SORTING

B.Vidyullatha
Lecturer in Computer Science
D.K.Govt. College for Women(A),Nellore

Outline

- Introduction
- Sorting Techniques
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort

Introduction

- Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order.
- Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion.
- The records are sorted either numerically or alphanumerically.
- Categories of Sorting:
  - Internal Sorting
  - External Sorting

Bubble Sort

- **bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Procedure:
  - scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order

Bubble Sort – Example

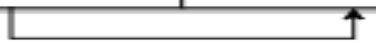
List is unsorted. We are going to sort this in ASCENDING order using Bubble Sort

5	4	3	2	1

First Pass of the first For loop.

1=0, a[j]=5, a[j+1] =4 (Second For loop starts working)

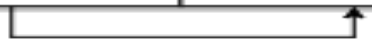
5	4	3	2	1



Here a[j]=5 and a[j+1]=4 are compared. Their positions are then interchanged.

Second For loop gets iterated. Now j=1 a[j]=5 and a[j+1]=3

4	5	3	2	1

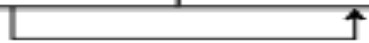


j=2 a[j]=5 and a[j+1]=2

4	3	5	2	1

j=3 a[j]=5 and a[j+1]=1

4	3	2	5	1



Largest element of the list is moved to last position after first pass.

4	3	2	1	5

Selection Sort

- **selection sort**: orders a list of values by repetitively putting a particular value into its final position

- Procedure:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Selection Sort – Example

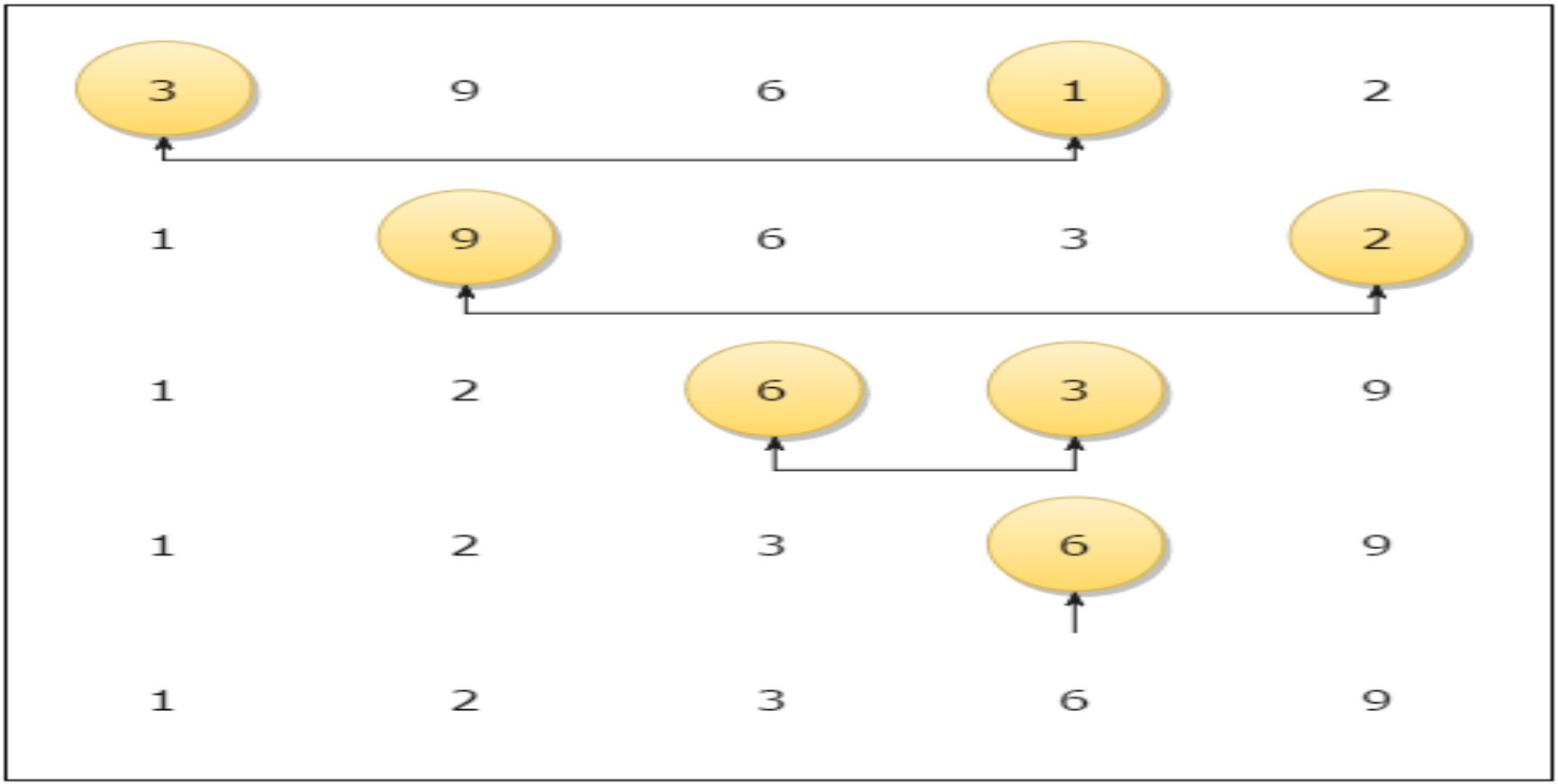


Fig. Selection Sort Technique

Insertion Sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- Procedure:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

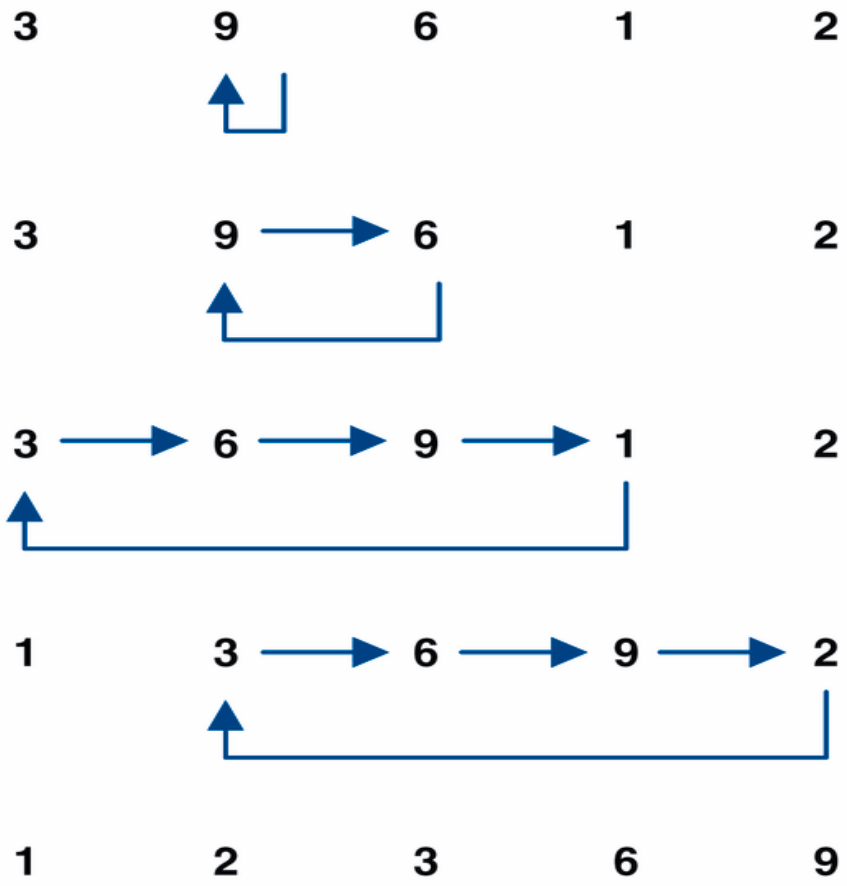
Insertion Sort – Example

3 is sorted.
Shift nothing. Insert 9.

