1 SORTING ALGORITHMS

- In-Place vs. Not In-Place:
  - **In-Place**: Uses a constant amount of extra space (does not require additional memory proportional to the input size).
  - **Not In-Place**: Requires additional memory proportional to the input size.

- Stable vs. Unstable:
  - **Stable**: Preserves the relative order of elements with equal values.
  - **Unstable**: May change the relative order of elements with equal values.

Sorting Algorithm	In-Place	Not In-Place	Stable	Unstable
Selection Sort	Yes	No	No	Yes
Insertion Sort	Yes	No	Yes	No
Quick Sort	Yes	No	No	Yes
Merge Sort	No	Yes	Yes	No
Shell Sort	Yes	No	No	Yes

SELECTION SORT

- Selection sort is one of the easiest approaches to sorting.
- It is inspired from the way in which we sort things out in day to day life.
- It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

HOW IT WORKS?

- Find the smallest element in the array.
- Swap with the first element of the unordered list.
- Finds the next smallest element in the array.
- Swap with the second element of the unordered list.
- Continue till all elements are sorted.

SELECTION SORT

5	1	3	4	6	2

Comparison

Data Movement

Sorted

SELECTION SORT

5	**1**	**3**	**4**	**6**	**2**

minIndex =0

SELECTION SORT

5	1	3	4	6	2

1 < 5

minIndex =1

SELECTION SORT

5	1	3	4	6	2

minIndex = 1

SELECTION SORT

5	1	3	4	6	2

minIndex = 1

SELECTION SORT

5	1	3	4	6	2

minIndex = 1

SELECTION SORT

5	1	3	4	6	2

minIndex = 1

SELECTION SORT

5	1	3	4	6	2

↑

Smallest

SELECTION SORT

1	5	3	4	2	6

SELECTION SORT

1	5	3	4	2	6

SELECTION SORT

1	5	3	4	2	6

minIndex = 1

SELECTION SORT

1	5	3	4	2	6

minIndex = 2

SELECTION SORT

1	5	3	4	2	6

minIndex = 2

SELECTION SORT

1	5	3	4	2	6

minIndex = 4

SELECTION SORT

1	5	3	4	2	6

↑ **Smallest**

minIndex = 4

SELECTION SORT

1	2	3	4	5	6

Comparison

Data Movement

Sorted

SELECTION SORT

1	2	3	4	5	6

minIndex = 2

SELECTION SORT

1	2	3	4	5	6

minIndex = 2

SELECTION SORT

1	2	3	4	5	6

minIndex = 2

SELECTION SORT

1	2	3	4	5	6

minIndex = 2

SELECTION SORT

1	2	3	4	5	6

↑
Smallest

minIndex = 2

SELECTION SORT

1	2	3	4	5	6

DONE!

PSEUDOCODE

```
SelectionSort(array)
    n = length of array
    for i = 0 to n - 1 do
        minIndex = i
        for j = i + 1 to n - 1 do
            if array[j] < array[minIndex] then
                minIndex = j
        if minIndex ≠ i then
            swap array[i] with array[minIndex]
```

ANALYSIS OF SELECTION SORT

Time Complexity Analysis-

- Selection sort has nested loops that both depend on the size of input n
- $O(n^2)$ - time complexity in Both Best and Worst Case

Space Complexity Analysis-

- Selection sort is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- Space complexity is O(1).

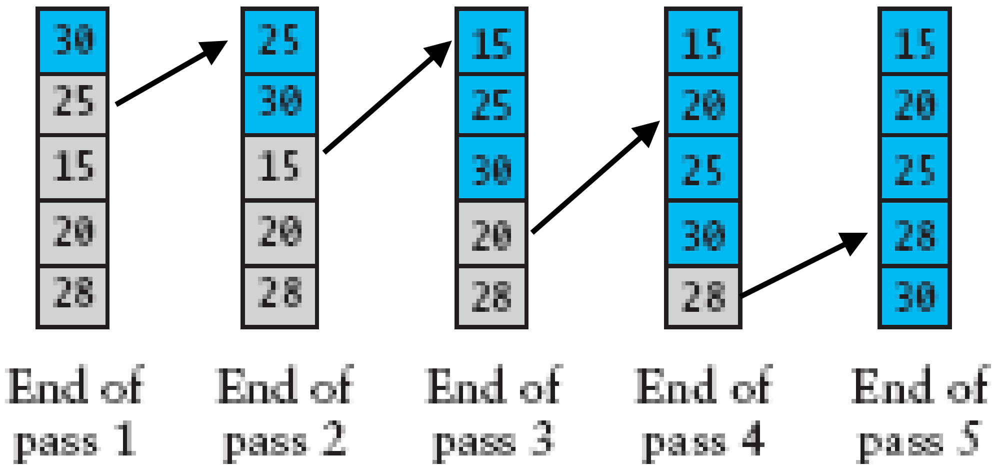
INSERTION SORT

- Based on technique of card players to arrange a hand
  - Player keeps cards picked up so far in *sorted order*
  - When the player picks up a new card
    - Makes room for the new card
    - Then inserts it in its proper place

