Sorting

Based on Chapter 10 of
Koffmann and Wolfgang

Chapter Outline

- How to use standard sorting methods in the Java API
- How to implement these sorting algorithms:
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Shell sort
  - Merge sort
  - Heapsort
  - Quicksort

Chapter Outline (2)

- Understand the performance of these algorithms
  - Which to use for small arrays
  - Which to use for medium arrays
  - Which to use for large arrays

Using Java API Sorting Methods

- Java API provides a class **`Arrays`** with several overloaded sort methods for different array types
- Class **`Collections`** provides similar sorting methods
- Sorting methods for arrays of primitive types:
    - Based on the *Quicksort* algorithm
- Method of sorting for arrays of objects (and **`List`**):
    - Based on *Mergesort*
- In practice you would tend to use these
    - In this class, you will implement some yourself

Java API Sorting Interface

`Arrays` methods:

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

public static void sort (Object[] a)
    // requires Comparable

public static <T> void sort (T[] a,
  Comparator<? super T> comp)
    // uses given Comparator
```

- These also have versions giving a fromIndex/toIndex range of elements to sort

Java API Sorting Interface (2)

`Collections` methods:

```
public static <T extends Comparable<T>>
    void sort (List<T> list)

public static <T> void sort (List<T> l,
    Comparator<? super T> comp)
```

- Note that these are *generic methods*, in effect having different versions for each type **`T`**
  - In reality, there is only one code body at run time

Using Java API Sorting Methods

```
int[] items;
Arrays.sort(items, 0, items.length / 2);
Arrays.sort(items);

public class Person
    implements Comparable<Person> { ... }
Person[] people;
Arrays.sort(people);
  // uses Person.compareTo

public class ComparePerson
    implements Comparator<Person> { ... }
Arrays.sort(people, new ComparePerson());
  // uses ComparePerson.compare
```

Using Java API Sorting Methods (2)

```
List<Person> plist;
Collections.sort(plist);
  // uses Person.compareTo

Collections.sort(plist,
                 new ComparePerson());
  // uses ComparePerson.compare
```

Conventions of Presentation

- Write algorithms for arrays of **`Comparable`** objects
- For convenience, examples show integers
  - These would be wrapped as **`Integer`**; or
  - You can implement separately for **`int`** arrays
- Generally use *n* for the length of the array
  - Elements *0* through *n-1*

Selection Sort

- A relatively easy to understand algorithm
- Sorts an array in ***passes***
  - Each pass selects the next smallest element
  - At the end of the pass, places it where it belongs
- Efficiency is $O(n^2)$, hence called a ***quadratic*** sort
- Performs:
  - $O(n^2)$ comparisons
  - $O(n)$ exchanges (swaps)

Selection Sort Algorithm

1. for **fill** = 0 to n-2 do // steps 2-6 form a *pass*
2. set **posMin** to **fill**
3. for **next** = **fill**+1 to n-1 do
4. if item at **next** < item at **posMin**
5. set **posMin** to **next**
6. Exchange item at **posMin** with one at **fill**

Selection Sort Example

35	65	30	60	20	scan 0-4, smallest 20
					swap 35 and 20
20	65	30	60	35	scan 1-4, smallest 30
					swap 65 and 30
20	30	65	60	35	scan 2-4, smallest 35
					swap 65 and 35
20	30	35	60	65	scan 3-4, smallest 60
					swap 60 and 60
20	30	35	60	65	done

Selection Sort Code

