Lecture 24: Faster Sorting

CS 0445: Data Structures

Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Nov 07, 2019, 8:00-9:15

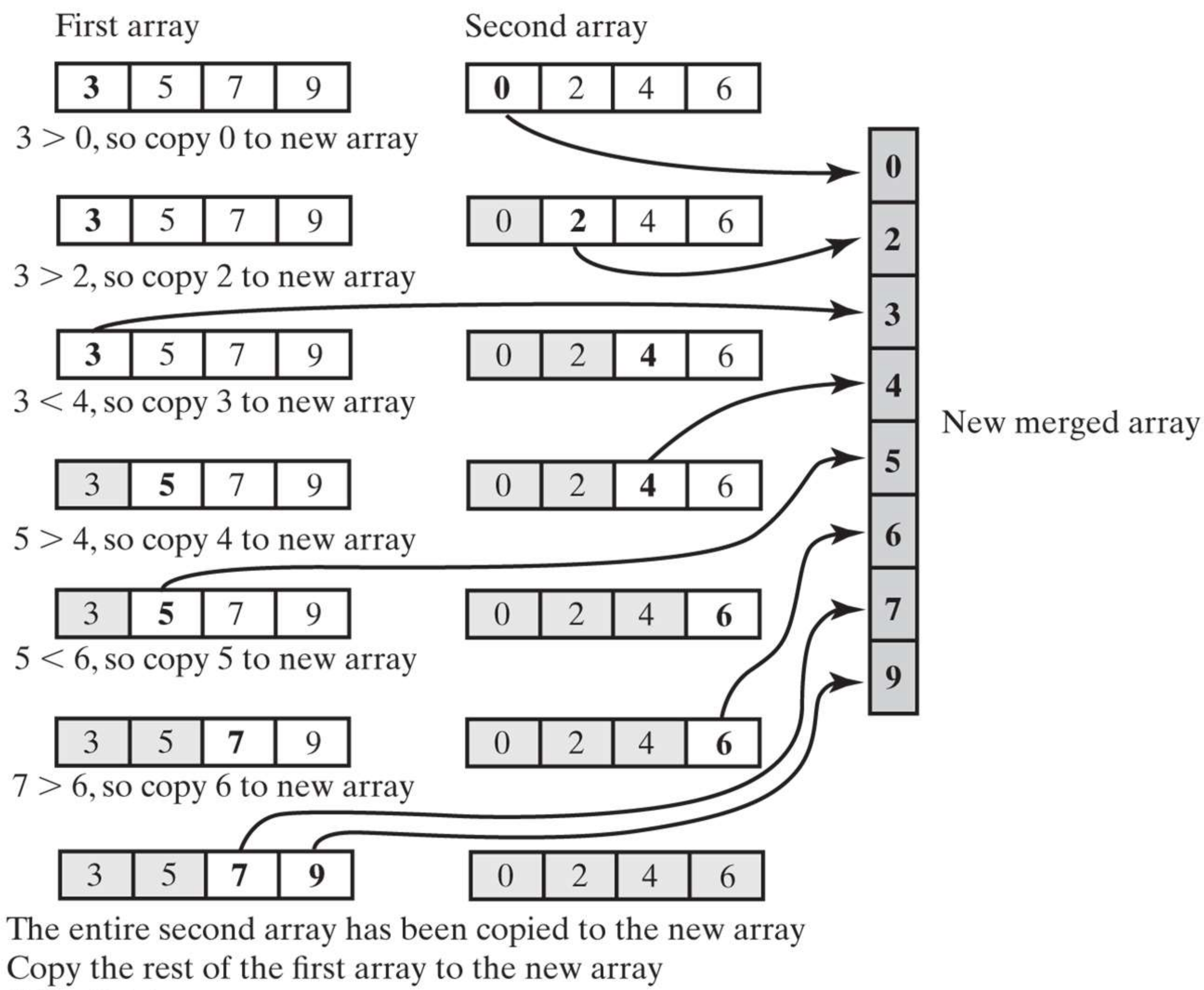
University of Pittsburgh, Pittsburgh, PA

