# ICT1008 Data Structures and Algorithms

## Lecture 5: Sorting Algorithms

# Agenda

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

# Recommended Readings

1. Runestone Interactive book: “Problem Solving with Algorithms and Data Structures Using Python”
   – Section: “Sorting & Searching”
2. Algorithms by Robert Sedgewick and Kevin Wayne. Addison-Wesley Professional. 4th edition, 2011.
   – Chapter 2: “Sorting”

# Bubble Sort

1. Step through the list to be sorted.
2. Compare two adjacent items at a time
   until you reach the end of the list;
   swap the two adjacent items
   if they are in the wrong order.
3. Repeat from the beginning of the list
   until no swaps are needed.

# Bubble Sort Algorithm

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

# Bubble Sort Analysis

- Regardless of how the items are arranged in the initial list, (n−1) passes will be made to sort a list of size n.
- Total number of comparisons
  = 1 + 2 + 3 + … + (n−1) $= \frac{1}{2}n^2 - \frac{1}{2}n$.
- Complexity is O($n^2$) comparisons.
- In the best case, if the list is already ordered,
  no exchanges will be made.
- In the worst case, every comparison will cause an exchange.
- On average, bubble sort exchanges half of the time.

# Agenda

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

# Selection Sort

The array is divided into two parts:
sorted & unsorted.
Initially, the sorted part is empty.

1. Find the minimum value in the list.
2. Swap it with the value in the first position of the array. The sorted part grows from here.
3. Find the next minimum value in the list.
4. Swap it with the value in the (second) next position of the array.
5. Repeat steps 3 & 4 till the end of the list.

https://youtu.be/Ns4TPTC8whw

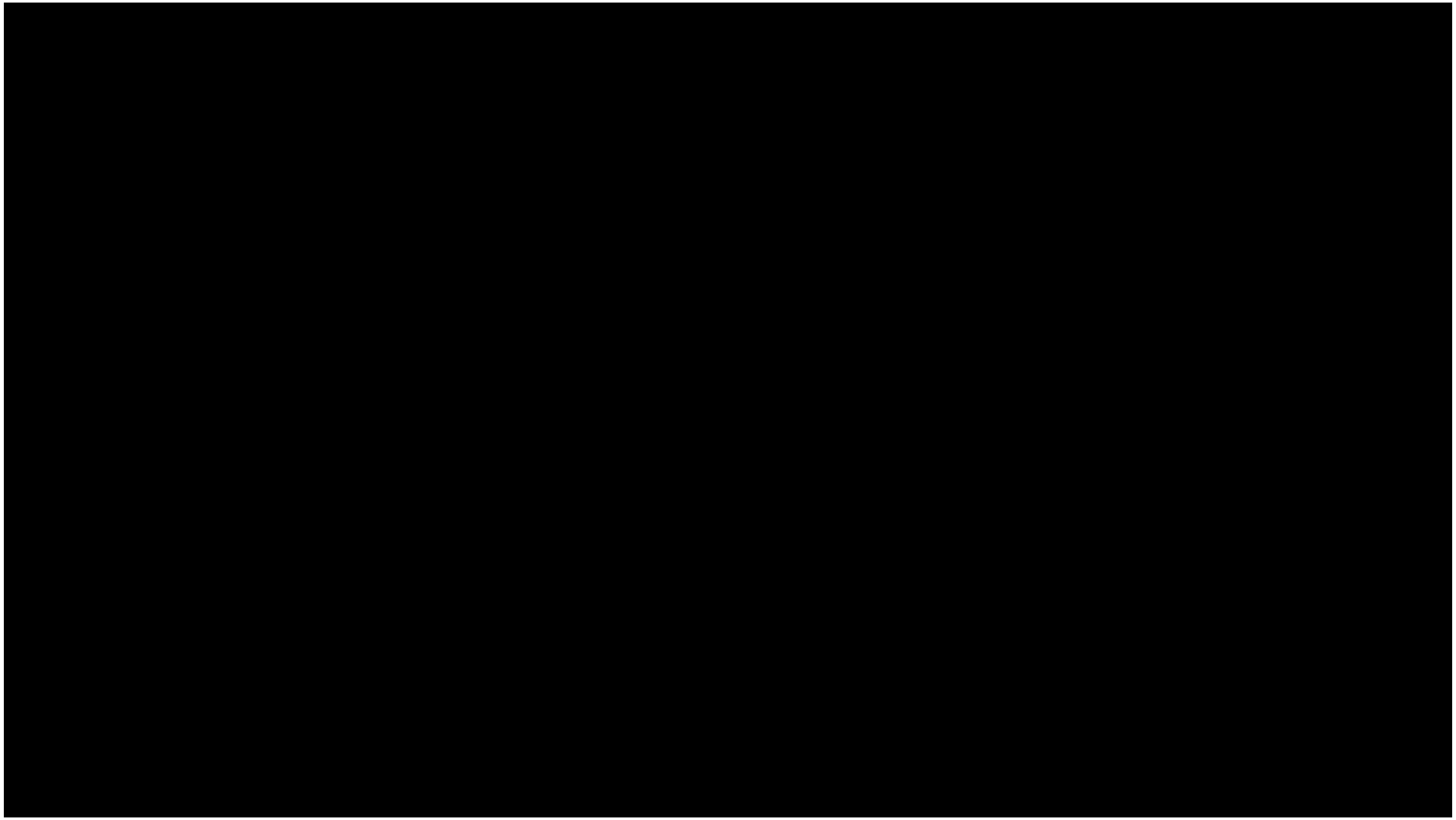
# Selection Sort Algorithm

```
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

# Selection Sort Analysis

- Selection sort makes the same number of comparisons as bubble sort.
- Complexity is therefore also O($n^2$).
- However, due to the reduction in the number of exchanges, selection sort typically executes faster than bubble sort in benchmark studies.

# Selection Sort Demo



https://youtu.be/Ns4TPTC8whw

# Agenda

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

# Insertion Sort

The array is divided into two parts:
sorted & unsorted.
Initially, the sorted part is empty.

1. Every iteration of insertion sort removes an element (normally the first one) from the input data, and inserts it into the correct position in the already-sorted list.
2. Repeat step 1 until no input element remains.

https://youtu.be/ROalU379l3U

# Insertion Sort Algorithm

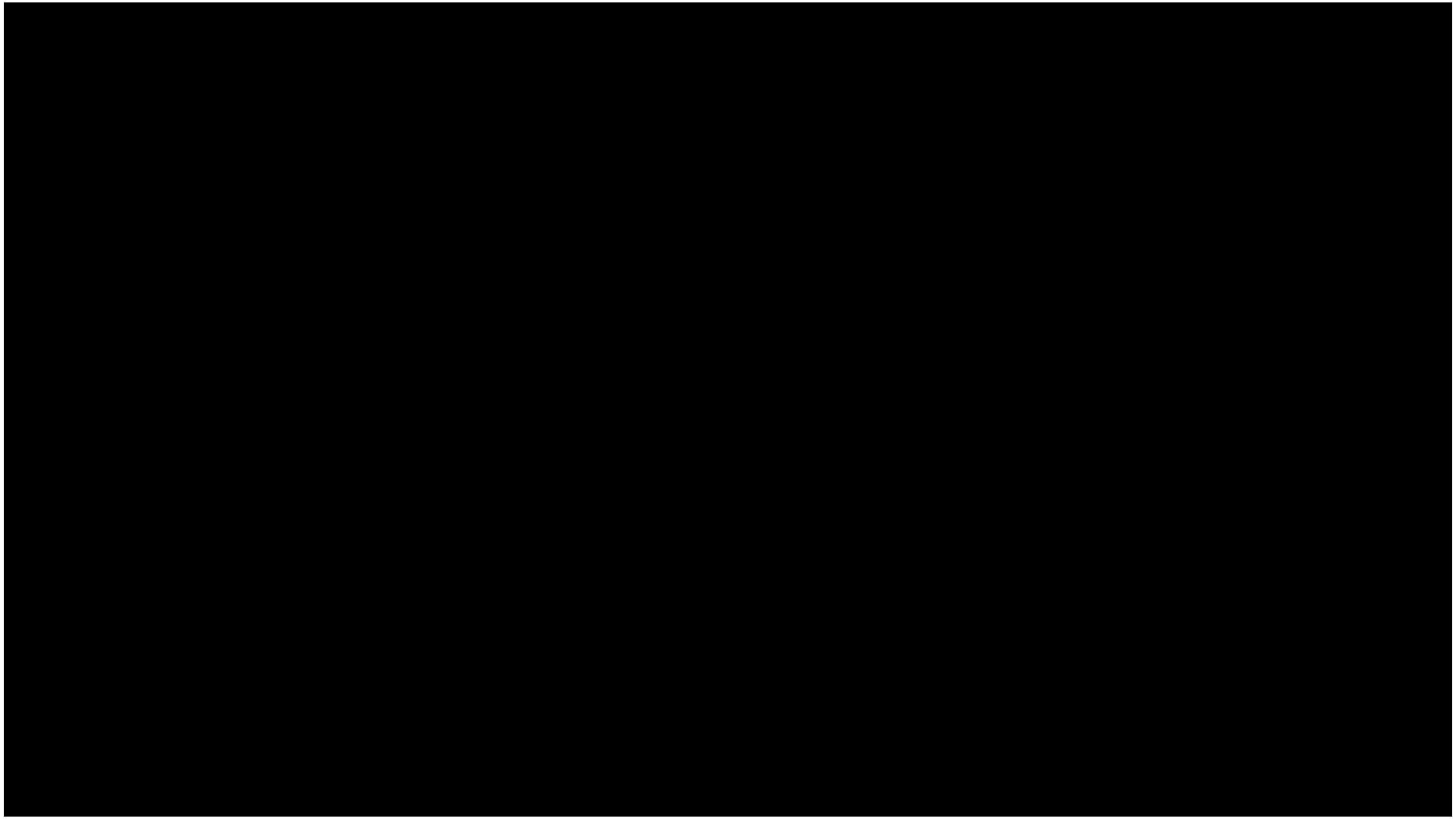
```python
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position>0 and alist[position-1]>currentvalue:
         alist[position]=alist[position-1]
         position = position-1

     alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
insertionSort(alist)
print(alist)
```

# Insertion Sort Analysis

- Insertion Sort uses (n-1) passes to sort n items.
- Maximum number of comparisons is $O(n^2)$ complexity:

  = 1 + 2 + 3 + … + (n−1) $= \frac{1}{2}n^2 - \frac{1}{2}n.$
- Best case: only one comparison needs to be done on each pass for an already sorted list.

# Insertion Sort Demo



https://youtu.be/ROalU379l3U

# Agenda

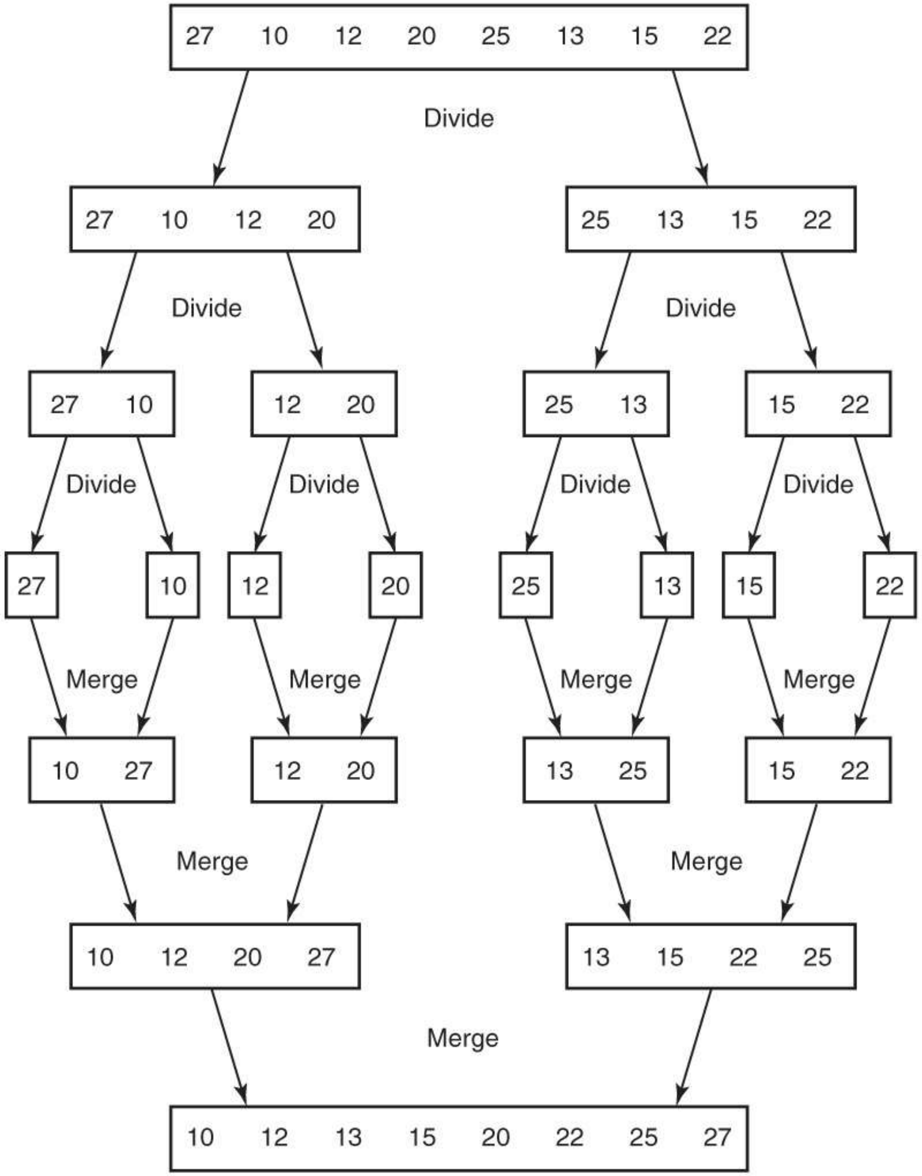
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

# Merge Sort

Method: Divide & conquer

1. Divide the unsorted list into two nearly equal size sub-lists.
2. Sort each sub-list recursively by applying merge sort.
3. Merge the two sub-lists back into one sorted list.

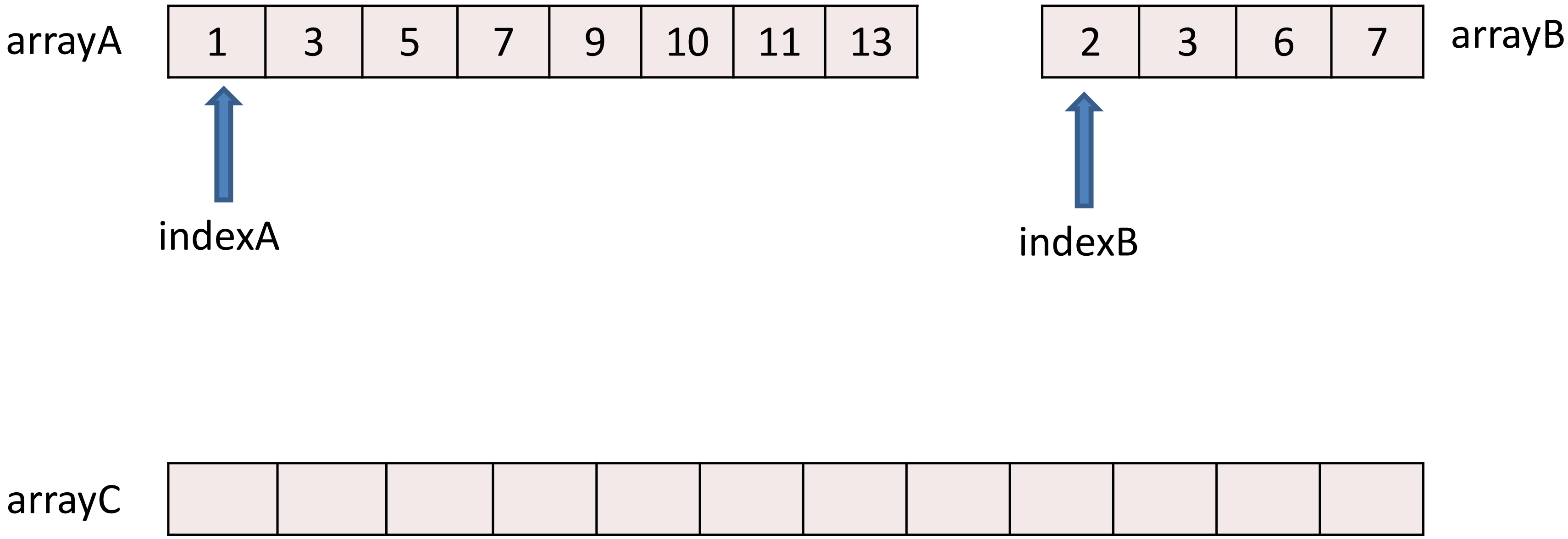
# Merge Sort: Example Lecture 5 slide 9



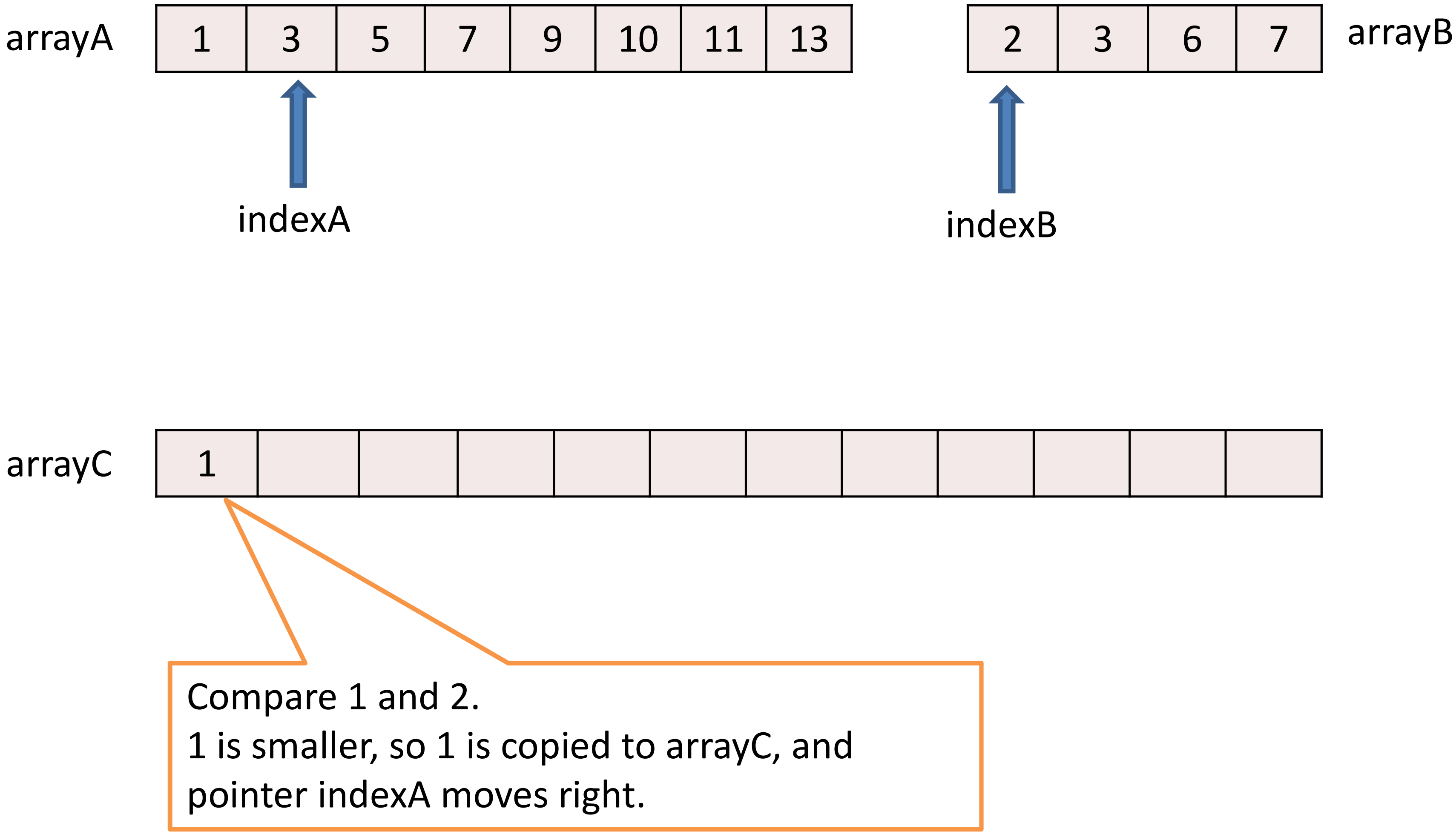
# Merge algorithm