```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  int n = a.length;
  for (int fill = 0; fill < n-1; fill++) {
    int posMin = fill;
    for (int nxt = fill+1; nxt < n; nxt++)
      if (a[nxt].compareTo(a[posMin])<0)
        posMin = nxt;
    T tmp = a[fill];
    a[fill] = a[posMin];
    a[posMin] = tmp;
  }
}
```

Bubble Sort

- *Compares adjacent* array elements
  - *Exchanges* their values if they are *out of order*

- Smaller values *bubble up* to the top of the array
  - Larger values sink to the bottom

Bubble Sort Example

FIGURE 10.1
One Pass of Bubble Sort

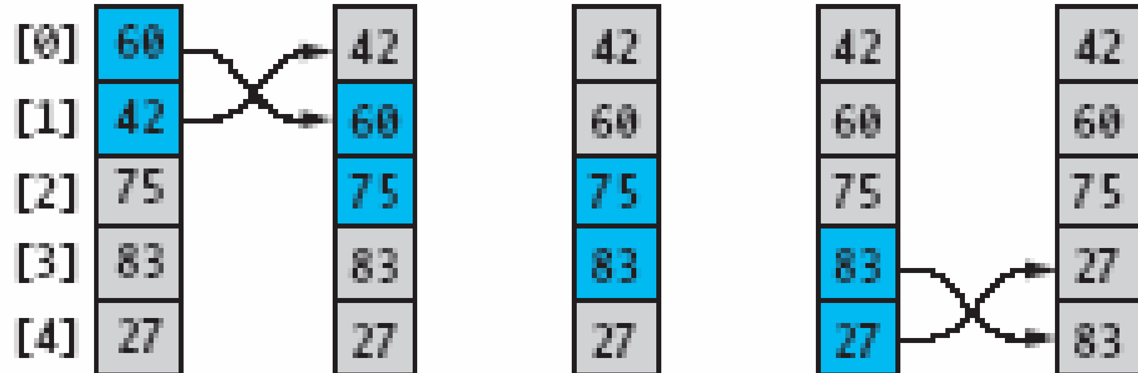


FIGURE 10.2
Array After Completion of Each Pass

Bubble Sort Algorithm

1. do
2. for each pair of adjacent array elements
3. if values are out of order
4. Exchange the values
5. while the array is not sorted

Bubble Sort Algorithm, Refined

1. do
2. Initialize **exchanges** to **false**
3. for each pair of adjacent array elements
4. if values are out of order
5. Exchange the values
6. Set **exchanges** to **true**
7. while **exchanges**

Analysis of Bubble Sort

- Excellent performance *in some cases*
  - But *very poor* performance in others!
- Works ***best*** when array is nearly sorted to begin with
- Worst case number of comparisons: $O(n^2)$
- Worst case number of exchanges: $O(n^2)$
- *Best case* occurs when the array is already sorted:
  - $O(n)$ comparisons
  - $O(1)$ exchanges (none actually)

Bubble Sort Code

```
int pass = 1;
boolean exchanges;
do {
  exchanges = false;
  for (int i = 0; i < a.length-pass; i++)
    if (a[i].compareTo(a[i+1]) > 0) {
      T tmp = a[i];
      a[i] = a[i+1];
      a[i+1] = tmp;
      exchanges = true;
    }
pass++;
} while (exchanges);
```

Insertion Sort

- Based on technique of card players to arrange a hand
  - Player keeps cards picked up so far in *sorted order*
  - When the player picks up a new card
    - Makes room for the new card
    - Then inserts it in its proper place

FIGURE 10.3
Picking Up a Hand of Cards

Insertion Sort Algorithm

- For each element from 2nd (nextPos = 1) to last:
  - Insert element at nextPos where it belongs
  - Increases sorted subarray size by 1

- To make room:
  - Hold nextPos value in a variable
  - Shuffle elements to the right until gap at right place

Insertion Sort Example

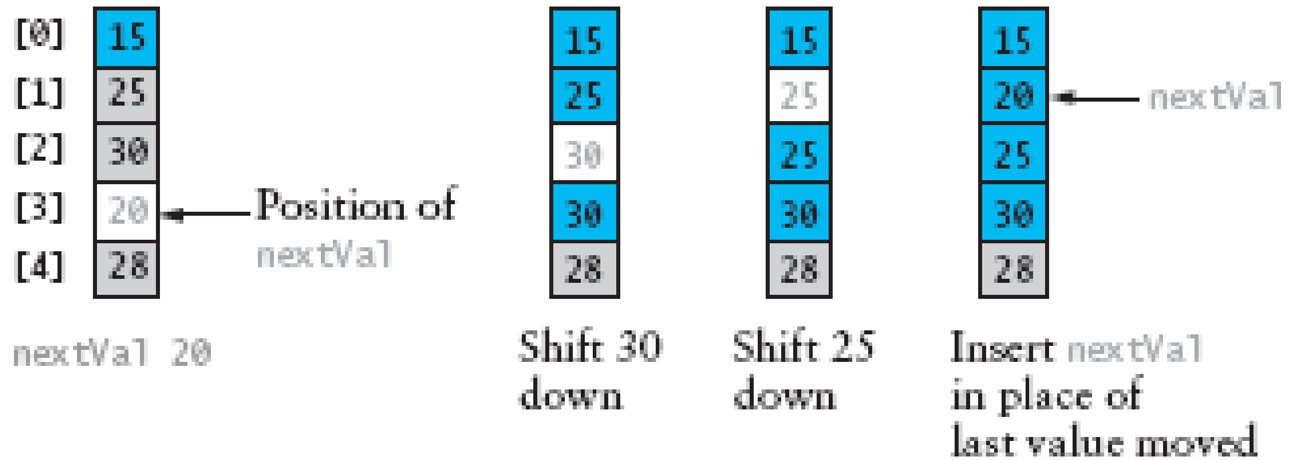


FIGURE 10.5
Inserting the Fourth Array Element

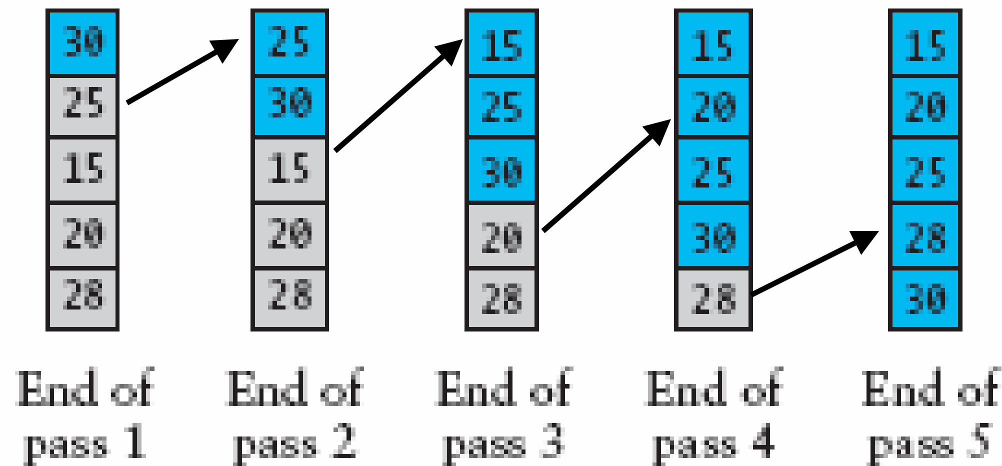


FIGURE 10.4
An Insertion Sort

Insertion Sort Code