FIGURE 10.3
Picking Up a Hand of Cards

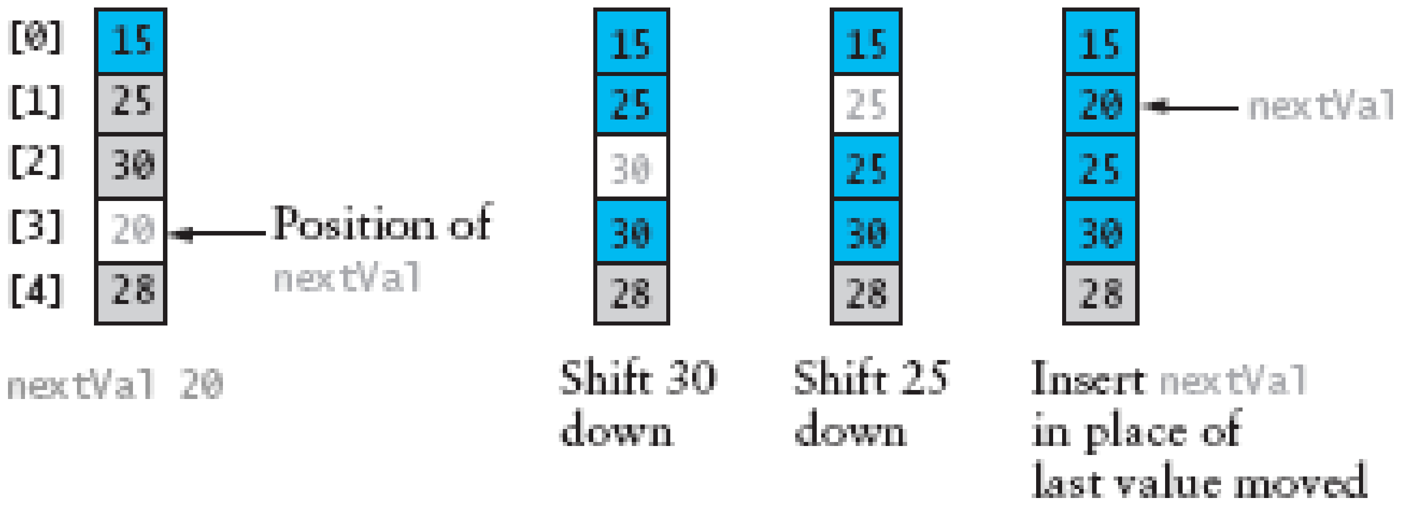
INSERTION SORT

FIGURE 10.4
An Insertion Sort



INSERTION SORT

FIGURE 10.5
Inserting the Fourth Array Element



PSEUDOCODE

```
InsertionSort(array)
    n = length of array
    for i = 1 to n - 1 do
        key = array[i]
        j = i - 1
        while j >= 0 and array[j] > key do
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key
```

ANALYSIS OF INSERTION SORT

- Best Case : The array is already sorted.
  - Each element is only compared with its previous position element.
  - In this case Insertion sort runs in linear time : O(n)

- Worst Case: The array is in decreasing order.
  - The n-1th element needs n-1 swap
  - The n-2nd element needs n-2 swaps and so on
  - Using summation, in this case the algorithm runs O(n^2)

- Insertion Sort is also an in-place sort
  - Since no auxillary storage is used space somplexity is O(1)

QUICKSORT

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that Merge Sort does
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

QUICKSORT - STEPS

```
QuickSort(A)
    if length(A) ≤ 1 then
        return A

    p = pick a pivot value from A

    A1 = {all values x < p}
    A2 = {all values x > p}
    A1 and A2 are disjoint sets
    P = {all values x = p}  # All values equal to the pivot

    return QuickSort(A1) + P + QuickSort(A2)
```

QUICKSORT - PARTITIONING

- Partition the array into left and right sub-arrays
  - left sub-array -> elements <= pivot
  - right sub-array are -> elements > pivot
- Getting the elements to the correct sub-array:
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions

IN-PLACE PARTITIONING

References **`front`** and **`back`** to start and end of array

Increment **`front`** until -> **`A[front] > pivot`**

Decrement **`back`** until -> **`A[back] < pivot`**

Swap **`A[front]`** and **`A[back]`**

Repeat until **`front`** and **`back`** cross

Swap pivot with **`A[back]`**

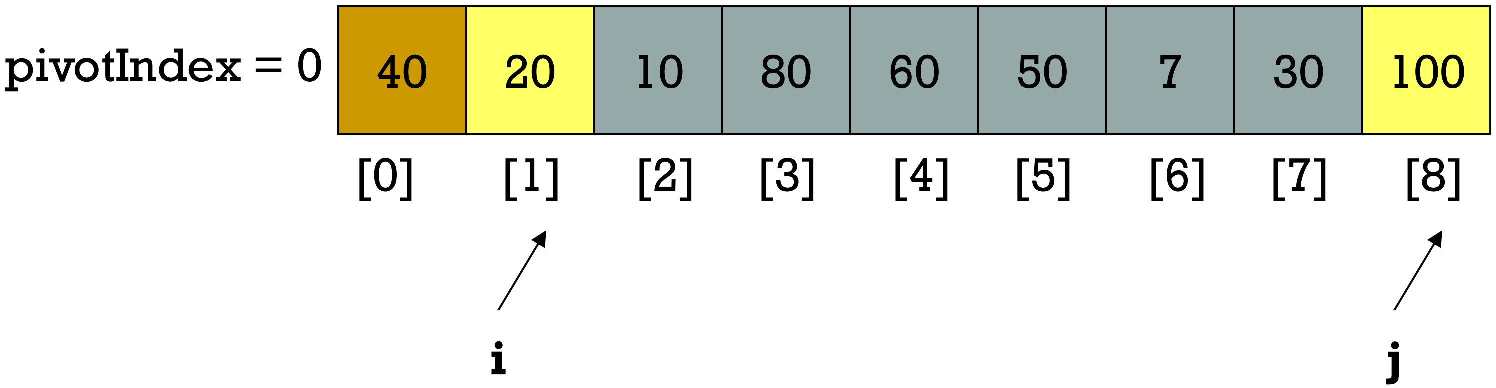
swapIndex = 0
endIndex = 8

pivotIndex = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

i **j**

1. While array[i] <= array[pivotIndex]
   ++i;



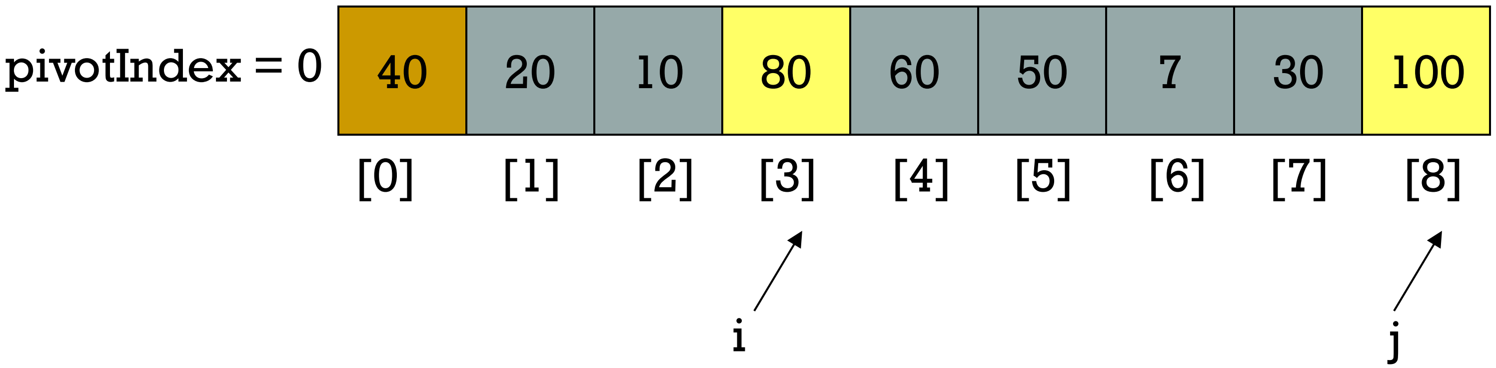
1. While array[i] <= array[pivotIndex]
   ++i

pivotIndex = 0	40	20	10	80	60	50	7	30	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
			i						j

