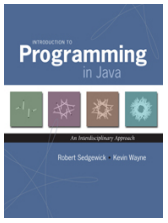
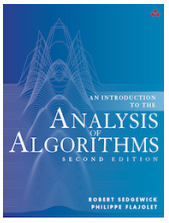
- Algorithms, 4th edition
  - 1. Fundamentals
    - 1.1 Programming Model
    - 1.2 Data Abstraction
    - 1.3 Stacks and Queues
    - 1.4 Analysis of Algorithms
    - 1.5 Case Study: Union-Find
  - 2. Sorting
    - 2.1 Elementary Sorts
    - 2.2 Mergesort
    - 2.3 Quicksort
    - 2.4 Priority Queues
    - 2.5 Sorting Applications
  - 3. Searching
    - 3.1 Symbol Tables
    - 3.2 Binary Search Trees
    - 3.3 Balanced Search Trees
    - 3.4 Hash Tables
    - 3.5 Searching Applications
  - 4. Graphs
    - 4.1 Undirected Graphs
    - 4.2 Directed Graphs
    - 4.3 Minimum Spanning Trees
    - 4.4 Shortest Paths
  - 5. Strings
    - 5.1 String Sorts
    - 5.2 Tries
    - 5.3 Substring Search
    - 5.4 Regular Expressions
    - 5.5 Data Compression
  - 6. Context
    - 6.1 Event-Driven Simulation
    - 6.2 B-trees
    - 6.3 Suffix Arrays
    - 6.4 Maxflow
    - 6.5 Reductions
    - 6.6 Intractability
- Related Booksites





2.1 Elementary Sorts

In this section, we shall study two elementary sorting methods (selection sort and insertion sort) and a variation of one of them (shellsort).

Rules of the game.

Our primary concern is algorithms for rearranging arrays of items where each item contains a *key*. The objective is to rearrange the items such that their keys are in ascending order. In Java, the abstract notion of a key is captured in a built-in mechanism—the `Comparable` interface. With but a few exceptions, our sort code refers to the data only through two operations: the method `less()` that compares objects and the method `exch()` that exchanges them.

```
private static boolean less(Comparable v, Comparable w) {
   return (v.compareTo(w) < 0);
}

private static void exch(Comparable[] a, int i, int j) {
   Comparable swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

- *Sorting cost model.* When studying sorting algorithms, we count *compares* and *exchanges*. For algorithms that do not use exchanges, we count *array accesses*.

- *Extra memory.* The sorting algorithms we consider divide into two basic types: those that sort *in place* (no extra memory except perhaps for a small function-call stack or a constant number of instance variables), and those that need enough extra memory to hold another copy of the array to be sorted.

- *Types of data.* Our sort code is effective for any type of data that implements Java's Comparable interface. This means that there is a method `compareTo()` for which `v.compareTo(w)` returns an integer that is negative, zero, or positive when v < w, v = w, or v > w, respectively. The method must implement a *total order*:
  - *Reflexive:* for all v, v = v.
  - *Antisymmetric:* for all v and w, if (v < w) then (w > v); and if (v = w) then (w = v).
  - *Transitive:* for all v, w, and x, if (v ≤ w) and (w ≤ x), then v ≤ x.

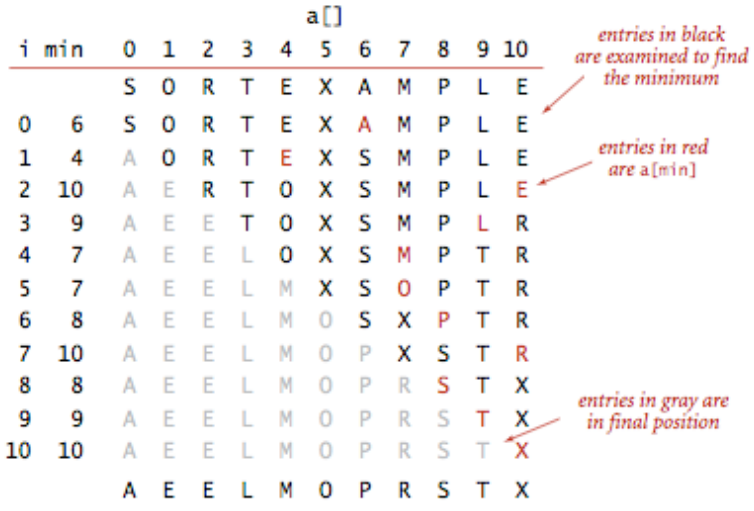
  In addition, `v.compareTo(w)` must throw an exception if `v` and `w` are of incompatible types or if either is

`null`.

