Data Structures and Algorithms
(ES221)

Quick Sort

Dr. Zubair Ahmad

Quick Sort

- As its name implies, *quicksort* is the fastest known sorting algorithm in practice.
  - It is very fast, mainly due to a very tight and highly optimized inner loop
  - Its average running time is $O(n \log n)$
  - It has $O(n^2)$ worst-case performance,
    - but this can be made exponentially unlikely with a little effort
  - The quicksort algorithm is simple to understand and prove correct
  - Like mergesort, quicksort is a divide-and-conquer recursive algorithm

2

Quick Sort

- The basic algorithm to sort an array *S* consists of the following four easy steps:
  - If the number of elements in ***S*** is **0** or **1**, then return
  - 2. Pick any element ***v*** in ***S***. This is called the ***pivot***.
  - 3. *Partition* $S - \{v\}$ (the remaining elements in ***S***) into two disjoint groups: $S_1 = \{x \in S - \{v\} \mid x \leq v\}$ *and* $S_2 = \{x \in S - \{v\} / x \geq v\}$
  - 4. Return { **quicksort(S_1)** followed by ***v*** followed by **quicksort(S_2)** }

3

Quick Sort

- Since the partition step ambiguously describes what to do with elements equal to the pivot, this becomes a design decision.
- Part of a good implementation is handling this case as efficiently as possible.
- Intuitively, we would hope that
  - about half the keys that are equal to the pivot go into S_1
  - while the other half into S_2, much as we like binary search trees to be balanced.

4

Quick Sort – Selecting the Pivot

1- The popular, uninformed choice:

- Use the first element as the pivot
  - This is acceptable if the input is random
  - But, if the input is presorted or in reverse order, then the pivot provides a poor partition, because virtually all the elements go into S_1 or S_2
  - It happens consistently throughout the recursive calls

➔ Quicksort will take quadratic time to do essentially nothing at all, which is quite embarrassing

2- A Safe Maneuver

- A safe course is merely to choose the pivot randomly
  - This strategy is generally perfectly safe, unless the random number generator has a flaw
- However, random number generation is generally an expensive commodity and does not reduce the average running time of the rest of the algorithm at all

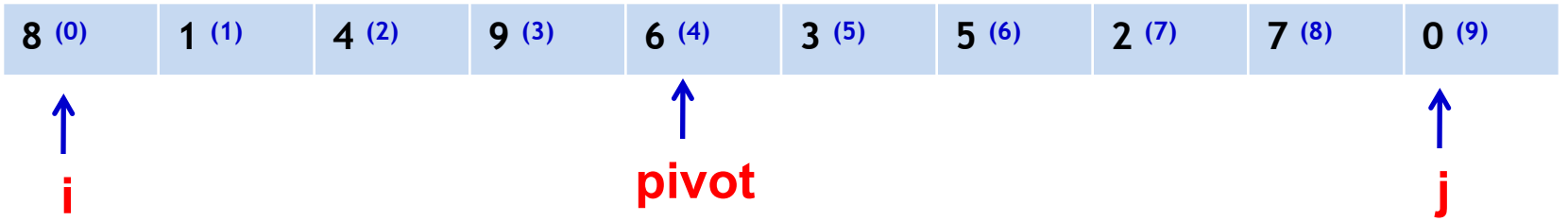
Quick Sort – Selecting the Pivot

- The best choice of pivot would be the median of the file.
  - The median of a group of ***n*** numbers is the (*n*/2)-th largest number
  - Unfortunately, this is hard to calculate and would slow down quicksort considerably
- A good estimate can be obtained by picking three elements randomly and using the median of these three as pivot.
  - The randomness turns out not to help much
- So, the common course is to use as pivot the median of the left, right and center elements

A = {8, 1, 4, 9, 6, 3, 5, 2, 7, 0} ⟶ center = A[*left* + *right*)/2]
= A [(0+9)/2] = **A[4] = 6**
pivot: v= median(8, 6, 0) = **6**

6

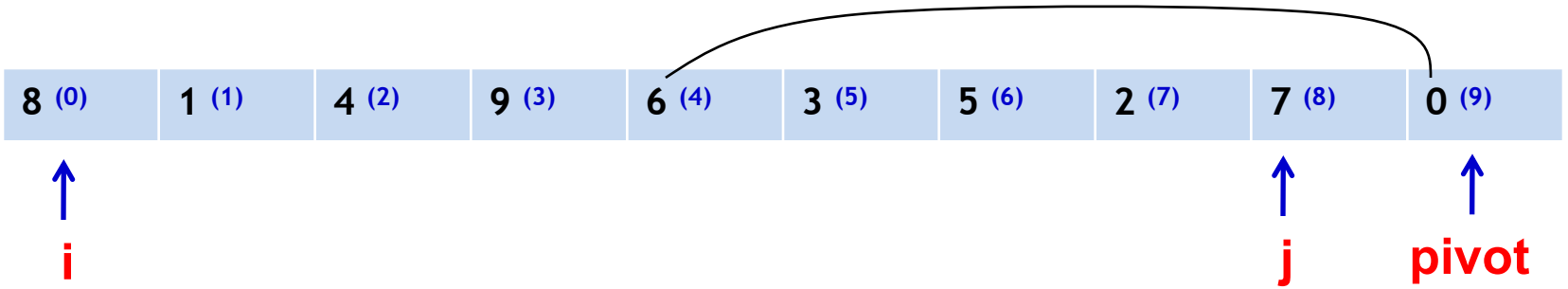
QuickSort :Partitioning strategy Example



