Sorting

Sorting

- Arranging a group of items into order.

How to sort?

- What sorting algorithm to use?
  - Many choices!
- What is the efficiency (Big-O)?
- How much extra space is needed?
- Other characteristics to consider

How to order elements?

- All algorithms require some way to compare two elements.
- So what makes one element smaller than or larger than another? What characteristic should you use?
  - Example: Books could be sorted by title, author, subject, publication year, etc.
  - Example: Students could be sorted by name, id, registration date, etc.

Defining Order in Java

- How to objects should be ordered is defined by the compareTo method.
  - This method defines the *natural ordering*
- When a class implements Comparable (and thus has a compareTo method), objects of that class can be sorted.
- Sorting algorithms use the compareTo method to order/sort objects!

Implementing the Comparable Interface

- To implement Comparable, update the class header using generics:

```
public class Student implements Comparable<Student> {
```

Implementing the Comparable Interface

- Implement the compareTo method, using the class as the parameter type

```
public class Student implements Comparable<Student> {

   @Override
   public int compareTo(Student otherStudent) {

   }
```

The compareTo Method

- Returns an integer:
  - Negative: the current element is smaller than the parameter
  - Positive: the current element is bigger than the parameter
  - Zero otherwise

- Be Careful: There is no guarantee of what the actual number is! It is **not** always -1, 1, or 0. You can only rely on it being negative, positive, or 0.
- When invoking compareTo, check if the result is < 0 or > 0 (Not, for example, ==-1)

The compareTo Method

- The compareTo method will contain code to compare the characteristics of the two objects in order to determine which is smaller or larger.
- This often means it will include code like:
  - obj1.variable.compareTo(obj2.variable)
  - Integer.compare(obj1.variable, obj2.variable)
  - obj1.intVariable < or > obj2.intVariable

Implementing the Comparable Interface

```
public class Student implements Comparable<Student> {

   // order by name
   public int compareTo(Student otherStudent) {
      return this.name.compareTo(otherStudent.name);
   }
```

Implementing the Comparable Interface

```
public class Student implements Comparable<Student> {

    // order by name, ignoring capitalization
    public int compareTo(Student otherStudent) {
        return
          this.name.compareToIgnoreCase(otherStudent.name);
    }
```

Implementing the Comparable Interface

```
public class Student implements Comparable<Student> {

    // order by name, then id
   public int compareTo(Student otherStudent) {
      int nameCompare = this.name.compareTo(otherStudent.name);
      if(nameCompare==0) { // names are the same- now order by id
         return Integer.compare(this.id, otherStudent.id);
      } else { // names are different- order by name
         return nameCompare;
      }
    }
```

Polymorphism!

```
List<Student> studentList = new ArrayList<>();
// add students
Collections.sort(studentList);
```

- The sorting method only knows that the list holds elements whose class implements Comparable.
  - Behind the scenes, the compareTo method is invoked to decide how to order/compare objects.
- Polymorphism: At runtime, the correct compareTo method will be invoked for the type stored in the list!

Sorting

- Sorting is very important and useful… but certainly not easy!
- There are many different sorting algorithms.
  - Less efficient: selection, insertion, bubble, shell
  - More efficient: merge sort, quicksort, heapsort
  - Specialty: radix sort

Sorting

- For each sorting algorithm we cover, you should know:
  - The general approach of how it works
  - How to trace an example of the sort
  - The efficiency of the sort
  - Any sort of limitations, unique aspects, best case/worst cases, etc. specific to that sort

SELECTION SORT

Selection Sort

- One of the less efficient sorts
- Relatively easy to write and understand
- Can be written iteratively or recursively

- The general approach of selection sort:
  - **Select** the smallest unsorted value and **swap** it in its final place in the list.
  - Repeat for all values.

