Merge- and Quick Sort

- **Merge Sort**
- **Quick Sort**
- **Exercises**

Merge Sort

Merge Sort uses the algorithmic paradigm of **divide and conquer:**

- **Divide** the block into two subblocks of "equal" size;
- **Merge Sort** each block **recursively**
- **Merge** the two sorted sub-blocks (next to each other) to give a sorted block.

The following is the merge sort algorithm to sort a sub-array from index low to index high:

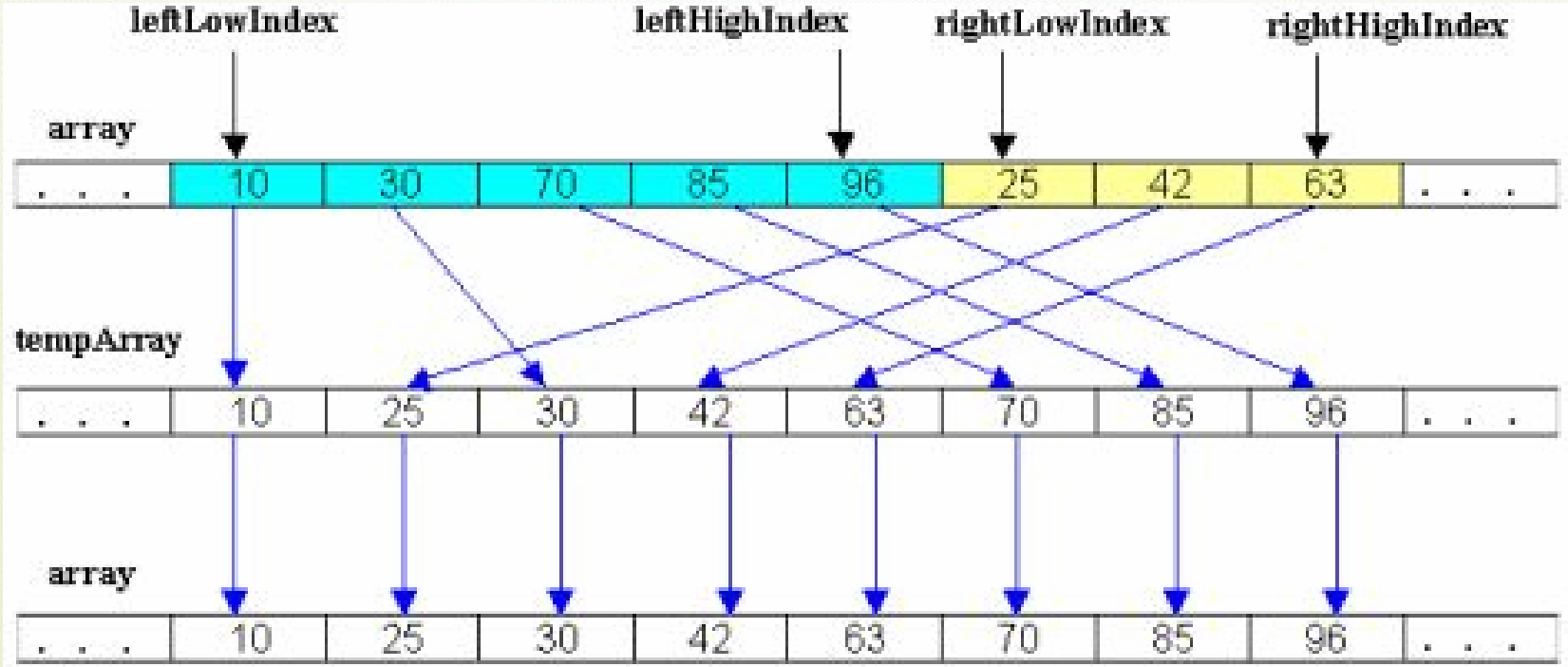
```
if(high > low){
    Split the sub-array into two halves
    merge sort the left half
    merge sort the right half
    merge the two sorted halves into one sorted array
}
```

