# Algorithms and Data Structures 1
# CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Textbook slides and Dr. Ramirez's CS 0445 slides)

# Announcements

- Upcoming Deadlines:
  - Lab 9: next Monday 11/21 @ 11:59 pm
  - Homework 9: next Monday 11/21 @ 11:59 pm

# Sorting Algorithms

- $O(n^2)$
  - Selection Sort
  - Insertion Sort
  - Shell Sort
- $O(n \log n)$
  - Merge Sort
  - Quick Sort
- $O(n)$ Sorting
  - Radix Sort

# Muddiest Points

- **Q: What makes insertion sort stable?**
- Assume two items i and j are equal and i is before j in the original array.
- Since Insertion Sort iterates over the array from left to right, it is going to insert item i into the sorted region before item j
- When item j gets inserted later, the loop inside insertInOrder will stop right after item i
- Item j will then be inserted after item i

# Muddiest Points

- **Q: What makes selection sort unstable?**
- Assume items i and j are equal and i is before j in the original array.
- Consider the moments when item i is swapped with the item x and item j is swapped with item y
  - If item x is after item y in the original array, after swapping item i becomes after item j
- When item i and item j become the smallest items in the unsorted array, they will be swapped into the sorted subarray in their current relative order

# Muddiest Points

- **Q: Why can't we make selection sort be stable?**
- Well, we can!
- Let’s keep track of the original position of each item in a separate array
  - updated that array with every swap
- When breaking ties in findSmallestItem, use the original position

# Muddiest Points

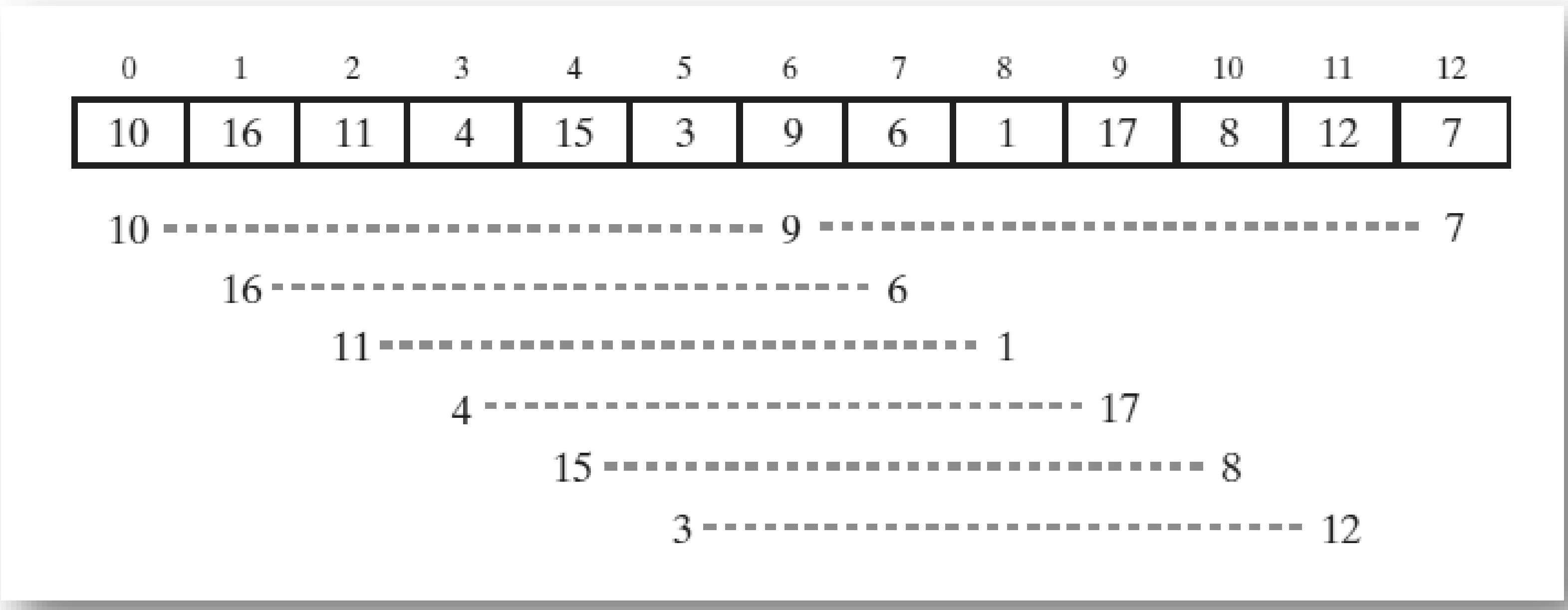
- **Q: I still can not understand why we do not use <T> after creating the constructor or why do we have it in the first place**
- Without the constructor, Node has to be a static class
- Thus, it cannot use the non-static data type parameter T of class SortingAlgorithms<T>
    - We had to define a separate static type parameter for Node
        - class Node<S> { … }
- With the constructor, Node is made non-static, and can use the same type parameter of SortingAlgorithms<T>