- The basic algorithm
- While ***i*** is to the left of ***j***,
  - we move ***i*** right, skipping over elements smaller than the pivot
  - We move ***j*** left, skipping over elements larger than the pivot
- When ***i*** and ***j*** have stopped,
  - ***i*** is pointing at a large element, and
  - ***j*** is pointing at a small element
- If ***i*** is to the left of ***j***,
  - those elements are swapped
- The effect is to push a large element to the right and a small element to the left.
- In the example above, ***i*** would not move and ***j*** would slide over one place

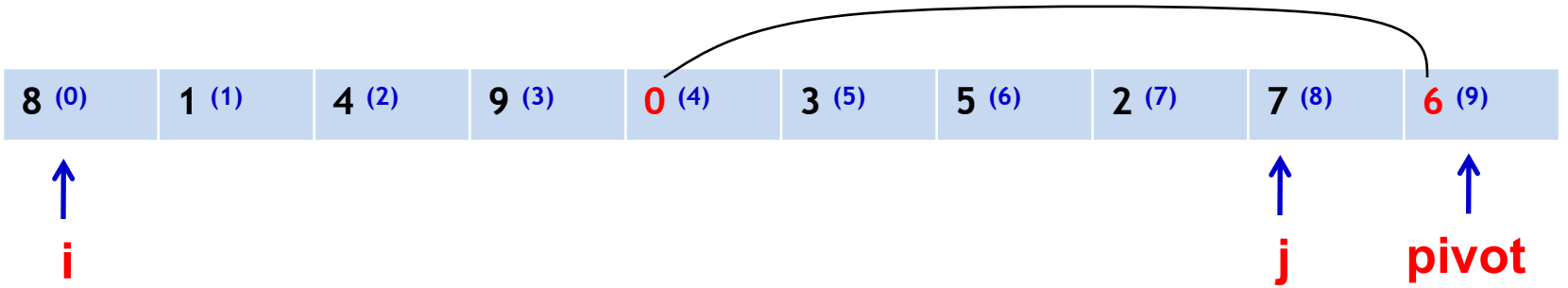
7

QuickSort :Partitioning strategy Step 1



8

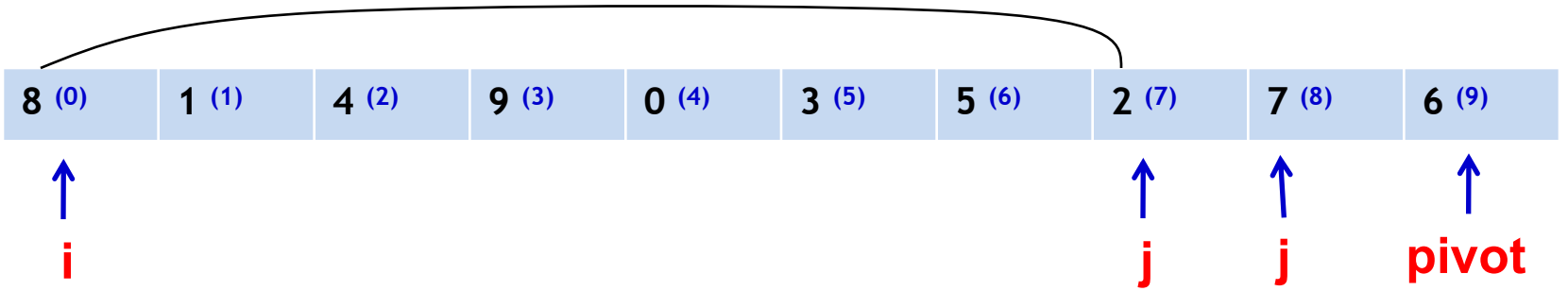
QuickSort :Partitioning strategy Step 1



- Start by swapping the *pivot* with the *right*, starting *j* at *right* -1
- *A*[*i*] = 8 > pivot
  - Stop ***i*** right over here

