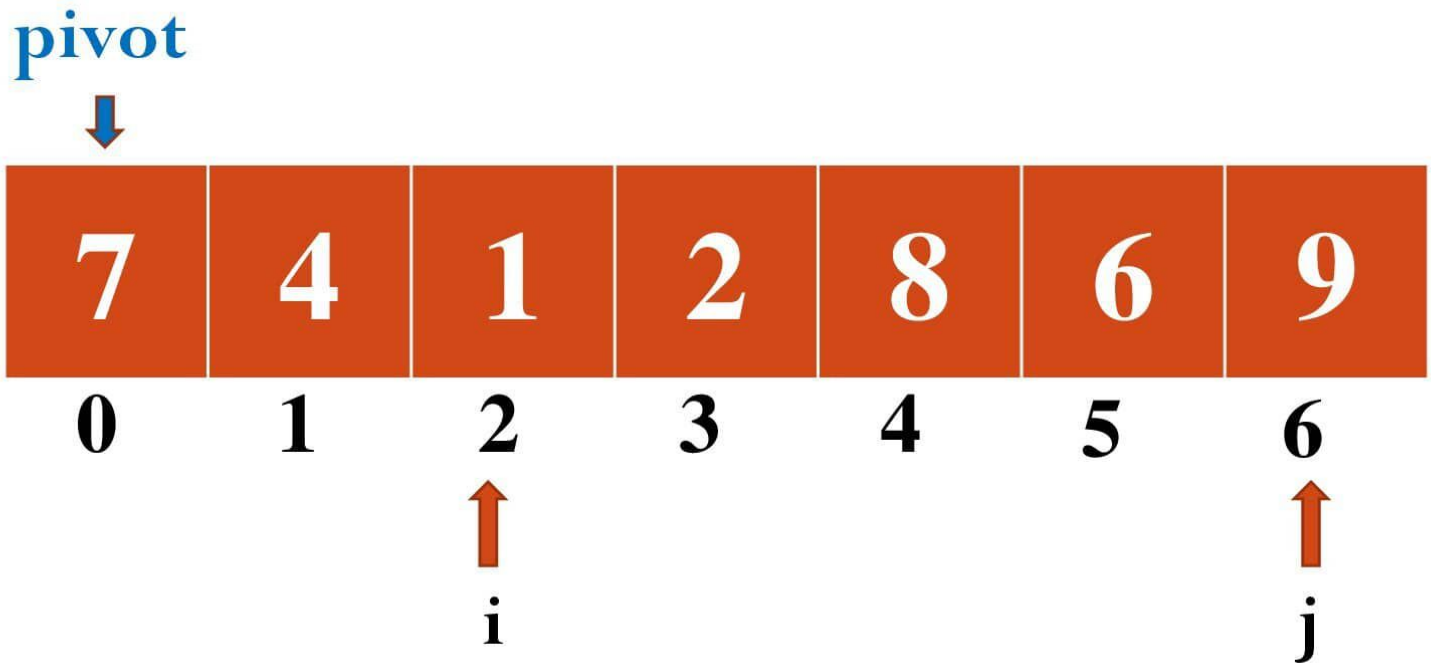
Quick Sort



Quick Sort

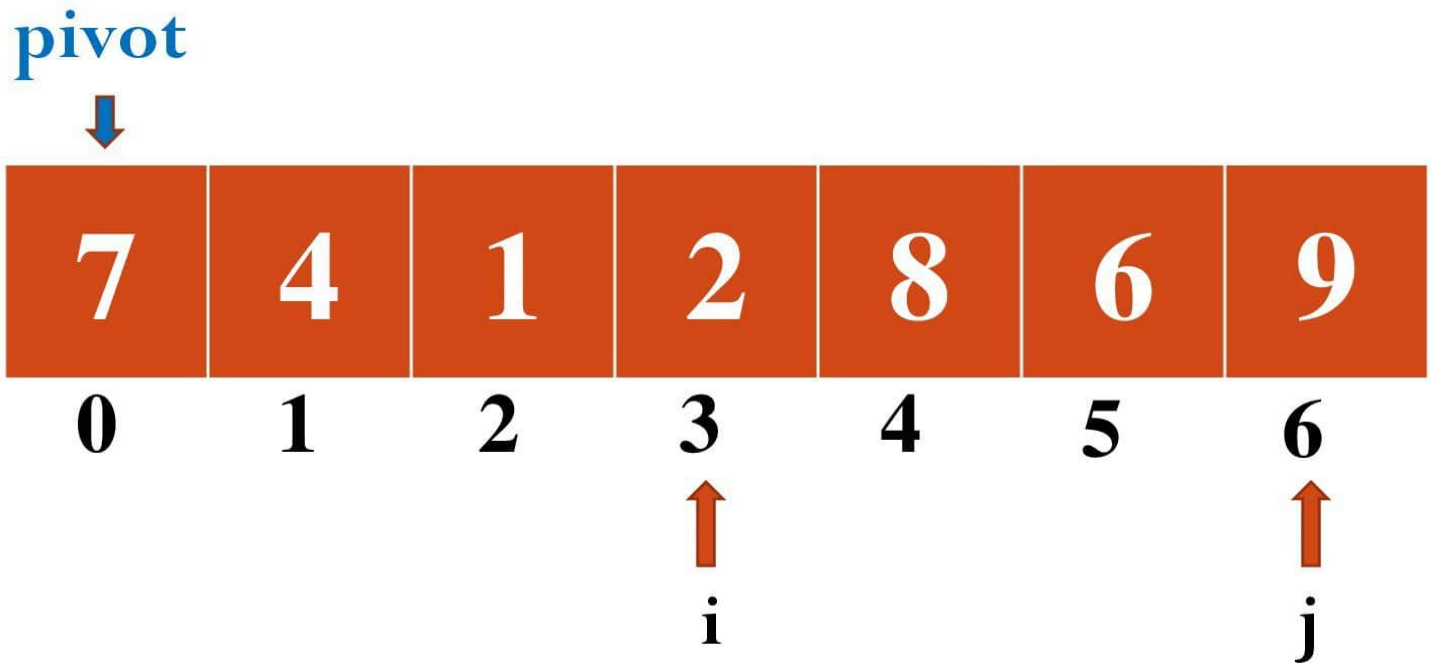


Quick Sort

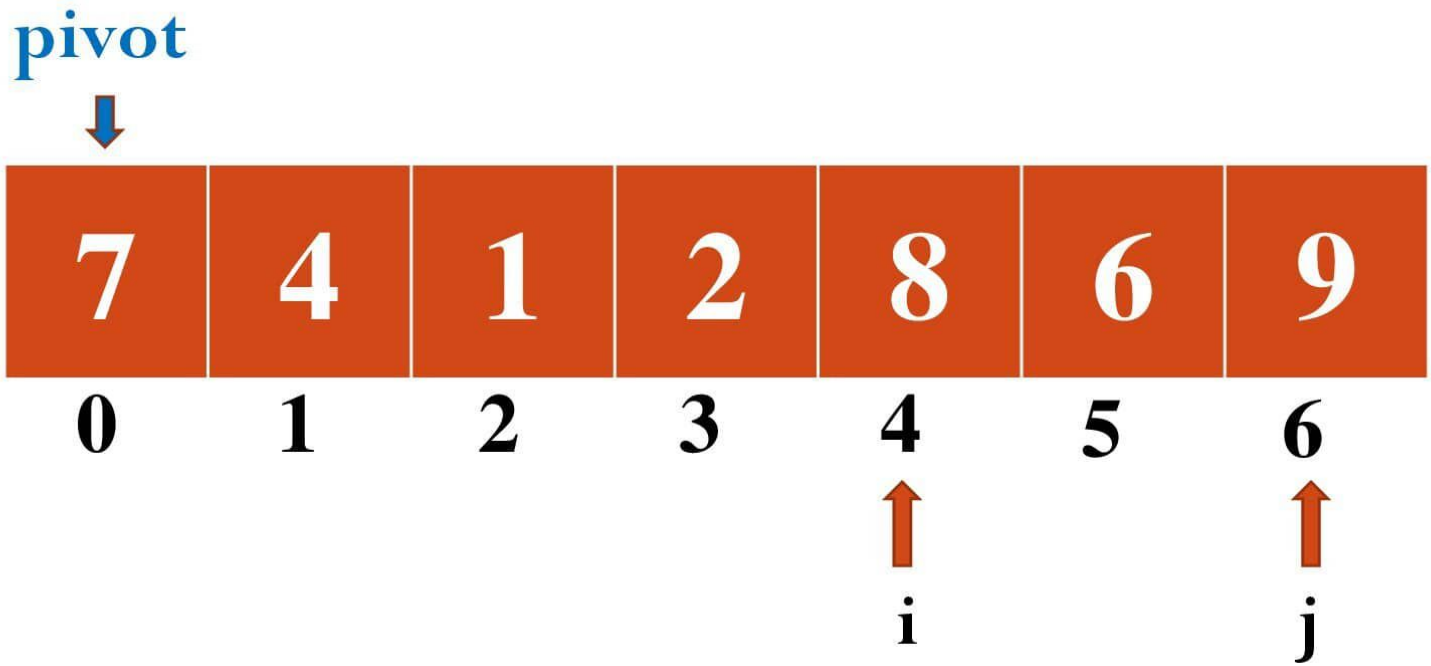


Swap A[i] and A[j]