# Shell Sort

- Algorithms seen so far are simple but inefficient for large arrays
- Improved insertion sort developed by Donald Shell

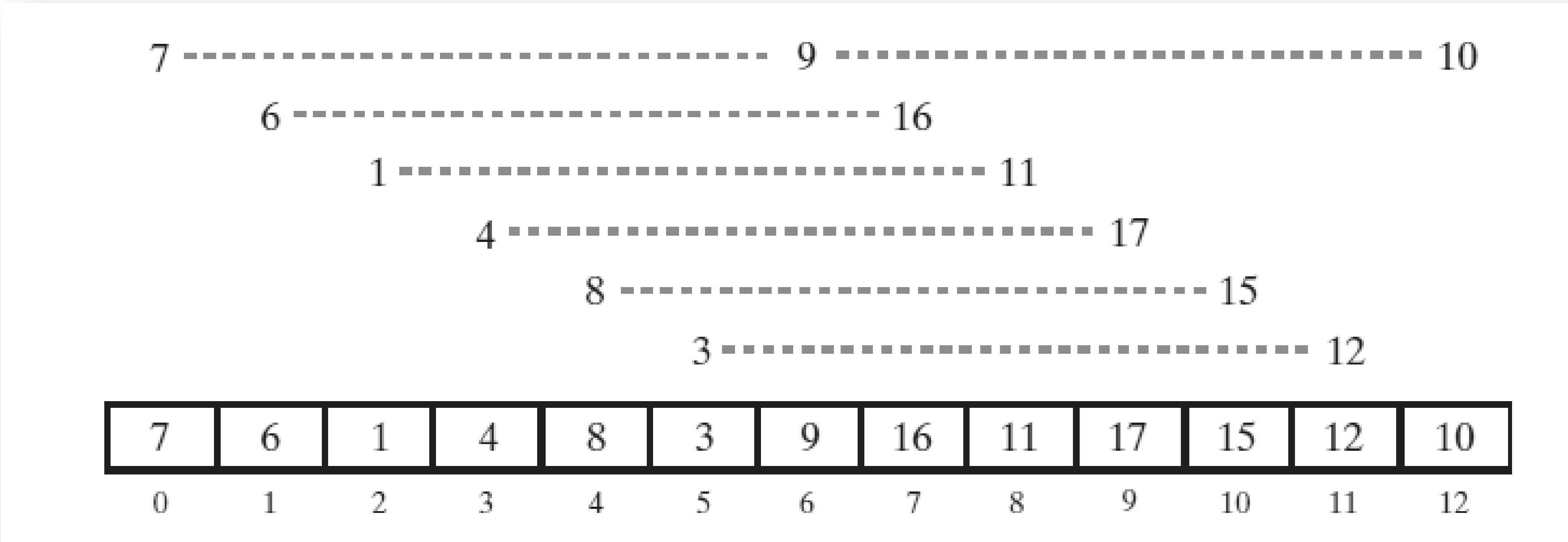
# Shell Sort

- An array and the subarrays formed by grouping entries whose indices are 6 apart.