9

QuickSort :Partitioning strategy Step 1

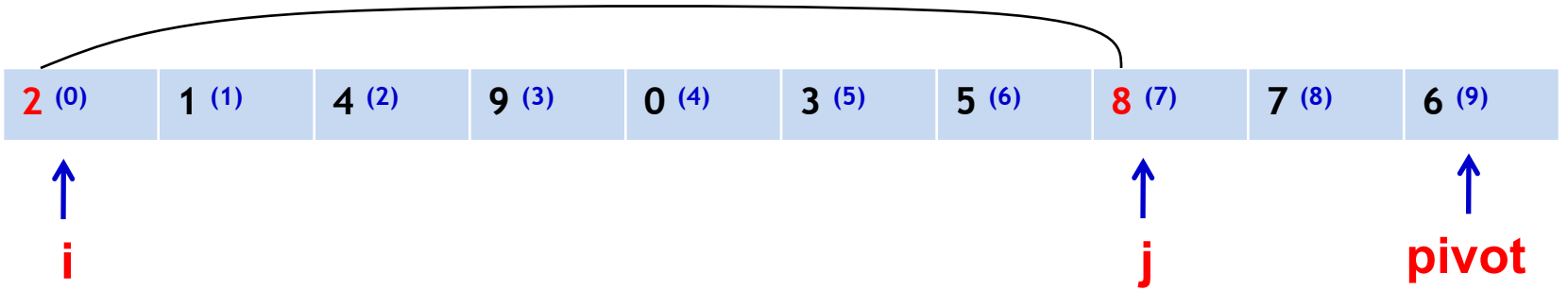


- $A[j] = 7 > pivot$
  - Move Left
- $A[j] = 2 < pivot$
  - Stop ***j*** right over here

- Swap $A[i]$ and $A[j]$

10

QuickSort :Partitioning strategy Step 1

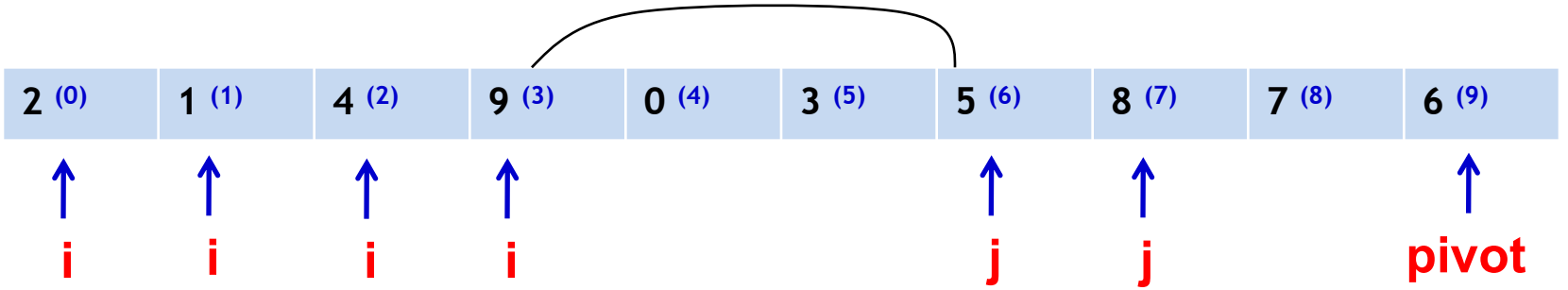


- $A[j] = 7 > pivot$
  - Move Right
- $A[j] = 2 < pivot$
  - Stop ***j*** right over here

- Swap $A[i]$ and $A[j]$

11

QuickSort :Partitioning strategy Step 2

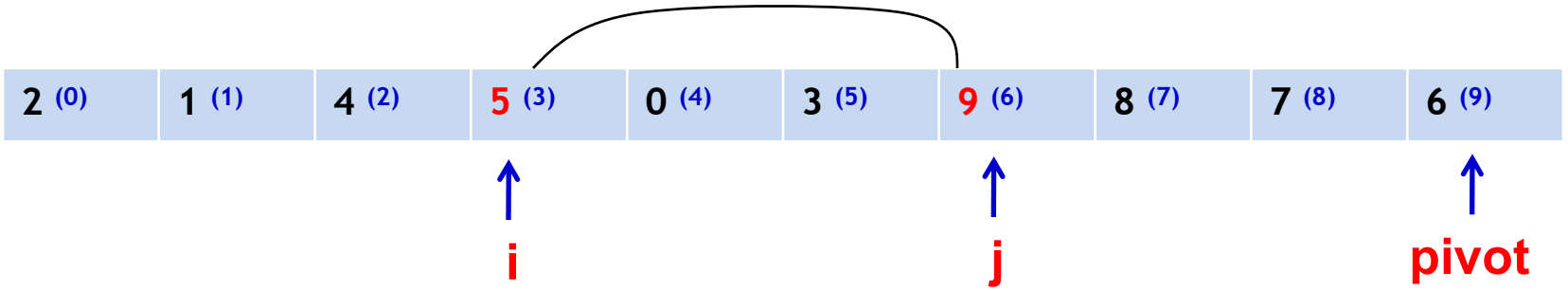


- $A[i] = 2 < pivot$
  - Move Right
- $A[i] = 1 < pivot$
  - Move Right
- $A[i] = 4 < pivot$
  - Move Right
- $A[i] = 9 > pivot$
  - Stop ***i*** right over here

- $A[j] = 8 > pivot$
  - Move Left
- $A[j] = 5 < pivot$
  - Stop ***j*** right over here