3 and 9 are sorted.
Shift 9 to the right. Insert 6.

3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.

1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.



Merge Sort

- This technique is based on splitting a list, into two comparable sized lists, i.e., left and right and then sorting each list individually and then merging the two sorted lists back together as one.

Merge Sort – Example

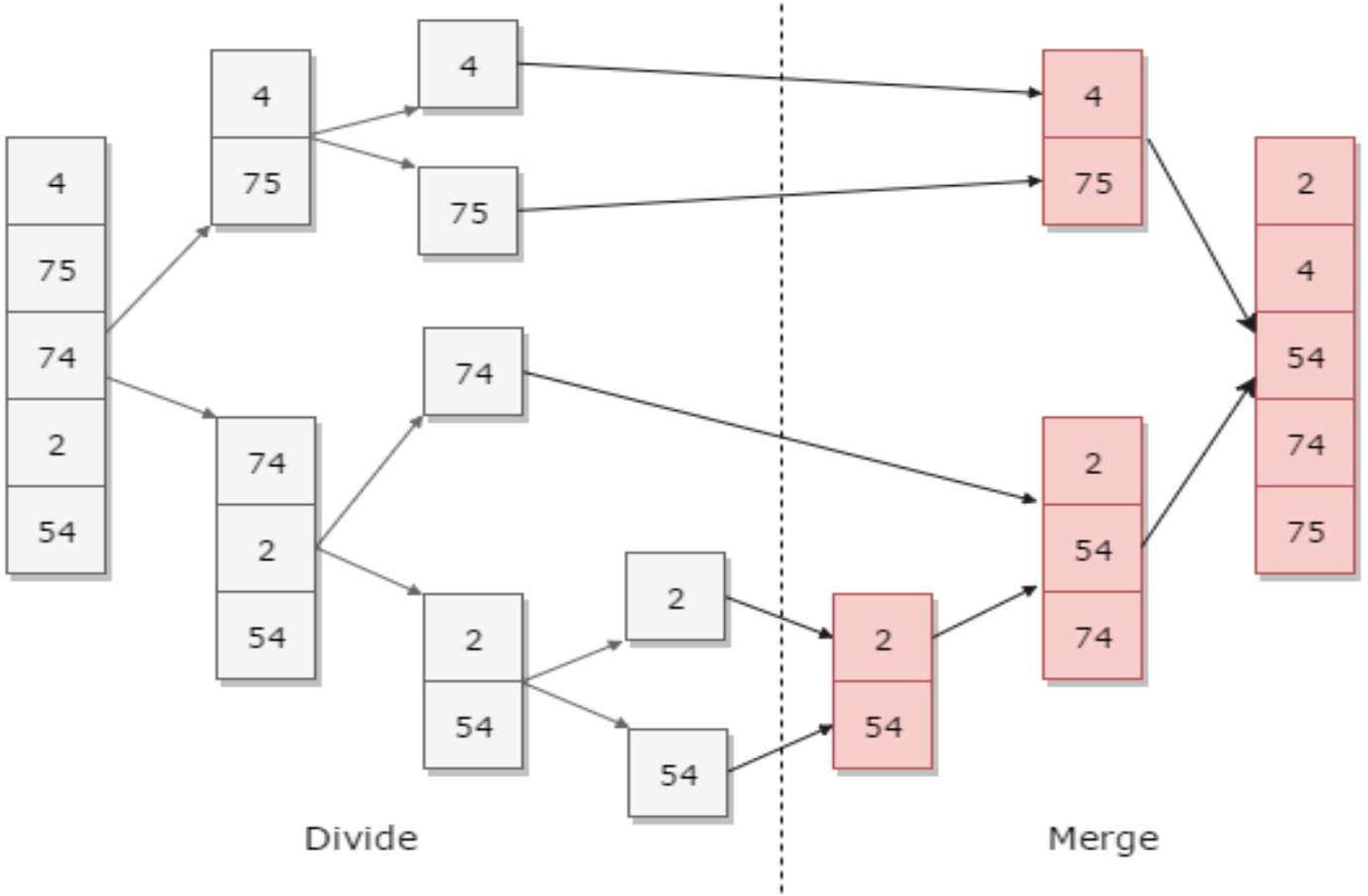


Fig: Merge Sort Technique

Quick Sort

- Quick sort is one of the most famous sorting techniques based on divide and conquer strategy.
- This technique starts by picking a single item which is called pivot and moving all smaller items before it, while all greater elements in the later portion of the list.
- This is the main quick sort operation named as a partition, recursively repeated on lesser and greater sublists until their size is one or zero – in which case the list is wholly sorted.

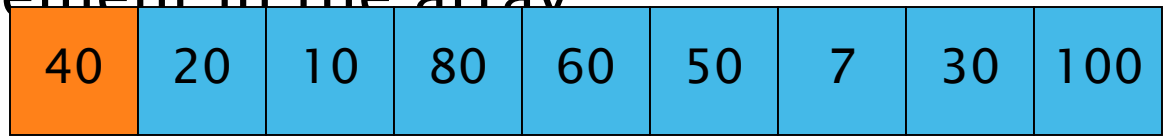
Quick Sort – Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:



Partitioning Array

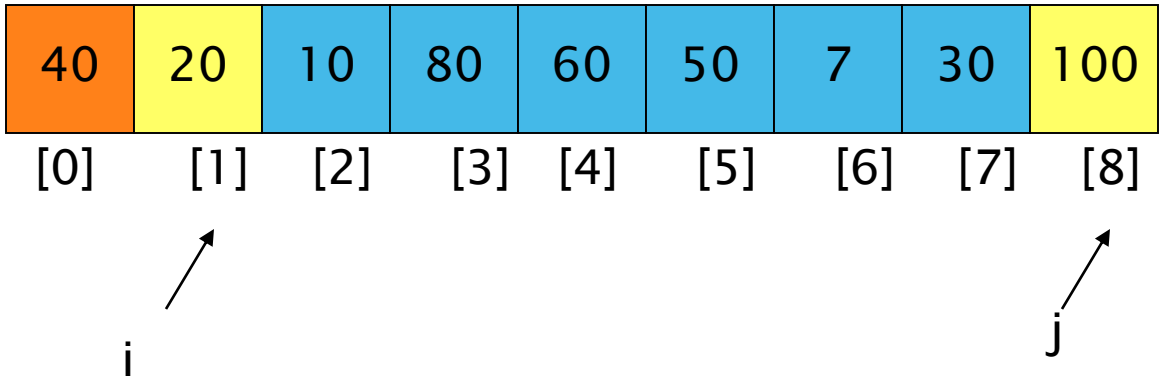
Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

