Course Name: CS302-Design an analysis of Algorithm

Credit Hours: 3

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,
  - The quicksort algorithm is simple to understand and prove correct
  - Like mergesort, quicksort is a divide-and-conquer recursive algorithm

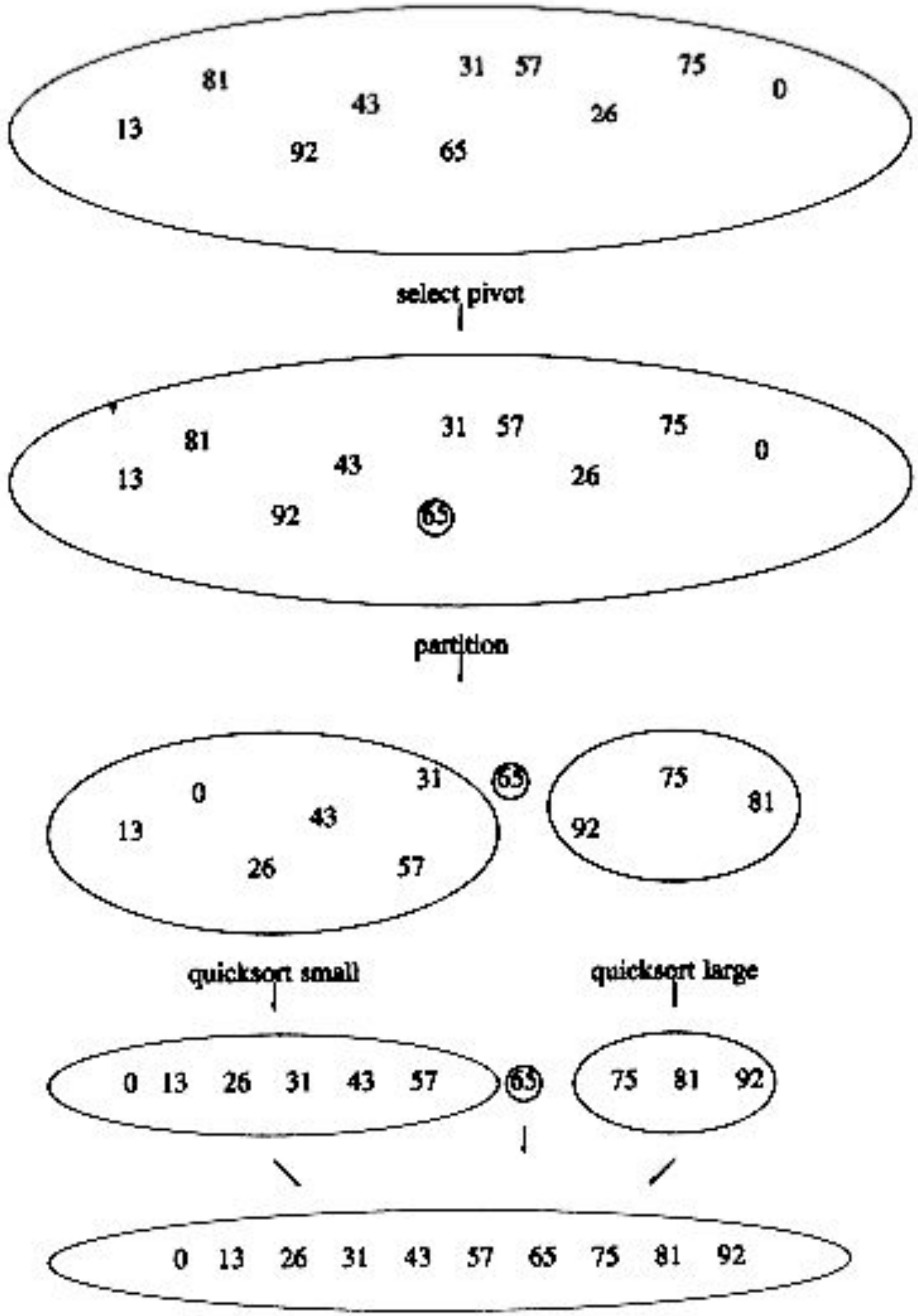
Quick Sort

- 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: $S_1 = \{x \in S - \{v\} \mid x \le v\}$ *and* $S_2 = \{x \in S - \{v\} \mid x \ge v\}$
  - 4. Return { **quicksort(S_1)** followed by ***v*** followed by **quicksort(S_2)** }

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.

