COMP 3203
Introduction to Data Structures and Algorithms

Quick Sort

Quick Sort Algorithm

- Given an array of *n* elements (e.g., integers):
  - If array only contains one element, return
  - If array contains two element
    - swap the elements if they are not in-order
  - Else
    - pick one element to use as *pivot.*
    - Partition elements into two sub-arrays:
      - □ Elements less than or equal to pivot
      - □ Elements greater than pivot
    - Quicksort the two sub-arrays
    - Return results

Partitioning Array

- Given a pivot (can be any array element, for example the first or the last), partition the elements of the array such that the resulting array consists of:
  - One sub-array that contains elements <= pivot
  - Another sub-array that contains elements > pivot
- The sub-arrays are stored in the original data array.
- Partitioning loops swapping elements below/above pivot.

Quick Sort - Code

```
static void QuickSort(int[] data, int left, int right)
{
    if ( right <= left +1) //0,1,or 2 items
    {
        if ( right == left + 1)// 2 items
            compareSwap(data, left, right);
        return;
    }
    int pivot = partition(data, left, right);
    QuickSort(data, left, pivot-1);
    QuickSort(data, pivot+1, right);
}
```

Swap Function

```
static void compareSwap(int[] data,int left,int right)
{
    if ( data[right] < data[left])
    {
        int temp = data[left];
        data[left] = data[right];
        data[right] = temp;
    }
}
```

Partition: Quick Sort - Code

```
static int partition(int[] data, int left, int right)
{
     int pivot(left), i(left+1), j(right), temp;
     while  (j >  i){
         while ((i < data.length) && (data[i] <= data[pivot])) i++;
         while (data[j] > data[pivot]) j--;
         if (i < j){ //swap
             temp = data[i];
             data[i] = data[j];
             data[j] = temp;
         }//if
     }//while

     //Swap data[j] and data[pivot]
     if(pivot != j) {
       temp = data[pivot];
       data[pivot] = data[j];
       data[j] = temp;
     }
     return j;
}
```

Example

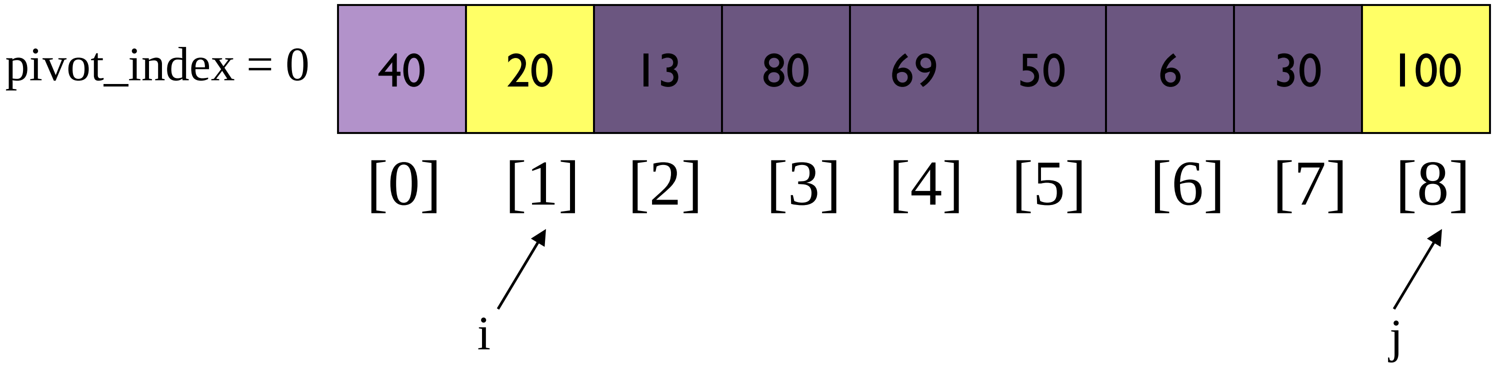
- We are given array of *n* integers to sort:

40	20	13	80	69	50	6	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:

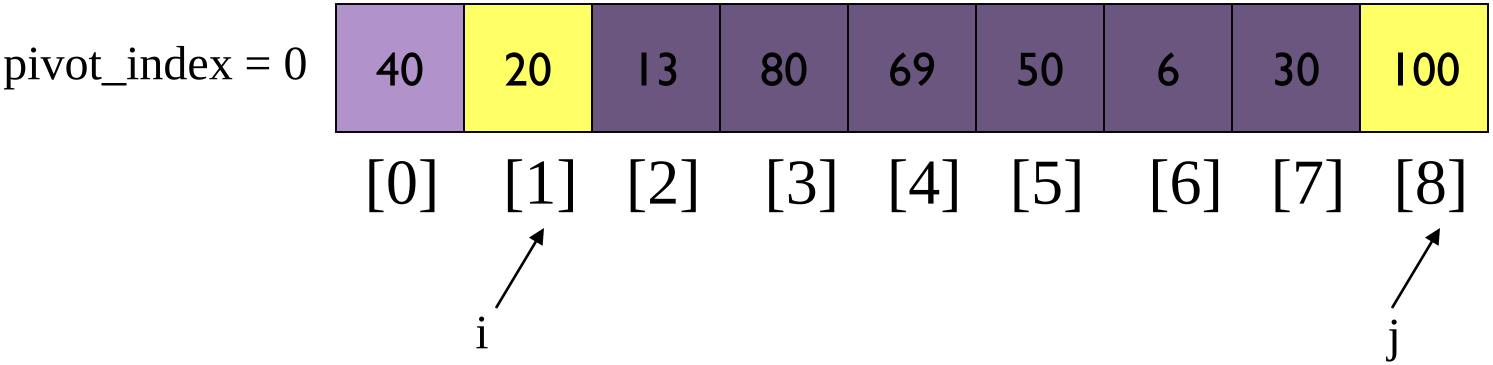
40	20	13	80	69	50	6	30	100



pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	80	69	50	6	30	100

```
1. while (data[i] <= data[pivot]) i++;
```



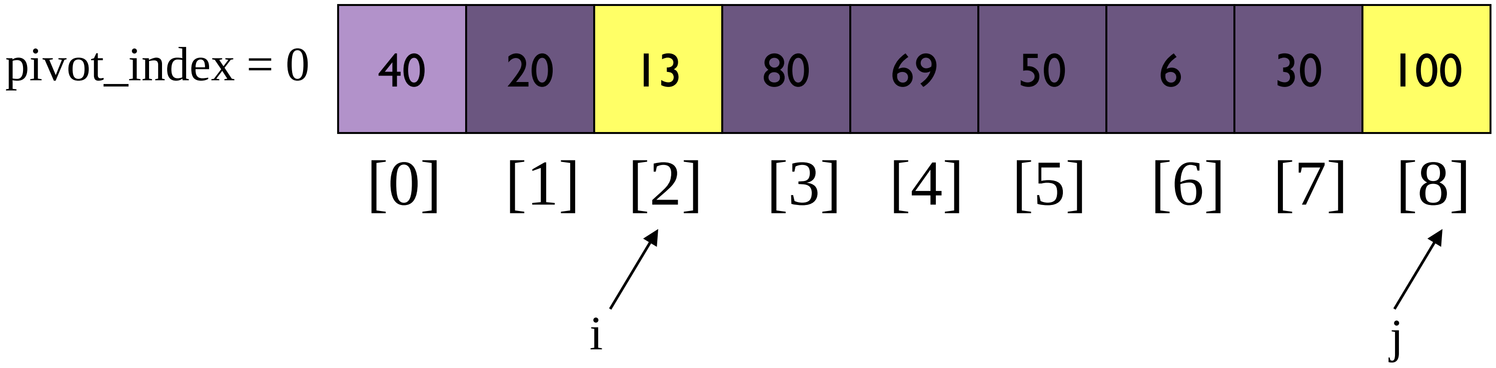
pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	80	69	50	6	30	100

i

j

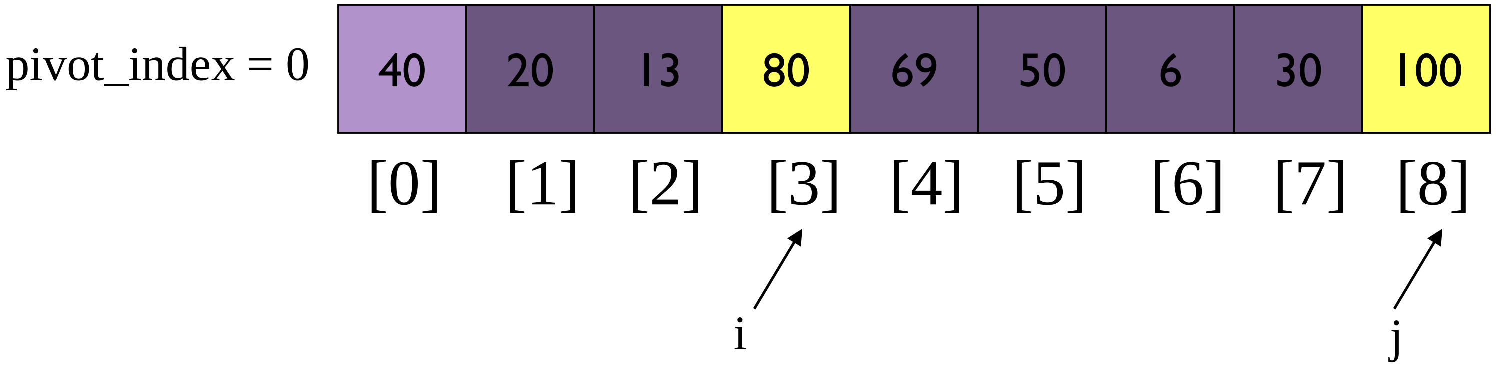
```
1. while (data[i] <= data[pivot]) i++;
```



pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	80	69	50	6	30	100

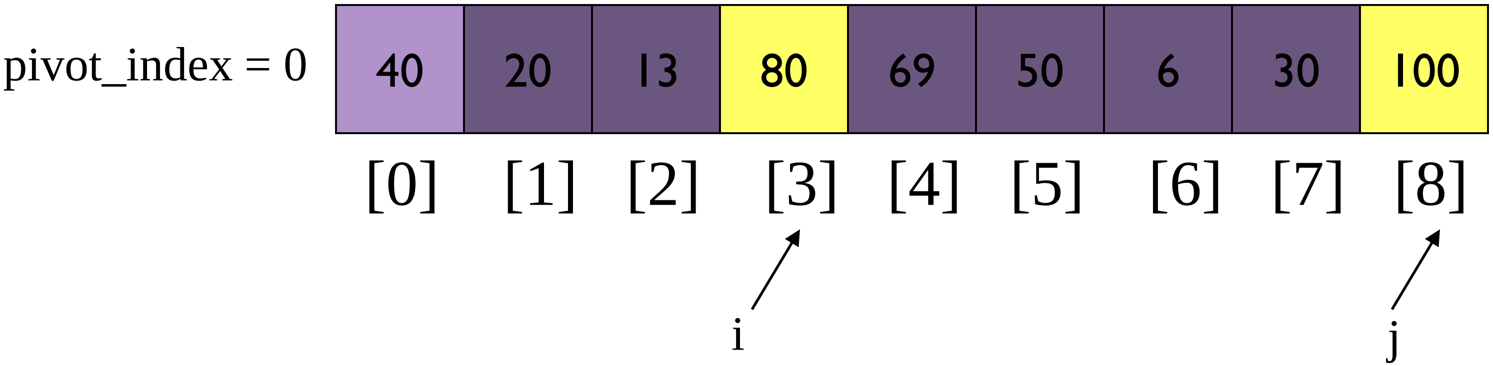
```
1. while (data[i] <= data[pivot]) i++;
```



pivot_index = 0

40	20	13	80	69	50	6	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
```



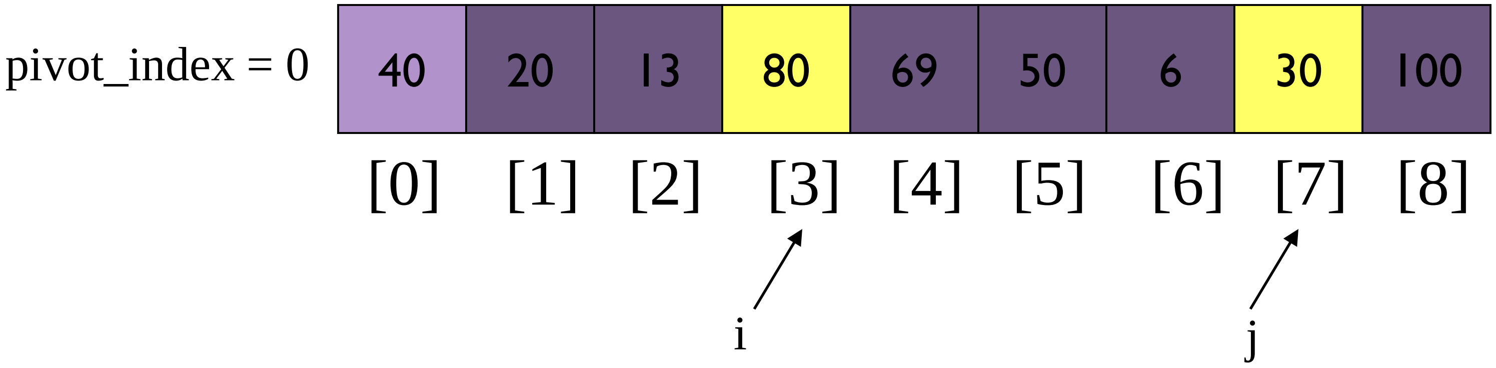
pivot_index = 0

40	20	13	80	69	50	6	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

i

j

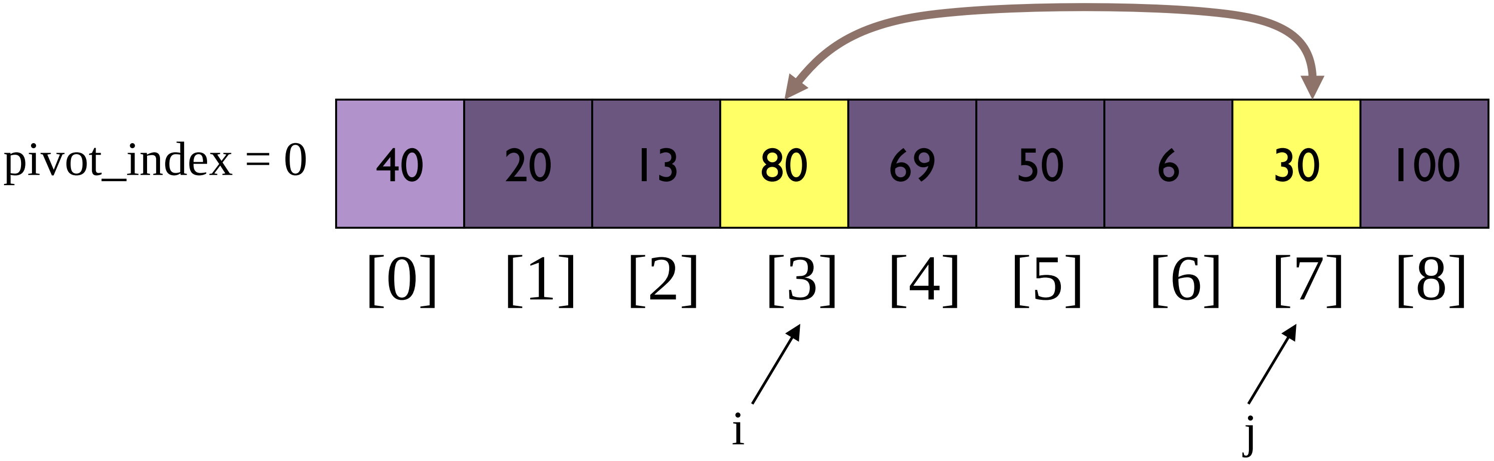
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
```



pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	80	69	50	6	30	100

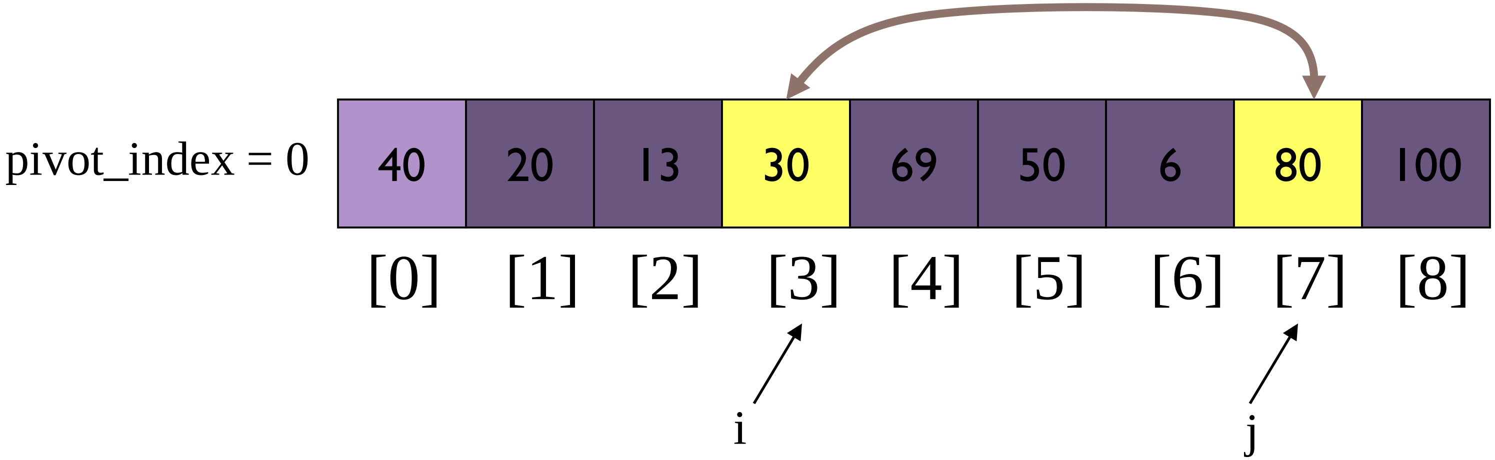
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
      swap data[i] and data[j]
```



pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	80	69	50	6	30	100

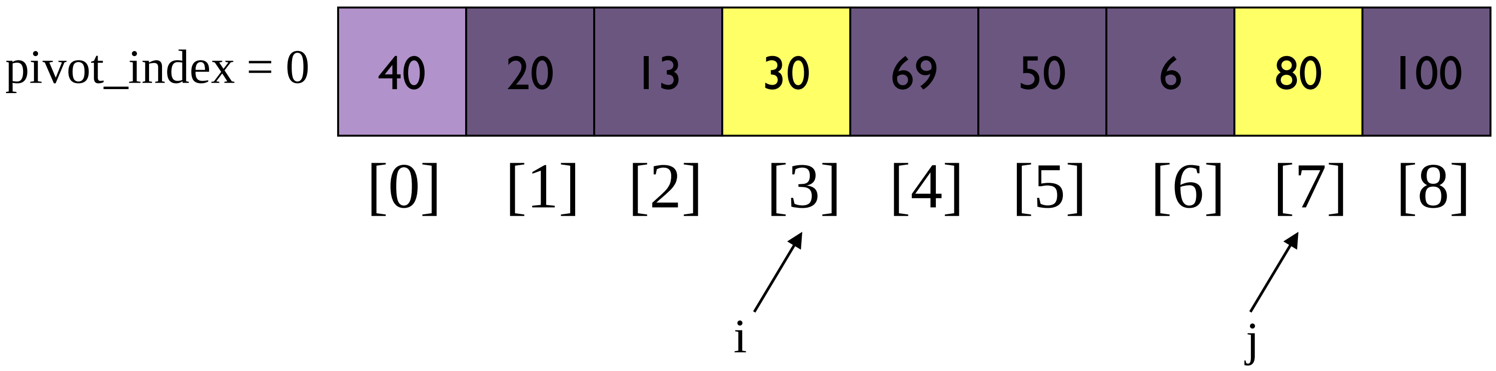
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
```



pivot_index = 0

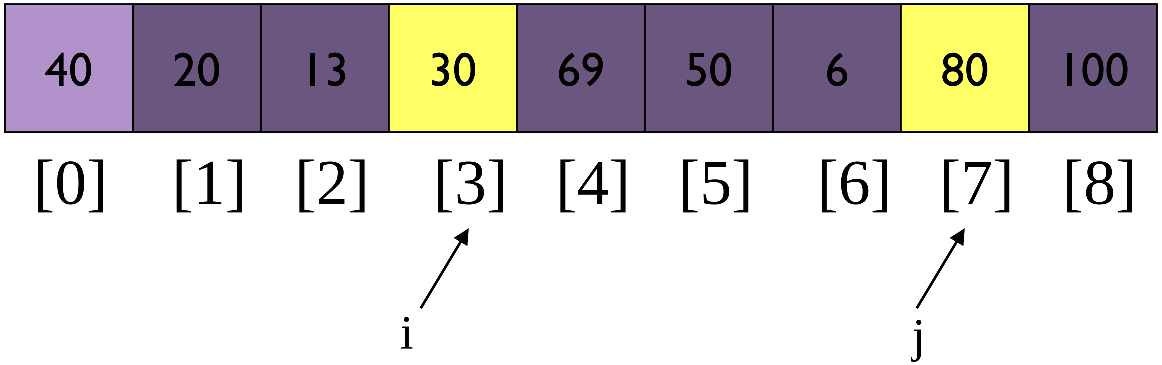
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	30	69	50	6	80	100

```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
      swap data[i] and data[j]
4. while j > i, go to 1.
```



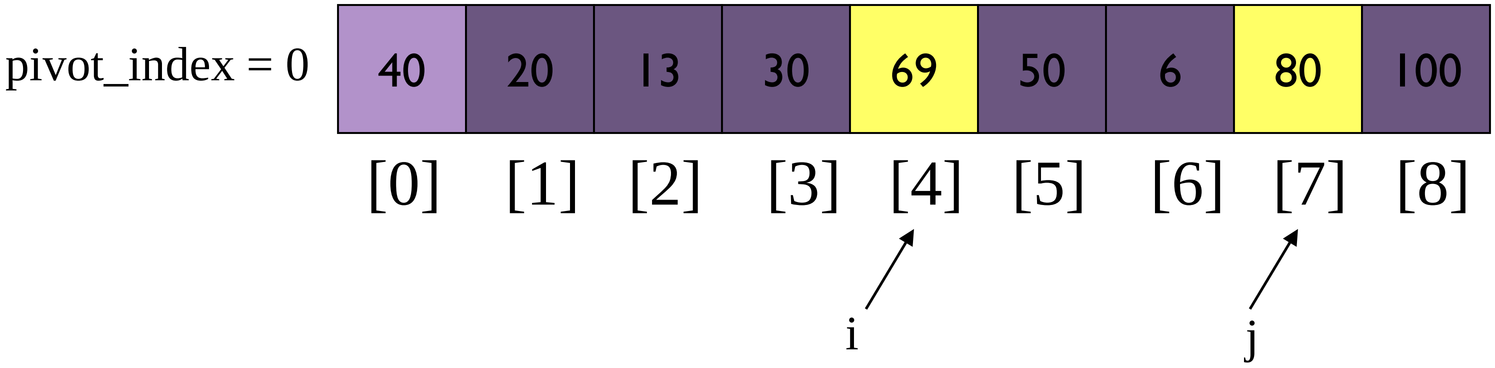
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```

pivot_index = 0

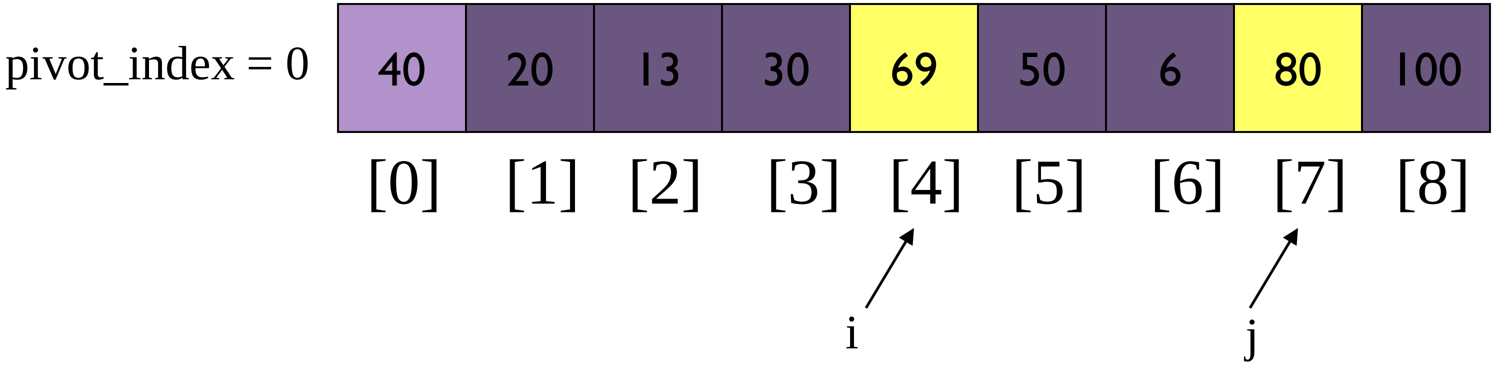


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	30	69	50	6	80	100

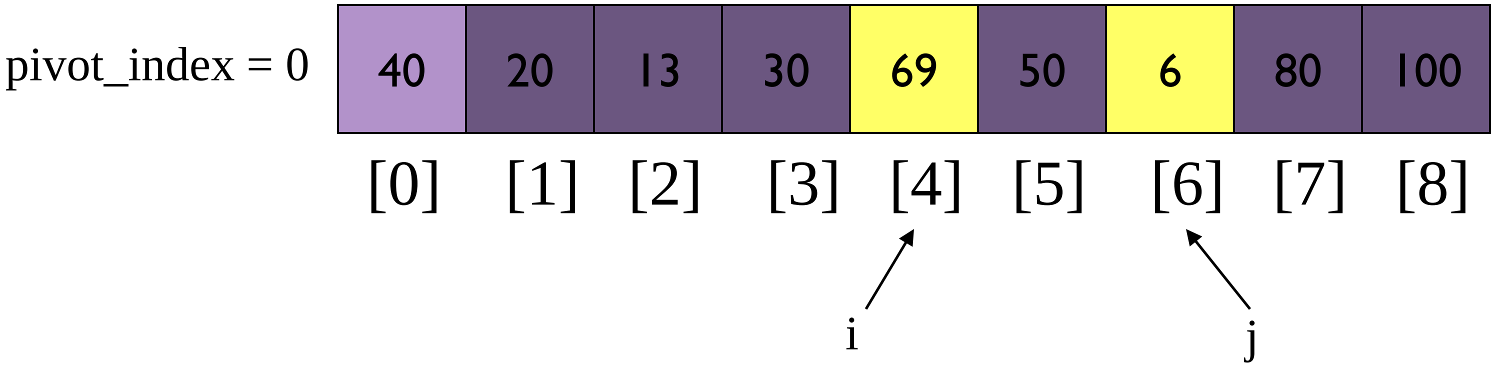
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```



