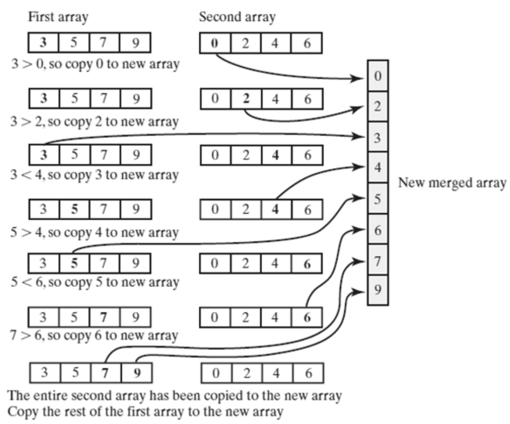
CSC 273 – Data Structures

Lecture 6 - Faster Sorting Methods

Merge Sort

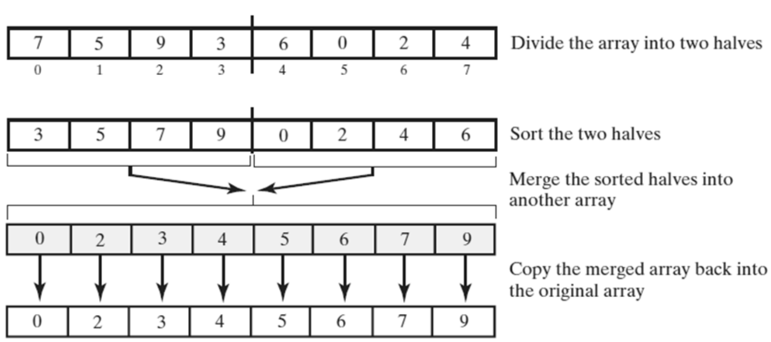
- Divides an array into halves
- Sorts the two halves,
  - Then merges them into one sorted array.
- The algorithm for merge sort is usually stated recursively.
- Major programming effort is in the merge process

Merging Arrays



Merging two sorted arrays into one sorted array

Recursive Merge Sort



The major steps in a merge sort

Recursive Merge Sort

```
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 algorithm for merge sort

Recursive Merge Sort

```
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++
   }
```

Pseudocode which describes the merge step

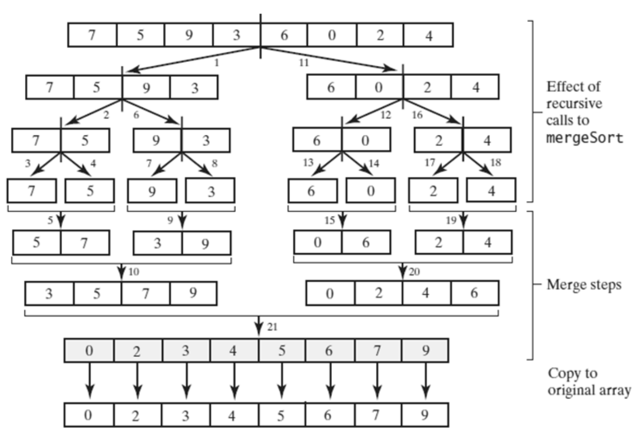
Recursive Merge Sort

```
      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
```

Pseudocode which describes the merge step

Recursive Merge Sort

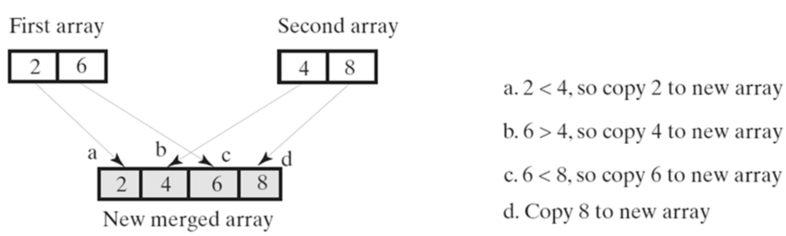


The effect of the recursive calls and the merges during a merge sort

Recursive Merge Sort

```
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
```

Efficiency of Merge Sort



A worst-case merge of two sorted arrays - 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

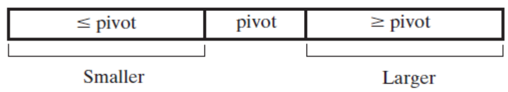
- When pivot chosen, array rearranged such that:
  - Pivot is in position that it will occupy in final sorted array
  - Entries in positions before pivot are less than or equal to pivot
  - Entries in positions after pivot are greater than or equal to pivot

Quick Sort

Algorithm that describes our sorting strategy:

```
Algorithm quickSort(a, first, last)
// Sorts the array entries a[first] 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
}
```

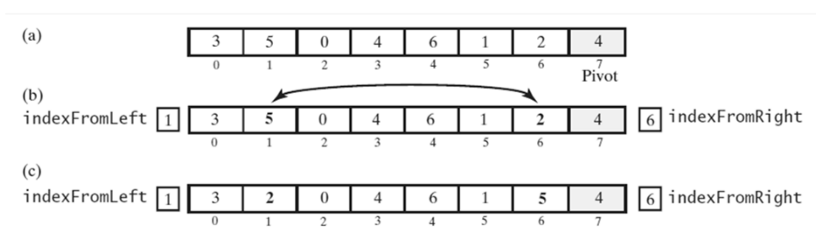
Quick Sort



- Quick sort is $O(n \log n)$ in average case, $O(n^2)$ in worst case.
- Choice of pivots affects behavior

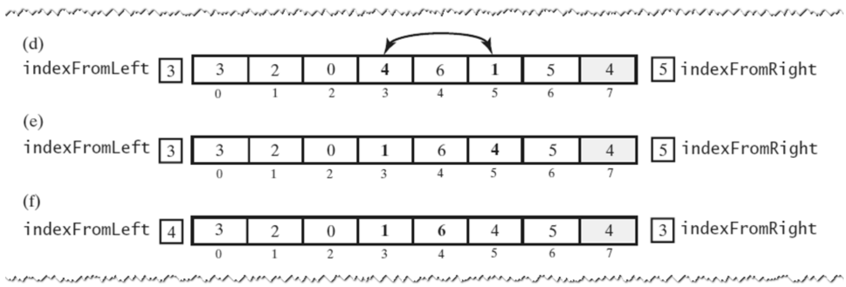
Creating the Partition

A partitioning strategy for quick sort



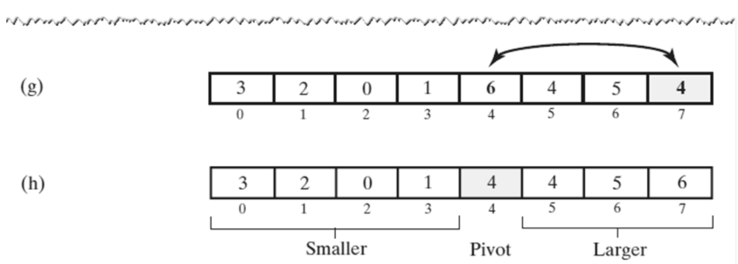
Creating the Partition

A partitioning strategy for quick sort

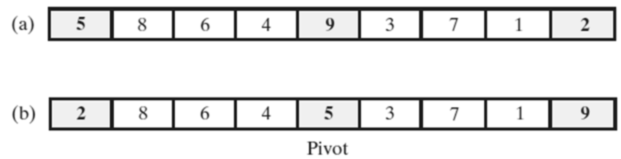


Creating the Partition

A partitioning strategy for quick sort

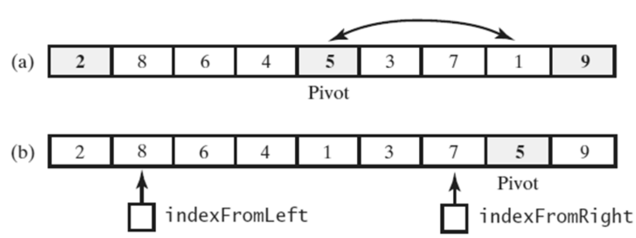


Creating the Partition



Median-of-three pivot selection:
(a) The original array
(b) The array with its first, middle, and last entries sorted

Adjusting the Partition Algorithm



a) The array with its first, middle, and last entries sorted
b) The array after positioning the pivot and just before partitioning

Adjusting the Partition Algorithm

```
Algorithm partition(a, first, last)
// Partitions an array a[first..last] as part of quick sort into two subarrays named
// Smaller and Larger that are separated by a single entry—the pivot— named pivotValue.
// Entries in Smaller are <= pivotValue and appear before pivotValue in the array.
// Entries in Larger are >= pivotValue and appear after pivotValue in the array.
// first >= 0; first < a.length; last - first >= 3; last < a.length.
// Returns the index of the pivot.
    mid = index of the array's middle entry
    sortFirstMiddleLast(a, first, mid, last)
    // Assertion: a[mid] is the pivot, that is, pivotValue;
    // a[first] <= pivotValue and a[last] >= pivotValue, so do not compare these two
    // array entries with pivotValue.