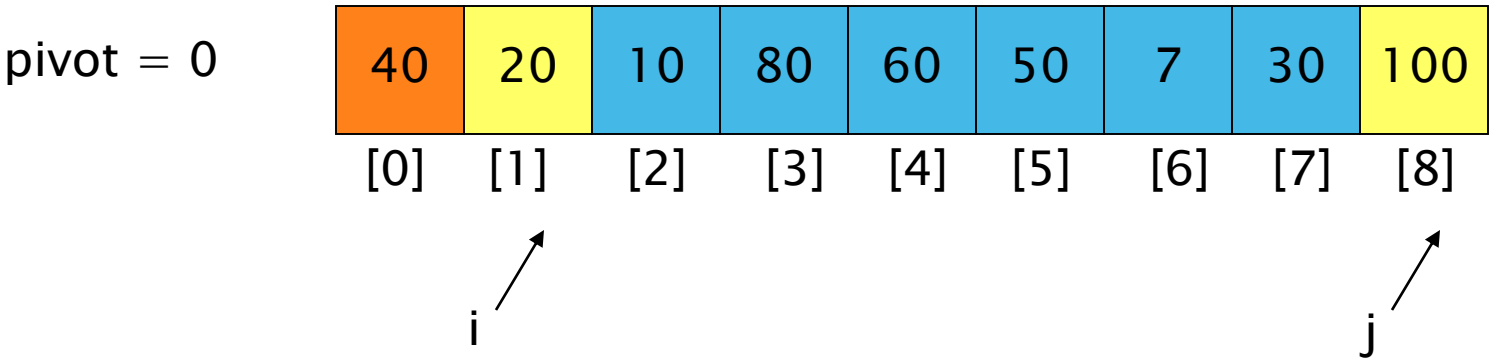
The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

