Sorting Techniques

Prof. Pujashree Vidap

Objectives

- In this session, you will learn to:
  - Identify the algorithms that can be used to sort data
  - Sort data by using bubble sort
  - Sort data by using selection sort
  - Sort data by using insertion sort
  - Sort data by using shell sort
  - Sort data by using Quick Sort

Sorting Data (Contd.)

- Sorting is the process of arranging data in some pre-defined order or sequence. The order can be either ascending or descending.
- If the data is ordered, you can directly go to that section , thereby reducing the number of records to be traversed.

Sorting data by Using Bubble Sort

- Bubble sort algorithm:
  - Is one of the simplest sorting algorithms
  - Has a quadratic order of growth and is therefore suitable for sorting small lists only
  - Works by repeatedly scanning through the list, comparing adjacent elements, and swapping them if they are in the wrong order

Implementing Bubble Sort Algorithm

- To understand the implementation of bubble sort algorithm, consider an unsorted list of numbers stored in an array.

	0	1	2	3	4
arr	5	2	6	7	3

Implementing Bubble Sort Algorithm (Contd.)

- Let us sort this unsorted list.

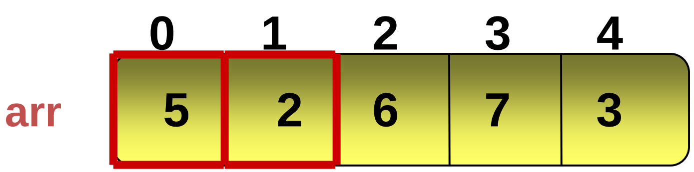
	0	1	2	3	4
arr	5	2	6	7	3

Implementing Bubble Sort Algorithm (Contd.)

Pass 1

n = 5

- Compare the element stored at index 0 with the element stored at index 1.

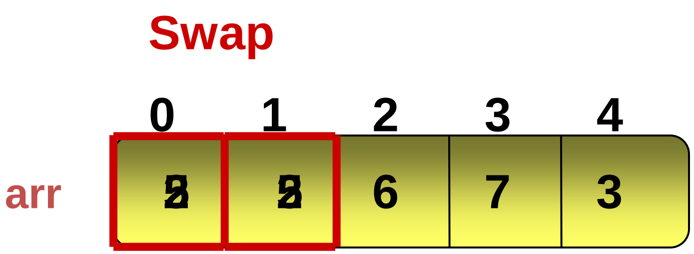


Implementing Bubble Sort Algorithm (Contd.)

Pass 1

n = 5

- Swap the values if they are not in the correct order.



Implementing Bubble Sort Algorithm (Contd.)

Pass 1

n = 5

- Compare the element stored at index 1 with the element stored at index 2 and swap the values if the value at index 1 is greater than the value at index 2.

No Change

	0	1	2	3	4
arr	2	5	6	7	3

Implementing Bubble Sort Algorithm (Contd.)

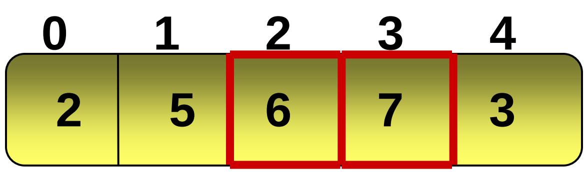
Pass 1

n = 5

- Compare the element stored at index 2 with the element stored at index 3 and swap the values if the value at index 2 is greater than the value at index 3.

No Change

arr

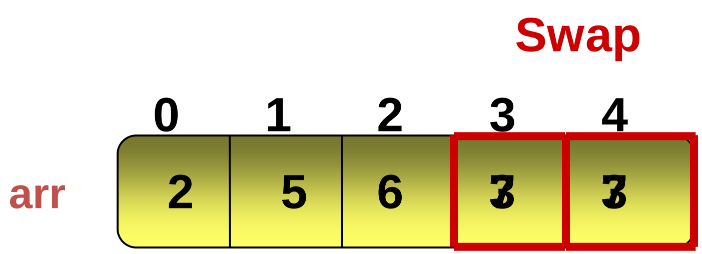


Implementing Bubble Sort Algorithm (Contd.)

Pass 1

n = 5

- Compare the element stored at index 3 with the element stored at index 4 and swap the values if the value at index 3 is greater than the value at index 4.

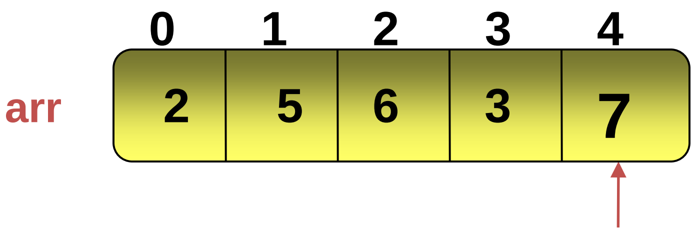


Implementing Bubble Sort Algorithm (Contd.)

Pass 1

n = 5

- Compare the element stored at index 3 with the element stored at index 4 and swap the values if the value at index 3 is greater than the value at index 4.



Largest element is placed at its correct position after Pass 1

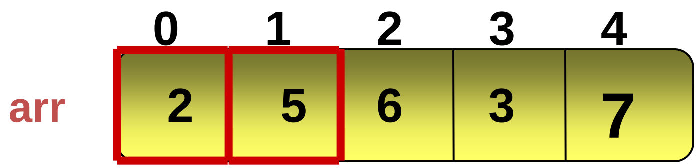
Implementing Bubble Sort Algorithm (Contd.)

Pass 2

n = 5

- Compare the element stored at index 0 with the element stored at index 1 and swap the values if the value at index 0 is greater than the value at index 1.

No Change



Implementing Bubble Sort Algorithm (Contd.)

Pass 2

n = 5

- Compare the element stored at index 1 with the element stored at index 2 and swap the values if the value at index 1 is greater than the value at index 2.

No Change

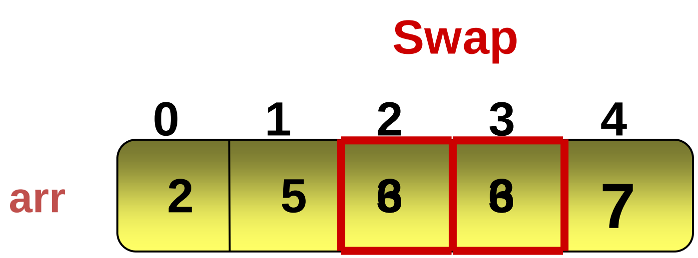
	0	1	2	3	4
arr	2	5	6	3	7

Implementing Bubble Sort Algorithm (Contd.)

Pass 2

n = 5

- Compare the element stored at index 2 with the element stored at index 3 and swap the values if the value at index 2 is greater than the value at index 3.

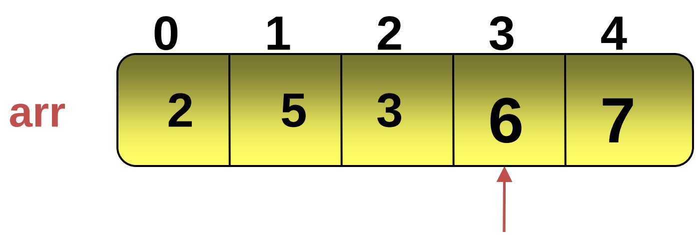


Implementing Bubble Sort Algorithm (Contd.)

Pass 2

n = 5

- Compare the element stored at index 2 with the element stored at index 3 and swap the values if the value at index 2 is greater than the value at index 3.



Second largest element is placed at its correct position after Pass 2

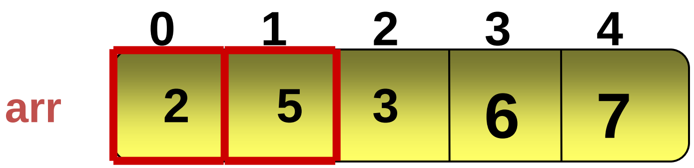
Implementing Bubble Sort Algorithm (Contd.)

Pass 3

n = 5

- Compare the element stored at index 0 with the element stored at index 1 and swap the values if the value at index 0 is greater than the value at index 1.

No Change

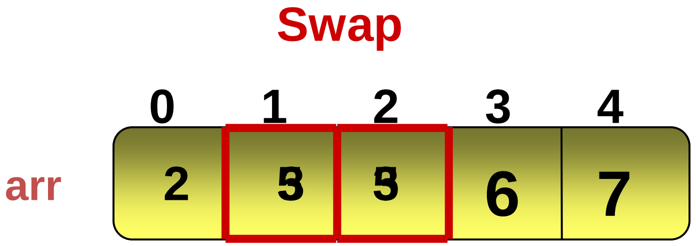


Implementing Bubble Sort Algorithm (Contd.)

Pass 3

n = 5

- Compare the element stored at index 1 with the element stored at index 2 and swap the values if the value at index 1 is greater than the value at index 2.



Implementing Bubble Sort Algorithm (Contd.)

Pass 3

n = 5

- Compare the element stored at index 2 with the element stored at index 3 and swap the values if the value at index 2 is greater than the value at index 3.

	0	1	2	3	4
arr	2	3	5	6	7

Third largest element is placed at its correct position after Pass 3

Implementing Bubble Sort Algorithm (Contd.)

Pass 4

n = 5

- Compare the element stored at index 0 with the element stored at index 1 and swap the values if the value at index 0 is greater than the value at index 1.

No Change

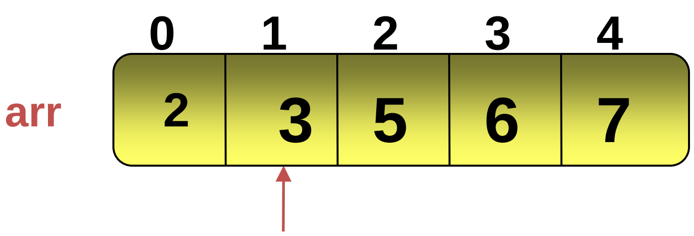
	0	1	2	3	4
arr	2	3	5	6	7

Implementing Bubble Sort Algorithm (Contd.)

Pass 4

n = 5

- Compare the element stored at index 0 with the element stored at index 1 and swap the values if the value at index 0 is greater than the value at index 1.



Fourth largest element is placed at its correct position after Pass 4

Implementing Bubble Sort Algorithm (Contd.)

Pass 4

n = 5

- At the end of Pass 4, the elements are sorted.

	0	1	2	3	4
arr	**2**	**3**	**5**	**6**	**7**

Implementing Bubble Sort Algorithm (Contd.)

- Write an algorithm to implement bubble sort.
- Algorithm Bubble(A[n])

```
{
  1.Enter n
  2.for i =0 to n-1
      {
      2.1 enter A[i]
      }
   3.For pass=1 to n-1
     {
        3.1 for j=0 to n-1-pass
         {
             3.1.1   If(a[j]>a[j+1])
                     {
                            swap(a[j],a[j+1])                    }
          }
    4.for i=0 to n-1
      {
      4.1 print A[i]
      }
   }
```

Determining the Efficiency of Bubble Sort Algorithm

- The efficiency of a sorting algorithm is measured in terms of number of comparisons.
- In bubble sort, there are n – 1 comparisons in Pass 1, n – 2 comparisons in Pass 2, and so on.
- Total number of comparisons = $(n - 1) + (n - 2) + (n - 3) + \ldots + 3 + 2 + 1 = n(n - 1)/2$.
- $n(n - 1)/2$ is of $O(n^2)$ order. Therefore, the bubble sort algorithm is of the order $O(n^2)$.

Just a minute

- What is the order of growth of the bubble sort algorithm?

- Answer:
  - The bubble sort algorithm has a quadratic order of growth.

Just a minute

- While implementing bubble sort algorithm, how many comparisons will be performed in Pass 1.

- Answer:
  - n – 1 comparisons

Sorting Data by Using Selection Sort

- Selection sort algorithm:
  - Has a quadratic order of growth and is therefore suitable for sorting small lists only
  - Scans through the list iteratively, selects one item in each scan, and moves the item to its correct position in the list

Implementing Selection Sort Algorithm

- To understand the implementation of selection sort algorithm, consider an unsorted list of numbers stored in an array.

	0	1	2	3	4
arr	**105**	**120**	**10**	**200**	**20**

Implementing Selection Sort Algorithm (Contd.)

- Let us sort this unsorted list.

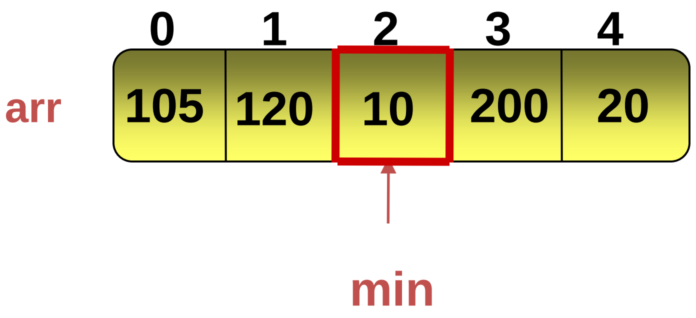
	0	1	2	3	4
arr	105	120	10	200	20

Implementing Selection Sort Algorithm (Contd.)

Pass 1

n = 5

- Search the minimum value in the array, arr[0] to arr[n – 1].

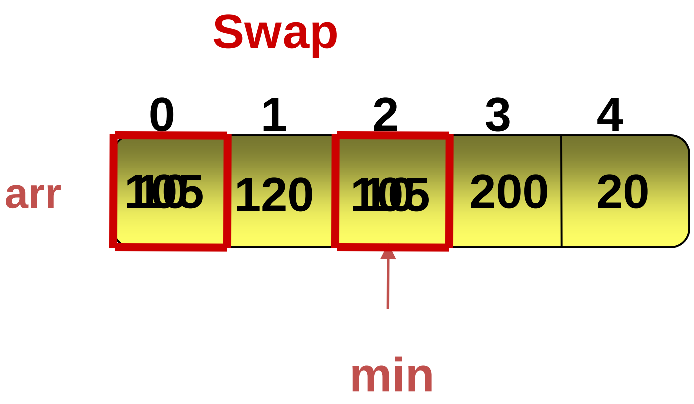


Implementing Selection Sort Algorithm (Contd.)

Pass 1

n = 5

- Search the minimum value in the array, arr[0] to arr[n – 1].
- Swap the minimum value with the value at index 0.



Implementing Selection Sort Algorithm (Contd.)

Pass 1

n = 5

- Search the minimum value in the array, arr[0] to arr[n – 1].
- Swap the minimum value with the value at index 0.

arr

0	1	2	3	4
10	120	105	200	20

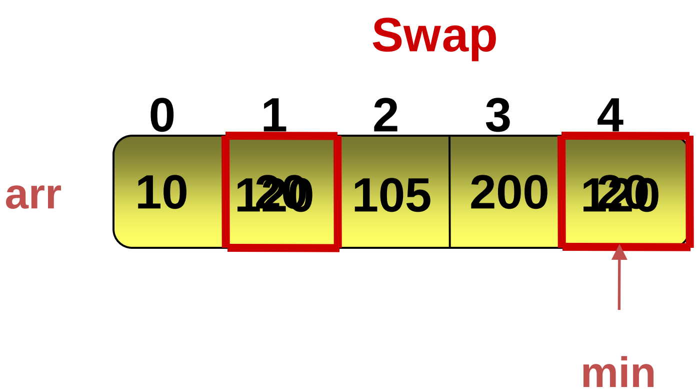
The smallest value is placed at its correct location after Pass 1

Implementing Selection Sort Algorithm (Contd.)

Pass 2

n = 5

- Search the minimum value in the array, arr[1] to arr[n – 1].
- Swap the minimum value with the value at index 1.

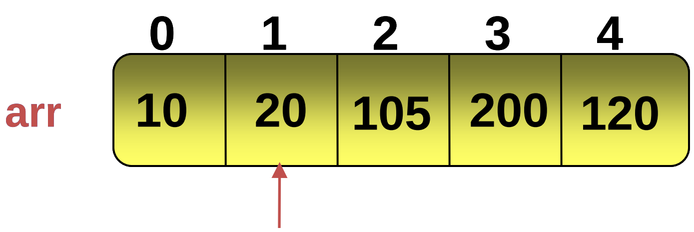


Implementing Selection Sort Algorithm (Contd.)

Pass 2

n = 5

- Search the minimum value in the array, arr[1] to arr[n – 1].
- Swap the minimum value with the value at index 1.



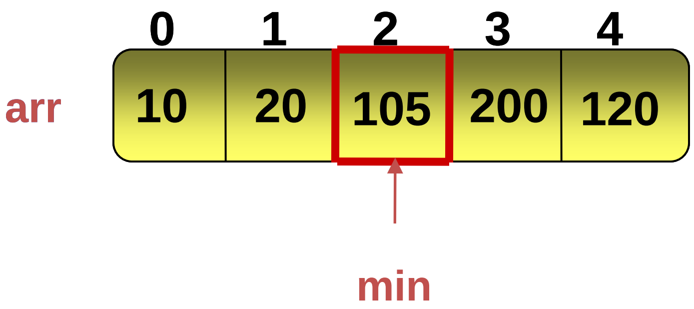
The second smallest value is placed at its correct location after Pass 2

Implementing Selection Sort Algorithm (Contd.)

Pass 3

n = 5

- Search the minimum value in the array, arr[2] to arr[n – 1].
- Swap the minimum value with the value at index 2.

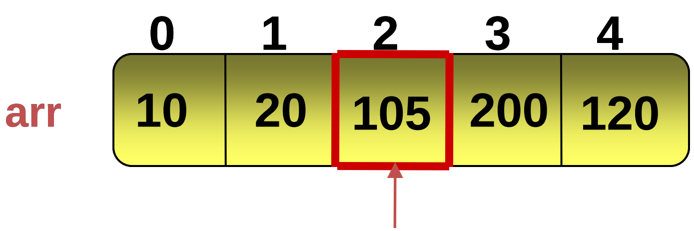


Implementing Selection Sort Algorithm (Contd.)

Pass 3

n = 5

- Search the minimum value in the array, arr[2] to arr[n – 1].
- Swap the minimum value with the value at index 2.



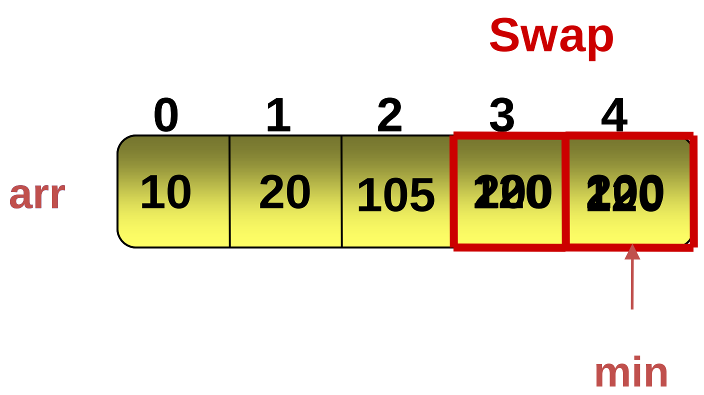
The third smallest value is placed at its correct location after Pass 3

Implementing Selection Sort Algorithm (Contd.)

Pass 4

n = 5

- Search the minimum value in the array, arr[3] to arr[n – 1].
- Swap the minimum value with the value at index 3.



Implementing Selection Sort Algorithm (Contd.)

Pass 4

n = 5

- Search the minimum value in the array, arr[3] to arr[n – 1].
- Swap the minimum value with the value at index 3.

	0	1	2	3	4
arr	**10**	**20**	**105**	**120**	**200**

The fourth smallest value is placed at its correct location after Pass 4

Implementing Selection Sort Algorithm (Contd.)

Pass 4

n = 5

- The list is now sorted.

	0	1	2	3	4
arr	10	20	105	120	200

Implementing Selection Sort Algorithm (Contd.)

- Write an algorithm to implement selection sort.
- Algorithm Selection (A[n])

```
1.Enter n
2.for i=0 t n-1
   {
      2.1 enter A[i]
   }
3.For i=0 to n-1
      {
         3.1 Set min_index = i
         3.2 for j=i+1 to n-1
         {
               3.2.1   If(A[j]<A[min_index])
                  {
                     Min_index=j
                  }
         }
         3.3swap A[i] and A[min_index]
      }
4.For i=0 to n-1
      {
      4.1 print A[i]
      }
```

Determining the Efficiency of Selection Sort Algorithm

- In selection sort, there are n – 1 comparisons during Pass 1 to find the smallest element, n – 2 comparisons during Pass 2 to find the second smallest element, and so on.
- Total number of comparisons = (n – 1) + (n – 2) + (n – 3) + … + 3 + 2 + 1 = n(n – 1)/2
- n(n – 1)/2 is of $O(n^2)$ order. Therefore, the selection sort algorithm is of the order $O(n^2)$.

Just a minute

- Read the following statement and specify whether it is true or false:
  - The worst case complexity of selection sort algorithm is same as that of the bubble sort algorithm.

- Answer:
  - True

Just a minute

- How many comparisons are performed in the second pass of the selection sort algorithm?

- Answer:
  - n – 2

Sorting Data by Using Insertion Sort

- Insertion sort algorithm:
  - Has a quadratic order of growth and is therefore suitable for sorting small lists only
  - Is much more efficient than bubble sort, and selection sort, if the list that needs to be sorted is nearly sorted

Implementing Insertion Sort Algorithm

- To understand the implementation of insertion sort algorithm, consider an unsorted list of numbers stored in an array.