Selection Sort

- Find the smallest value in the list
  - **Swap** it with the value in the first position
- Find the 2nd smallest value in the list
  - **Swap** it with the value in the second position
- Find the 3rd smallest value in the list
  - **Swap** it with the value in the third position
- Continue until all values are in their proper place

Selection Sort

- Find the smallest remaining value and swap with the element in the current index.
- Increment that index and repeat.

- Note: The current minimum element is ***swapped*-** values are **not** shifted.
  - Shifting would make this less inefficient algorithm even **less** efficient!!!

Swapping

- Selection Sort relies on *swapping* two values
- *Swapping* requires three assignment statements:

```
temp = first;
first = second;
second = temp;
```

Selection Sort Efficiency

- Selection sort uses nested loops.
  - The outer loop keeps track of the current index that values are swapped into.
  - The inner loop finds the next-smallest value to swap.

– Selection sort is $O(n^2)$.
  - This is the same for any initial ordering of the array.

– Selection sort does not need any additional memory ($O(1)$ for memory).

Selection Sort Examples

- Review the trace and code.

Selection Sort Additional Online Resources

- http://en.wikipedia.org/wiki/Selection_sort#mediaviewer/File:Selection-Sort-Animation.gif
- http://www.youtube.com/watch?v=MZ-ZeQnUL1Q
- http://www.youtube.com/watch?v=6kg9Dx72pzs
- https://www.youtube.com/watch?v=xWBP4lzkoyM

INSERTION SORT

Insertion Sort

- One of the less efficient sorts
- Relatively easy to write and understand
- Can be written iteratively or recursively
  – Recursive insertion sort is good for linked nodes!

- The general approach of insertion sort:
  – Pick the next item to be sorted and **shift** (insert) it into its proper place in an already-sorted sublist.
  – Repeat until all items have been inserted.

Insertion Sort

- Consider the first item to be a sorted sublist (of one item).
- Insert the second item into this sorted sublist, *shifting* the first item as needed to make room to insert the new addition.
- Insert the third item into the sorted sublist of two items, shifting items as necessary
- Repeat until all values are inserted into their proper position

Insertion Sort

- Find where the next in line goes and put it into sorted order.

- Note: Insertion sort does not pay any attention to mins or maxes!
  - Looking for the current min and putting it into place would make this less efficient algorithm even **less** efficient!

Insertion Sort Efficiency

- Insertion sort uses nested loops.
  - The outer loop keeps track of where the border of the sorted sublist is.
  - The inner loop finds the correct position of the next element to be inserted into the sorted sublist.

– Insertion sort is $O(n^2)$.
  - Best case: the array is already sorted- this is $O(n)$

– Insertion sort does not need any additional memory ($O(1)$ for memory).

Insertion Sort Examples

- Review the trace and code.

Insertion Sort Additional Online Resources

- http://en.wikipedia.org/wiki/Insertion_sort#mediaviewer/File:Insertion-sort-example-300px.gif
- https://www.youtube.com/watch?v=c4BRHC7kTaQ
- https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort
- https://www.youtube.com/watch?v=lCDZ0IprFw4

SHELL SORT

Shell Sort

- Somewhere between a less and more efficient sort.
- A variation of insertion sort that leverages the fact that insertion sort works more efficiently when data is “more sorted.”

- The general approach of Shell sort:
  - Look at all elements at a given space from each other. Sort those elements among each other.
  - Reduce the space and repeat again.

Shell Sort Efficiency

- Shell sort uses repeated insertion sorts, but on increasingly sorted data.

- In the worst case, Shell sort is $O(n^2)$.
- However, a simple improvement of making sure the spacing/gap variable is always odd improves this to $O(n^{1.5})$.
  – Shell sort does not need any additional memory ($O(1)$ for memory).

Shell Sort Examples

- Review the trace and code.