1. While array[i] <= array[pivotIndex]
   ++i

pivotIndex = 0	40	20	10	80	60	50	7	30	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
				i					j

1. While array[i] <= array[pivotIndex]
    ++i
2. While array[j] > array[pivotIndex]
    --j

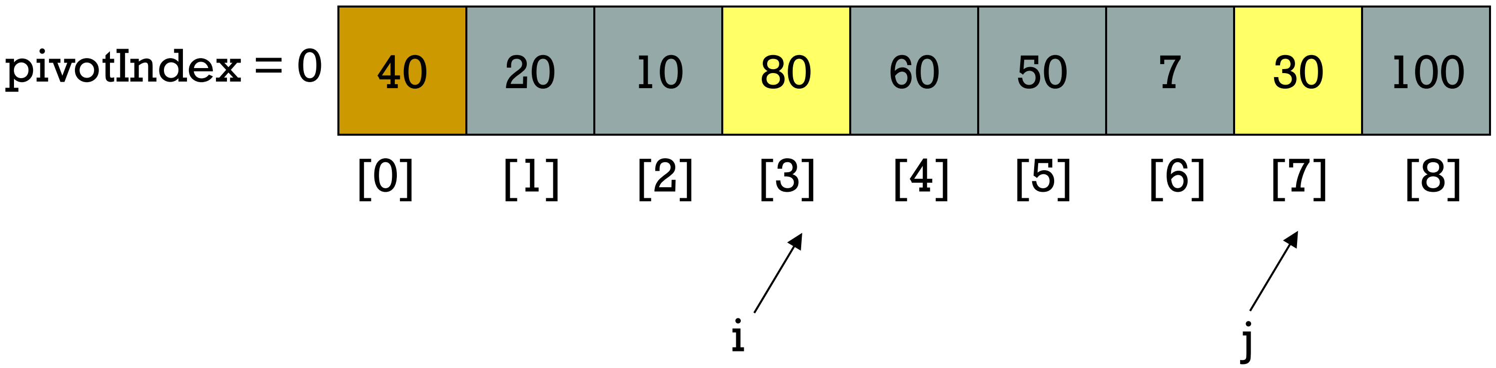


1. While array[i] <= array[pivotIndex]

    ++i

2. While array[j] > array[pivotIndex]

    --j



1. While array[i] <= array[pivotIndex]

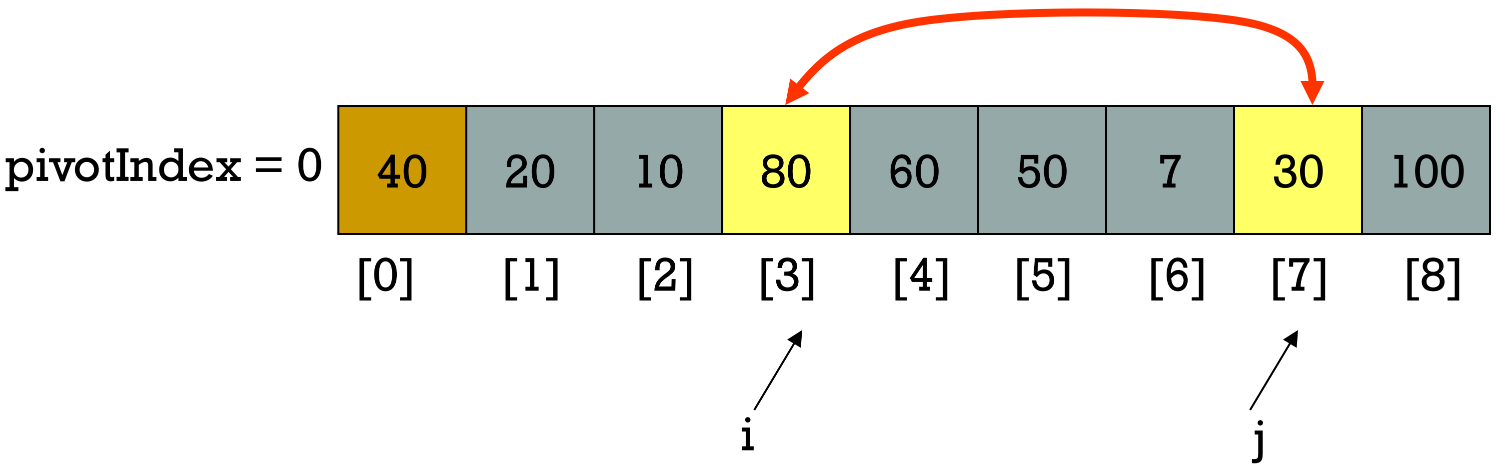
    ++i

2. While array[j] > array[pivotIndex]

    --j

3. If i < j

    swap array[i] and array[j]