	0	1	2	3	4
arr	70	80	30	10	20

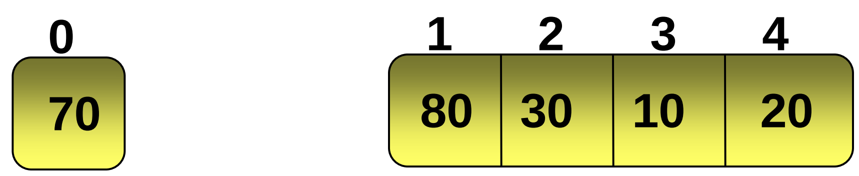
Implementing Insertion Sort Algorithm (Contd.)

- To sort this list by using the insertion sort algorithm:
  - You need to divide the list into two sublists, sorted and unsorted.

	0	1	2	3	4
arr	**70**	**80**	**30**	**10**	**20**

Implementing Insertion Sort Algorithm (Contd.)

- To sort this list by using the insertion sort algorithm:
  - You need to divide the list into two sublists, sorted and unsorted.
  - Initially, the sorted list has the first element and the unsorted list has the remaining 4 elements.



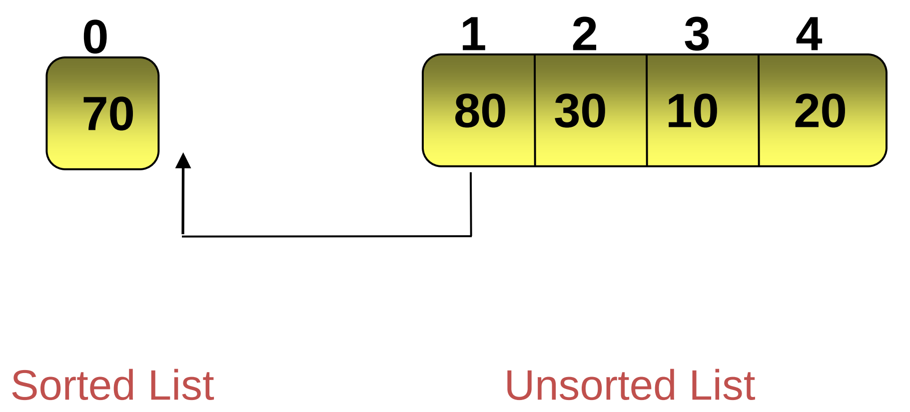
Sorted List

Unsorted List

Implementing Insertion Sort Algorithm (Contd.)

Pass 1

- Place the first element from the unsorted list at its correct position in the sorted list.



Implementing Insertion Sort Algorithm (Contd.)

Pass 1

- Place the first element from the unsorted list at its correct position in the sorted list.

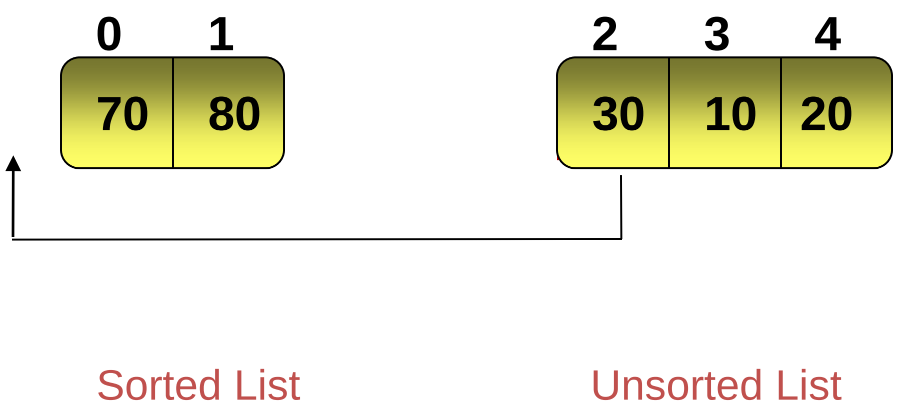
Sorted List

Unsorted List

Implementing Insertion Sort Algorithm (Contd.)

Pass 2

- Place the first element from the unsorted list at its correct position in the sorted list.



Implementing Insertion Sort Algorithm (Contd.)

Pass 2

- Place the first element from the unsorted list at its correct position in the sorted list.

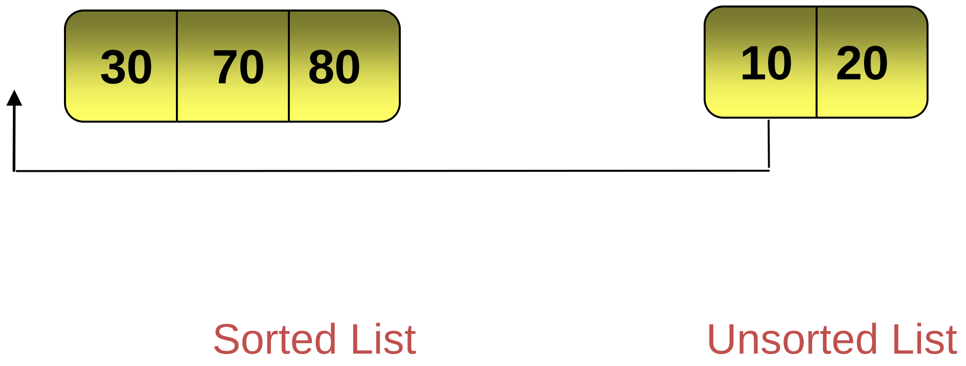
Sorted List

Unsorted List

Implementing Insertion Sort Algorithm (Contd.)

Pass 3

- Place the first element from the unsorted list at its correct position in the sorted list.



Implementing Insertion Sort Algorithm (Contd.)

Pass 3

- Place the first element from the unsorted list at its correct position in the sorted list.

0	1	2	3
10	30	70	80

Sorted List

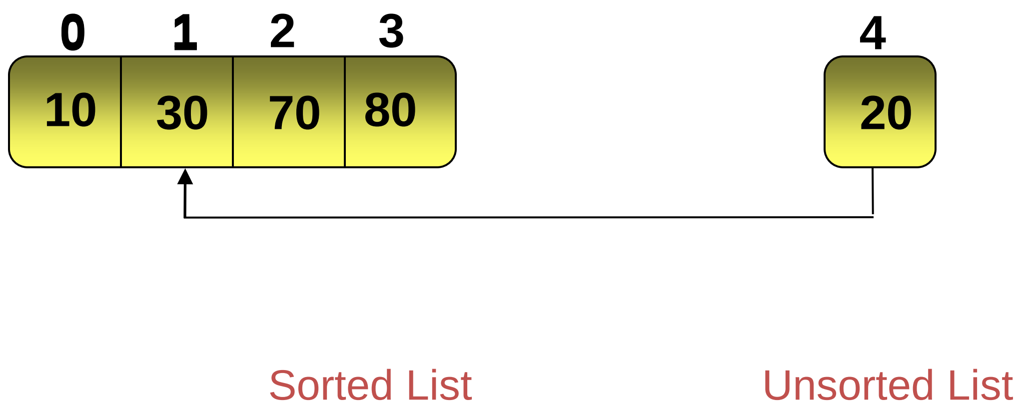
4
20

Unsorted List

Implementing Insertion Sort Algorithm (Contd.)

Pass 4

- Place the first element from the unsorted list at its correct position in the sorted list.



Implementing Insertion Sort Algorithm (Contd.)

Pass 4

- Place the first element from the unsorted list at its correct position in the sorted list.

0	1	2	3	4
10	20	30	70	80

Sorted List

Unsorted List

Implementing Insertion Sort Algorithm (Contd.)

- Let us now write an algorithm to implement insertion sort algorithm.

	0	1	2	3	4
arr	70	80	30	10	20

Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 1

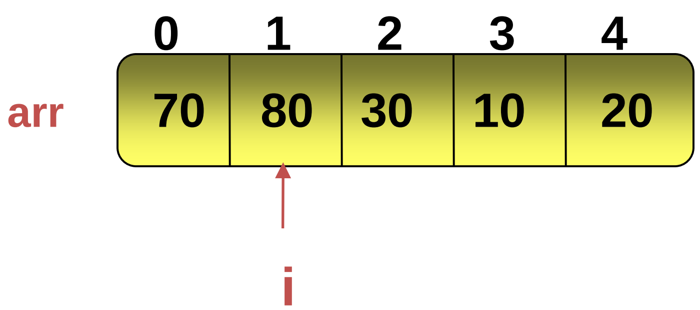
	0	1	2	3	4
arr	**70**	**80**	**30**	**10**	**20**

```
Algorithm InsertionSort(A[ ],n)
 {

for i:=1 to n-1
{

 temp := A[i]

 j := i – 1

 while(A[j] > temp && j >=0)
        {
        A[j+1]  :=A[j]
        J :=j-1
        }

 //Store temp at index j + 1
A[j+1]:=temp
}
}
```

Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 1



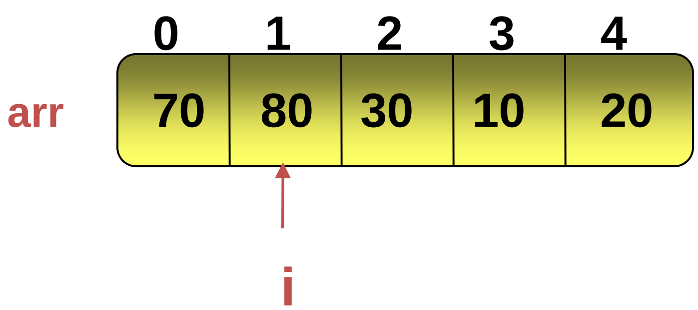

```
Algorithm(A,n)
 {

For(i=0;i<=n-1;i++)

{
      1.1Enter A[I]

}

2.for(i=1;i<=n-1;i++

{

 2.1Set temp = A[i]

 2.2Set j = i – 1

 2.3while(A[j] > temp && j >=0)
          {
           A[j+1]=A[J]
           J=J-1
           }

 2.4 Store temp at index j + 1

}
}
```

Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 1

temp = 80

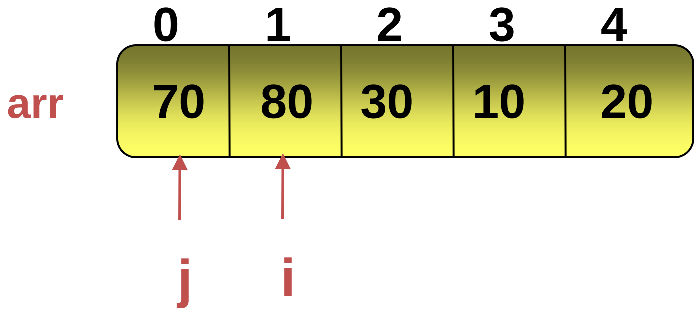


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 1

temp = 80



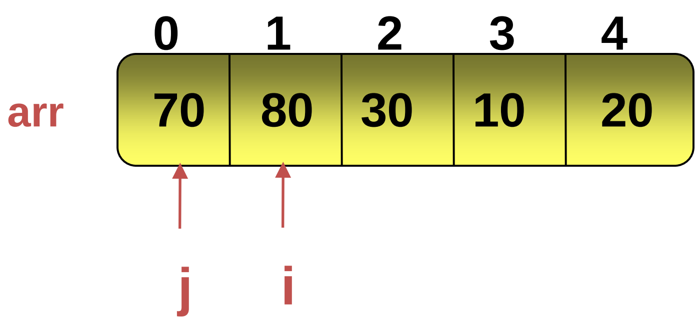
Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 1

temp = 80

arr[j] < temp



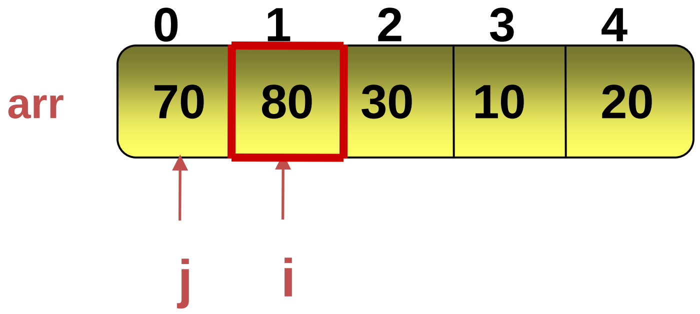
Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 1

temp = 80

arr[j] < temp



Value temp is stored at its correct position in the sorted list

Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 1

	0	1	2	3	4
arr	70	80	30	10	20

Implementing Insertion Sort Algorithm (Contd.)

n = 5

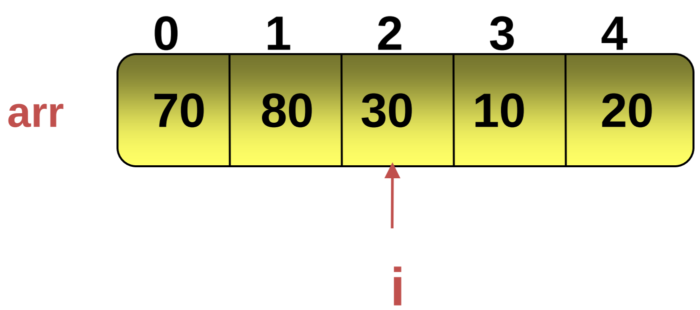
i = 2

	0	1	2	3	4
arr	**70**	**80**	**30**	**10**	**20**

Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

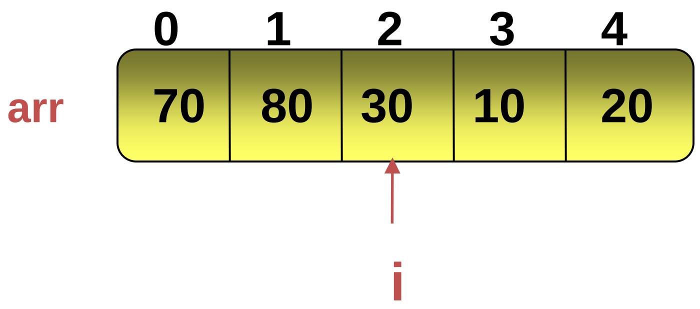


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30

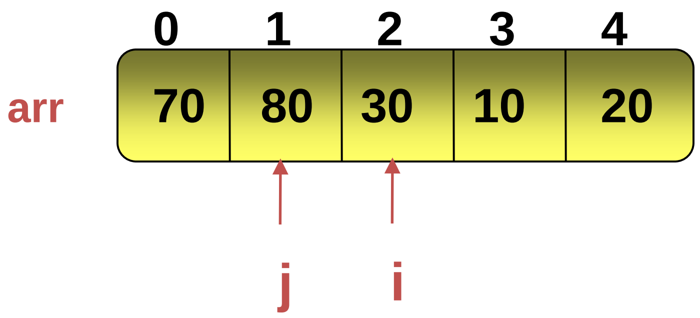


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30

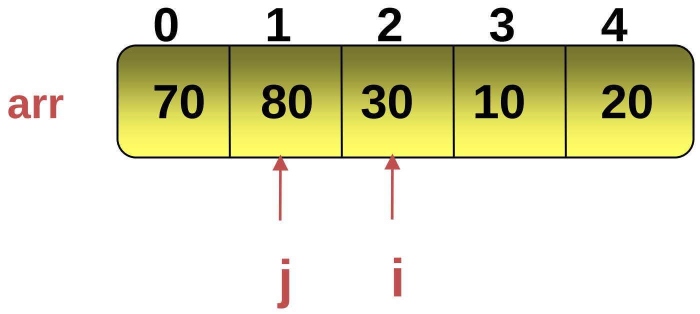


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30

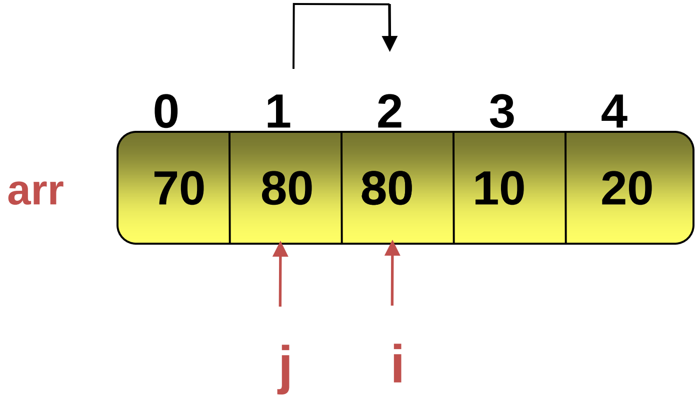


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30

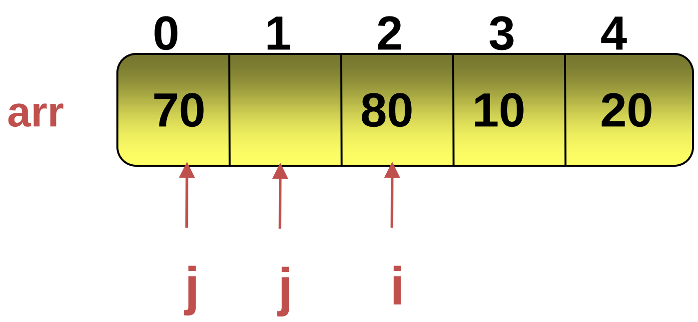


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30

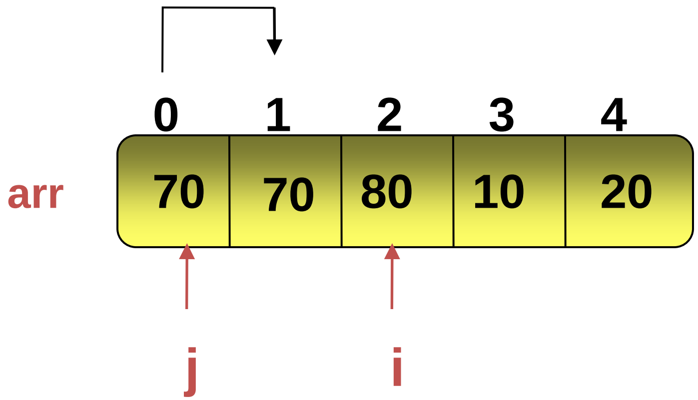


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30



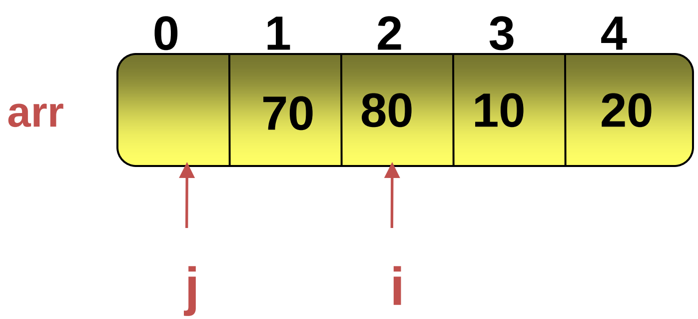
Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30

j = –1



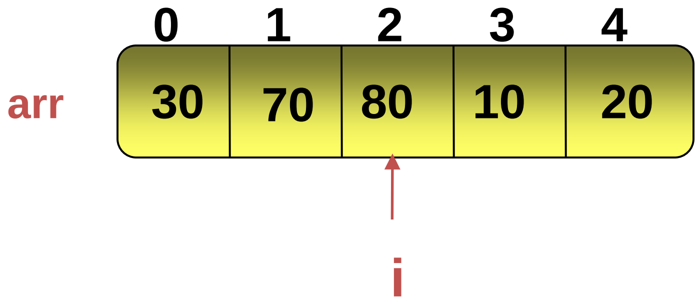
Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 2

temp = 30

j = –1



Implementing Insertion Sort Algorithm (Contd.)