```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  for (int nextPos = 1;
       nextPos < a.length;
       nextPos++) {
    insert(a, nextPos);
  }
}
```

Insertion Sort Code (2)

```
private static <T extends Comparable<T>>
    void insert (T[] a, int nextPos) {
  T nextVal = a[nextPos];
  while
    (nextPos > 0 &&
     nextVal.compareTo(a[nextPos-1]) < 0){
    a[nextPos] = a[nextPos-1];
    nextPos--;
  }
  a[nextPos] = nextVal;
}
```

Analysis of Insertion Sort

- Maximum number of comparisons: $O(n^2)$
- In the best case, number of comparisons: $O(n)$
- # shifts for an insertion = # comparisons - 1
  - When new value smallest so far, # comparisons
- A shift in insertion sort moves only *one item*
  - Bubble or selection sort exchange: *3 assignments*

Comparison of Quadratic Sorts

- None good for large arrays!

TABLE 10.2
Comparison of Quadratic Sorts

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

TABLE 10.3
Comparison of Rates of Growth

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Shell Sort: A Better Insertion Sort

- Shell sort is a variant of insertion sort
  - It is named after Donald Shell
  - Average performance: $O(n^{3/2})$ or better
- Divide and conquer approach to insertion sort
  - Sort many smaller subarrays using insertion sort
  - Sort progressively larger arrays
  - Finally sort the entire array
- These arrays are elements separated by a *gap*
  - Start with *large gap*
  - Decrease the gap on each “pass”

Shell Sort: The Varying Gap

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65

Before and after sorting with gap = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

Before and after sorting with gap = 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	60	35	55	62	57	60	65	70	75	75	85	80	90

Analysis of Shell Sort

- ***Intuition:***

  Reduces work by moving elements farther earlier
- Its general analysis is an open research problem
- Performance depends on sequence of gap values
- For sequence 2^k, performance is $O(n^2)$
- Hibbard's sequence (2^k-1), performance is $O(n^{3/2})$
- We start with n/2 and repeatedly divide by 2.2
  - Empirical results show this is $O(n^{5/4})$ or $O(n^{7/6})$
  - No *theoretical* basis (proof) that this holds

Shell Sort Algorithm

1. Set **gap** to n/2
2. while **gap** > 0
3. for each element from **gap** to end, by **gap**
4. Insert element in its **gap**-separated sub-array
5. if **gap** is 2, set it to 1
6. otherwise set it to **gap** / 2.2

Shell Sort Algorithm: Inner Loop

3.1 set **nextPos** to position of element to insert

3.2 set **nextVal** to value of that element

3.3 while **nextPos** > **gap** and
  element at **nextPos-gap** is > **nextVal**

3.4   Shift element at **nextPos-gap** to **nextPos**

3.5   Decrement **nextPos** by **gap**

3.6 Insert **nextVal** at **nextPos**

Shell Sort Code