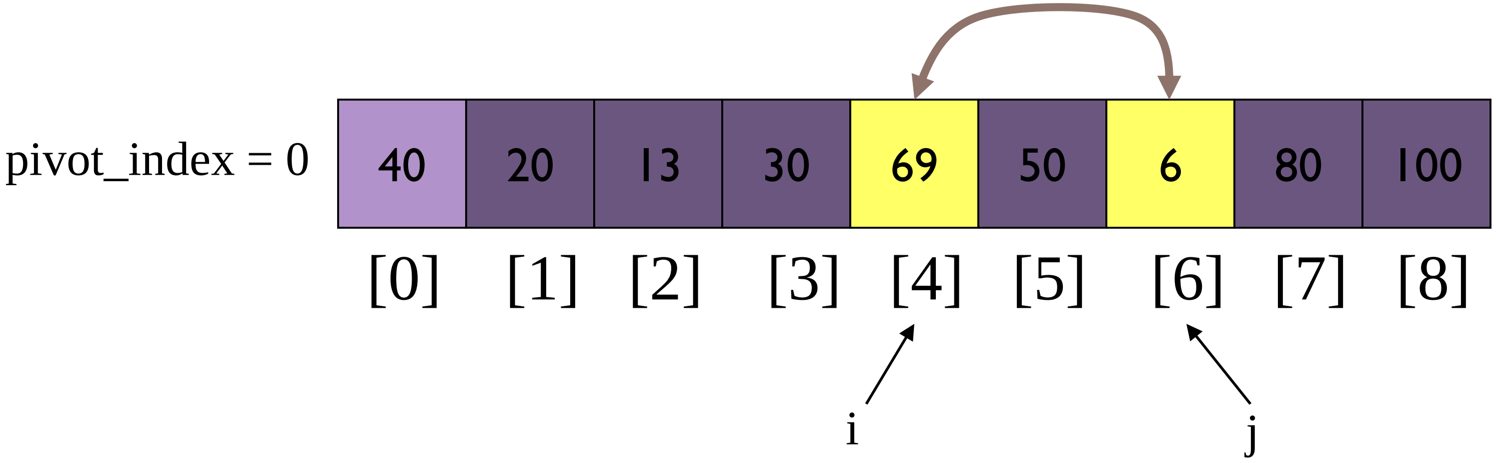
```
   1. while (data[i] <= data[pivot]) i++;
→  2. while (data[j] > data[pivot]) j--;
   3. if(i < j)
          swap data[i] and data[j]
   4. while j > i, go to 1.
```



```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```



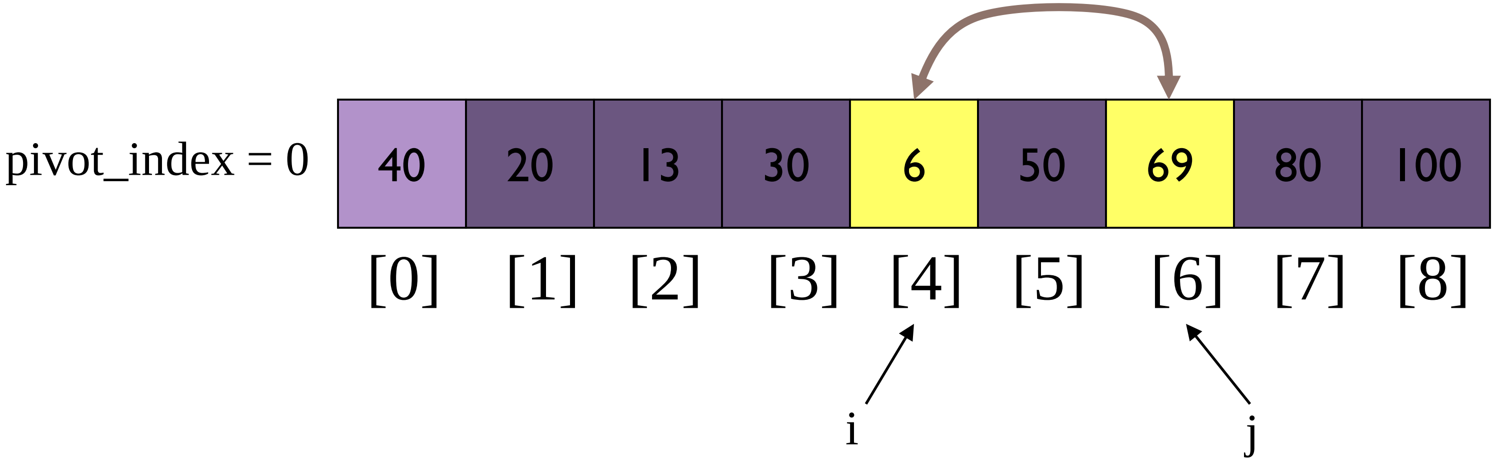
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```



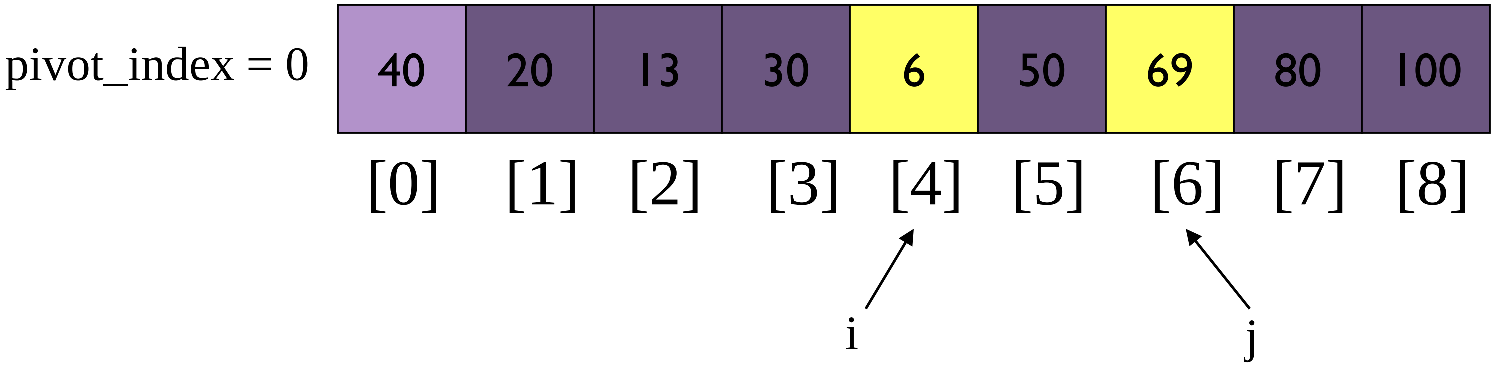
pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	30	69	50	6	80	100