    // Move pivotValue to next-to-last position in array
```

Adjusting the Partition Algorithm

```
    // Move pivotValue to next-to-last position in array
    Exchange a[mid] and a[last - 1]
    pivotIndex = last - 1
    pivotValue = a[pivotIndex]

    // Determine two subarrays:
    //    Smaller = a[first..endSmaller] and
    //    Larger  = a[endSmaller+1..last-1]
    // such that entries in Smaller are <= pivotValue and
    // entries in Larger are >= pivotValue.
    // Initially, these subarrays are empty.
    indexFromLeft = first + 1
    indexFromRight = last - 2
    done = false
    while (!done)
```

Adjusting the Partition Algorithm

```
while (!done)
{
   // Starting at the beginning of the array, leave entries that are < pivotValue and
   // locate the first entry that is >= pivotValue. You will find one, since the last
   // entry is >= pivotValue.
   while (a[indexFromLeft] < pivotValue)
      indexFromLeft++

   // Starting at the end of the array, leave entries that are > pivotValue and
   // locate the first entry that is <= pivotValue. You will find one, since the first
   // entry is <= pivotValue.
   while (a[indexFromRight] > pivotValue)
      indexFromRight--

   // Assertion: a[indexFromLeft] >= pivotValue and
   //        a[indexFromRight] <= pivotValue
   if (indexFromLeft < indexFromRight)
```

Adjusting the Partition Algorithm

```
   //        a[indexFromRight] <= pivotValue
   if (indexFromLeft < indexFromRight)
   {
      Exchange a[indexFromLeft] and a[indexFromRight]
      indexFromLeft++
      indexFromRight--

   }
   else
      done = true
}

Exchange a[pivotIndex] and a[indexFromLeft]
pivotIndex = indexFromLeft

// Assertion: Smaller = a[first..pivotIndex-1]
//           pivotValue = a[pivotIndex]
//           Larger = a[pivotIndex+1..last]
return pivotIndex
```

The Quick Sort Method

```
/** Sorts an array into ascending order. Uses quick sort with
    median-of-three pivot selection for arrays of at least
    MIN_SIZE entries, and uses insertion sort for smaller arrays. */
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
```

Quick Sort in the Java Class Library

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

ts an entire array a, while the method

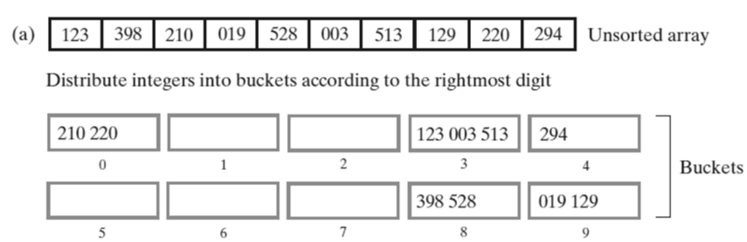
```
public static void sort(type[] a, int first, int after)
```

Class **`Arrays`** in the package **`java.util`** uses a quick sort to sort arrays of primitive types into ascending order

Radix Sort

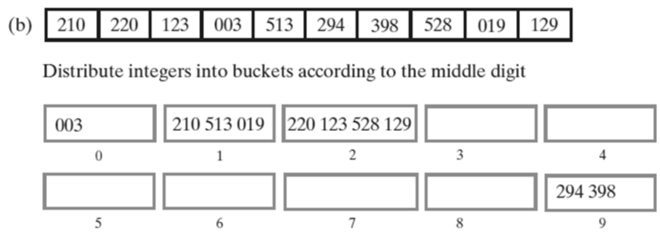
- Does not use comparison
- Treats array entries as if they were strings that have the same length.
  - Group integers according to their rightmost character (digit) into "buckets"
  - Repeat with next character (digit), etc.

Radix Sort



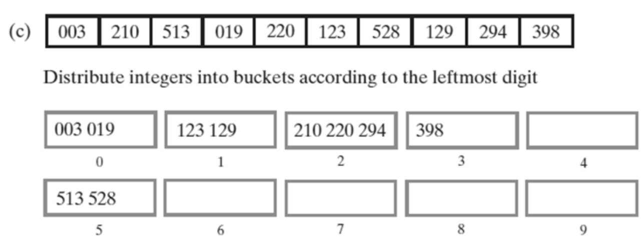
*Original array and buckets after first distribut*ion

Radix Sort

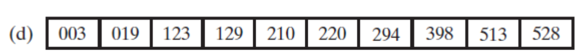


Reordered array and buckets after second distribution

Radix Sort



Reordered array and buckets after third distribution



Sorted array

Pseudocode for Radix Sort

```
Algorithm radixSort(a, first, last, maxDigits)
// Sorts the array of positive decimal integers a[first..last] into ascending order;
// maxDigits is the number of digits in the longest integer.

for (i = 0 to maxDigits - 1)
{
   Clear bucket[0], bucket[1], . . . , bucket[9]
   for (index = first to last)
   {
      digit = digit i of a[index]
      Place a[index] at end of bucket[digit]
   }
   Place contents of bucket[0], bucket[1], . . . , bucket[9] into the array a
}
```

Radix sort is an O(n) algorithm for certain data,
it is not appropriate for all data

Comparing the Algorithms

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

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

Comparing the Algorithms

n	10	10^2	10^3	10^4	10^5	10^6
$n \log_2 n$	33	664	9966	132,877	1,660,964	19,931,569
$n^{1.5}$	32	10^3	31,623	10^6	31,622,777	10^9
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}

A comparison of growth-rate functions as n *increases*