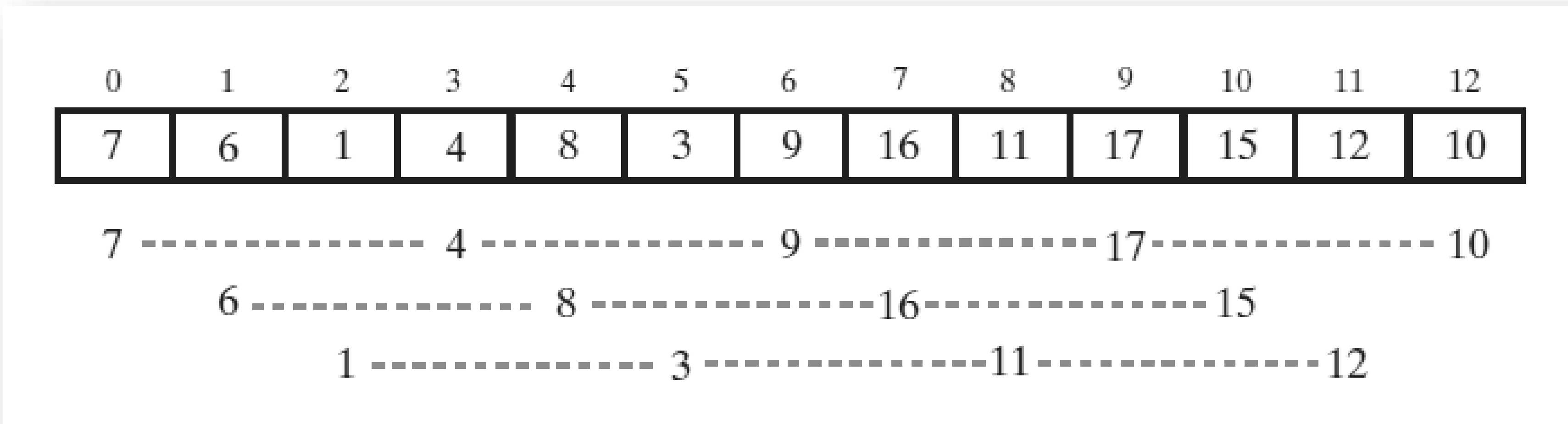
# Shell Sort

- The subarrays of after each is sorted, and the array that contains them



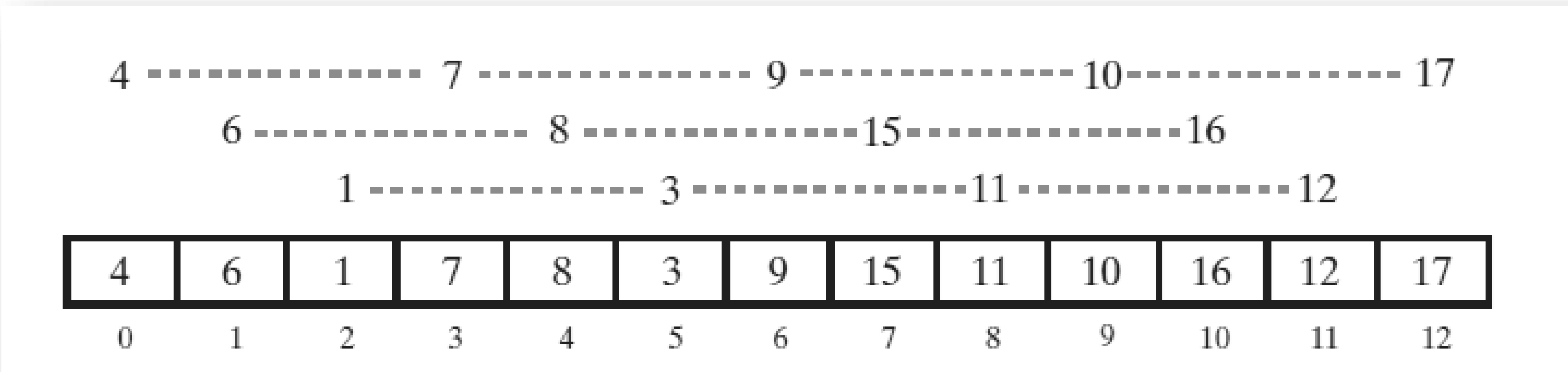
# Shell Sort

- The subarrays formed by grouping entries whose indices are 3 apart



# Shell Sort

- The subarrays after each is sorted, and the array that contains them



# Shell Sort

- Algorithm that sorts array entries whose indices are separated by an increment of **`space`**.

```
Algorithm incrementalInsertionSort(a, first, last, space)
// Sorts equally spaced entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length;
// space is the difference between the indices of the entries to sort.

   for (unsorted = first + space through last at increments of space)
   {
      nextToInsert = a[unsorted]
      index = unsorted - space
      while ( (index >= first) and (nextToInsert.compareTo(a[index]) < 0) )
      {
         a[index + space] = a[index]
         index = index - space
      }
      a[index + space] = nextToInsert
   }
```