```
  1. while (data[i] <= data[pivot]) i++;
  2. while (data[j] > data[pivot]) j--;
  3. if(i < j)
→         swap data[i] and data[j]
  4. while j > i, go to 1.
```



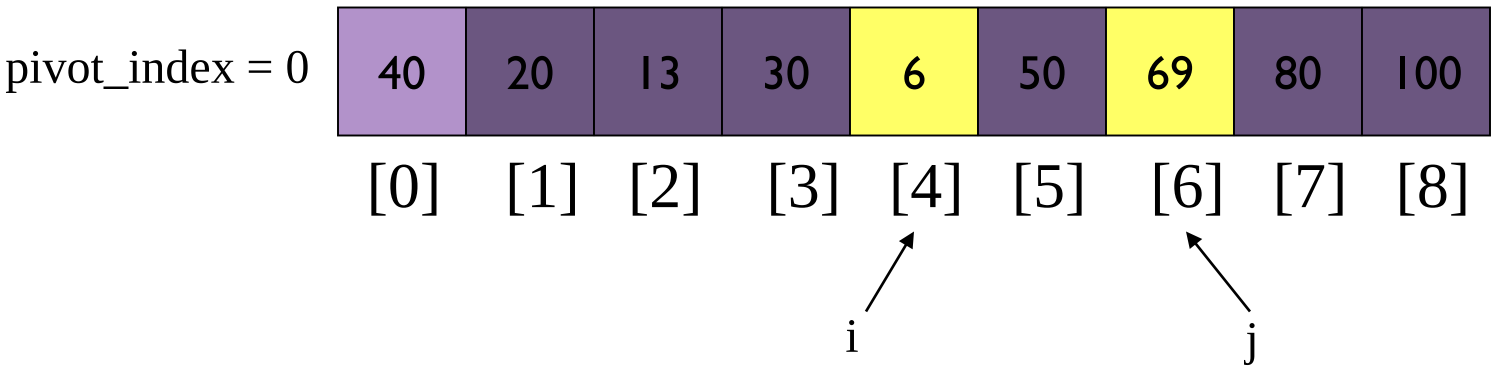
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```



pivot_index = 0

40	20	13	30	6	50	69	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```

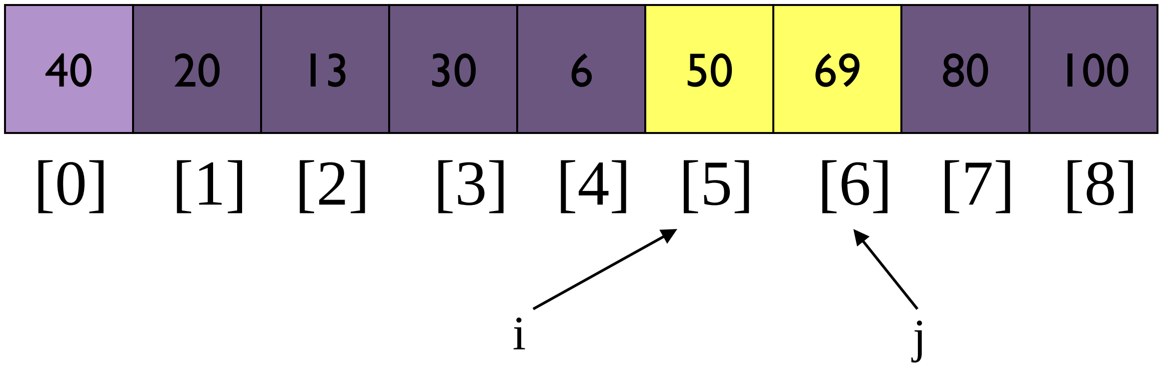


pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	30	6	50	69	80	100

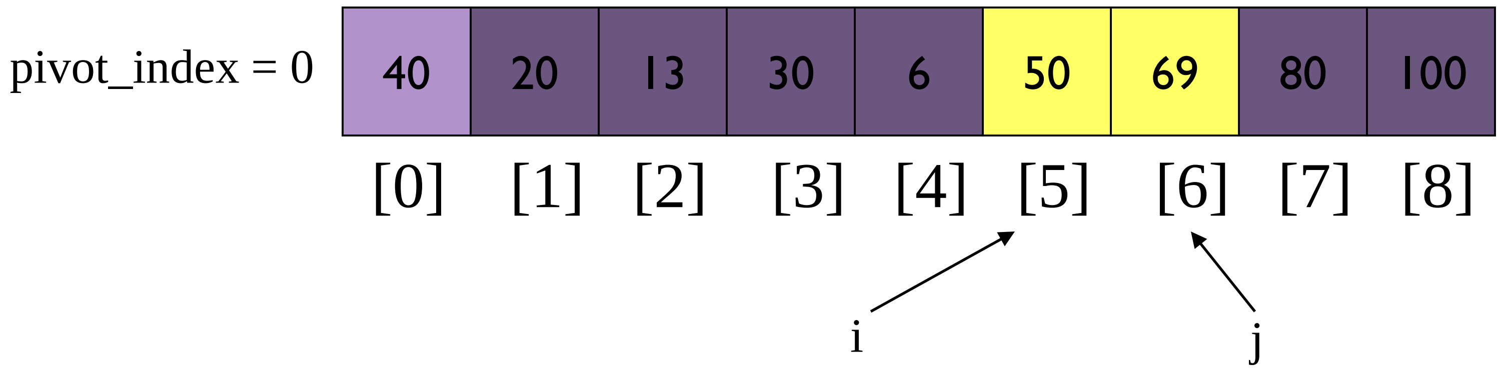
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```

pivot_index = 0



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	30	6	50	69	80	100

```
   1. while (data[i] <= data[pivot]) i++;
→  2. while (data[j] > data[pivot]) j--;
   3. if(i < j)
          swap data[i] and data[j]
   4. while j > i, go to 1.
```

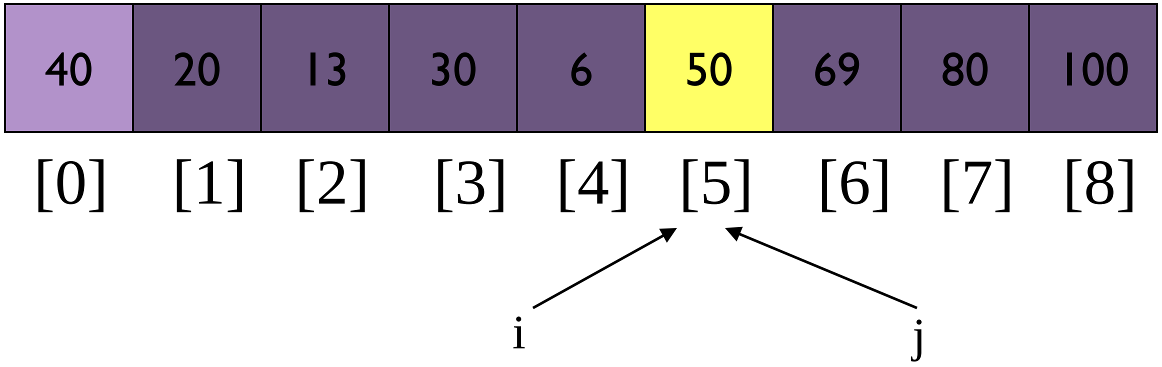


pivot_index = 0

40	20	13	30	6	50	69	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

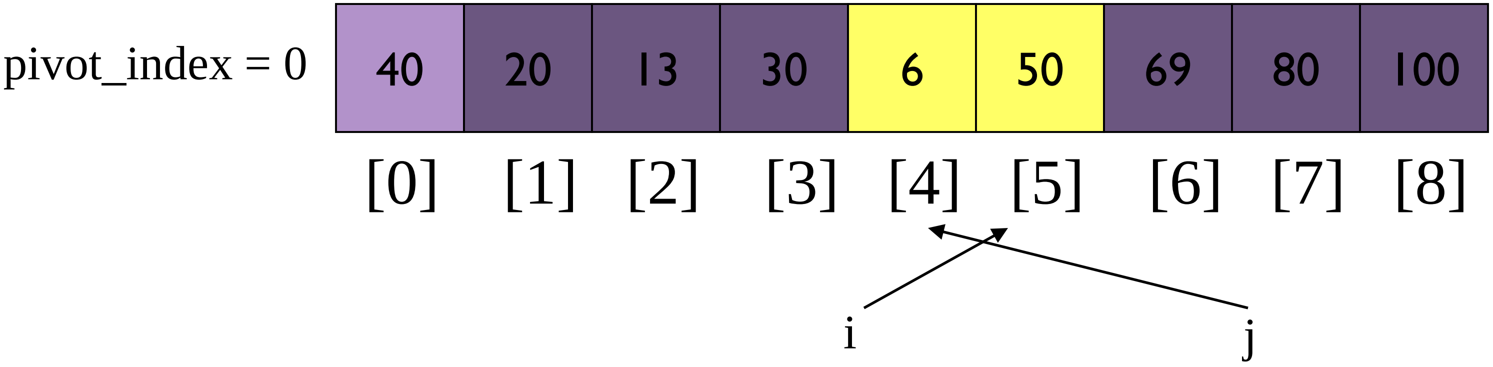
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```

pivot_index = 0



40	20	13	30	6	50	69	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```

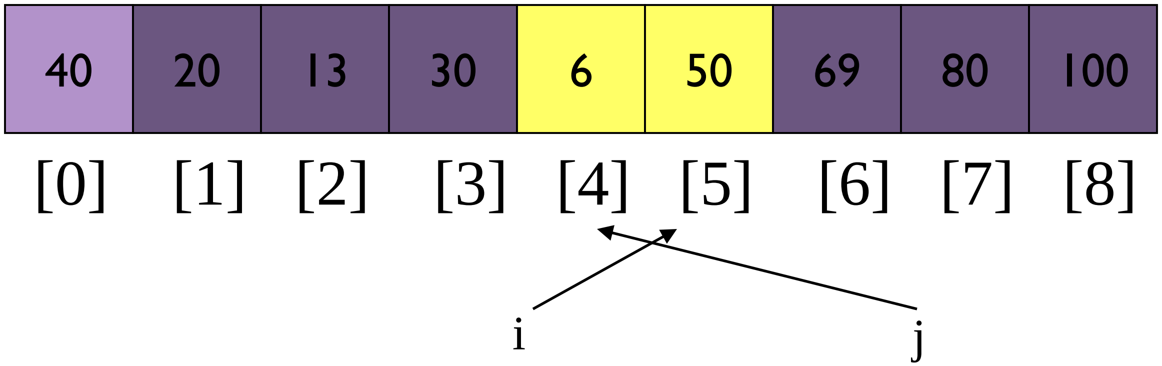


pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	30	6	50	69	80	100

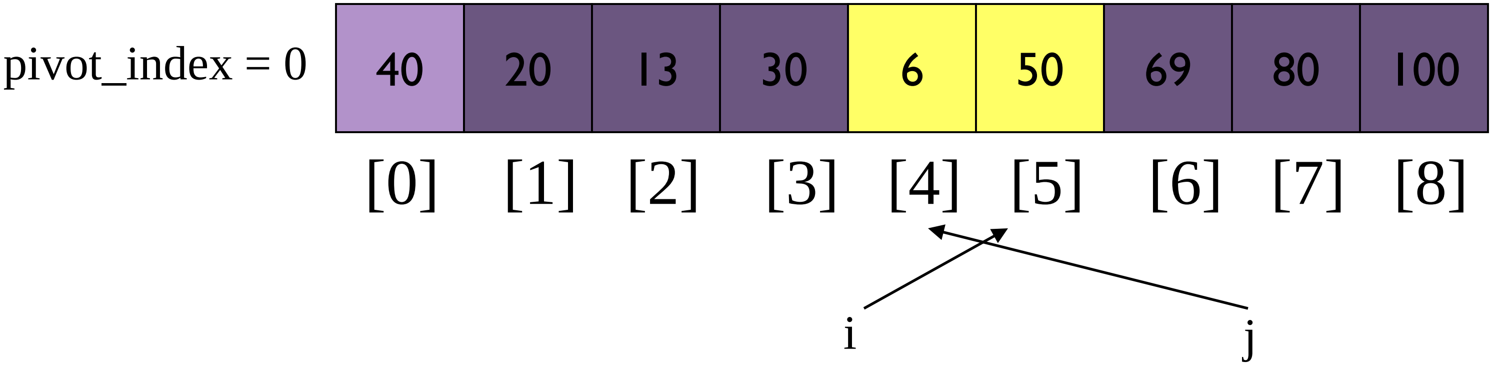
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```

pivot_index = 0



40	20	13	30	6	50	69	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

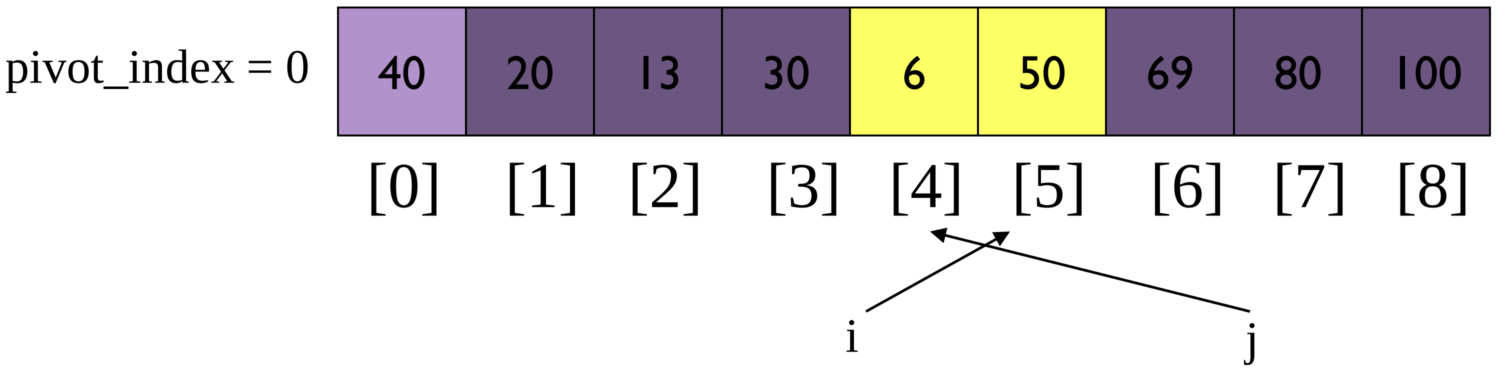
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
```



