Vincent Chee
26/02/15
Professor Kristina Striegnitz
Data Structures

Project 7 – Analyzing Time Complexity

Part 1: Source Code and Algorithm Analysis

a. The big-O notation in terms of the number of input is $O(n)$There is a for-loop that iterates over a third of the array (because j=j+3), but it still iterates through the entire array.

b. Big-O notation in terms of number of input for findMax is $O(n)$ because you have to iterate through the entire array.

c. Big-O notation in terms of the number of input for three is $O(n \log n)$, the do-while part of this method is $O(n)$, but it's not very important cause the nested for loop section is the big-O notation which we actually need to analyze. The outer for-loop has big-O notation $O(\log n)$ because the array length impacts count. When the length of array is 10, count is 3; when myArray.length is 100, count is 6, when myArray.length is 1000, count is 9, and so on, therefore it's big-O notation is $O(\log n)$The inner for-loop is $O(n)$ since it runs through the entire array. Therefore, $O(n) * O(logn) = O(n \log n)$.

d. The big-O notation for four is $O(n^2)$, the outer for-loop iterates through the entire array of A by finding and adding the max number into array B by calling findMax, therefore two nested for-loops which are both $O(n)$ gives $O(n) * O(n) = O(n^2)$.

e. The big-O notation for five is $O(n^2)$, the function goes through 2 nested for-loops one after another so it's $O(n^2) + O(n^2)$, therefore it's big-O notation is just $O(n^2)$.

Vincent Chee
26/02/15
Professor Kristina Striegnitz
Data Structures

Part 2: Runtime Analysis

Each of the methods sorts the eight different types of array using different implementation. The eight different array types are: random arrays, identical arrays, reverse sorted arrays, sorted arrays, largest first element array, largest end element array, smallest start element array and smallest end element array.

Method 1: Based on figures 1 and 2, its big-O notation is O(n). Looking at the graph we can see that the first method didn't have a stand out worst-case scenario. However, by looking at the two figures we can see that the big-O notation cannot be anything other than linear. The time graph and the number of array accesses graphs are somewhat similar, although the time graph has a lot of zigzags, the overall pattern of the two graphs are quite similar. These differences are probably due to rounding off.