```
public static <T extends <Comparable<T>>
    void sort (T[] a) {
  int gap = a.length / 2;
  while (gap > 0) {
    for (int nextPos = gap;
         nextPos < a.length; nextPos++)
      insert(a, nextPos, gap);
    if (gap == 2)
      gap = 1;
    else
      gap = (int)(gap / 2.2);
  }
}
```

Shell Sort Code (2)

```
private static <T extends Comparable<T>>
    void insert
      (T[] a, int NextPos, int gap) {
  T val = a[nextPos];
  while ((nextPos >= gap) &&
      (val.compareTo(a[nextPos-gap])<0)) {
    a[nextPos] = a[nextPos-gap];
    nextPos -= gap;
  }
  a[nextPos] = val;
}
```

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

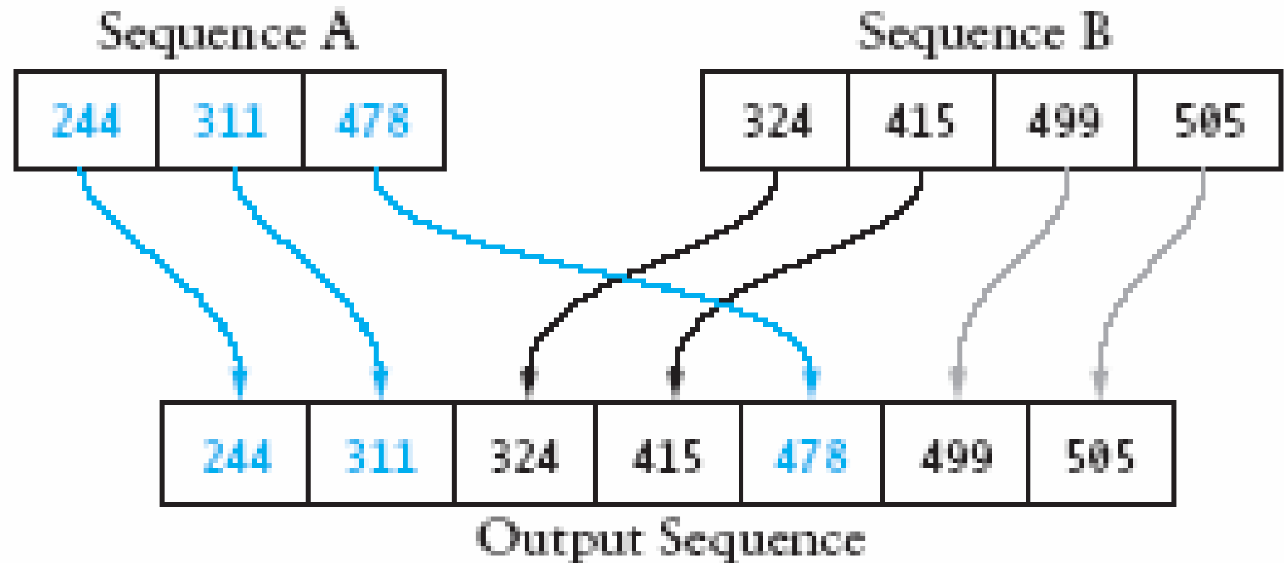
Merge Algorithm (Two Sequences)

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:

1. Split array into two halves
2. Sort the left half (recursively)
3. Sort the right half (recursively)
4. Merge the two sorted halves

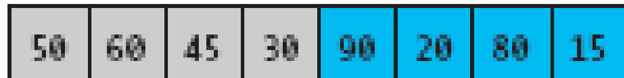
Merge *Sort* Algorithm (2)

Detailed algorithm:

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

Merge Sort Example

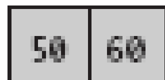
FIGURE 10.7
Trace of Merge Sort



1. *Split array into two 4-element arrays*

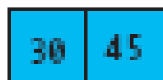
2. *Split left array into two 2-element arrays*

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++];
}
```