n = 5

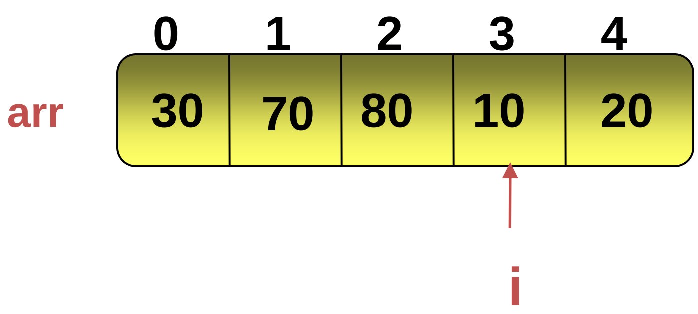
i = 2

	0	1	2	3	4
arr	**30**	**70**	**80**	**10**	**20**

Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

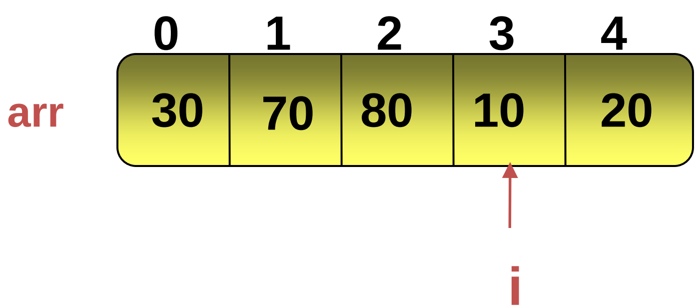


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

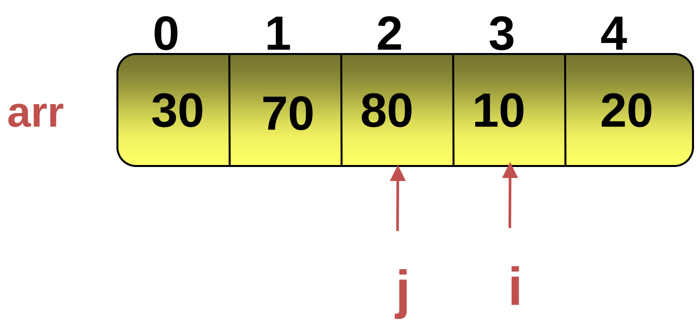


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

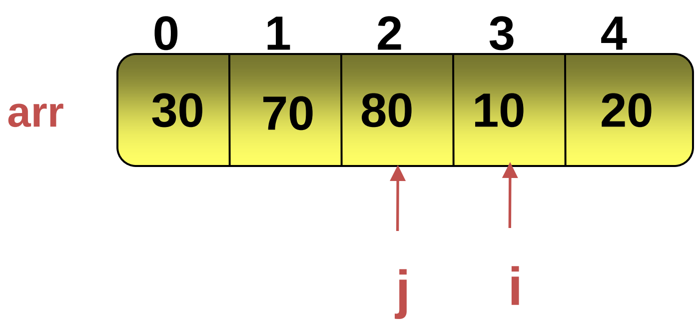


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

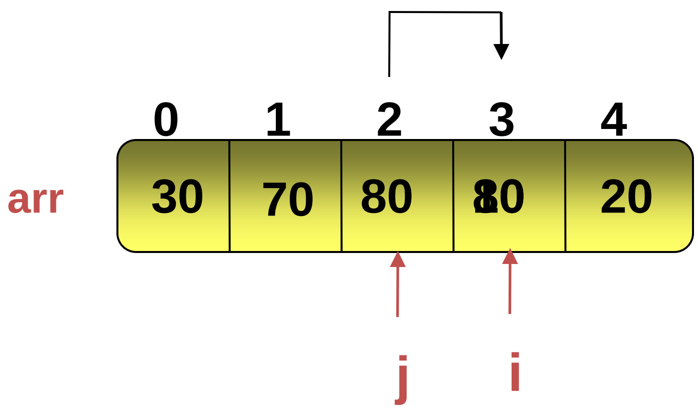


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

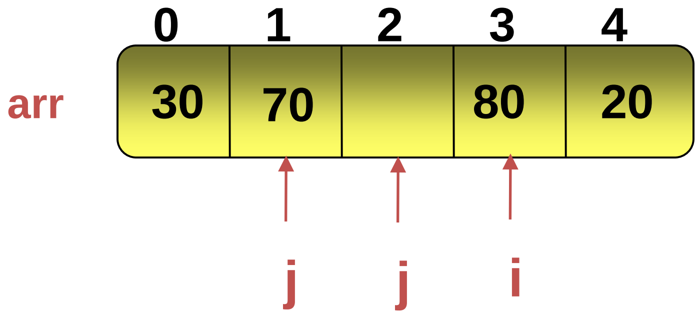


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

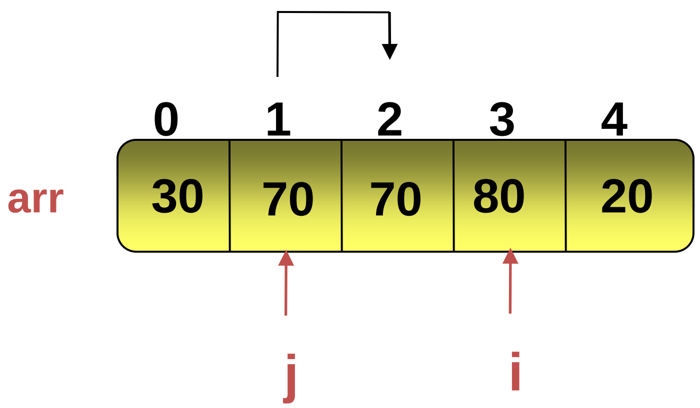


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

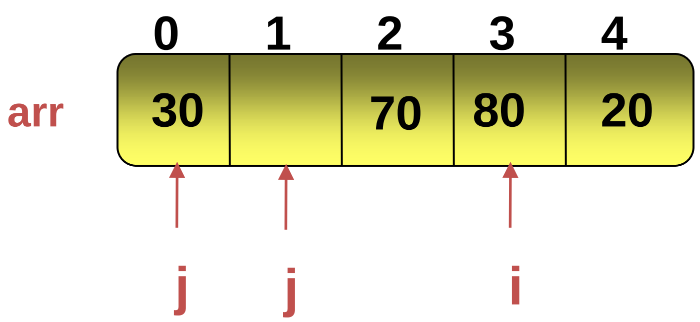


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

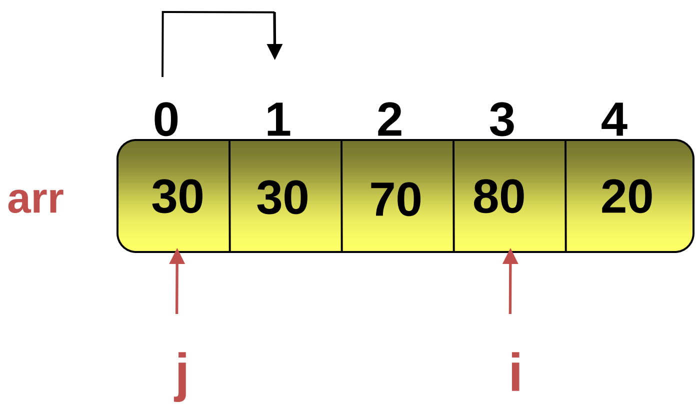


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10



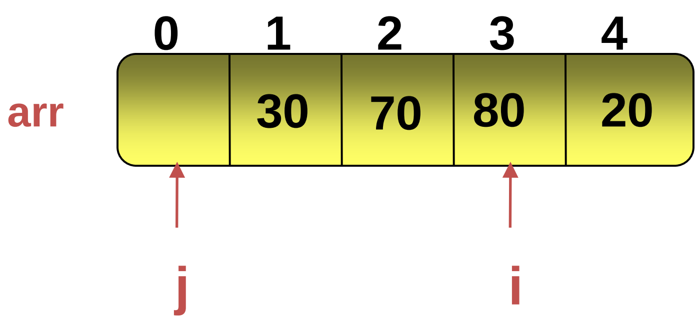
Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

j = –1



Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 3

temp = 10

j = –1

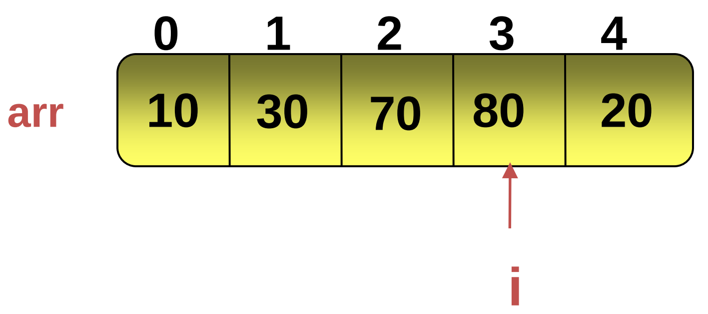
	0	1	2	3	4
arr	**10**	**30**	**70**	**80**	**20**

i

Implementing Insertion Sort Algorithm (Contd.)

n = 5

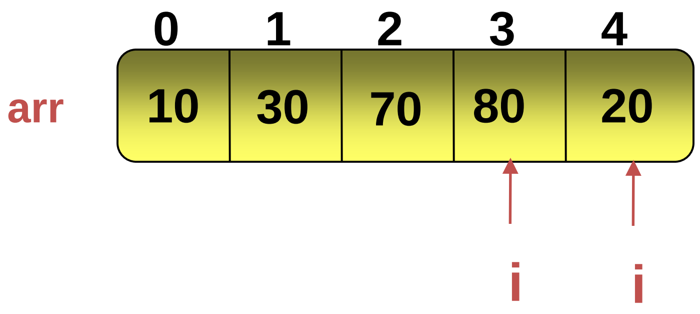
i = 3



Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

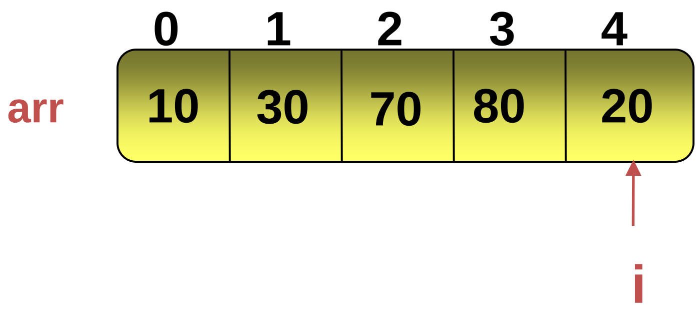


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

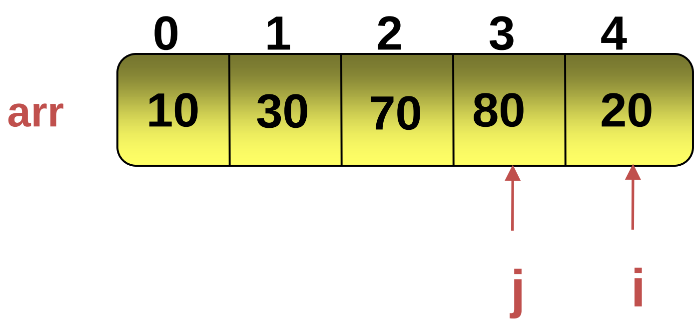


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

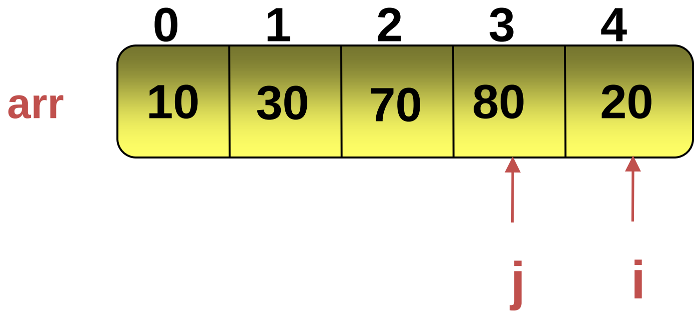


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

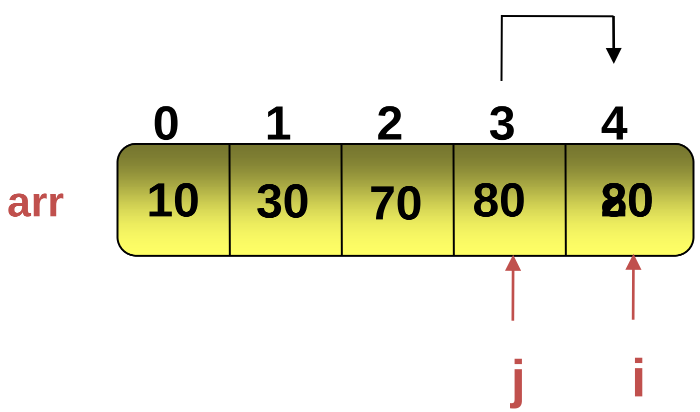


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

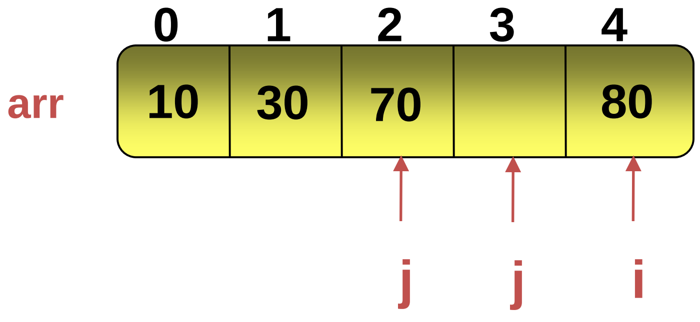


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

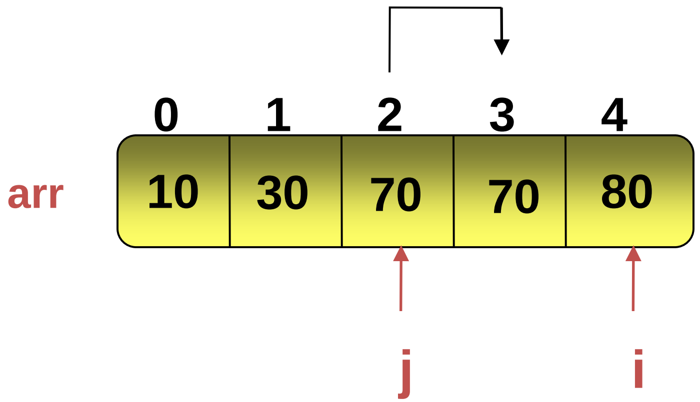


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

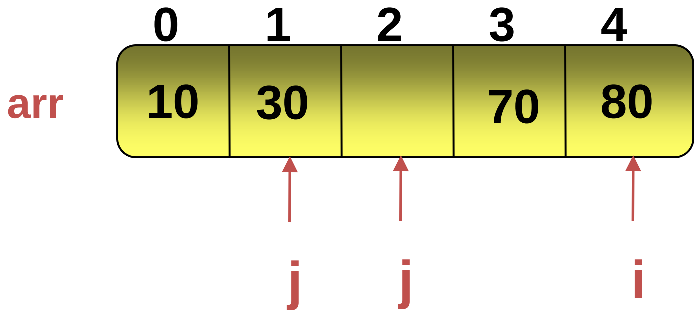


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

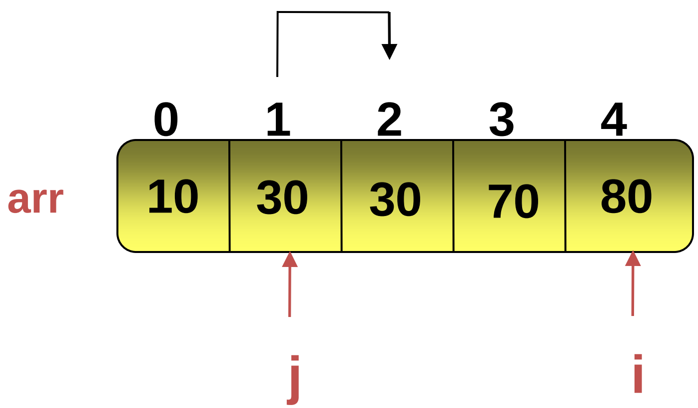


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

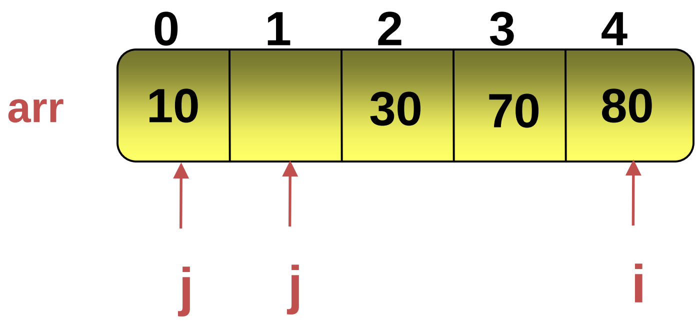


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

temp = 20

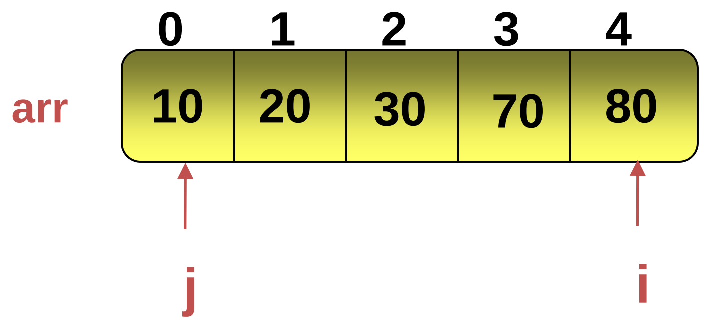


Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4

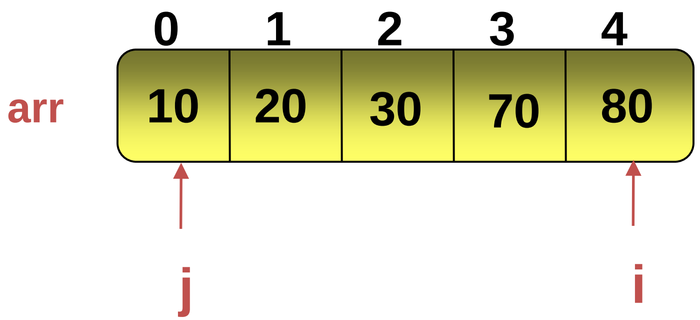
temp = 20



Implementing Insertion Sort Algorithm (Contd.)

n = 5

i = 4



The list is now sorted

Algorithm InsertionSort(A [ ],n)

{

1.For i=0 to n-1

1.1Enter A[i]

2.for(i=1 to n-1)

{

2.1Set temp = A[i]

2.2Set j = i – 1

2.3while(A[j] > temp&& j >=0)

{

a. A[j+1]=A[j]

b. j=j-1

}

2.4 Store temp at index j + 1

}

}

Determining the Efficiency of Insertion Sort Algorithm

- Best Case Efficiency:
  - Best case occurs when the list is already sorted.
  - In this case, you will have to make only one comparison in each pass.
  - In n – 1 passes, you will need to make n – 1 comparisons.
  - The best case efficiency of insertion sort is of the order O(n).

To sort a list of size n by using insertion sort, you need to perform (n – 1) passes.

- Worst Case Efficiency:
  - Worst case occurs when the list is sorted in the reverse order.
  - In this case, you need to perform one comparison in the first pass, two comparisons in the second pass, three comparisons in the third pass, and n – 1 comparisons in the $(n - 1)^{th}$ pass.
  - The worst case efficiency of insertion sort is of the order $O(n^2)$.

Just a minute

- A sales manager has to do a research on best seller cold drinks in the market for the year 2004-2006. David, the software developer, has a list of all the cold drink brands along with their sales figures stored in a file. David has to provide the sorted data to the sales manager. The data in the file is more or less sorted. Which sorting algorithm will be most efficient for sorting this data and why?

- Answer:
  - insertion sort algorithm.

Sorting Data by Using Shell Sort

- Shell sort algorithm:
  - Insertion sort is an efficient algorithm only if the list is already partially sorted and results in an inefficient solution in an average case.
  - To overcome this limitation, a computer scientist, D.L. Shell proposed an improvement over the insertion sort algorithm.
  - The new algorithm was called shell sort after the name of its proposer.

Implementing Shell Sort Algorithm

- Shell sort algorithm:
  - Improves insertion sort by comparing the elements separated by a distance of several positions to form multiple sublists
  - Applies insertion sort on each sublist to move the elements towards their correct positions
  - Helps an element to take a bigger step towards its correct position, thereby reducing the number of comparisons

Implementing Shell Sort Algorithm (Contd.)

- To understand the implementation of shell sort algorithm, consider an unsorted list of numbers stored in an array.

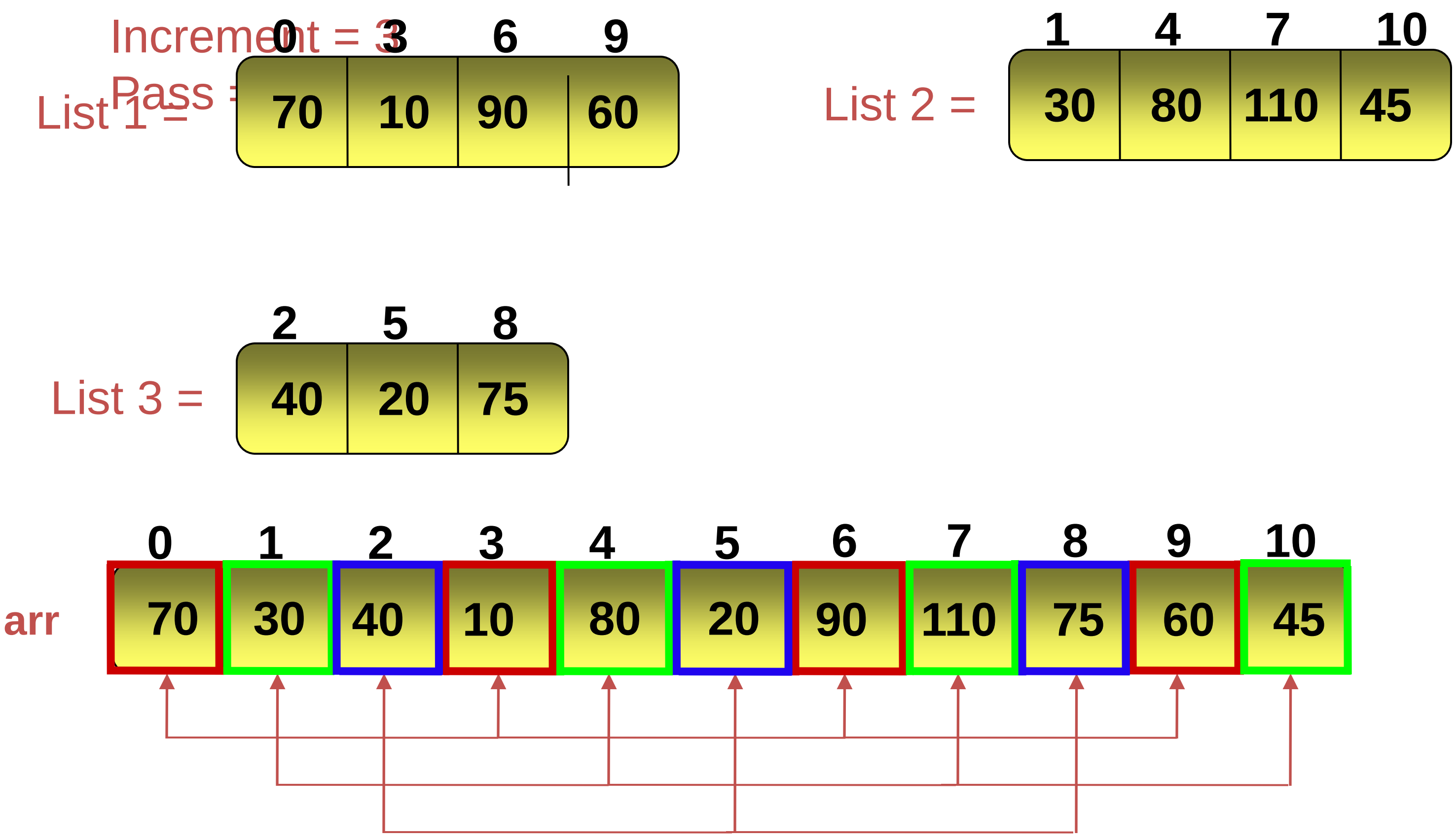
	0	1	2	3	4	5	6	7	8	9	10
arr	70	30	40	10	80	20	90	110	75	60	45

Implementing Shell Sort Algorithm (Contd.)