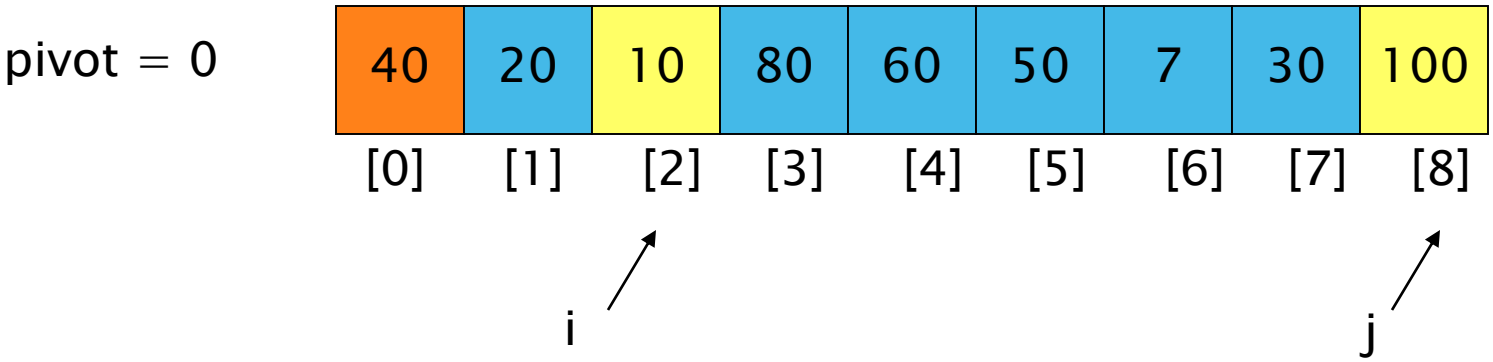
Pivot = 0



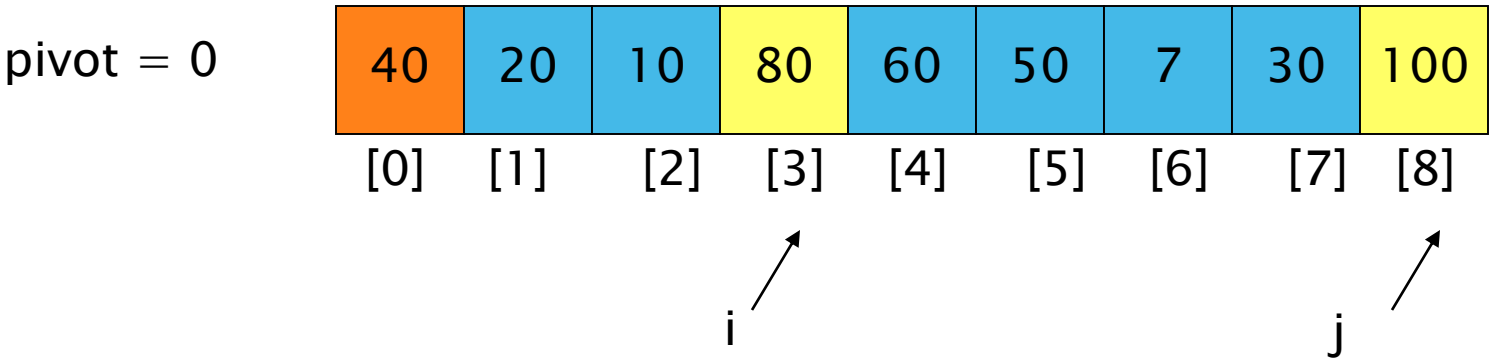
1. While a[i] <= a[pivot]
   ++i



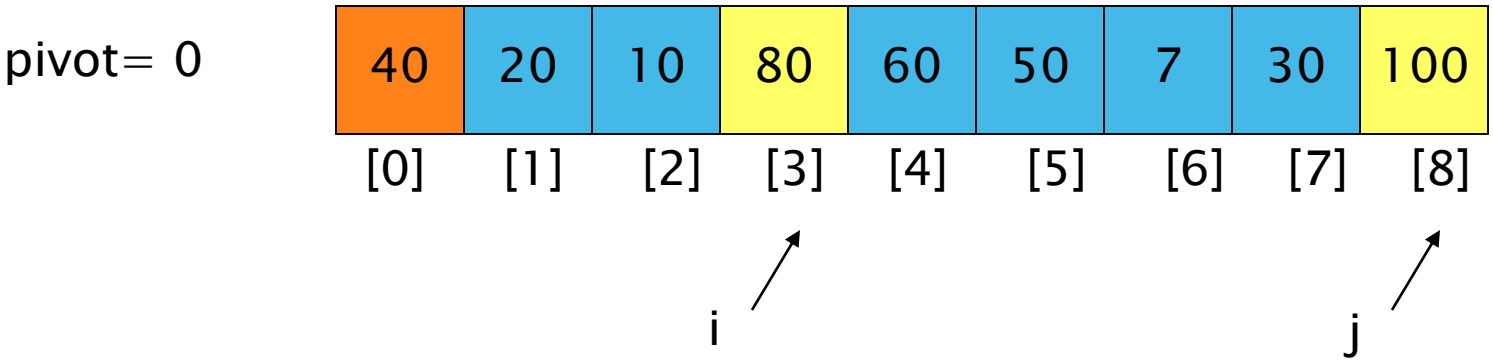
1. While a[i] <= a[pivot]
   ++i



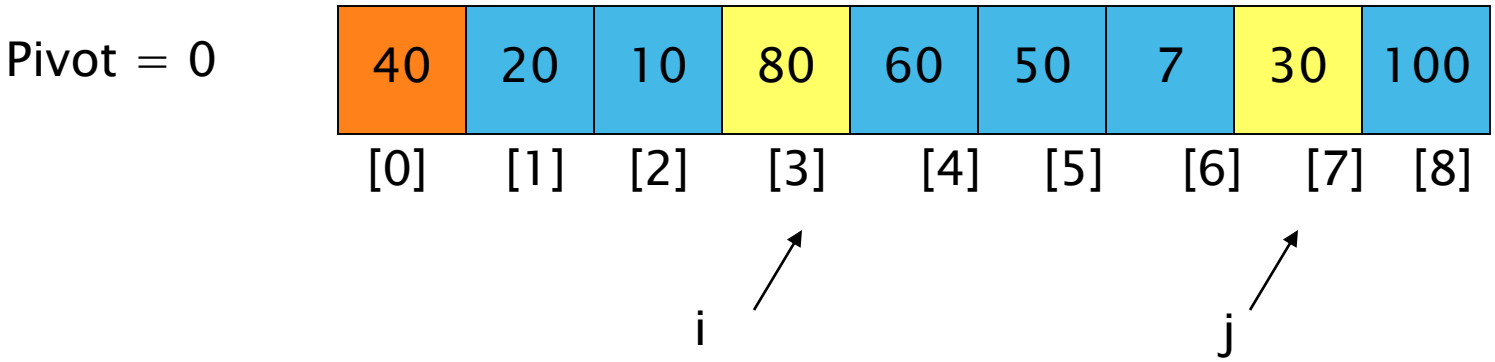
1. While a[i] <= a[pivot]
   ++i



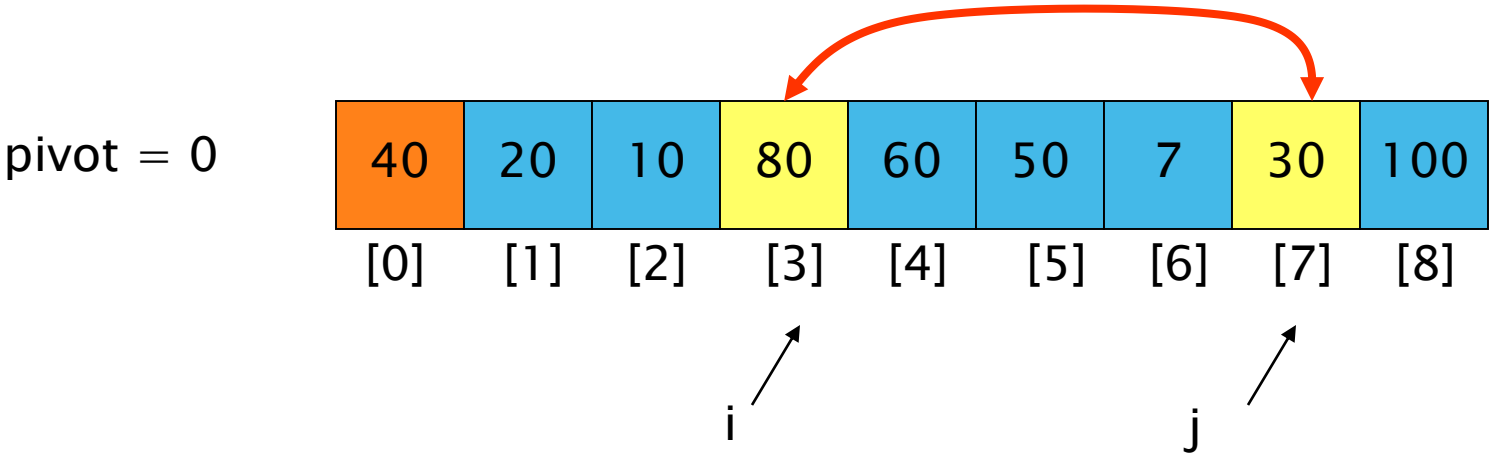
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j



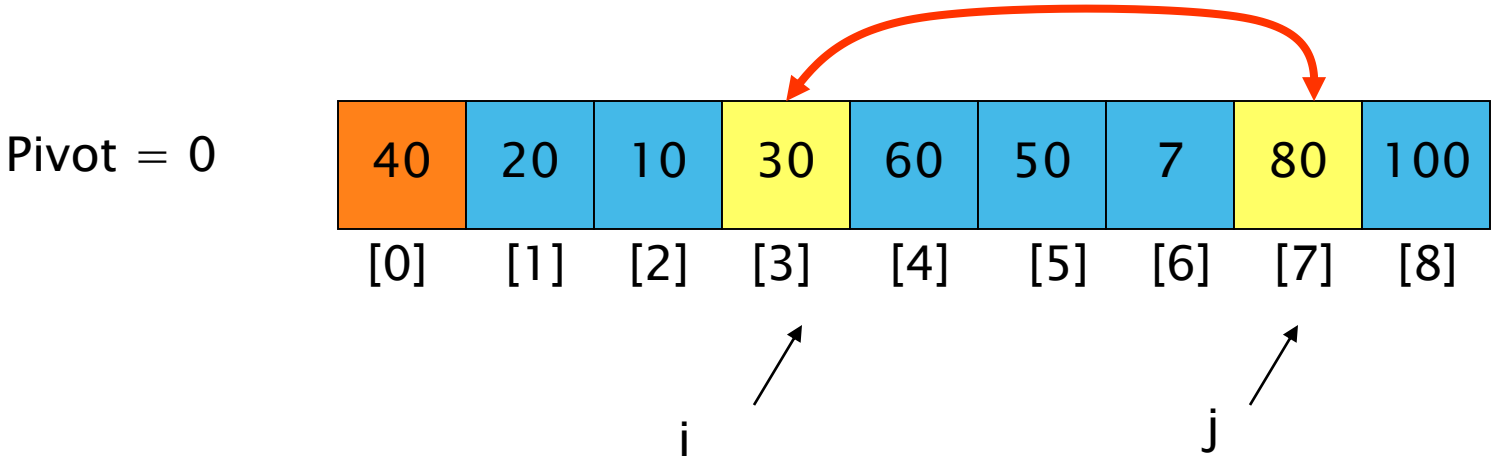
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j