Heapsort

- Merge sort time is O(n log n)
  - ***But*** requires (temporarily) n extra storage items
- *Heapsort*
  - Works *in place:* no additional storage
  - Offers same O(n log n) performance
- Idea (not quite in-place):
  - Insert each element into a priority queue
  - Repeatedly remove from priority queue to array
    - Array slots go from 0 to n-1

Heapsort Picture

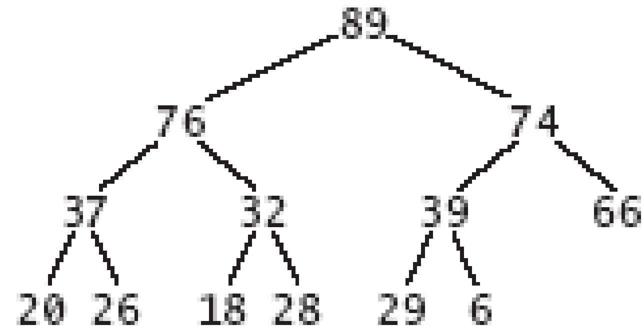


FIGURE 10.8
Example of a Heap with Largest Value in Root

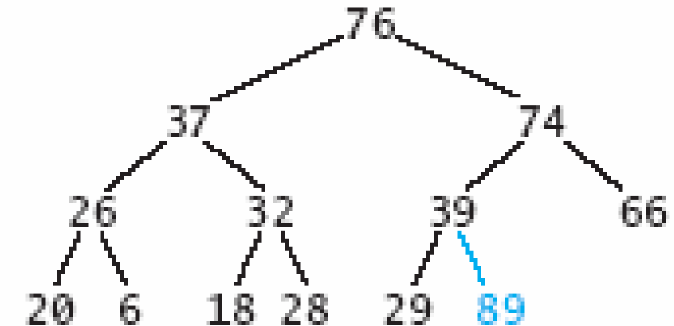


FIGURE 10.9
Heap After Removal of Largest Item

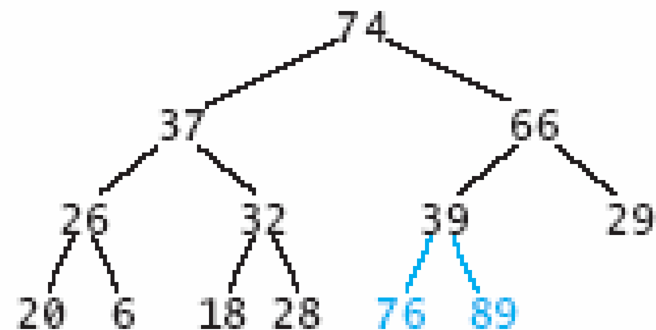


FIGURE 10.10
Heap After Removal of Two Largest Items

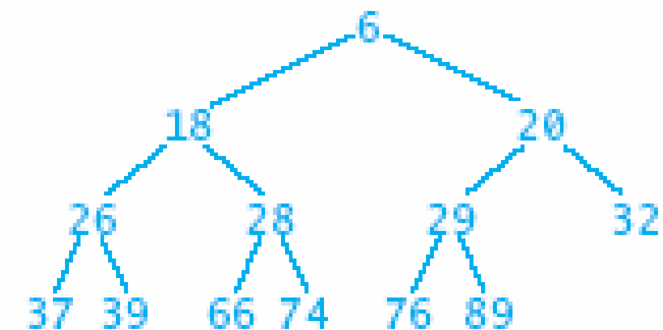


FIGURE 10.11
Heap After Removal of All Its Items

Heapsort Picture (2)

FIGURE 10.12

Internal Representation of the Heap Shown in Figure 10.8

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
89	76	74	37	32	39	66	20	26	18	28	29	6

FIGURE 10.13

Internal Representation of the Heaps Shown in Figures 10.9 through 10.11

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
76	37	74	26	32	39	66	20	6	18	28	29	89

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
74	37	66	26	32	39	29	20	6	18	28	76	89

⋮

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
6	18	20	26	28	29	32	37	39	66	74	76	89