- To apply shell sort on this array, you need to:
  - Select the distance by which the elements in a group will be separated to form multiple sublists.
  - Apply insertion sort on each sublist to move the elements towards their correct positions.

	0	1	2	3	4	5	6	7	8	9	10
arr	70	30	40	10	80	20	90	110	75	60	45

Implementing Shell Sort Algorithm (Contd.)



Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	3	6	9
70 / 10	60 / 10	70 / 90	80 / 90

List 2 =

1	4	7	10
30	85 / 40 / 65	110 / 80 / 100	150 / 45

List 3 =

2	5	8
20 / 40	40 / 20	75

Apply insertion sort to sort the three lists

The lists are sorted

Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	3	6	9
10	60	70	90

List 2 =

1	4	7	10
30	45	80	110

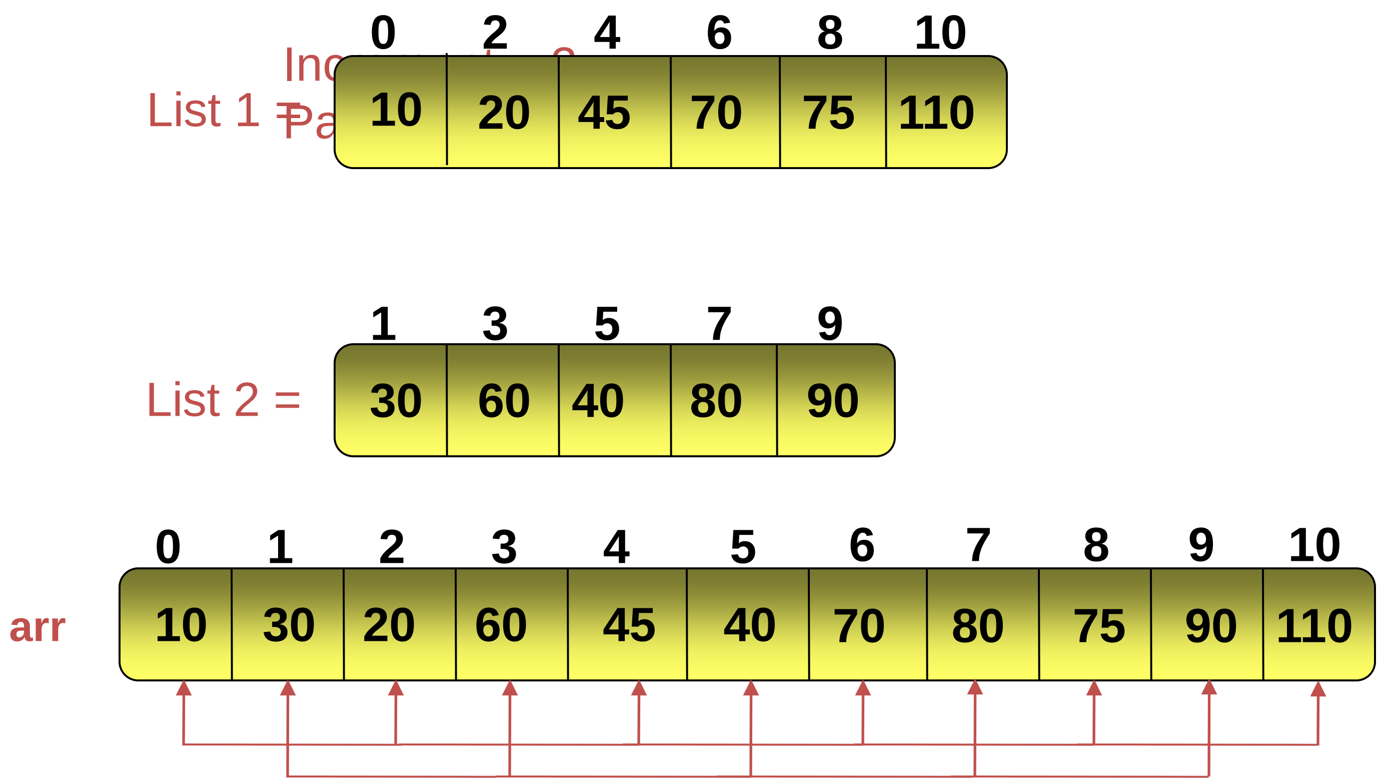
List 3 =

2	5	8
20	40	75

arr

0	1	2	3	4	5	6	7	8	9	10
10	30	20	60	45	40	70	80	75	90	110

Implementing Shell Sort Algorithm (Contd.)



Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	2	4	6	8	10
10	20	45	70	75	110

List 2 =

1	3	5	7	9
30	60	40	80	90

Apply insertion sort on each sublist

Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	2	4	6	8	10
10	20	45	70	75	110

List 2 =

1	3	5	7	9
30	40	60	80	90

The lists are now sorted

Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	2	4	6	8	10
10	20	45	70	75	110

List 2 =

1	3	5	7	9
30	40	60	80	90

arr

0	1	2	3	4	5	6	7	8	9	10
10	30	20	40	45	60	70	80	75	90	110

Implementing Shell Sort Algorithm (Contd.)

Increment = 1

Pass = 3

	0	1	2	3	4	5	6	7	8	9	10
arr	10	30	20	40	45	60	70	80	75	90	110

Apply insertion sort to sort the list

Implementing Shell Sort Algorithm (Contd.)

Increment = 1

Pass = 3

	0	1	2	3	4	5	6	7	8	9	10
arr	10	20	30	40	45	60	70	75	80	90	110

The list is now sorted

Pseudo Code

```
# Sort an array a[0...n-1].
//gaps = [1, 4, 13, 40, 121.. and so on]. [gaps=gaps*3+1]
//Knuth's formula
// Start with a big gap, then reduce the gap
    for (gap = n/2; gap > 0; gap /= 2) //original shell gap
    {
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already in
gapped order
        // keep adding one more element until the entire array is
        // gap sorted
```

```
for (i = gap; i < n; i += 1)
     {
          // add a[i] to the elements that have been gap sorted
          // save a[i] in temp and make a hole at position i
          temp = arr[i];
           // shift earlier gap-sorted elements up until the correct
          // location for a[i] is found
        for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
              arr[j] = arr[j - gap];

          //  put temp (the original a[i]) in its correct location
          arr[j] = temp;
     }
```

Just a minute

- Which of the following sorting algorithms compares the elements separated by a distance of several positions to sort the data? The options are:
  1. Insertion sort
  2. Selection sort
  3. Bubble sort
  4. Shell sort

- Answer:
  4. Shell sort

Sorting Data by Using Quick Sort

- Quick sort algorithm:
  - Is one of the most efficient sorting algorithms
  - Is based on the divide and conquer approach
  - Successively divides the problem into smaller parts until the problems become so small that they can be directly solved

Implementing Quick Sort Algorithm

- In quick sort algorithm, you:
  - Select an element from the list called as pivot.
  - Partition the list into two parts such that:
    - **All the elements towards the left end of the list are smaller than the pivot.**
    - **All the elements towards the right end of the list are greater than the pivot.**
  - Store the pivot at its correct position between the two parts of the list.
- You repeat this process for each of the two sublists created after partitioning.
- This process continues until one element is left in each sublist.

Implementing Quick Sort Algorithm (Contd.)

- To understand the implementation of quick sort algorithm, consider an unsorted list of numbers stored in an array.

	0	1	2	3	4	5	6	7
arr	28	55	46	38	16	89	83	30

Implementing Quick Sort Algorithm (Contd.)

- Let us sort this unsorted list.

	0	1	2	3	4	5	6	7
arr	28	55	46	38	16	89	83	30

Implementing Quick Sort Algorithm (Contd.)

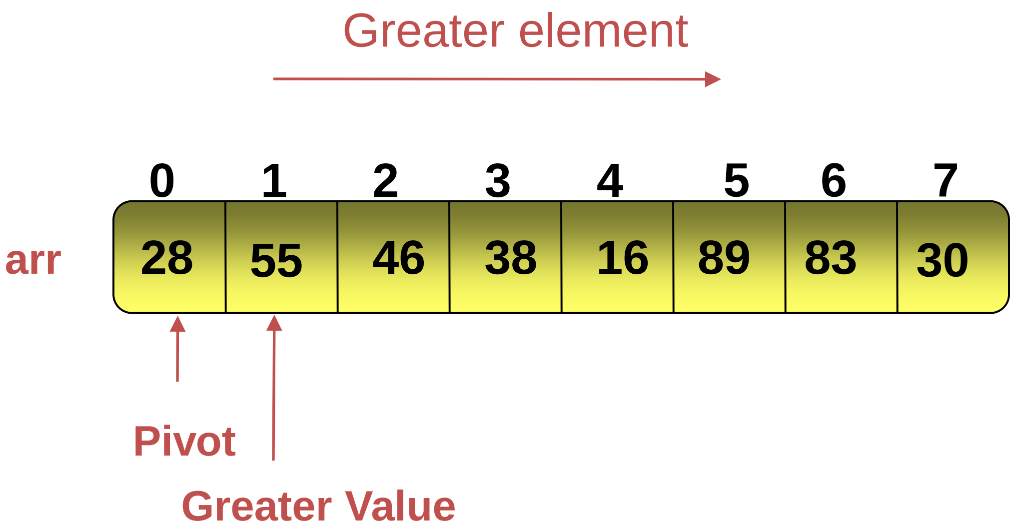
- Select a Pivot.

	0	1	2	3	4	5	6	7
arr	**28**	**55**	**46**	**38**	**16**	**89**	**83**	**30**

Pivot

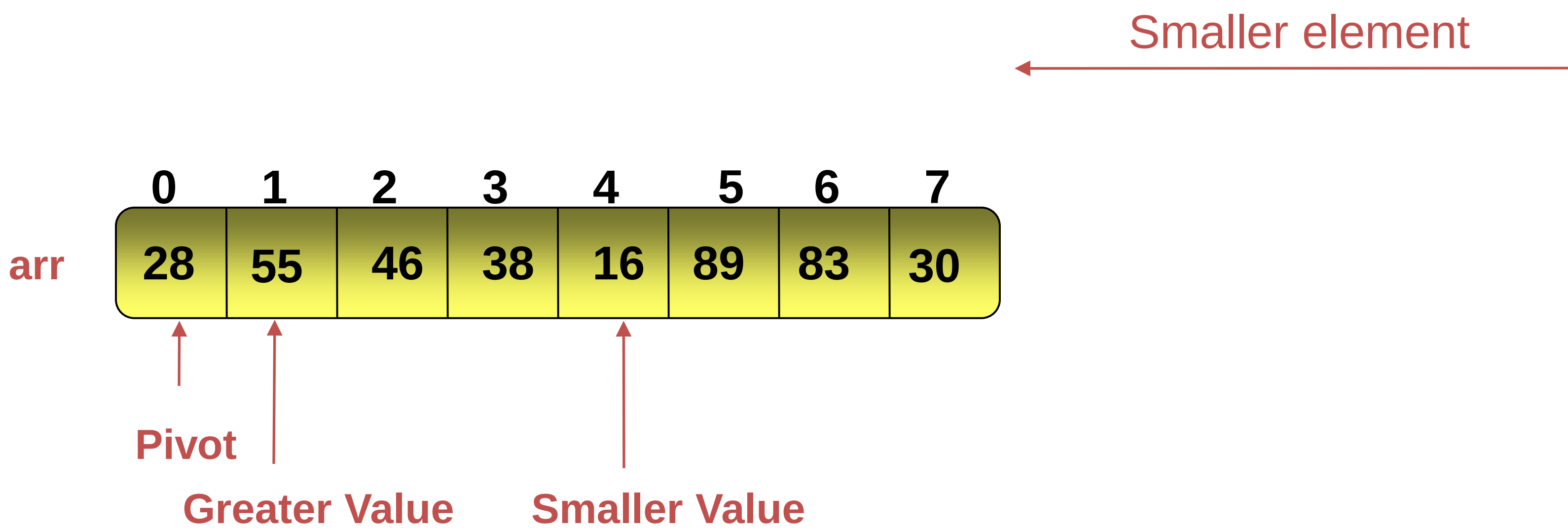
Implementing Quick Sort Algorithm (Contd.)

- Start from the left end of the list (at index 1).
- Move in the left to right direction.
- Search for the first element that is greater than the pivot value.



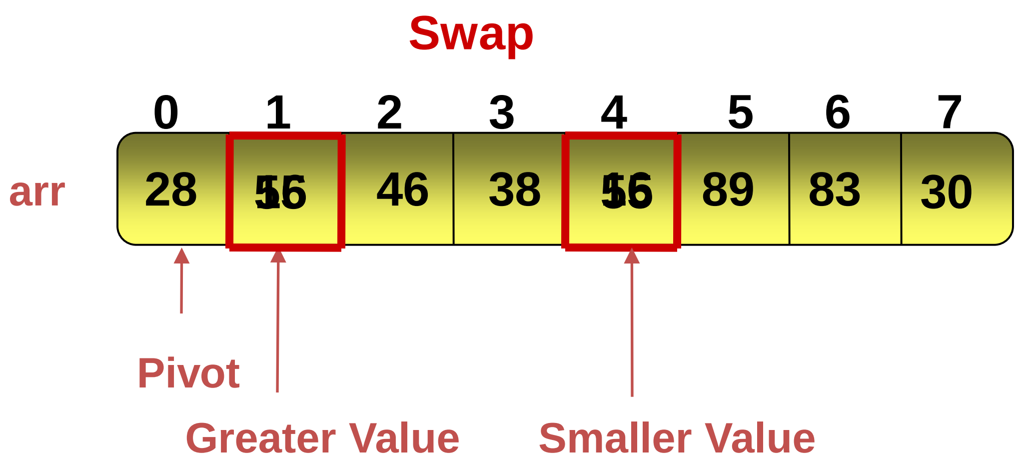
Implementing Quick Sort Algorithm (Contd.)

- Start from the right end of the list.
- Move in the right to left direction.
- Search for the first element that is smaller than or equal to the pivot value.



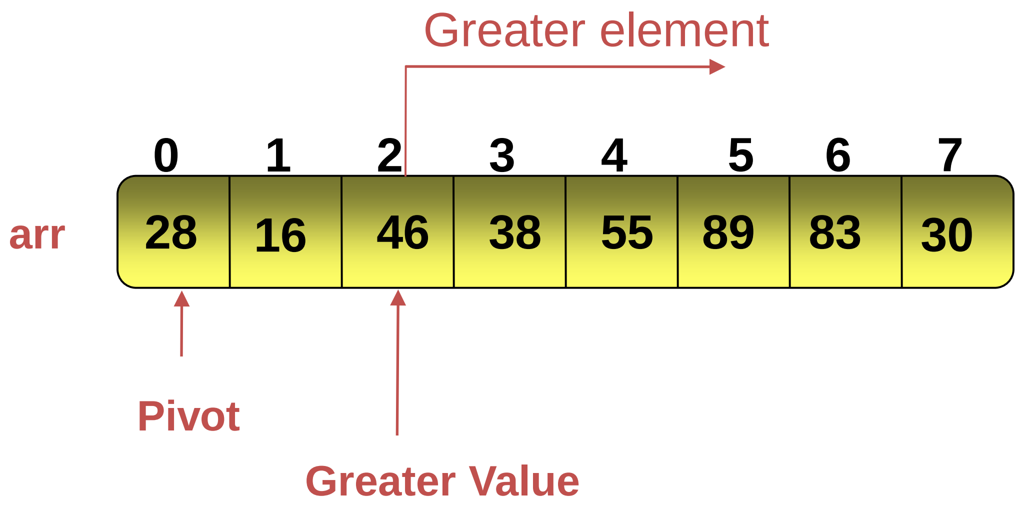
Implementing Quick Sort Algorithm (Contd.)

- Interchange the greater value with smaller value.



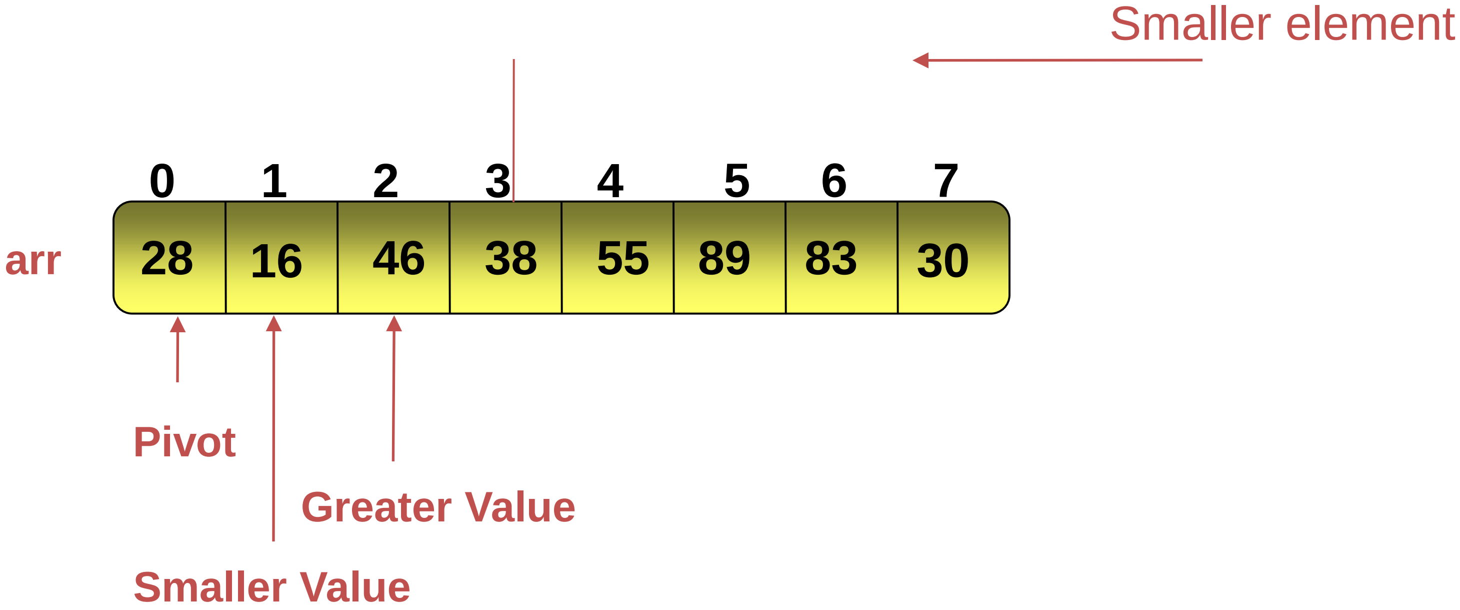
Implementing Quick Sort Algorithm (Contd.)

- Continue the search for an element greater than the pivot.
- Start from arr[2] and move in the left to right direction.
- Search for the first element that is greater than the pivot value.



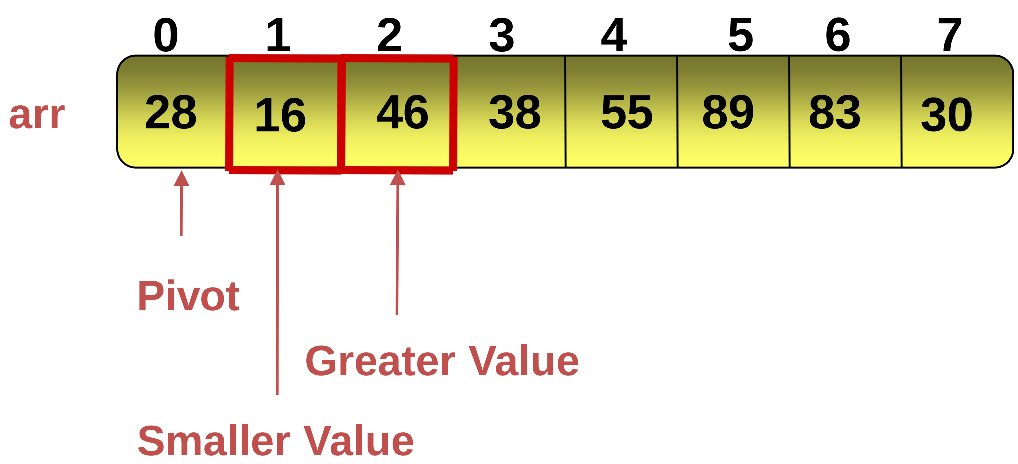
Implementing Quick Sort Algorithm (Contd.)