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

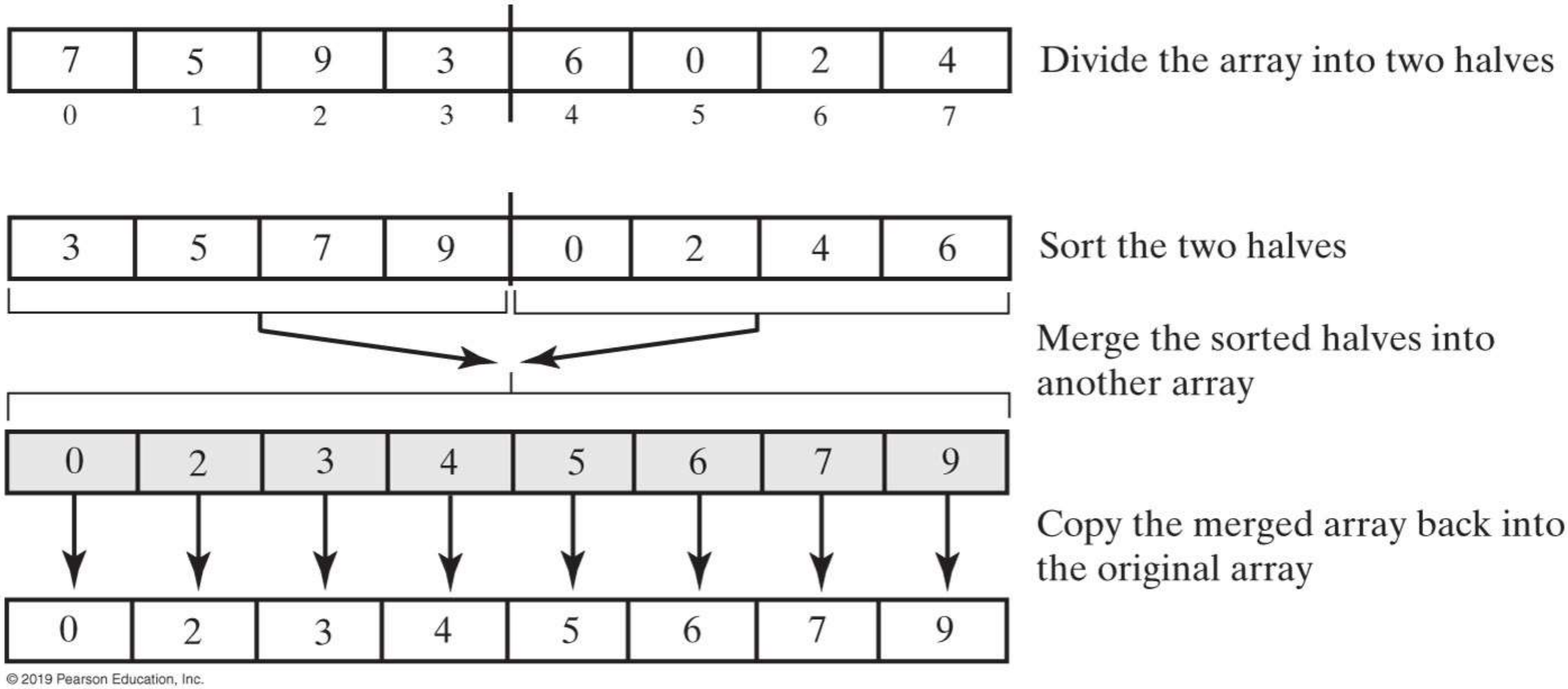
Merge Sort

- Merging two sorted arrays into one sorted array



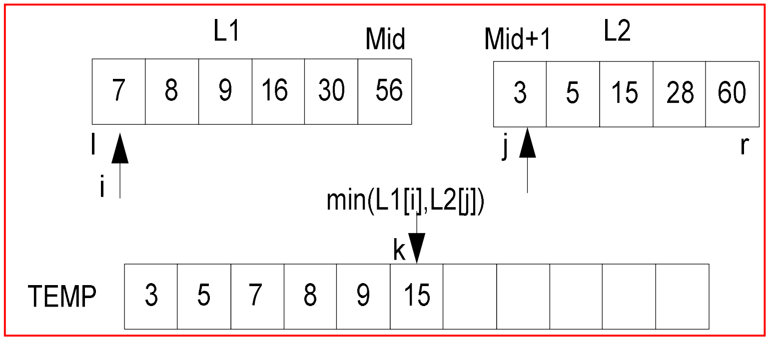
Merge Sort

- The major steps in a merge sort



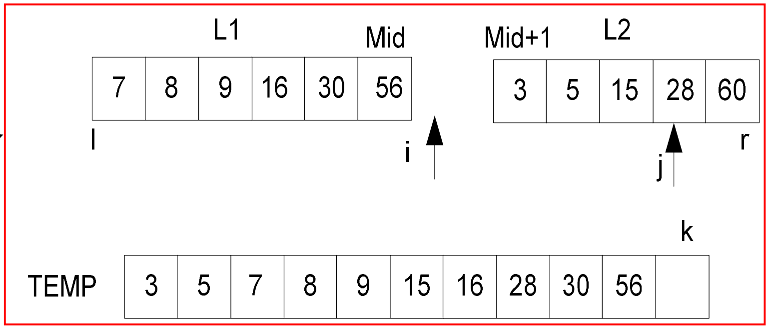
Merge Sort

```
void MergeSort(int A[], int temp[], int l, int r){

  // terminal condition of the recursion
  if (l==r) return;
  int mid = (l+r)/2;
  // minimizing overflow(for big l,r)
  // int mid = l + ((r - l) / 2);

  // split the array recursively
  Mergesort(A, temp, l, mid);
  Mergesort(A, temp, mid+1, r);

  // Now the arrays [l..mid] and [mid+1..r] are sorted
  // Merge procedure
  k=l, i=l; j=mid+1;

  // Merge in TEMP until one of the two lists is empty
  while ((i<=mid) && (j<=r)) {
        if (A[i]<A[j]){
                temp[k] = A[i]; i++; }
        else {
                temp[k] = A[j]; j++; }
        k++;
   }    continues in the next slide …
```