```python
def merge(arrayA, arrayB):
    arrayC = []
    sizeA = len(arrayA)
    sizeB = len(arrayB)

    indexA = indexB = 0
    while indexA < sizeA and indexB < sizeB:
        if arrayA[indexA] < arrayB[indexB]:
            arrayC.append(arrayA[indexA])
            indexA = indexA+1
        else:
            arrayC.append(arrayB[indexB])
            indexB = indexB+1

    for i in range(indexA,sizeA):
        arrayC.append(arrayA[i])

    for i in range(indexB, sizeB):
        arrayC.append(arrayB[i])
    return arrayC
```

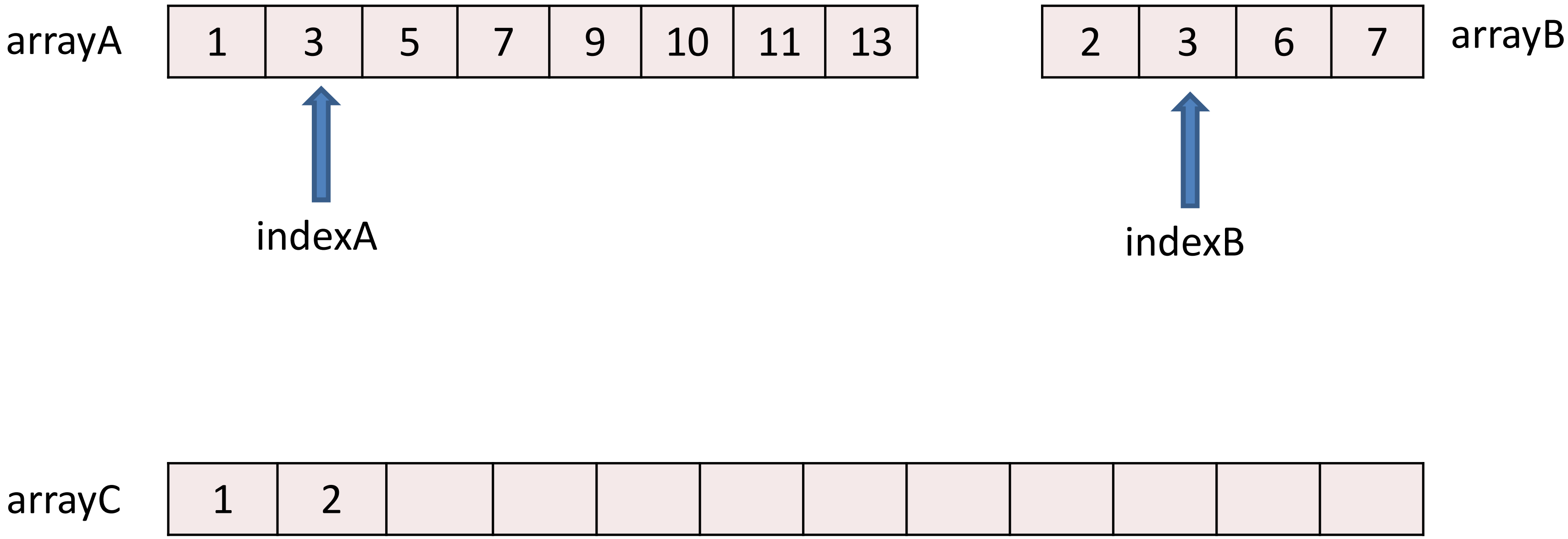
# Merge Algorithm Demo



# Merge Algorithm Demo

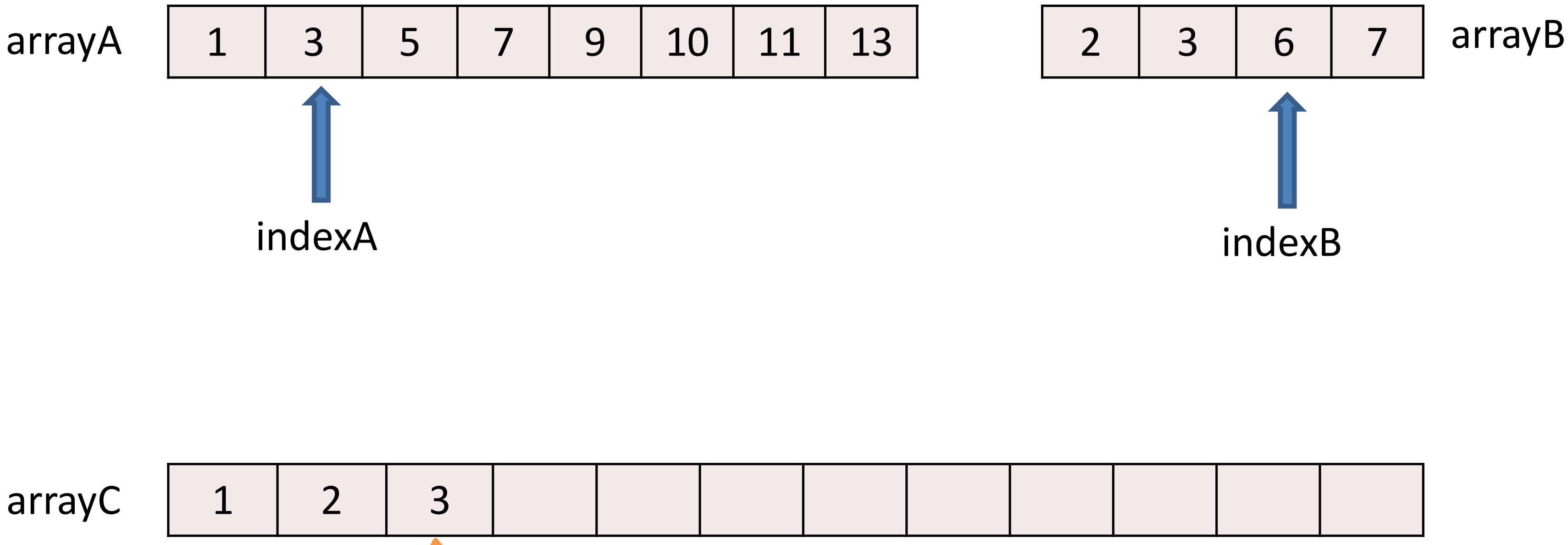


# Merge Algorithm Demo



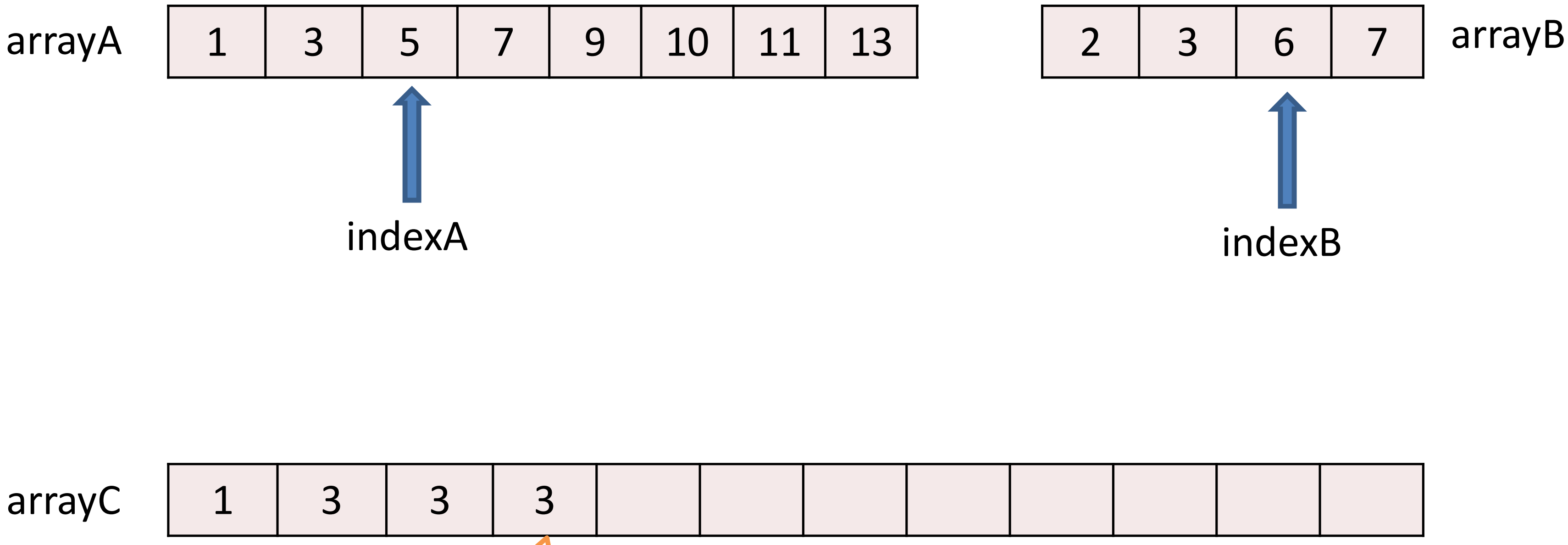

Compare 3 and 2.
2 is smaller, so 2 is copied to arrayC, and pointer indexB moves right.

# Merge Algorithm Demo




Compare 3 and 3.
3 in arrayA is not smaller than 3 in arrayB,
so the 3 in arrayB is copied to arrayC, and
