# Student LESSON

18 – Quadratic Sorting Algorithms

**INTRODUCTION:** In this lesson, you will learn about three sorting algorithms: bubble, selection, and insertion. You are responsible for knowing how they work, but you do not necessarily need to memorize and reproduce the code. After counting the number of steps of each algorithm, you will have a sense of the relative speeds of these three sorts.

The key topics for this lesson are:

A. Sorting Template Program

B. Bubble Sort

C. Selection Sort

D. Insertion Sort

E. Counting Steps - Quadratic Algorithms

1. Animated Sort Simulations
2. Sorting Objects

**VOCABULARY:** BUBBLE SORT INSERTION SORT

NONDECREASING ORDER QUADRATIC

SELECTION SORT STUB

SWAP

**DISCUSSION:** A. Sorting Template Program

|  |  |
| --- | --- |
| **See *SortStep.java*** and *SortsTemplate. java* | 1. A program shell has been provided in the curriculum as *SortStep.java* (the main test method), and *SortsTemplate.java* (the sort class template). |

2. The program asks the user to select a sorting algorithm, fills the array with an amount of data chosen by the user, calls the sorting algorithm, and prints out the data after it has been sorted.

3. At this point, each sorting algorithm has been left as a method stub. A stub is an incomplete routine that can be called but does not do anything yet. The stub will be filled in later as each algorithm is developed and understood.

4. Stub programming is a programming strategy. It allows for the coding and testing of algorithms in the context of a working program. As each sorting algorithm is completed, it can be added to the program shell and tested without having to complete the other sections.

5. This stepwise development of programs using stub programming will be used extensively in future lessons.

B. Bubble Sort

1. Bubble Sort is the simplest of the three sorting algorithms, and also the slowest. The Bubble Sort gets its name from the way that the largest items “bubble” to the top (end). The procedure goes like this.

a. Move the largest remaining item in the current pass to the end of the data as follows. 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.

b. Remove the largest item most recently found from the data to be searched and perform another pass with this new data at step a.

c. Repeat steps a and b above until the number of items to be searched is one.

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 |

2. The following program implements the Bubble Sort algorithm.

**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);

}

}

}

}

3. Given a list of 6 values, the loop variables outer and inner will evaluate as follows.

|  |  |
| --- | --- |
| **When outer =** | **inner ranges from 0 to < (6 – outer – 1)** |
| 0 | 0 to 4 |
| 1 | 0 to 3 |
| 2 | 0 to 2 |
| 3 | 0 to 1 |
| 4 | 0 to 0 |

4. When outer = 0, then the inner loop will do 5 comparisons of pairs of values. As inner ranges from 0 to 4, it does the following comparisons:

|  |  |
| --- | --- |
| **inner** | **if (list.get(inner) > list.get(inner + 1))** |
| 0 | **if** list[0] > list[1] |
| 1 | **if** list[1] > list[2] |
| … | … |
| 4 | **if** list[4] > list[5] |

5. If (list.get(inner) > list.get(inner + 1)) is **true**, then the values are out of order and a swap takes place. The swap takes three lines of code and uses a temporary variable temp.

6. After the first pass (outer = 0), the largest value will be in its final resting place (and may it rest in peace). When outer = 1, the inner loop only goes from 0 to 3 because a comparison between positions 4 and 5 is unnecessary. The inner loop is shrinking.

7. Because of the presence of duplicate values, this algorithm will result in a list sorted in non-decreasing order.

8. Here is a small list of data to test your understanding of Bubble Sort. Write in the correct sequence of integers after each advance of outer. (Answers are found in Handout A17.1, *Sorting Answers*.)

*outer* 57 95 88 14 25 6

1 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

2 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

3 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

4 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

5 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

C. Selection Sort

1. The Selection Sort also makes several passes through the list. On each pass, it compares each remaining item to the smallest (or largest) item that has been found so far in the pass. In the example below, the Selection Sort method 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 only occurs once for each pass. Reducing the number of swaps makes the algorithm more efficient.

2. The logic of Selection Sort is similar to Bubble Sort except that fewer swaps are executed.

**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);

}

}

3. Again, assuming that we have a list of 6 numbers, the outer loop will range from 1 to 5. When outer = 1, we will look for the smallest value in the list and move it to the first position in the array.

4. However, when looking for the smallest value to place in position 1, we will not swap as we search through the list. The algorithm will check from indexes 1 to 5, keeping track of where the smallest value is found by saving the index of the smallest value in min. After we have found the location of the smallest value, we swap list[outer] and list[min].