1. While array[i] <= array[pivotIndex]

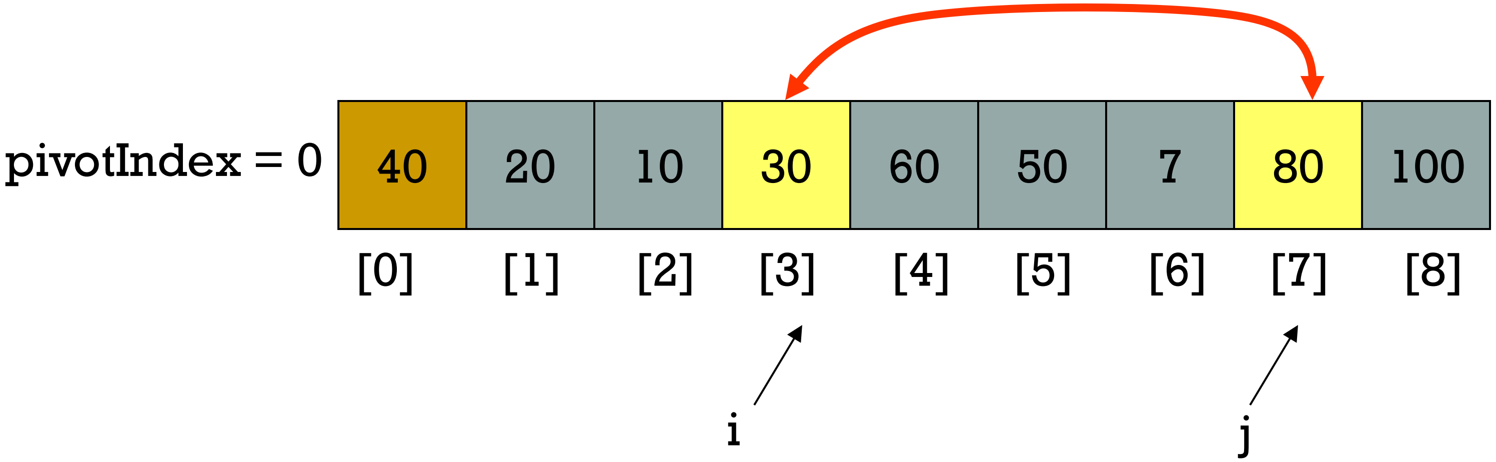
    ++i

2. While array[j] > array[pivotIndex]

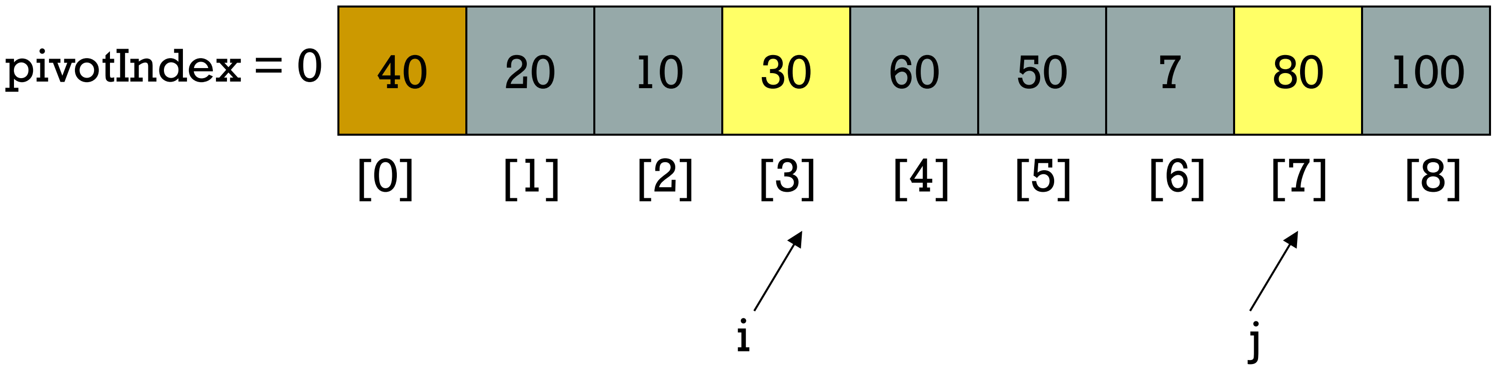
    --j

3. If i < j

    swap array[i] and array[j]



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



1. While array[i] <= array[pivotIndex]

   ++i

2. While array[j] > array[pivotIndex]

   --j

3. If i < j

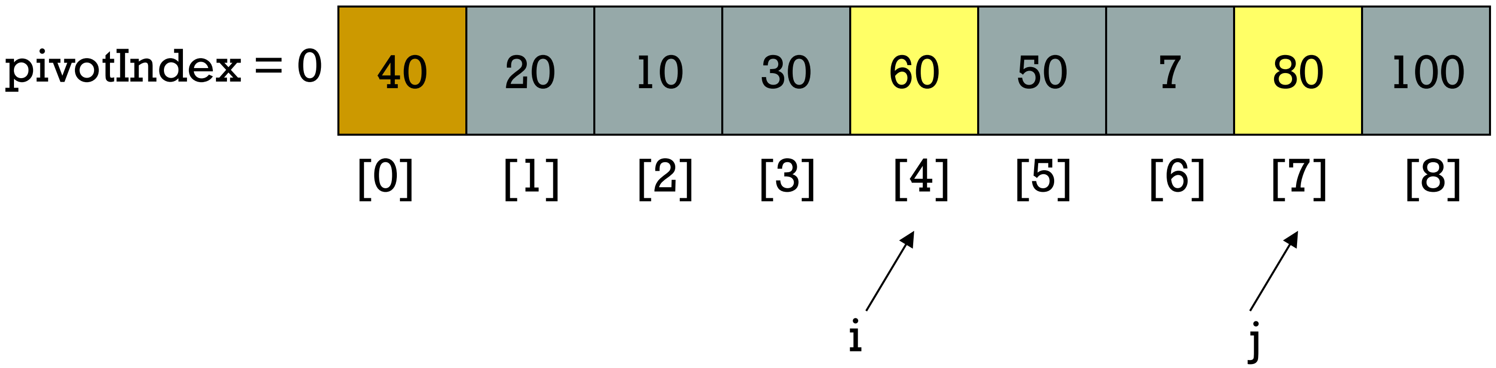
   swap array[i] and array[j]

4. While j > i, go to 1.

pivotIndex = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
			i				j	

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



1. While array[i] <= array[pivotIndex]

    ++i

2. While array[j] > array[pivotIndex]

    --j

3. If i < j

    swap array[i] and array[j]