12

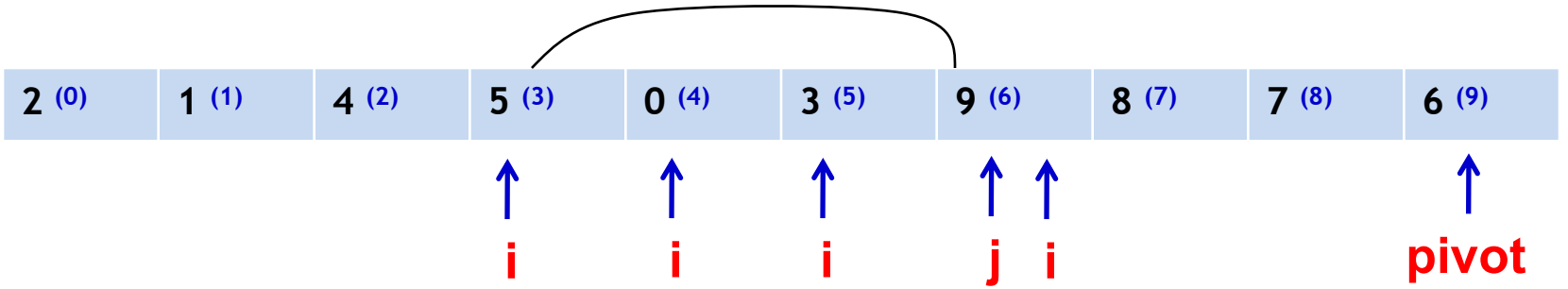
QuickSort :Partitioning strategy Step 2



- $A[i] = 2 < pivot$
  - Move Right
- $A[i] = 1 < pivot$
  - Move Right
- $A[i] = 4 < pivot$
  - Move Right
- $A[i] = 9 > pivot$
  - Stop ***i*** right over here

- $A[j] = 8 > pivot$
  - Move Left
- $A[j] = 5 < pivot$
  - Stop ***j*** right over here
- Swap $A[i]$ and $A[j]$

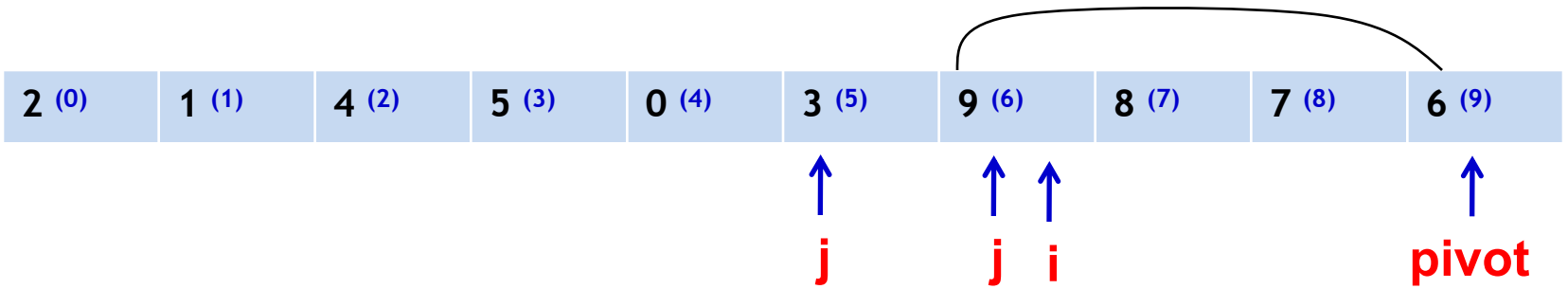
QuickSort :Partitioning strategy Step 3



- $A[i] = 5 < pivot$
  - Move Right
- $A[i] = 0 < pivot$
  - Move Right
- $A[i] = 3 < pivot$
  - Move Right
- $A[i] = 9 > pivot$
  - Stop ***i*** right over here

14

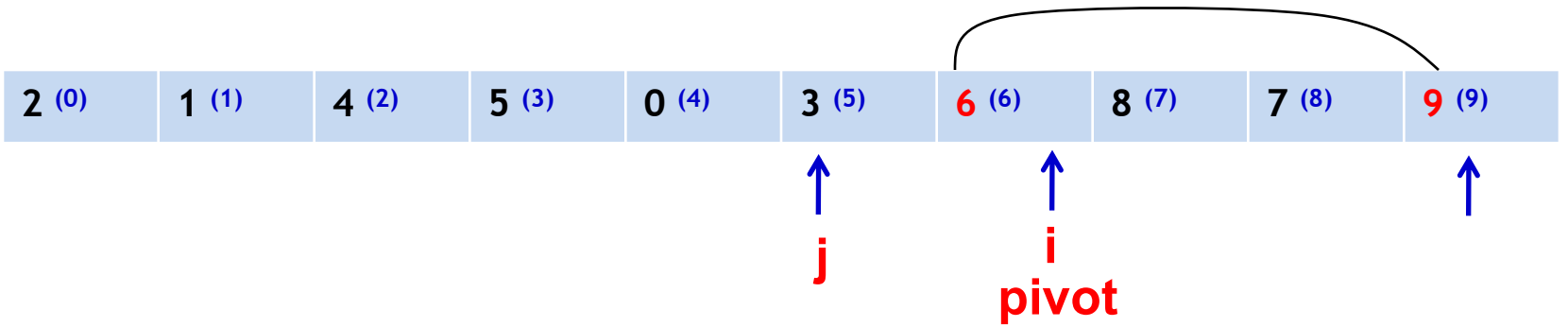
QuickSort :Partitioning strategy Step 3



- $A[i] = 5 < pivot$
  - Move Right
- $A[i] = 0 < pivot$
  - Move Right
- $A[i] = 3 < pivot$
  - Move Right
- $A[i] = 9 > pivot$
  - Stop ***i*** right over here
- $A[j] = 9 > pivot$
  - Move Left
- $A[j] = 3 < pivot$
  - Stop ***j*** right over here
- *i* and *j* have crossed
  - So no swap for $A[i]$ and $A[j]$
- Instead Swap $A[i]$ and $A[pivot]$

