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:

```java
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

```java
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
```
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 \texttt{Comparable} objects
- For convenience, examples show integers
  - These would be wrapped as \texttt{Integer}; or
  - You can implement separately for \texttt{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

<table>
<thead>
<tr>
<th>35</th>
<th>65</th>
<th>30</th>
<th>60</th>
<th>20</th>
<th>scan 0-4, smallest 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>65</td>
<td>30</td>
<td>60</td>
<td>35</td>
<td>swap 35 and 20</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>65</td>
<td>60</td>
<td>35</td>
<td>scan 1-4, smallest 30</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>35</td>
<td>60</td>
<td>65</td>
<td>scan 2-4, smallest 35</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>35</td>
<td>60</td>
<td>65</td>
<td>scan 3-4, smallest 60</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>35</td>
<td>60</td>
<td>65</td>
<td>done</td>
</tr>
</tbody>
</table>
Selection Sort Code

```java
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)
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
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 10.4**
An Insertion Sort

**Figure 10.5**
Inserting the Fourth Array Element
Insertion Sort Code

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

<table>
<thead>
<tr>
<th></th>
<th>Number of Comparisons</th>
<th>Number of Exchanges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Worst</td>
</tr>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**TABLE 10.3**
Comparison of Rates of Growth

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>64</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>1,024</td>
<td>160</td>
</tr>
<tr>
<td>64</td>
<td>4,096</td>
<td>384</td>
</tr>
<tr>
<td>128</td>
<td>16,384</td>
<td>896</td>
</tr>
<tr>
<td>256</td>
<td>65,536</td>
<td>2,048</td>
</tr>
<tr>
<td>512</td>
<td>262,144</td>
<td>4,608</td>
</tr>
</tbody>
</table>
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

Before and after sorting with gap = 7

Before and after sorting with gap = 3
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);
        }
    }
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
Figure 10.6: Merge Operation

Sequence A: 244 311 478
Sequence B: 324 415 499 505
Output Sequence: 244 311 324 415 478 499 505
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 \( t\text{Size} \leq 1 \), return (no sorting required)
2. set \( h\text{Size} \) to \( t\text{Size} / 2 \)
3. Allocate \( \text{LTab} \) of size \( h\text{Size} \)
4. Allocate \( \text{RTab} \) of size \( t\text{Size} – h\text{Size} \)
5. Copy elements 0 .. \( h\text{Size} – 1 \) to \( \text{LTab} \)
6. Copy elements \( h\text{Size} .. t\text{Size} – 1 \) to \( \text{RTab} \)
7. Sort \( \text{LTab} \) recursively
8. Sort \( \text{RTab} \) recursively
9. Merge \( \text{LTab} \) and \( \text{RTab} \) into a
Merge Sort Example

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

```java
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);
    }
```
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)
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)

```java
if (a[prnt].compareTo(a[maxc])<0) { 
    swap(a, prnt, maxc);
    prnt = maxc;
} else { 
    break;
}
```

```java
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 = $\log(n) + \log(n-1) + \ldots + \log(1)$
  • 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
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

```java
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 \( a[\text{fst}] \)
2. Set \textit{up} to \textit{fst} and \textit{down} to \textit{lst}
3. do
4. Increment \textit{up} until \( a[\text{up}] > \) pivot or \( \text{up} = \text{lst} \)
5. Decrement \textit{down} until \( a[\text{down}] \leq \) pivot or \( \text{down} = \text{fst} \)
6. if \( \text{up} < \text{down} \), swap \( a[\text{up}] \) and \( a[\text{down}] \)
7. while \( \text{up} \) is to the left of \( \text{down} \)
8. swap \( a[\text{fst}] \) and \( a[\text{down}] \)
9. return \( \text{down} \) as \( \text{pivIndex} \)
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)

```c
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
Testing Sorting 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
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n^{7/6})$</td>
<td>$O(n^{5/4})$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>