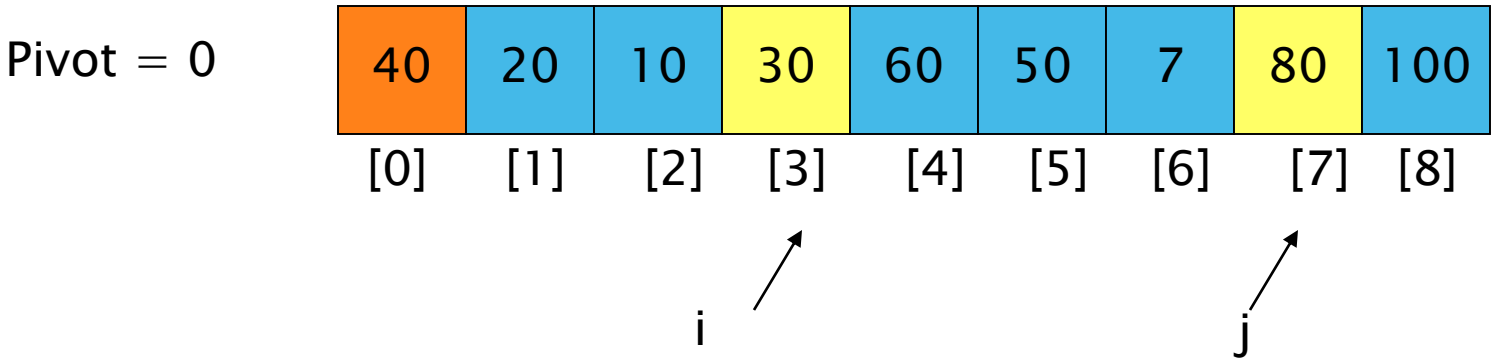
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]



1. While a[i] <= a[pivot]
   ++i
2. While a[j] > a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]

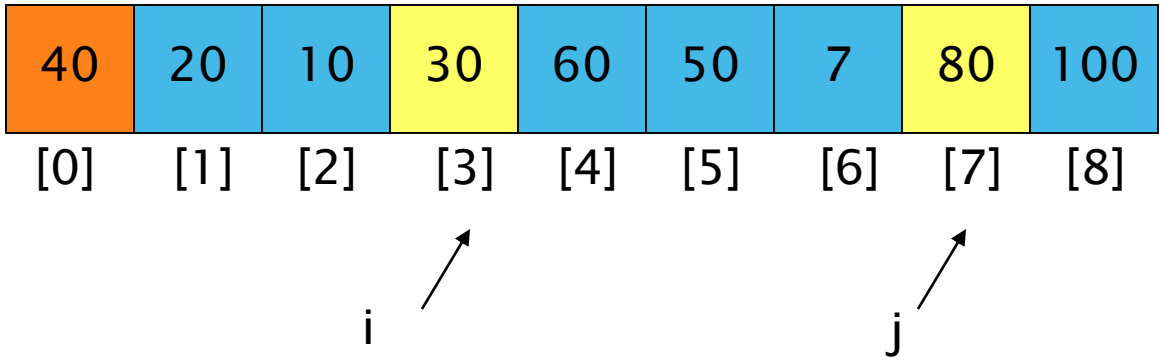


1. While a[i] <= a[pivot]
   ++i
2. While a[i] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.



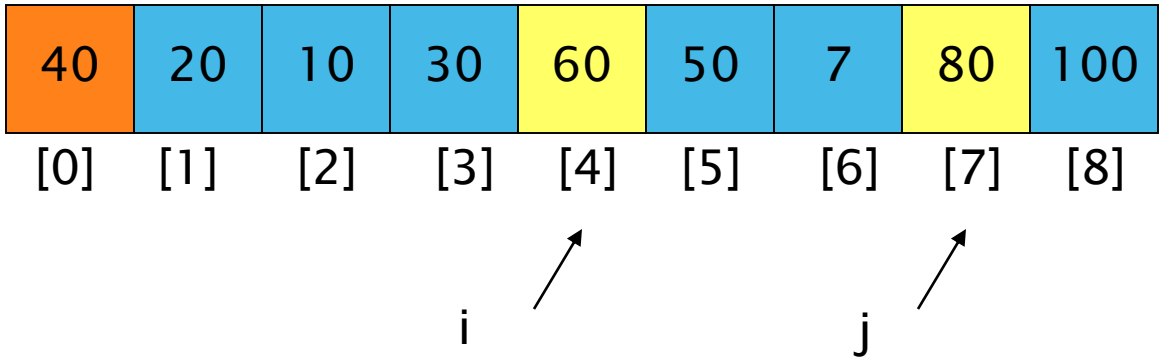
1. While a[i] <= a[pivot]
   ++i
2. While a[i] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0



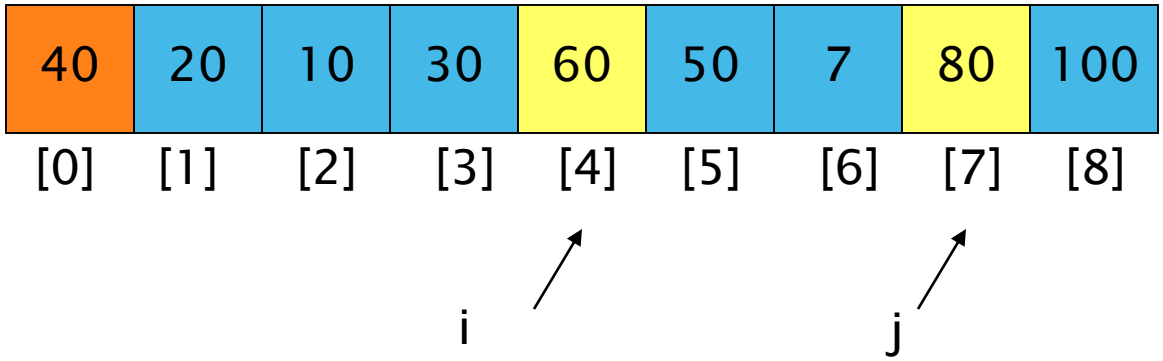
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0



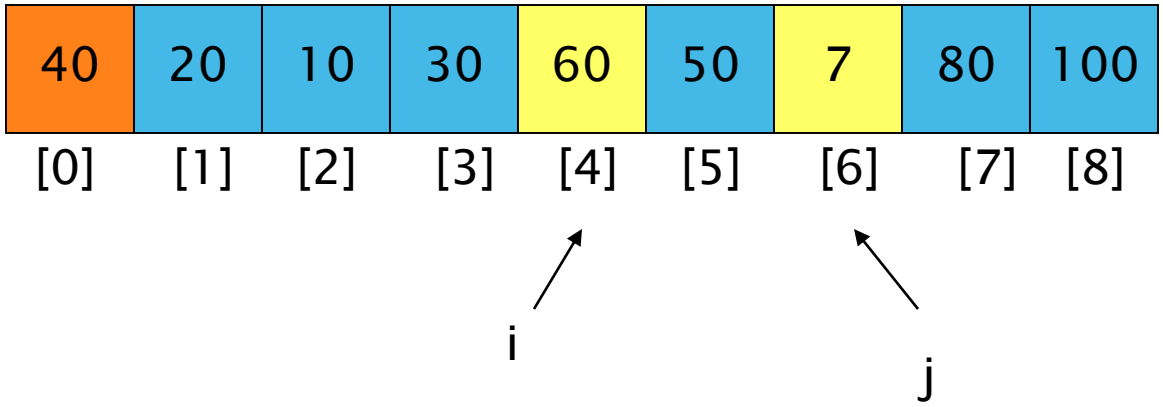
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0

