OzoBlockly Bubble Sort

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

Introduction

The release of the Master Mode 5 of OzoBlockly contains many powerful programming features, one of which is the inclusion of *arrays*—a group of sequential memory locations for storing similar information. The ability to sort the values in an array in either ascending or descending order is a topic generally covered in introductory computer science courses. One of the most commonly discussed algorithms is the bubble sort. Though by no means the most efficient algorithm, it is an algorithm that is easy to understand for the beginning programmer. In this lesson, the student interactively inputs numbers into an array and specifies if the sort is to be ascending or descending. After sorting the array, Evo then speaks the number of comparisons and number of swaps as measures of efficiency, displays the largest and smallest elements in the array, and speaks the numbers in the sorted array. This is accomplished via a provided OzoBlockly program and Ozomap. See Figure 1 for a small copy of the Ozomap. A full page version of the Ozomap, for use by students with Evo, can be found on the last page of this document.

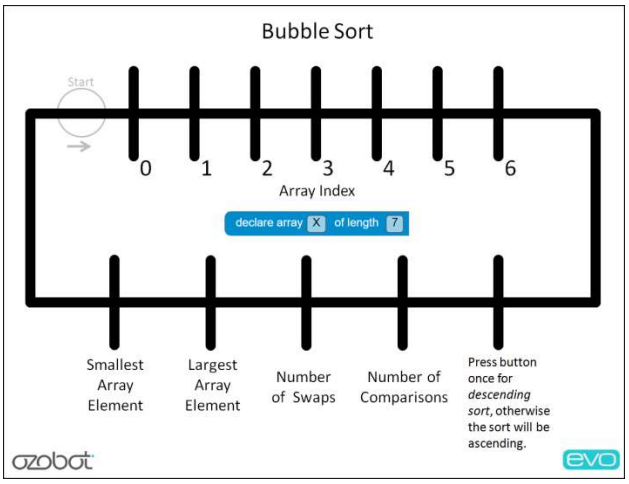


Figure 1

If your students are beginning the study of computer science, they might find it helpful to first look at the lesson entitled ***OzoBlockly Array Primer*** and found in the online OzoBlockly Lesson Library. For the more advance computer science students, there are a number of other OzoBlockly lessons making use of arrays including:

- Ozobot Bit/Evo Lesson: Repeat a Saved Path Using an Array
- Ozobot Bit/Evo Lesson: Remembering Bridge Crossing Colors
- OzoBlockly Lesson: Bit and Evo Deal Poker Hands

The Bubble Sort Algorithm

There are many sorting algorithms, with the most well-known including bubble sort, insertion sort, selection sort, merge sort, shell sort, and quick sort. As previously mentioned, bubble sort is one of the easiest for the beginning student to understand. For this reason, bubble sort has been selected for Evo's OzoBlockly sorting program.

The bubble sort passes through the array that is to be sorted, comparing each pair of adjacent values, swapping them in the event that they are not in the correct order. Passes through the array are repeated until no swaps are required for a pass, at which time the array has been sorted.

Figure 2 shows the step-by-step details for an ascending bubble sort with six elements. Values shown in red indicate the pair being compared. The number of comparisons in the first pass is always one less than the length of the array. Also it should be noted that with each pass, one less comparison is required. This is because there is no need to compare items that have already "bubbled" to the top of the array. The number of swaps in a given pass through the array can vary significantly.

Ascending Bubble Sort with Comparisons Shown in Red

PASS 1 *(5 comparisons)*

7 2 5 3 6 4	Swap since 7 > 2
2 7 5 3 6 4	Swap since 7 > 5
2 5 7 3 6 4	Swap since 7 > 3
2 5 3 7 6 4	Swap since 7 > 6
2 5 3 6 7 4	Swap since 7 > 4
2 5 3 6 4 7	Largest value now in its place

PASS 2 *(4 comparisons)*

2 5 3 6 4 7	No swap since 5 > 2
2 5 3 6 4 7	Swap since 5 > 3
2 3 5 6 4 7	No swap since 6 > 5
2 3 5 6 4 7	Swap since 6 > 4
2 3 5 4 6 7	2nd largest value is now in its place

PASS 3 *(3 comparisons)*

2 3 5 4 6 7	No swap since 3 > 2
2 3 5 4 6 7	No swap since 5 > 3
2 3 5 4 6 7	Swap since 5 > 4
2 3 4 5 6 7	3rd largest value is now in its place

*The array is now sorted, but the algorithm requires a **complete** pass with no swaps to know this.*

PASS 4 *(2 comparisons)*

2 3 4 5 6 7	No swap since 3 > 2
2 3 4 5 6 7	No swap since 4 > 3
2 3 4 5 6 7	4th largest value is now in its place

Since there were no swaps in pass 4, sorting is complete.