Quick Sort - Selecting the Pivot



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**

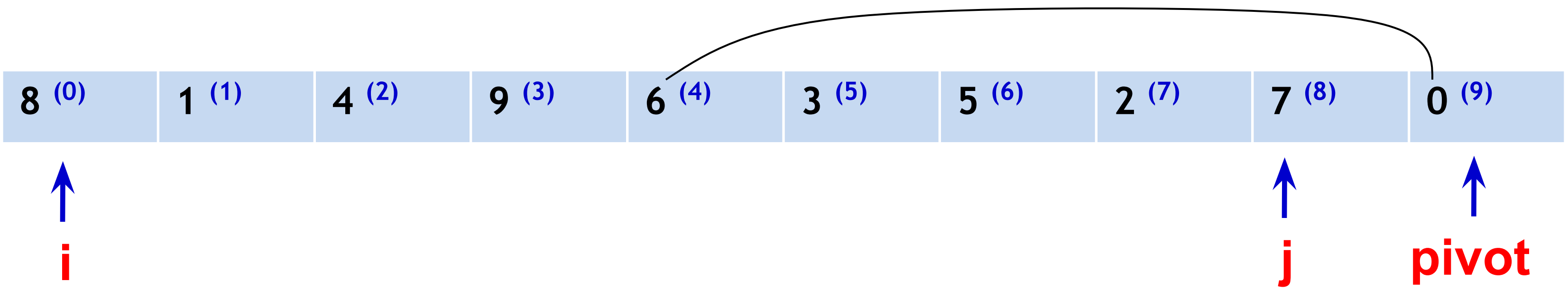
QuickSort :Partitioning strategy Example