- Continue the search for an element smaller than the pivot.
- Start from arr[3] and move in the right to left direction.
- Search for the first element that is smaller than or equal to the pivot value.



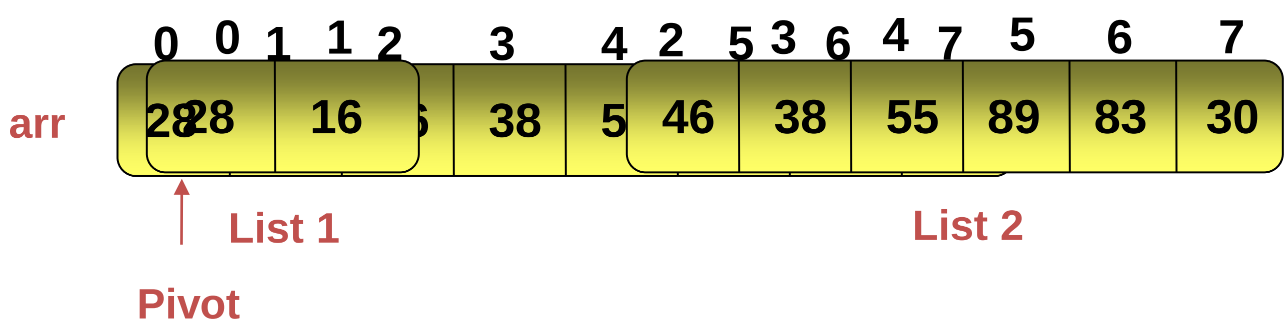
Implementing Quick Sort Algorithm (Contd.)

- The smaller value is on the left hand side of the greater value.
- Values remain same.



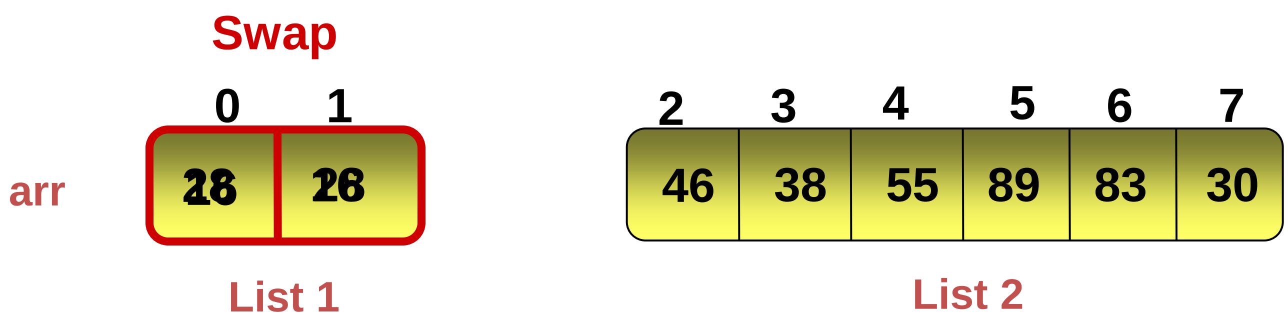
Implementing Quick Sort Algorithm (Contd.)

- List is now partitioned into two sublists.
- List 1 contains all values less than or equal to the pivot.
- List 2 contains all the values greater than the pivot.



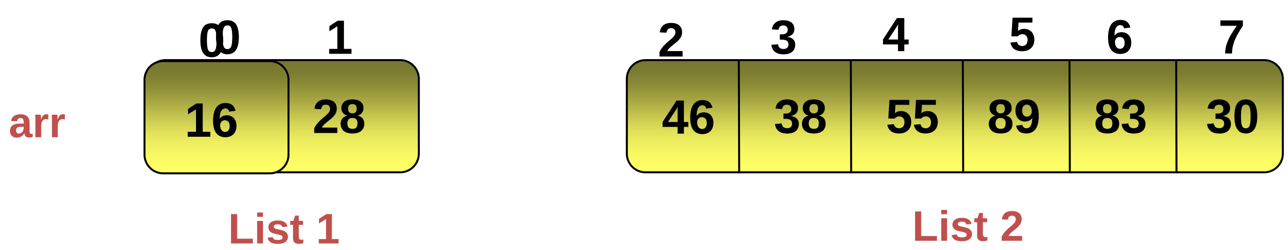
Implementing Quick Sort Algorithm (Contd.)

- Replace the pivot value with the last element of List 1.
- The pivot value, 28 is now placed at its correct position in the list.



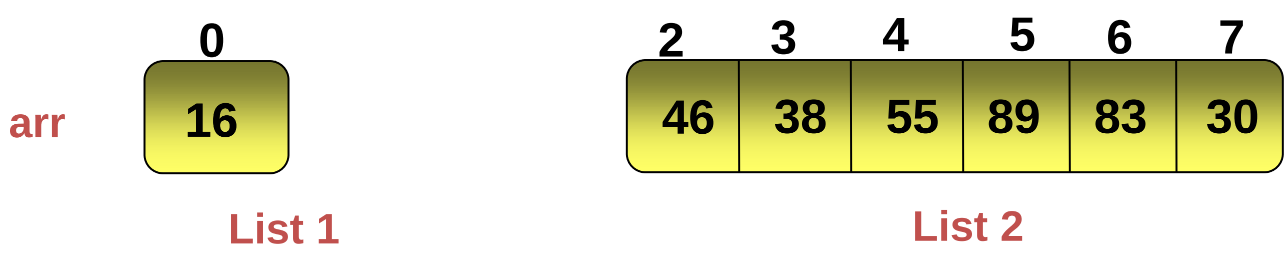
Implementing Quick Sort Algorithm (Contd.)

- Truncate the last element, that is, pivot from List 1.



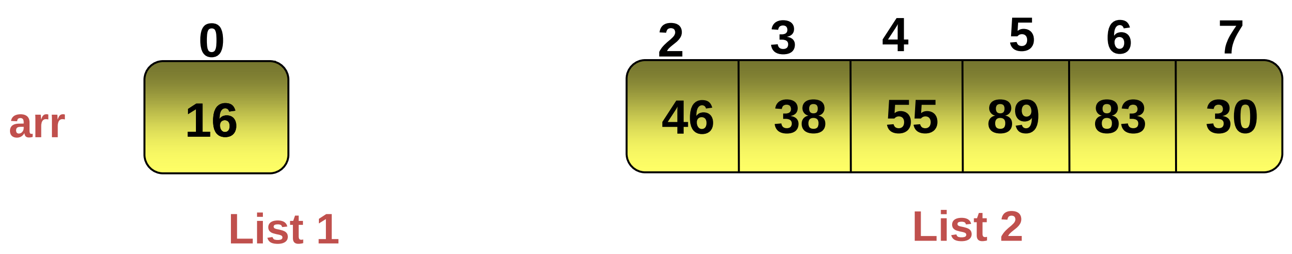
Implementing Quick Sort Algorithm (Contd.)

- List 1 has only one element.
- Therefore, no sorting required.



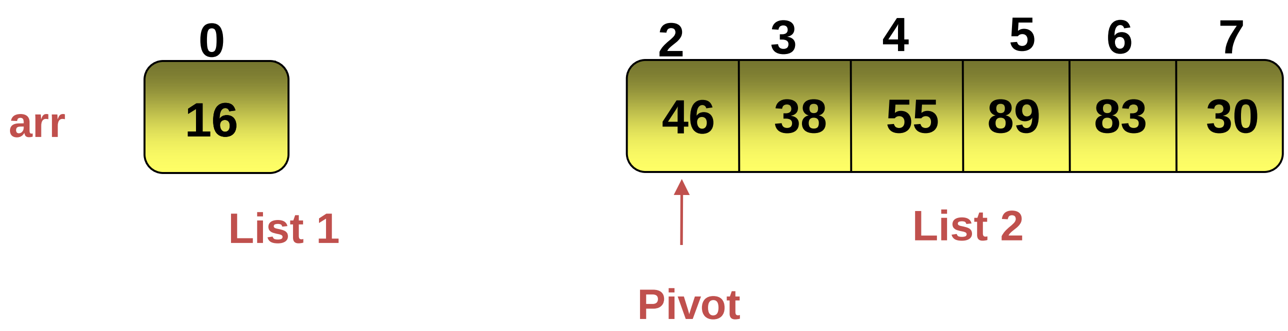
Implementing Quick Sort Algorithm (Contd.)

- Sort the second list, List 2.



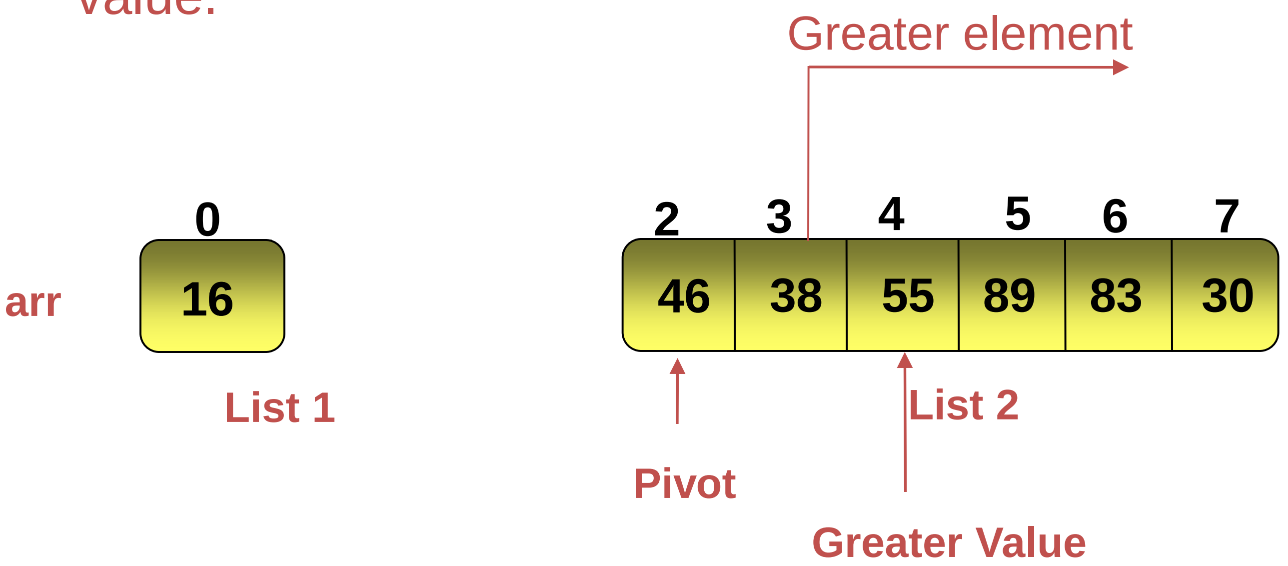
Implementing Quick Sort Algorithm (Contd.)

- Select a pivot.
- The pivot in this case will be arr[2], that is, 46.



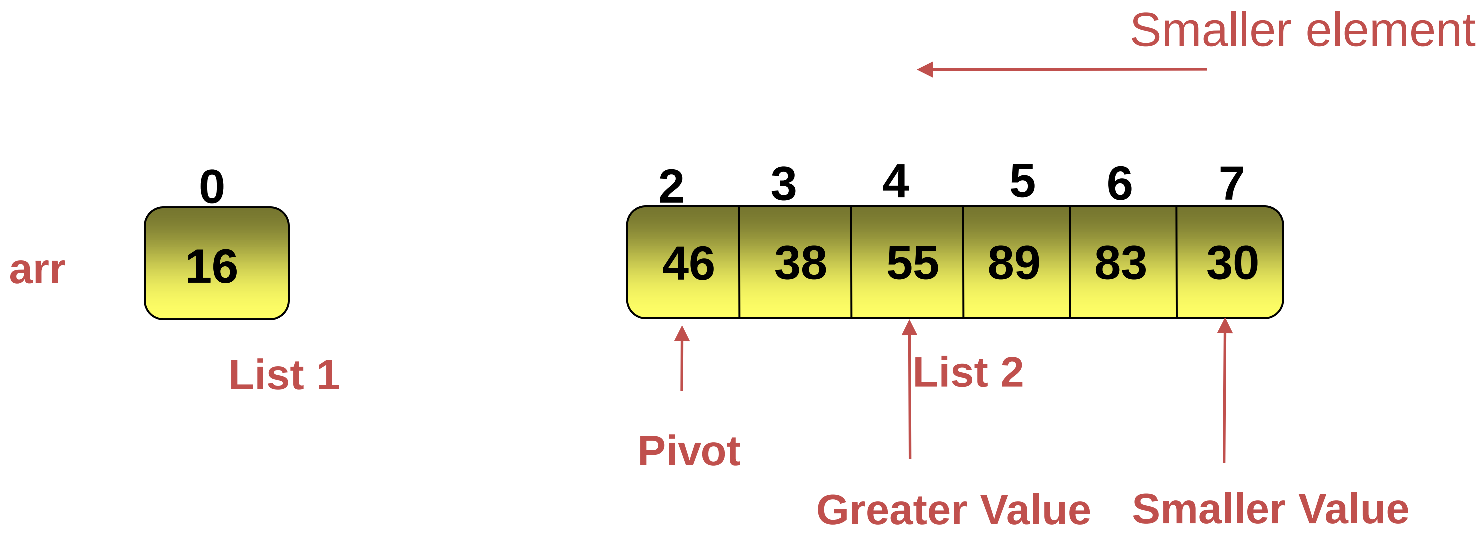
Implementing Quick Sort Algorithm (Contd.)

- Start from the left end of the list (at index 3).
- Move in the left to right direction.
- Search for the first element that is greater than the pivot value.



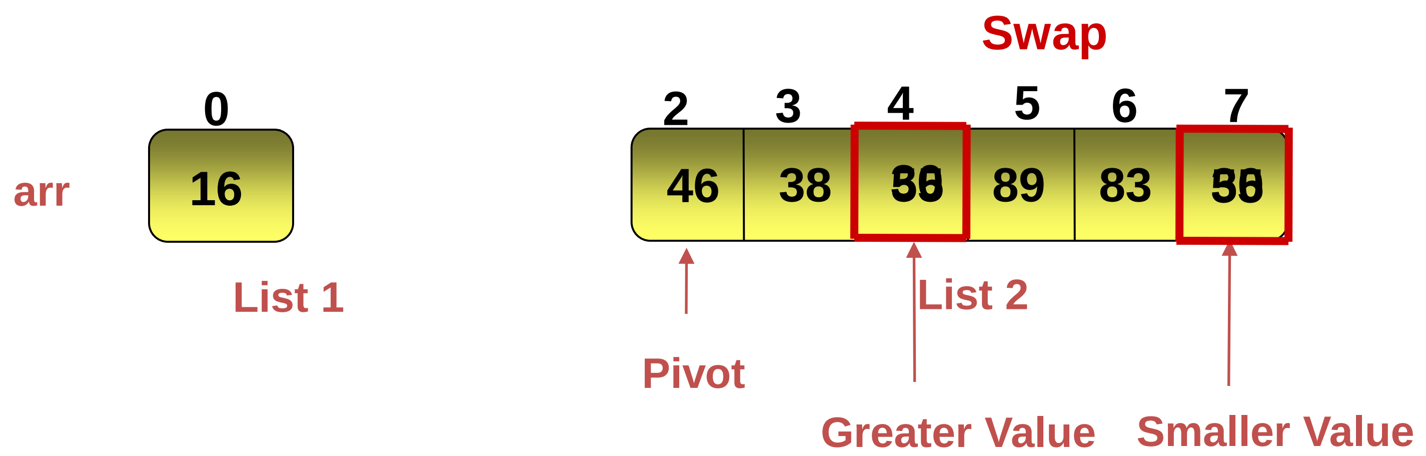
Implementing Quick Sort Algorithm (Contd.)

- Start from the right end of the list (at index 7).
- Move in the right to left direction.
- Search for the first element that is smaller than or equal to the pivot value.



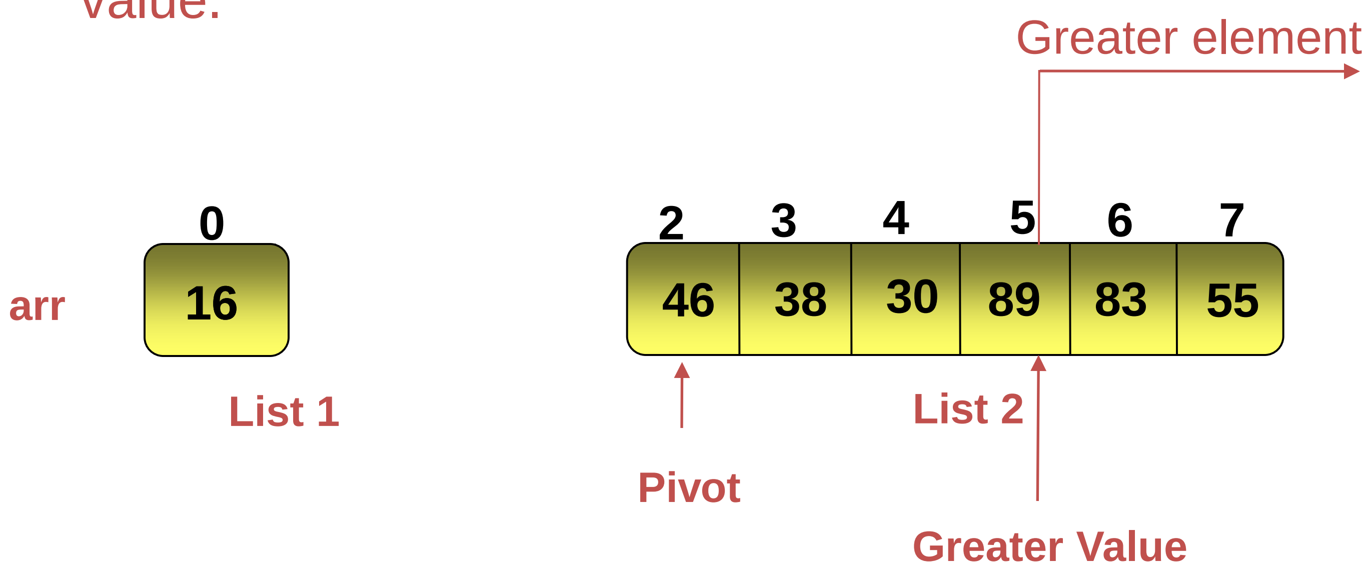
Implementing Quick Sort Algorithm (Contd.)

- Interchange the greater value with smaller value.



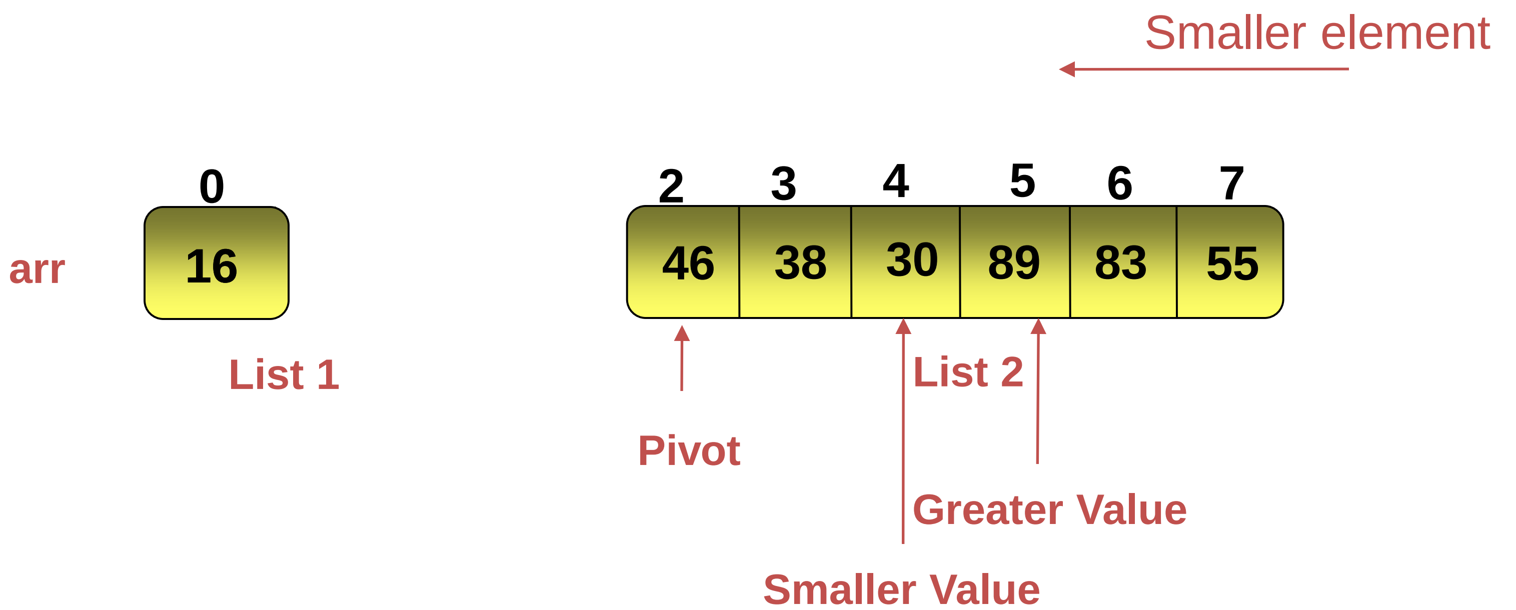
Implementing Quick Sort Algorithm (Contd.)