Algorithm for In-Place Heapsort

- Build heap starting from unsorted array
- While the heap is not empty
  - Remove the first item from the heap:
    - Swap it with the last item
  - Restore the heap property

Heapsort Code

```
public static <T extends Comparable<T>>
    void sort (T[] a) {
  buildHp(a);
  shrinkHp(a);
}
private static ... void buildHp (T[] a) {
  for (int n = 2; n <= a.length; n++) {
    int chld = n-1; // add item and reheap
    int prnt = (chld-1) / 2;
    while (prnt >= 0 &&
           a[prnt].compareTo(a[chld])<0) {
      swap(a, prnt, chld);
      chld = prnt; prnt = (chld-1)/2
    } } }
```

Heapsort Code (2)

```
private static ... void shrinkHp (T[] a) {
  int n = a.length;
  for (int n = a.length-1; n > 0; --n) {
    swap(a, 0, n);  // max -> next posn
    int prnt = 0;
    while (true) {
      int lc = 2 * prnt + 1;
      if (lc >= n) break;
      int rc = lc + 1;
      int maxc = lc;
      if (rc < n &&
          a[lc].compareTo(a[rc]) < 0)
        maxc = rc;
      ....
```

Heapsort Code (3)

```
      if (a[prnt].compareTo(a[maxc])<0) {
        swap(a, prnt, maxc);
        prnt = maxc;
      } else {
        break;
      }
    }
  }
}
private static ... void swap
    (T[] a, int i, int j) {
  T tmp = a[i]; a[i] = a[j]; a[j] = tmp;
}
```

Heapsort Analysis

- Insertion cost is log i for heap of size i
  - Total insertion cost = log(n)+log(n-1)+...+log(1)
  - This is O(n log n)
- Removal cost is also log i for heap of size i
  - Total removal cost = O(n log n)

- Total cost is O(n log n)

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

44 75 23 43 55 12 64 77 33

12 33 23 43 44 55 64 77 75

12 33 23 43

55 64 77 75

12 23 33 43

75 77

Algorithm for Quicksort

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

1. if **first** < **last**
2. Partition using **pivot**, which ends in **pivIndex**
3. Apply Quicksort recursively to left subarray
4. 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);
  }
}
```

Algorithm for Partitioning

1. Set pivot value to a[fst]
2. Set **up** to fst and **down** to lst
3. do
4. Increment **up** until a[**up**] > pivot or **up** = lst
5. Decrement **down** until a[**down**] <= pivot or **down** = fst
6. if **up** < **down**, swap a[**up**] and a[**down**]
7. while **up** is to the left of **down**
8. swap a[fst] and a[**down**]
9. return **down** as pivIndex

Trace of Algorithm for Partitioning

FIGURE 10.14
Locating First Values to Exchange

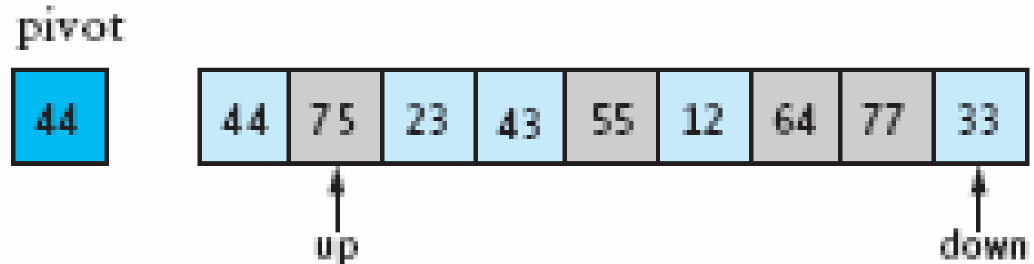


FIGURE 10.15
Array After the First Exchange

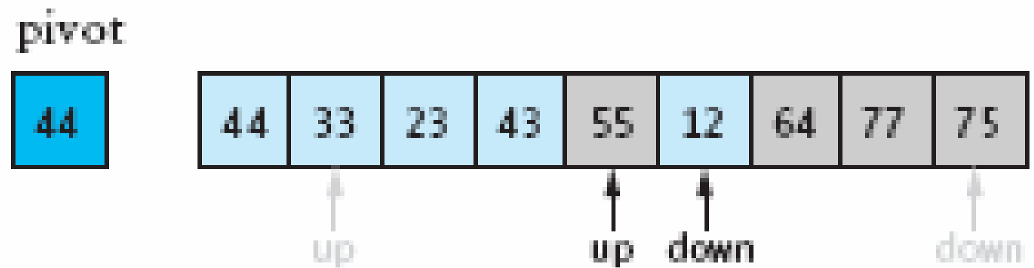
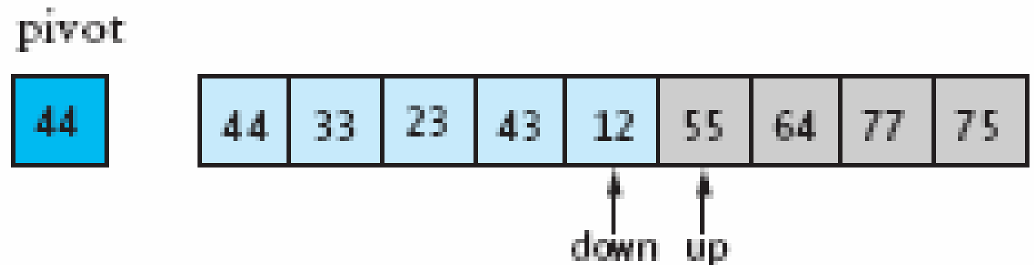


FIGURE 10.16
Array After the Second Exchange



Partitioning Code