Date.java illustrates how to implement the `Comparable` interface for a user-defined type.

Selection sort.

One of the simplest sorting algorithms works as follows: First, find the smallest item in the array, and exchange it with the first entry. Then, find the next smallest item and exchange it with the second entry. Continue in this way until the entire array is sorted. This method is called *selection sort* because it works by repeatedly selecting the smallest remaining item. Selection.java is an implementation of this method.



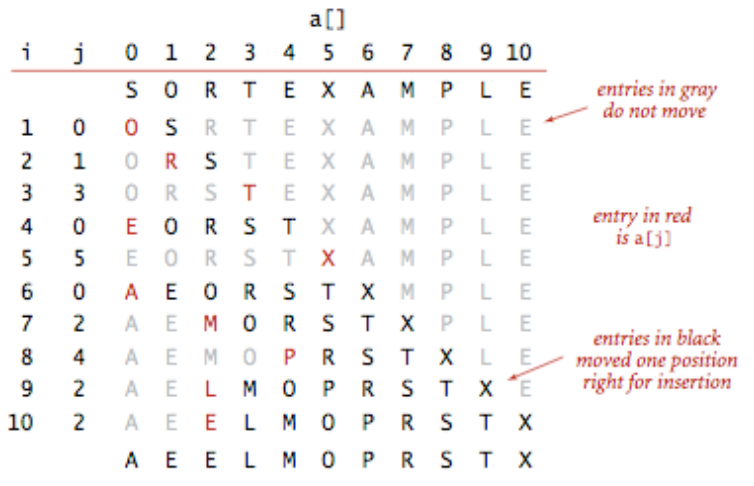
Trace of selection sort (array contents just after each exchange)

Proposition.

Selection sort uses ~$N^2/2$ compares and N exchanges to sort an array of length N.

Insertion sort.

The algorithm that people often use to sort bridge hands is to consider the cards one at a time, inserting each into its proper place among those already considered (keeping them sorted). In a computer implementation, we need to make space for the current item by moving larger items one position to the right, before inserting the current item into the vacated position. Insertion.java is an implementation of this method, which is called *insertion sort*.



Trace of insertion sort (array contents just after each insertion)

Proposition.

For randomly ordered arrays of length N with with distinct keys, insertion sort uses ~$N^2/4$ compares and ~$N^2/4$ exchanges on the average. The worst case is ~ $N^2/2$ compares and ~ $N^2/2$ exchanges and the best case is N-1 compares and 0 exchanges.

Insertion sort works well for certain types of nonrandom arrays that often arise in practice, even if they are huge. An *inversion* is a pair of keys that are out of order in the array. For instance, E X A M P L E has 11 inversions: E-A, X-A, X-M, X-P, X-L, X-E, M-L, M-E, P-L, P-E, and L-E. If the number of inversions in an array is less than a constant multiple of the array size, we say that the array is *partially sorted*.

Proposition.

The number of exchanges used by insertion sort is equal to the number of inversions in the array, and the number of compares is at least equal to the number of inversions and at most equal to the number of inversions plus the array size.

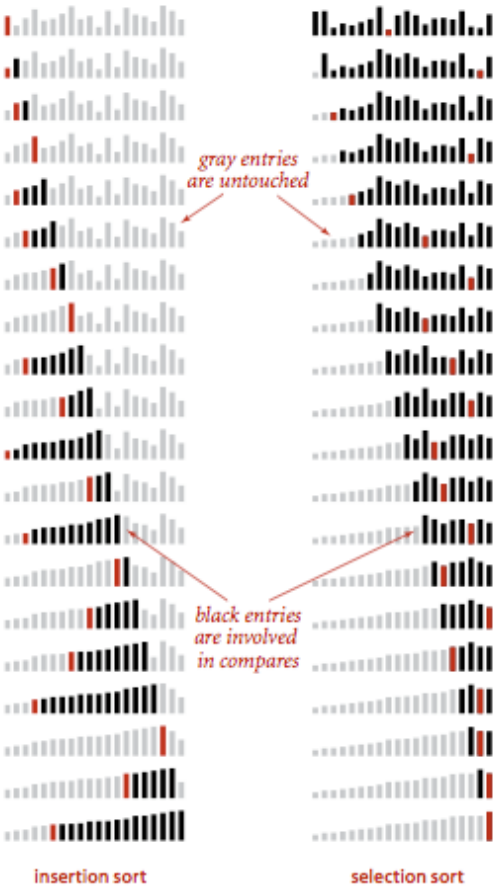
Property.

