Sorting

Java API sort methods, simple sorts, selection sort, insertion sort, sort computational complexitites, mergesort, quicksort

Introduction to sorting

Sorting means arranging the values in a list so that they occur in a specific order.

- Most sorting is comparison based, using <, <=, >, >=, and compareTo().
- Classes that implement Comparable<T> have instances eligible for sorting in a collection.

Sorts built into the Java API

Classes supporting sorts:

- Arrays
  - public static void sort(Object[] a).
  - public static void sort(Object[] a, int fromIndex, int toIndex)
  - Similar sorts on primitive types.
- Collections
  - public static void sort(List list).
- The methods above sort the specified array/list into ascending order, according to the *natural ordering* of its elements.
- Some methods sort according to a specific comparator.

Calling Arrays.sort()

- The Arrays class contains various methods for manipulating arrays (such as sorting and searching).
- In the java.util package.
- Example:

```
String[] names = {"John", "Paul", "George", "Ringo"};
Arrays.sort(names);
// names = [“George”, “John”, “Paul”, “Ringo”]
```

Calling Collections.sort()

- The Collections class consists exclusively of static methods that operate on or return collections.
- In the java.util package.
- Example:

```
List<String> list = new LinkedList<String>();
list.add("John");
list.add("Paul");
list.add("George");
list.add("Ringo");
Collections.sort(list);
// list = [“George”, “John”, “Paul”, “Ringo”]
```

Simple sorting algorithms

Three simple sorts are useful for small amounts of data:

- **Selection sort**. The front part of the array consists of the smallest elements and is completely sorted. Technique: look for the smallest element not in that front part and move it to the end of the front part.
- **Insertion sort**. The front part of the array is sorted relative to its elements. Technique: take the first element of the array not in the front part and insert it into its proper position in the front part.
- **Bubble sort**. Swap out of order adjacent pairs of elements.

All of these can be applied to arrays. Insertion sorts are easily applied to linked lists.

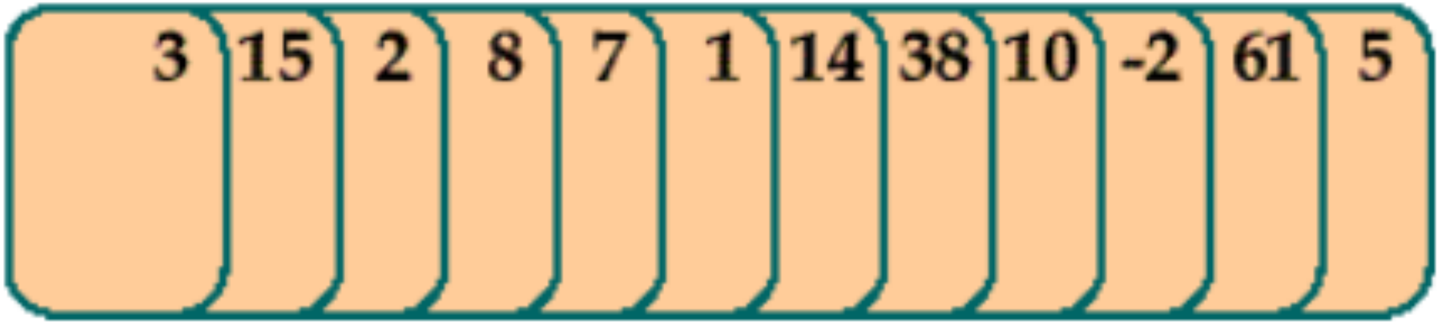
Selection sort

Algorithm

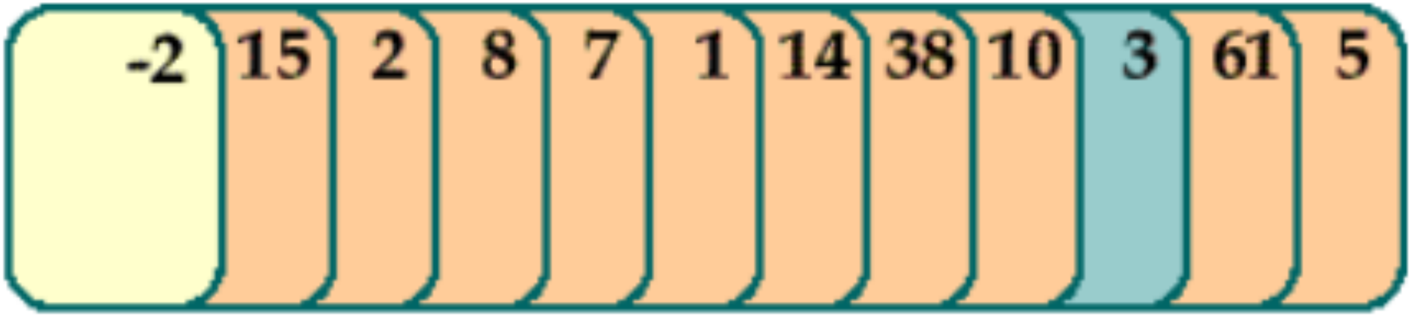
- Find smallest array element; exchange it with the element at index 0.
- Find second smallest element; exchange with the element at index 1.
- ***n***th step: Find the ***n***th smallest element and exchange it with the element at index ***n*** – 1.