Merge Sort

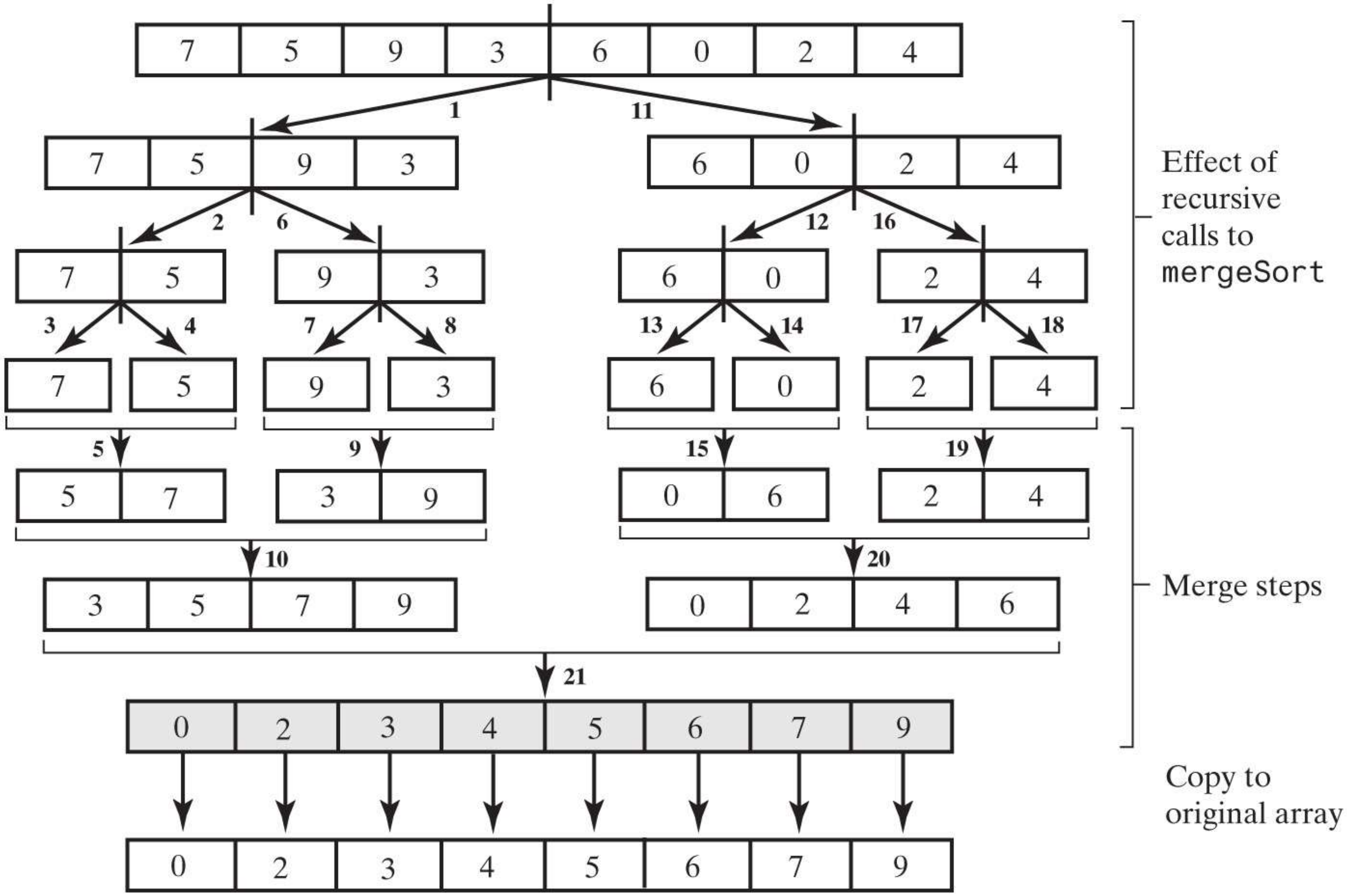
```
// copy all of the remaining elements in the list L1
while (i<=mid) {
     temp[k] = A[i];
     k++;i++;
}
```




```
// copy all of the remaining elements in the list L2
while (j<=r) {
     temp[k] = A[j];
     k++;j++;
}

// copy all elements from TEMP -> A
for (i=l; i<=r; i++) {
     A[i] = temp[i];
}
}
```

Merge Sort

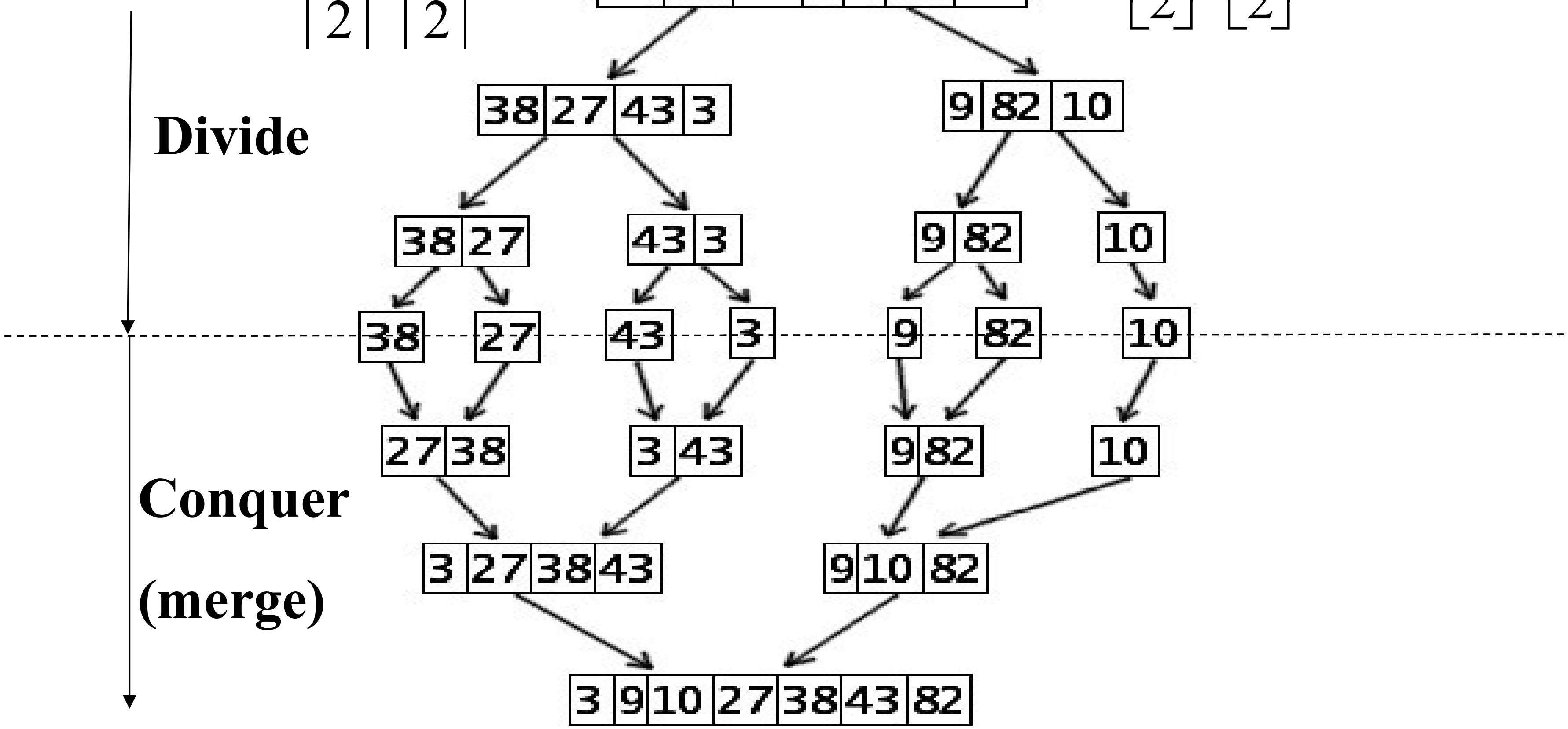
- The effect of the recursive calls and the merges during a merge sort