For randomly ordered arrays of distinct values, the running times of insertion sort and selection sort are quadratic and within a small constant factor of one another.

SortCompare.java uses the `sort()` methods in the classes named as command-line arguments to perform the given number of experiments (sorting arrays of the given size) and prints the ratio of the observed running times of the algorithms.

Visualizing sorting algorithms.

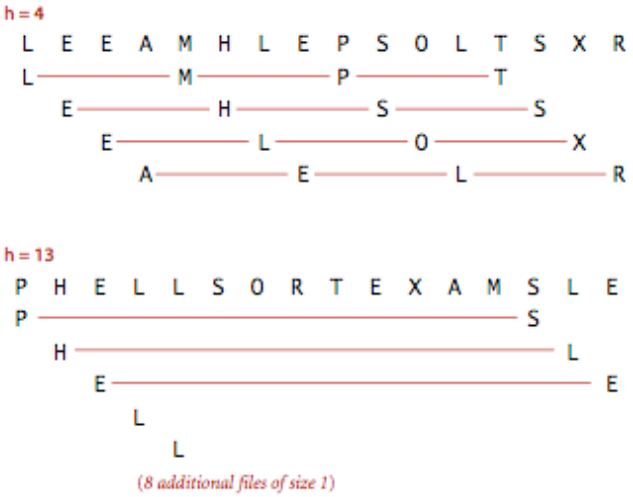
We use a simple visual representation to help describe the properties of sorting algorithms. We use vertical bars, to be sorted by their heights. SelectionBars.java and InsertionBars.java produce these visualizations.



Visual traces of elementary sorting algorithms

Shellsort.

Shellsort is a simple extension of insertion sort that gains speed by allowing exchanges of entries that are far apart, to produce partially sorted arrays that can be efficiently sorted, eventually by insertion sort. The idea is to rearrange the array to give it the property that taking every *h*th entry (starting anywhere) yields a sorted sequence. Such an array is said to be *h-sorted*.



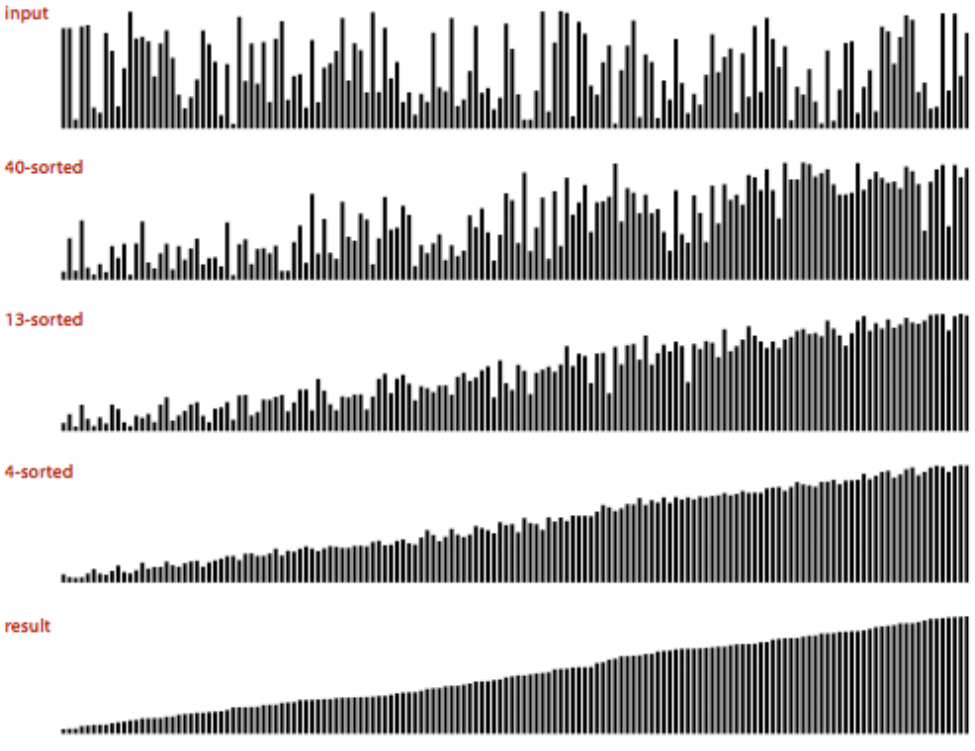
An h-sorted file is h interleaved sorted files

By h-sorting for some large values of h, we can move entries in the array long distances and thus make it easier to h-sort for smaller values of h. Using such a procedure for any increment sequence of values of h that ends in 1 will produce a sorted array: that is shellsort. Shell.java is an implementation of this method.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	A	P	H	O	L	T	S	X	R	M	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	L	L	P	S	O	R	T	S	X	E
	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	E	L	E	A	M	H	L	E	P	S	O	L	T	S	X	R
	E	E	L	A	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	M	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	L	M	E	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	O	P	S	L	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X
result	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Detailed trace of shellsort (insertions)

ShellBars.java produces a visualization of shellsort.