Figure 2

The OzoBlockly Bubble Sort Subroutine

Figure 3 contains the OzoBlockly subroutine to sort an array X in this lesson. Since we want to collect some algorithm efficiency statistics, variables that will count swaps and comparisons are first initialized to zero. The outer loop lets the number of comparisons vary from the *length of the array minus 1* to 1 by steps of -1. (We observed this pattern while studying the example of Figure 2.) Before starting the inner loop, the number of swaps for the current pass is set to 0. The inner loop varies a counter control variable *i* from the lowest index in the array to one less than the outer loop's current number of comparisons. We then get the two values currently being compared and store them into variables called *left* and *right*. Since we are about to do a comparison, the total comparison count is incremented by 1. Next there are some nested *if* blocks that look at whether the desired ordering is ascending or descending, and then swap the left and right values if necessary. These *if* blocks also increment the total swap count and swaps for the current pass by 1 in the event a swap is made. Finally, when an iteration of the inner loop is complete, we check to see if there were any swaps for the pass. If not we break out of the outer loop as sorting is complete.

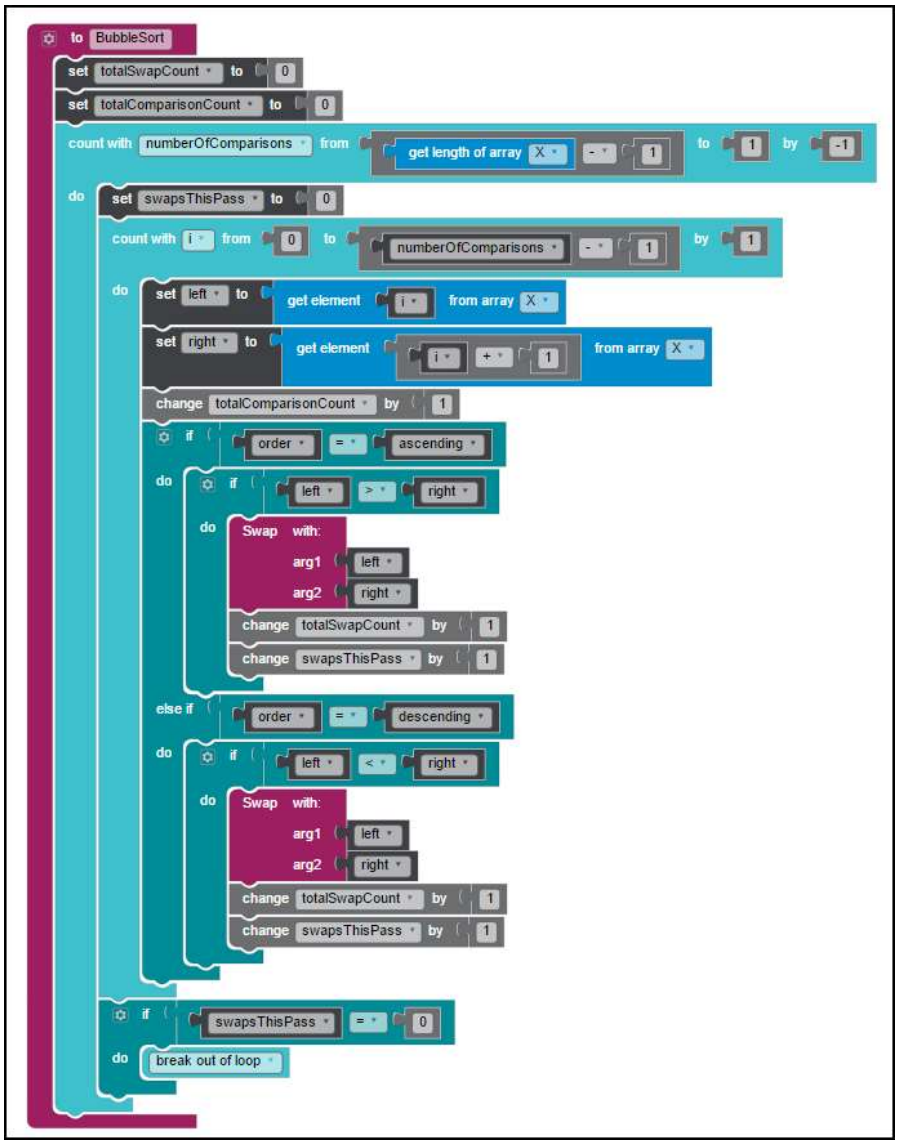


Figure 3

Working with the Evo Bubble Sort Program

1. Open the provided *Bubble Sort.ozocode* file. Then load the program onto Evo and use a printed copy of the Ozomap on the last page of this document with Evo.
2. To run the loaded program on Evo, begin with Evo powered down. Place Evo centered on the gray circle on the Ozomap and facing the direction shown by the gray arrow.
3. Press Evo's button once to turn Evo on. Then when the front lights show light blue, quickly double-press the button. The OzoBlockly program has started running.
4. Evo will then move to the intersection for X[0] and the top LED will turn red. You then have three seconds to either press Evo's button or not press the button. Pressing the button means that you want to enter a negative number. Not pressing the button means that you want to enter a positive number. After the 3 seconds with a red LED have elapsed, Evo's top light will turn green. You then have 10 seconds to push the button as many times as you want. The number of pushes is the absolute value of the number you want for the value of X[0]. If you don't press the button at all while the LED is green, then you want the value for X[0] to be 0.
5. After the 10 seconds of the green LED has elapsed, Evo will speak the value entered and then will move to the intersection for X[1]. In the same manner as discussed in step 4, enter the desired value for X [1].
6. Enter the remaining values for X[2] through X[6].
