DESCRIBES THE EFFICIENCY

NEXT FEW WEEKS

STUDENT OUTLINE

Lesson 22 – Quadratic Sorting Algorithms

INTRODUCTION:

MEASURING EFFICIENCY

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 relative sense of which is the fastest and the slowest sorting algorithm.

For a visual comparison of the sort algorithms described in this lesson, see the menu page in the html version of this lesson.

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
F. Animated Sort Simulations

VOCABULARY:

BUBBLE SORT
SELECTION SORT
QUADRATIC
NONDECREASING ORDER
SWAP
INSERTION SORT
STUB

DISCUSSION:

A. Sorting Template Program

DRIVER

DOES ALL THIS

See *SortStep.java.* and *Sorts.java*

CARRIES FORWARD OVER NEXT FEW WEEKS

1. A program shell has been provided as *SortStep.java* (the main test method), and *Sorts.java* (the sort class template). **These shells will be used for labs in Lessons 22 - 25.**

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 gives an option of printing 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 methodology strategy used during the implementation stage of program development. 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.

"STONES SINK FASTER THAN BUBBLES RISE"

B. Bubble Sort

1. Bubble Sort is the simplest algorithm for sorting a set of data, and also the slowest. The procedure goes like this:

   a. Find the largest item and place it at the end of the items that were searched.
   b. Remove the largest item most recently found from the set to be searched and go to step a.
   c. (Repeat parts a. and b. until the number of items to be searched is zero.)

   The Bubble Sort algorithm uses the simplest method for determining the largest item in a list: it compares two items at a time and then moves on to compare the larger of the two with the next item in the list. After each comparison, the larger item must be identified and "swapped" into position to be compared with the next item.

2. The following program implements the Bubble Sort algorithm.

CONTROLS # OF TRAVERSALS

ACTUALLY PERFORMS A PASS

VALUE OF WHERE I AM

MAKES A BIT MORE EFFICIENT

SWAP

VALUE OF NEXT DOOR NEIGHBOR

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

3. Given an array of 6 values, the loop variables `outer` and `inner` will evaluate as follows:

MEANS EACH SUCCESSIVE PASS IS ONE VISIT SHORTER

ALTERATION (c)

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:

1ST PASS

`inner`	`if list[inner] > list[inner + 1]`
0	`if list[0] > list[1]`
1	`if list[1] > list[2]`
...	...
4	`if list[4] > list[5]`

5. If `(list[inner] > list[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. 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.

NO DUPLICATES, IT'S IN ASCENDING ORDER

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`.

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

= FINAL LOCATION

C. Selection Sort

1. The Selection Sort increases efficiency over the Bubble Sort by making the comparison between an item and the smallest or largest that has been found so far in the passage through the items. (In our example below our Selection Sort procedure identifies the smaller item found on each pass through the list.) Thus swapping only occurs once each 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.

CONTROLS # OF PASSES

PERFORMS 1 PASS

THE PLACE I VISIT

KEY VALUE FOUND SO FAR

```
void selectionSort(int[] list)
{
  int min, temp;

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

SWAP

3. Selection Sort also uses passes to sort for a position in the array. Again, assuming 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 the entire list. The algorithm will check from positions 2 to 6, keeping track of where the smallest value is found by comparing with `list[min]`. After we have found the location of the smallest value in index position `min`, we swap `list[outer]` and `list[min]`.

5. By keeping track of where the smallest value is located and swapping 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.

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

= = FINAL LOCATION

D. Insertion Sort

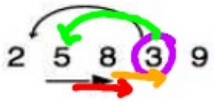
1. Insertion Sort takes advantage of this logic:

   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.



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

3. Following is the code:

```
void insertionSort(int[] list)
{
  for (int outer = 1; outer < list.length; outer++)
  {
    int position = outer;
    int key = list[position];

    // Shift larger values to the right
    while (position > 0 && list[position - 1] > key)
    {
      list[position] = list[position - 1];
      position--;
    }
    list[position] = key;
  }
}
```

MAKES 1 TRAVERSAL

INITIALIZE TO "LOOK BACK"

STOPS AT BEGINNING

INSERT VALUE TO CORRECT LOCATION

LOOKING FOR THINGS THAT ARE OUT OF ORDER

RIGHT-SHIFT TO FIND INSERT POINT

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

5. For each pass of `outer`, the algorithm will solve two things concerning the value stored in `list[outer]`. 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.

WE DO THIS AS WE LOOK FOR THAT

WHAT MUST BE TRUE WHEN THE LOOP DONE

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[position - 1] <= key)
   ```

APPLY DEMORGAN

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

   ```
   (0 != position && list[position - 1] > key)
   ```

   d. This can also be rewritten as:

   ```
   ((position > 0) && (list[position - 1] > key))
   ```

THE LOOP BOUNDARY TO ALLOW LOOP TO CONTINUE!!

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

   `(position > 0)` – we are still on 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 very few 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. 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.

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

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.

4. 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:

```
void bubbleSort (int[] list)
{
  steps++;      // initialization of outer
  for (int outer = 0; outer < list.length-1; outer++)
  {
    steps += 3;
    //  1 - boundary check of outer loop;
    //  2 - increment, outer++
    //  3 - initialization of inner loop
    for (int inner = 0; inner < list.length-outer-1; inner++)
    {
      steps += 3;
      //  1 - boundary check of inner loop
      //  2 - increment, inner++
      //  3 - if comparison
      if (list[inner] > list[inner+1])
      {
        int temp = list[inner]
        list[inner] = list[inner + 1]);
        list[inner + 1] = temp;
        steps += 3;     // swap of list[inner] & list[inner + 1]
      }
    }
  }
}
```

5. 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.

6. 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.

7. 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.

8. A quadratic equation in algebra is one with a squared term, like x^2. In our sorting example, as the size of the array increases by a factor of N, the number of steps required for any of the quadratic sorts increases by a factor of N^2.

F. Animated Sort Simulations

1. The web site http://visualgo.net/sorting, provides a wide variety of animated simulations.

SUMMARY/ REVIEW: Sorting data by the computer 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.

ASSIGNMENT: Lab Exercise L.A.22.1, *Quadratics*