Sorting Algorithms

Probably the most frequent operation computers do.
Sorting

- *Sorting* is the process of arranging a group of *n* elements into a defined order based on particular criteria.
- Many sorting algorithms have been designed.
- *Sequential sorts* require approximately $O(n^2)$ comparisons.
- *Logarithmic sorts* typically require $O(n \times \log n)$ comparisons.
- The elements must be **Comparable**, that is, we must be able to *compare* one element to another.
Given a dataset with Comparable data, (without loss of generality, let’s say they are integers)

You want to sort that data (increasing or decreasing order).
public class SortPhoneList
{
    public static void main(String[] args)
    {
        Contact[] friends = new Contact[7];

        friends[0] = new Contact("John", "Smith", "610-555-7384");
        friends[2] = new Contact("Mark", "Riley", "733-555-2969");

        Sorting.selectionSort(friends);

        for (Contact friend : friends)
            System.out.println(friend);
    }
}
/**
 * Contact represents a phone contact.
 * @author Java Foundations
 */

public class Contact implements Comparable<Contact>
{

    private String firstName, lastName, phone;

    /**
     * Sets up this contact with the specified information.
     *
     * @param first     a string representation of a first name
     * @param last      a string representation of a last name
     * @param telephone a string representation of a phone number
     */

    public Contact(String first, String last, String telephone)
    {
        firstName = first;
        lastName = last;
        phone = telephone;
    }

Example of data to sort
Example on value to sort on

/**
 * Uses both last and first names to determine lexical ordering.
 *
 * @param other the contact to be compared to this contact
 * @return the integer result of the comparison
 */

public int compareTo(Contact other)
{
    int result;

    if (lastName.equals(other.lastName))
        result = firstName.compareTo(other.firstName);
    else
        result = lastName.compareTo(other.lastName);

    return result;
}
Selection Sort
Select the next element to place in its proper position
Sorting using selectionSort

SelectionSort orders values by repeatedly putting a particular value into its final sorted position.

The algorithm:
1. find the smallest value in the dataset
2. switch it with the value in the first position
3. find the next smallest value in the dataset
4. switch it with the value in the second position
5. repeat until all values are in their sorted positions

Visualization [https://visualgo.net/sorting](https://visualgo.net/sorting)
public static void selectionSort (Comparable[] data){
    int min;

    for(int index = 0; index < data.length - 1; index++){
        min = index;

        for(int scan = index+1; scan < data.length; scan++)
            if(data[scan].compareTo(data[min]) < 0)
                min = scan;

        swap(data, min, index);
    }
} //What is its running time?
Insertion Sort

Given a partially sorted set of elements,
Insert the next element in its proper position
Sorting using insertionSort

- **InsertionSort** orders a list of values by repeatedly inserting a particular value into a sorted subset of the list.

- **The algorithm**
  1. Consider the first item to be a sorted sublist of length 1
  2. Insert the second item in the sorted sublist, shifting the first item if needed
  3. Insert the third item into the sorted sublist, shifting the other items as needed
  4. Repeat until all values inserted into their proper positions

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

(3) is sorted sublist. Consider 9.

Shift nothing, **insert 9**. (3, 9) are sorted. Consider 6.

Shift 9, **insert 6**. (3, 6, 9) are sorted. Consider 1.

Shift 3, 6, 9, **insert 1**. (1, 3, 6, 9) sorted. Consider 2.

Shift 3, 6, 9, **insert 2**. (1, 2, 3, 6, 9) sorted.
public static void insertionSort(Comparable[] data) {
    for (int index = 1; index < data.length; index++) {
        Comparable key = data[index];
        int position = index;
        while (position > 0 && data[position - 1].compareTo(key) > 0) {
            // shift larger values to the right
            data[position] = data[position - 1];
            position--;
        }
        data[position] = key;
    }
} // What is its running time?
Merge Sort

Given two partially sorted sets of elements, Merge them into a new sorted set
Sorting using mergeSort

- Merge sort orders a list of values by recursively dividing the list in half until each sublist has one element, then recombining (merging) the sublists.

- The algorithm

**Decomposition Step**
1. divide the list into two (roughly) equal parts
2. recursively divide each part in half, continuing until a part contains only one element

**Merging Step**
1. merge the two parts into one sorted list
2. continue to merge parts as the recursion unfolds
Merge sort – Decomposition step
Merge sort – Merging step

7  8  65  90  110  120  305
mergeSort()

public static void mergeSort(Comparable[] data, int min, int max){
    if (min < max) {
        int mid = (min + max) / 2;
        mergeSort(data, min, mid);
        mergeSort(data, mid+1, max);
        merge(data, min, mid, max);
    }
}

WARNING: This is NOT the exact code executing in the visualization of the previous slide
public static void merge(Comparable[] data, int first, int mid, int last) {
    Comparable[] temp = new Comparable[data.length];

    int first1 = first, last1 = mid;  // endpoints for 1st subarray
    int first2 = mid + 1, last2 = last;  // endpoints for 2nd subarray

    int index = first1;  // next index open in temp array
    // copy smaller item from each subarray into temp
    // until one of the subarrays is exhausted
    while (first1 <= last1 && first2 <= last2) {
        if (data[first1].compareTo(data[first2]) < 0) {
            temp[index] = data[first1];
            first1++;
        } else {
            temp[index] = data[first2];
            first2++;
        }
    }
    index++;
}

merge (1/2)
//copy remaining elements from first subarray, if any
while (first1 <= last1) {
    temp[index] = data[first1];
    first1++;
    index++;
}

//copy remaining elements from second subarray, if any
while (first2 <= last2) {
    temp[index] = data[first2];
    first2++;
    index++;
}

//copy merged data into original array
for (index = first; index <= last; index++) {
    data[index] = temp[index];
}

merge (2/2)
Quick Sort
Select an element at random and place it in its proper location, with smaller elements to its left and larger elements to its right.
Quick sort orders a list of values by partitioning the list around some element (the pivot), then sorting each partition.

The algorithm:
1. Choose one element in the list to be the partition element.
2. Organize the elements so that all elements less than the partition element are to the left and all elements greater than the partition element are to the right.
3. Apply quickSort (recursively) to each partition.

Nice if the partition element divides the list roughly in half.

Quick sort has two methods:
- quickSort – performs recursive algorithm
- partition – rearranges elements into two partitions
Example of Quicksort

Input array

First element becomes the pivot

Pivot is placed to its final position during partition

Two sub-arrays are sorted recursively

First elements become pivots

Pivots are placed in final position

Four sub-arrays are sorted recursively…
public static void quickSort(Comparable[] data, int min, int max) {

    int pivot;

    if(min < max){
        pivot = partition(data, min, max); //make partitions
        quickSort(data, min, pivot-1); //sort left partition
        quickSort(data, pivot+1, max); //sort right partition
    }
}

Example of partition

1. Input array

2. First element becomes the pivot

3. Elements are exchanged to separate smaller from larger

4. Pivot is exchanged to its final position

Partition finished. After that, the two sub-arrays are sorted recursively.
private static int partition(Comparable[] data, int min, int max) {

    Comparable partitionValue = data[min];
    int left = min; int right = max;

    while (left < right) {
        // search for an element that is > the partition element
        while (data[left].compareTo(partitionValue) <= 0 && left < right)
            left++;

        // search for an element that is < the partition element
        while (data[right].compareTo(partitionValue) > 0)
            right--;

        if (left < right)
            swap(data, left, right);
    }

    // move the partition element to its final position
    swap(data, min, right);

    return right; // will become the pivot