Merge Sort

n=7

$$\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{7}{2} \right\rceil = 4 \qquad \left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{7}{2} \right\rfloor = 3$$



Merge Sort Execution

BEFORE:[8,4,8,43,3,5,2,1,10,]

Index: 0 1 2 3 4 5 6 7 8

0,8: [8,4,8,43,3,|5,2,1,10,]

0,4: [8,4,8,|43,3,]

0,2: [8,4,|8,]

0,1: [8,|4,]

0,0: [8,]

1,1: [4,]

divide

Merging: [A0,A0] [A1,A1] => [4,8,]

2,2: [8,]

divide

Merging: [A0,A1] [A2,A2] => [4,8,8,]

3,4: [43,3,]

3,3: [43,]

4,4: [3,]

divide

Merging: [A3,A3] [A4,A4] => [3,43,]

Merging: [A0,A2] [A3,A4] => [3,4,8,8,43,]

5,8: [5,2,|1,10,]

5,6: [5,|2,]

5,5: [5,]

6,6: [2,]

divide

Merging: [A5,A5] [A6,A6] => [2,5,]

7,8: [1,|10,]

7,7: [1,]

8,8: [10,]

divide

Merging: [A7,A7] [A8,A8] => [1,10,]

Merging: [A5,A6] [A7,A8] => [1,2,5,10,]

Merging: [A0,A4] [A5,A8] => [1,2,3,4,5,8,8,10,43,]

AFTER:[1,2,3,4,5,8,8,10,43,]

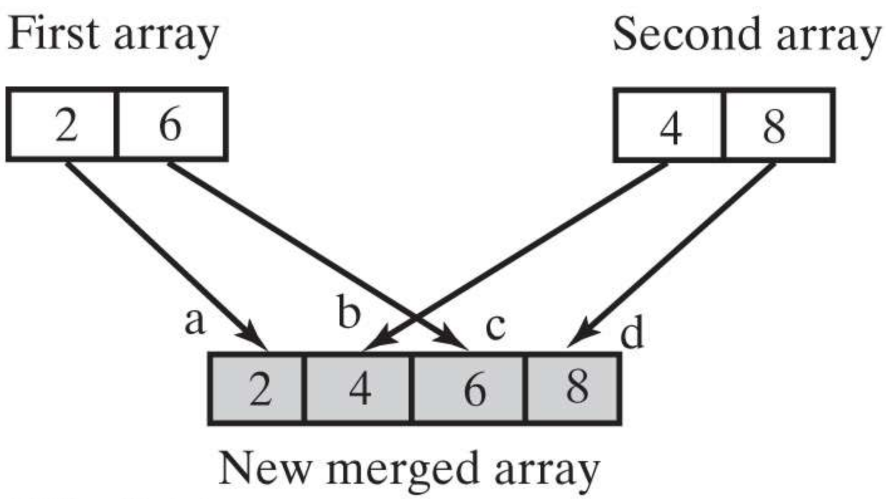
Recursive Merge Sort

- 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

- worst-case merge of two sorted arrays



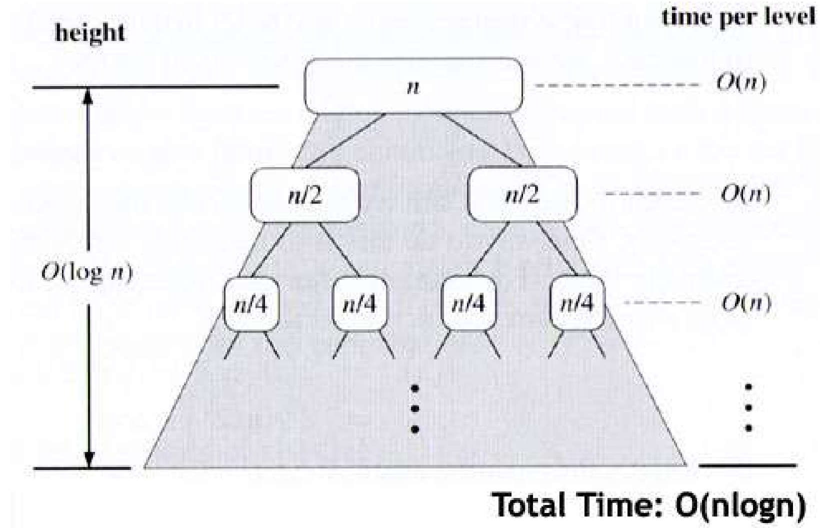
a. 2 < 4, so copy 2 to new array

b. 6 > 4, so copy 4 to new array

c. 6 < 8, so copy 6 to new array

d. Copy 8 to new array

Efficiency is $O(n \log n)$



Iterative Merge Sort

- Less simple than recursive version.
  - Need to control the merges.
- Will be more efficient of both time and space.
  - But, trickier to code without error.

