Lesson A17

Quadratic Sorting Algorithms

Objectives

- Sorting Template Program
- Bubble Sort
- Selection Sort
- Insertion Sort
- Counting Steps – Quadratic Algorithms
- Animated Sort Simulations
- Sorting Objects

Vocabulary

- BUBBLE SORT
- NONDECREASING ORDER
- SELECTION SORT
- SWAP
- INSERTION SORT
- QUADRATIC
- STUB

Stub

- Incomplete routine that can be called but does nothing
- Stubs are filled in later
- Stub programming is a programming strategy that allows for the coding and testing of algorithms in the context of a working program

Sorting Template Program

- The program shell SortStep.java and SortsTemplate.java have method stubs that will be filled in as we go along.
- The program asks the user to select a sorting algorithm, fills the array with data chosen by the user, calls the sorting algorithms and prints out the sorted data
- The sorting algorithm has been left as a method stub

Sorting algorithms

- 3 sorting algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort

Bubble sort

- Simplest, but slowest
- The largest item bubbles to the end
- **Algorithm**: Move the largest remaining item in the current pass to the end of the data as follows.
  1. Starting with the first two items, swap them if necessary so that the larger item is after the smaller item. Now move over one position in the list and compare to the next item. Again swap the items if necessary.
  2. Remove the largest item most recently found from the data to be searched and perform another pass with this new data at step a.
  3. Repeat steps a and b above until the number of items to be searched is one.

Example – Bubble Sort

To see how Bubble Sort works, let's try an example:

Steps	Data for pass	Sorted data
Start pass 1: compare 4 & 1.	4 1 3 2	
4 > 1 so swapped, now compare 4 & 3.	1 4 3 2	
4 > 3 so swapped, now compare 4 & 2.	1 3 4 2	
4 > 2 so swapped, end of pass.	1 3 2 4	
Start pass 2: compare 1 & 3.	1 3 2	4
3 > 1 so no swap, now compare 3 & 2.	1 3 2	4
3 > 2 so swapped, end of pass.	1 2 3	4
Start pass 3: now compare 1 & 2.	1 2	3 4
2 > 1 so no swap.	1 2	3 4
Only one item in this pass so it is done.	1	2 3 4
Done.		1 2 3 4

Bubble Sort Implementation

```
void bubbleSort(ArrayList <Integer> list){
  for (int outer = 0; outer < list.size() - 1; outer++){
      for (int inner = 0; inner < list.size()-outer-1; inner++){
        if (list.get(inner) > list.get(inner + 1)){
          //swap list[inner] & list[inner+1]
          int temp = list.get(inner);
          list.set(inner, list.get(inner + 1));
          list.set(inner + 1, temp);
        }
      }
    }
  }
```

- After the first pass (outer = 0), the largest value will be in its final resting place. When outer = 1, the inner loop goes from 0 to 1 because a comparison between positions 2 and 3 is unnecessary. The inner loop is shrinking.

Try it– Bubble Sort

Outer	57	95	88	14	25	6
0	57	88	14	25	6	95
1	57	14	25	6	88	95
2	14	25	6	57	88	95
3	14	6	25	57	88	95
4	6	14	25	57	88	95

Selection Sort

- Also makes several passes through the list
- Finds the smallest item on each pass
- At the end of a pass, the smallest item found is swapped with the last remaining item for that pass
- Thus swapping occurs only once for each pass, thus making this algorithm more efficient than bubble sort

Selection Sort Implementation

```
void selectionSort(ArrayList <Integer> list){
    int min, temp;

    for (int outer = 0; outer < list.size() - 1; outer++){
      min = outer;
      for (int inner = outer + 1; inner < list.size(); inner++){
        if (list.get(inner) < list.get(min)) {
          min = inner; // a new smallest item is found
        }
      }
      //swap list[outer] & list[min]
      temp = list.get(outer);
      list.set(outer, list.get(min));
      list.set(min, temp);
    }
  }
```

Try it: Selection Sort

Outer	57	95	88	14	25	6
0	6	95	88	14	25	57
1	6	14	88	95	25	57
2	6	14	25	95	88	57
3	6	14	25	57	88	95
4	6	14	25	57	88	95

Insertion Sort

- Insertion Sort takes advantage of the following fact.