- The above algorithm translates to the following pseudo-code:

```
mergeSort(array, low, high){
    middle = (low + high) / 2;
    mergeSort(array, low, middle);
    mergeSort(array, middle + 1, high);
    merge(array, low, middle, middle + 1, high);
}
```

Merge Sort (cont'd)

- Note that in a merge sort, the splitting and merging process are intermingled. However, we can view merge sort as:
  - 1. Continually splitting the original array of size n until it has created n one element subarrays (each of which is a sorted subarray).
  - 2. Adjacent sorted subarrays are then merged into larger and larger sorted subarrays, until the entire array is merged back.
- Example: merging two adjacent, sorted portions of an array



Merge Sort (cont'd)

The merge Sort algorithm is implemented for sorting an array of objects based on their natural ordering. Another version can be implemented for sorting according to a comparator object.

```
public static void mergeSort(Object[] array, Comparator comp){
      mergeSort(array, 0, array.length - 1, comp);
}
private static void mergeSort(Object[] array, int low, int high,
 Comparator comp){
  if(last > first){
    int mid = (low + high)/2;
    mergeSort(array, low, mid, comp);
    mergeSort(array, mid + 1, high, comp);
    merge(array, low, mid, mid + 1, high, comp);
  }
}
```

Merge Sort (cont'd)

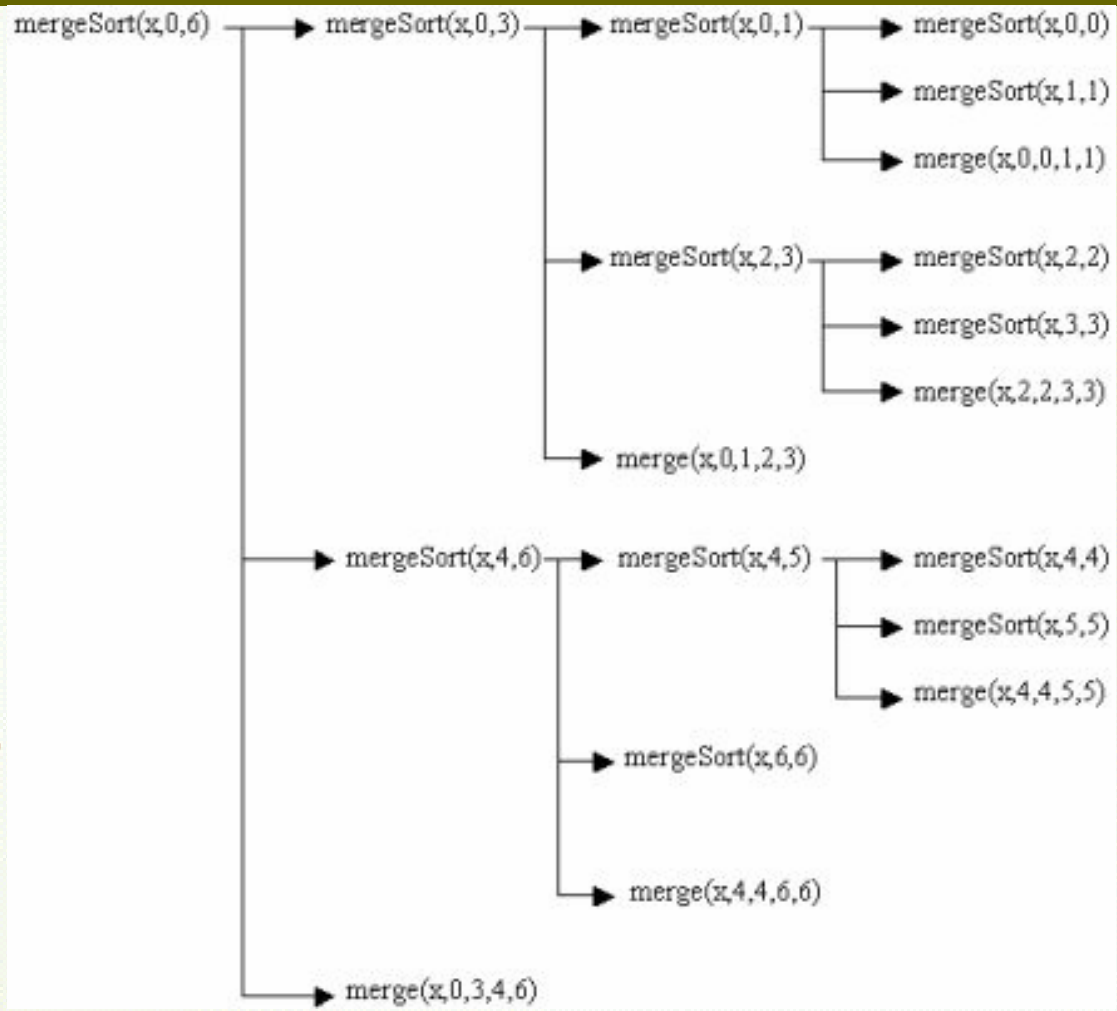
In Merge Sort algorithm, the main work is done by the merge method as shown below.

```
private static void merge(Object[] array,int leftLowIndex, int leftHighIndex,int
    rightLowIndex,  int rightHighIndex,  Comparator comp){
    int low = leftLowIndex;
    int index = leftLowIndex;
    Object[] tempArray = new Object[array.length]; // temporary array
    while(leftLowIndex <= leftHighIndex &&   rightLowIndex <= rightHighIndex){
     if(comp.compare(array[leftLowIndex], array[rightLowIndex]) < 0)
                tempArray[index++] = array[leftLowIndex++];
     else        tempArray[index++] = array[rightLowIndex++];
     }
     while (leftLowIndex <= leftHighIndex)
                tempArray[index++] = array[leftLowIndex++];
     while (rightLowIndex <= rightHighIndex)
                tempArray[index++] = array[rightLowIndex++];
     for(int k = low; k <= rightHighIndex; k++)
                array[k] = tempArray[k];
}
```