4. While j > i, go to 1.

pivotIndex = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100
				i			j	

1. While array[i] <= array[pivotIndex]

   ++i

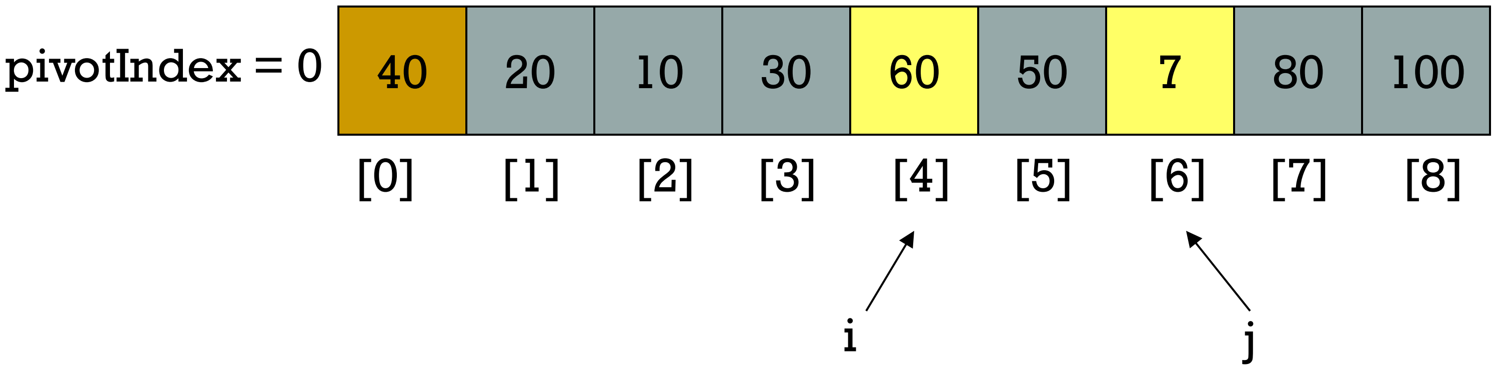
2. While array[j] > array[pivotIndex]

   --j

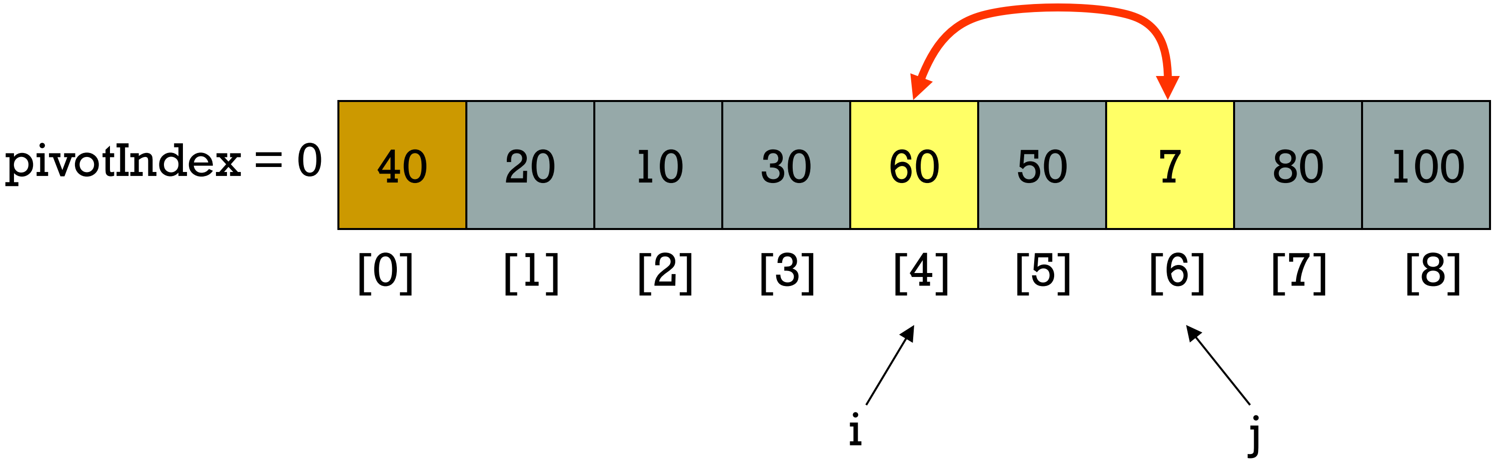
3. If i < j

   swap array[i] and array[j]

4. While j > i, go to 1.



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



1. While array[i] <= array[pivotIndex]

    ++i

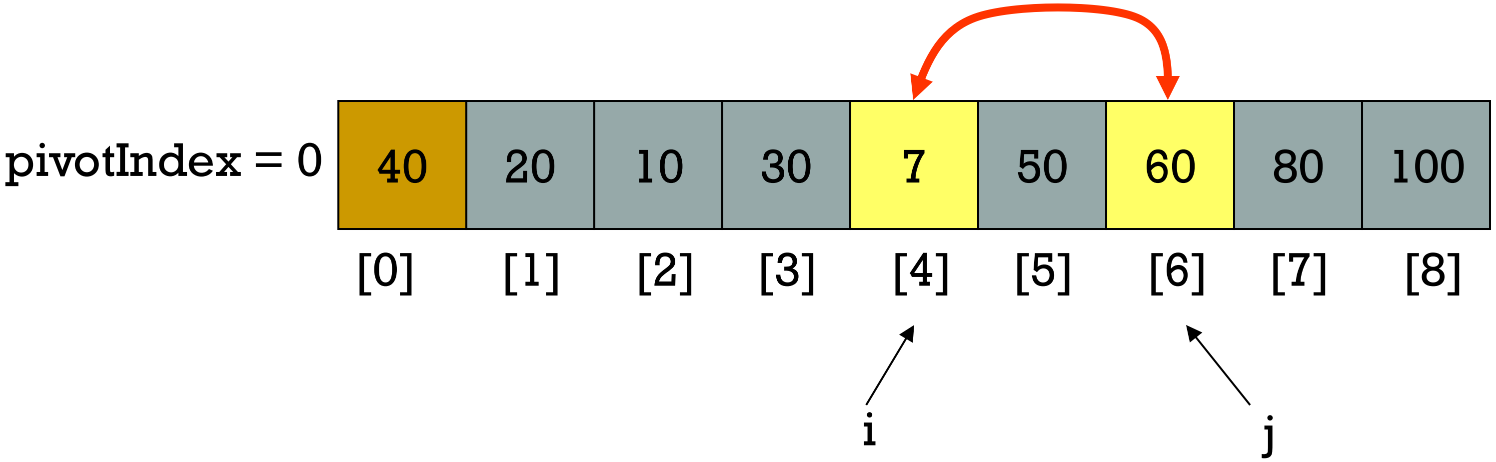
2. While array[j] > array[pivotIndex]