  If A < B and B < C, then it follows that A < C. We can skip the comparison of A and C.
- Consider the following partially sorted list of numbers.
- 2 5 8 3 9 7
- The first three values of the list are sorted. The 4th value in the list, (3), needs to move back in the list between the 2 and 5.

- This involves two tasks, finding the correct insert point and a right shift of any values between the start and insertion point.

Insertion Sort Implementation

```
void insertionSort(ArrayList <Integer> list){
    for (int outer = 1; outer < list.size(); outer++){
      int position = outer;
      int key = list.get(position);

      // Shift larger values to the right
      while (position > 0 && list.get(position-1) > key){
        list.set(position, list.get(position - 1));
        position--;
      }
      list.set(position, key);
    }
  }
```

- For each pass of outer, the algorithm finds the location where list[outer] must be inserted and it does a right shift on sections of the array to make room for the inserted value

Try it – Insertion Sort

Outer	57	95	88	14	25	6
1	57	95	88	14	25	6
2	57	88	95	14	25	6
3	14	57	88	95	25	6
4	14	25	57	88	95	6
5	6	14	25	57	88	95

Quadratic Algorithms

- These three sorting algorithms are categorized as quadratic sorts because the number of steps increases as a quadratic equation of the size of the list
- It will be very helpful to study algorithms based on the number of steps they require to solve a problem.
- We will add code to the sorting template program and count the number of steps for each algorithm.
- This will require the use of an instance variable – we'll call it steps.
- The steps variable will be maintained within the sorting class and be accessed through appropriate accessor and modifier methods.
- You will need to initialize steps to 0 at the appropriate spot in the main menu method.
- For our purposes, we will only count comparisons of items in the list, and gets or sets within the list. These operations are typically the most expensive (time-wise) operations in a sort.

Example to count steps – Bubble Sort

```
public void bubbleSort(ArrayList <Comparable> list){
    steps = 0;
    for (int outer = 0; outer < list.size() - 1; outer++){
      for (int inner = 0; inner < list.size()-outer-1; inner++){
          steps += 3;//count 1 compare and 2 gets
          if (list.get(inner).compareTo(list.get(inner+1)) > 0){
             steps += 4;//count 2 gets and 2 sets
             Comparable temp = list.get(inner);
             list.set(inner,list.get(inner + 1));
             list.set(inner + 1,temp);
          }
      }
    }
  }
```

- As the size of data doubles, the number of steps increases approximately four times, a quadratic rate

Sorting Objects

- What if you wanted to sort Strings instead of integers?
  - You cannot use $<$ on Strings
  - You have to use compareTo method

Bubble Sort for Sorting Strings

```
void bubbleSort(ArrayList <String> list){
    for (int outer = 0; outer < list.length - 1; outer++){
      for (int inner = 0; inner < list.size()-outer-1; inner++){
        if (list.get(inner).compareTo(list.get(inner + 1) > 0){
            //swap list[inner] & list[inner+1]
            String temp = list.get(inner);
            list.set(inner, list.get(inner + 1));
            list.set(inner + 1, temp);
        }
      }
    }
  }
```

Comparable Interface

- If I am able to sort my data, there must be an order defined for it.
- Classes that have an order should have a compareTo method.
- Java defines an Interface, Comparable, just for this purpose.
- To make a BubbleSort that will work on any objects that implement Comparable, make the following changes, again highlighted.

Bubble Sort to sort any object

```
void bubbleSort(ArrayList <Comparable> list){
    for (int outer = 0; outer < list.length - 1;
  outer++){
      for (int inner = 0; inner < list.size()-outer-
  1; inner++){
        if (list.get(inner).compareTo(list.get(inner
  + 1) > 0){
            //swap list[inner] & list[inner+1]
            Comparable temp = list.get(inner);
            list.set(inner, list.get(inner + 1));
            list.set(inner + 1, temp);
        }
      }
    }
  }
```

Comparable Interface, cont'd

- When designing objects, consider whether or not it makes sense to compare objects that you build. If it does, implement the Comparable Interface and provide the equals method for your class.