Merge Sort (cont'd)

- A recursive-tree for merge sorting an array x with 7

```
mergeSort(array, low, high){
    if(high > low){
        middle = (low + high) / 2;
        mergeSort(array,low,middle);
        mergeSort(array, middle + 1,
            high);
        merge(array, low, middle,
            middle + 1, high);
    }
}
```



Quick Sort

- Quick Sort also uses the algorithmic paradigm of divide and conquer.
- The following is a quick sort algorithm to sort an array from index low to high:

```
if(high > low){  // number of elements in the sub-array greater then 1 (condition for recursive step)
  1. Select a pivot element p to be used to partition the array into two partitions.
  2. Scan through the array, moving all elements smaller than p to the lower partition,
       and all elements equal to or greater than p to the upper partition.
  3. Sort the low and upper partitions (without the pivot element) recursively using quick sort.
 }
```

- The above algorithm can be translated to the following pseudo-code:

```
quickSort(array, low, high){
  if(high > low){
        elect a pivotValue
        partition the array so that:
              array[low]...array[pivotIndex –1] < pivotValue
               array[pivotIndex] = pivotValue
               array[pivotIndex + 1]...array[high] >= pivotValue
               quickSort(array, low, pivotIndex –1);
               quickSort(array, pivotIndex + 1, high);
    }
}
```

Quick Sort (cont'd)