Visual trace of shellsort

Property.

The number of compares used by shellsort with the increments 1, 4, 13, 40, 121, 364, ... is bounded by a small multiple of N times the number of increments used.

Proposition.

The number of compares used by shellsort with the increments 1, 4, 13, 40, 121, 364, ... is $O(N^{3/2})$.

Q + A

Q. The compiler gives a warning when I compile Insertion.java. Is ther any way to avoid this?

```
Insertion.java:73: warning: [unchecked] unchecked call to compareTo(T)
                   as a member of the raw type java.lang.Comparable
        return (v.compareTo(w) < 0);
```

A. Yes, if you use static generics, as in InsertionPedantic.java. It leads to awkward (but warning-free) code.

Exercises

1. Show in the style of the example trace with selection sort, how selection sort sorts the array

   ```
   E A S Y Q U E S T I O N
   ```

   Solution.

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		E	A	S	Y	Q	U	E	S	T	I	O	N
0	1	E	A	S	Y	Q	U	E	S	T	I	O	N
1	1	A	E	S	Y	Q	U	E	S	T	I	O	N
2	6	A	E	S	Y	Q	U	E	S	T	I	O	N
3	9	A	E	E	Y	Q	U	S	S	T	I	O	N
4	11	A	E	E	I	Q	U	S	S	T	Y	O	N
5	10	A	E	E	I	N	U	S	S	T	Y	O	Q
6	11	A	E	E	I	N	O	S	S	T	Y	U	Q
7	7	A	E	E	I	N	O	Q	S	T	Y	U	S
8	11	A	E	E	I	N	O	Q	S	T	Y	U	S
9	11	A	E	E	I	N	O	Q	S	S	Y	U	T
10	10	A	E	E	I	N	O	Q	S	S	T	U	Y
11	11	A	E	E	I	N	O	Q	S	S	T	U	Y
		A	E	E	I	N	O	Q	S	S	T	U	Y

2. What is the maximum number of exchanges involving any particular item during selection sort? What is the average number of exchanges involving an item?

   Solution. The average number of exchanges is exactly 1 because there are exactly N exchanges and N items. The maximum number of exchanges is N, as in the following example.

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		Z	A	B	C	D	E	F	G	H	I	J	K
0	1	Z	A	B	C	D	E	F	G	H	I	J	K
1	2	A	Z	B	C	D	E	F	G	H	I	J	K
2	3	A	B	Z	C	D	E	F	G	H	I	J	K
3	4	A	B	C	Z	D	E	F	G	H	I	J	K
4	5	A	B	C	D	Z	E	F	G	H	I	J	K
5	6	A	B	C	D	E	Z	F	G	H	I	J	K
6	7	A	B	C	D	E	F	Z	G	H	I	J	K
7	8	A	B	C	D	E	F	G	Z	H	I	J	K
8	9	A	B	C	D	E	F	G	H	Z	I	J	K
9	10	A	B	C	D	E	F	G	H	I	Z	J	K
10	11	A	B	C	D	E	F	G	H	I	J	Z	K
11	11	A	B	C	D	E	F	G	H	I	J	K	Z
		A	B	C	D	E	F	G	H	I	J	K	Z

4. Show in the style of the example trace with insertion sort, how insertion sort sorts the array

   E A S Y Q U E S T I O N

   Solution.