pointer indexB moves right.

# Merge Algorithm Demo




Compare 3 and 6.
3 is smaller than 6, so the 3 is copied to arrayC, and pointer indexA moves right.

# Merge Algorithm Demo

arrayA

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 7 | 9 | 10 | 11 | 13 |

indexA

arrayB

| | | | |
|---|---|---|---|
| 2 | 3 | 6 | 7 |

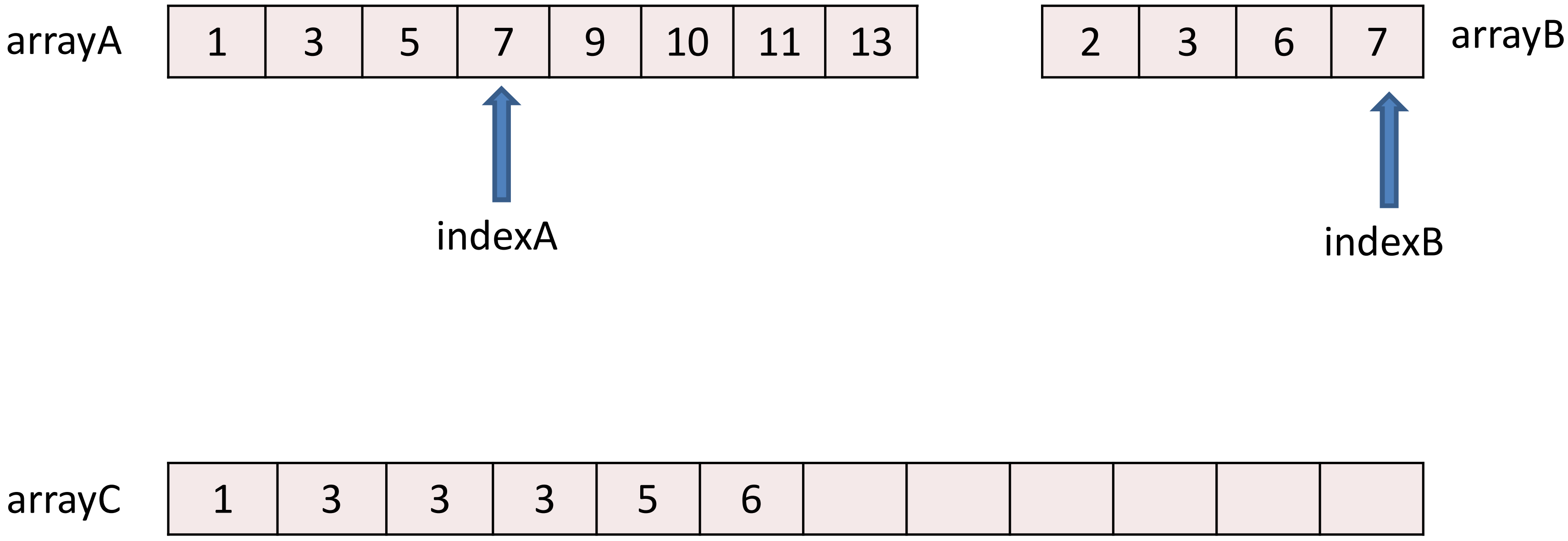
indexB

arrayC

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 3 | 3 | 5 | | | | | | | |

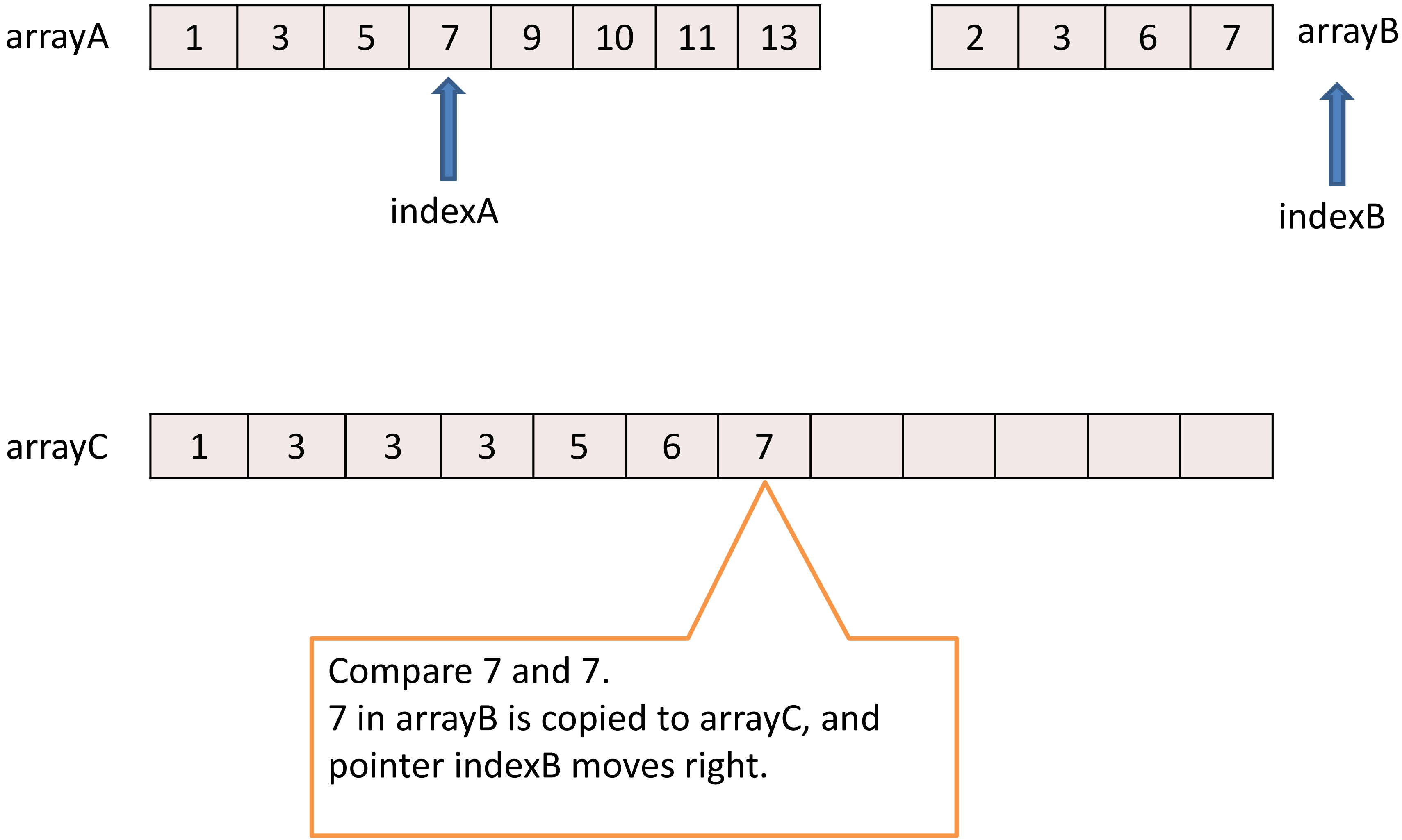
Compare 5 and 6.
5 is smaller than 6, so 5 is copied to arrayC, and pointer indexA moves right.

# Merge Algorithm Demo



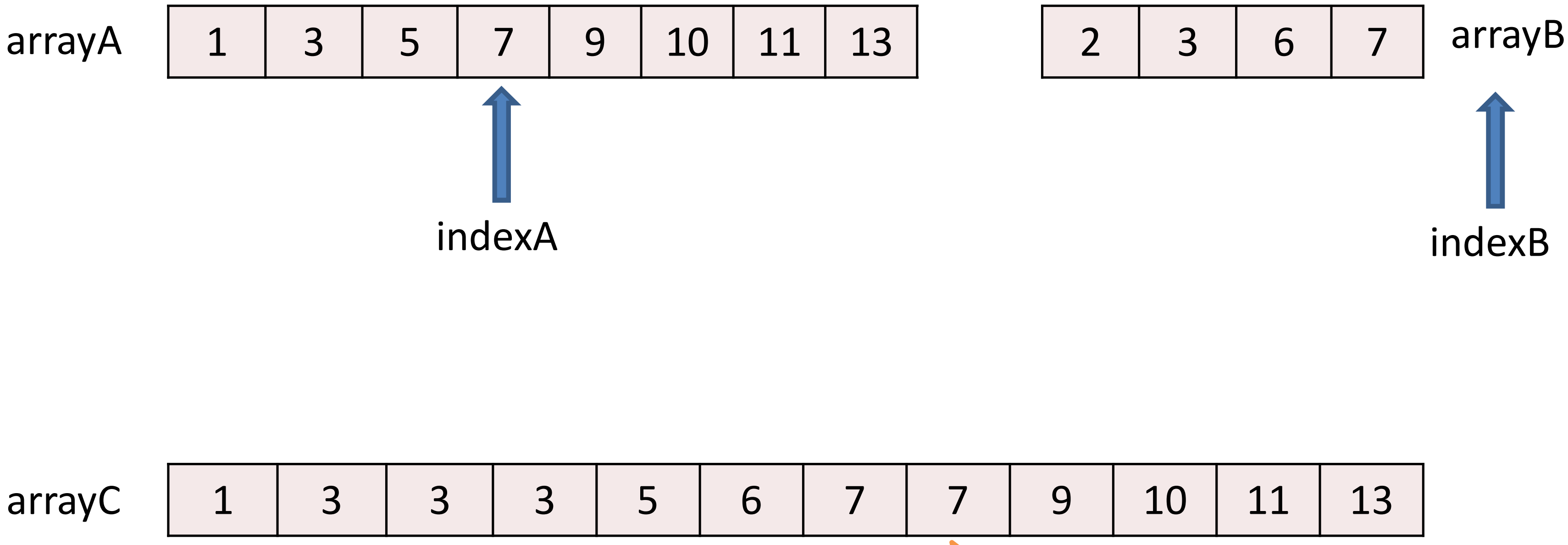

Compare 7 and 6.
6 is smaller than 7, so 6 is copied to arrayC, and pointer indexB moves right.

# Merge Algorithm Demo



# Merge Algorithm Demo

SINGAPORE INSTITUTE OF TECHNOLOGY




The remaining elements of arrayA are copied into arrayC.

SIT Internal

# Merge Sort algorithm