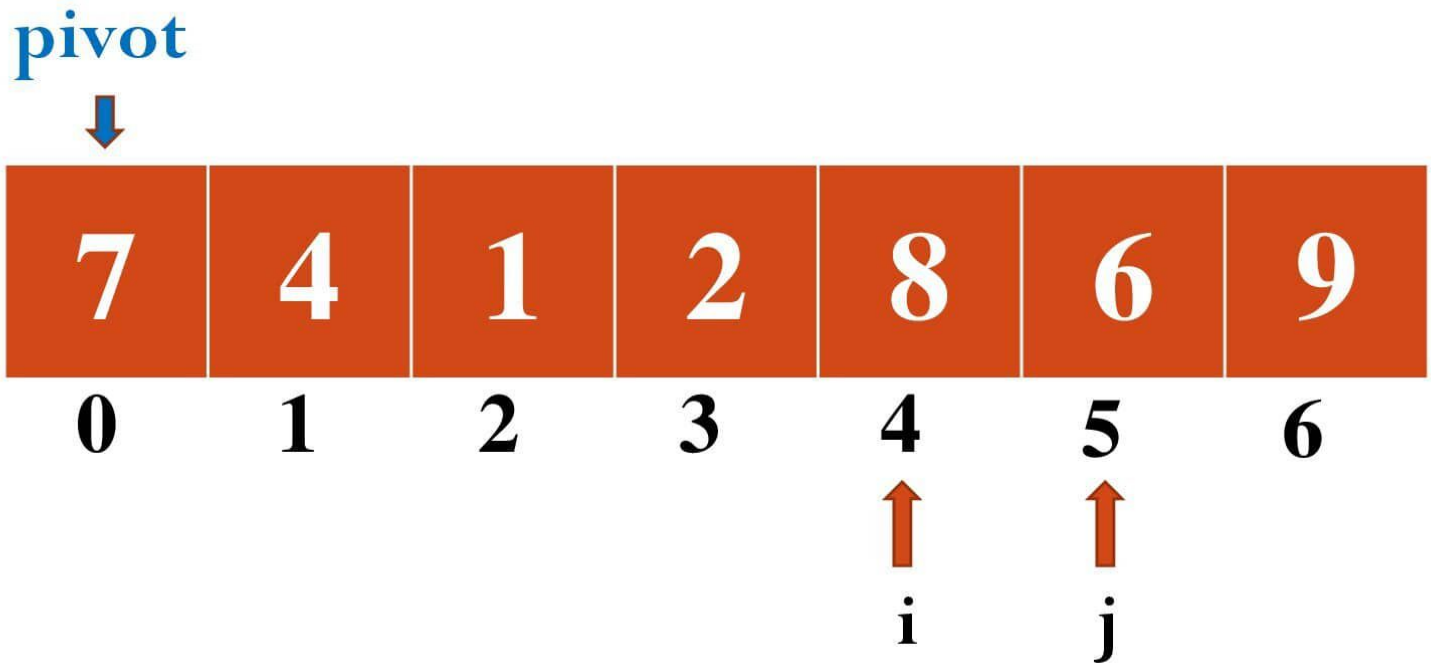
Quick Sort



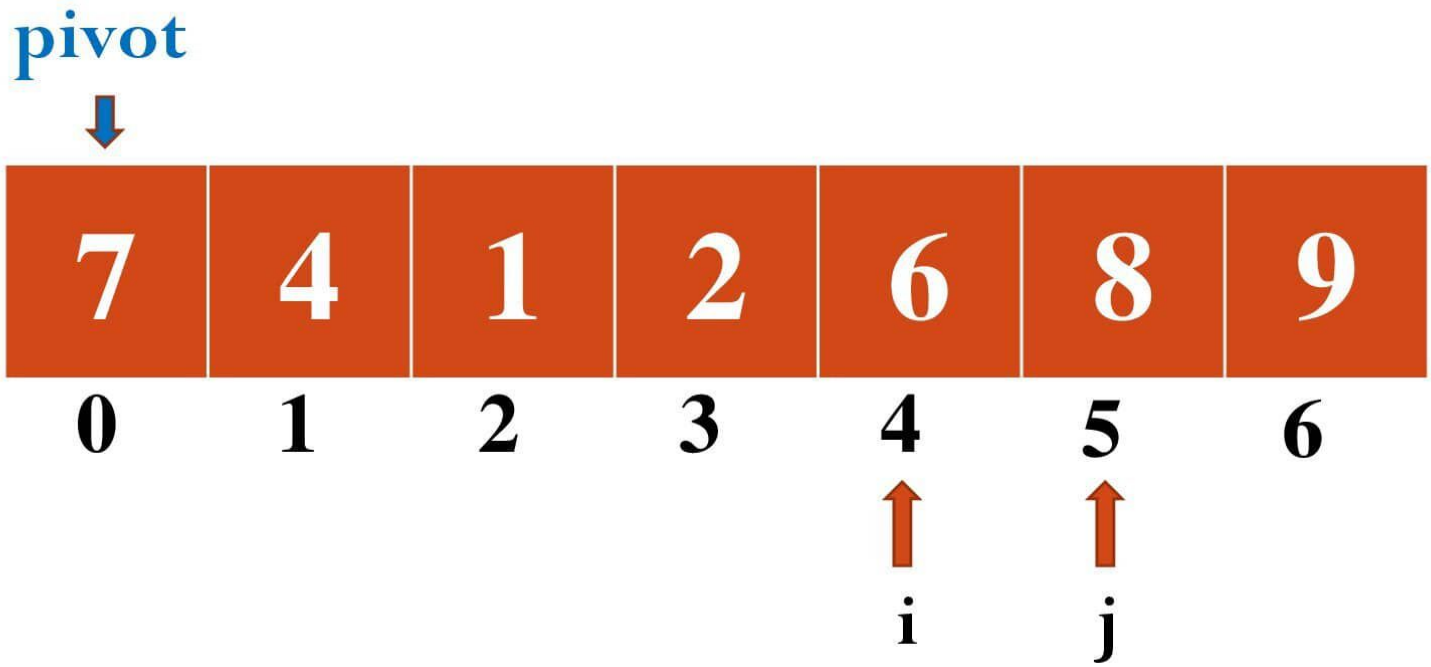
Quick Sort



Quick Sort

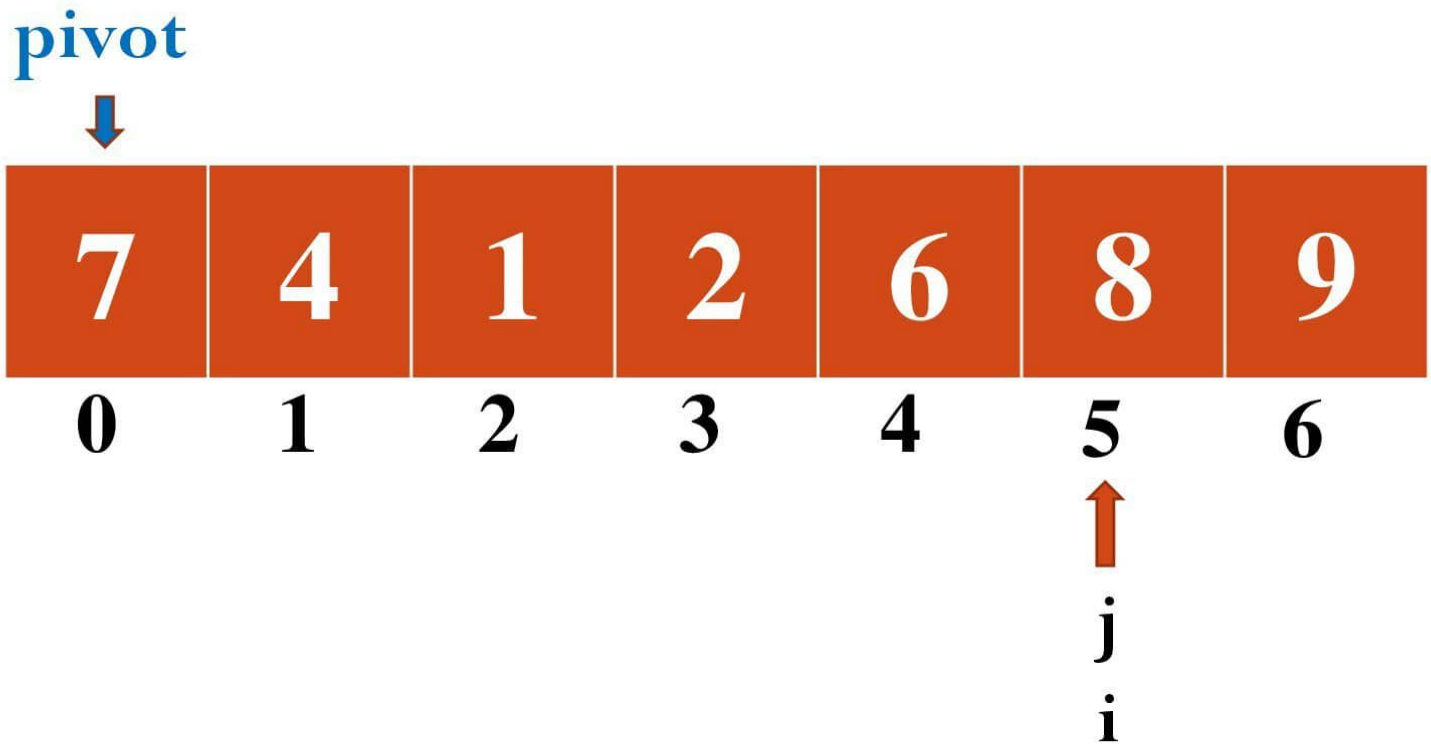


Quick Sort

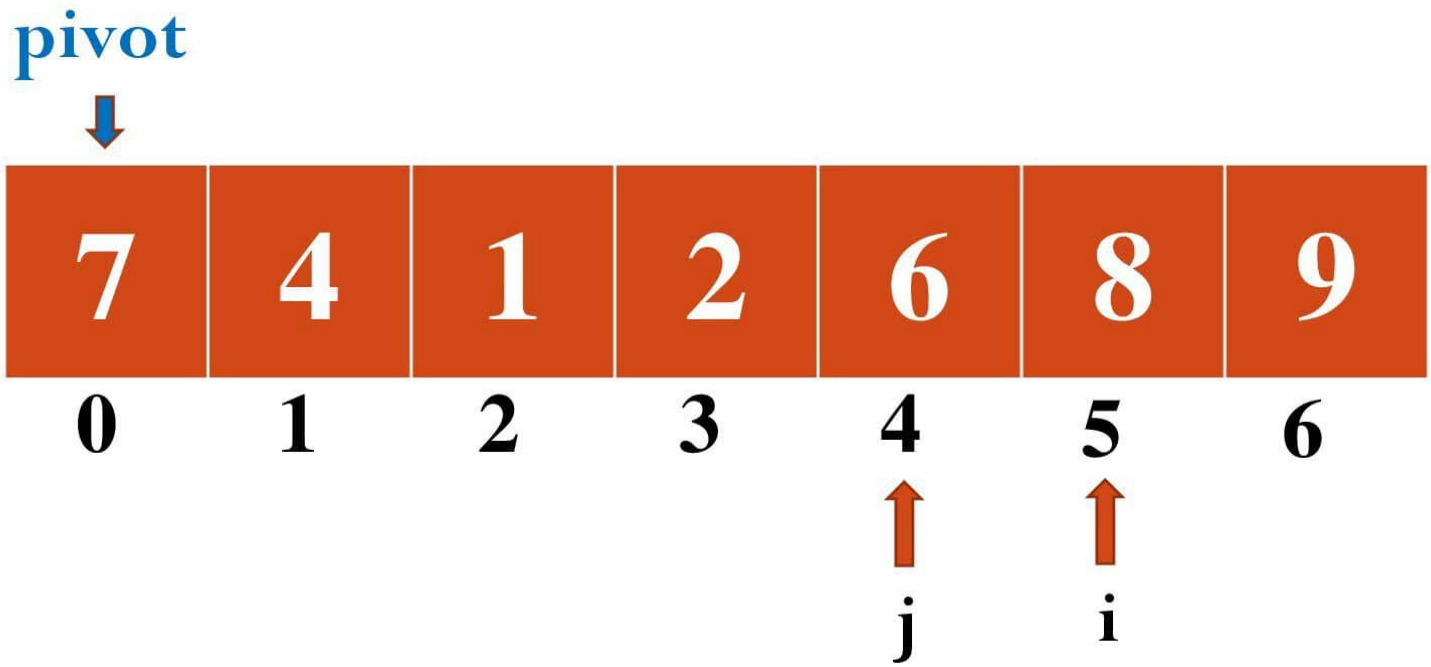


Swap A[i] and A[j]