# Shell Sort

- Algorithm to perform a Shell sort will invoke **`incrementalInsertionSort`** and supply any sequence of spacing factors.

```
Algorithm shellSort(a, first, last)
// Sorts the entries of an array a[first..last] into ascending order.
// first >= 0 and < a.length; last >= first and < a.length.

   n = number of array entries
   space = n / 2
   while (space > 0)
   {
      for (begin = first through first + space - 1)
      {
         incrementalInsertionSort(a, begin, last, space)
      }
      space = space / 2
   }
```

# Efficiency of Shell Sort

- Efficiency highly depends on the spacing
  - Average case O(n $\sqrt{n}$ )
- Best case
  - O(n) when the number of space values is constant
  - O(n log n) when space is divided by 2 in each iteration

# Comparing the Algorithms

- The time efficiencies of three sorting algorithms, expressed in Big Oh notation

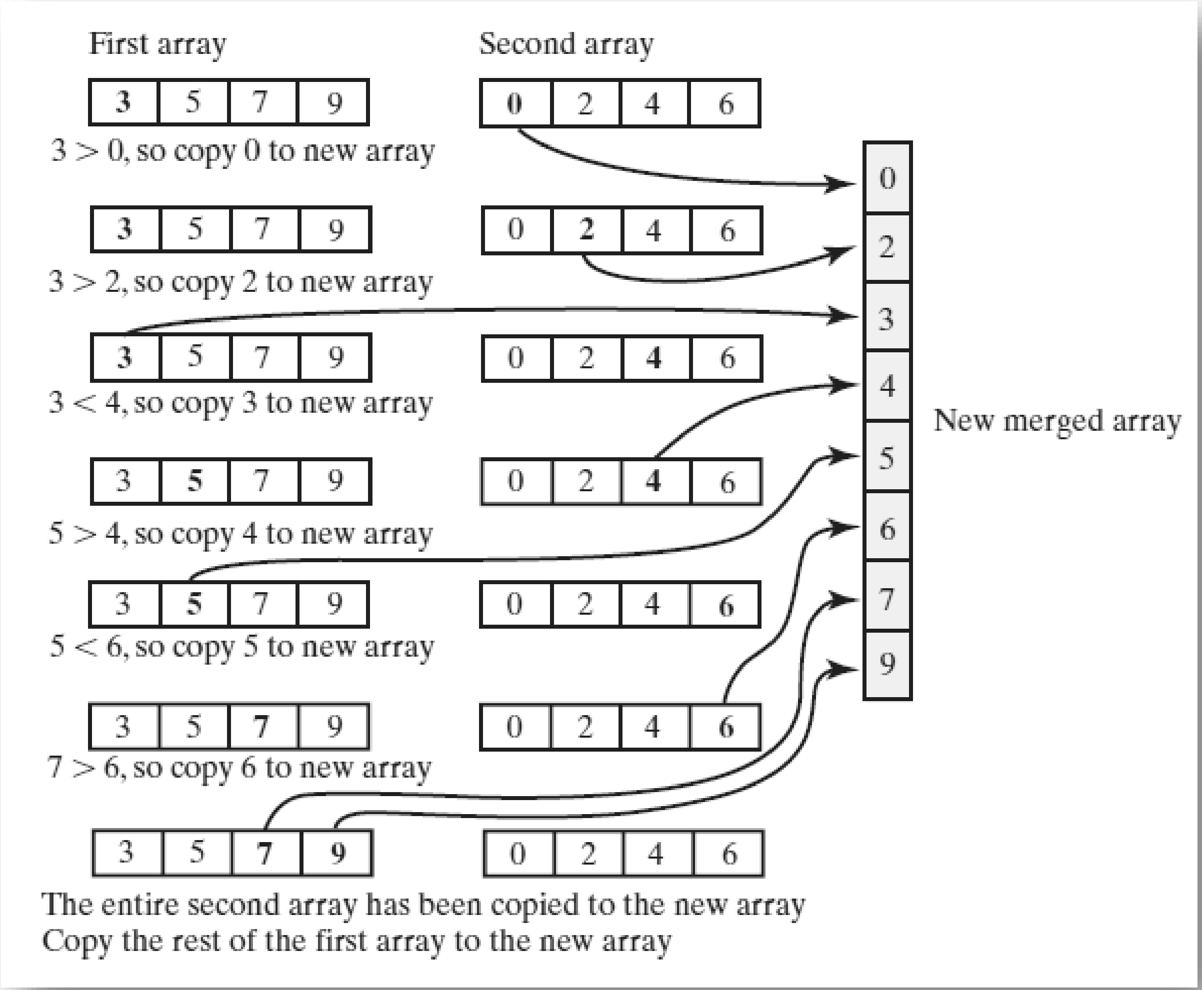
| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Shell sort | $O(n)$ | $O(n^{1.5})$ | $O(n^2)$ or $O(n^{1.5})$ |