```python
def mergeSort(array):
    size = len(array)

    if size is 1:
        return array

    midIndex = size/2
    firstHalf = array[0:midIndex]
    secondHalf = array[midIndex:size]

    firstHalf = mergeSort(firstHalf)
    secondHalf = mergeSort(secondHalf)
    array = merge(firstHalf, secondHalf)

    return array

print mergeSort([27,10,12,20,25,13,15,22])
```

Base case:
When the array has one element, it is already sorted.
So return the array for merging.

Divide the array into two almost equal halves: firstHalf and secondHalf.

Recursively MergeSort divide firstHalf and divide secondHalf.

Merge the sorted firstHalf and secondHalf.

Return the sorted array.

# Merge Sort – Complexity

- Time complexity
  - $merge$ is $O(n)$.
  - $merge$ is called $O(\log n)$ times recursively.
  - $mergeSort$ is $O(n \log n)$.
- Space complexity
  - $merge$ uses an additional $arrayC$.
  - If $arrayC$ was local inside $merge$, much more storage would be used because of recursive calls.
  - Consider using a global $arrayC$ in the implementation.

# Agenda

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

# Quick Sort

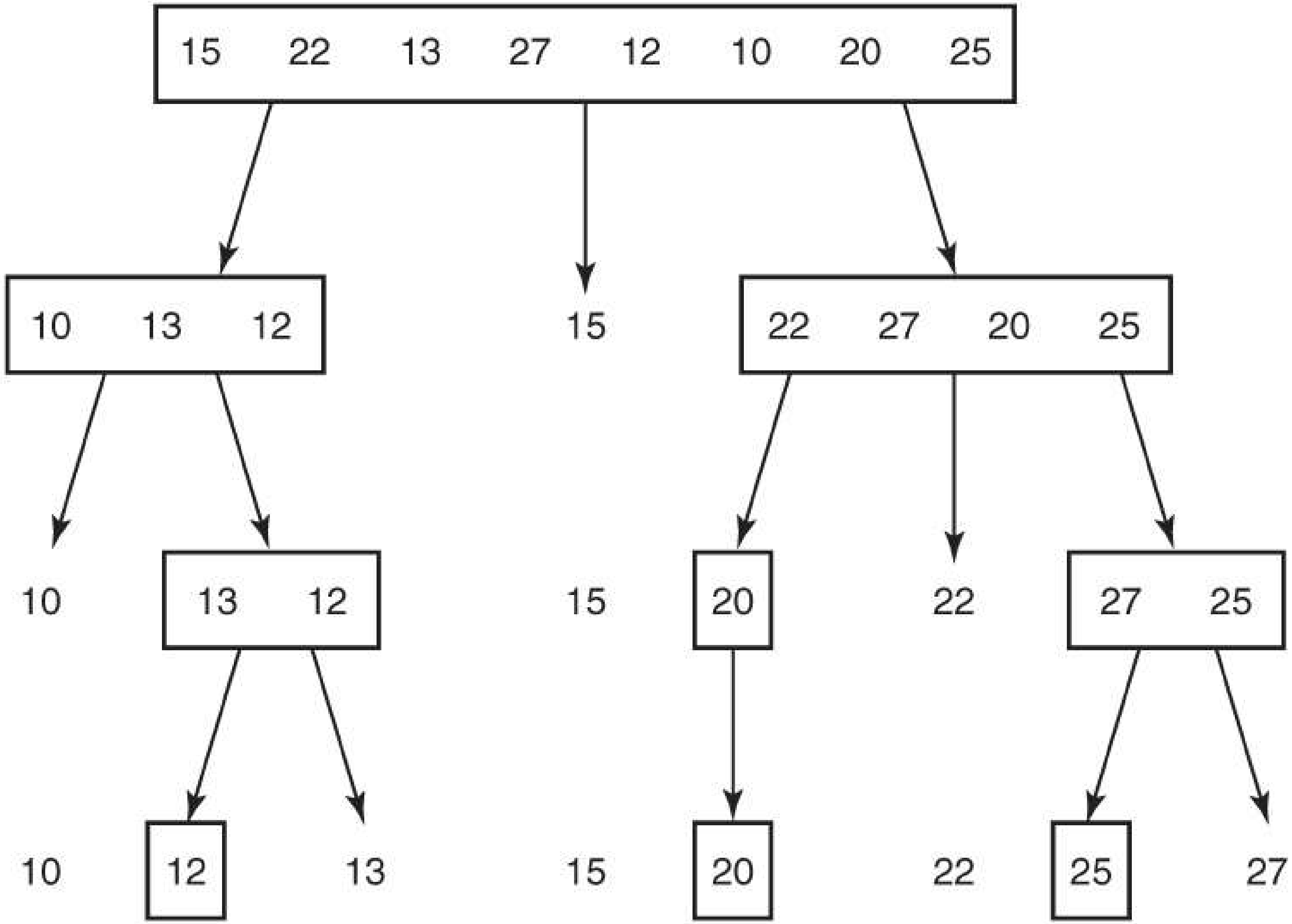
Method: Divide-and-conquer.

- Pick an element ($pivot$) from the list.
  - $pivot$ is arbitrarily chosen.
  - Normally, the first element is selected.
- Partition the list into two halves such that:
  - All the elements in the first half are smaller than the pivot.
  - All the elements in the second half are greater than or equal to the pivot.

| | | |
|---|---|---|
| $1^{st}$ half | pivot | $2^{nd}$ half |

- Quick-sort the $1^{st}$ half.
- Quick-sort the $2^{nd}$ half.

# Quick Sort: Example Lecture 5 slide 10



# Quick Sort algorithm

```
def quickSort(a):
    size = len(a)
    if size > 1:
        pivotIndex = partition(a)
        a[0:pivotIndex] = quickSort(a[0:pivotIndex])
        a[pivotIndex+1:size] = quickSort(a[pivotIndex+1:size])
    return a
```

3. Recursively Quicksort the right half.

2. Recursively Quicksort the left half.

1. Partition the array.

# Partition Algorithm

3. Move the pivot into place.

2. Move element into left half.

1. Element at index is smaller than pivot
=> Belongs to left half. Increment pivotIndex
to increase size of left half by 1.

```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

# Partition Algorithm

```python
def quickSort(a):
    size = len(a)
    if size > 1:
        pivotIndex = partition(a)
        a[0:pivotIndex] = quickSort(a[0:pivotIndex])
        a[pivotIndex+1:size] = quickSort(a[pivotIndex+1:size])
    return a
```

```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

# Quick Sort: Partition Demo

pivotIndex

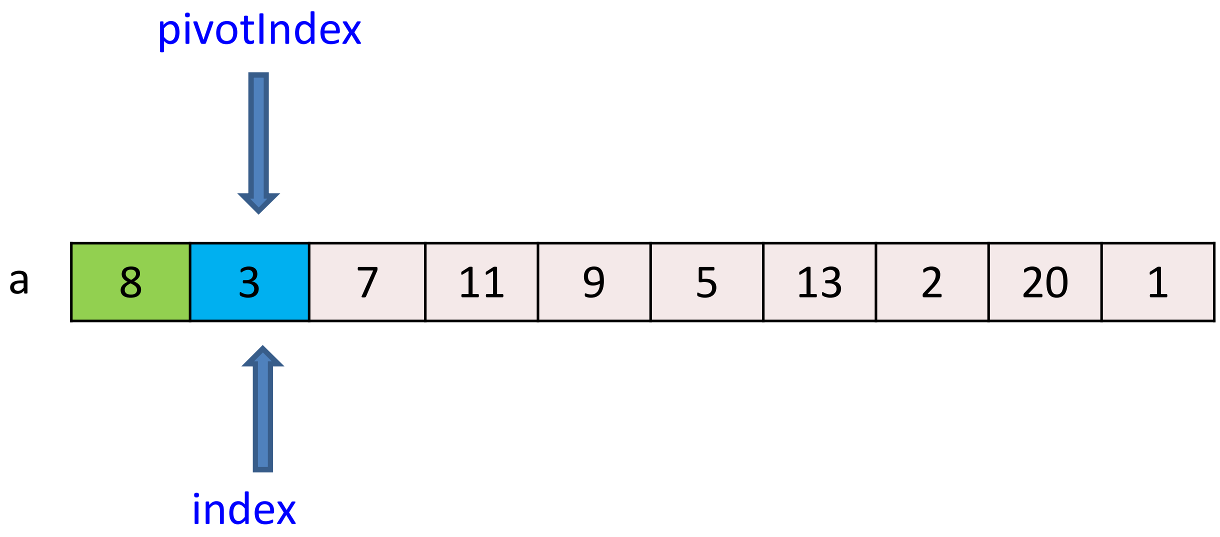
| a | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 8 | 3 | 7 | 11 | 9 | 5 | 13 | 2 | 20 | 1 |

index

```
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

pivot = a[0] = 8
pivotIndex = 0
index = 1

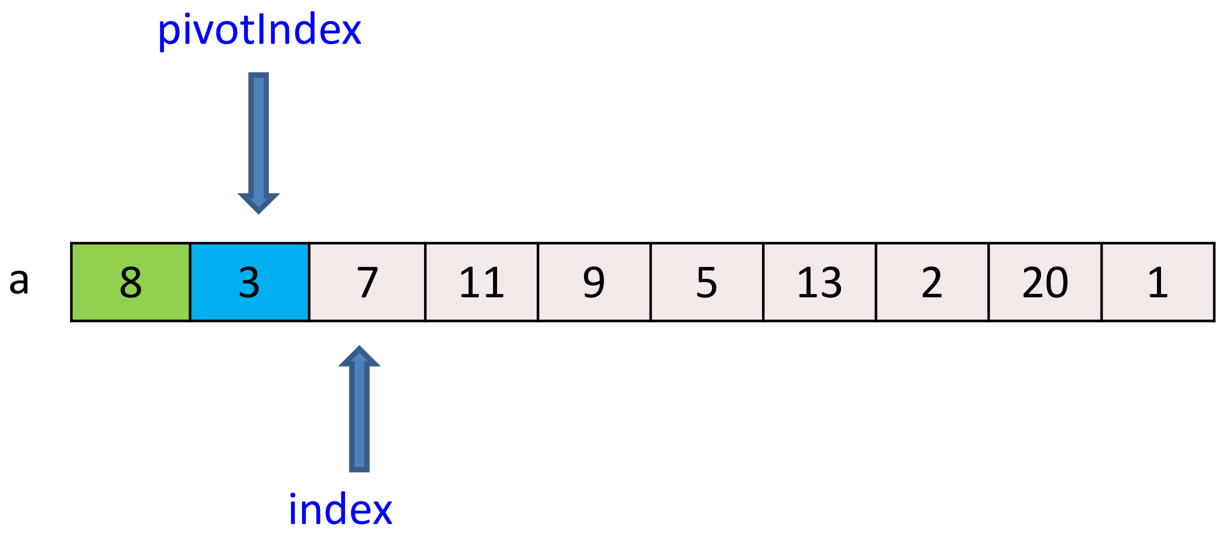
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

3 < 8.
pivotIndex shift right.
Swap pivotIndex with index => no change to array.