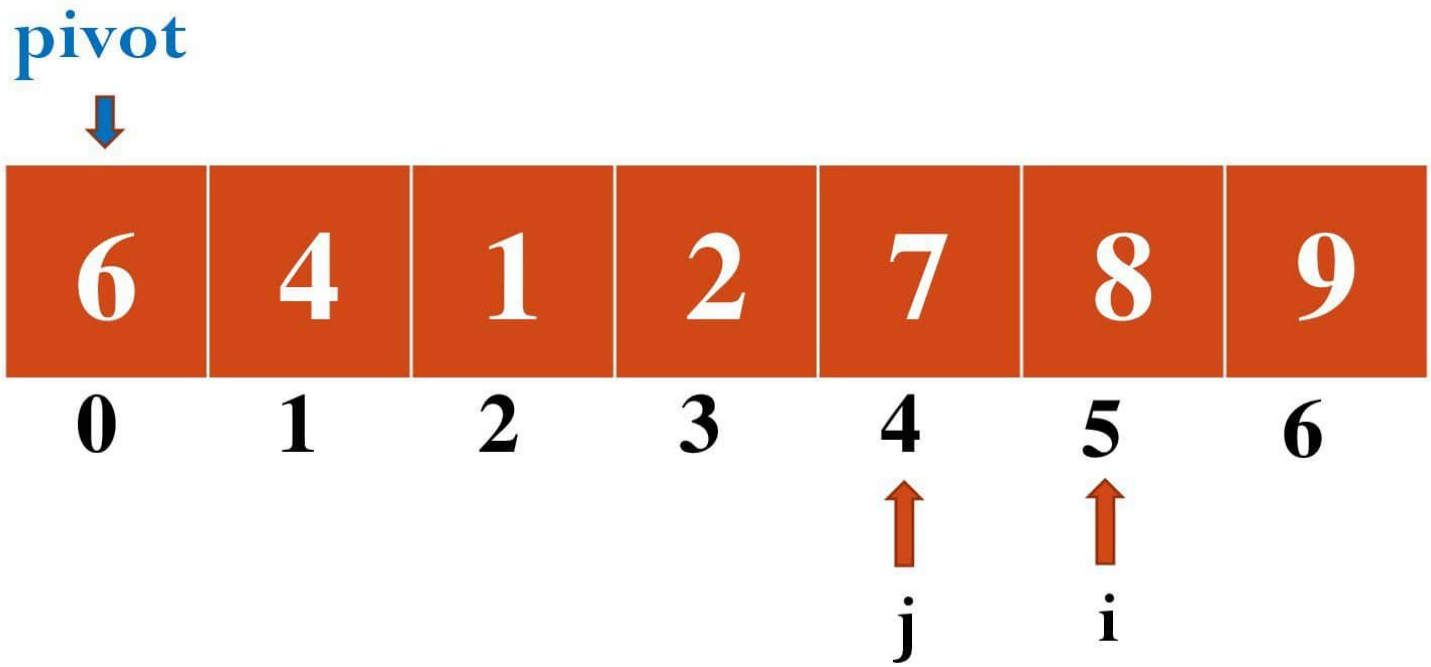
Quick Sort



Quick Sort



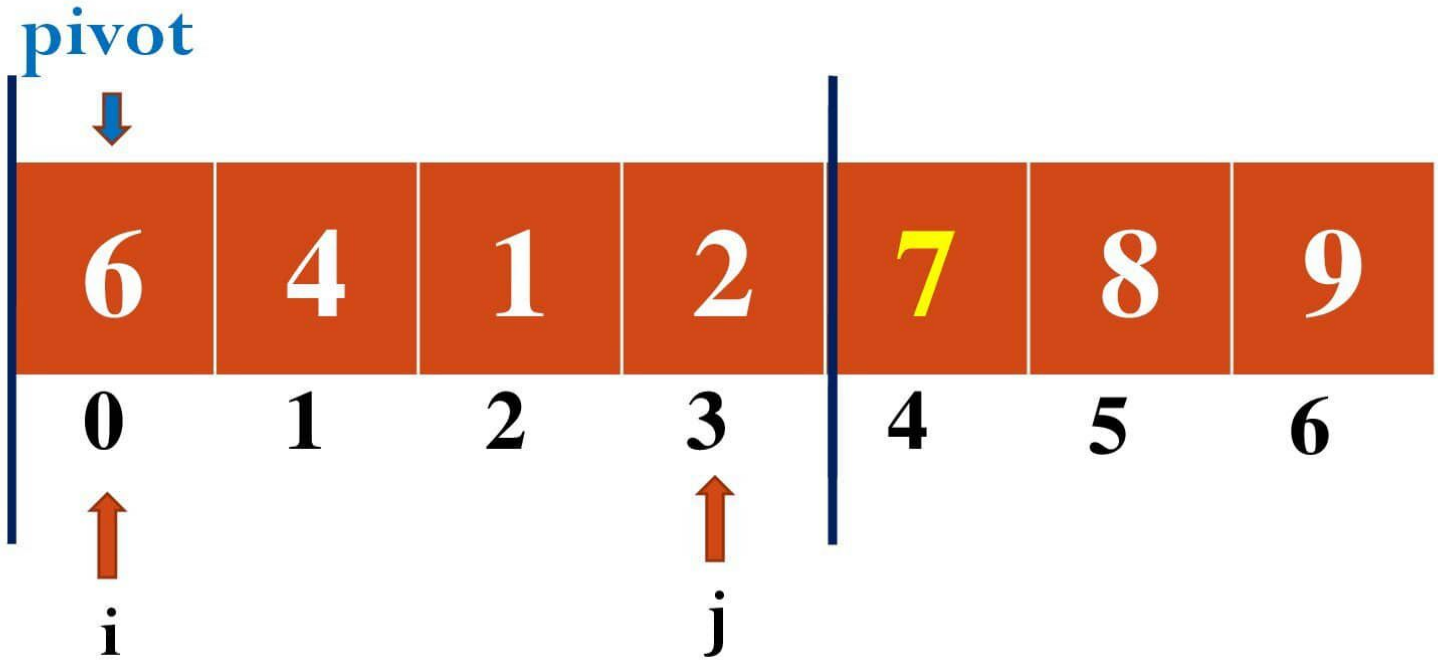
Quick Sort



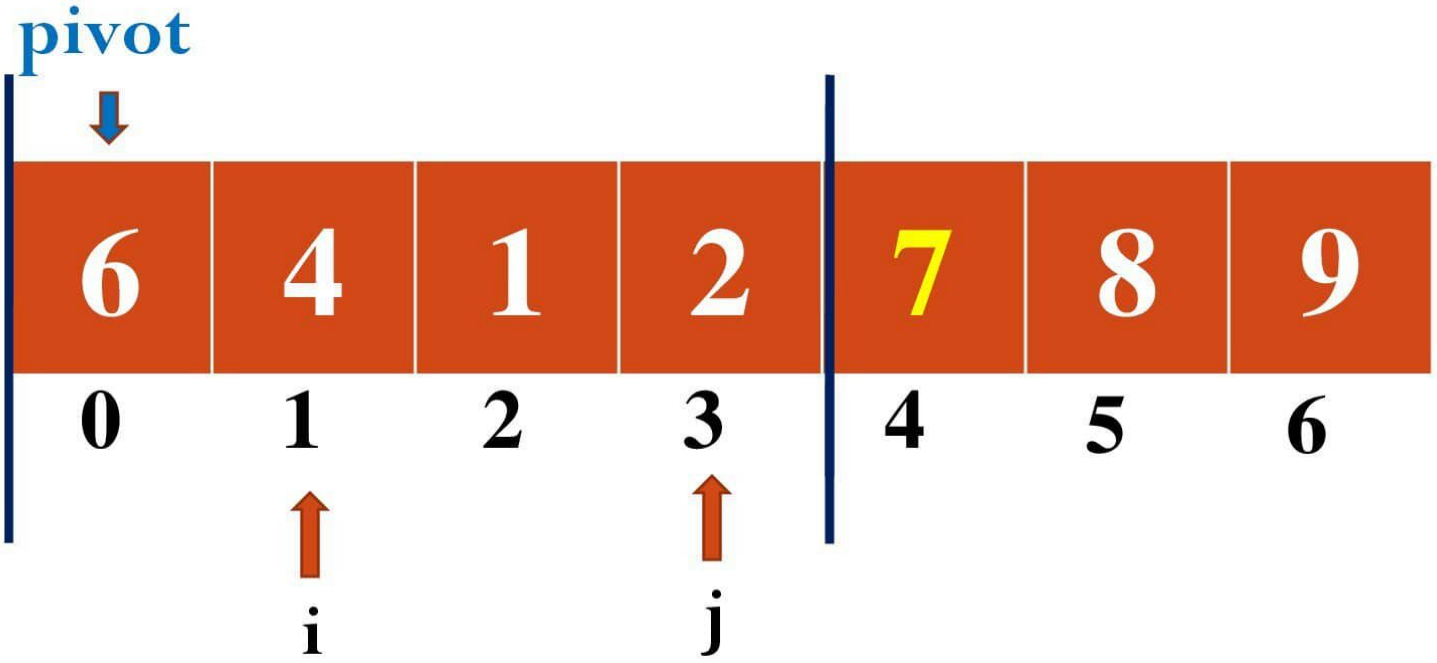
Swap A[pivot] and A[j]