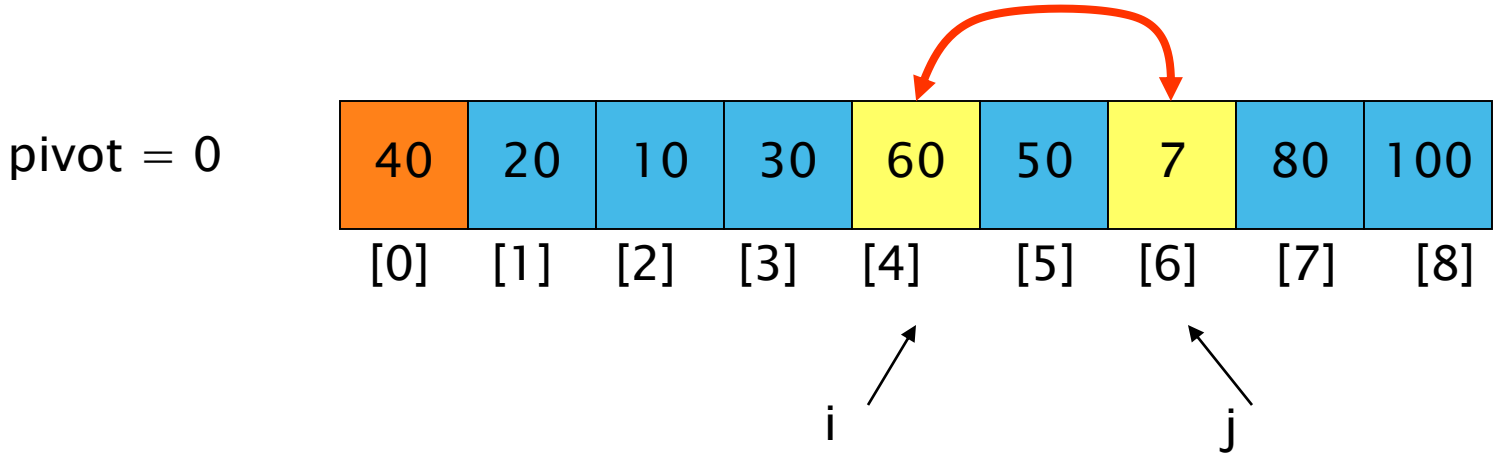


1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i< j
   swap a[i] and a[j]
4. While j > i, go to 1.

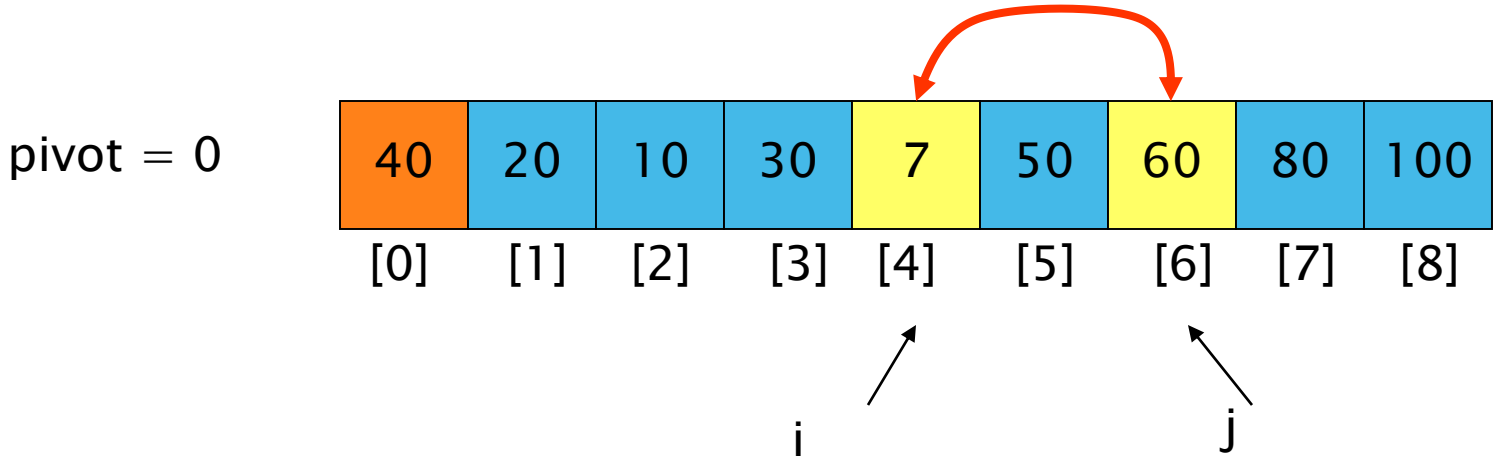
Pivot = 0



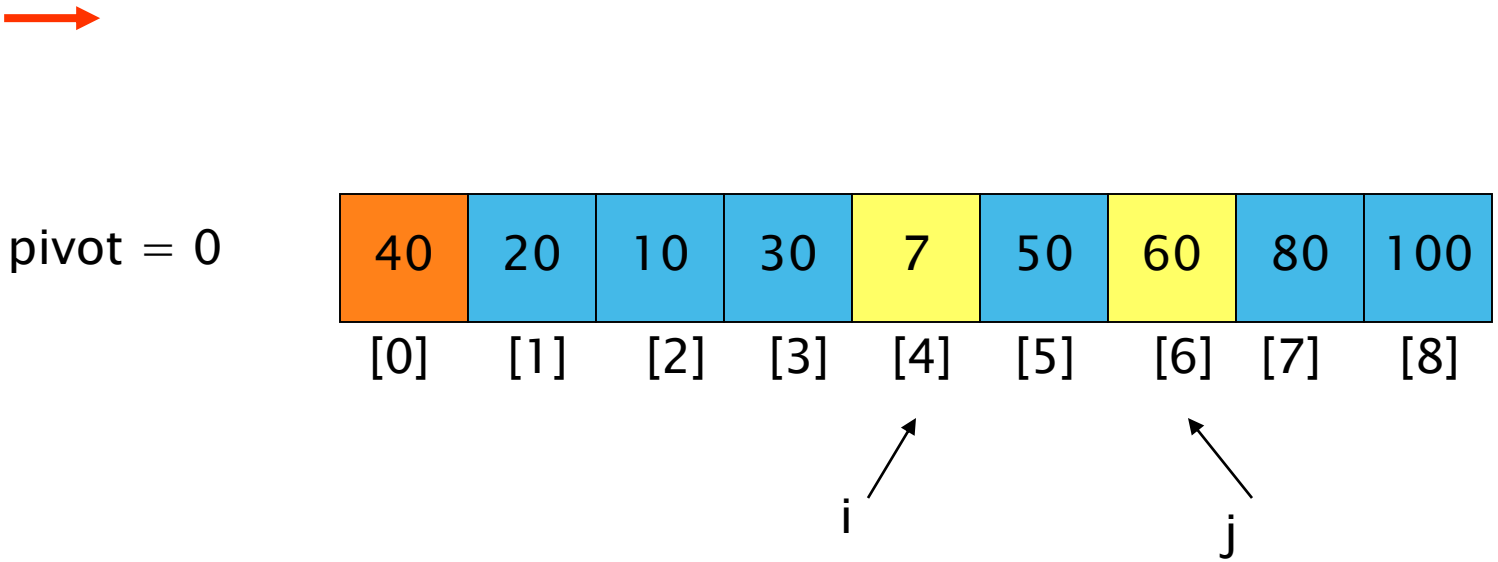
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.