15

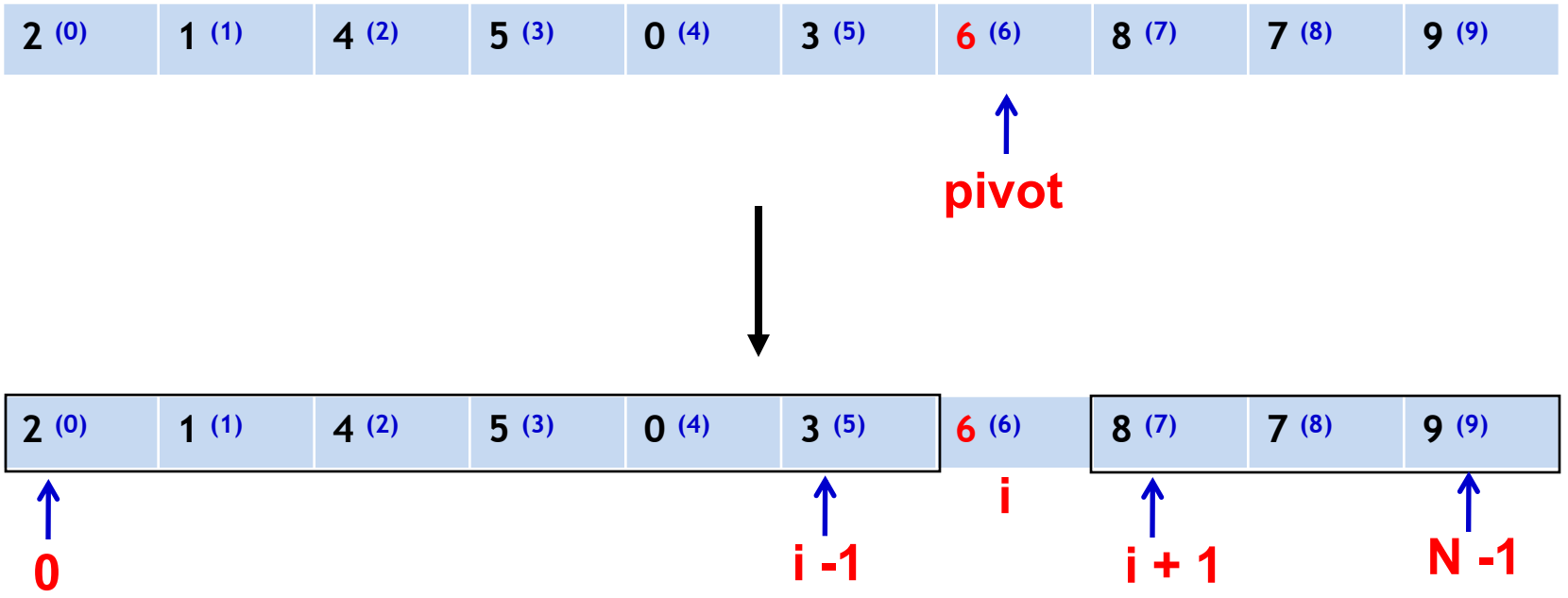
QuickSort :Partitioning strategy Step 3



- $A[i] = 5 < pivot$
  - Move Right
- $A[i] = 0 < pivot$
  - Move Right
- $A[i] = 3 < pivot$
  - Move Right
- $A[i] = 9 > pivot$
  - Stop ***i*** right over here
- $A[j] = 9 > pivot$
  - Move Left
- $A[j] = 3 < pivot$
  - Stop ***j*** right over here
- *i* and *j* have crossed
  - So no swap for $A[i]$ and $A[j]$
- Instead Swap $A[i]$ and $A[pivot]$

16

QuickSort : Recursive calls



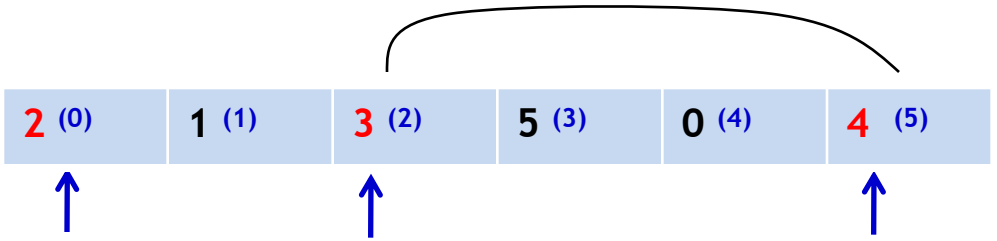
17

QuickSort : Left Recursive call

2 (0)	1 (1)	4 (2)	5 (3)	0 (4)	3 (5)