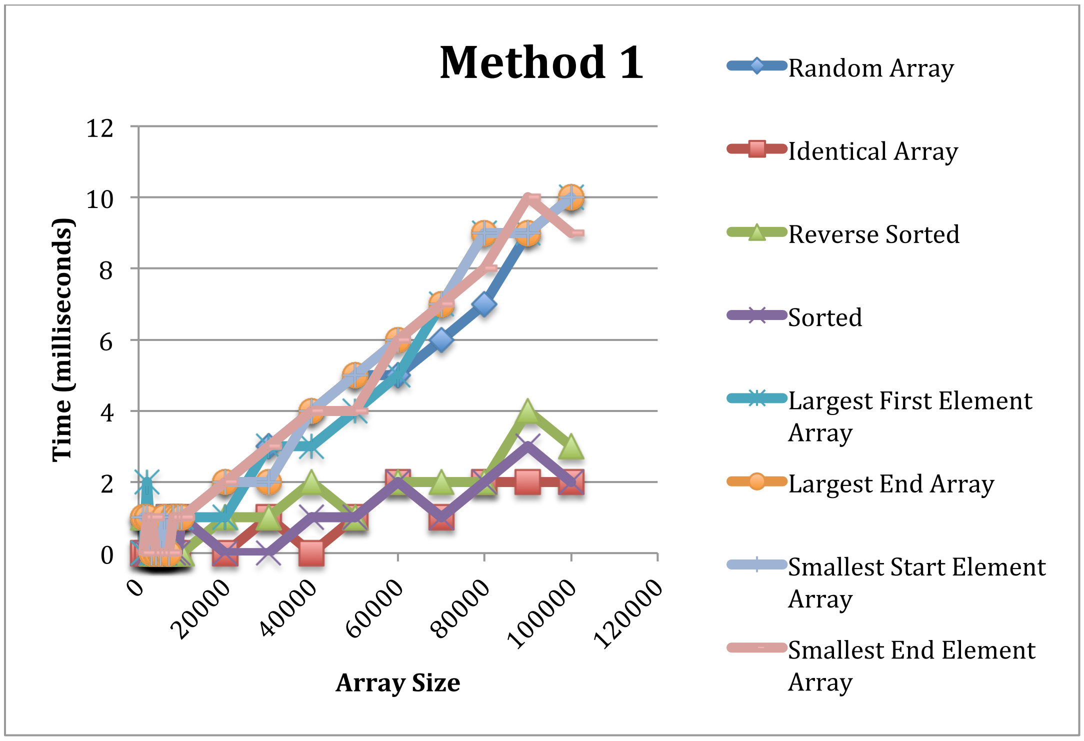

Figure 1: Graph of method 1 that shows the time it took as array size increased.

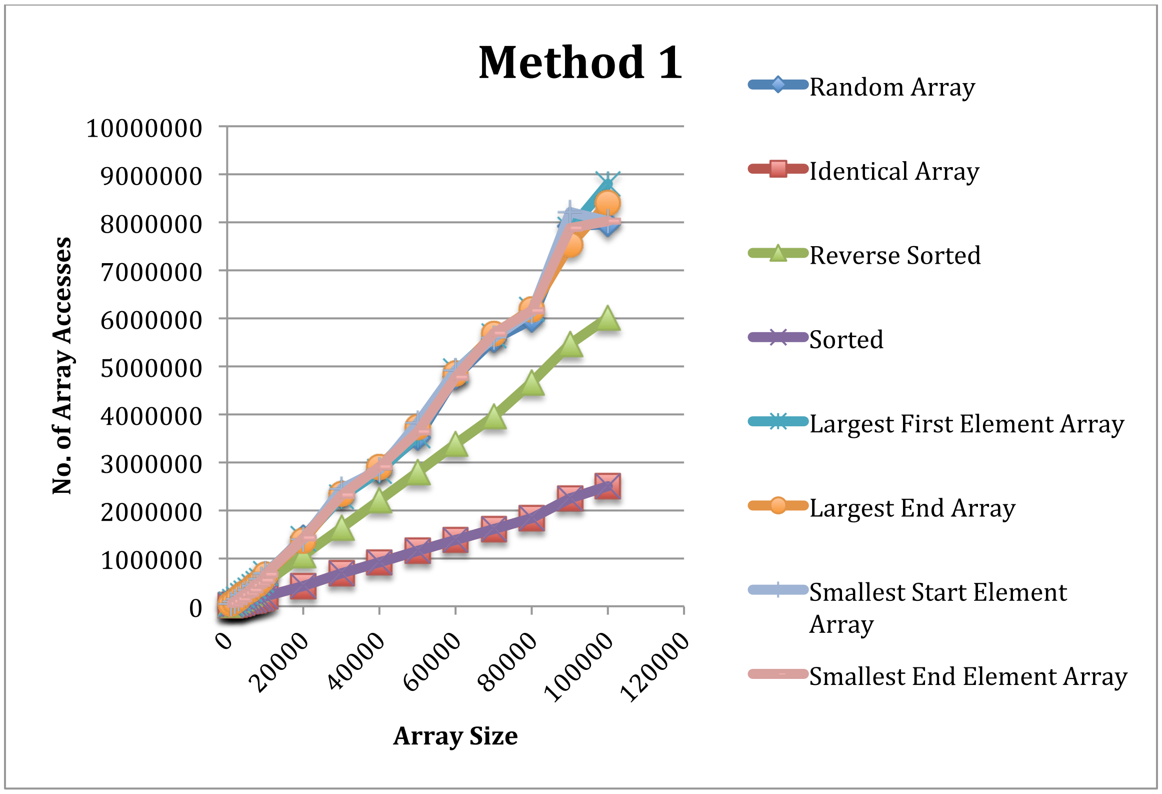

Figure 2: Graph of method 1 that shows the number of array accesses as array size increased.

Method 2: Based on figures 1 and 2, its big-O notation is O(n^2). Just by looking at the graph we can see that there is a clear worst-case scenario. For the sorted and reverse sorted arrays, the number of array accesses had a quadratic increase as the array size increased. Also, just to ensure that the increase is quadratic, we can look at some numbers for the number of accesses for the sorted array:

When the size is 1000, number of accesses is 506495.