7. Evo will then move to the intersection for the sort order and stop. The top LED will be green for four seconds. If you want the sort to be ascending, do nothing. If you want the sort to be descending, press Evo's button once before the four second interval has elapsed.
8. Evo will then move to the next three intersections, stopping at each. Evo will speak the comparisons and swaps statistics, and then speak the values for the largest and smallest array elements.
9. Evo will loop back to the top of the Ozomap, stop at each intersection, and speak the sorted values for X[0] through X[6].
10. Finally, Evo will power down after speaking the value for X[6].

Exercises

1. Consider the following array X of seven values:

   7 2 8 5 3 6 4

   Run the OzoBlockly Bubble Sort program with Evo and the provided Ozomap to sort this array in ***ascending*** order.

   a. How many comparisons and swaps are required for this array sort?
   b. What are the indexes of the largest and the smallest array elements following the sort?
   c. What is the value of X[3] following the sort?

2. You can also have input values of 0 as well as negative integers with the OzoBlockly bubble sort. Consider the following array X of seven values:

   5 -2 -1 0 7 -4 3

   Run the OzoBlockly Bubble Sort program with Evo and the provided Ozomap to sort this array in *descending* order.

   a. How many comparisons and swaps are required for this array sort?
   b. What are the indexes of the largest and the smallest array elements following the sort?
   c. What is the value of X[4] following the sort?

3. Consider the following array X of seven values:

   8 6 3 0 -1 -4 -7

   a. What do you predict regarding the number of comparisons and number of swaps if this array is sorted in ***descending*** order? Justify your prediction based upon how the bubble sort algorithm works, as described on page 2.
   b. Run the OzoBlockly Bubble Sort program with Evo and the provided Ozomap to sort this array in ***descending*** order. How do your predictions in part **a** compare to the actual statistics that you obtained in part **b**? Explain any discrepancies.
   c. What do you predict regarding the number of comparisons and number of swaps if this array is sorted in ***ascending*** order? Justify your prediction based upon how the bubble sort algorithm works, as described on page 2.
   d. Run the OzoBlockly Bubble Sort program with Evo and the provided Ozomap to sort this array in ***ascending*** order. How do your predictions in part **c** compare to the actual statistics that you obtained in part **d**? Explain any discrepancies.

4. An array of values that needs to be sorted can have duplicate entries as in the following array:

   4 3 9 3 9 4 3

   a. Run the OzoBlockly Bubble Sort program with Evo and the provided Ozomap to sort this array in ***ascending*** order. How many comparisons and swap are required to sort this array?