- Continue the search for an element greater than the pivot.
- Start from arr[5] and move in the left to right direction.
- Search for the first element that is greater than the pivot value.



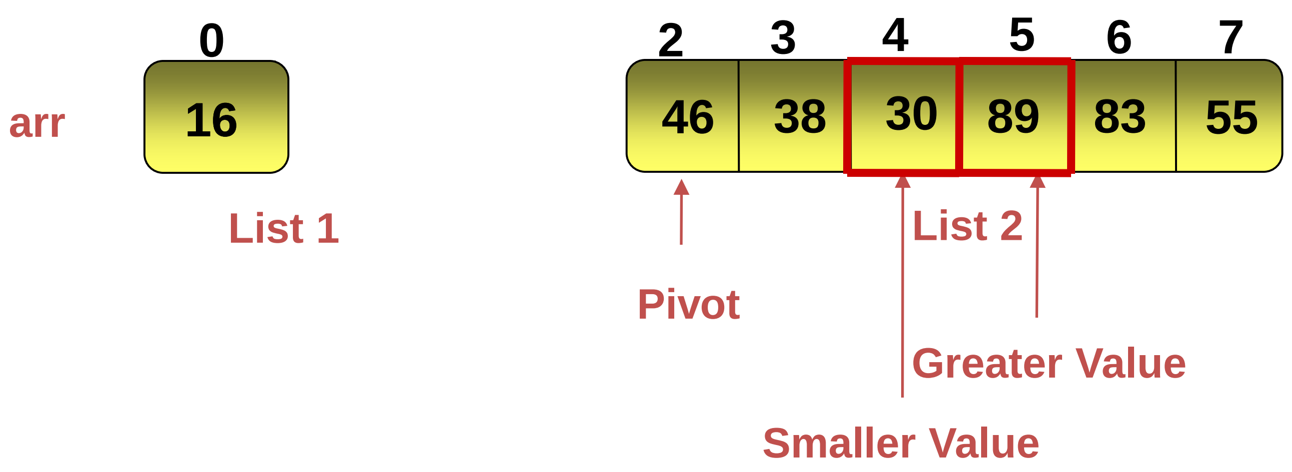
Implementing Quick Sort Algorithm (Contd.)

- Continue the search for an element smaller than the pivot.
- Start from arr[6] and move in the right to left direction.
- Search for the first element that is smaller than the pivot value.



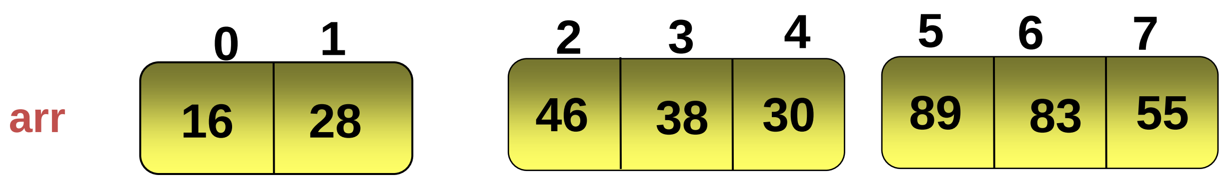
Implementing Quick Sort Algorithm (Contd.)

- The smaller value is on the left hand side of the greater value.
- Values remain same.



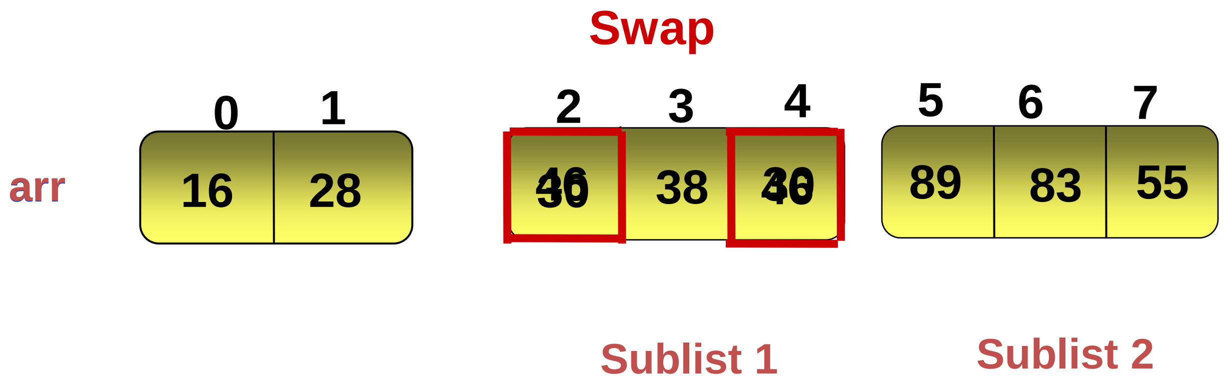
Implementing Quick Sort Algorithm (Contd.)

- Divide the list into two sublists.
- Sublist 1 contains all values less than or equal to the pivot.
- Sublist 2 contains all the values greater than the pivot.



Implementing Quick Sort Algorithm (Contd.)

- Replace the pivot value with the last element of Sublist 1.
- The pivot value, 46 is now placed at its correct position in the list.
- This process is repeated until all elements reach their correct position.



```
Algorithm  QuickSort (arr [ n], low, high)
{   //  arr is an array of n elements, low is lower bound of an array and high
    is upper bound.
1.  If (low >= high): Return;
2.  else
3.    pivot = arr [low]
4.    i = low + 1
5.    j = high
6.    do
      {
      5.1 while ( arr [i]<pivot and i<=high)     // search element higher
                                                   then pivot
            5.1.1 i=i+1
      5.2 while ( arr [j]>pivot and j>=low )     // Search for an element
                                                   smaller than pivot
            5.2.1 j=j-1
```

```
        5.3 If (i < j)
              5.3.1 Swap arr [i] with arr [j]
               5.3.2   i = i+ 1
               5.3.3   j = j - 1  // to start searching from next element
                                     after swapping
  } while (i <= j);

7.Swap arr [low] with arr [j]     //Swap pivot with last element in
                                   first part of the list
8.QuickSort( arr [n], low, j – 1)   // Apply quicksort on list
                                     left to pivot
9.QuickSort(arr [n], j + 1, high)    // Apply quicksort on list
                                     right to pivot
}
```

```
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