When the size is doubled to 2000, the number of accesses quadruples and is 2012995.

The pattern of the time and array access plots for method two looks exactly the same.

Vincent Chee
26/02/15
Professor Kristina Striegnitz
Data Structures

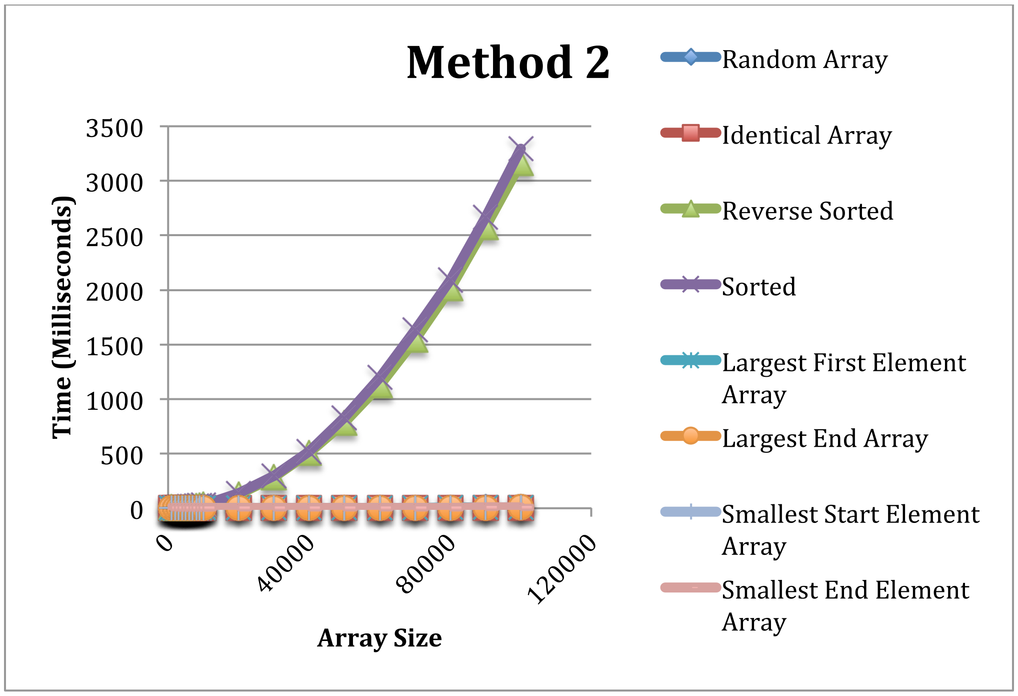

Figure 3: Graph of method 1 that shows the time as array size increased.

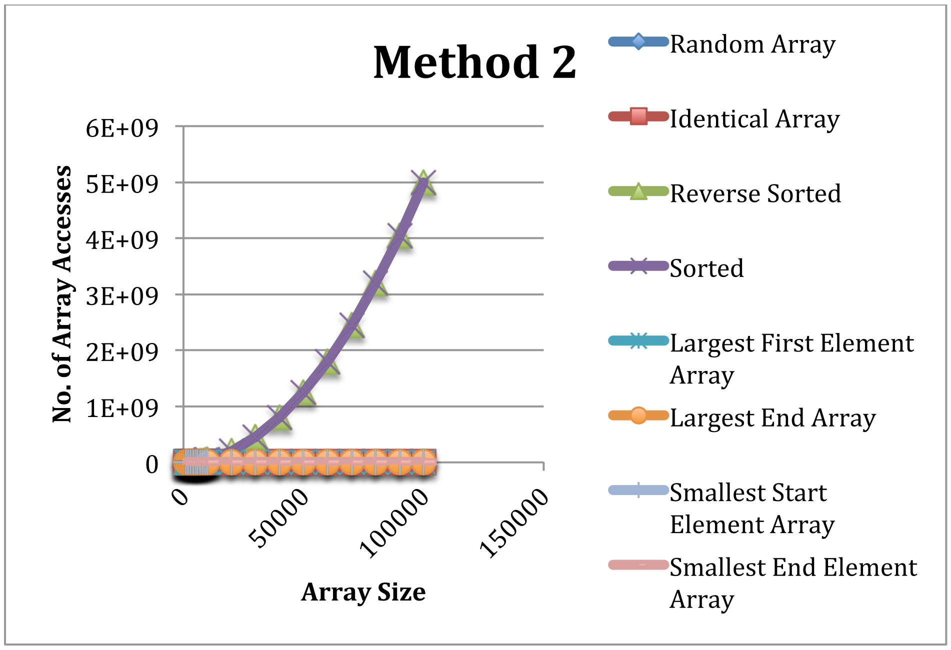

Figure 4: Graph of method 1 that shows the number of array accesses as array size increased.