Shell Sort Additional Online Resources

- http://www.youtube.com/watch?v=IlRyO9dXsYE
- http://www.youtube.com/watch?v=qzXAVXddcPU

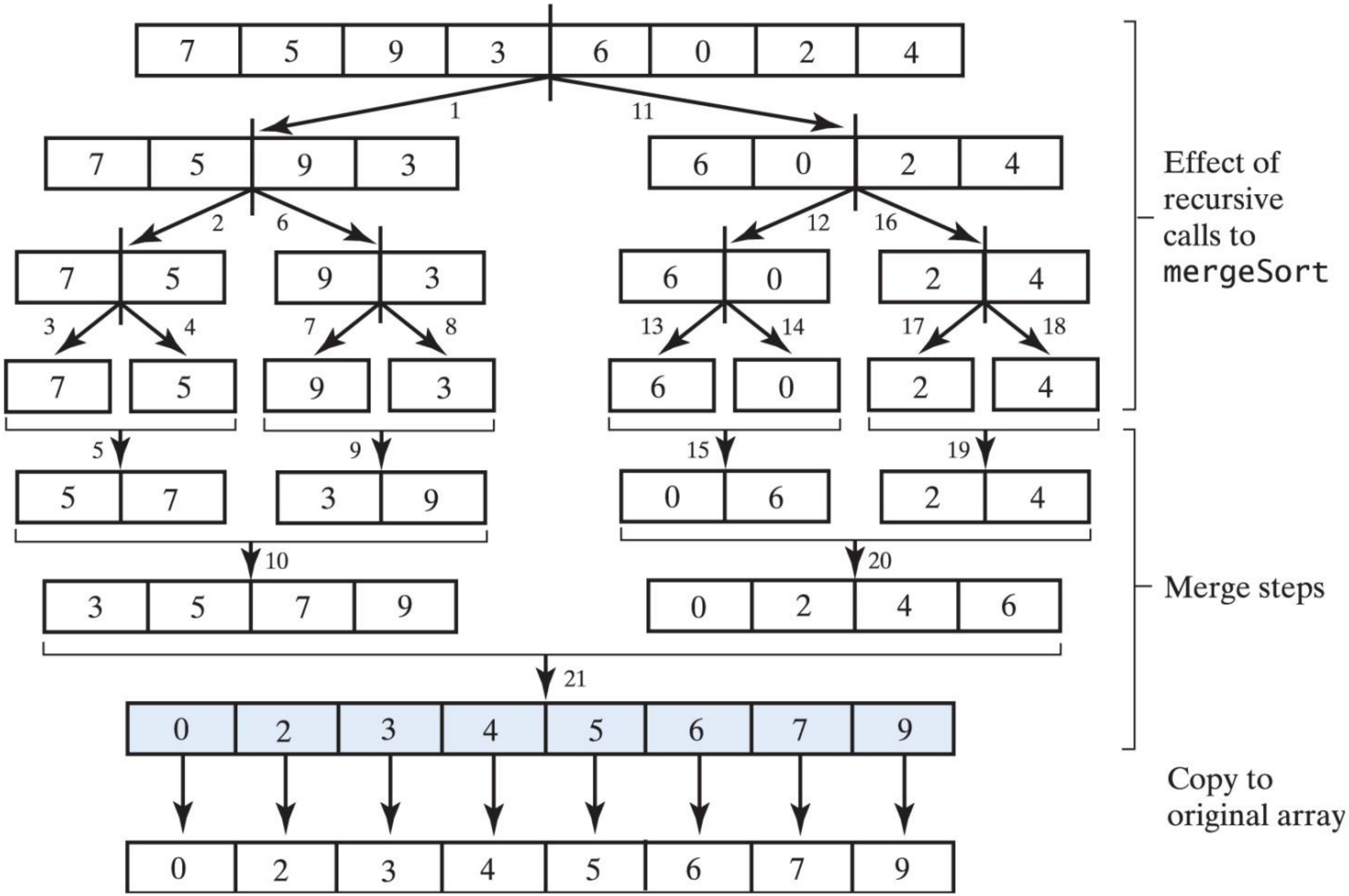
MERGE SORT

Merge Sort

- One of the more efficient sorts
- Uses a *divide and conquer* approach
- Can be written iteratively, but is most often seen written with recursion

- The general approach of Merge Sort:
  - Divide a list into smaller and smaller sublists.
  - Sort that sublists.
  - Merge the sorted sublists back together.