Selection sorting a card deck

Deck of cards before the algorithm begins:

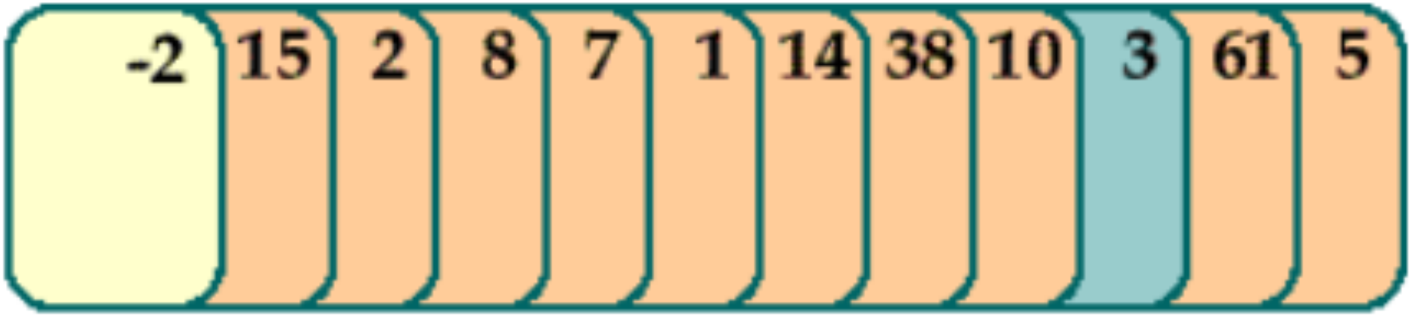


Deck after properly placing the smallest element.

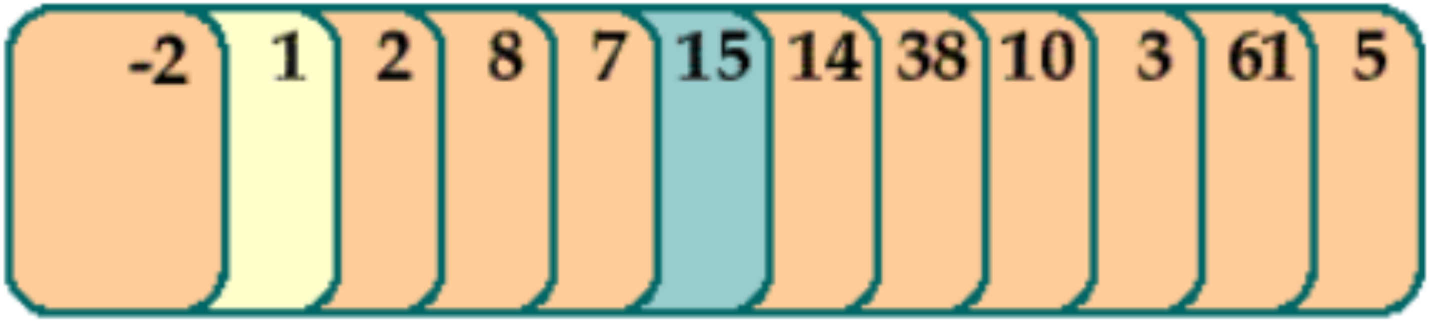


Selection sort (cont)

Deck after properly placing the smallest element.



Deck after properly placing the second smallest element.



Selection sort code

```
public static void selectionSort(String[] a)
{
   for (int k = 0; k < a.length - 1; k++) {
     // Put the appropriate item into kth position.

     int small = k; // Subscript of smallest item found
                    // so far among k through a.length - 1
     // Look for smallest item
     for(int j = k + 1; j < a.length; j++)
        if (a[small].compareTo(a[j]) > 0)
           small = j;
     // Exchange the smallest item and the kth item
     String temp = a[k];
     a[k] = a[small];
     a[small] = temp;
   }
}
```

Tracing selection sort on ints

```
int[] list = {13,15,27,1,14,38,-2,61,5};
```

k	a[small]	list (at end of outer for loop)								
0	-2	-2	15	27	1	14	38	13	61	5
1	1	-2	1	27	15	13	14	38	61	5
2	5	-2	1	5	15	13	14	38	61	27
3	13	-2	1	5	13	15	14	38	61	27
4	14	-2	1	5	13	14	15	38	61	27
5	15	-2	1	5	13	14	15	38	61	27
6	27	-2	1	5	13	14	15	27	61	38
7	38	-2	1	5	13	14	15	27	38	61

Selection sort efficiency

Assume the array has n elements.