```
partition (arr[], low, high)
{
    pivot = arr[low];
     i = low + 1
     j = high
Do {
     while ( arr [i]<pivot and i<=high)      // search element higher than
     pivot
             i=i+1
     while ( arr [j]>pivot and j>=low )      // Search for an element
     smaller than pivot
             j=j-1
      If (i < j)
             Swap arr [i] with arr [j]
}while(i<=j)
swap(a[low],a[j])
 return (j)
}
```

Quicksort run-time

Worst case: already sorted (among others) –

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

$\Rightarrow = n + (n-1) + (n-2) + ... + 1 = n(n+1)/2=O(n^2)$

Best case: pivot is always median

$T(n)=n+2T(n/2)$ ->1 $\quad$ $T(n/2)=2T(n/2^2)+n/2$

$= 2[2T(n/2^2)+n/2]+n$

$=2^2T(n/2^2)+2n$ ->2 $\quad$ $T(n/2^2) =2T(n/2^3)+n/2^2$

$=2^2 (2T(n/2^3)+n/2^2)+2n$

$=2^3T(n/2^3)+3n$ $\rightarrow$3

$= 2^kT(n/2^k)+kn$ ---k time partition $n=2^k$ $\quad$ $k=\log n$

$= n+ n \log n= O(n\log n)$

Merge sort Algorithm

- Is based on the divide and conquer approach
- Divides the list into two sublists of sizes as nearly equal as possible
- Sorts the two sublists separately by using merge sort
- Merges the sorted sublists into one single list

Implementing Merge Sort Algorithm

- To understand the implementation of merge sort algorithm, consider an unsorted list of numbers stored in an array.

	0	1	2	3	4	5	6
arr	53	10	30	76	3	57	24

Implementing Merge Sort Algorithm (Contd.)

- Let us sort this unsorted list.

	0	1	2	3	4	5	6
arr	53	10	30	76	3	57	24

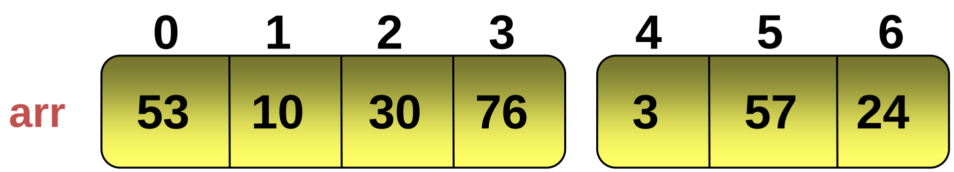
Implementing Merge Sort Algorithm (Contd.)

- The first step to sort data by using merge sort is to split the list into two parts.

	0	1	2	3	4	5	6
arr	53	10	30	76	3	57	24

Implementing Merge Sort Algorithm (Contd.)

- The first step to sort data by using merge sort is to split the list into two parts.



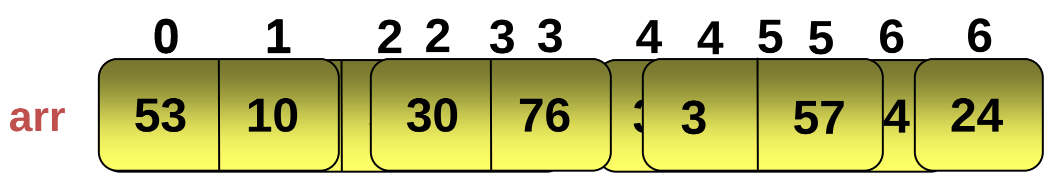
Implementing Merge Sort Algorithm (Contd.)

- The list has odd number of elements, therefore, the left sublist is longer than the right sublist by one entry.

	0	1	2	3	4	5	6
arr	53	10	30	76	3	57	24

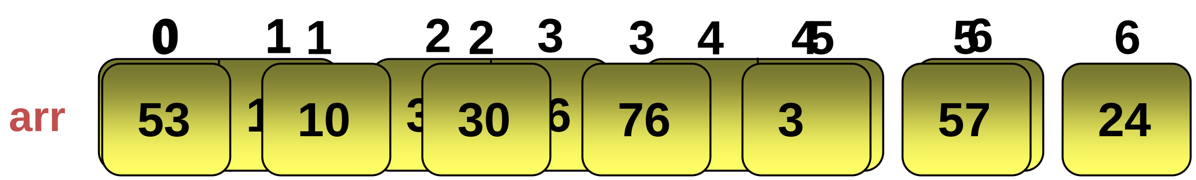
Implementing Merge Sort Algorithm (Contd.)

- Further divide the two sublists into nearly equal parts.



Implementing Merge Sort Algorithm (Contd.)

- Further divide the sublists.



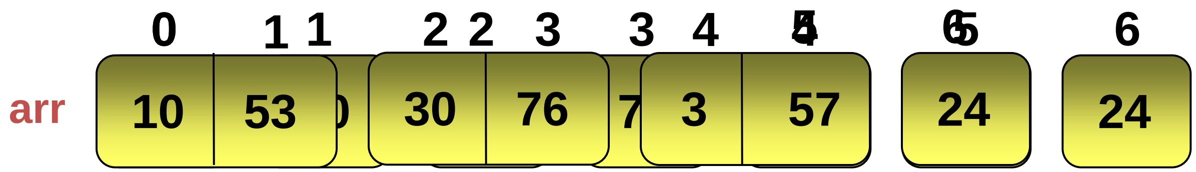
Implementing Merge Sort Algorithm (Contd.)

- There is a single element left in each sublist.
- Sublists with one element require no sorting.

	0	1	2	3	4	5	6
arr	53	10	30	76	3	57	24

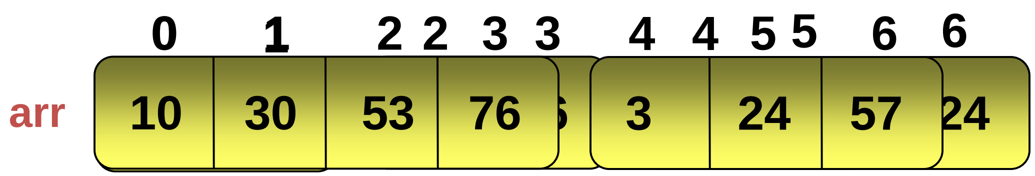
Implementing Merge Sort Algorithm (Contd.)

- Start merging the sublists to obtain a sorted list.



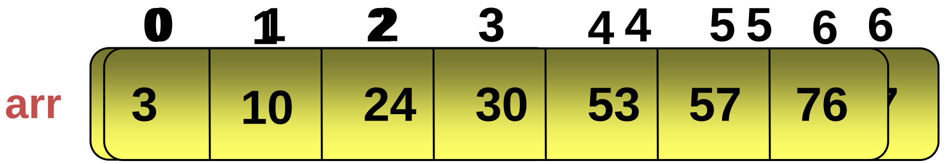
Implementing Merge Sort Algorithm (Contd.)

- Further merge the sublists.



Implementing Merge Sort Algorithm (Contd.)

- Again, merge the sublists.



Implementing Merge Sort Algorithm (Contd.)

- The list is now sorted.

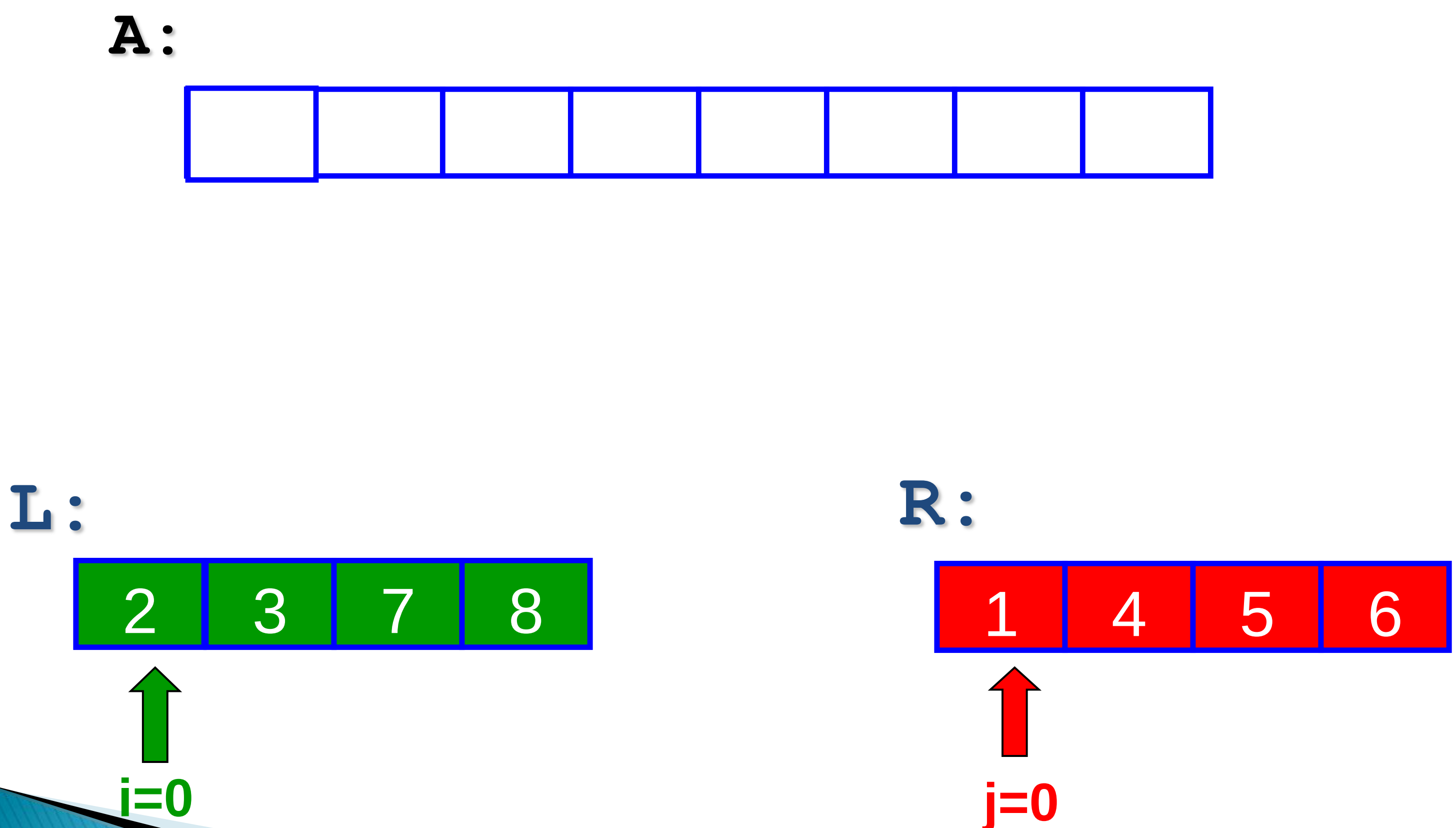
	0	1	2	3	4	5	6
arr	3	10	24	30	53	57	76

Write an algorithm to implement Merge sort:

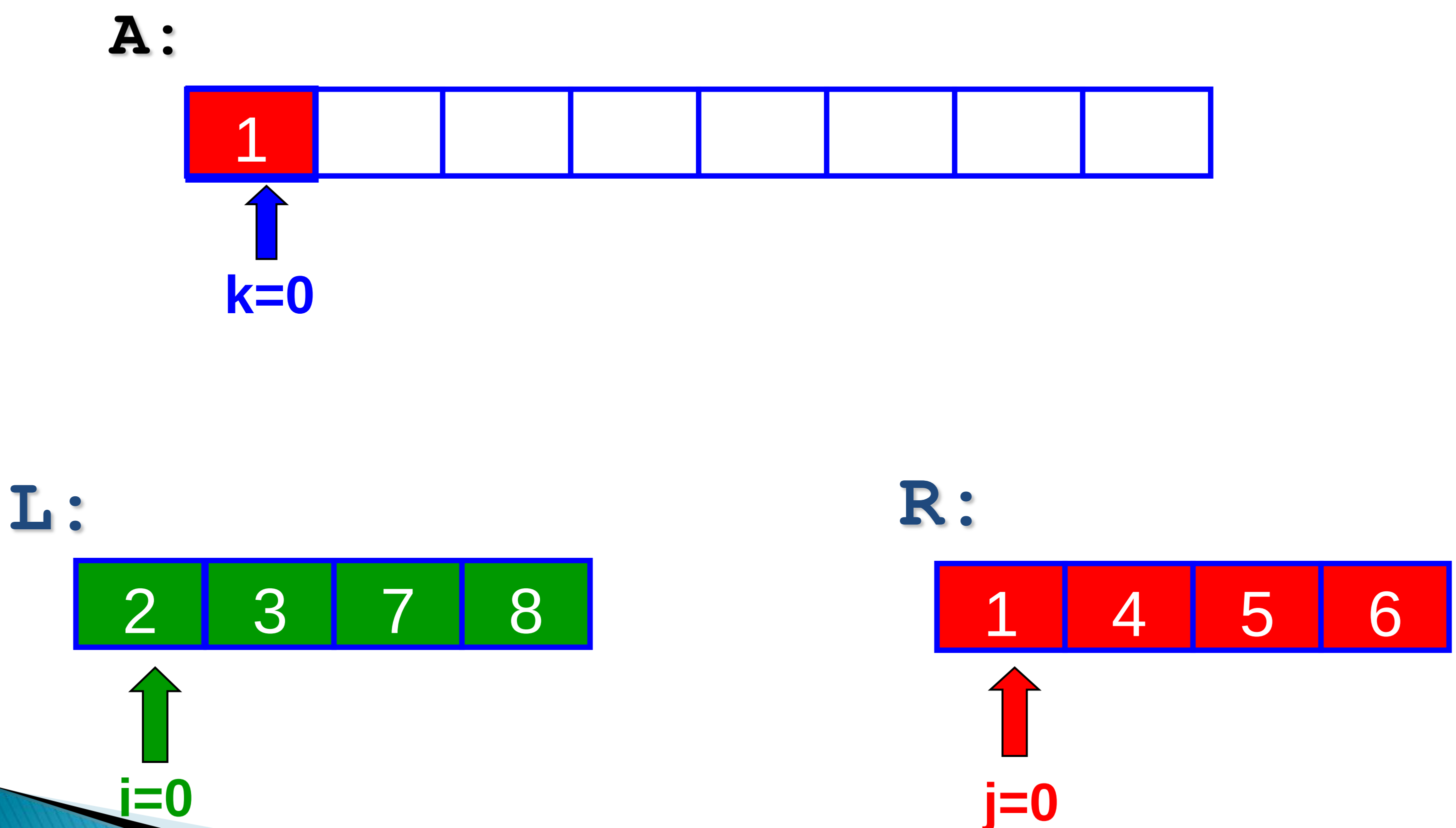
```
Merge_sort ( A[n], low, high )
  {
   //  A is an array of n elements,low is lower bound
       of an array and high is upper bound.
   1. if (low > = high)
      1.1 Return
   2. else
      2.1 mid = (low + high)/2
      2.2 Merge_sort ( A[], low , mid )
      2.3 Merge_sort (A[], mid+1 , high )
      2.4 Merge ( A[], low, mid, high )
```

Recursive Call

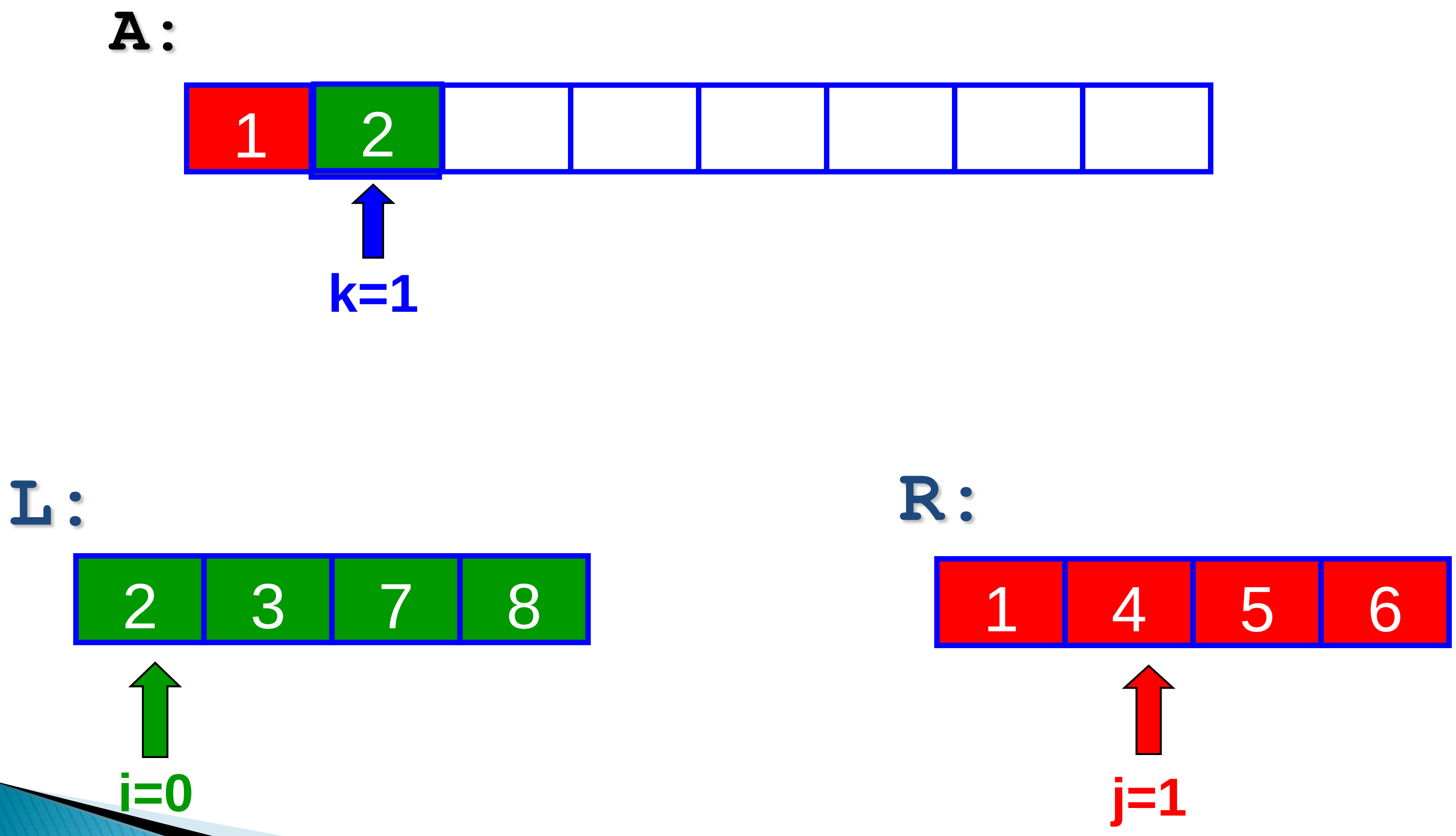
Merge-Sort: Merge Example



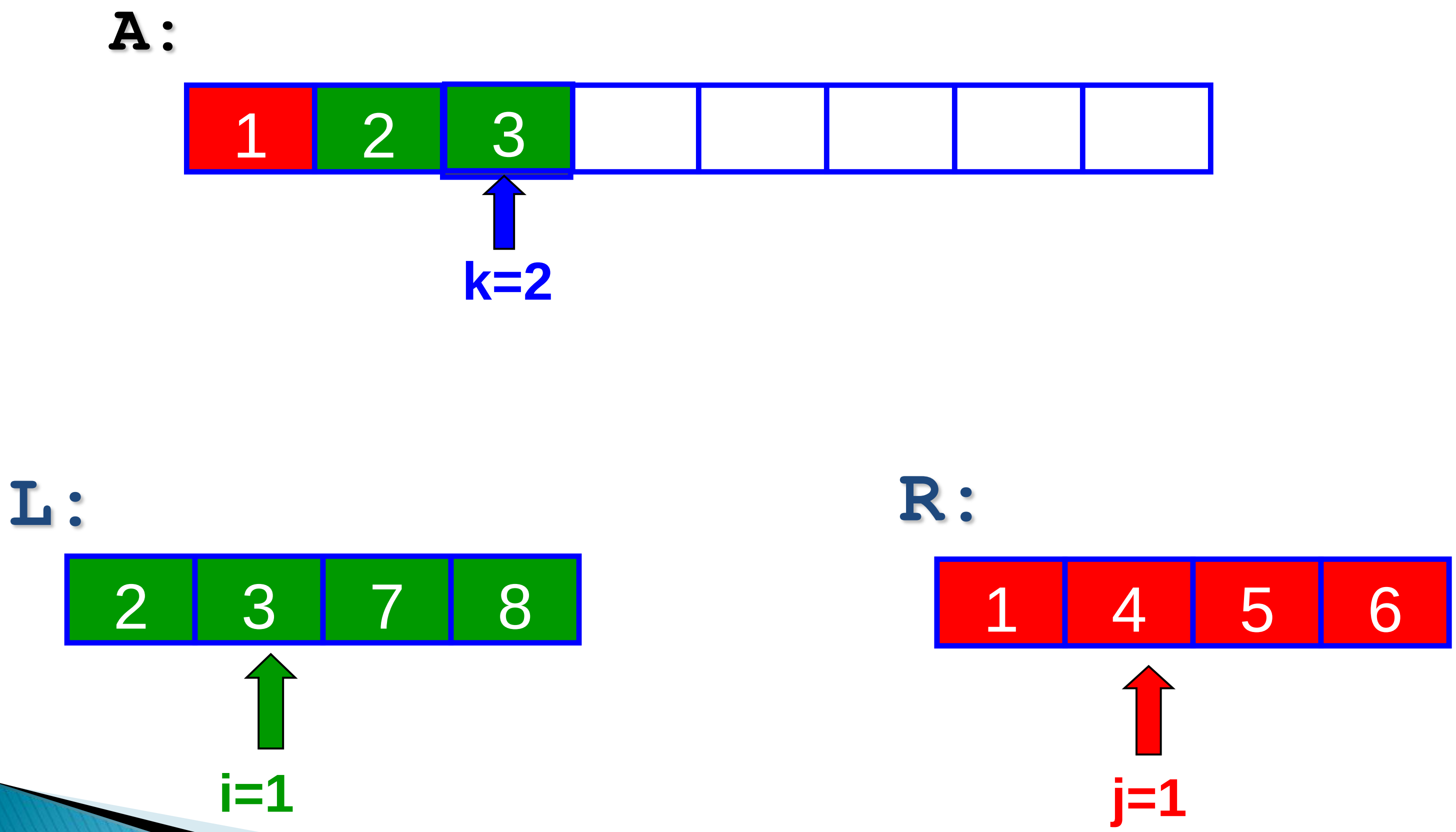
Merge-Sort: Merge Example



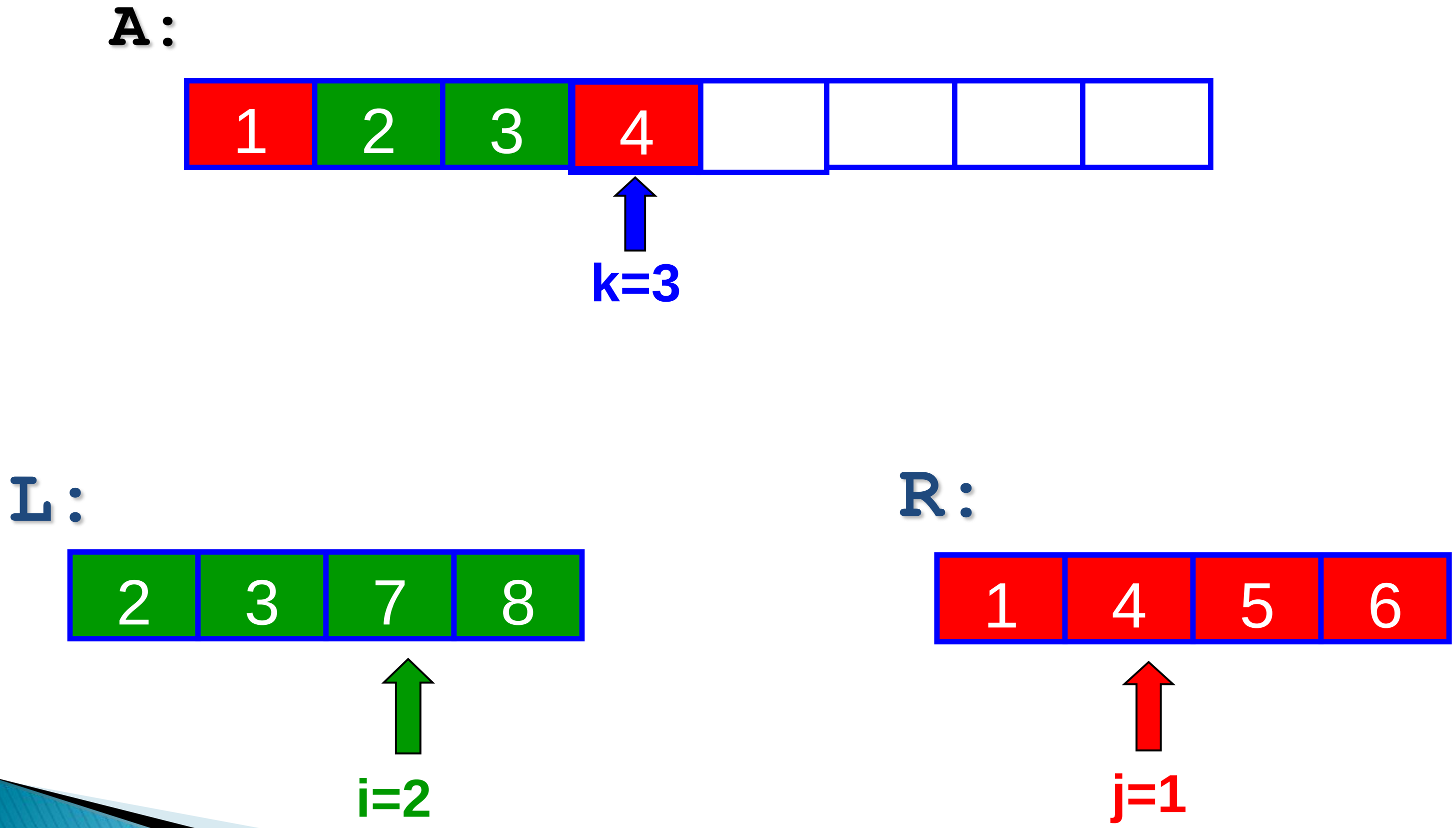
Merge-Sort: Merge Example



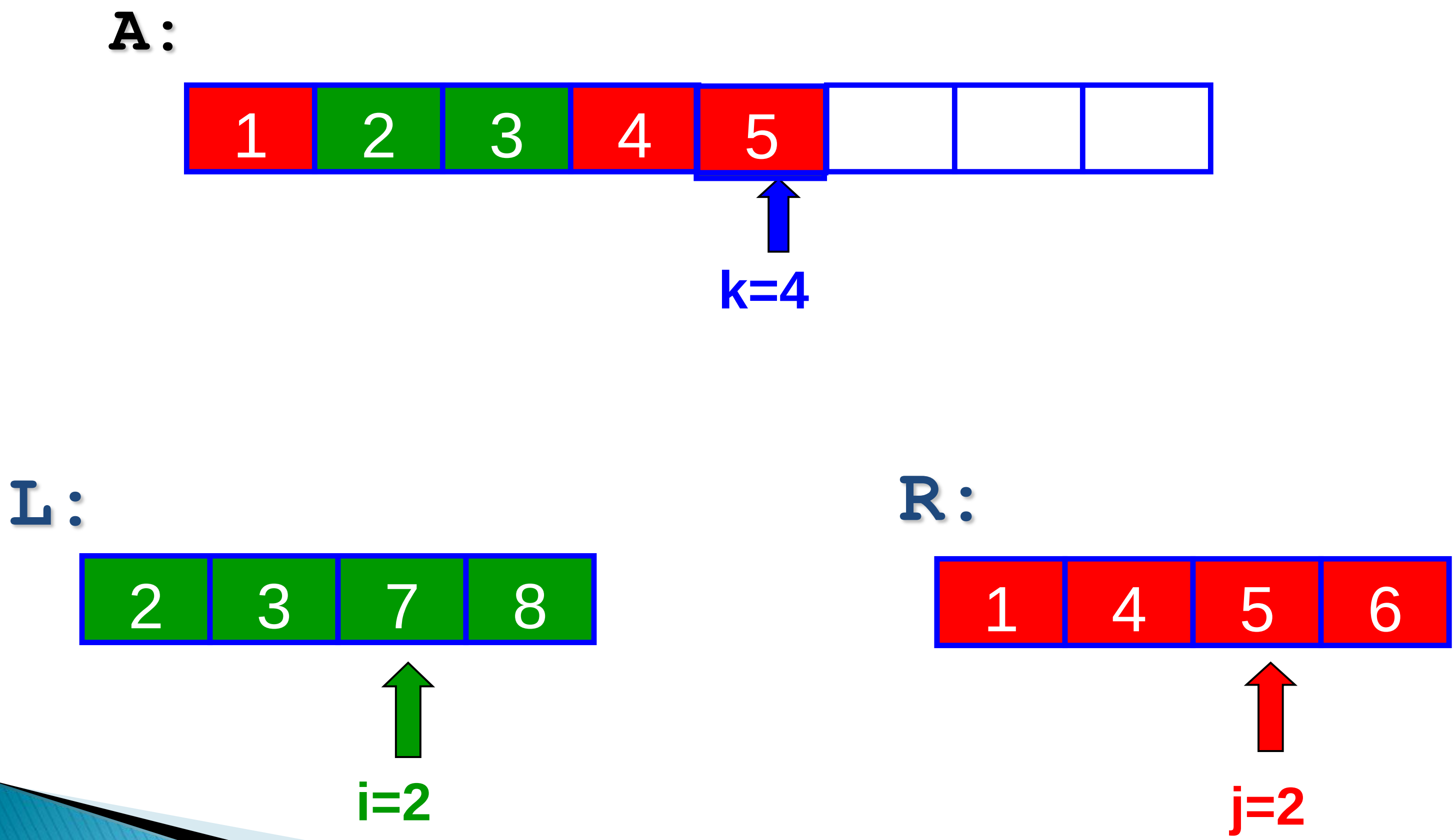
Merge-Sort: Merge Example



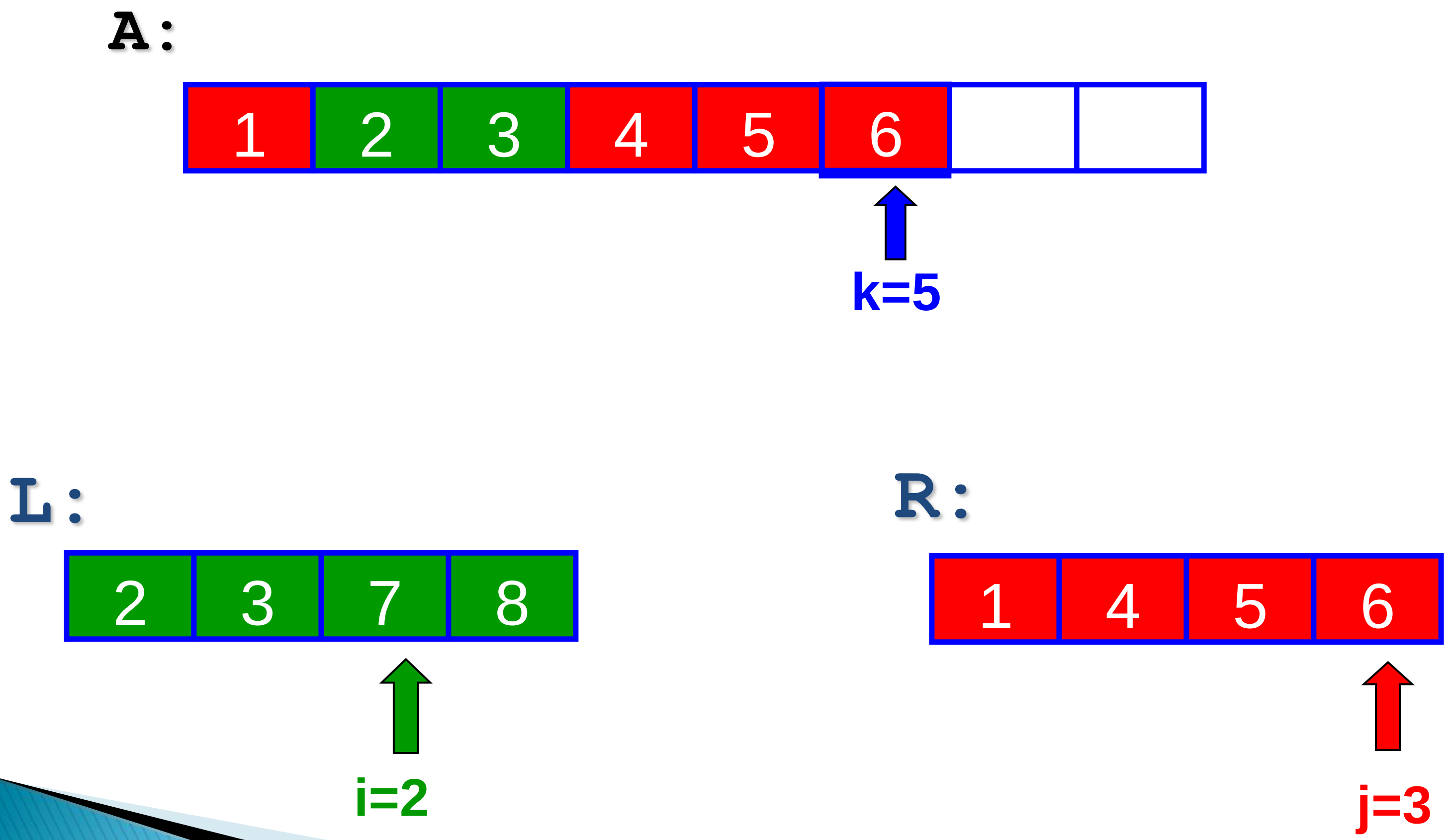
Merge-Sort: Merge Example



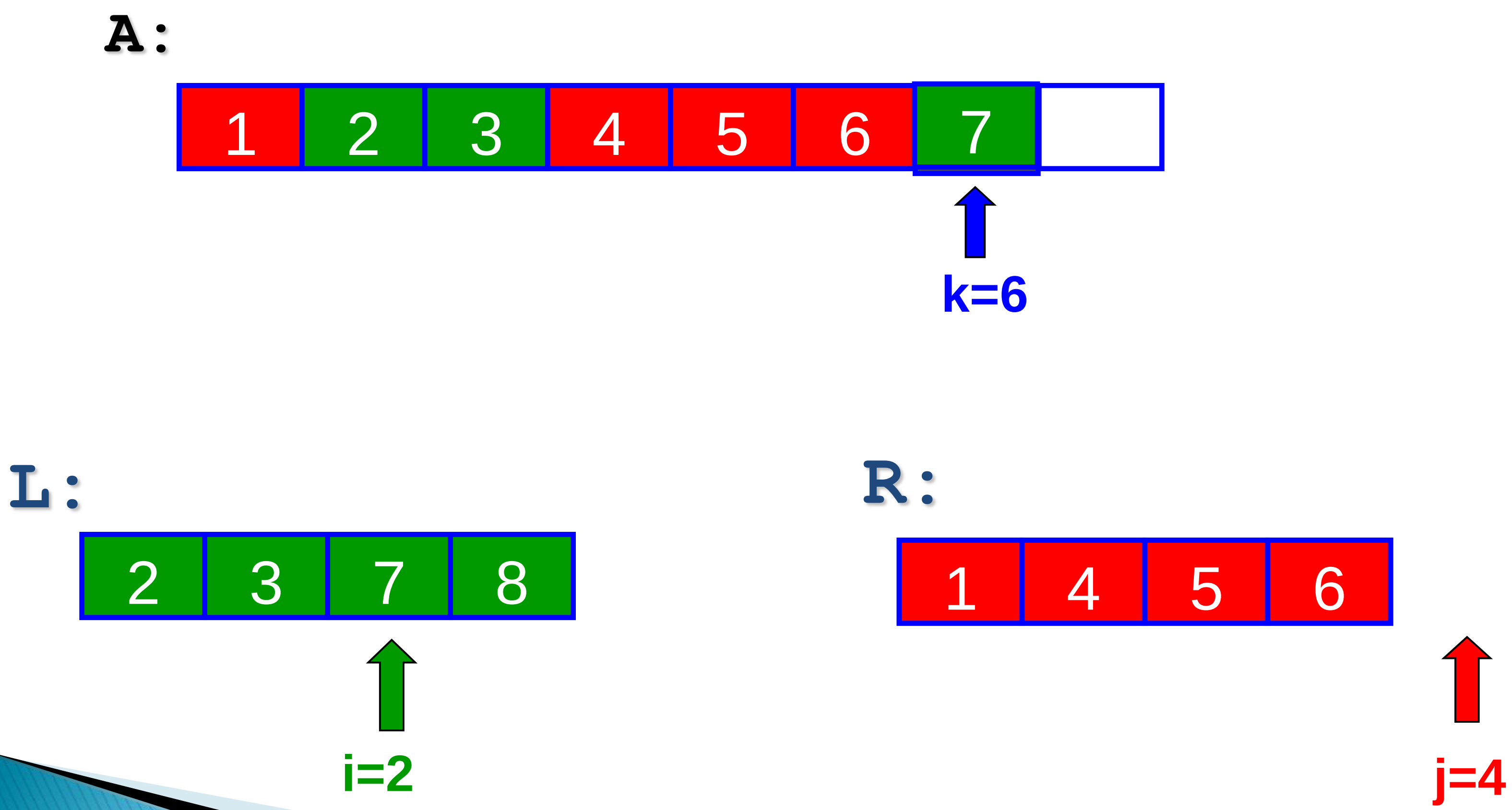
Merge-Sort: Merge Example



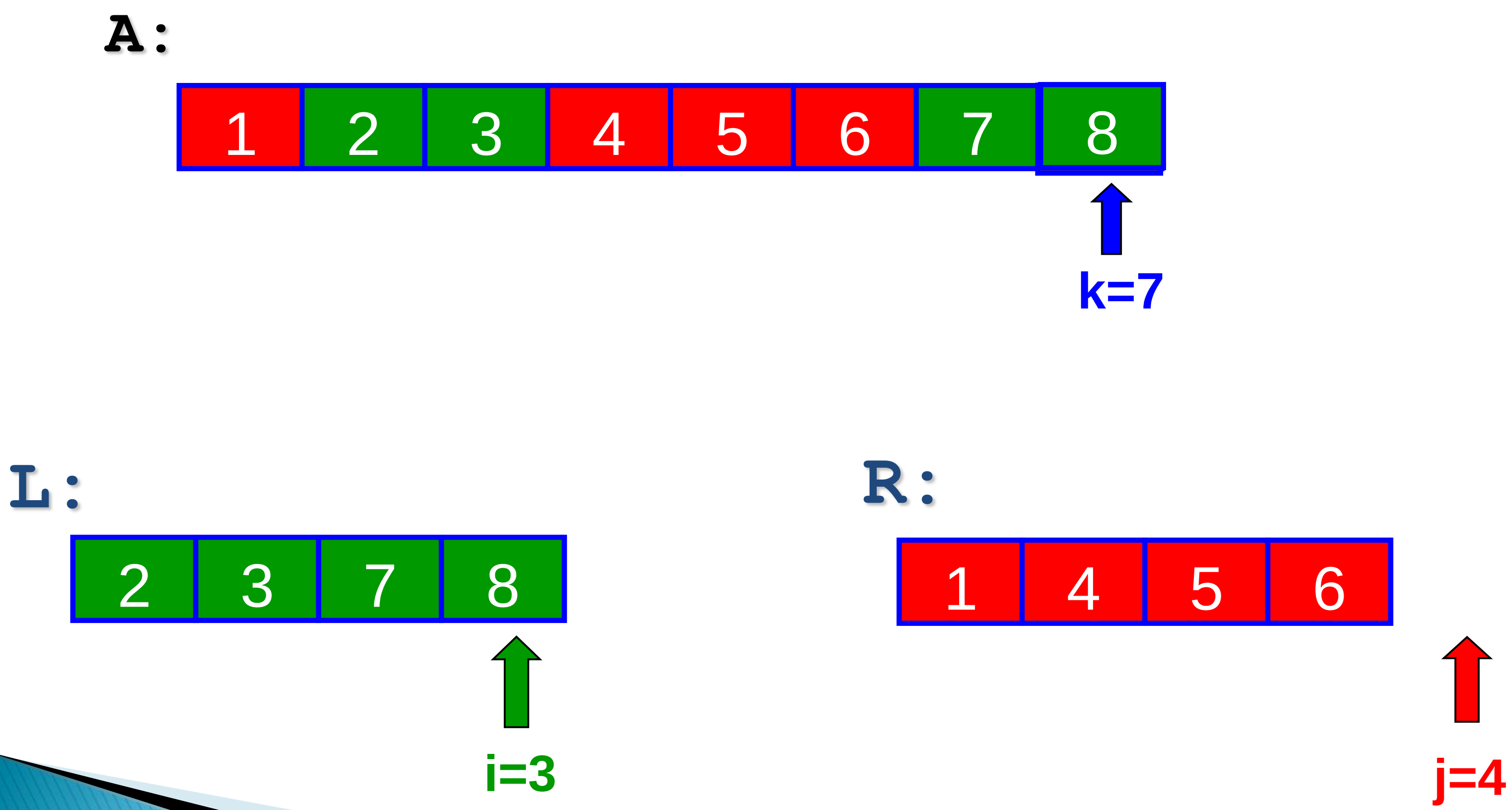
Merge-Sort: Merge Example



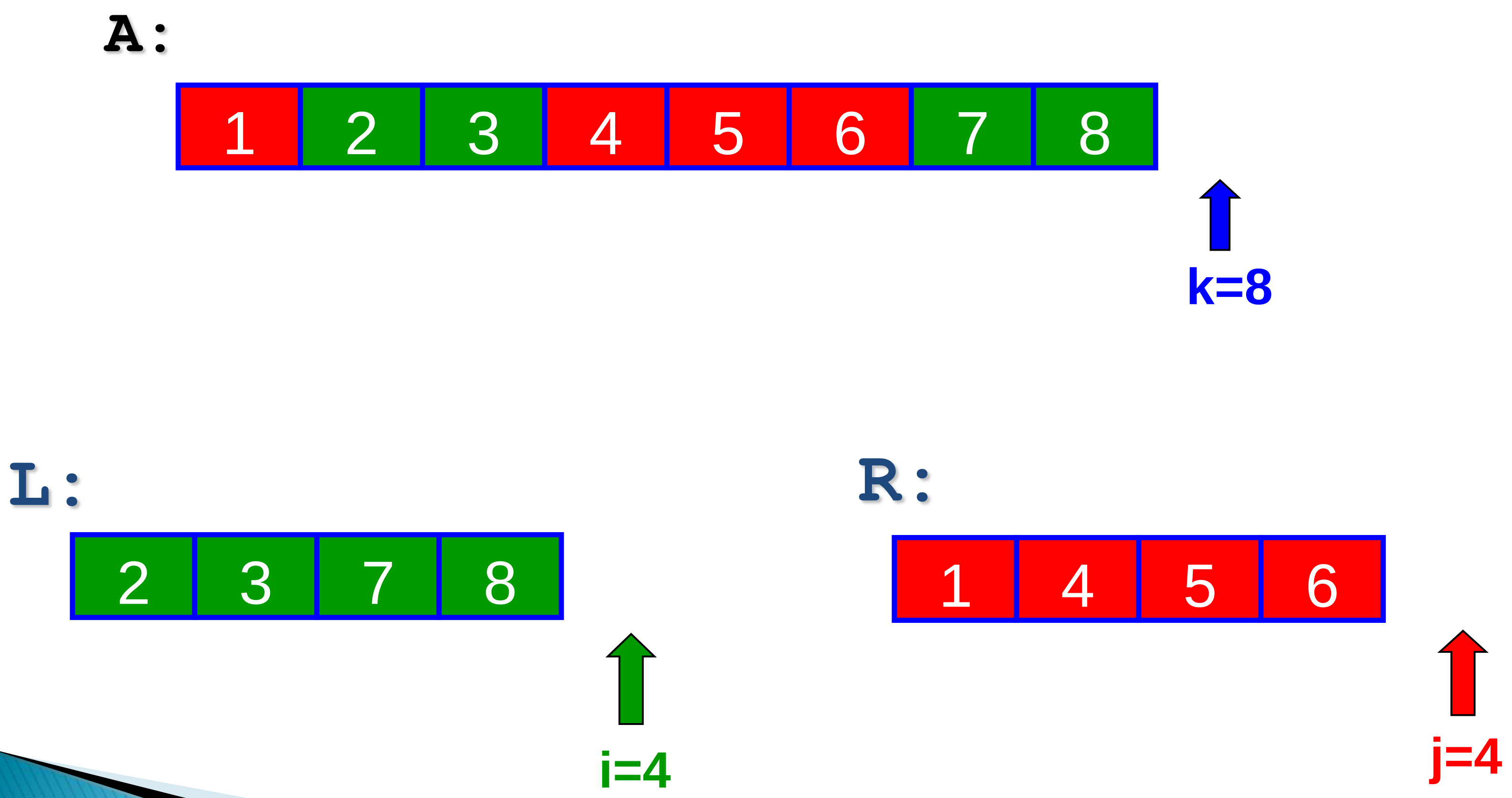
Merge-Sort: Merge Example



Merge-Sort: Merge Example



Merge-Sort: Merge Example



Algorithm:- Merge (A [n], low, mid, high)