Quick Sort

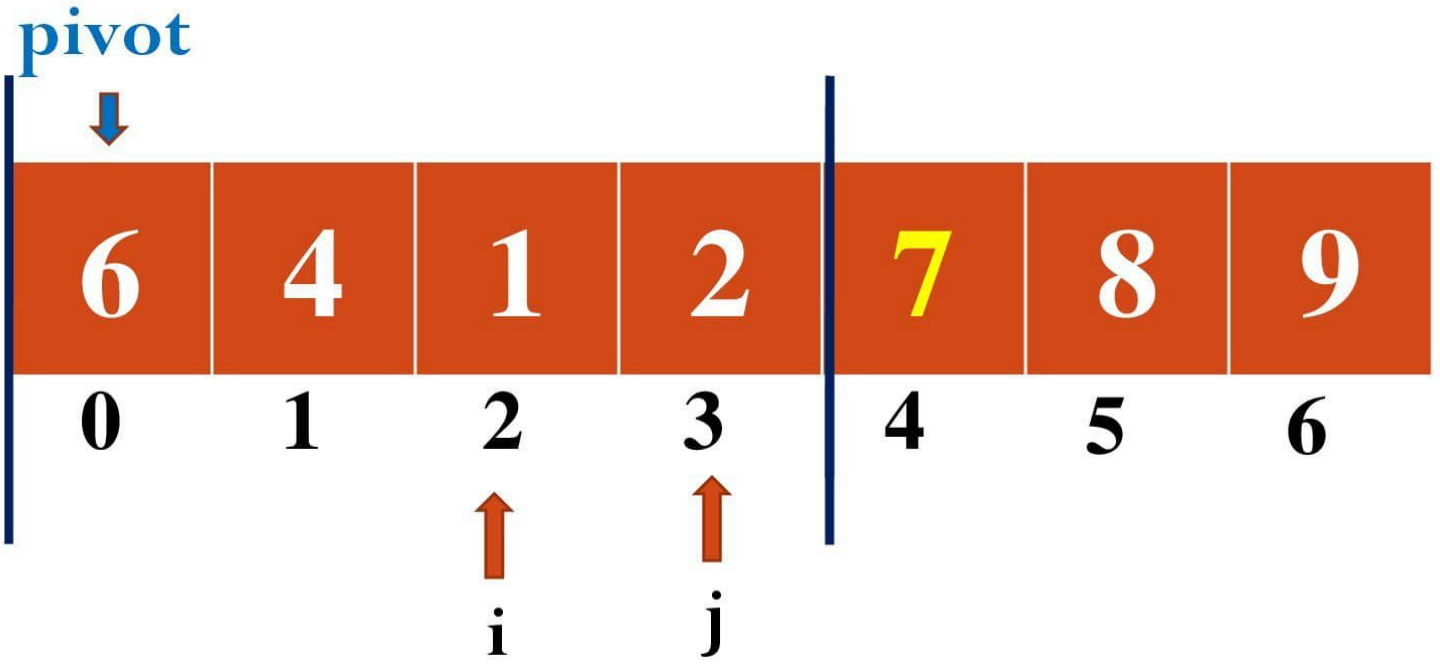
Quick Sort



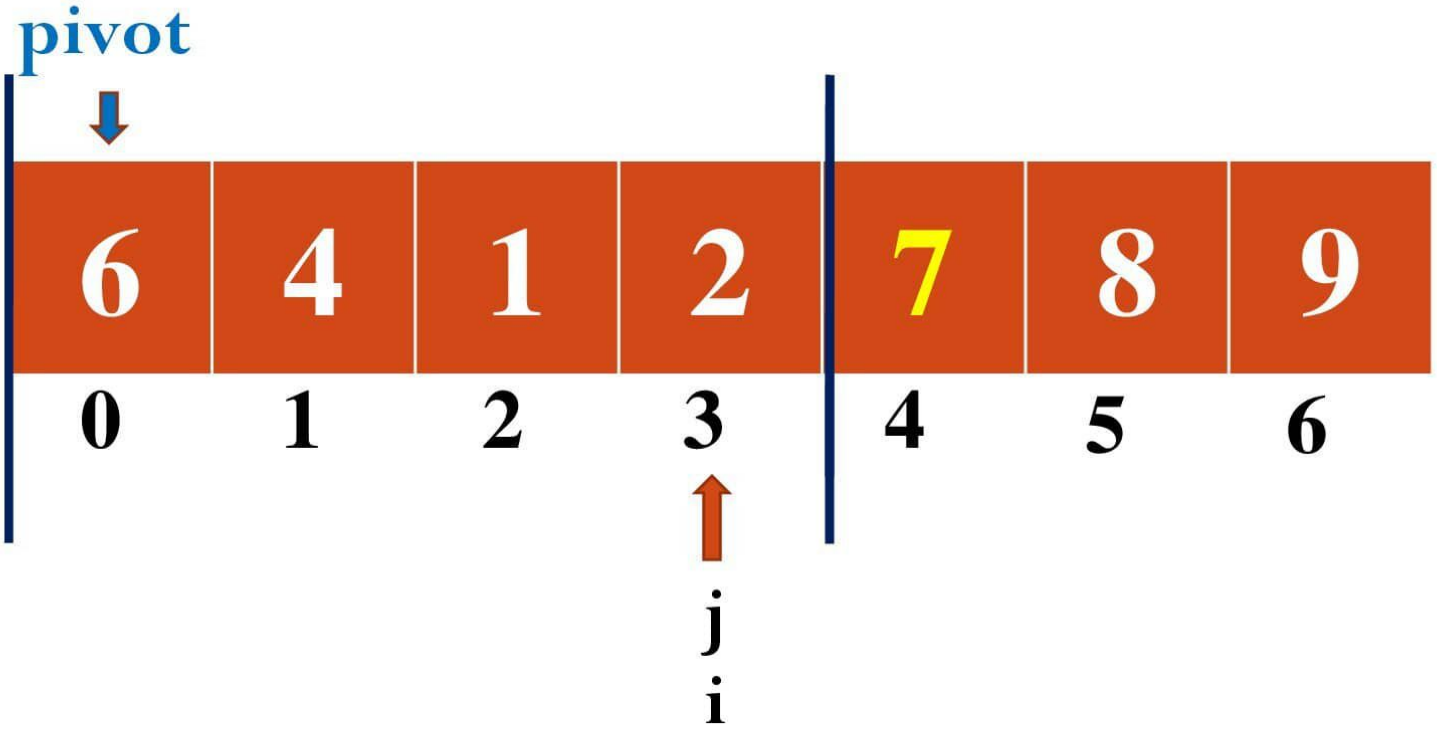
Quick Sort



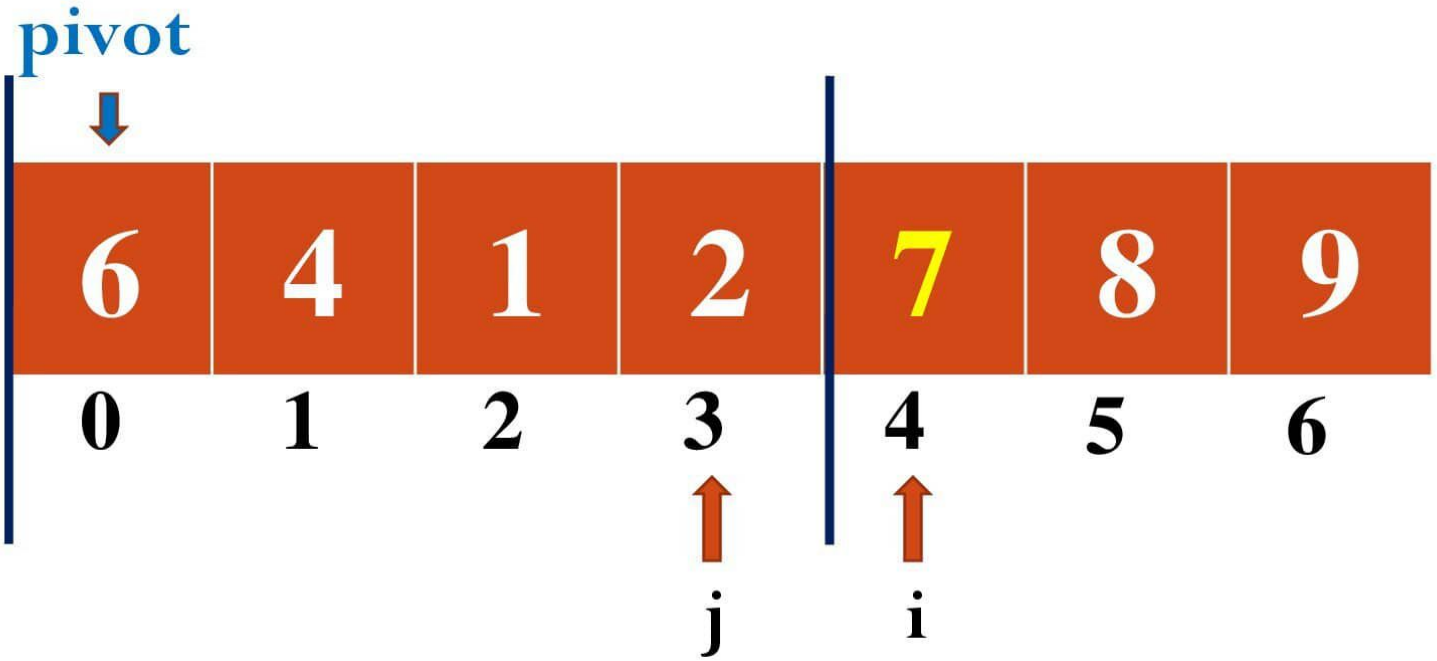
Quick Sort