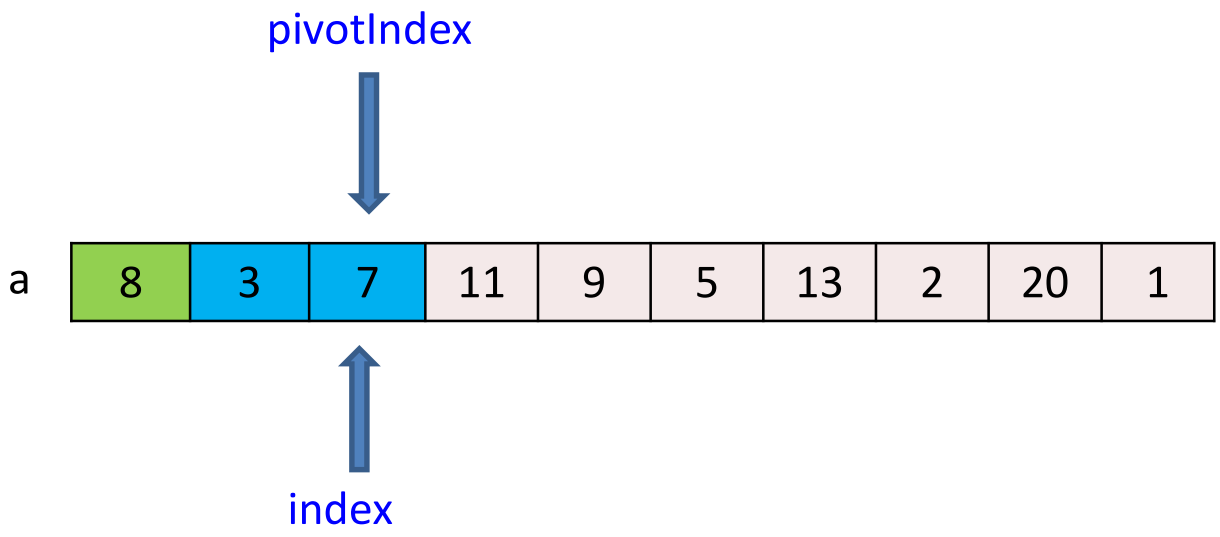
# Quick Sort: Partition Demo




```
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.

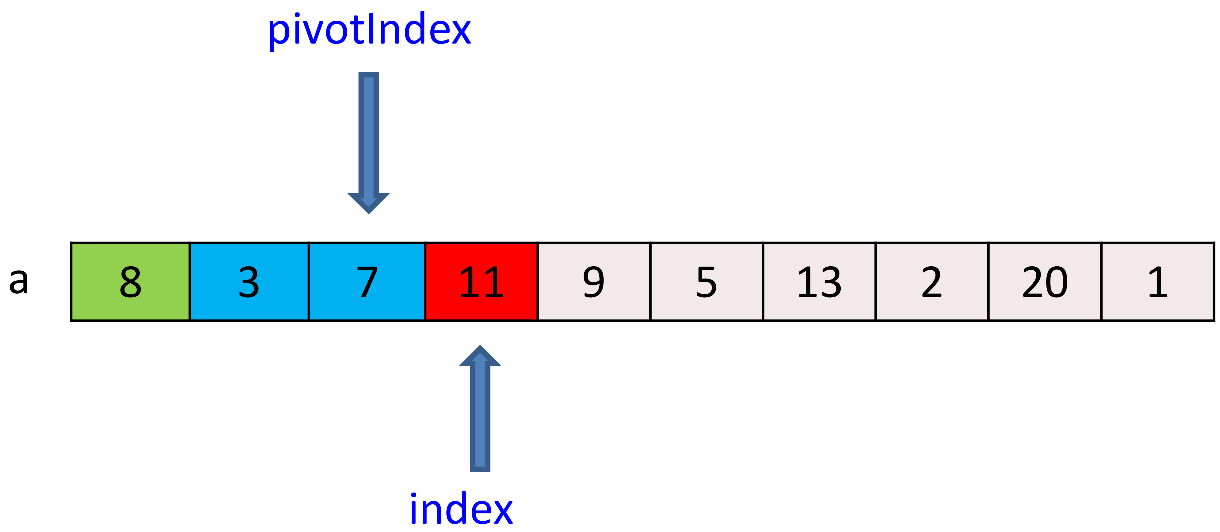
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

7 < 8.
pivotIndex shift right.
Swap pivotIndex with index => no change to array.

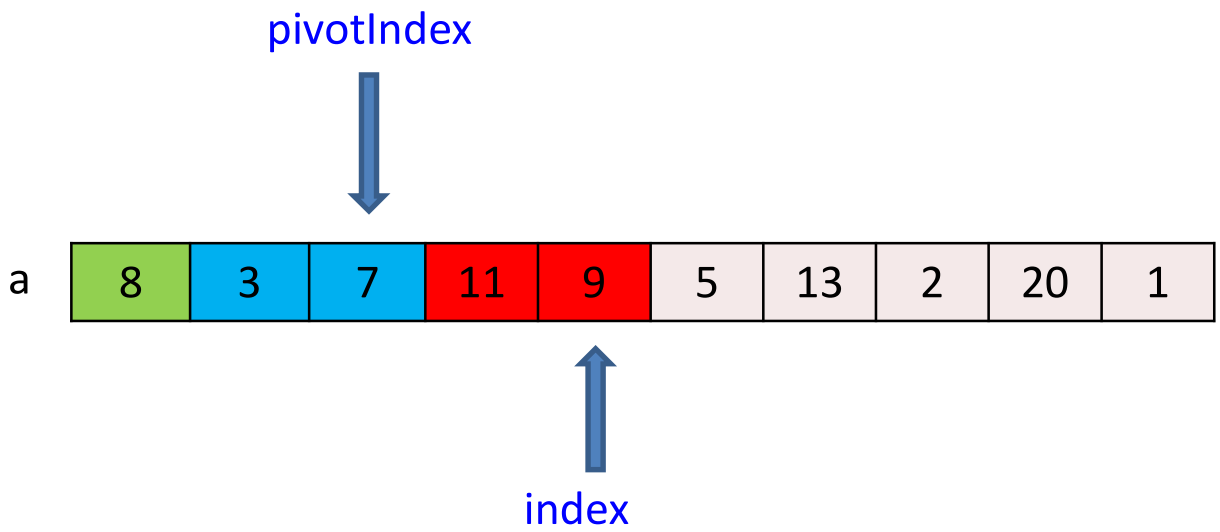
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.
11>8, do nothing.

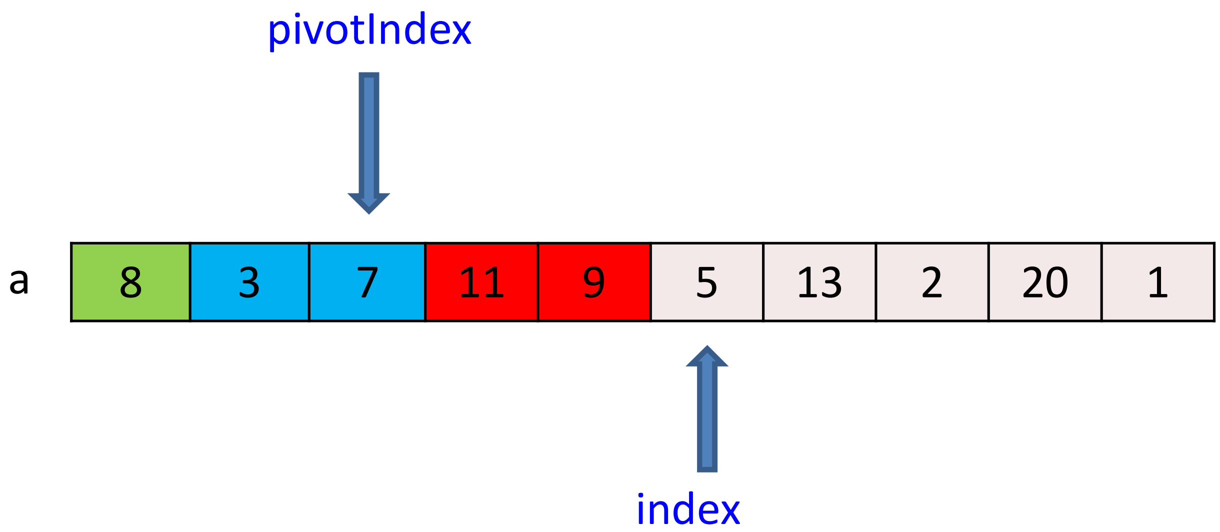
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.
9>8, do nothing.

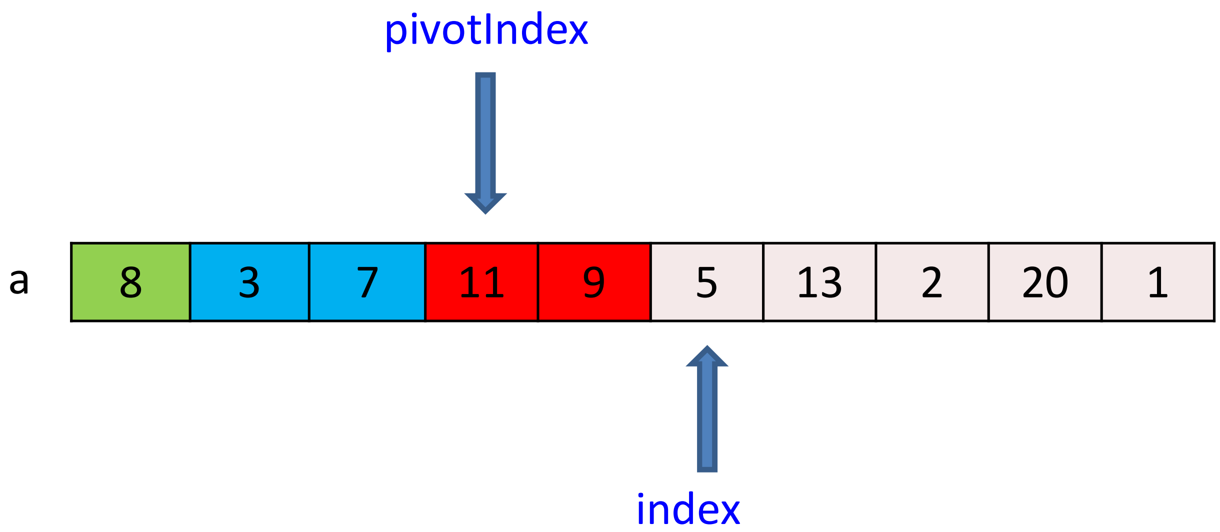
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.

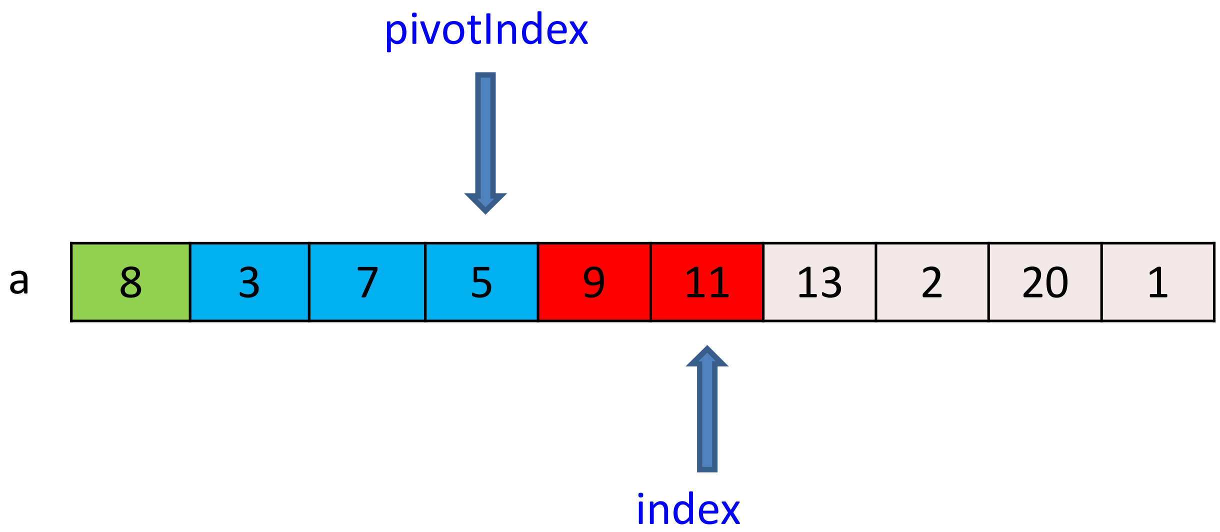
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

5<8.
pivotIndex shift right.

# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

Swap 5 and 11.

# Quick Sort: Partition Demo

pivotIndex

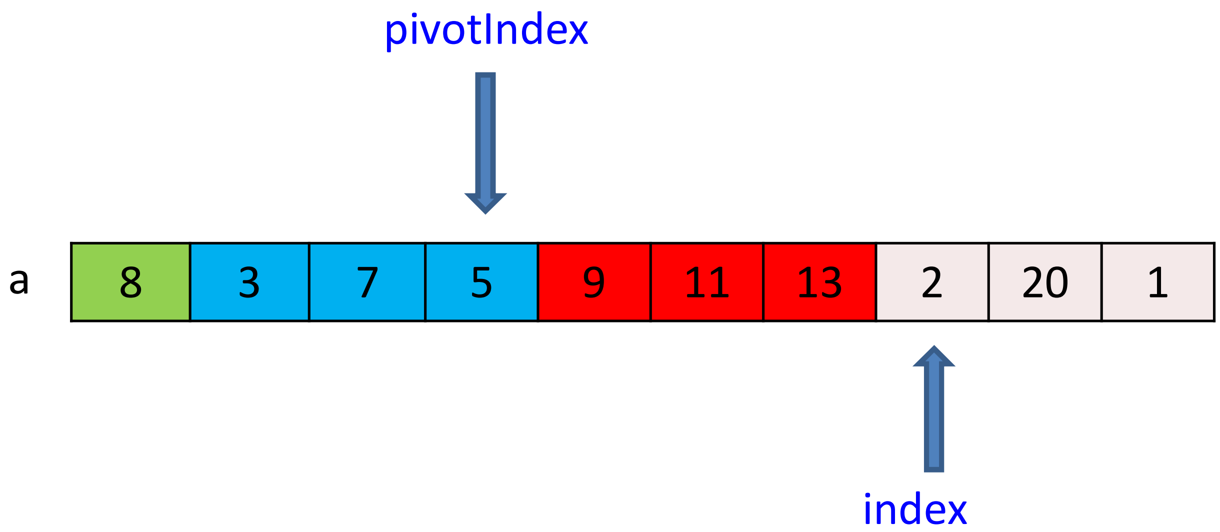
| a | 8 | 3 | 7 | 5 | 9 | 11 | 13 | 2 | 20 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|

index