pivot_index = 0

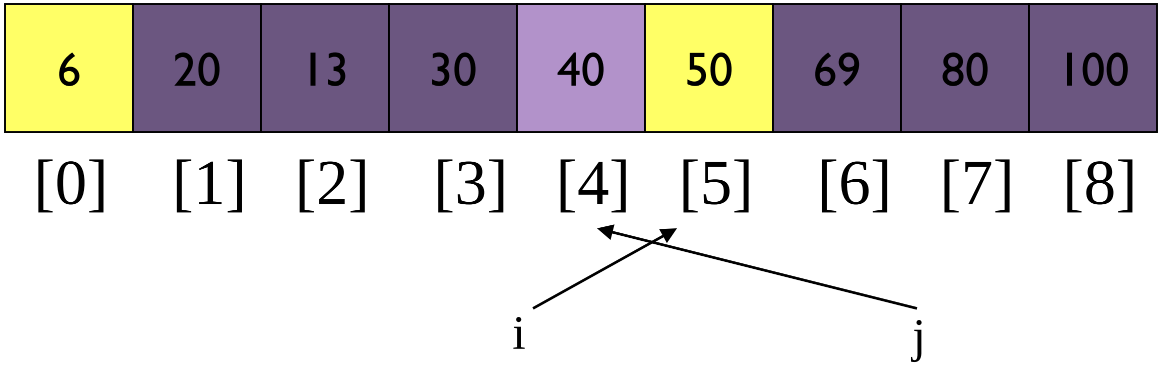
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	13	30	6	50	69	80	100

```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
5. swap data[j] and data[pivot_index]
```