# Merge Sort

- Divides an array into halves
- Sorts the two halves,
  - Then merges them into one sorted array.
- The algorithm for merge sort is usually stated recursively.
- Major programming effort is in the merge process

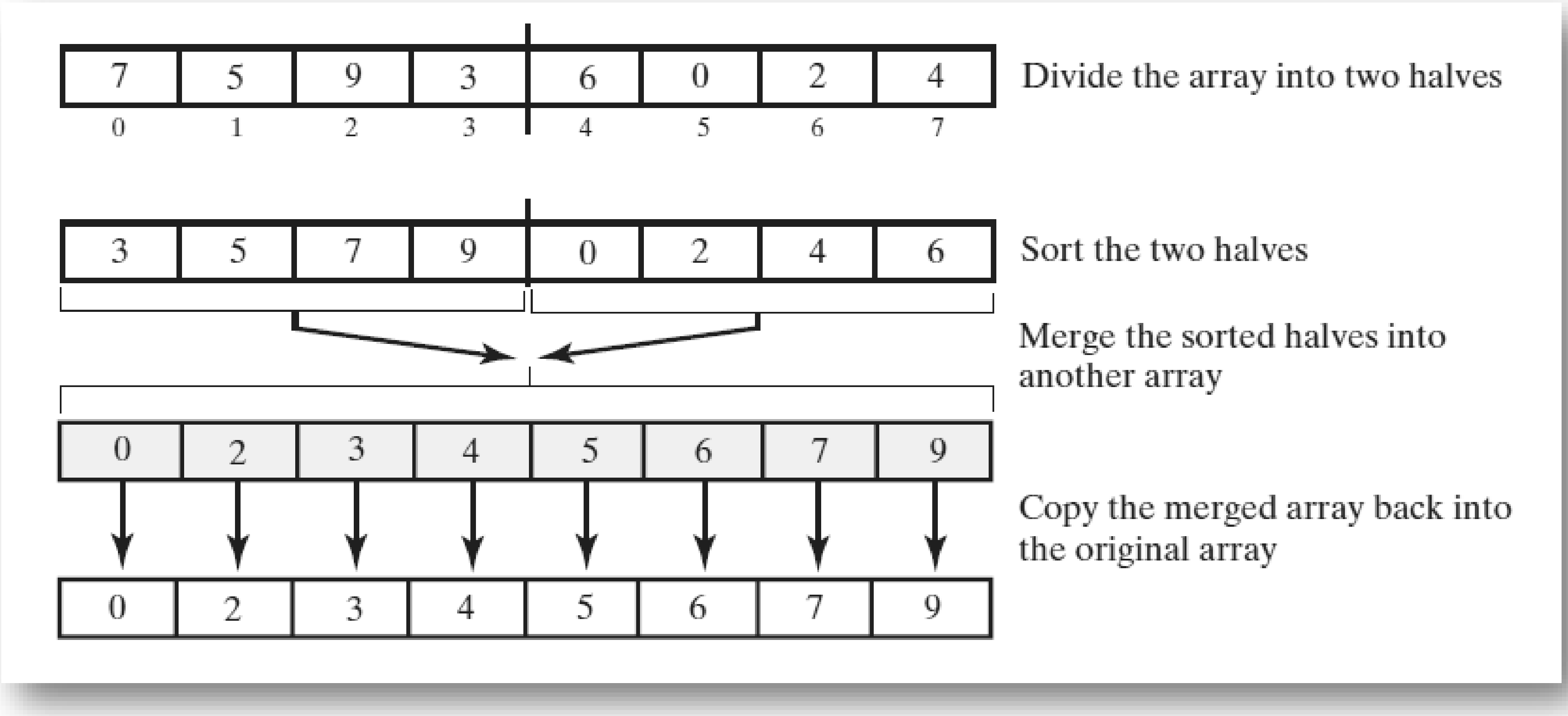
# Merging Arrays

- Merging two sorted arrays into one sorted array



# Recursive Merge Sort

- The major steps in a merge sort



# Recursive Merge Sort

- Recursive algorithm for merge sort.

```
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
   mid = approximate midpoint between first and last
   mergeSort(a, tempArray, first, mid)
   mergeSort(a, tempArray, mid + 1,last)
   Merge the sorted halves a[first..mid] and a[mid + 1..last] using the array tempArray
}
```

# Recursive Merge Sort

- Pseudocode which describes the merge step.

```
Algorithm merge(a, tempArray, first, mid, last)
// Merges the adjacent subarrays a[first..mid] and a[mid + 1..last].
beginHalf1 = first
endHalf1 = mid
beginHalf2 = mid + 1
endHalf2 = last
// While both subarrays are not empty, compare an entry in one subarray with
// an entry in the other; then copy the smaller item into the temporary array
index = 0 // Next available location in tempArray
while ( (beginHalf1 <= endHalf1) and (beginHalf2 <= endHalf2) )
{
   if (a[beginHalf1] <= a[beginHalf2])
   {
      tempArray[index] = a[beginHalf1]
      beginHalf1++
   }
```