Iterative Merge Sort

- Starts at beginning of array
  - Merges pairs of individual entries to form two-entry subarrays
- Returns to the beginning of array and merges pairs of the two-entry subarrays to form four-entry subarrays
  - And so on
- After merging all pairs of subarrays of a particular length, might have entries left over.

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)
```

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

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..last]** *recursively.*

if (first < last)

{

Choose a pivot

Partition the array about the pivot

pivotIndex = *index of pivot*

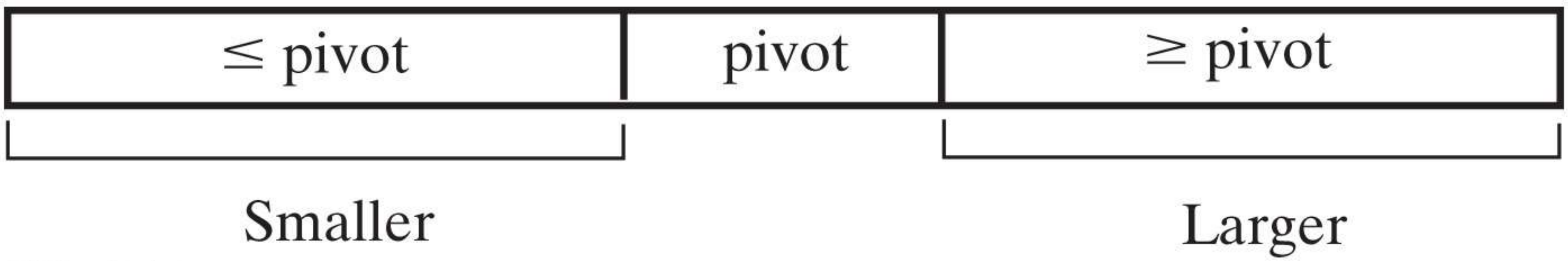
quickSort(a, first, pivotIndex − 1) *// Sort Smaller*

quickSort(a, pivotIndex + 1, last) *//Sort Larger*

}

Quick Sort

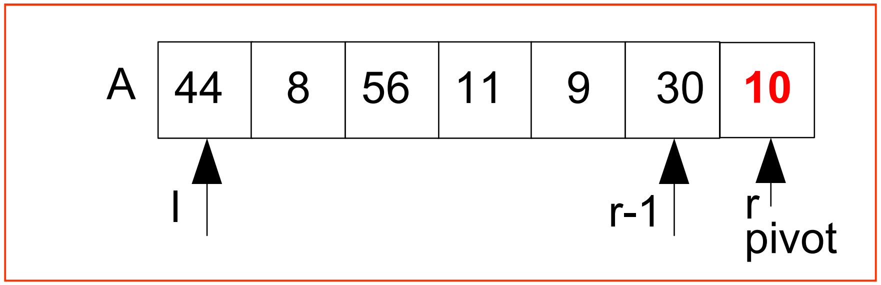
- A partition of an array during a quick sort



Pseudocode QuickSort

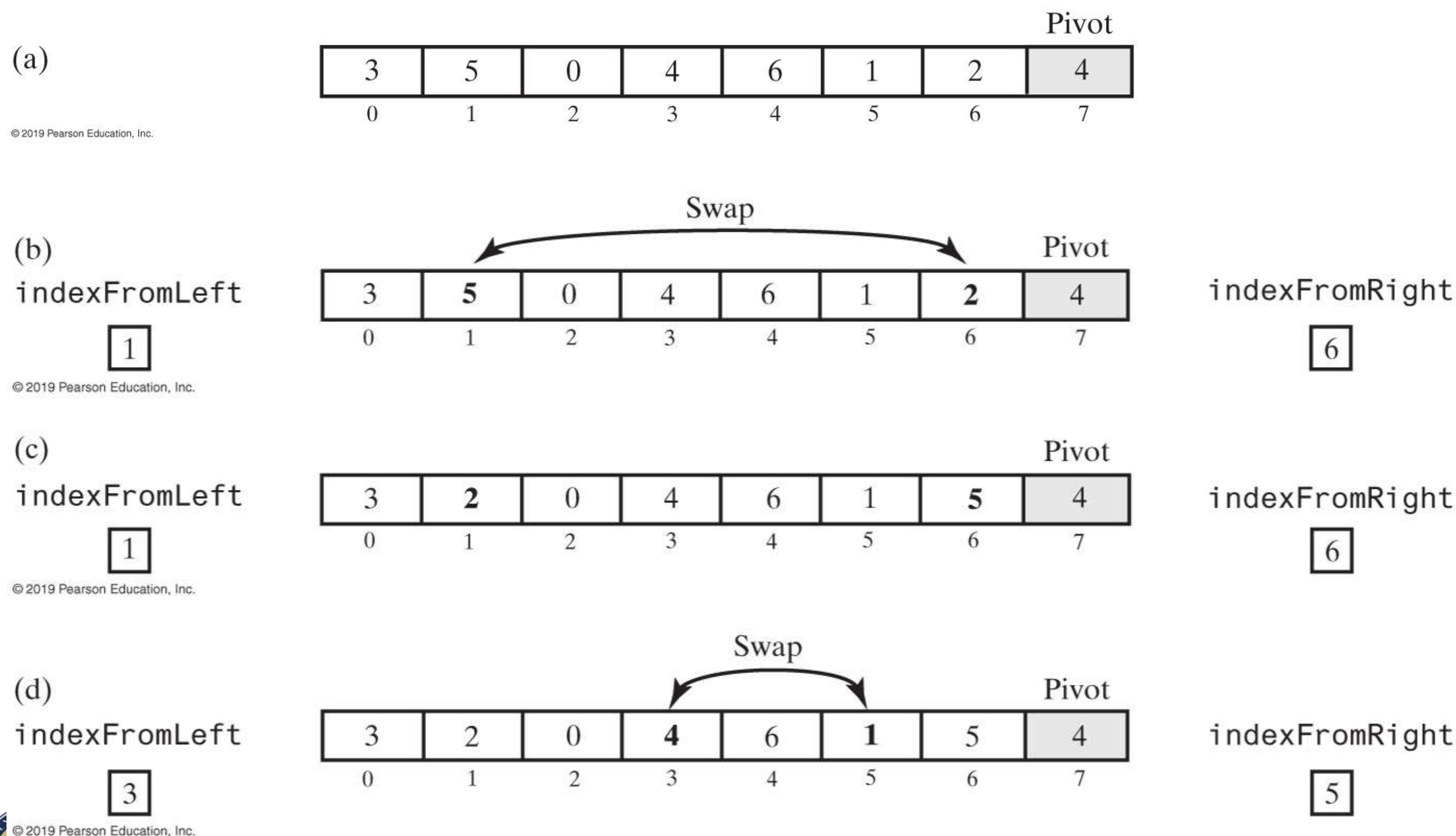
```
void Quicksort(int A[], int l, int r){
   if (l>=r) return;

