Merge and Quick Sort

Merge Sort

- A *merge* is a common data processing operation:
  - Performed on *two sequences* of data
    - Items in both sequences use same **`compareTo`**
    - Both sequences in ordered of this **`compareTo`**
- **Goal:** Combine the two sorted sequences in one larger sorted sequence

- *Merge sort* merges longer and longer sequences

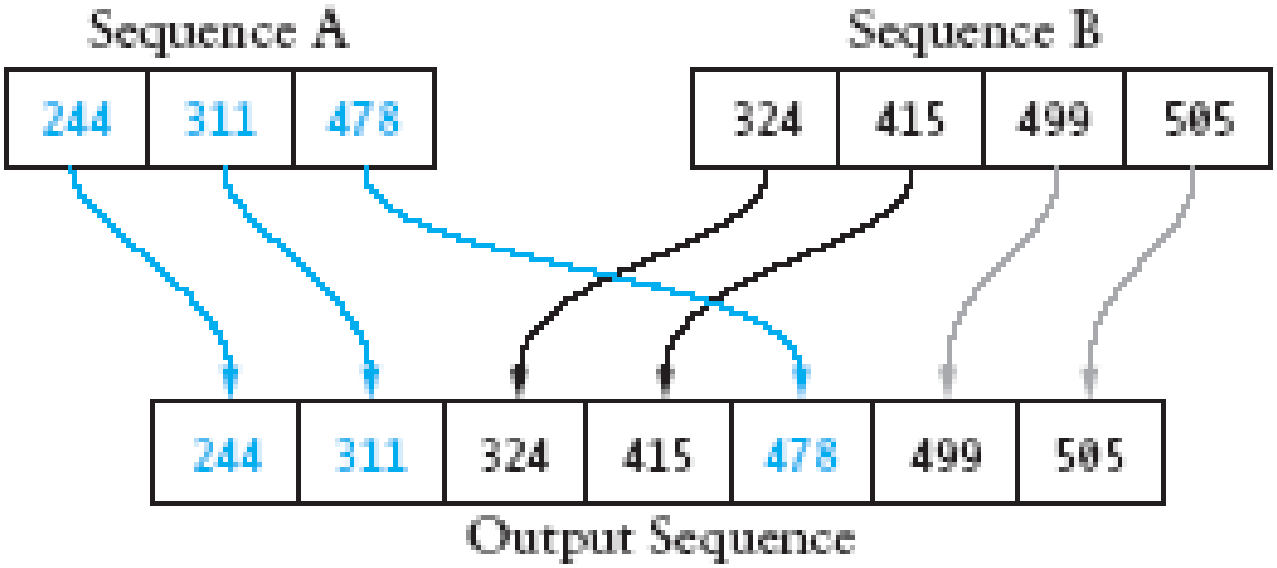
Merge Algorithm (Two Sequences)

Merging two sequences:

1. Access the first item from both sequences
2. While neither sequence is finished
    1. Compare the current items of both
    2. Copy smaller current item to the output
    3. Access next item from that input sequence
3. Copy any remaining from first sequence to output
4. Copy any remaining from second to output

Picture of Merge

FIGURE 10.6
Merge Operation



Analysis of Merge

- Two input sequences, total length n elements
  - Must move each element to the output
  - Merge time is O(n)

- Must store *both input* and *output* sequences
  - An array cannot be merged *in place*
  - Additional space needed: O(n)

Merge *Sort* Algorithm

Overview:

- Split array into two halves
- Sort the left half (recursively)
- Sort the right half (recursively)
- Merge the two sorted halves

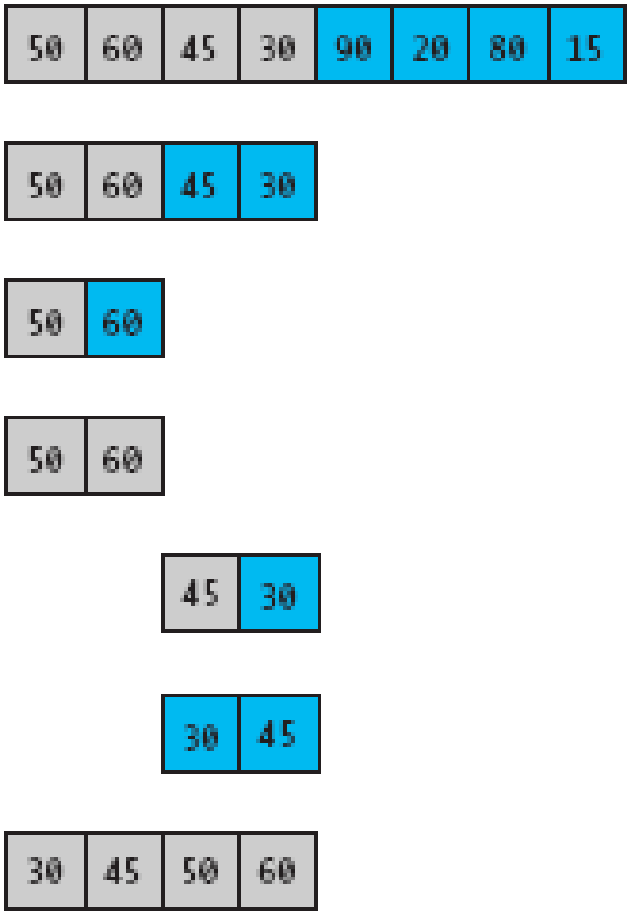
Merge *Sort* Algorithm (2)