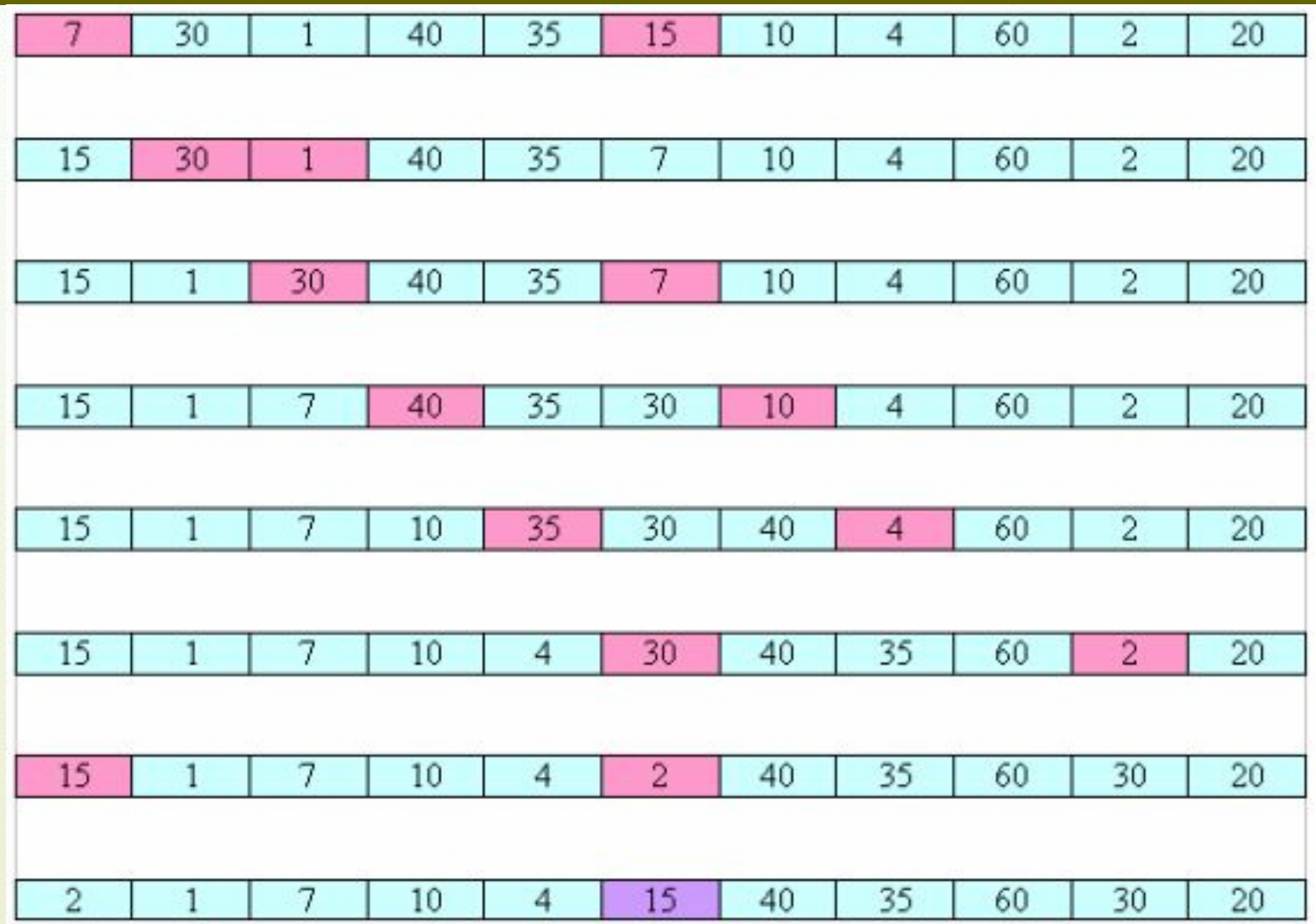
- Any array element can be selected as pivot; but ideally the pivot should be the median value. Here the pivot is taken as the middle value of the sub-array.
- Quick sort is most efficient when the pivot is as close to the median as possible; otherwise the depth of recursion increases resulting in a decrease in efficiency.
- Computing median values in each partition degrades the performance of quick sort.
- The method implemented here sorts based on the natural ordering. The version using the comparator object can be implemented too.
- The Quick sort method is:

```
public static void quickSort(Comparable[] array ){
 quickSort(array, 0, array.length - 1, comp);
}
private static void quickSort(Comparable[] array, int low,int high ){
 if(low < high){
     int pivotIndex = partition(array, low, high, comparator);
     quickSort(array, low, pivotIndex - 1);
     quickSort(array, pivotIndex + 1, high);
}
}
```