18

QuickSort : Left Recursive call
Pivot Selection



2 > 4 ✖

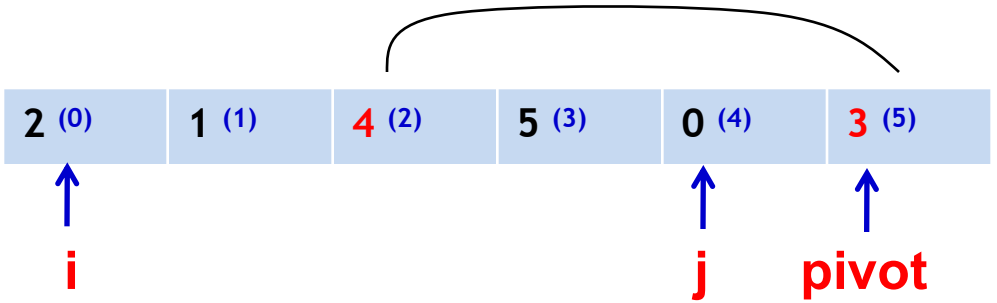
2 > 3 ✖

4 > 3

19

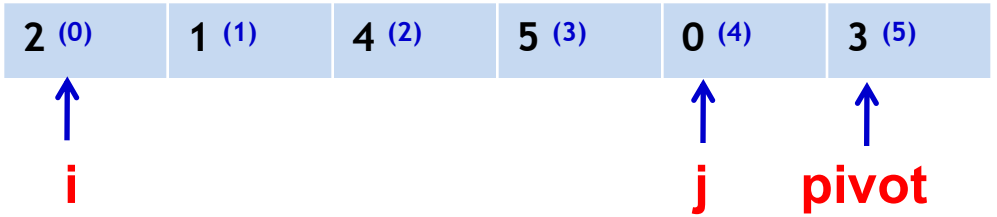
QuickSort : Left Recursive call

Pivot Swap out



20

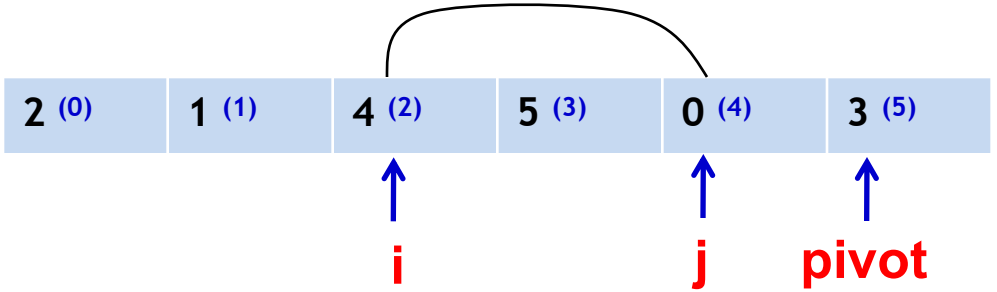
QuickSort : Left Recursive call Movements