		a[ ]											
i	j	0	1	2	3	4	5	6	7	8	9	10	11
		E	A	S	Y	Q	U	E	S	T	I	O	N
0	0	E	A	S	Y	Q	U	E	S	T	I	O	N
1	0	A	E	S	Y	Q	U	E	S	T	I	O	N
2	2	A	E	S	Y	Q	U	E	S	T	I	O	N
3	3	A	E	S	Y	Q	U	E	S	T	I	O	N
4	2	A	E	Q	S	Y	U	E	S	T	I	O	N
5	4	A	E	Q	S	U	Y	E	S	T	I	O	N
6	2	A	E	E	Q	S	U	Y	S	T	I	O	N
7	5	A	E	E	Q	S	S	U	Y	T	I	O	N
8	6	A	E	E	Q	S	S	T	U	Y	I	O	N
9	3	A	E	E	I	Q	S	S	T	U	Y	O	N
10	4	A	E	E	I	O	Q	S	S	T	U	Y	N
11	4	A	E	E	I	N	O	Q	S	S	T	U	Y
		A	E	E	I	N	O	Q	S	S	T	U	Y

6. Which method runs fastest for an array with all keys identical, selection sort or insertion sort?

   Solution. Insertion sort runs in linear time when all keys are equal.

8. Suppose that we use insertion sort on a randomly ordered array where items have only one of three key values. Is the running time linear, quadratic, or something in between?

   Solution. Quadratic.

9. Show in the style of the example trace with shellsort, how shellsort sort sorts the array

E A S Y S H E L L S O R T Q U E S T I O N

Solution.

			a[ ]																			
h	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
			E	A	S	Y	S	H	E	L	L	S	O	R	T	Q	U	E	S	T	I	O	N
13	13	13	E	A	S	Y	S	H	E	L	L	S	O	R	T	Q	U	E	S	T	I	O	N
13	14	14	E	A	S	Y	S	H	E	L	L	S	O	R	T	Q	U	E	S	T	I	O	N
13	15	2	E	A	E	Y	S	H	E	L	L	S	O	R	T	Q	U	S	S	T	I	O	N
13	16	3	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
13	17	17	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
13	18	18	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
13	19	19	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
13	20	20	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
			E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
4	4	4	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
4	5	5	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
4	6	6	E	A	E	S	S	H	E	L	L	S	O	R	T	Q	U	S	Y	T	I	O	N
4	7	3	E	A	E	L	S	H	E	S	L	S	O	R	T	Q	U	S	Y	T	I	O	N
4	8	4	E	A	E	L	L	H	E	S	S	S	O	R	T	Q	U	S	Y	T	I	O	N
4	9	9	E	A	E	L	L	H	E	S	S	S	O	R	T	Q	U	S	Y	T	I	O	N
4	10	10	E	A	E	L	L	H	E	S	S	S	O	R	T	Q	U	S	Y	T	I	O	N
4	11	7	E	A	E	L	L	H	E	R	S	S	O	S	T	Q	U	S	Y	T	I	O	N
4	12	12	E	A	E	L	L	H	E	R	S	S	O	S	T	Q	U	S	Y	T	I	O	N
4	13	9	E	A	E	L	L	H	E	R	S	Q	O	S	T	S	U	S	Y	T	I	O	N
4	14	14	E	A	E	L	L	H	E	R	S	Q	O	S	T	S	U	S	Y	T	I	O	N
4	15	15	E	A	E	L	L	H	E	R	S	Q	O	S	T	S	U	S	Y	T	I	O	N
4	16	16	E	A	E	L	L	H	E	R	S	Q	O	S	T	S	U	S	Y	T	I	O	N
4	17	17	E	A	E	L	L	H	E	R	S	Q	O	S	T	S	U	S	Y	T	I	O	N
4	18	10	E	A	E	L	L	H	E	R	S	Q	I	S	T	S	O	S	Y	T	U	O	N
4	19	7	E	A	E	L	L	H	E	O	S	Q	I	R	T	S	O	S	Y	T	U	S	N
4	20	8	E	A	E	L	L	H	E	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
			E	A	E	L	L	H	E	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
1	1	0	A	E	E	L	L	H	E	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
1	2	2	A	E	E	L	L	H	E	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
1	3	3	A	E	E	L	L	H	E	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
1	4	4	A	E	E	L	L	H	E	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
1	5	3	A	E	E	H	L	L	E	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
1	6	3	A	E	E	E	H	L	L	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y

1	7	7	A	E	E	E	H	L	L	O	N	Q	I	R	S	S	O	S	T	T	U	S	Y
1	8	7	A	E	E	E	H	L	L	N	O	Q	I	R	S	S	O	S	T	T	U	S	Y
1	9	9	A	E	E	E	H	L	L	N	O	Q	I	R	S	S	O	S	T	T	U	S	Y
1	10	5	A	E	E	E	H	I	L	L	N	O	Q	R	S	S	O	S	T	T	U	S	Y
1	11	11	A	E	E	E	H	I	L	L	N	O	Q	R	S	S	O	S	T	T	U	S	Y
1	12	12	A	E	E	E	H	I	L	L	N	O	Q	R	S	S	O	S	T	T	U	S	Y
1	13	13	A	E	E	E	H	I	L	L	N	O	Q	R	S	S	O	S	T	T	U	S	Y
1	14	10	A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	T	T	U	S	Y