1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

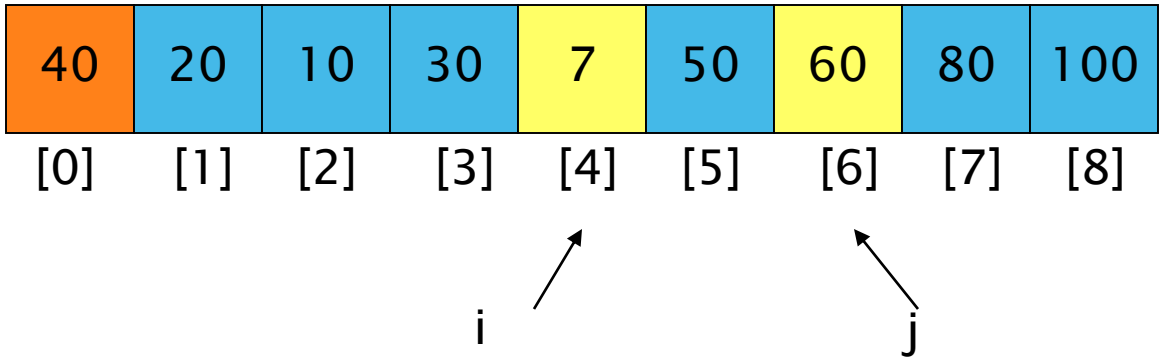


1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.



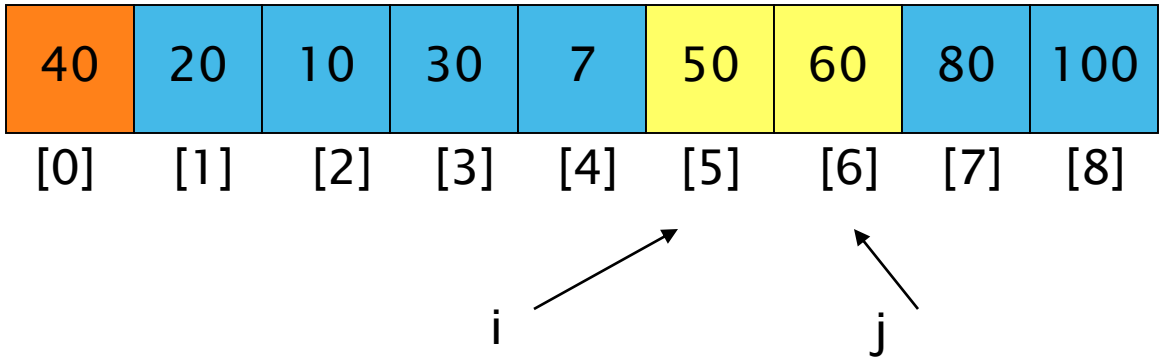
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0

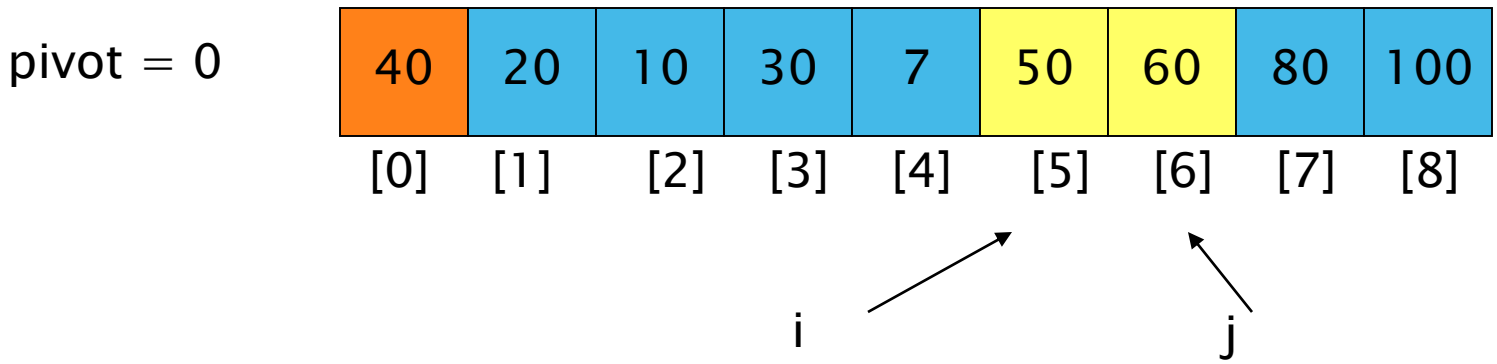


1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0

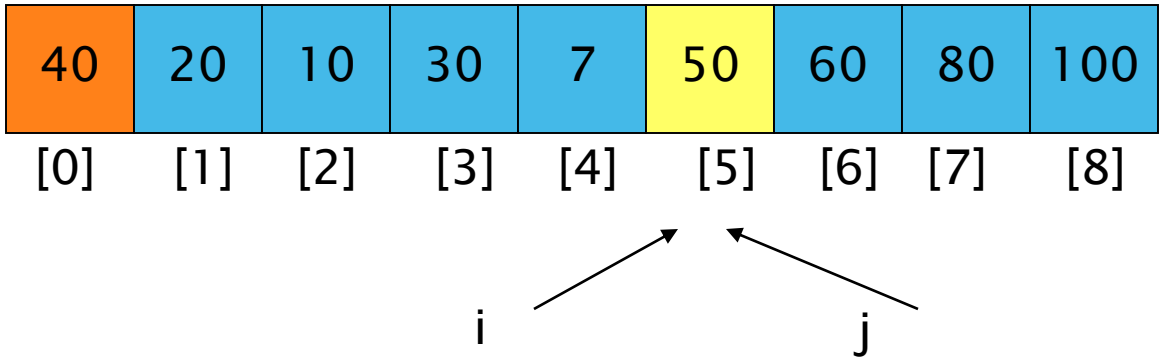


1. While a[i] <= a[pivot]
   ++i
2. While a[j] > a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.



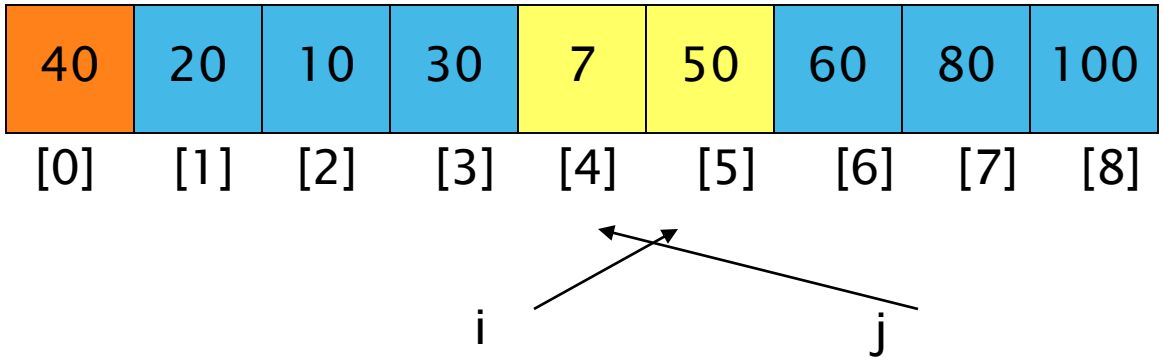
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0