Detailed algorithm:

- if **tSize** ≤ 1, return (no sorting required)
- set **hSize** to **tSize** / 2
- Allocate **LTab** of size **hSize**
- Allocate **RTab** of size **tSize** – **hSize**
- Copy elements 0 .. **hSize** – 1 to **LTab**
- Copy elements **hSize** .. **tSize** – 1 to **RTab**
- Sort **LTab** recursively
- Sort **RTab** recursively
- Merge **LTab** and **RTab** into **a**

Merge Sort Example

FIGURE 10.7
Trace of Merge Sort



1. *Split array into two 4-element arrays*
2. *Split left array into two 2-element arrays*
3. *Split left array (50, 60) into two 1-element arrays*
4. *Merge two 1-element arrays into a 2-element array*
5. *Split right array from Step 2 into two 2-element arrays*
6. *Merge two 1-element arrays into a 2-element array*
7. *Merge two 2-element arrays into a 4-element array*

Merge Sort Analysis

- Splitting/copying n elements to subarrays: O(n)
- Merging back into original array: O(n)
- Recursive calls: 2, each of size n/2
  - Their total non-recursive work: O(n)
- Next level: 4 calls, each of size n/4
  - Non-recursive work again O(n)
- Size sequence: n, n/2, n/4, ..., 1
  - Number of levels = log n
  - Total work: O(n log n)

Merge Sort Code

```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  if (a.length <= 1) return;
  int hSize = a.length / 2;
  T[] lTab = (T[])new Comparable[hSize];
  T[] rTab =
    (T[])new Comparable[a.length-hSize];
  System.arraycopy(a, 0, lTab, 0, hSize);
  System.arraycopy(a, hSize, rTab, 0,
                   a.length-hSize);
  sort(lTab); sort(rTab);
  merge(a, lTab, rTab);
}
```

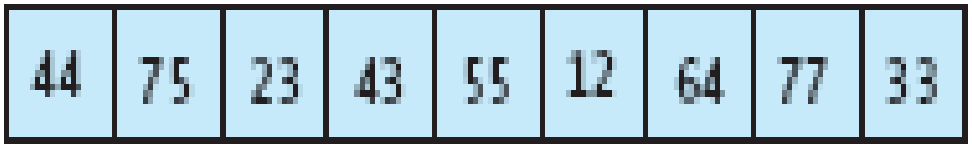
Merge Sort Code (2)

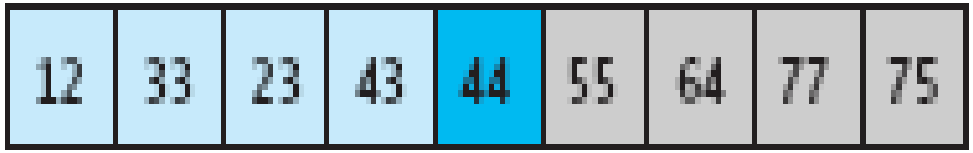
```
private static <T extends Comparable<T>>
    void merge (T[] a, T[] l, T[] r) {
  int i = 0;  // indexes l
  int j = 0;  // indexes r
  int k = 0;  // indexes a
  while (i < l.length && j < r.length)
    if (l[i].compareTo(r[j]) < 0)
      a[k++] = l[i++];
    else
      a[k++] = r[j++];
  while (i < l.length) a[k++] = l[i++];
  while (j < r.length) a[k++] = r[j++];
}
```

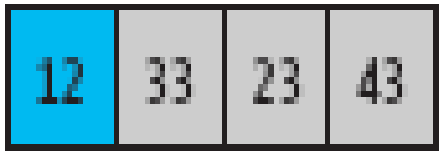
Quicksort

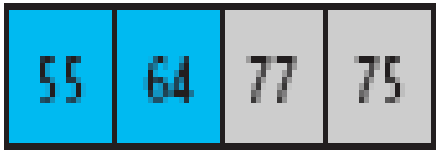
- Developed in 1962 by C. A. R. Hoare
- Given a *pivot value:*
  - Rearranges array into two parts:
    - Left part $\leq$ pivot value
    - Right part > pivot value
- Average case for Quicksort is O(n log n)
  - Worst case is $O(n^2)$

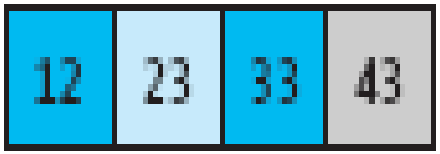
Quicksort Example

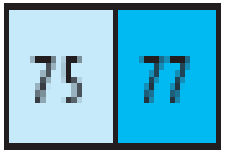












Algorithm for Quicksort

first and **last** are end points of region to sort

- if **first** < **last**
- Partition using **pivot**, which ends in **pivIndex**
- Apply Quicksort recursively to left subarray
- Apply Quicksort recursively to right subarray

Performance: O(n log n) provide **pivIndex** not always too close to the end

Performance $O(n^2)$ when **pivIndex** always near end

Quicksort Code

```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  qSort(a, 0, a.length-1);
}
private static <T extends Comparable<T>>
    void qSort (T[] a, int fst, int lst) {
  if (fst < lst) {
    int pivIndex = partition(a, fst, lst);
    qSort(a, fst, pivIndex-1);
    qSort(a, pivIndex+1, lst);
  }
}
```