    --j

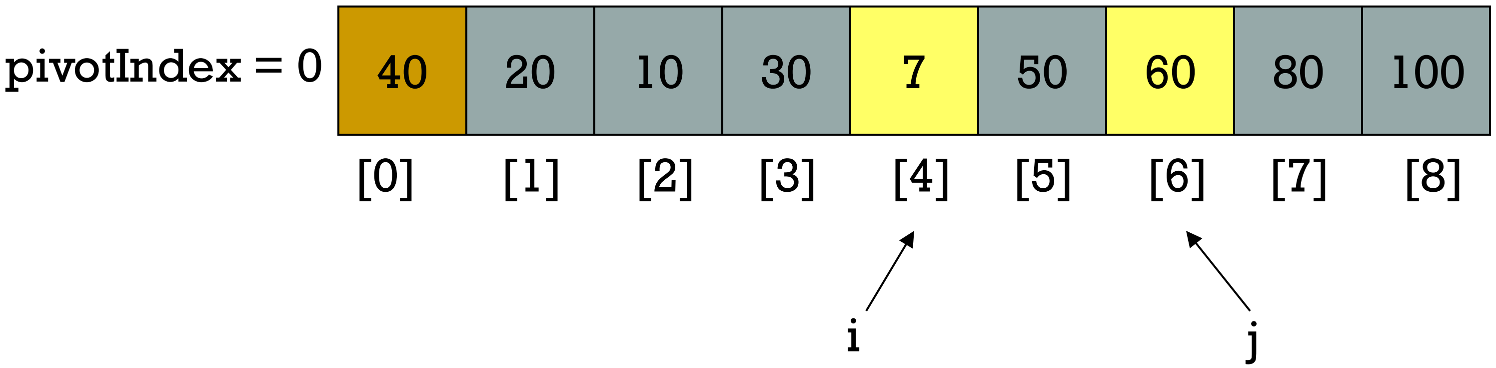
3. If i < j

    swap array[i] and array[j]

4. While j > i, go to 1.

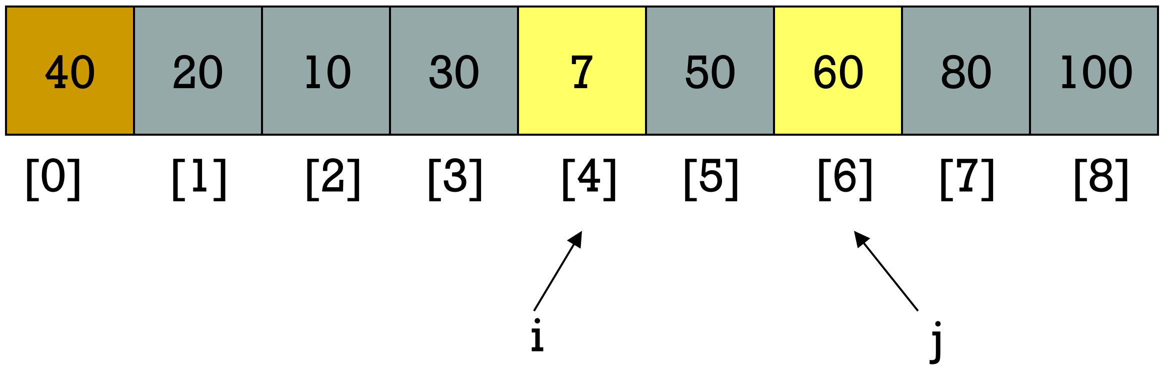


1. While array[i] <= array[pivotIndex]
   ++i
2. While array[j] > array[pivotIndex]
   --j
3. If i < j
   swap array[i] and array[j]
4. While j > i, go to 1.



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

pivotIndex = 0



1. While array[i] <= array[pivotIndex]

   ++i

2. While array[j] > array[pivotIndex]

   --j

3. If i < j

   swap array[i] and array[j]

4. While j > i, go to 1.

pivotIndex = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

i

j

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

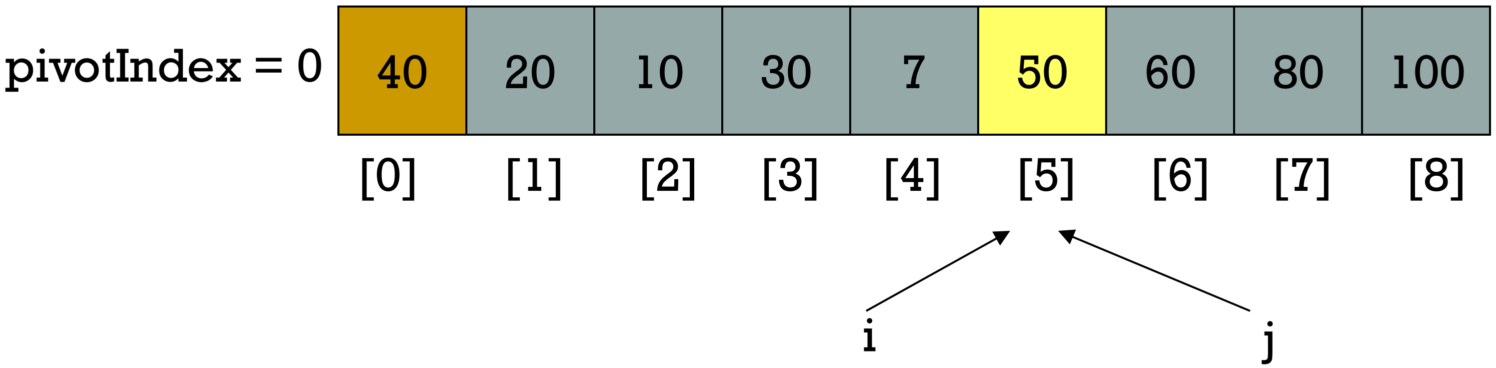
pivotIndex = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

i

j

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



1. While array[i] <= array[pivotIndex]

    ++i

2. While array[j] > array[pivotIndex]

    --j

3. If i < j

    swap array[i] and array[j]

4. While j > i, go to 1.

pivotIndex = 0	40	20	10	30	7	50	60	80	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

i

j

1. While array[i] <= array[pivotIndex]

   ++i

2. While array[j] > array[pivotIndex]

   --j

3. If i < j

   swap array[i] and array[j]