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

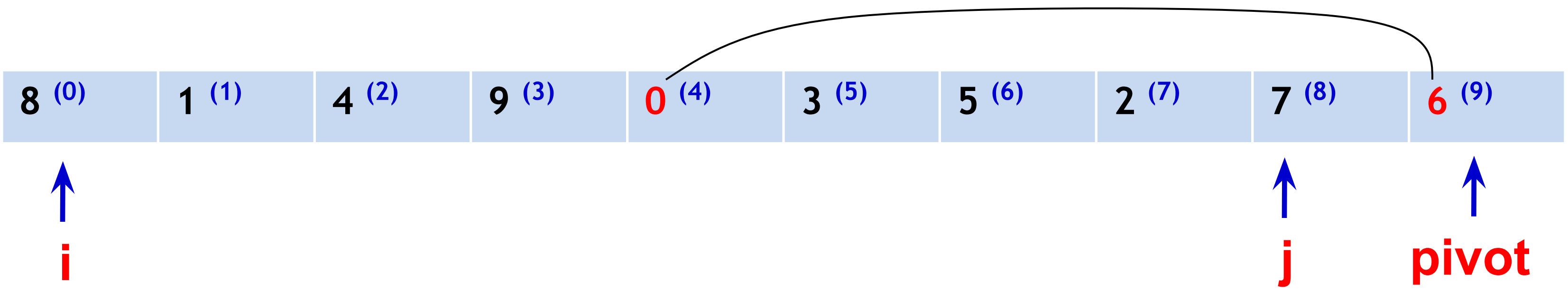
i pivot j

- 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 same for the ***j***

QuickSort :Partitioning strategy Step 1

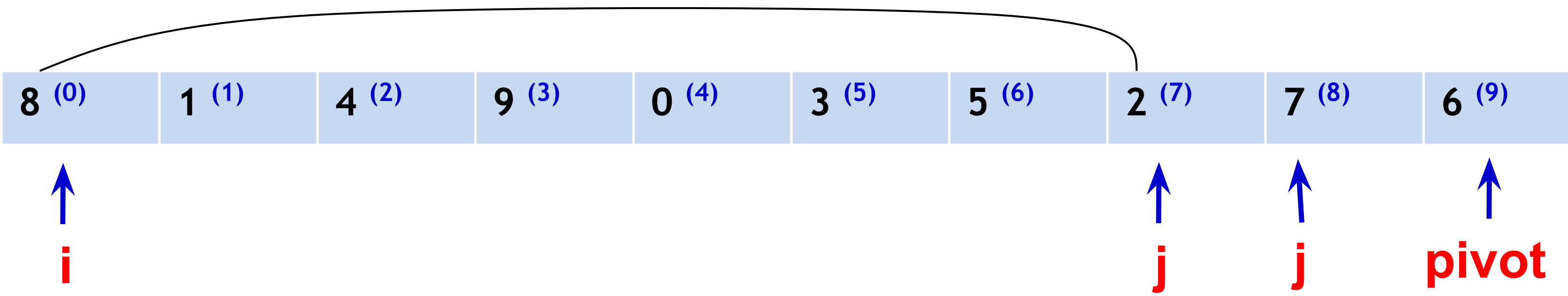


QuickSort :Partitioning strategy
Step 1



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

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*]

QuickSort :Partitioning strategy
Step 1

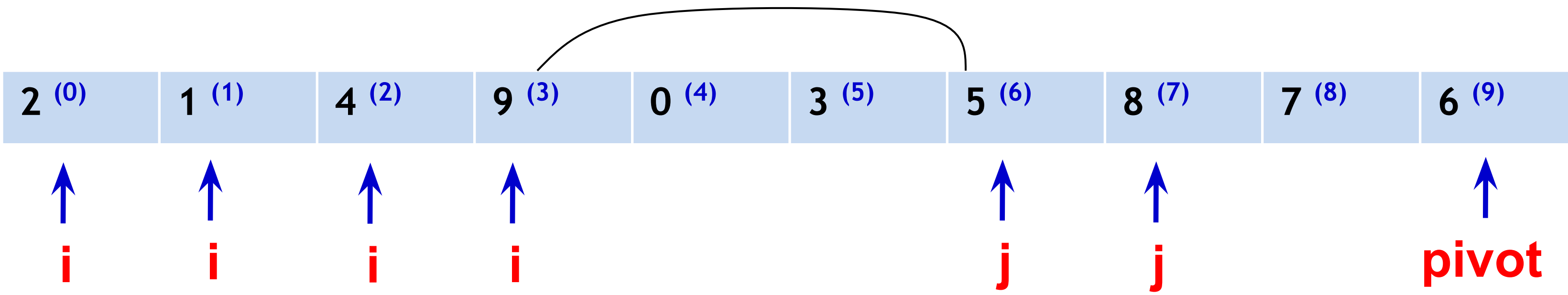
2 (0)	1 (1)	4 (2)	9 (3)	0 (4)	3 (5)	5 (6)	8 (7)	7 (8)	6 (9)