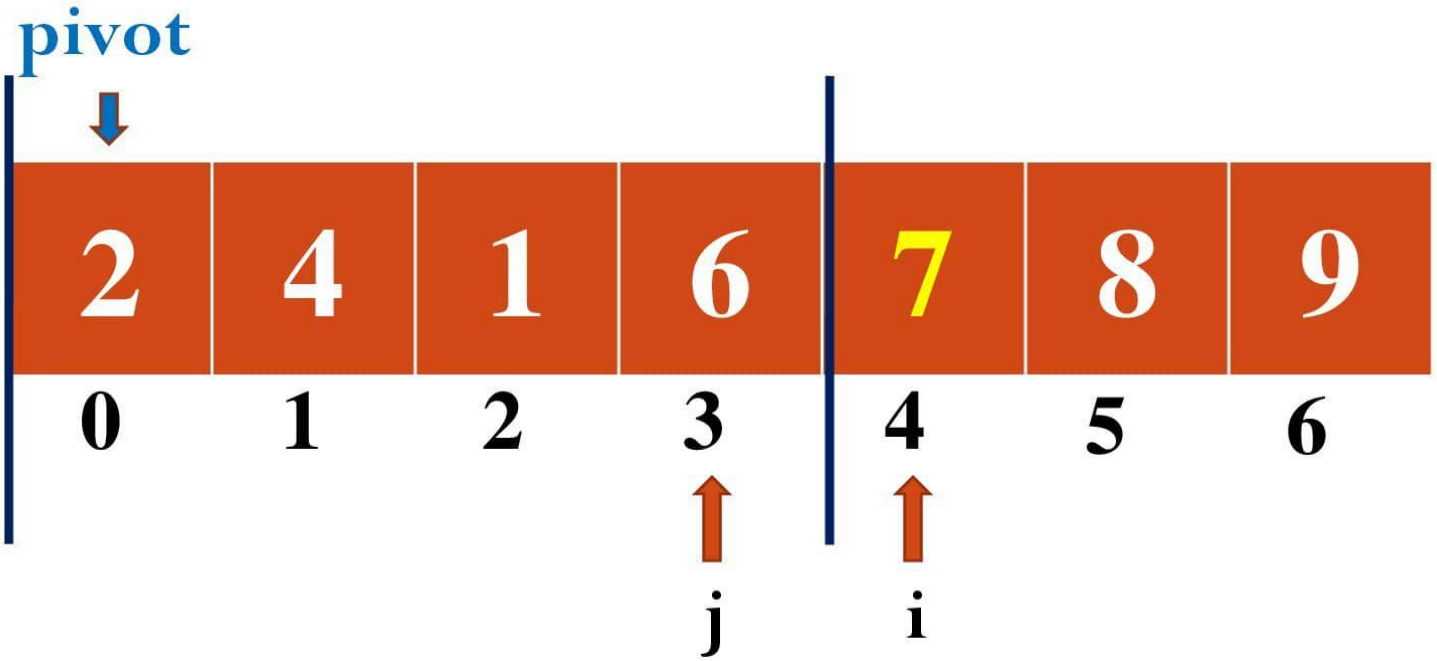
Quick Sort



Quick Sort

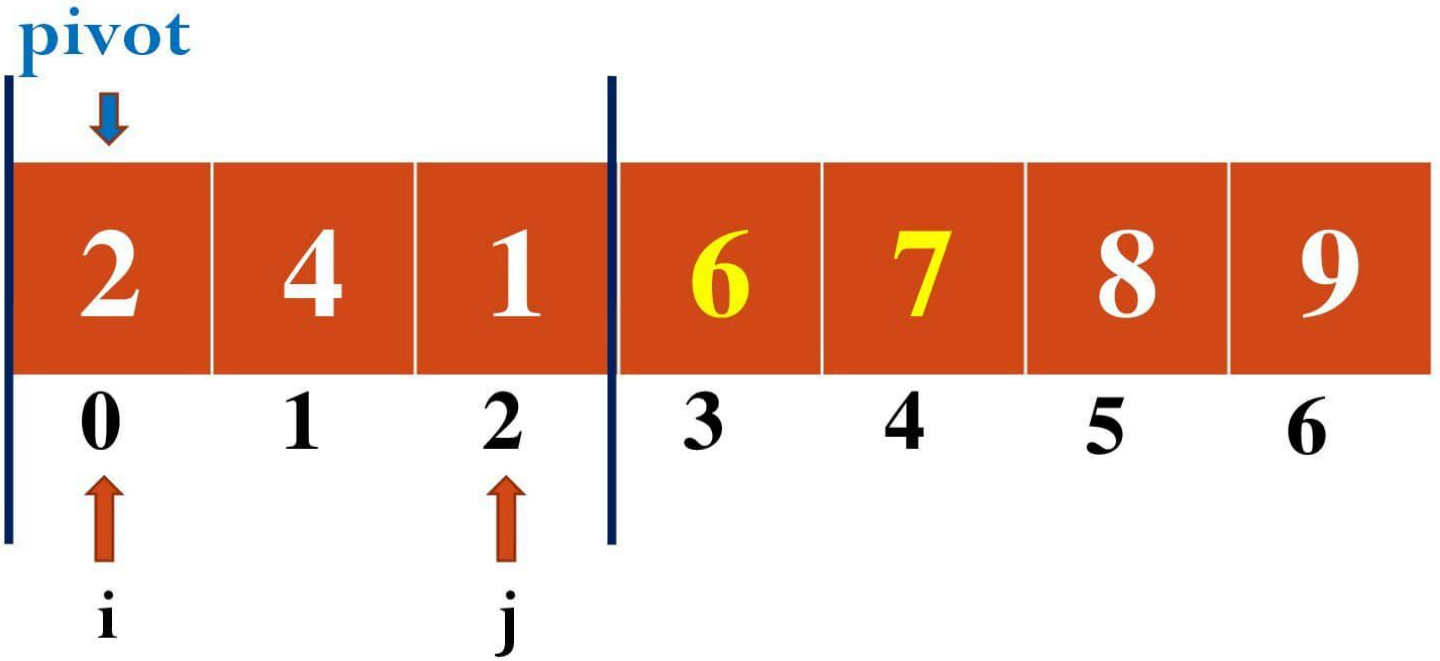


Quick Sort

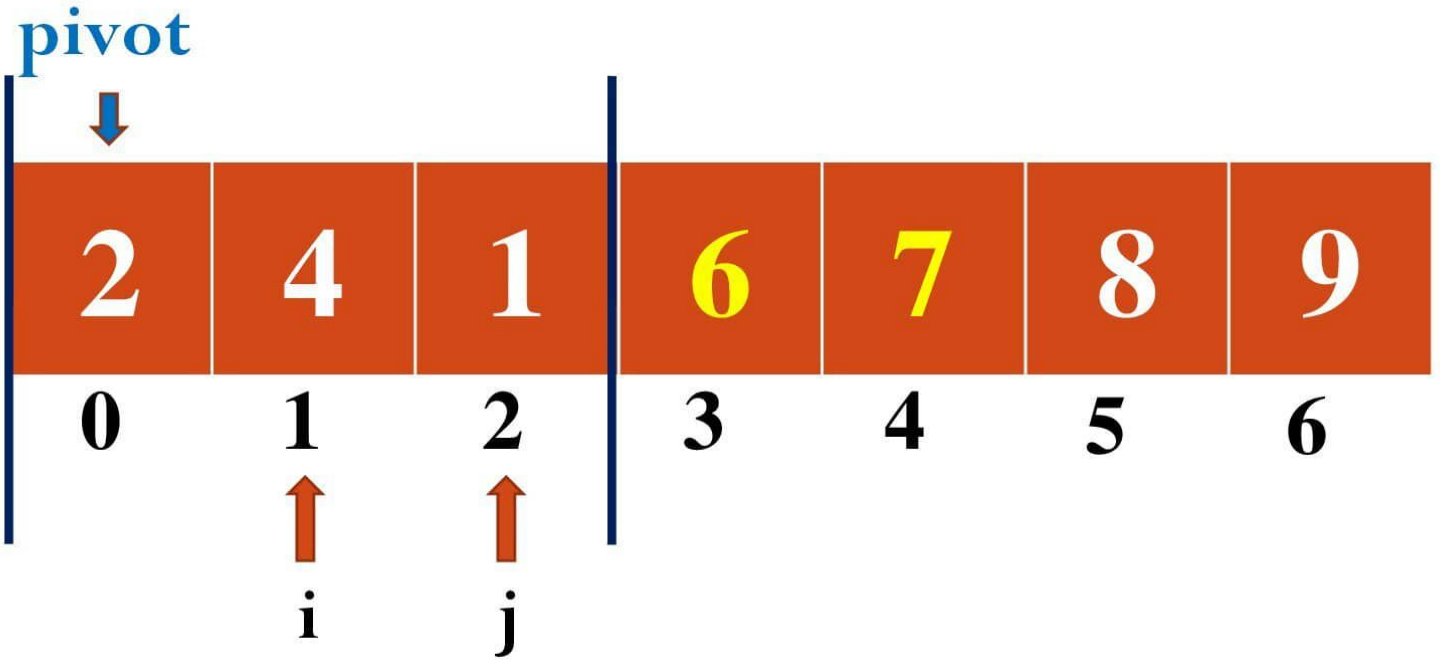


Swap A[pivot] and A[j]