4. While j > i, go to 1.

pivotIndex = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

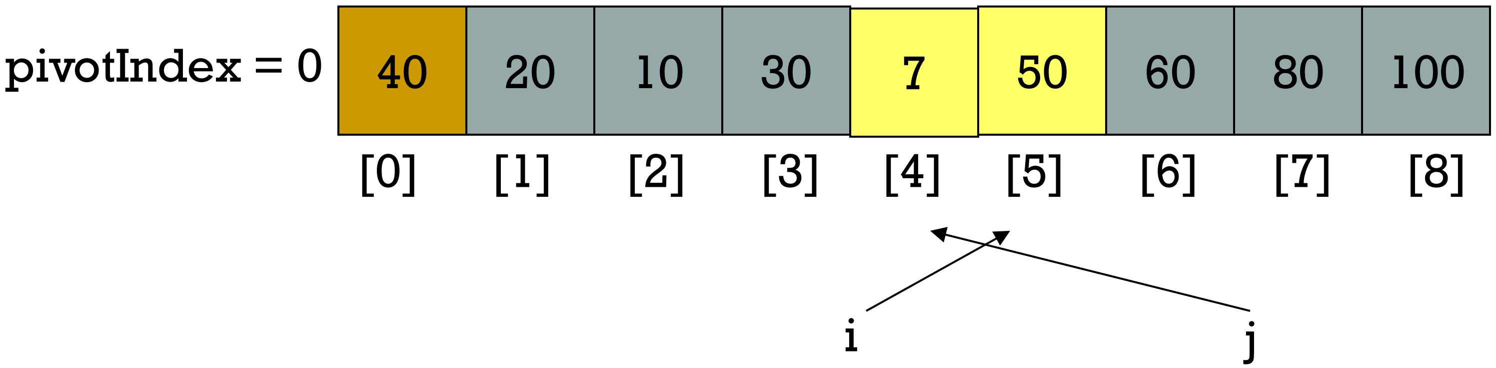
i j

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

pivotIndex = 0	40	20	10	30	7	50	60	80	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

i j

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



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

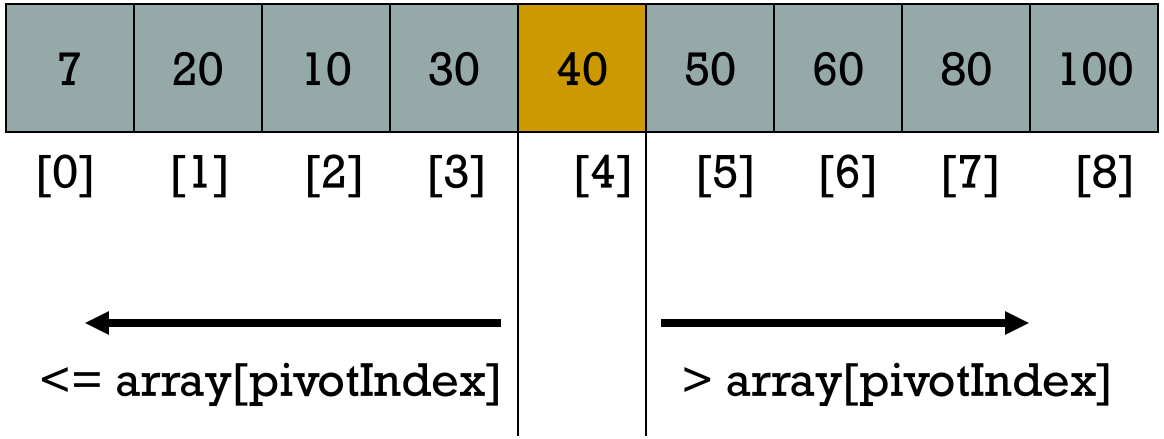
pivotIndex = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
7	20	10	30	40	50	60	80	100

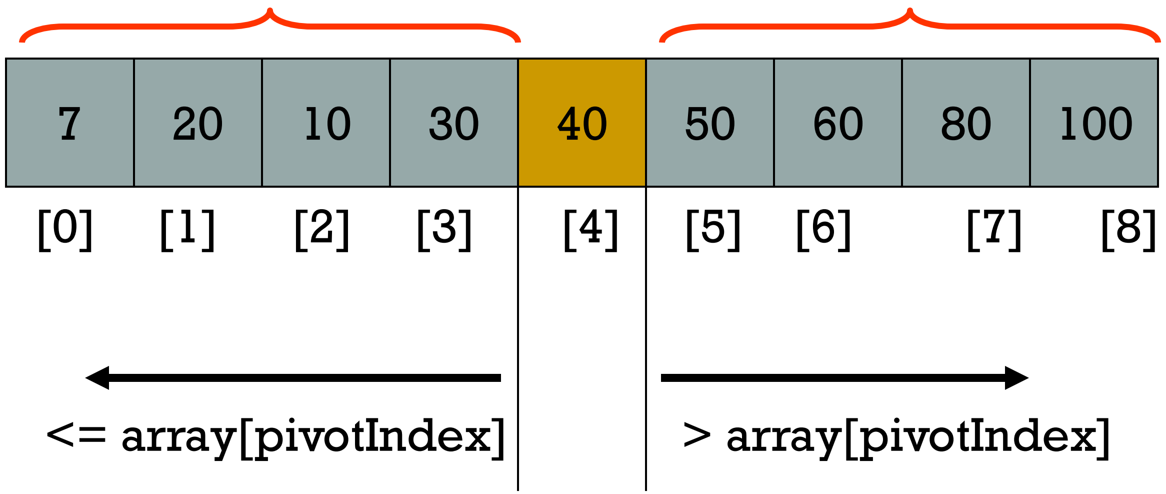
i

j

PARTITION RESULT



RECURSION: QUICKSORT SUB-ARRAYS



ANALYSIS OF QUICKSORT – BEST CASE

For each sub array pivot is placed approximately in the middle of the array.

Array size n -> 2 subarrays ~ n/2 -> 4 subarrays ~ n/4 -> 8 subarrays ~ n/8

⇒n subarrays of size 1

n elements are repeatedly dived in half approximately $\log_2 n$ times

After splitting the array $\log_2 n$ times we get n subarrays of size 1

Depth of recursion : O(logn)

Number of accesses in partition : O(n)

=>O(n logn)

ANALYSIS OF QUICKSORT – WORST CASE

- If Pivot is the smallest or largest element of the list
  - One empty subarray and other has n-1 elements

- Worst case is when this happens for every subarray

- If the first element is pivot :
  - This worse case occurs when the list is already sorted / is in descending order.