i j pivot

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

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

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

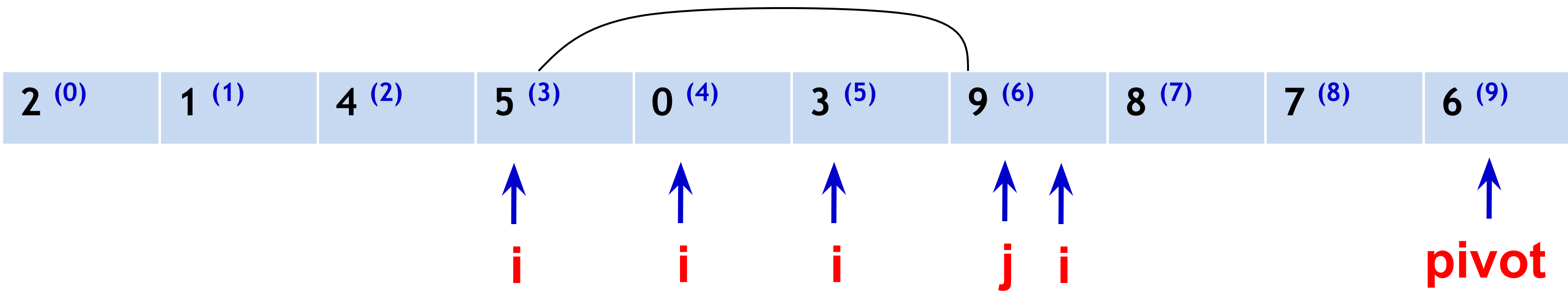
QuickSort :Partitioning strategy
Step 2

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

i j pivot

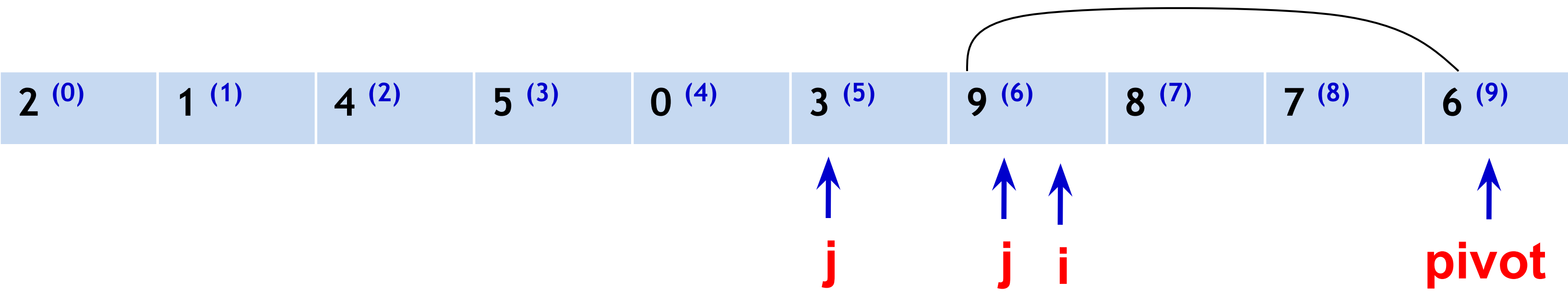
- $A[i] = 2 <$ pivot
  - Move Right
- $A[i] = 1 <$ pivot
  - Move Right
- $A[i] = 4 <$ pivot
  - Move Right
- $A[i] = 9 >$ pivot
  - Stop ***i*** right over here
- $A[j] = 8 >$ pivot
  - Move Left
- $A[j] = 5 <$ pivot
  - Stop ***j*** right over here
- Swap $A[i]$ and $A[j]$

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

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*]

QuickSort :Partitioning strategy
Step 3

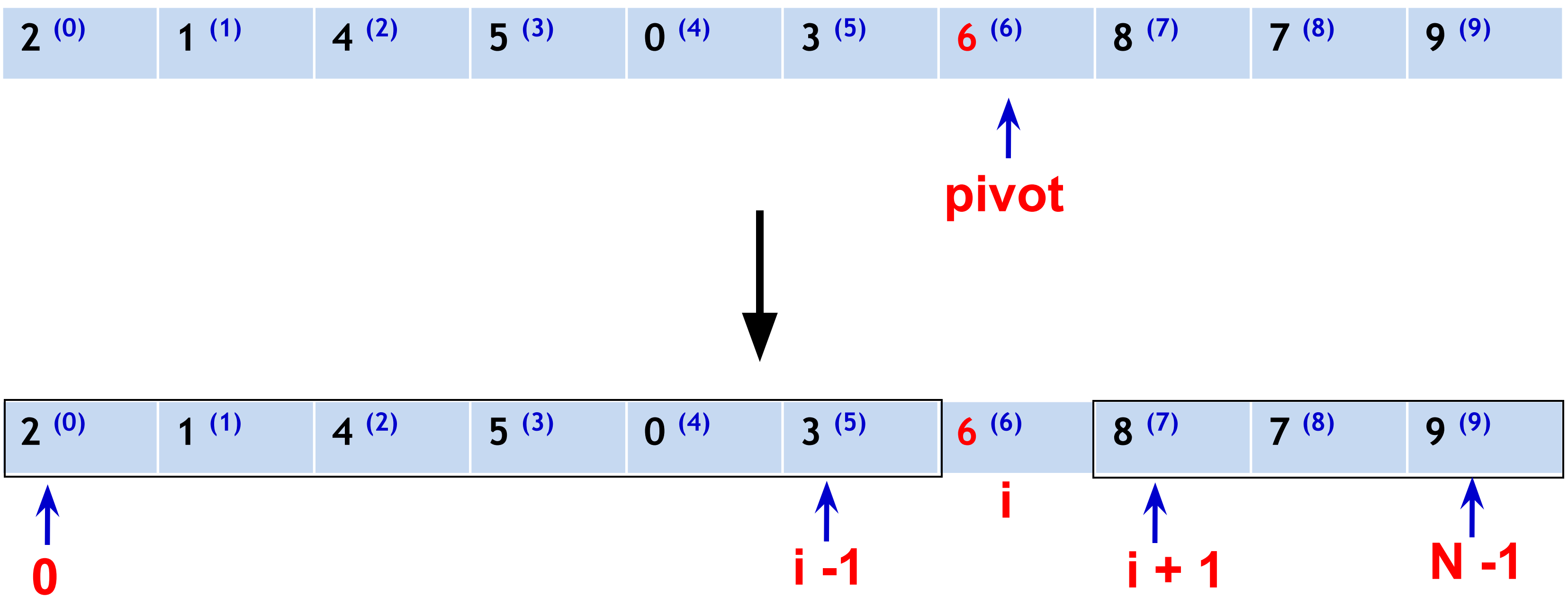
2 (0)	1 (1)	4 (2)	5 (3)	0 (4)	3 (5)	6 (6)	8 (7)	7 (8)	9 (9)