```
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.
13>8, do nothing.

# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.

# Quick Sort: Partition Demo

pivotIndex

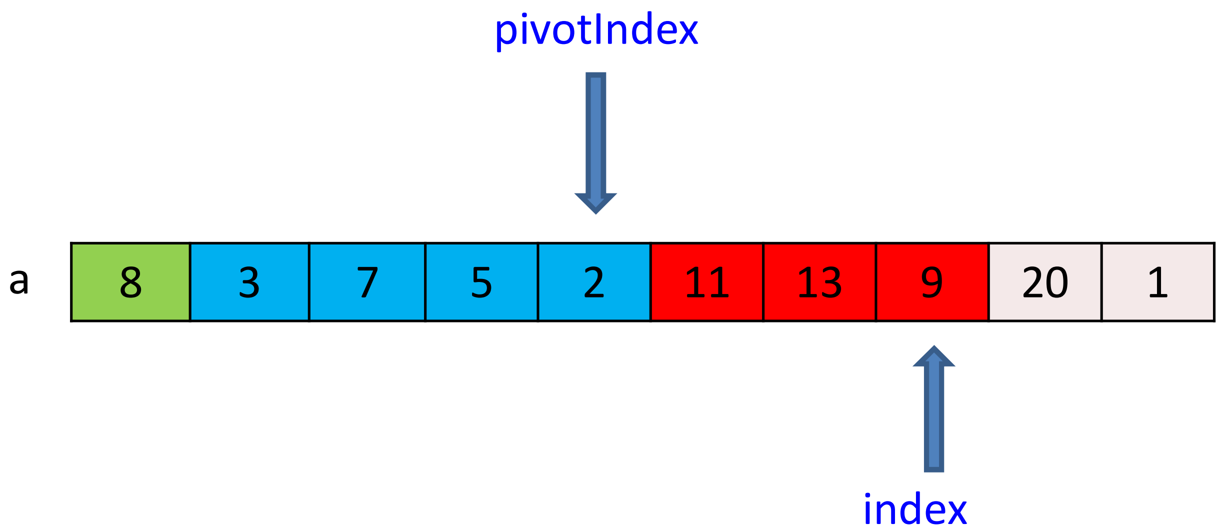
| a | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| a | 8 | 3 | 7 | 5 | 9 | 11 | 13 | 2 | 20 | 1 |

index

```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

2<8.
pivotIndex shift right.

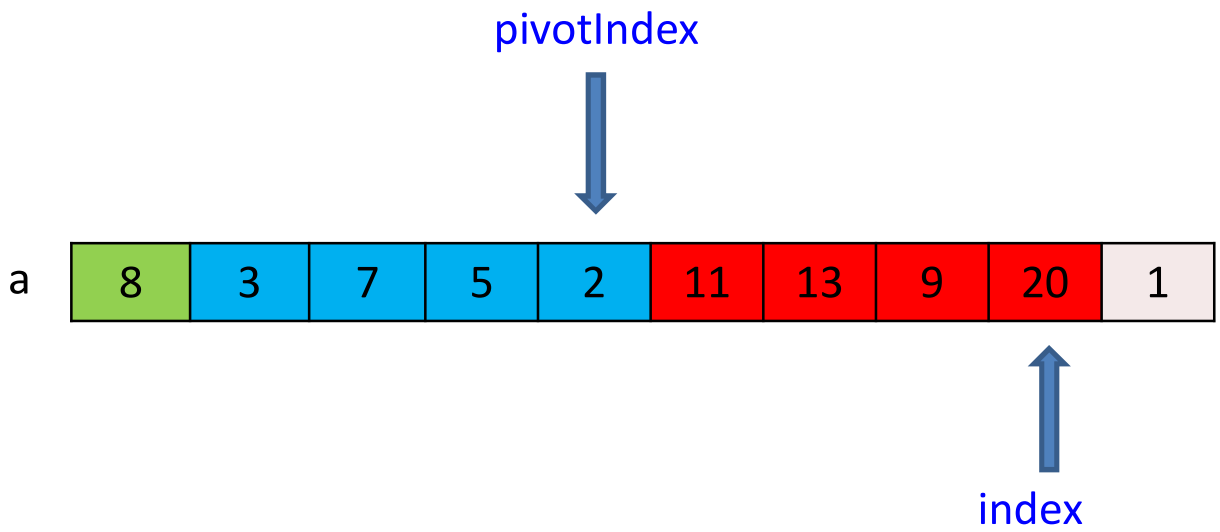
# Quick Sort: Partition Demo




```
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

Swap 2 and 9.