5. By keeping track of where the smallest value is located and swapping only once, we have a more efficient algorithm than Bubble Sort.

6. Here is a small list of numbers to test your understanding of Selection Sort. Fill in the correct numbers for each line after the execution of the outer loop. (Answers are found in Handout A17.1, *Sorting Answers*.)

outer 57 95 88 14 25 6

1 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

2 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

3 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

4 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

5 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

D. Insertion Sort

1. 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.

2. 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.

insertFig

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

3. The code follows.

**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);

}

}

4. By default, a list of one number is already sorted. Hence the outer loop skips position 0 and ranges from positions 1 to list.size(). For the sake of discussion, let us assume a list of 6 numbers.

5. For each pass of outer, the algorithm will determine two things concerning the value stored in list[outer]. First, it finds the location where list[outer] needs to be inserted in the list. Second, it does a right shift on sections of the array to make room for the inserted value if necessary.

6. Constructing the inner **while** loop is an appropriate place to apply DeMorgan's laws:

a. The inner while loop postcondition has two possibilities:

The value (key) is larger than its left neighbor.

The value (key) moves all the way back to position 0.

b. This can be summarized as:

(0 == position || list.get(position – 1) <= key)

c. If we negate the loop postcondition, we get the while loop boundary condition:

(0 != position && list.get(position – 1) > key)

d. This can also be rewritten as:

((position > 0) && (list.get(position – 1) > key))

7. The two halves of the boundary condition cover these situations:

(position > 0) –> we are still within the list, keep processing

list[position - 1] > key –> the value in list[pos-1] is larger than key, keep moving left (position--) to find the first value smaller than key.

8. The Insertion Sort algorithm is appropriate when a list of data is kept in sorted order with infrequent changes. If a new piece of data is added, probably at the end of the list, it will get quickly inserted into the correct position in the list. Many of the other values in the list do not move and the inner **while** loop will not be used except when inserting a new value into the list.

9. Here is the same list of six integers to practice Insertion Sort. (Answers are found in Handout A17.1, *Sorting Answers*.)

outer 57 95 88 14 25 6

2 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

3 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

4 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

5 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

6 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

E. Counting Steps - Quadratic Algorithms

1. These three sorting algorithms are categorized as quadratic sorts because the number of steps increases as a quadratic function of the size of the list.

2. 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.

3. 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.

1. 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.

5. As you type in the sorting algorithms, add increment statements for the instance variable steps. For example, here is a revised version of the bubbleSort method:

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 one 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);

}

}

}

}

6. It is helpful to remember that a **for** statement is simply a compressed **while** statement. Each **for** loop has three parts: initialization, boundary check, and incrementation.

7. As you count the number of steps, an interesting result will show up in your data. As the size of the data set doubles, the number of steps executed increases by approximately four times, a “quadratic” rate.

8. Bubble Sort is an example of a quadratic algorithm in which the number of steps required increases at a quadratic rate as the size of the data set increases.

9. A quadratic equation in algebra is one with a squared term, like x2. In our sorting example, as the size of the array increases to N, the number of steps required for any of the quadratic sorts increases as a function of N2.

F. Animated Sort Simulations

1. The web site titled "The Complete Collection of Algorithm Animations (CCAA)" located at <http://www.cs.hope.edu/~alganim/ccaa/sorting.html>, provides a wide variety of animated sorting simulations.
   1. Sorting Objects

1. Notice that the sorts we developed above know how to compare Integers. Comparison is built into the Integer class. What if we wanted to write a sort that could work on Strings? You cannot use ‘<’ on Strings. Remember you have to use the compareTo method.

1. To convert the BubbleSort, make the following changes that are highlighted in yellow.

**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);

}

}

}

}

3. 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 (see below for some information on Comparable). To make a BubbleSort that will work on any objects that implement Comparable, make the following changes, again highlighted in yellow.

**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);

}

}

}

Now this method is quite reusable because we can use it to sort any Comparable object. The compareTo interface follows.

Interface java.lang.Comparable

int compareTo(Object other)

// returns value < 0 if this is less than other

// returns value = 0 if this is equal to other

// returns value > 0 if this is greater than other

Remember to consider whether or not it makes sense to compare objects that you build. If it does, implement the Comparable Interface. It would also make sense to provide an equals method for your class.

|  |  |
| --- | --- |
| **SUMMARY/ REVIEW:** | Sorting data is one of the best applications of computers and software. What takes hours or days by hand can be sorted in seconds or minutes by a computer. However, these quadratic algorithms have problems sorting large amounts of data. More efficient sorting algorithms will be covered in later lessons. |