j

i
pivot

- $A[i]$ = 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 $A[pivot]$

QuickSort : Recursive calls



QuickSort : Left Recursive call

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

Always select pivot in one manner. If you select pivot as first/middle/Last element always select that element as pivot in recursive calls

Better approach. Select pivot as Array[(L+R)/2] as pivot

QuickSort : Left Recursive call

An example to select last element as pivot

2 (0)	1 (1)	4 (2)	5 (3)	0 (4)	3 (5)
↑ i				↑ j	↑ pivot

QuickSort : Left Recursive call
Movements

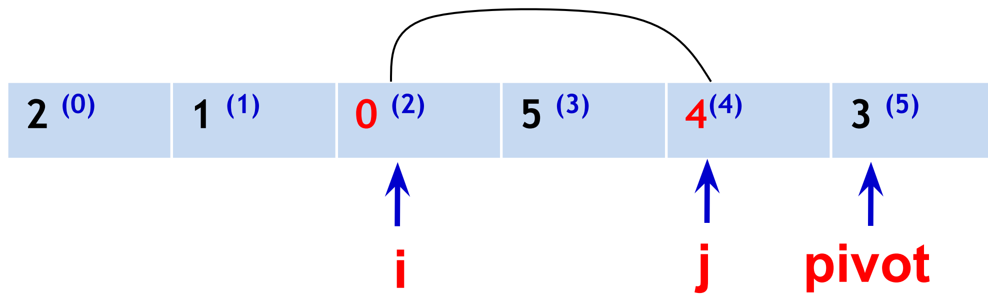
2 (0)	1 (1)	4 (2)	5 (3)	0 (4)	3 (5)
i				j	pivot

QuickSort : Left Recursive call

Swap

2 $^{(0)}$	1 $^{(1)}$	4 $^{(2)}$	5 $^{(3)}$	0 $^{(4)}$	3 $^{(5)}$
		i		j	pivot

QuickSort : Left Recursive call
Step 1: Swap



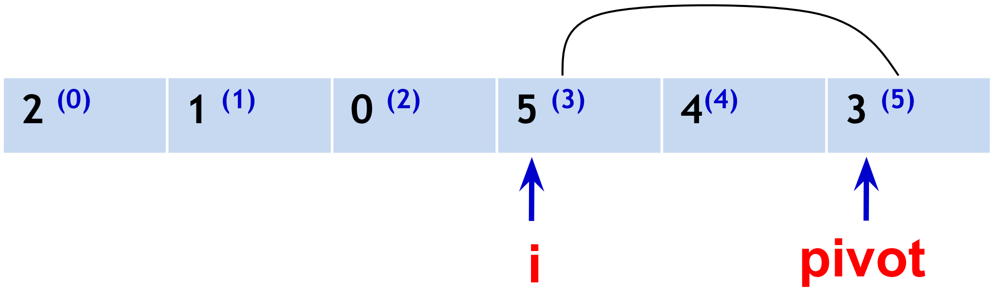
QuickSort : Left Recursive call
Movements