```
private static <T extends Comparable<T>>
    int partition
      (T[] a, int fst, int lst) {
  T pivot = a[fst];
  int u = fst;
  int d = lst;
  do {
    while ((u < lst) &&
           (pivot.compareTo(a[u]) >= 0))
      u++;
    while (pivot.compareTo(a[d]) < 0)
      d++;
    if (u < d) swap(a, u, d);
  } while (u < d);
```

Partitioning Code (2)

```
    swap(a, fst, d);
    return d;
}
```

Revised Partitioning Algorithm

- Quicksort is $O(n^2)$ when each split gives 1 empty array
- This happens when the array is already sorted
- Solution approach: pick better pivot values
- Use three "marker" elements: first, middle, last
- Let pivot be one whose value is between the others

FIGURE 10.20
Sorting First, Middle, and Last Elements in Array

After sorting, median is in `table[middle]`

Testing Sortiing Algorithms

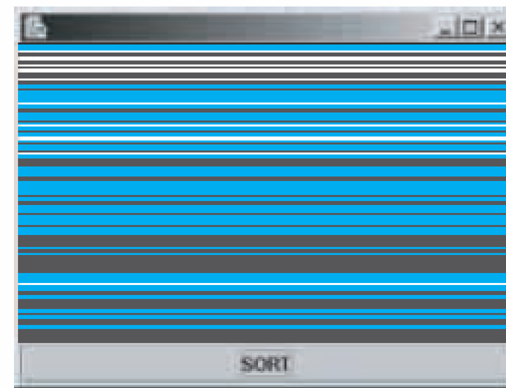
- Need to use a variety of test cases
  - Small and large arrays
  - Arrays in random order
  - Arrays that are already sorted (and reverse order)
  - Arrays with duplicate values
- Compare performance on each type of array

The Dutch National Flag Problem

- Variety of partitioning algorithms have been published
- One that partitions an array into three segments was introduced by Edsger W. Dijkstra
- Problem: partition a disordered three-color flag into three contiguous segments
- Segments represent < = > the pivot value

FIGURE 10.21
The Dutch National Flag

FIGURE 10.22
Scrambled Dutch National Flag



The Dutch National Flag Problem

Algorithm We can solve our problem by establishing the loop invariant and then executing a loop that both preserves the loop invariant and shrinks the unknown region.

1. Set `red` to `0`, `white` to `HEIGHT - 1`, and `blue` to `HEIGHT - 1`. This establishes our loop invariant with the unknown region the whole flag and the red, white, and blue regions empty.
2. `while red < white`
3. Shrink the distance between `red` and `white` while preserving the loop invariant.

FIGURE 10.23
Dutch National Flag Loop Invariant

Chapter Summary

- Three quadratic sorting algorithms:
  - Selection sort, bubble sort, insertion sort
- Shell sort: good performance for up to 5000 elements
- Quicksort: average-case O(n log n)
  - If the pivot is picked poorly, get worst case: $O(n^2)$
- Merge sort and heapsort: guaranteed O(n log n)
  - Merge sort: space overhead is O(n)
- Java API has good implementations

Chapter Summary (2)

TABLE 10.4

Comparison of Sort Algorithms

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$