Quick Sort (cont'd)

```
private static int partition(Comparable[] array, int firstIndex,
int lastIndex ){
   int middleIndex = (firstIndex + lastIndex)/2;
   Comparable pivotValue = array[middleIndex];
   /* Temporarily place pivotValue into first position of current
   partition */
   swap(array, firstIndex, middleIndex);  //implementation not shown
   int pivotIndex = firstIndex;
   int index = firstIndex + 1;
   while(index <= lastIndex){ // check all elements if < or >= to pivot
        if(array[index].compareTo(pivotValue) < 0){
          ++pivotIndex;
          swap(array, pivotIndex, index);
        }
        index++;
   }
   // Return pivotValue to partition point
   swap(array, firstIndex, pivotIndex);
   return pivotIndex; // return location of pivot after partitioning done
}
```

Quick Sort (cont'd)



Exercises

1. What is the advantage of the bubble sort algorithm we have studied over all the other sorting methods we have studied?
2. When are bubble, insertion, and selection sort methods more efficient than merge sort and quicksort?
3. What is the disadvantage of merge sort as compared to quicksort?
4. What role, if any, does the size of data objects play in a sort?
5. Give the merge sort recursion-tree for the array:

   43, 7, 10, 23, 18, 4, 19, 5, 66, 14, 2
6. A merge sort is used to sort an array of 1000 integers in descending order. Which of the following statements is true?

   (a) The sort is fastest if the original integers are sorted in ascending order.

   (b) The sort is fastest if the original integers are sorted in descending order.

   (c) The sort is fastest if the original integers are completely in random order.

   (d) The sort is the same, no matter what the order of the original integers.
7. A different method for choosing the pivot for each partition in quicksort is to take the median of the first, last and central keys. Implement a quicksort algorithm using this approach.
8. A different approach to the selection of a pivot in quicksort is to take the average of all the keys in a partition (assuming that the keys are numeric) as the pivot for that partition. The resulting algorithm is called **meansort**. Implement a quicksort algorithm using this approach. Note: The average of the keys is not necessarily one of the keys in the array.

Exercises (cont'd)

9. In quicksort, why is it better to choose the pivot from the center of the array rather from one of the ends?
10. What is the property of the best pivot in a quicksort?
11. What is the property of the worst pivot in a quicksort?
12. Suppose that, instead of sorting, we wish only to find the m^{th} quicksort can be adapted to this problem, doing much less work than a complete sort.
13. Implement a non-recursive mergesort, where the length of the array is a power of 2. First merge adjacent regions of size 1, then adjacent regions of size 2, then adjacent regions of size 4, and so on.
14. Implement a non-recursive mergesort, where the length of the array is an arbitrary number.

    Hint: Use a stack to keep track of which subarrays have been sorted.

15. Implement each of the sorting algorithms we have studied such that each displays the number of swaps and comparisons. Each of the implementations should be used to sort integer arrays with the same 100 random integers.

Sort method	Number of swaps	Number of comparisons
Bubble sort		
Selection sort		
Insertion sort		
Merge sort		
Quick sort		

Exercises (cont'd)

16. Design a test program to compare the execution times of bubble sort, selection sort, insertion sort, merge sort, and quick sort on integers arrays of equal lengths with similar random integers. For shorter lengths, sort many arrays and obtain the average execution time.