2 $^{(0)}$	1 $^{(1)}$	0 $^{(2)}$	5 $^{(3)}$	4$^{(4)}$	3 $^{(5)}$
		↑ i		↑ j	↑ pivot

i & j crossed

QuickSort : Left Recursive call

Swap with the pivot



QuickSort : Left Recursive call

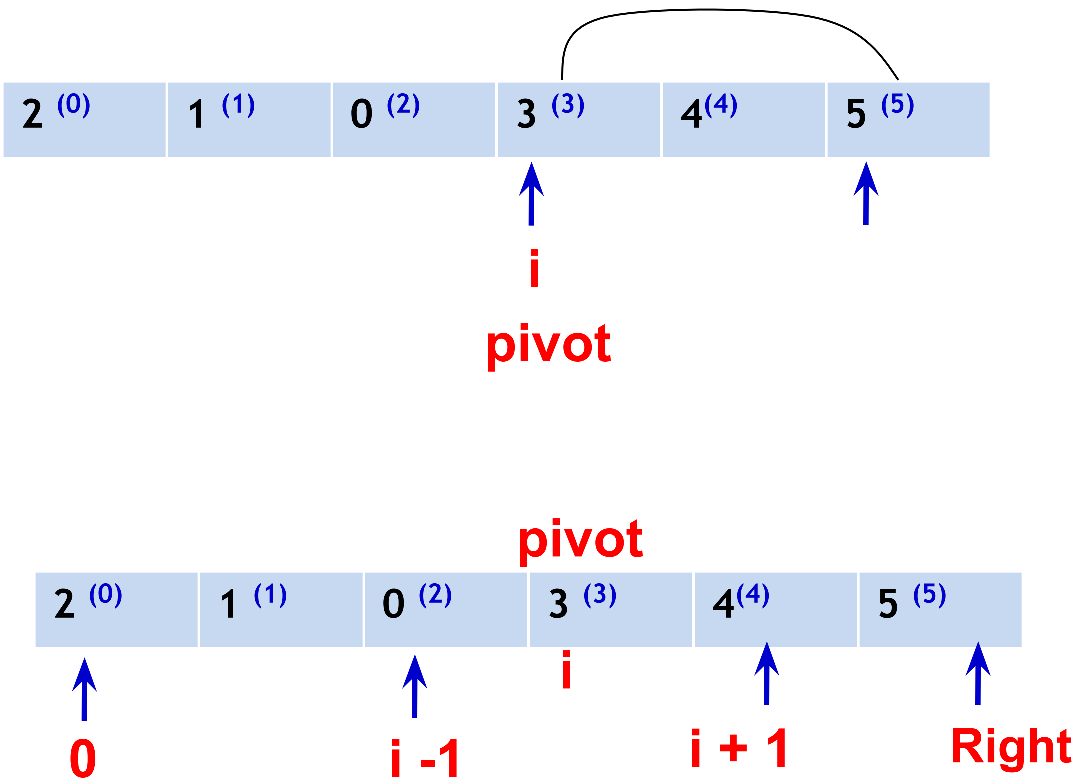
Swap with the pivot

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

i

pivot

QuickSort :Recursive Calls



QuickSort

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

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 */
}
```

QuickSort: Medians

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

left center right

```
/* Return median of left, center, and right. */
/* Order these and hide pivot */
{
center = (left + right) / 2;
if ( a[left] > a[center] )          8 > 6
      swap( &a[left], &a[center] );
if ( a[left] > a[right] )           6 > 0
      swap( &a[left], &a[right] );
if ( a[center] > a[right] )         8 > 6
      swap( &a[center], &a[right] );
/* a[left] <= a[center] <= a[right] */
swap( &a[center], &a[right-1] );
return a[right-1]; /* return pivot */
}
```

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)

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

left j i right -1 right

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] );    /*restore pivot*/
    q_sort( a, left, i-1 );    // left recursive call
    q_sort( a, i+1, right ); // lright recursive call
    }
}
```

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

QuickSort Analysis

- 2 cases of Quick sort
  - Best/average case
    - where the partition of array is in middle like merge sort
  - Worst case (Imaginary)
    - Array is partition into two parts (1 items & n-1 items) in division
- For best/Average case mathematical function?

$$T(n) = 2\,T\left(\frac{n}{2}\right) + cn$$

- For worst case mathematical function?

$$T(n) = T(n-1) + T(0) + cn$$

QuickSort Analysis

- For best/Average case mathematical function

$$T(n) = 2\,T\left(\frac{n}{2}\right) + cn$$

- This is exactly same as merge sort algorithm
- Try to solve is again by al of the three method discuss last week

- Complexity?