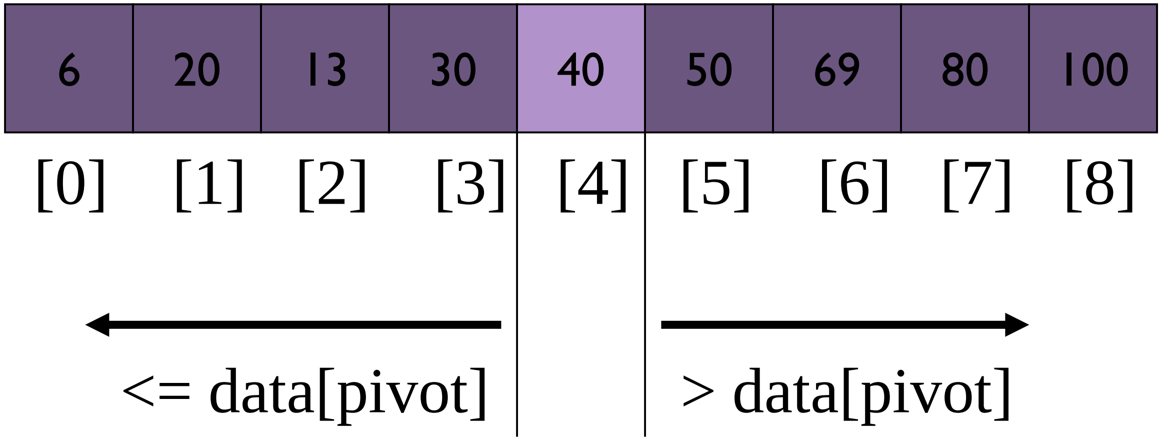
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
5. swap data[j] and data[pivot_index]
```

pivot_index = 4



6	20	13	30	40	50	69	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Partition Result

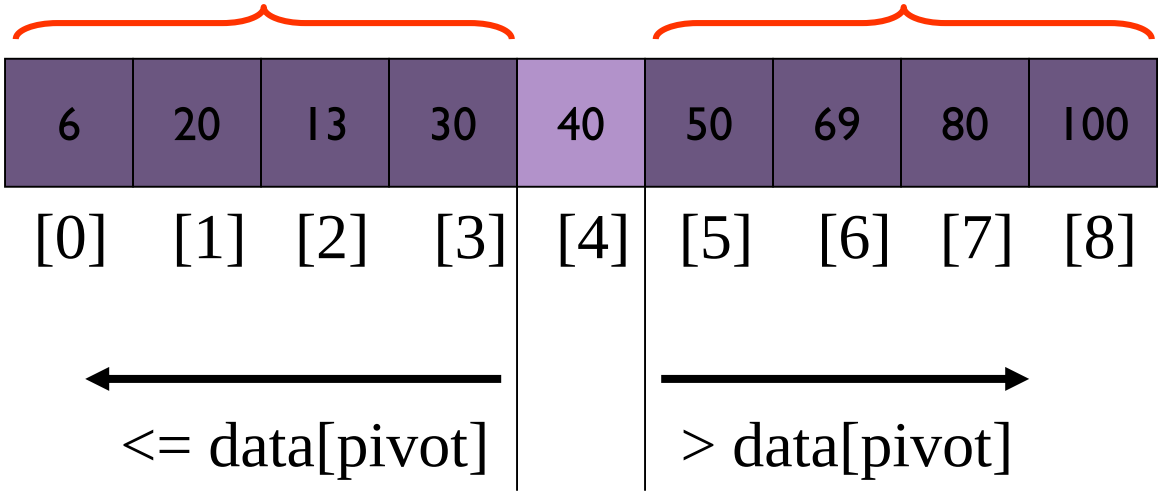


6	20	13	30	40	50	69	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<= data[pivot]

> data[pivot]

Recursion: Quicksort Sub-arrays



6	20	13	30	40	50	69	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<= data[pivot]

> data[pivot]

Quick Sort of N elements: How many comparisons?

N

For first call, when each of N elements is compared to the pivot value

2 * N/2

For the next pair of calls, when N/2 elements in each “half” of the original array are compared to their own pivot values.

4 * N/4

For the four calls when N/4 elements in each “quarter” of original array are compared to their own pivot values.

.

.

.

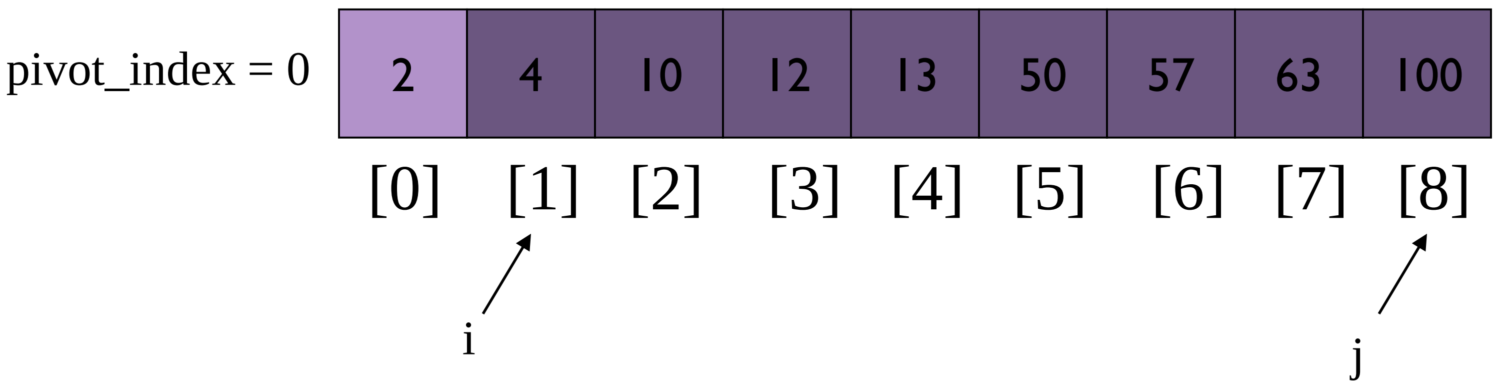
HOW MANY SPLITS CAN OCCUR?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    - Partition splits array in two sub-arrays of size n/2
    - Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Assume first element is chosen as pivot.
  - Assume we get array that is already in order:

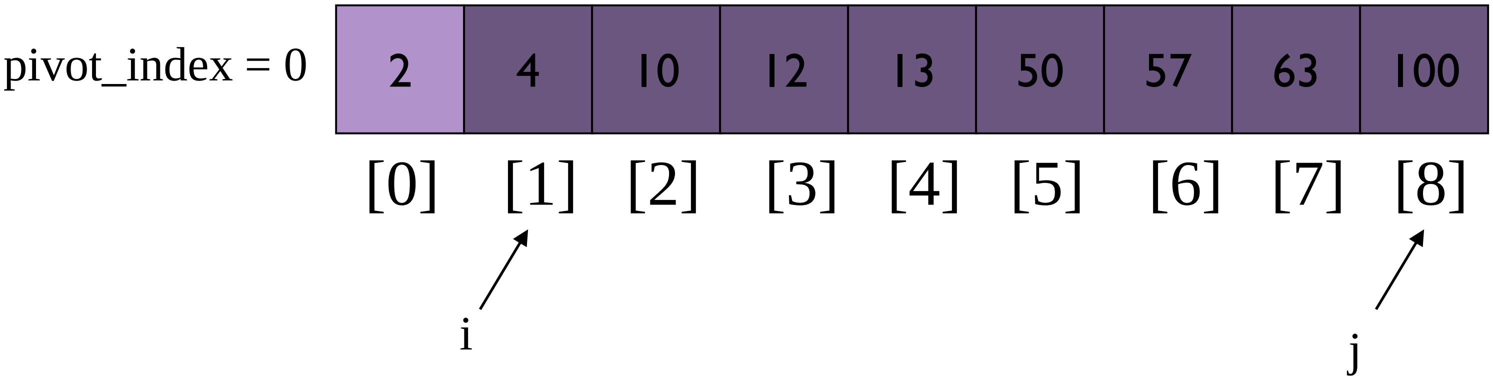


pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

Quicksort Worst Case

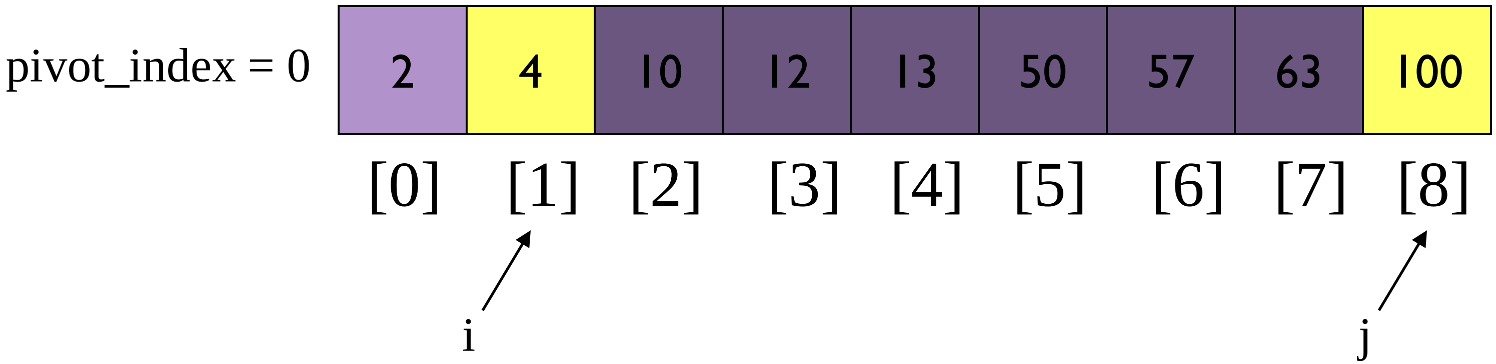
- Assume first element is chosen as pivot.
- Assume we have an array that is already in order:



pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
5. swap data[j] and data[pivot_index]
```



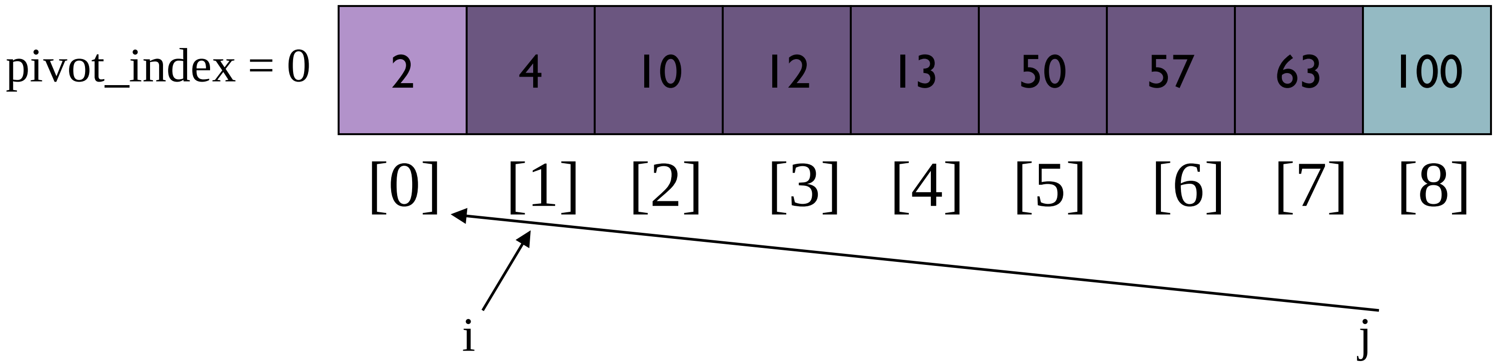
pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