   int pivotIndex = (l+r)/2;
   int pivot = A[pivotIndex];

   // swap the pivot with the last one.
   swap(A, pivotIndex, r);

   /* The partition procedure divides table A [l… r-1] so that
   A [l..k-1] contains elements < pivot, A [k… r-1] contains
   elements>= pivot, and returns the value k.*/
   int k = partition (A, l, r-1, pivot);

   // swap k with the last one.
   swap(A, k, r);

   Quicksort(A, l, k-1);
   QuickSort(A, k+1, r);
}
```

Here we chose the middle one. We could have chosen anyone else



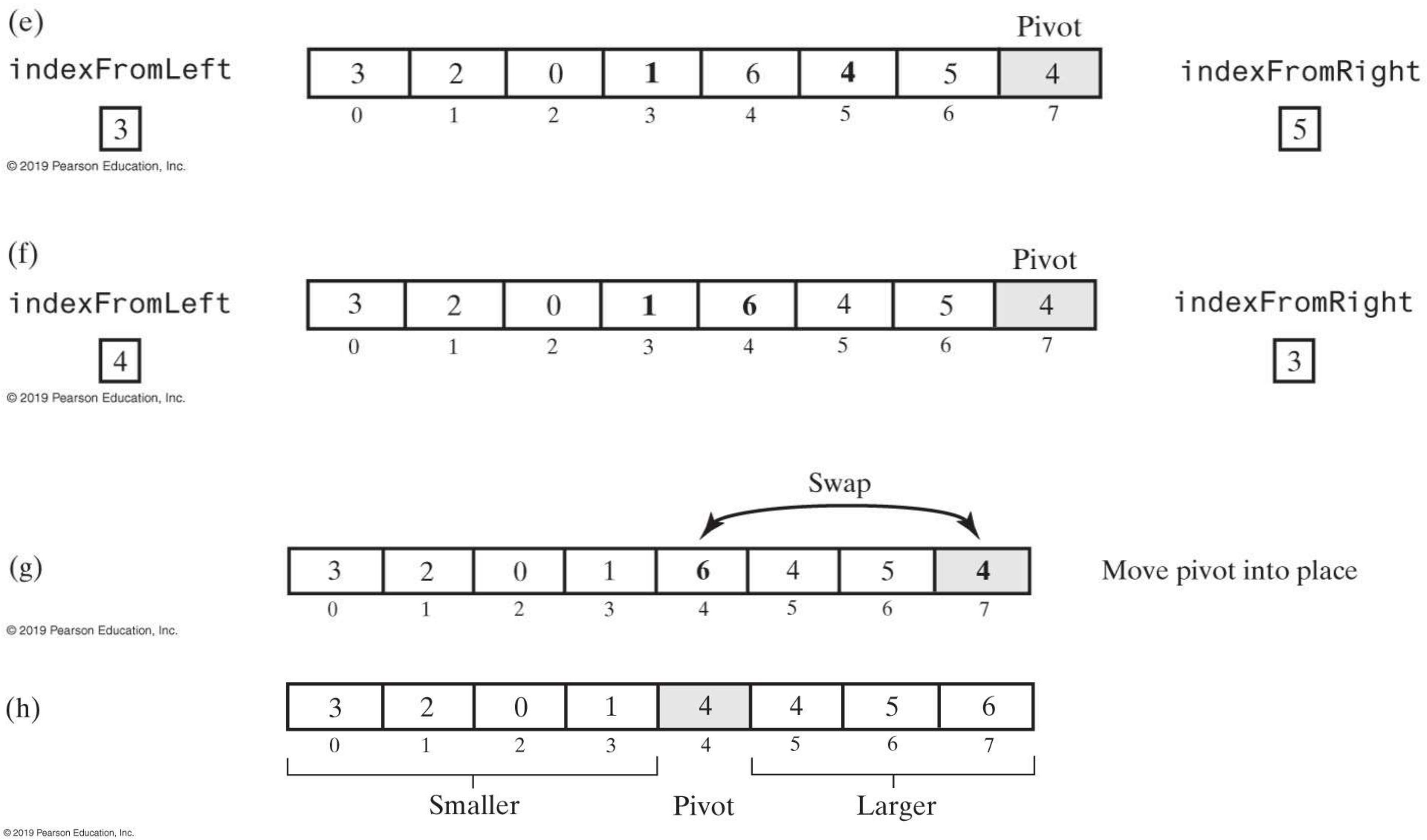
Quick Sort Partitioning (Part 1)

- A partitioning strategy for quick sort



Quick Sort Partitioning (Part 2)

- A partitioning strategy for quick sort



Partition Execution Example

Input:

index	0	1	2	3	4	5	6	7
	72	*6*	*37*	*48*	*30*	*42*	*83*	*75*

pivot = 48, move pivot at the end (swap(4, 8)):

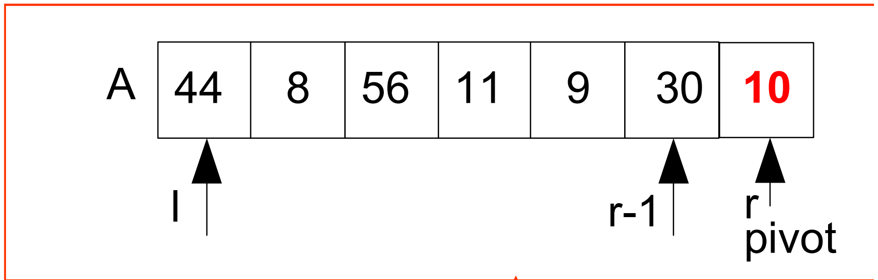
72	6	37	75	30	42	83	48	pivot
l						**r**		

Execute Partition(A, l, r, 48):

72	6	37	75	30	42	83	48
l					**r**		
42	6	37	75	30	72	83	48
			l	**r**			
42	6	37	30	75	72	83	48
			l	**r**			
42	6	37	30	75	72	83	48
			r	**l**			

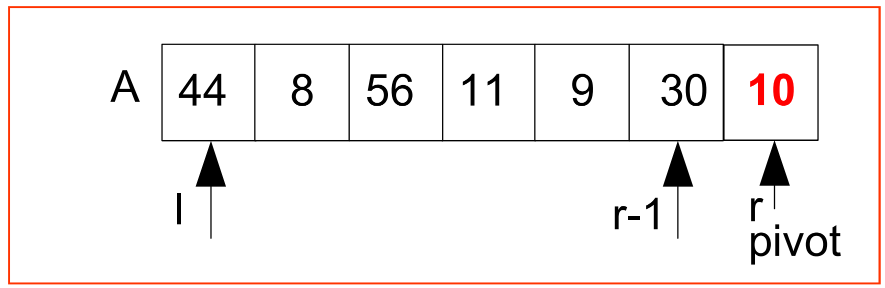
Quicksort