1	15	15	A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	T	T	U	S	Y
1	16	16	A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	T	T	U	S	Y
1	17	17	A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	T	T	U	S	Y
1	18	18	A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	T	T	U	S	Y
1	19	16	A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	S	T	T	U	Y
1	20	20	A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	S	T	T	U	Y
			A	E	E	E	H	I	L	L	N	O	O	Q	R	S	S	S	S	T	T	U	Y

10. Why not use selection sort for *h*-sorting in shellsort?

    Solution. Insertion sort is faster on inputs that are partially-sorted.

Creative Problems

15. **Expensive exchange.** A clerk at a shipping company is charged with the task of rearranging a number of large crates in order of the time they are to be shipped out. Thus, the cost of compares is very low (just look at the labels) relative to the cost of exchanges (move the crates). The warehouse is nearly full: there is extra space sufficient to hold any one of the crates, but not two. Which sorting method should the clerk use?

    Solution. Use selection sort because it minimizes the number of exchanges.

18. **Visual trace.** Modify your solution to the previous exercise to make Insertion.java and Selection.java produce visual traces such as those depicted in this section.

    Solution. TraceInsertion.java, TraceSelection.java, and TraceShell.java.

21. **Comparable transactions.** Expand your implementation of Transaction.java so that it implements `Comparable`, such that transactions are kept in order by amount.

22. **Transaction sort test client.** Write a class SortTransactions.java that consists of a static method `main()` that reads a sequence of transactions from standard input, sorts them, and prints the result on standard output.

Experiments

23. **Insertion sort with sentinel.** Develop an implementation InsertionX.java of insertion sort that eliminates the j > 0 test in the inner loop by first putting the smallest item into position. Use SortCompare.java to evaluate the effectiveness of doing so. *Note*: it is often possible to avoid an index-out-of-bounds test in this way—the item that enables the test to be eliminated is known as a *sentinel*.

24. **Insertion sort without exchanges.** Develop an implementation InsertionX.java of insertion sort that moves larger items to the right one position rather than doing full exchanges. Use SortCompare.java to

evaluate the effectiveness of doing so.

Web Exercises

1. **Sorting networks.** Write a program Sort3.java with three `if` statements (and no loops) that reads in three integers *a*, *b*, and *c* from the command line and prints them out in ascending order.

```
if (a > b) swap a and b
if (a > c) swap a and c
if (b > c) swap b and c
```

2. **Oblivious sorting network.** Convince yourself that the following code fragment rearranges the integers stored in the variables A, B, C, and D so that A <= B <= C <= D.

```
if (A > B) { t = A; A = B; B = t; }
if (B > C) { t = B; B = C; C = t; }
if (A > B) { t = A; A = B; B = t; }
if (C > D) { t = C; C = D; D = t; }
if (B > C) { t = B; B = C; C = t; }
if (A > B) { t = A; A = B; B = t; }
if (D > E) { t = D; D = E; E = t; }
if (C > D) { t = C; C = D; D = t; }
if (B > C) { t = B; B = C; C = t; }
if (A > B) { t = A; A = B; B = t; }
```

   Devise a sequence of statements that would sort 5 integers. How many `if` statements does your program use?