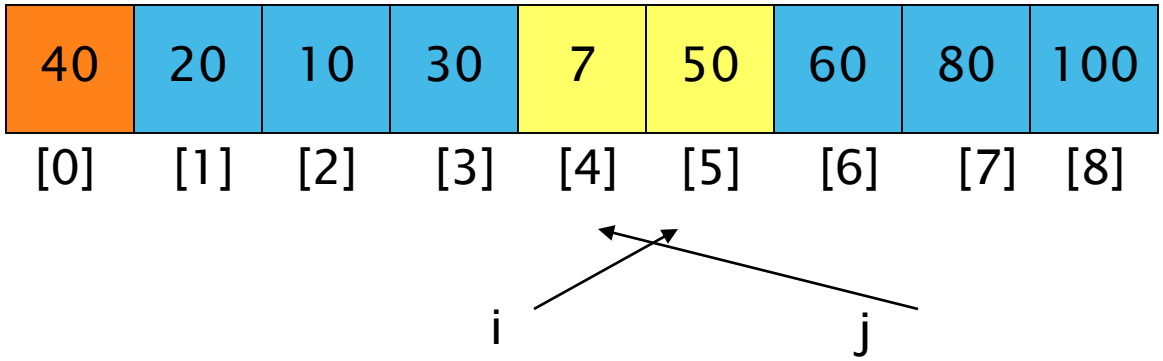
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0



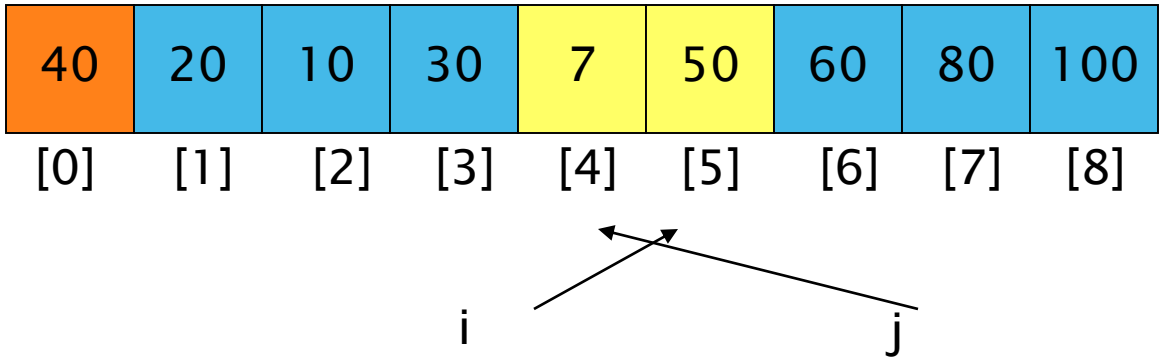
1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0

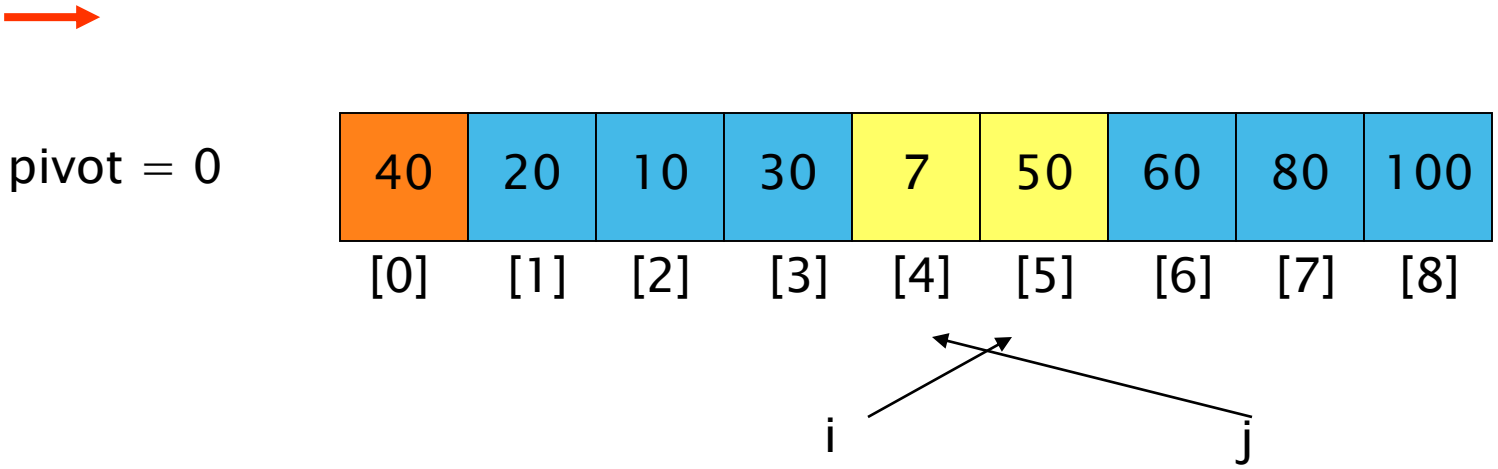


1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.

pivot = 0

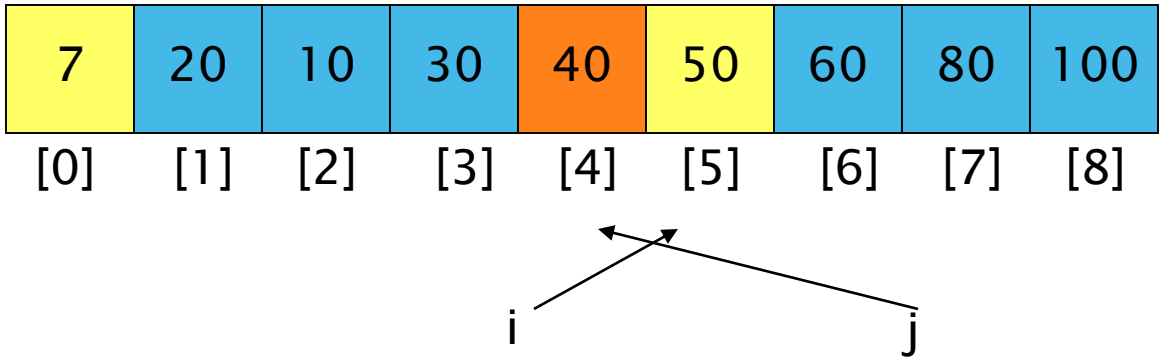


1. While a[i] <= a[pivot]
   ++i
2. While a[i] > a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.
5. Swap a[j] and a[pivot]



1. While a[i] <= a[pivot]
   ++i
2. While a[j] >= a[pivot]
   --j
3. If i < j
   swap a[i] and a[j]
4. While j > i, go to 1.
5. Swap a[j] and a[pivot]

pivot = 4



Partition Result