1. Pass 1: examine n – 1 elements.
2. Pass 2: examine n – 2 elements.
3. … Pass k: examine n – k elements.

Number of elements examined:

$\sum (k - 1)$, where k goes from n to 1

$= (n)(n - 1)/2$

= polynomial of degree 2

→ $O(n^2)$ computational complexity

Insertion sort

Algorithm

- Break the array/list into two parts. The first part is sorted and the second is not sorted. Initially, the first part is empty.
- Handle each element of the second part, one at a time. Insert the element into its proper position among the elements of the first part.
- For each insertion, the first part grows by 1 element and the second part decreases by one element.

Tracing insertion sort on ints

```
int[] list = {13,15,27,1,14,38,-2,61,5};
```

k	a[k]	list (at end of outer for loop)								
0	13	13	15	27	1	14	38	-2	61	5
1	15	13	15	27	1	14	38	-2	61	5
2	27	13	15	27	1	14	38	-2	61	5
3	1	1	13	15	27	14	38	-2	61	5
4	14	1	13	14	15	27	38	-2	61	5
5	38	1	13	14	15	27	38	-2	61	5
6	-2	-2	1	13	14	15	27	38	61	5
7	61	-2	1	13	14	15	27	38	61	5
8	5	-2	1	5	13	14	15	27	38	61

Insertion sort code

```
public static void insertionSort(int[] a) {
   for (int k = 1; k < a.length; k++) {
   // Insert kth element into its proper position
     int nextElement = a[k];
     int j = k - 1;
     while (j >= 0 && nextElement < a[j]) {
        // Shift the front of the array
        a[j + 1] = a[j];
        j = j - 1;
     }
     // Insert into the "open" slot
     a[j + 1] = nextElement;
     }
}
```

Insertion sort commentary

- Insertion sort has the same computational complexity as selection sort – O(n^2)
- Insertion sort can be used on linked lists since the shifting there is so easy.
- Be careful when writing array code. It's easy to be off by 1.

Merge Sort

Merge sort algorithm

1. Divide a list into two pieces.
2. Sort each piece.
3. Merge the two pieces together to form a sorted list.

- Merge sort is O(n log n) for both worst and average case performance.
- Invented in 1945 by John VonNeumann.
- A good example of a **divide and conquer** algorithm.
- Nice recursive solution. All the work is done via the merge, beginning with lists of 1 element.

Aside – Arrays.copyOfRange( )

- Mergesort on an array requires splitting the array into two pieces.
- **`Arrays`** can make a copy of part of an array with:
  `Arrays.copyOfRange(array, from, to)`:
  - **`array`** is the array to copy part of
  - **`from`** is the first index of the copied part
  - to is 1 + the last index of the copied part
- Example:

  -
    ```
    int[] array = {4, 18, 20, 37};
    arrayB = Arrays.copyOfRange(array, 0, 3);
    // arrayB is {4, 18, 20}
    ```

Mergesort code (int[] array)

```
public static void mergeSort(int[] a) {
  if (a.length >= 2) {
    int[] left = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right =
           Arrays.copyOfRange(a, a.length/2, a.length);
    mergeSort(left);
    mergeSort(right);
    merge(a, left, right);
  }
}
```

All of the heavy lifting for mergeSort is in the merging.

Merge code

```
// Merge arrays b and c into a
public static void merge(int[] a, int[] b, int[] c) {
  int indexB = 0;
  int indexC = 0;
  for (int k = 0; k < a.length; k++) {
     if (indexC >= c.length || (indexB < b.length
                            && b[indexB] <= c[indexC]))
           a[k] = b[indexB++];
     else
           a[k] = c[indexC++];
  }
}
```

Quicksort

Quicksort algorithm

1. Select an element as a pivot element.
2. Move all elements smaller than the pivot into the first part of the array (list).
3. Move all elements larger than the pivot into the second part of the array.
4. Quicksort each part.

- Developed by Tony Hoare in 1960
- Worst case complexity O(n^2)
- Average case complexity O(n log n)
- Space efficient

Swapping array elements

- Quicksorts are done in place – there is no copying of arrays or parts of arrays.
- Quicksort requires swapping array elements.
- The following method swaps elements of an int array. Any type would work.

```
// Swaps the values of array elements at j and k
private static void swap(int[] array, int j, int k)
{
   int temp = array[j];
   array[j] = array[k];
   array[k] = temp;
}
```

Quicksort code

```
public static void quicksort(int[] a,
                             int start, int finish) {
   if (start < finish) {
      int pivot = a[finish];
      int index = start - 1;
      for (int k = start; k < finish; k++) {
         if (a[k] <= pivot) {
            index++;
            swap(a, index, k);
         }
      }
      swap(a, index + 1, finish);
      quicksort(a, start, index);
      quicksort(a, index + 2, finish);
   }
}
```