3. **Optimal oblivious sorting networks.** Create a program that sorts four integers using only 5 `if` statements, and one that sorts five integers using only 9 `if` statements of the type above? Oblivious sorting networks are useful for implementing sorting algorithms in hardware. How can you check that your program works for all inputs?

   Answer: Sort4.java sorts 4 items using 5 compare-exchanges. Sort5.java sorts 5 items using 9 compare-exchanges.

   The 0-1 principle says that you can verify the correctness of a (deterministic) sorting network by checking whether it correctly sorts an input that is a sequence of 0s and 1s. Thus, to check that `Sort5.java` works, you only need to test it on the 2^5 = 32 possible inputs of 0s and 1s.

4. **Optimal oblivious sorting (challenging).** Find an optimal sorting network for 6, 7, and 8 inputs, using 12, 16, and 19 `if` statements of the form in the previous problem, respectively.

   Answer: Sort6.java is the solution for sorting 6 items.

5. **Optimal non-oblivious sorting.** Write a program that sorts 5 inputs using only 7 comparisons. *Hint*: First compare the first two numbers, the second two numbers, and the larger of the two groups, and label them so that a < b < d and c < d. Second, insert the remaining item e into its proper place in the chain a < b < d by first comparing against b, then either a or d depending on the outcome. Third, insert c into the proper place in the chain involving a, b, d, and e in the same manner that you inserted e (with the knowledge that c < d). This uses 3 (first step) + 2 (second step) + 2 (third step) = 7 comparisons. This method was first discovered by H. B. Demuth in 1956.
6. **Stupidsort.** Analyze the running time (worst case and best case), correctness, and stability of the following sorting algorithm. Scan the array from left to right until you find two consecutive items that are out-of-place. Swap them, and start over from the beginning. Repeat until the scan reaches the end of the array.

```
for (int i = 1; i < N; i++) {
```

```
      if (less(a[i], a[i-1])) {
         exch(i, i-1);
         i = 0;
      }
   }
```

Consider also the following recursive variant and analyze the worst case memory usage.

```
public static void sort(Comparable[] a) {
   for (int i = 1; i < a.length; i++) {
      if (less(a[i], a[i-1])) {
         exch(i, i-1);
         sort(a);
      }
   }
}
```

7. **Stoogesort.** Analyze the running time and correctness of the following recursive sorting algorithm: if the leftmost item is larger than the rightmost item, swap them. If there are 2 or more items in the current subarray, (i) sort the initial two-thirds of the array recursively, (ii) sort the final two-thirds of the array, (iii) sort the initial two-thirds of the array again.
8. **Guess-sort.** Pick two indices i and j at random; if a[i] > a[j], then swap them. Repeat until the input is sorted. Analyze the expected running time of this algorithm. *Hint*: after each swap, the number of inversions strictly decreases. If there are m bad pairs, then the expected time to find a bad pair is Theta(n^2/m). Summing up from m =1 to n^2 yields O(N^2 log N) overall, ala coupon collector. This bound is tight: consider input 1 0 3 2 5 4 7 6 ...
9. **Bogosort.** Bogosort is a randomized algorithm that works by throwing the N cards up in the air, collecting them, and checking whether they wound up in increasing order. If they didn't, repeat until they do. Implement bogosort using the shuffling algorithm from Section 1.4. Estimate the running time as a function of N.
10. **Slow sort.** Consider the following sorting algorithm: choose two integer i and j at random. If i < j, but a[i] > a[j], swap them. Repeat until the array is in ascending order. Argue that the algorithm will eventually finish (with probability 1). How long will it takes as a function of N? *Hint*: How many swaps will it make in the worst case?
11. **Minimum number of moves to sort an array.** Given a list of N keys, a *move operation* consists of removing any one key from the list and appending it to the end of the list. No other operations are permitted. Design an algorithm that sorts a given list using the minimum number of moves.
12. **Guess-Sort.** Consider the following exchanged-based sorting algorithm: pick two random indices; if a[i] and a[j] are an inversion, swap them; repeat. Show that the expected time to sort an array of size N is at most N^2 log N. See this paper for an analysis and related sorting algorithm known as Fun-Sort.
13. **Swapping an inversion.** Given an array of N keys, let a[i] and a[j] be an inversion (i < j but a[i] > a[j]). Prove or disprove: swapping a[i] and a[j] strictly decreases the number of inversions.

Last modified on September 19, 2014.