WORST CASE - CONTD

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 : O(n)

Number of accesses in partition : O(n)

=> $O(n^2)$

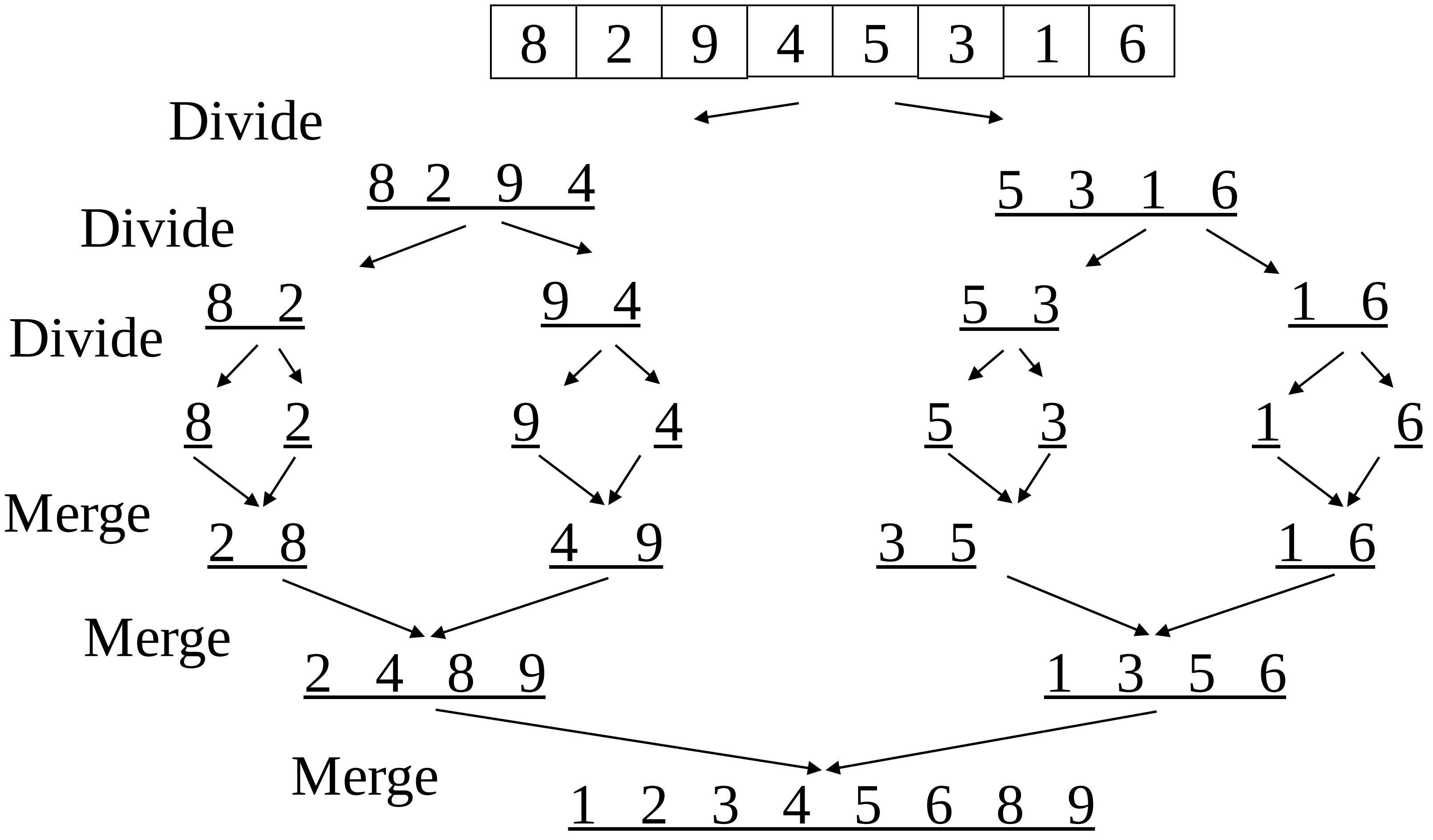
Quicksort is unstable and In-place

MERGE SORT

- Divide the list into two subarrays of almost equal size
- Sort the left subarray using merge sort
- Sort the right subarray using merge sort
- Merge the two sorted subarrays, in the order of splitting

- Terminating condition for recursion – subarray contains only one element

MERGE SORT - EXAMPLE



ANALYSIS OF MERGE SORT

- N elements are repeatedly divided in half approximately $\log_2 n$ times
- After splitting the array $\log_2 n$ times we get n sub arrays of size 1
- In each pass -> We merge n elements hence -> O(n)
- Therefore the performance of merge sort is O(nlogn) -> Best and Worst Case

- Stable sort
- Not an in-place sort -> Needs an auxiliary array to store n elements -> O(n)

SHELL SORT

- Named after Donald Shell.
- Also known as diminishing Increment Sort
- Problem with Insertion sort :
  - We have to move many items in the sorted part of the array to insert a new element.
  - The problem arises when we have to move a smaller element from the back of the array to the first.
- Divide and conquer approach to insertion sort
  - Sort many smaller subarrays using insertion sort
  - Sort progressively larger arrays
  - Finally sort the entire array
- Shell sort works by comparing elements that are distant rather than adjacent.
- These arrays are elements separated by a *gap*
  - Start with *large gap*
  - Decrease the gap on each "pass"

SHELL SORT

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

}

SHELL SORT – CHOOSING INCREMENTS

- Shell's suggestion :
  - arrayLength/2 initially then decrementing by 2 every pass
  - The elements at the odd places and even places are not compared until the last pass.

- Increments that are multiples of each other 1,3,6,9   1,2,4,8
  - In this case the same elements get compared over and over again till the last pass.
- Increments that are relatively prime are a good choice
- Knuth Sequence : h= 3*h + 1
  - ```
    int h = 1;
        while (h < numbers.length / 3)
            h = 3 * h + 1;
    ```
  - Decrement : h= (h-1) /3
    ```

SHELL SORT - ANALYSIS

- Its general analysis is an open research problem
- Performance depends on sequence of gap values
- For sequence 2^k, performance is $O(n^2)$
- Hibbard's sequence (2^k-1), performance is $O(n^{3/2})$
- Knuth's sequence, performance is $O(n^{3/2})$

For Shell sort, the running time is dependent on number of increments and their values.

Shell sort is *Unstable* and an *In-place Sort*

SHELL SORT - ANALYSIS

- Elements move long distances at a time, elements move to its final place quicker.

- Insertion sort is efficient :
  - When list is small
  - When list is almost sorted

- When increments are large – Size of the sublists are smaller
- When increments are small – Sublists are large, but they are almost sorted.