```
void quicksort(int A[], int l, int r) {
   int pivot, pivotIndex;

   if (l>=r) return;

   // Selecting the pivot
   pivotIndex = (l+r)/2;
   pivot = A[pivotIndex];
   // Move the pivot to the end of the list
   swap(A, pivotIndex, r);

   /* Calling the partition function (which places the smallest and
   largest elements left, right, respectively). */
   pivotIndex = partition(A, l, r-1, pivot);

   // Placing the pivot back to its original position
   if (A[r] < A[pivotIndex])
        swap(A, pivotIndex, r);

   quicksort(A, l, pivotIndex-1);
   quicksort(A, pivotIndex+1, r);
}
```



Partition

```
int partition(int A[], int l, int r, int pivot) {

  while(l<r) {
     // // move from l to r until we need a swap
     while (A[l]<pivot && l<r) // leave "<pivot" on left
          l++;
     // // move from r to l until we need a swap
     while (pivot<=A[r] && l<r) //leave ">=pivot" on right
          r--;

     if (l == r) break;

     // swap
     if (A[l]>=pivot)
         swap(A, l, r);  // move ">=" to the right
  }

  // return the point where we want the pivot to be inserted
       return l;
}
```



The Quick Sort Method

- Implementation of the quick sort.

```
public static <T extends Comparable<? super T>>
     void quickSort(T[] a, int first, int last)
{
  if (last - first + 1 < MIN_SIZE)
  {
    insertionSort(a, first, last);
  }
  else
  {
    // Create the partition: Smaller | Pivot | Larger
    int pivotIndex = partition(a, first, last);

    // Sort subarrays Smaller and Larger
    quickSort(a, first, pivotIndex - 1);
    quickSort(a, pivotIndex + 1, last);
  } // end if
} // end quickSort
```

QuickSort Execution Example

```
BEFORE:[72,6,37,48,30,42,83,75]

        Index: 0   1   2   3   4   5   6   7
** QuickSort [0,7]
[72,6,37,48,30,42,83,75,]
PivotIndex: 3(48)  =>   Swapping 48, 75
[72,6,37,75,30,42,83,48,]
Partitioning [0,6]
    Swapping 72, 42
[42,6,37,75,30,72,83,48,]
    Swapping 75, 30
[42,6,37,30,75,72,83,48,]
Inserting Pivot at Position:4
Swapping 75, 48
[42,6,37,30,48,72,83,75,]

** QuickSort [0,3]
[42,6,37,30,48,72,83,75,]
PivotIndex: 1(6)  =>    Swapping 6, 30
[42,30,37,6,48,72,83,75,]
Partitioning [0,2]
Inserting Pivot at Position:0
Swapping 42, 6

** QuickSort [0,-1] -> RETURN
```

```
** QuickSort [1,3]
[6,30,37,42,48,72,83,75,]
PivotIndex: 2(37)   =>   Swapping 37, 42
[6,30,42,37,48,72,83,75,]
Partitioning [1,2]
Inserting Pivot at Position:2
Swapping 42, 37