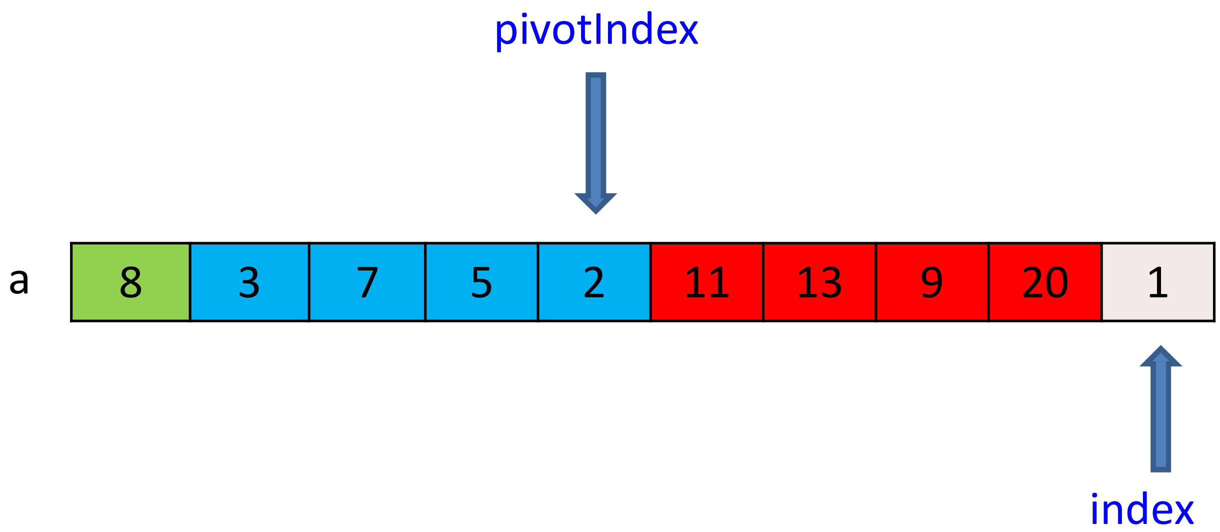
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.
20>8, do nothing.

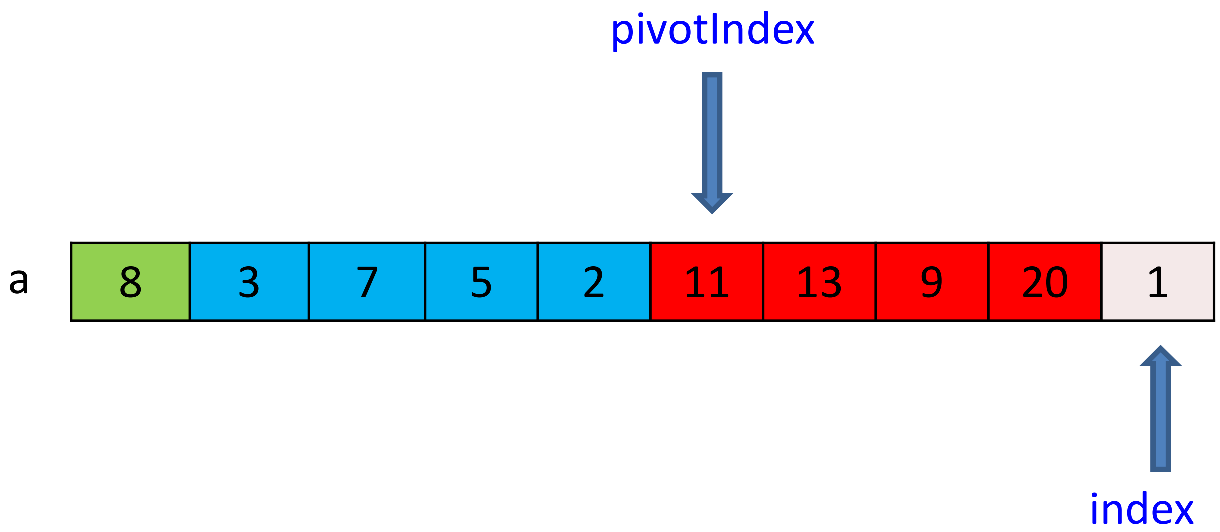
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

index shift right.

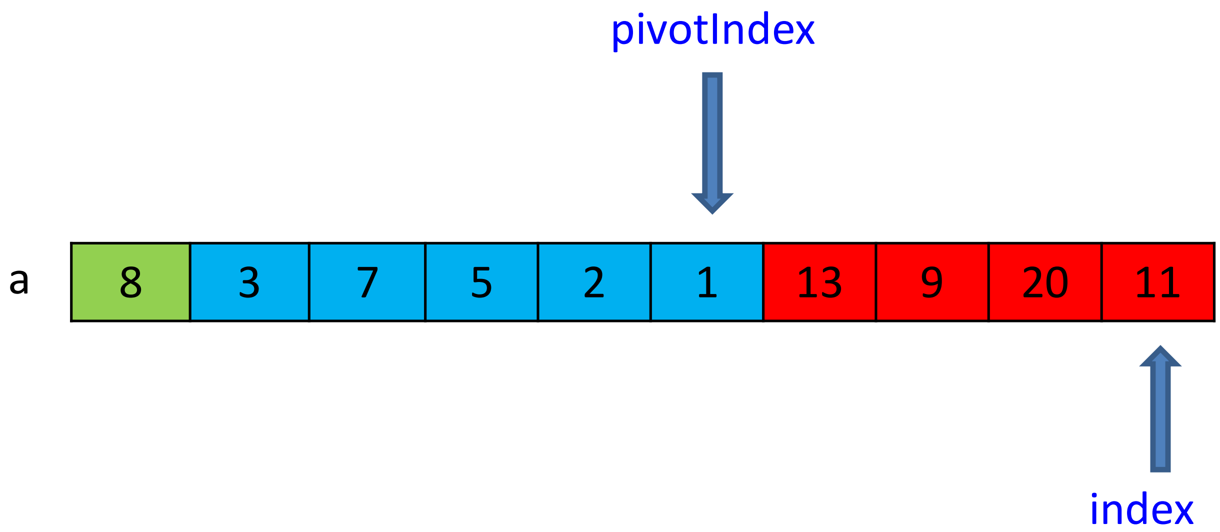
# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

1<8.
pivotIndex shift right.

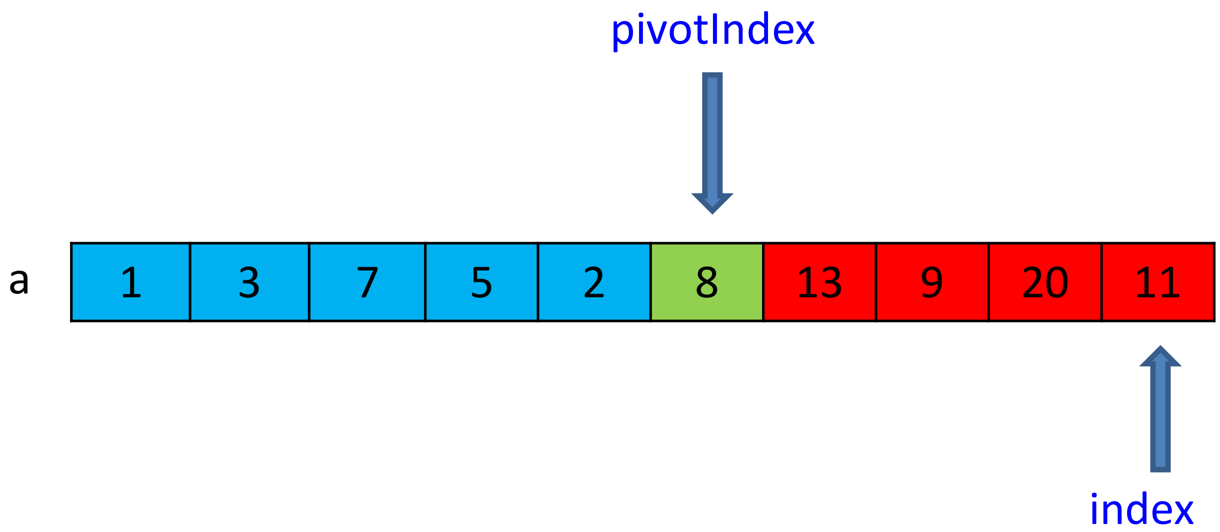
# Quick Sort: Partition Demo




```
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

Swap 11 and 1.

# Quick Sort: Partition Demo




```python
def partition(a):
    pivot = a[0]
    size = len(a)
    pivotIndex = 0
    for index in range(1,size):
        if a[index] < pivot:
            pivotIndex+=1
            a[index],a[pivotIndex] = a[pivotIndex], a[index]
    a[0],a[pivotIndex] = a[pivotIndex], a[0]
    return pivotIndex
```

Swap 8 and 1.

# Quick Sort - Complexity

**Time complexity**

- On average, each partition halves the size of the array to be sorted.
- On average, each partition swaps half the elements.
- On average, algorithm is $O(n \log n)$.
- Worst case, algorithm is $O\left(n^2\right)$.

# Quick Sort – Choice of Pivot

- In this version of quicksort, the leftmost element of the partition is used as the pivot element.
- Unfortunately, this causes worst-case behavior on already sorted arrays because the size of sub-array is only reduced by 1.

- This problem is easily solved by choosing:
  1. a random index for the pivot, or
  2. the middle index of the partition for the pivot, or
  3. the median of the first, middle and last elements of the partition for the pivot.