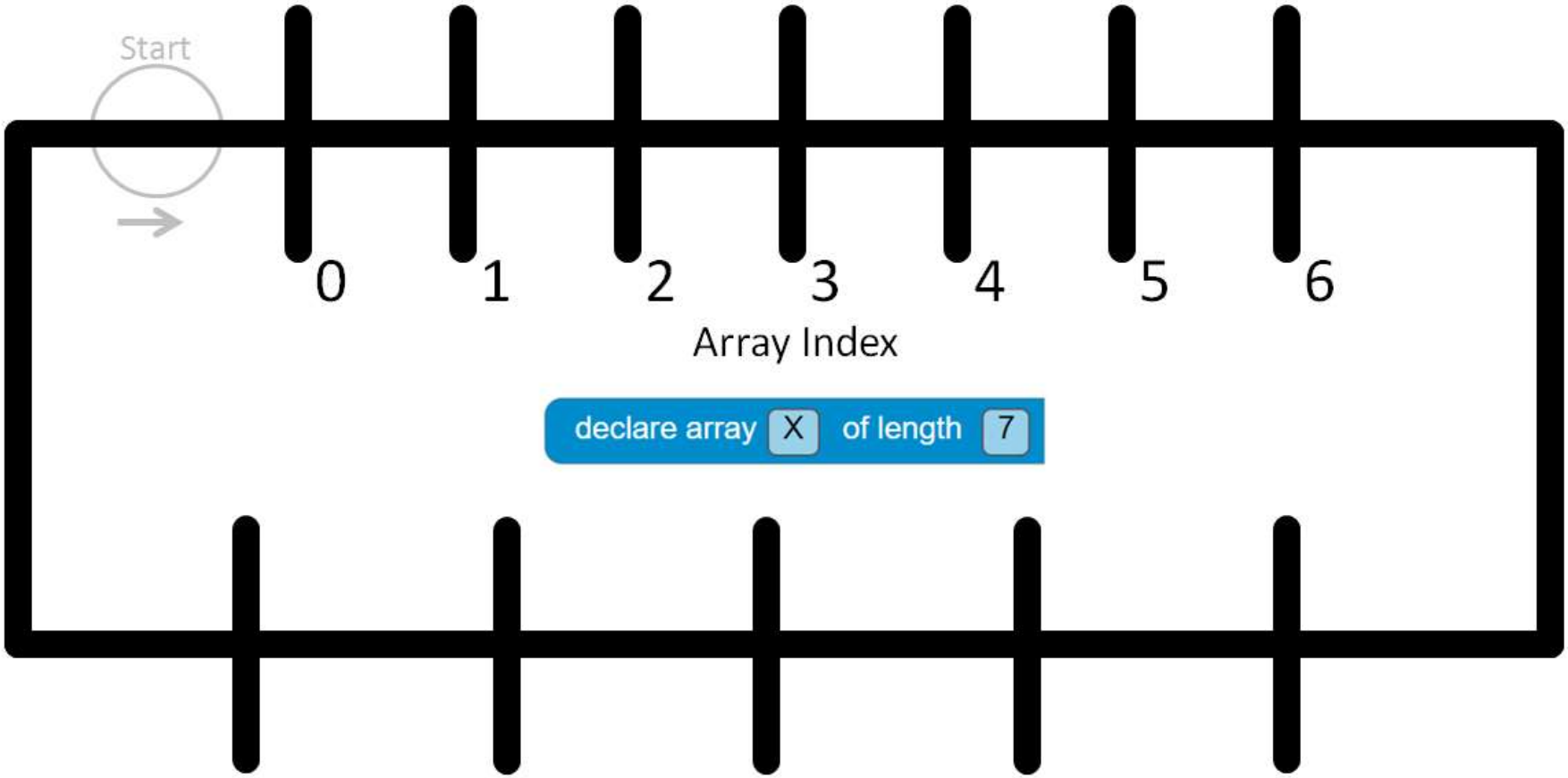
b. Study the OzoBlockly subroutine in Figure 3. How does the program handle swapping when comparing two values that are the same? Does this appear to help explain the smaller number of swaps required to bubble sort this array.

5. Exercise 3a and 3c deal with what are known as ***best*** and ***worst case*** scenarios for the bubble sort algorithm efficiency, respectively. In the best case, the values are already sorted correctly at the start. In the worst case, the values are in completely reversed order at the start. (When considering the following questions, note that mathematics tells us that the sum of the first *n* positive integers is n(n + 1)/2.)
    a. Suppose that you have an array of length 100. In a best case scenario, how many comparisons and how many swaps would be required? Explain.
    b. Suppose that you have an array of length 100. In a worst case scenario, how many comparisons and how many swaps would be required? Explain.
    c. Suppose that you have an array of length 1000. In a best case scenario, how many comparisons and how many swaps would be required? Explain.
    d. Suppose that you have an array of length 1000. In a worst case scenario, how many comparisons and how many swaps would be required? Explain.
    e. An algorithm's ***time complexity*** relates the length of its input to the number of steps it takes. In our bubble sort we have two measures of time complexity—number of comparisons and number of switches, as both of these consume valuable time. The ***order of complexity*** of an algorithm relates to the limiting behavior when the input ***n*** *tends toward infinity*. Some common orders for algorithms are described as O(log n), O(n), O(n log n), $O(n^2)$, $O(2^n)$, and O(n!)—from very good to terrible.
        i. Which of the above **O's** describes the best case scenario for the bubble sort comparisons? Explain.
        ii. Which of the above **O's** describes the worst case scenario for the bubble sort comparisons? Explain.

    We see that the use of a bubble sort should probably be avoided when the size of the array is large and quite unordered at the start.

6. Answer the following questions regarding the bubble sort OzoBlockly program:
    a. What change(s) would be required in the program if the array had five elements rather than seven?
    b. The OzoBlockly bubble sort shown in Figure 3 calls a user-defined subroutine called *Swap*. What is the purpose of the variable *temp* in that subroutine?

Bubble Sort


Start
0
1
2
3
4
5
6
Array Index
declare array X of length 7
Smallest Array Element
Largest Array Element
Number of Swaps
Number of Comparisons
Press button once for *descending sort*, otherwise the sort will be ascending.