# Recursive Merge Sort

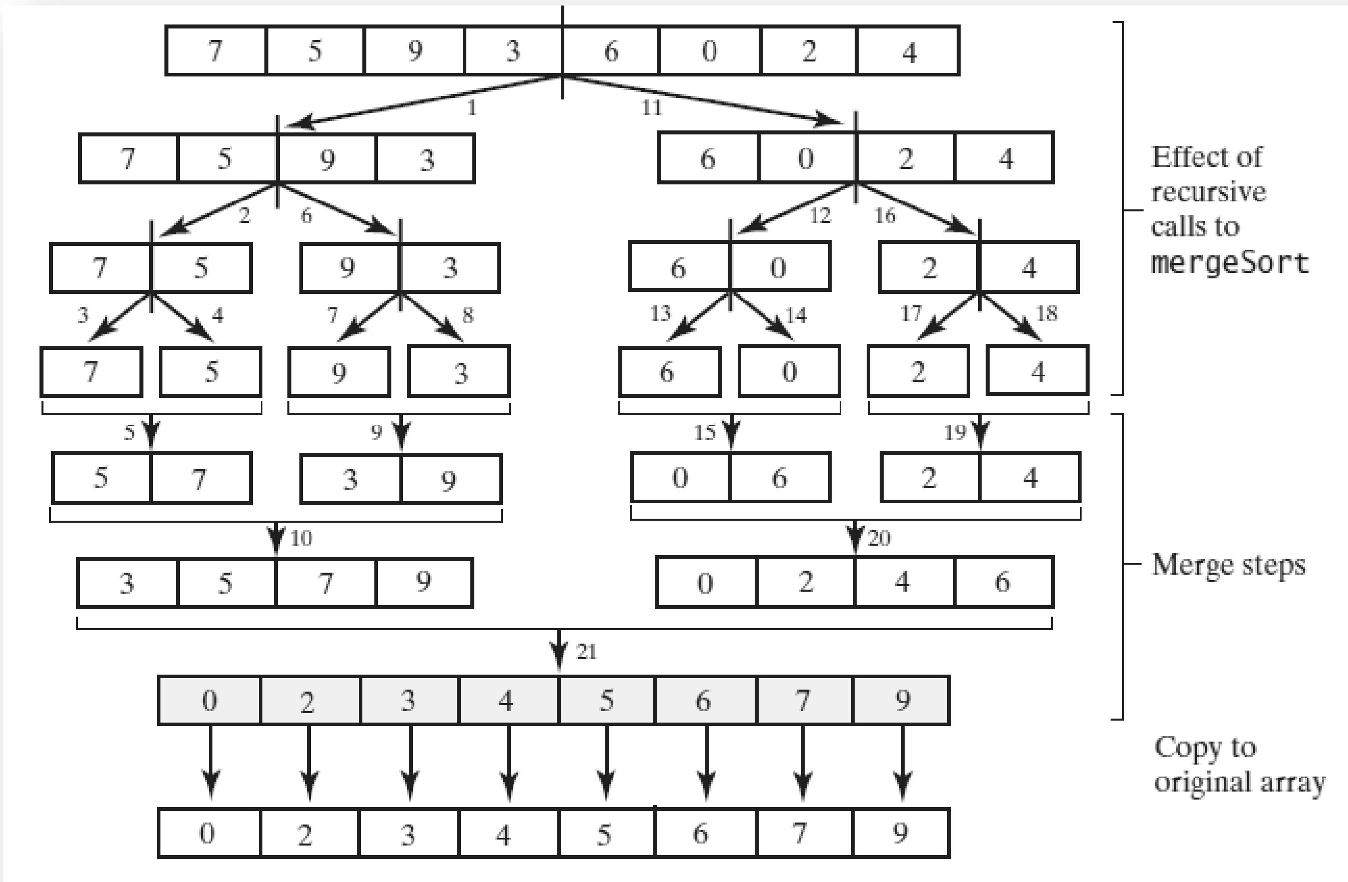
- Pseudocode which describes the merge step.

```
        tempArray[index] = a[beginHalf1]
        beginHalf1++
    }
    else
    {
        tempArray[index] = a[beginHalf2]
        beginHalf2++
    }
    index++
}
// Assertion: One subarray has been completely copied to tempArray.

Copy remaining entries from other subarray to tempArray
Copy entries from tempArray to array a
```

# Recursive Merge Sort

- FIGURE 9-3 The effect of the recursive calls and the merges during a merge sort



# Recursive Merge Sort

- Be careful to allocate the temporary array only once.

```
public static <T extends Comparable<? super T>>
       void mergeSort(T[] a, int first, int last)
{
   // The cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] tempArray = (T[])new Comparable<?>[a.length]; // Unchecked cast
   mergeSort(a, tempArray, first, last);
} // end mergeSort
```

# Merge Sort in the Java Class Library

- Class **Arrays** in the package **java.util** defines versions of a static method **sort**

```
public static void sort(Object[] a)

public static void sort(Object[] a, int first, int after)
```

# Merge Sort Properties

- stable
- not in-place
- can be made adaptive

# Quick Sort

- Divides an array into two pieces
  - Pieces are not necessarily halves of the array
  - Chooses one entry in the array—called the pivot
- Partitions the array into
  - <= pivot
  - >= pivot
  - places pivot in between the two parts
- Recursively sorts each part

# Quick Sort

- When pivot chosen, array rearranged such that:
  - Pivot is in position that it will occupy in final sorted array
  - Entries in positions before pivot are less than or equal to pivot
  - Entries in positions after pivot are greater than or equal to pivot

# Quick Sort

- Algorithm that describes our sorting strategy:

```
Algorithm quickSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.
if (first < last)
{
    Choose a pivot
    Partition the array about the pivot
    pivotIndex = index of pivot
    quickSort(a, first, pivotIndex - 1) // Sort Smaller
    quickSort(a, pivotIndex + 1, last)  // Sort Larger
}
```

# Quick Sort

- A partition of an array during a quick sort