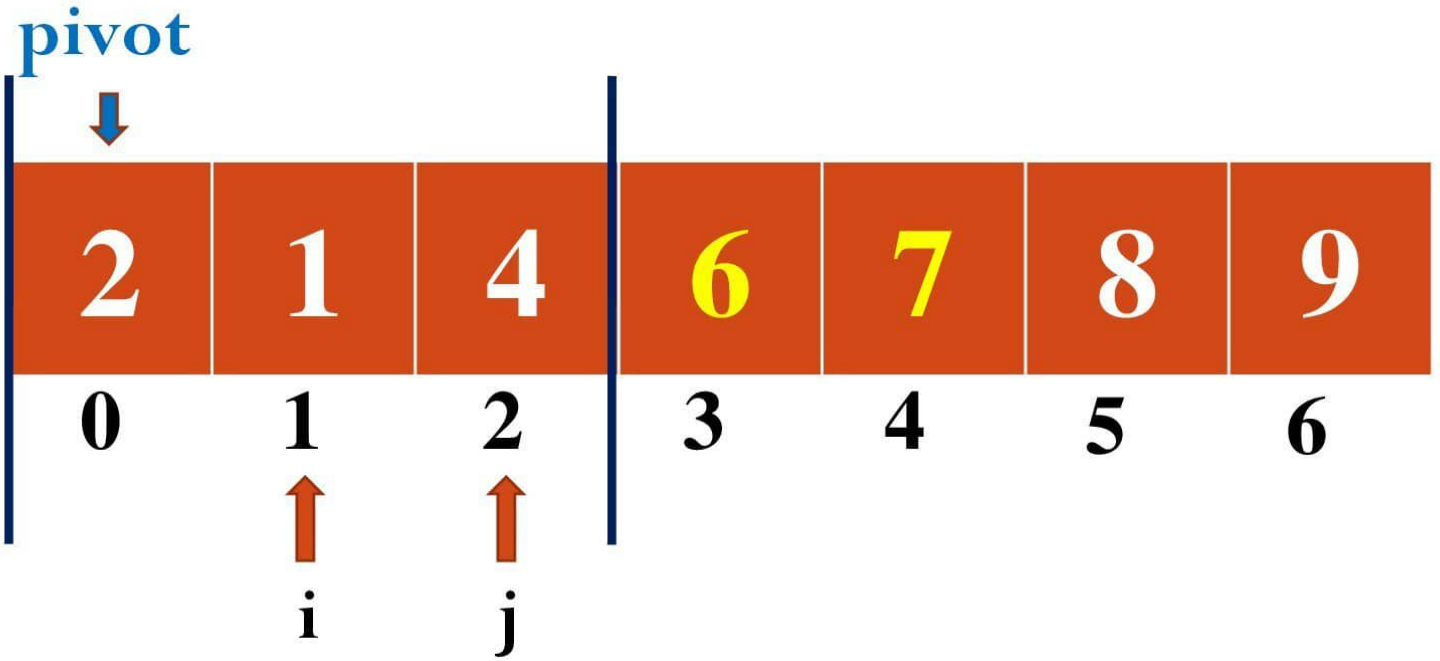
Quick Sort



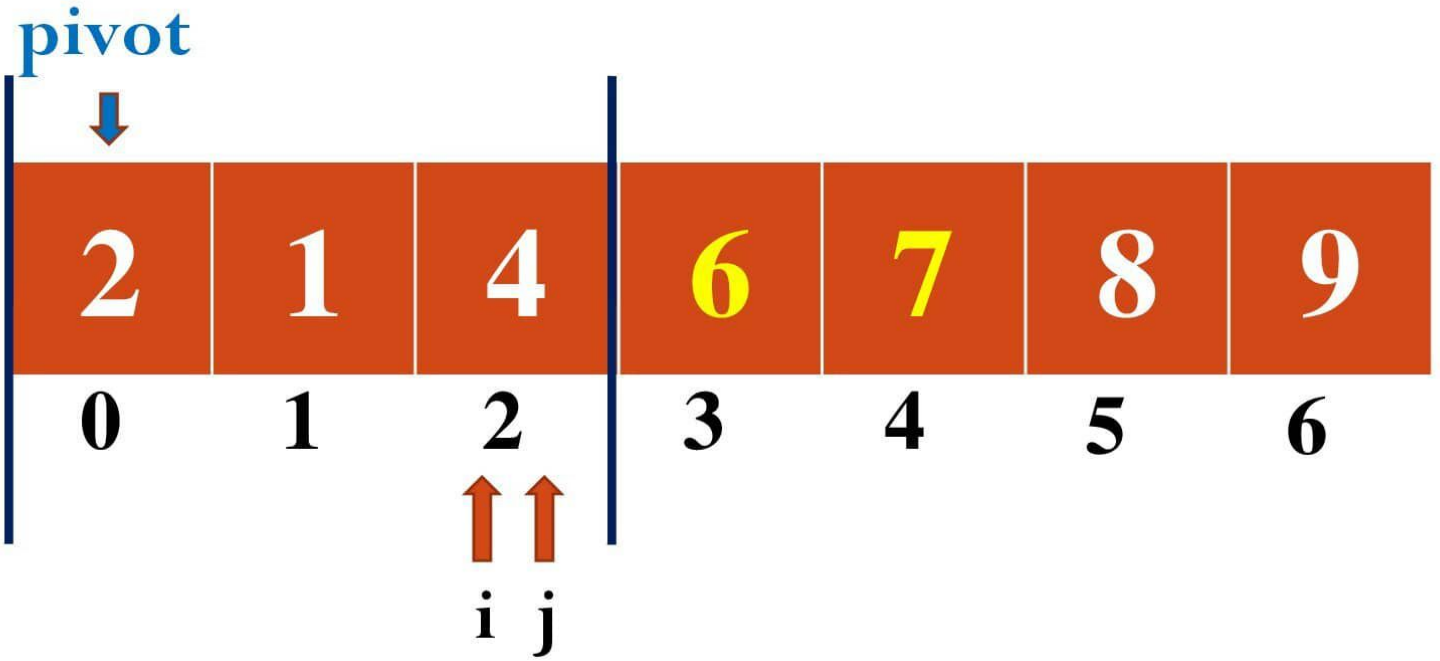
Quick Sort



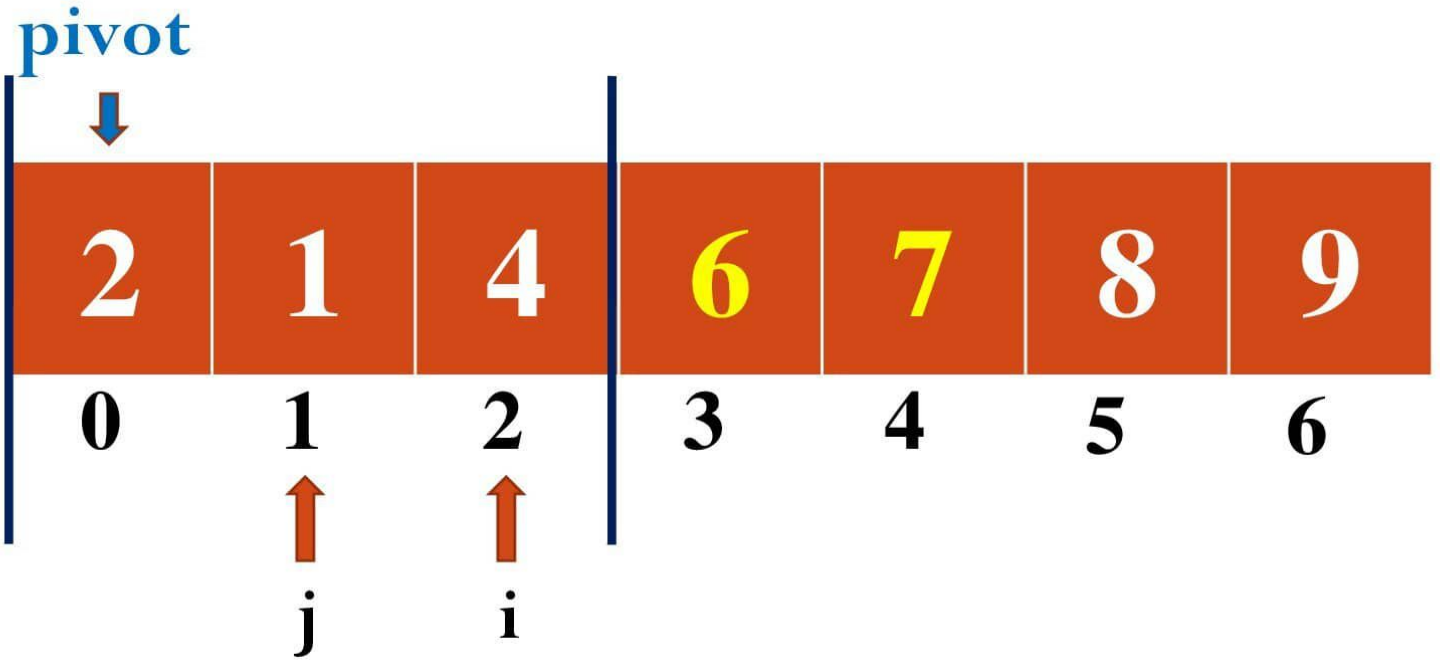
Quick Sort



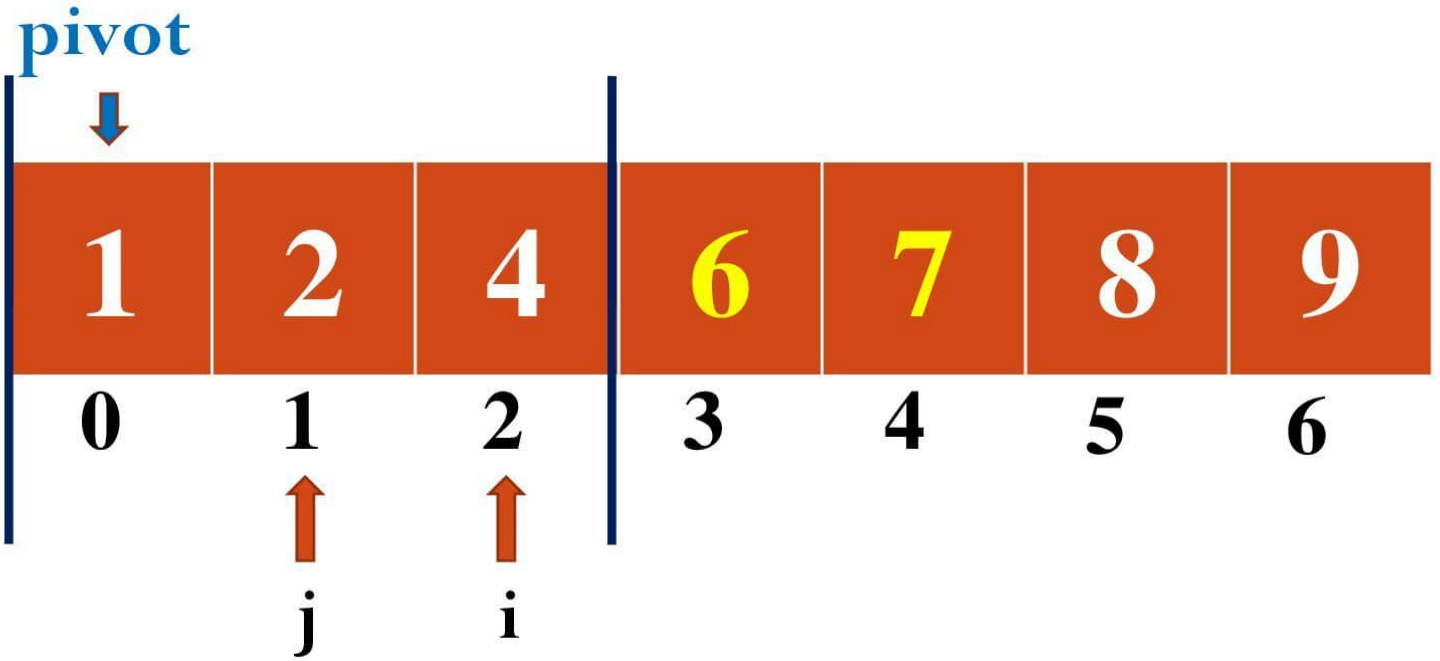
Quick Sort



Quick Sort

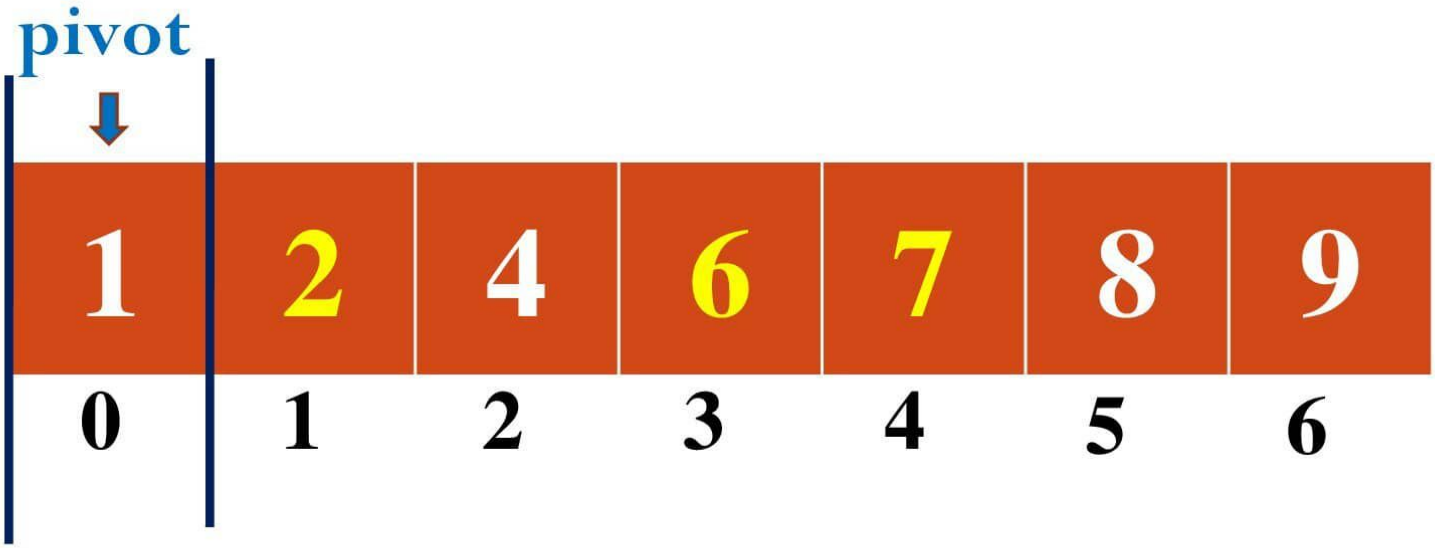


Quick Sort

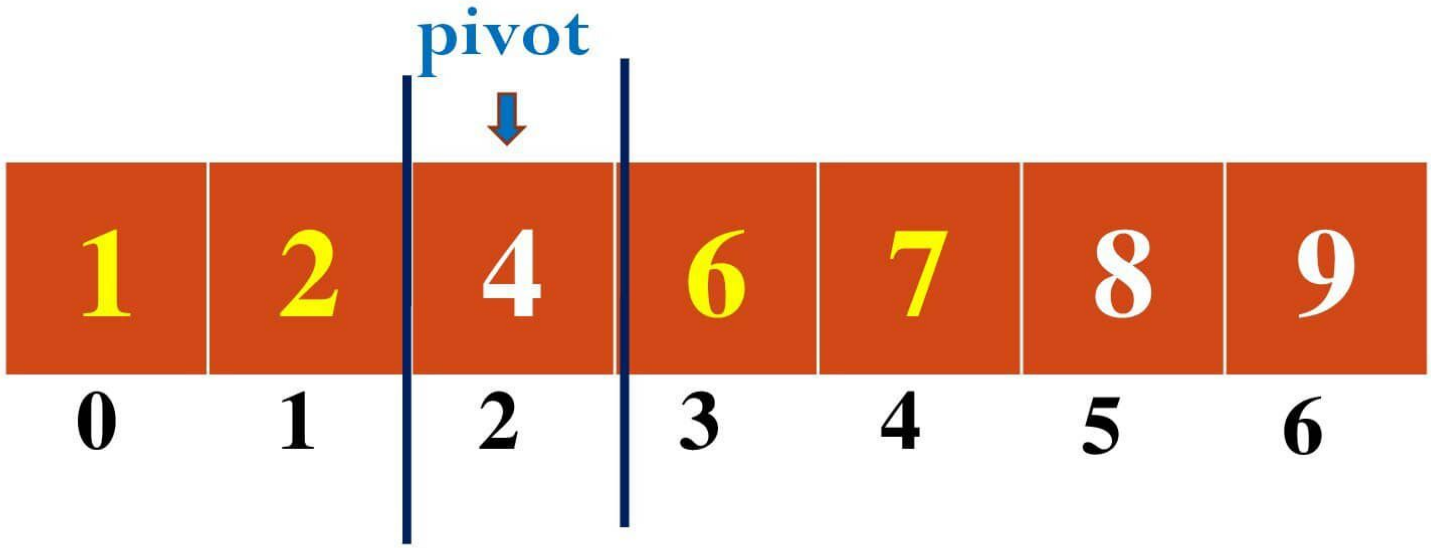


Swap pivot and A[j]