** QuickSort [1,1] -> RETURN
** QuickSort [3,3] -> RETURN

** QuickSort [5,7]
[6,30,37,42,48,72,83,75,]
PivotIndex: 6(83)   =>   Swapping 83, 75
[6,30,37,42,48,72,75,83,]
Partitioning [5,6] with pivot:83
Inserting Pivot at Position:6

** QuickSort [5,5] -> RETURN
** QuickSort [7,7] -> RETURN

AFTER:[6,30,37,42,48,72,75,83,]
```

Quick Sort in the Java Class Library

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

public static void sort(*type*[] a)

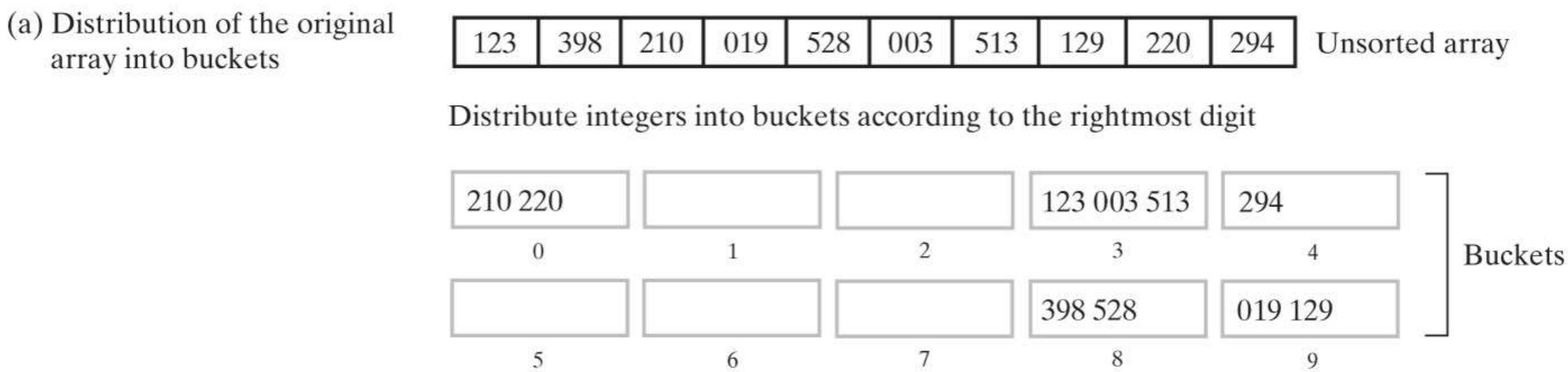
public static void sort(*type*[] a, int first, int after)

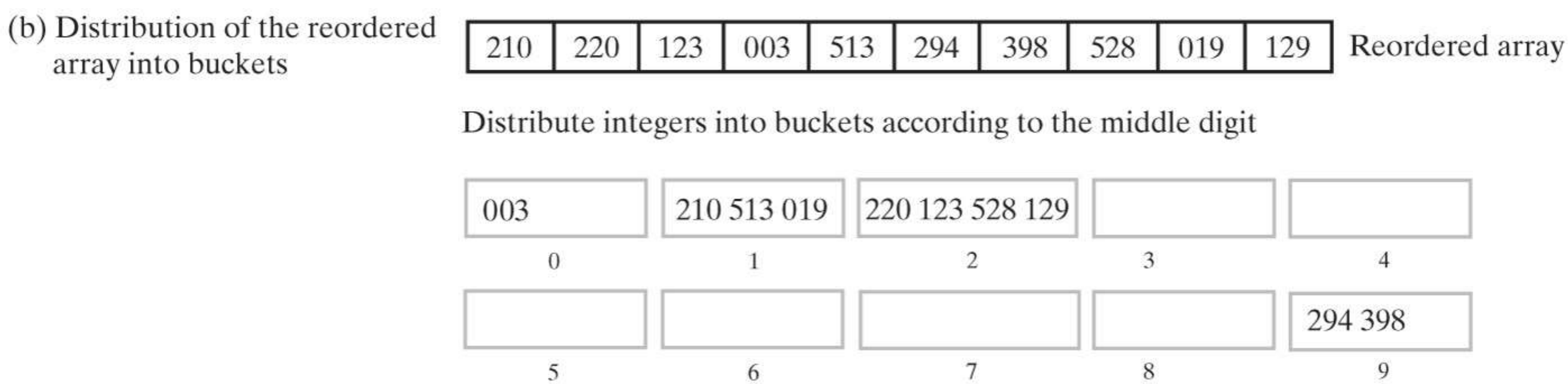
Radix Sort

- Does not use comparison
- Treats array entries as if they were strings that have the same length.
  - Group integers according to their rightmost character (digit) into “buckets”
  - Repeat with next character (digit), etc.

Radix Sort (Part 1)

- The steps of a radix sort





Radix Sort (Part 2)

- The steps of a radix sort

(c) Distribution of the reordered array into buckets

003	210	513	019	220	123	528	129	294	398

Reordered array

Distribute integers into buckets according to the leftmost digit

003 019	123 129	210 220 294	398	
0	1	2	3	4
513 528				
5	6	7	8	9

(d) Sorting is complete

003	019	123	129	210	220	294	398	513	528

Sorted array

Algorithm Comparison

- The time efficiency of various sorting algorithms, expressed in Big Oh notation

	Best Case	Average Case	Worst Case
Radix Sort	$O(n)$	$O(n)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Shell Sort	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Comparing Function Growth Rates

- A comparison of growth-rate functions as n increases

		10^2	10^3	10^4	10^5	10^6
n	10	100	1,000	10,000	100,000	1,000,000
$n \log n$	33	664	9,966	132,877	1,660,964	19,931,569
$n^{1.5}$	32	1,000	31,623	1,000,000	319,622,777	109
n^2	100	10,000	1,000,000	108	1,010	1,012