Merge Sort



From Data Structures and Abstractions with Java (Carrano & Henry)

Merge Sort Efficiency

- Merge sort is O(n log n).
- Merge sort needs O(n) extra memory.

MergeSort in the Java Standard Library

- Collections.sort method uses a version of merge sort.
- From the API: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

Merge Sort Examples

- Review the trace and code.

Merge Sort Additional Online Resources

- http://www.youtube.com/watch?v=GCae1WNvnZM
- https://www.youtube.com/watch?v=JSceec-wEyw
- https://www.youtube.com/watch?v=Pr2Jf83_kG0

QUICK SORT

Quick Sort

- One of the more efficient sorts
- Uses a *divide and conquer* approach
- Can be written iteratively, but is most often seen written with recursion

- The general approach of quick sort:
  - Divides the array into two parts (not necessarily halves) and select one element as the pivot.
  - Place that pivot in its final position (elements less than the pivot are to the left and elements greater than the pivot are to the right- but not necessarily sorted).
  - Repeat on each part.

Quick Sort

- Quick sort rearranges the elements in an array during a *partitioning* process.
- After each step in the process, one element (the pivot) is placed in its correct sorted position.

Quick Sort

- Choose a pivot point (or partition value).
- Scan from the right looking for a value that we need to move (a value smaller than the pivot).
  - Stop when we find one.
- Scan from the left looking for a value that we need to move (a value larger than the pivot).
  - Stop when we find one.
- Swap these values.
- Keep looking and repeating.
- Once the scans cross, swap the pivot with the value from the right-side scan.
- The pivot is now in the correct position.
- Repeat recursively on the left and right of the pivot.

Quick Sort Efficiency

- Quick sort is O(n log n) on average.
  - Worst case is $O(n^2)$ (an already sorted dataset)
  - Choice of partition affects efficiency!
- Quick sort does not require additional memory (O(1) for memory).

Quick Sort in the Java Standard Library

- [Arrays.sort method]uses a version of quick sort.
- From the API: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Quick Sort Examples

- Review the trace and code.

Quick Sort Additional Online Resources

- http://www.youtube.com/watch?v=8hHWpuAPBHo
- https://www.youtube.com/watch?v=mN5ib1XasSA
- https://www.youtube.com/watch?v=es2T6KY45cA

SUMMARY

Comparing the Algorithms

	Average Case	Best Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Shell Sort	$O(n^{1.5})$	$O(n)$	$O(n^{1.5})$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Other Resources

- https://www.toptal.com/developers/sorting-algorithms
- https://visualgo.net/bn/sorting
- http://bigocheatsheet.com/

How to choose a sort?

- Many things to consider!

- Efficiency!! (Usually we mean time!)
- Memory requirements (space efficiency)
- The size of the data set
- The structure of the dataset
  - Is it a best case for one of the sorts?
  - Is it an array or linked nodes?

Caution!

- There is **no** algorithm that combines the *swapping* approach of selection sort and the *shifting* approach of insertion sort approaches
  - For example, finding the minimum and then shifting that minimum into place.
- This would essentially be the *worst of both worlds!*
- Selection finds the next minimum and swaps it into place.
- Insertion shifts the next element into its proper place within a sorted subset.
- Don't combine these approaches!

Be Careful!

- Be careful with off-by-one errors in loops and recursive calls.
- Be careful about swaps vs. shifts.
- Pay attention to when an algorithm is using *values* and when it is using *indices.*
- Be careful to always check whether the value returned from compareTo is $<$ or $>$ 0.