```
{
    //A is an array of n elements, low is lower bound of
      an array and high is upper bound.
    1. i = low
    2. j = mid + 1
    3. k = low
    4. Do
       4.1 If( A [i] <= A [j])
           4.1.1  B [k] = A [i]
           4.1.2  i = i + 1
           4.1.3  k = k +1
       4.2 else
           4.2.1  B [k] = A [j]
           4.2.2  j = j +1
           4.2.3  k = k +1
       while (i  <= mid && j <= high);
```

6. While (j < = high)

6.1 B [k] = A [j]

6.2 j = j + 1

6.3 k = k +1

7. While(i <= mid)

7.1 B [k] = A [i]

7.2 i = i + 1

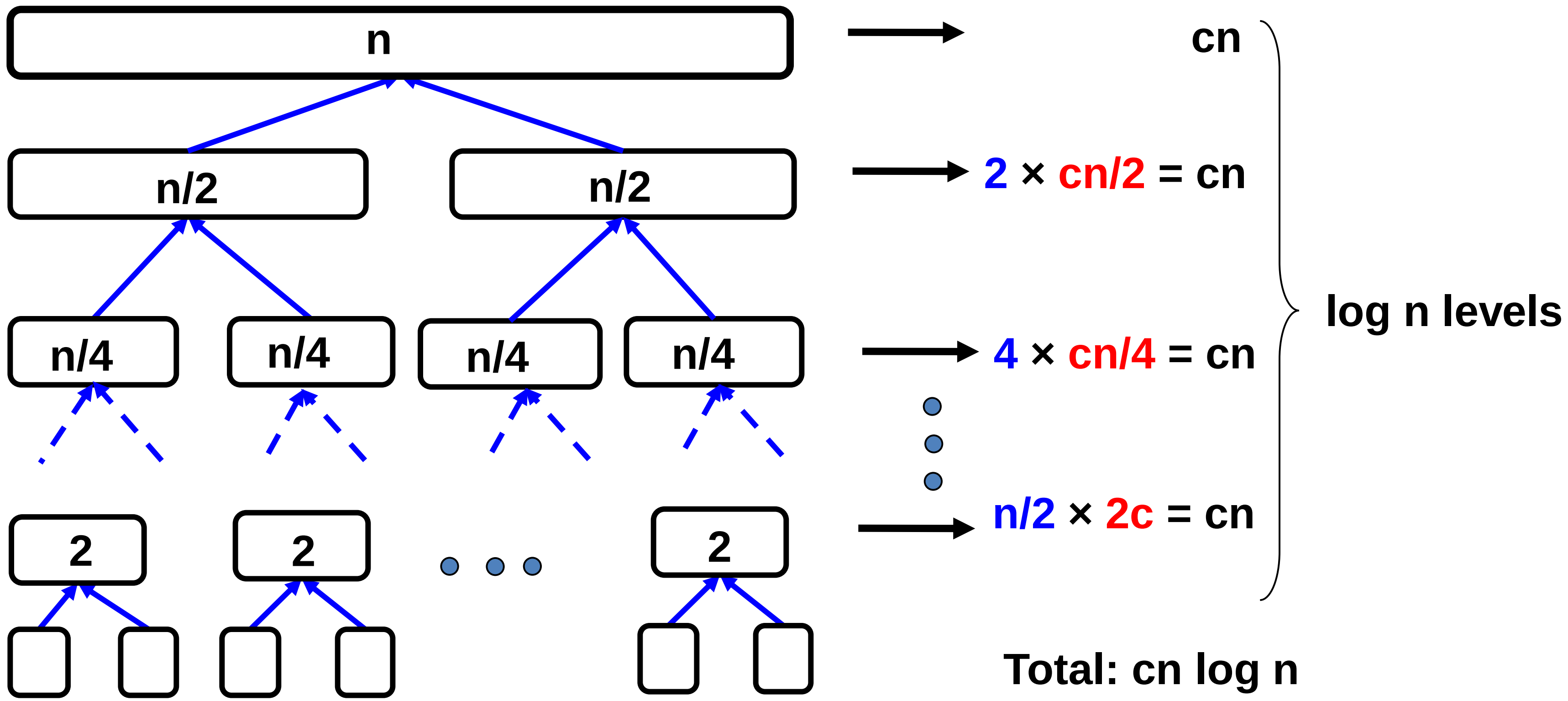
7.3 k = k +1

8. For (i=0 to n-1)

8.1 A [i] = B [i]

}

Merge-Sort Analysis



- Total running time: O (n logn)
- T(n)=2T(n/2)+Cn

Counting Sorting

- Sorting a collection of objects according to keys
- Sorts the elements of an array by counting the number of objects that have each distinct key value.
- Use arithmetic on those counts to determine the positions of each key value in the output sequence.
- It is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items

Counting sort

1. Array a [14] = 1 0 2 1 2 1 1 0 0 5 7 6 4 2
2. Create an array count of **key + 1** size
3. Initialize the array to zero

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

- **for** i=0 to n-1 count [i] = 0;

4. Fill the count array with the count of key values

0	1	2	3	4	5	6	7
3	4	3	0	1	1	1	1

for i=0 to n-1 count [a[i]] ++;

5. Update count array to identify the position of each element in result array

0	1	2	3	4	5	6	7
3	7	10	10	11	12	13	14

for i=1; i<=k ; i++ count [i] = count [i] + count [i-1];

6. For maintaining the stability of the array, start sorting the array from last element. 1 0 2 1 2 1 1 0 0 5 7 6 4 2

1. Refer to original array from last element
2. Go to the index (given by array value) in updated count array
3. Read position
4. Decrement it by one, as its position not index.
5. In an output array, write the current value of original array in position derived

0	1	2	3	4	5	6	7
3	7	10	10	11	12	13	14

B[] =

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	1	1	1	1	2	2	2	4	5	6	7

Pseudo Code

- count = array of k+1 zeros // k is max element, n is no of elements
- for i=0 to n−1 do
-     count[input[i]] += 1
- total = 0
- for i =1 to k do
-       count [i] = count [i] + count [i−1];
- output = array of the same length as input
- for i=n−1 to 0 do
-     output[−−count[input[i]]] = input[i]
- return output

RADIX SORT

- Radix sort is generalization of bucket Sort.
- To Sort the decimal numbers where the radix or base is 10,we need 10 buckets.
- In the first pass least significant digit are stored in the particular bucket.
- In the second pass numbers are sorted on the second least significant digit.
- At the end of every pass numbers in buckets are merged to produce a common list.

Algorithm for Radix sort

1. large=find the largest number in a[]
2. passes=number of digits in large
3. div=1 /* divisor for extracting the least significant digit */
4. for(i=1;i<=passes;i++)
    1. initialize all buckets b0 to b9
    2. for each number x from a[0] to a[N-1]

Bucket_no=(x/div)%10

Insert x in buckets with bucket_number

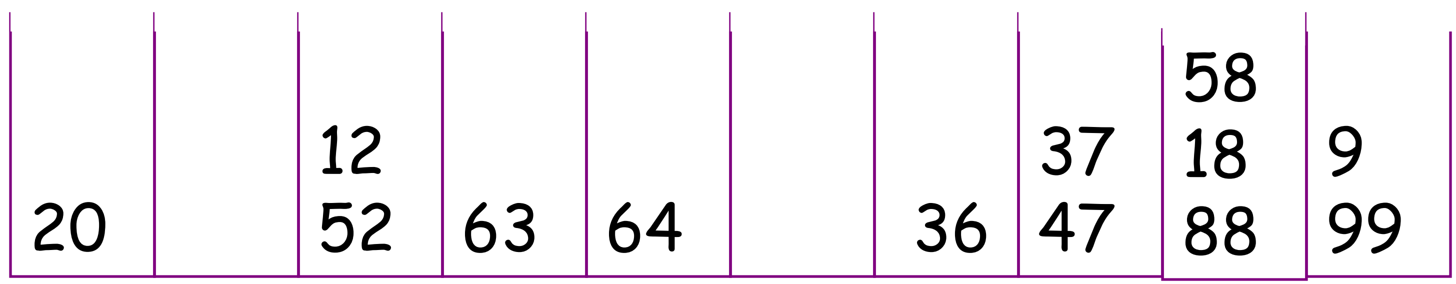
Copy elements of buckets back in array a []

Div=Div*10

5.exit

Example: first pass

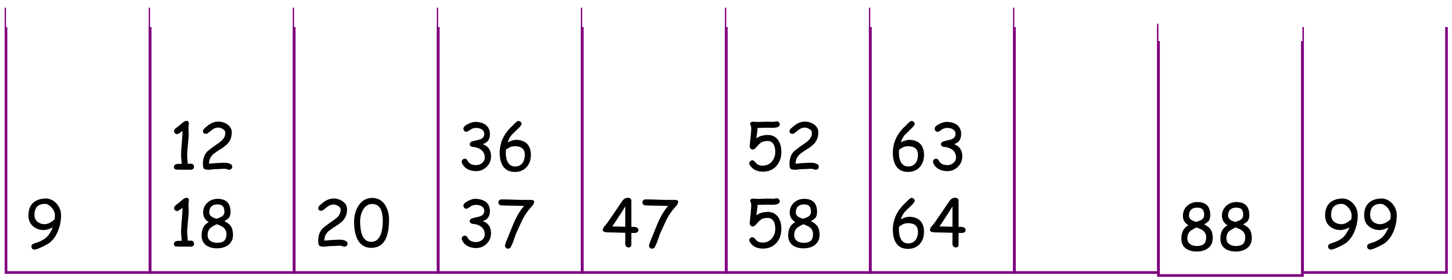
12 58 37 64 52 36 99 63 18 9 20 88 47



20 12 52 63 64 36 37 47 58 18 88 9 99

Example: second pass

20 12 52 63 64 36 37 47 58 18 88 9 99



9 12 18 20 36 37 47 52 58 63 64 88 99

Example: 1st and 2nd passes

12 58 37 64 52 36 99 63 18 9 20 88 47

sort by rightmost digit

20 12 52 63 64 36 37 47 58 18 88 9 99

sort by leftmost digit

9 12 18 20 36 37 47 52 58 63 64 88 99

RADIX SORT

- n= number of element in the array a[]
- a[] = array holding elements
- Buckets are represented using a two dimensional array buckets [10][10].each bucket can hold upto ten value. another array count[10] gives number of element in the corresponding bucket.
- If bucket[1] has five elements then the value of count[1] will be five
- Whenever an element is inserted in ith bucket count[i] should be incremented by 1.

```
Algorithm radixsort(arr[],n,buck[],bucket[][])
{
   for i=0 to n-1  do {
          if(arr[i] > large) large = arr[i];
    }
  while(large > 0)
   {
    num++;
    large = large/10;
    }
```

```
for(passes=0 ; passes < num ; passes++)
 {
   for(k=0 ; k < 10 ; k++)
   {
    buck[k] = 0;
   }
   for(i=0 ; i < n  ;i++)
  {
   l = ((arr[i]/div)%10);
   bucket[l][buck[l]++] = arr[i];
   }
```

```
 i=0;
for(k=0 ; k< 10 ; k++)
{
for(j=0 ; j < buck[k] ; j++)
{
 arr[i] = bucket[k][j];
   i++;
  }
}
div*=10;
}
}
```

Complexity

Range of digit is from 1 to k digits

There are d passes...counting sort time is O(n+k)

Complexity of radix sort is O(nk+nd)= O(nd)

Space complexity O(n+k)

Bucket Sort

- It is assumed that each integer is between 0 and M.
- B[1 . . . n] is an array of buckets (for a total number of n buckets) which is implemented using linked lists.
- Each input element is inserted into a bucket B[n · A[i]/M].
- They are then sorted with the insertion sort

Bucket Sort

```
Algorithm  BUCKET-SORT ( A[], n )
{ // n is number of elements , 0 to M is range of
elements
for i =1 to n do
   insert A[i] into list B[n · A[i]/M]
end for
for i = 0 → n − 1 do
sort list B[i] with insertion sort
end for
Concatenate lists B[0],B[1], . . ., B[n − 1] together
}
```

Types of sorting

- Internal sorting
- External sorting

Sorting concepts

Stable -bubble,selection,insertion,merge
unstable-quick ,heap,shell sortng
Sort eficiency-best case,average , worst case
passes