21

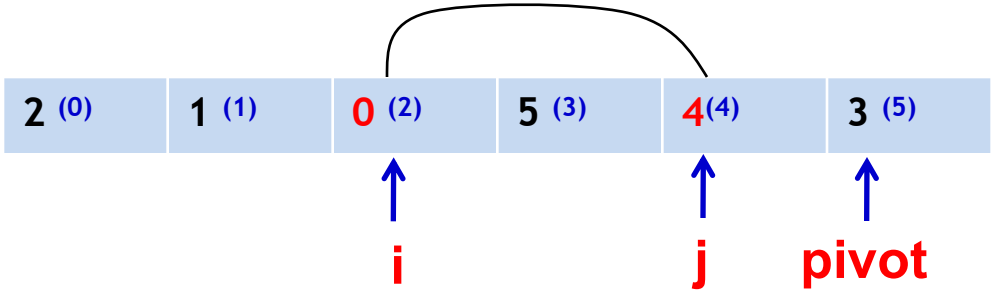
QuickSort : Left Recursive call

Swap



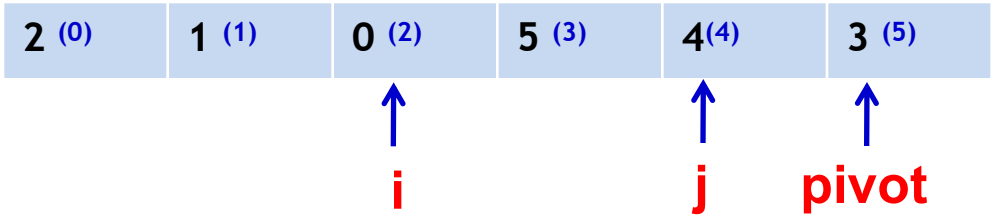
22

QuickSort : Left Recursive call
Step 1: Swap



23

QuickSort : Left Recursive call Movements

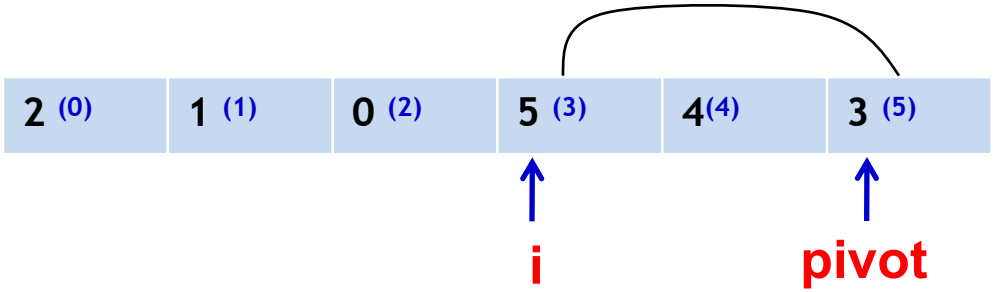


i & j crossed

24

QuickSort : Left Recursive call

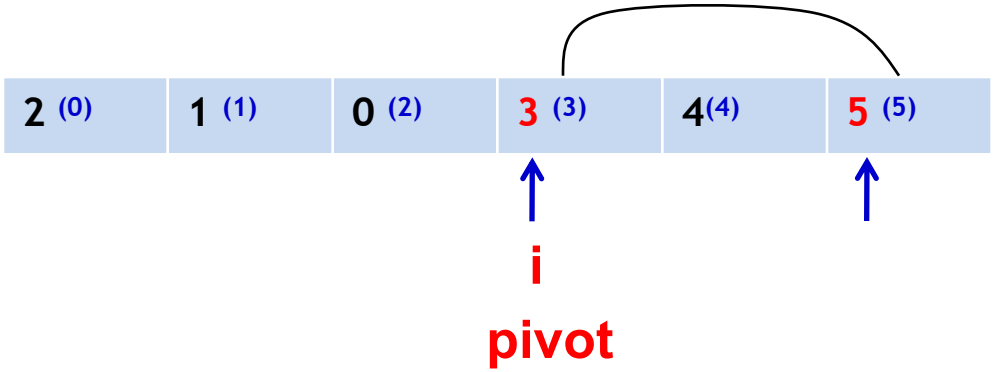
Swap with the pivot



25

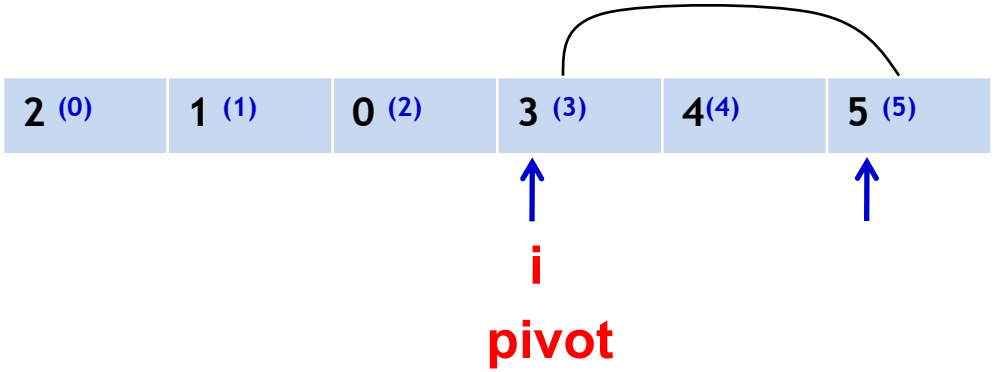
QuickSort : Left Recursive call

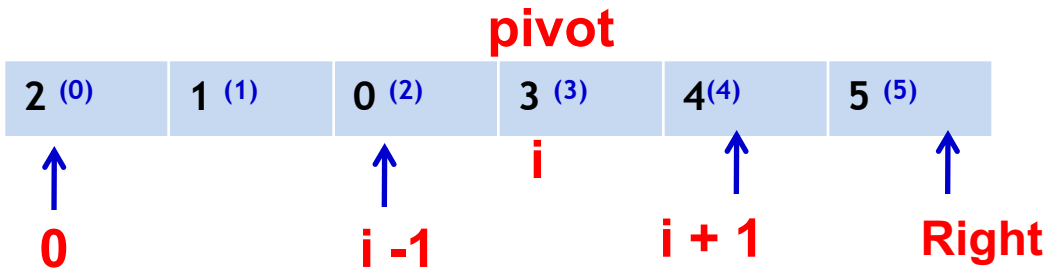
Swap with the pivot



26

QuickSort :Recursive Calls





27

QuickSort

```
void quick_sort( input_type a[ ], unsigned int n )
    {
    q_sort( a, 0, n-1 );
   }
```


28

QuickSort: Core Function

```
void q_sort( input_type a[], int left, int right )
{
int i, j;  int pivot;
if( left + CUTOFF <= right )
    {
     pivot = median3( a, left, right );
     i = left; j = right - 1;
     for ( ; ; )
             {
          while( a[++i] < pivot );
           while( a[--j] > pivot );
            if( i < j )
                swap( &a[i], &a[j] );
            else
                 break;
            } //end for loop
     swap( &a[i], &a[right-1] ); /*restore pivot*/
      q_sort( a, left, i -1 ); // left recursive call
      q_sort( a, i +1, right ); // lright recursive call
    }
}
```