i

j

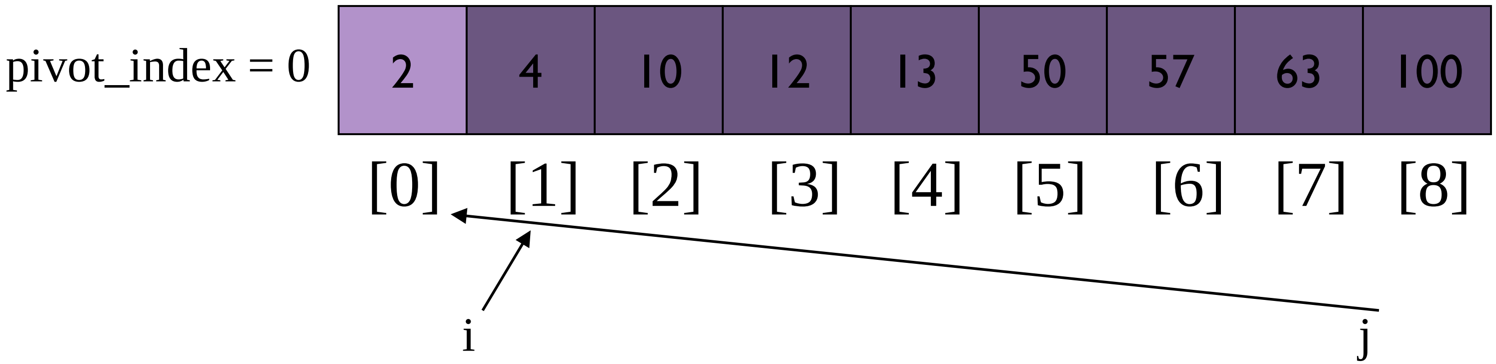
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
5. swap data[j] and data[pivot_index]
```



pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

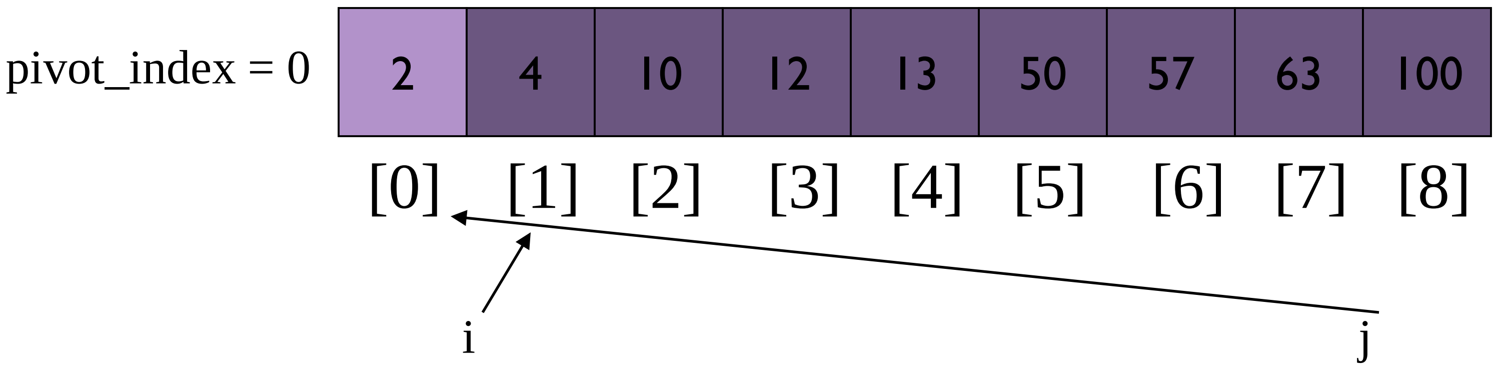
```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
      swap data[i] and data[j]
4. while j > i, go to 1.
5. swap data[j] and data[pivot_index]
```



pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

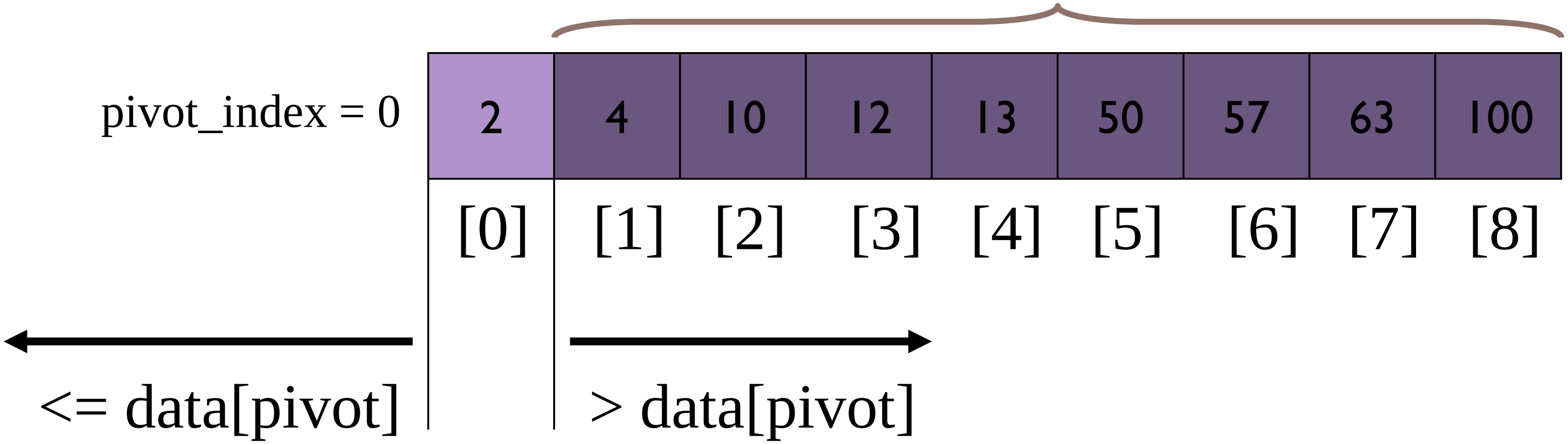
```
   1. while (data[i] <= data[pivot]) i++;
   2. while (data[j] > data[pivot]) j--;
   3. if(i < j)
          swap data[i] and data[j]
   4. while j > i, go to 1.
→  5. swap data[j] and data[pivot_index]
```



pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

```
1. while (data[i] <= data[pivot]) i++;
2. while (data[j] > data[pivot]) j--;
3. if(i < j)
       swap data[i] and data[j]
4. while j > i, go to 1.
5. swap data[j] and data[pivot_index]
```



pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

<= data[pivot]

> data[pivot]

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: O(n $\log_2 n$)
- Worst case running time?
  - Recursion:
    - Partition splits array in two sub-arrays:
      - □ one sub-array of size 0
      - □ the other sub-array of size n-1
    - Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quick Sort of N elements: How many splits can occur?

- It depends on the order of the original array elements!
- If each split divides the sub-array approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N \log_2 N)$.
- But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself. In this case, there can be as many as N-1 splits, and QuickSort is $O(N^2)$.

QuickSort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: O(n $\log_2 n$)
- Worst case running time: O(n^2)!!!
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

data[0], data[n/2], and data[n-1].

Use this median value as pivot.

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply basic force search:
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    - if(data[first] > data[second]) swap them
  - Sub-array of size 3: left as an exercise

Performance Characteristics of Quicksort

Property 1: Quicksort uses about $N^2/2$ comparisons in the worst case.

The number of comparisons used for input that is already in order (sorted) is

$$N + (N-1) + (N-2) + \ldots + 2 + 1 = N(N+1)/2$$

The best case for quicksort is when each partition stage divides the input exactly in half. This would make the number of comparisons used by quicksort satisfy the divide-and-conquer recurrence

$$C_N = 2C_{N/2} + N$$

- The $2C_{N/2}$ covers the cost of sorting the two sub-lists;
- the N is the cost of examining each element, using the partitioning pointer or the other.

Performance Characteristics of Quicksort

Property 2: Quicksort uses about 2N ln N comparisons on the average.

$$C_N = (N-1) + \frac{1}{N}\sum_{k=1}^{N}(C_{k-1} + C_{N-k}) \quad for\ N \geq 2, \qquad C_0 = C_1 = 0$$

- The *N*-1 is the cost of partitioning.
- The rest comes from the observation that each element *k* is likely to be the partitioning element with probability 1/*N*, after which we are left with two random partitions of size *k* - 1 and *N* – *k*.

Reference

- Part of this slide set is prepared or/and extracted from the following references:
  - Data Structures & Algorithm Analysis in Java; Clifford A. Shaffer; Dover Publications Inc.; 3rd edition, 2011 – Chapter 7
  - Algorithms in C++; Robert Sedgewick; Addison-Wesely Publishing Company Inc.; 3rd edition, 1998 – Chapter 7