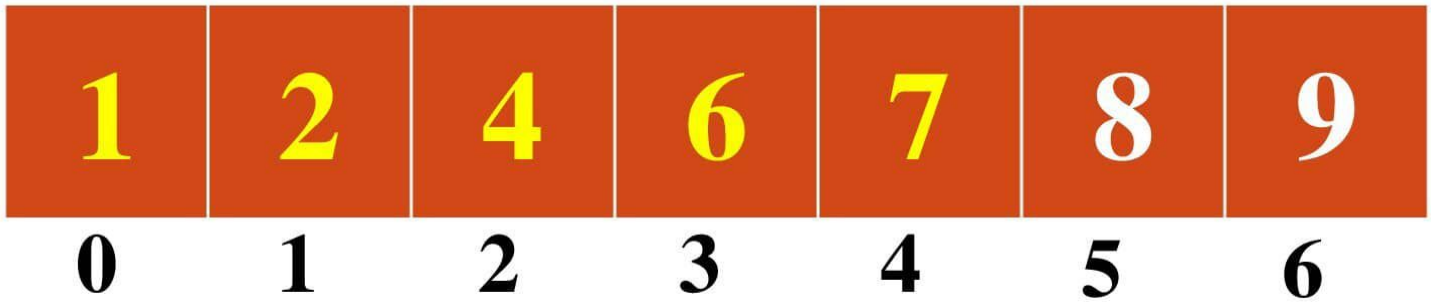
Quick Sort



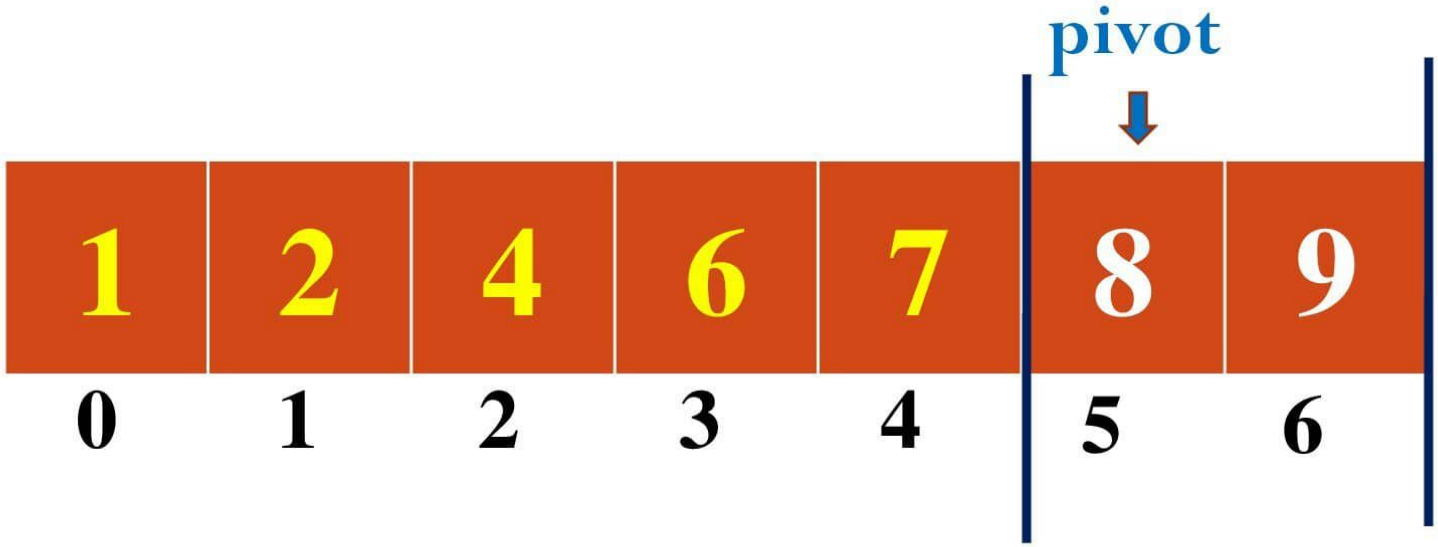
Quick Sort



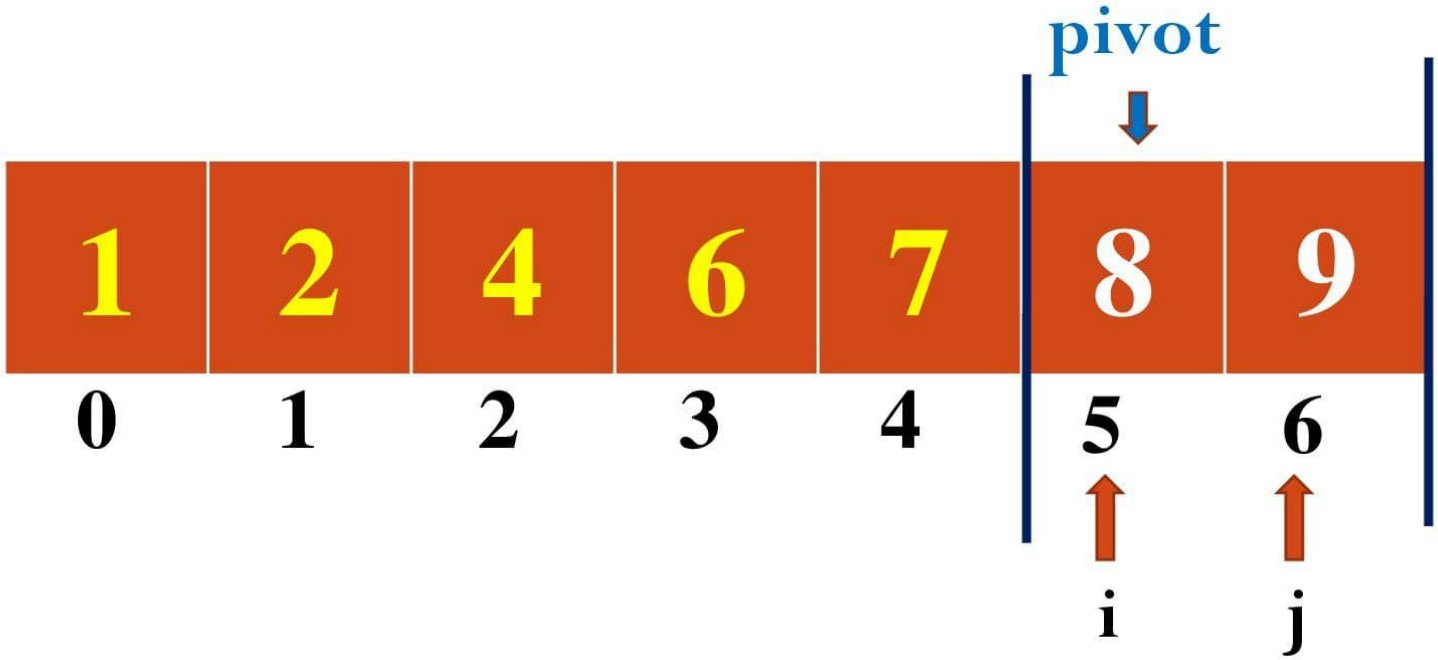
Quick Sort



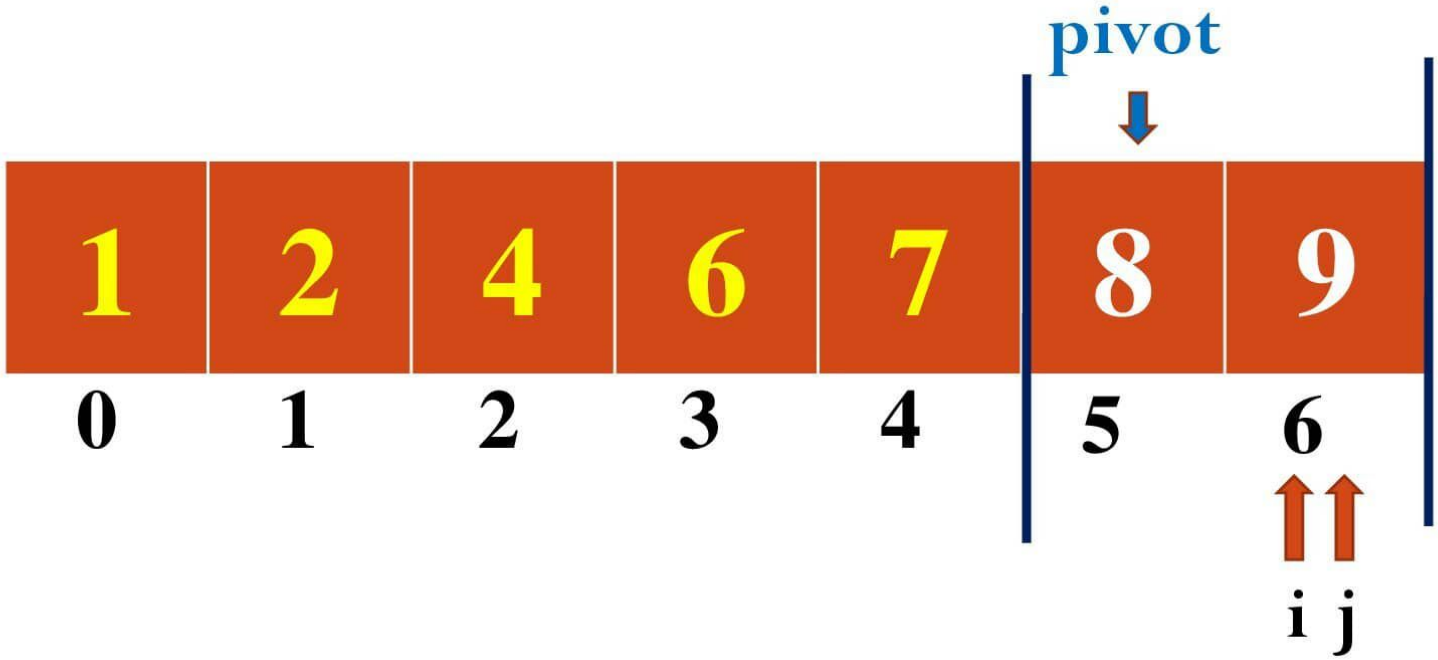
Quick Sort



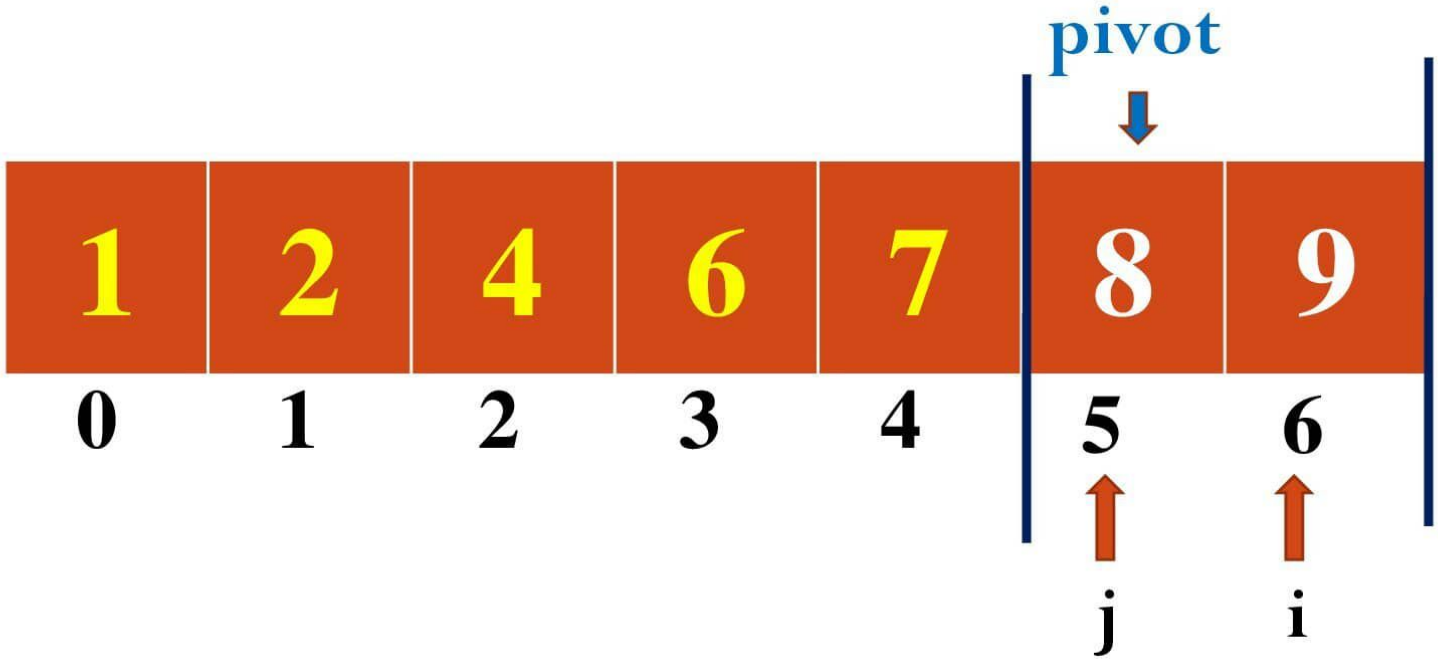
Quick Sort



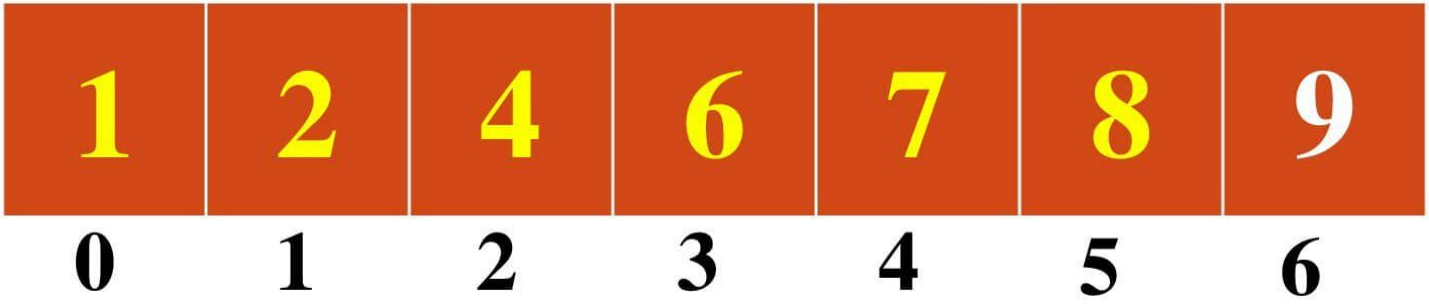
Quick Sort



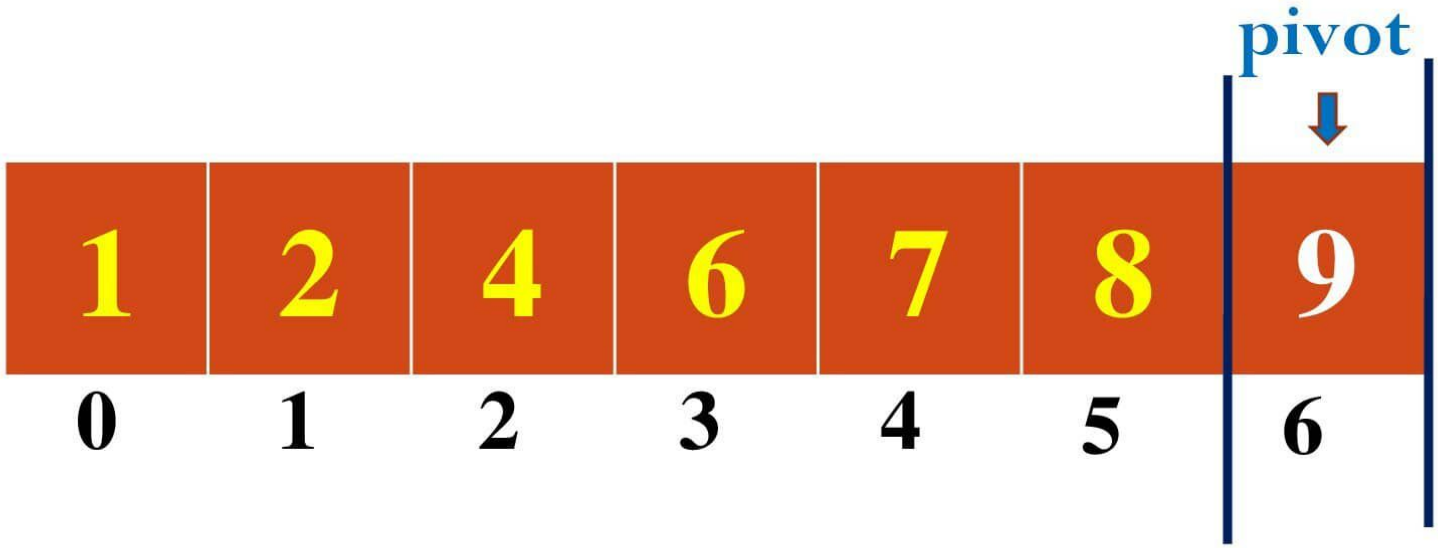
Quick Sort



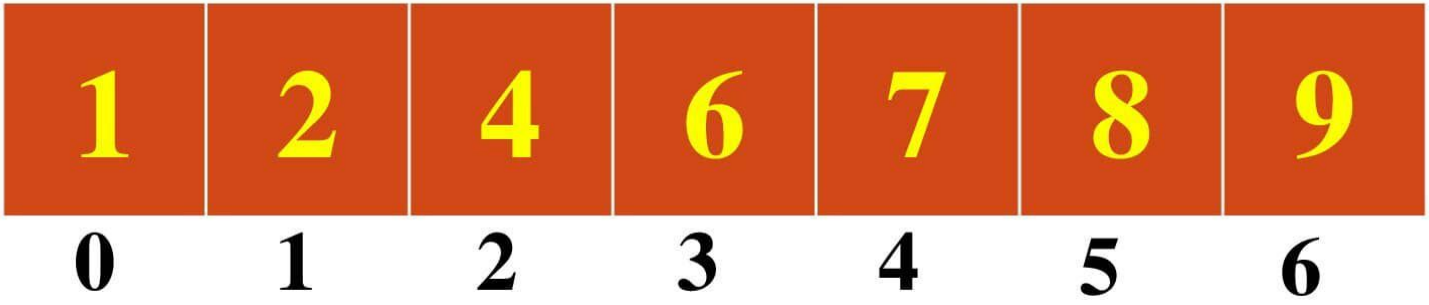
Quick Sort



Quick Sort



Quick Sort



Quick Sort

```
Algorithm QuickSort(A, low, high)
{
        if low < high then
        {
                j = partition(A, low, high);
                QuickSort(A, low, j-1);
                QuickSort(A, j+1, high);
        }
}
```

```
Algorithm partition(A, low, high)
{
        i=low,   j=high,   pivot=low
        while i < j do
        {
                while A[i]<=A[pivot] do
                        i++;
                while(A[j]>A[pivot])
                        j--;
                if i<j then
                        Swap A[i] and A[j]
        }
        Swap A[j] and A[pivot]
        return j
}
```