QuickSort: Medians


```
/* Return median of left, center, and right. */
/* Order these and hide pivot */
int median3( input_type a[], int left, int right )
{
int center;
center = (left + right) / 2;
if ( a[left] > a[center] )
      swap( &a[left], &a[center] );
if ( a[left] > a[right] )
      swap( &a[left], &a[right] );
if ( a[center] > a[right] )
      swap( &a[center], &a[right] );
/* a[left] <= a[center] <= a[right] */
swap( &a[center], &a[right-1] );
return a[right-1]; /* return pivot */
}
```


30

QuickSort: Medians

```
/* Return median of left, center, and right. */
/* Order these and hide pivot */
```

8 (0)	1 (1)	4 (2)	9 (3)	6 (4)	3 (5)	5 (6)	2 (7)	7 (8)	0 (9)

left center right

```
{
int center;

                                ight) / 2;
if ( a[left] > a[center] )
      swap( &a[left], &a[center] );
if ( a[left] > a[right] )
      swap( &a[left], &a[right] );
if ( a[center] > a[right] )
      swap( &a[center], &a[right] );
/* a[left] <= a[center] <= a[right] */
swap( &a[center], &a[right-1] );
return a[right-1]; /* return pivot */
}
```

8 > 6

6 > 0

8 > 6

31

QuickSort: Medians

```
/* Return median of left, center, and right. */
/* Order these and hide pivot */
```

0 (0)	1 (1)	4 (2)	9 (3)	6 (4)	3 (5)	5 (6)	2 (7)	7 (8)	8 (9)

center

right -1

pivot

```
{
int center;
center = (left + right) / 2;
if ( a[left] > a[center] )
      swap( &a[left], &a[center] );
if ( a[left] > a[right] )
      swap( &a[left], &a[right] );
if ( a[center] > a[right] )
      swap( &a[center], &a[right] );
/* a[left] <= a[center] <= a[right] */
swap( &a[center], &a[right-1] );
return a[right-1];
}
```

8 > 6

6 > 0

8 > 6

/* hide pivot */

/* return pivot */

32

QuickSort: Core Function

```
void q_sort( input_type a[], int left, int right )
```

0 (0)	1 (1)	4 (2)	9 (3)	7 (4)	3 (5)	5 (6)	2 (7)	6 (8)	8 (9)

i = left

center

j = right -1

pivot

```
    pivot = median3( a, left, right );
    i=left; j=right-1;
    for ( ; ; )
        {
          while ( a[++i] < pivot );
          while ( a[--j] > pivot );
          if ( i < j )
              swap( &a[i], &a[j] );
          else
              break;
          } //end for loop
    swap( &a[i], &a[right-1] );
     q_sort( a, left, i-1 );
     q_sort( a, i+1, right );
   }
}
```

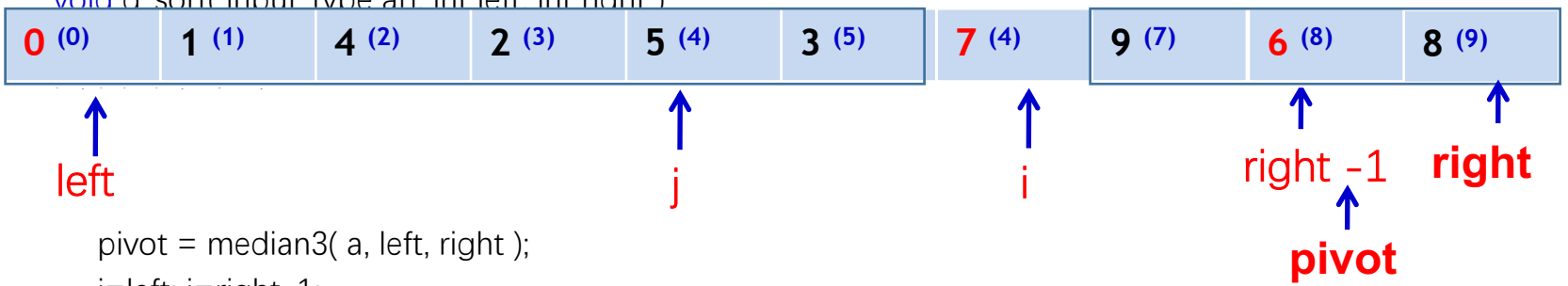
/* while ***i*** is to the left of ***j***,

- move ***i*** **right**, skipping over elements **smaller** than the pivot
- move ***j*** **left**, skipping over elements **larger** than the pivot. */

/* If ***i*** is to the left of ***j***,

- those elements are swapped */

QuickSort: Core Function




```
void q_sort( input_type a[], int left, int right )
{
    pivot = median3( a, left, right );
    i=left; j=right-1;
    for ( ; ; )
        {
          while ( a[++i] < pivot );
          while ( a[--j] > pivot );
          if ( i < j )
              swap( &a[i], &a[j] );
          else
              break;
          } //end for loop
    swap( &a[i], &a[right-1] );
     q_sort( a, left, i-1 );
     q_sort( a, i+1, right );
   }
}
```

/* loop terminated *when (i > j) i.e., i* and *j* have crossed → so no swap is performed */

/*restore pivot*/

// left recursive call

// lright recursive call

Questions?

zahmaad.github.io