Method 3: Based on figures 5 and 6, its big-O notation is O(logn). All the array types had a very similar number of array accesses. This method is also quite unique since as the array size increases the number of array accesses decreases when the array sizes are small, however when the array size reaches 150,000 the time and number of array sequences starts to increase. If we take a look at some of the numbers we may be able to determine when it's logarithmic:

When the size is 1000, the number of accesses is 29946232.

When the size is quadrupled to 4000, the number of accesses is 29744928.

Here, we can see that the number of accesses decreases as the size increases. However, the rate of decrease in number of accesses is not constant with the rate of increase in the size of the array. For example, when method 1 had size 1000, the number of accesses for sorted arrays was 14990, when we quadruple the size of the array to 4000; it accessed the array 75986 times. The size of the array quadruples and the number of accesses in turn also quadruples. However, once the size is over 150,000, the number of accesses and time increases. From the two figures the rate of increase seems to become linear. The pattern of the time and array access plots are very different; the time starts very high at the start and then drops really quickly then it increases and decreases until the array size is 150000 when it starts to increase. I'm unsure why these two graphs are so different; one possibility could just be the fact that the hardware performed differently or something trivial.

Vincent Chee
26/02/15
Professor Kristina Striegnitz
Data Structures

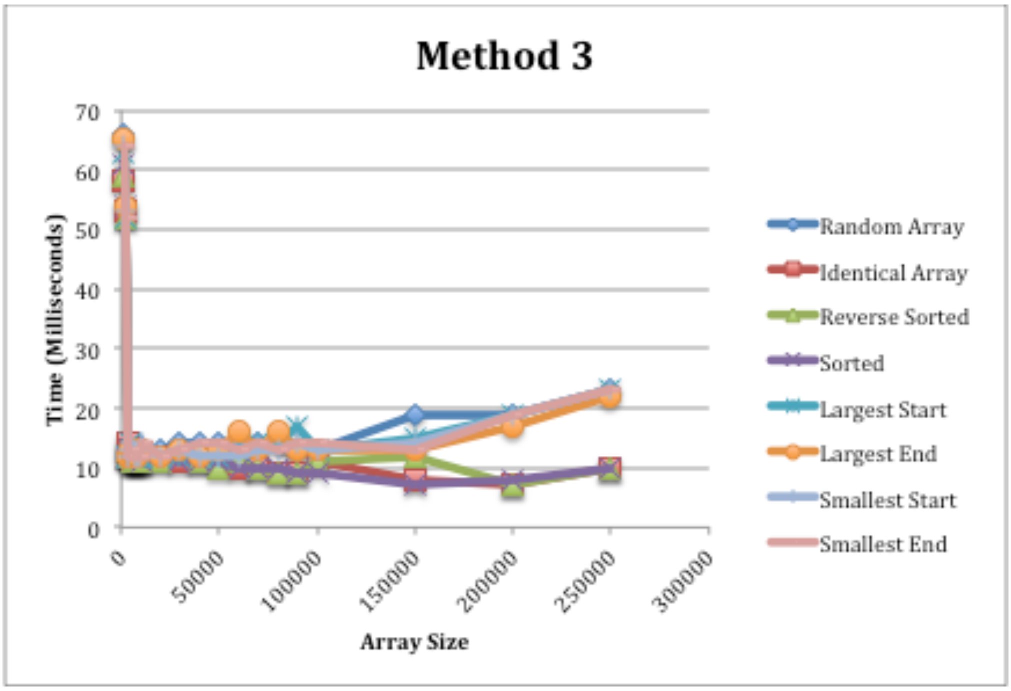

Figure 5: Graph of method 1 that shows the time as array size increased.

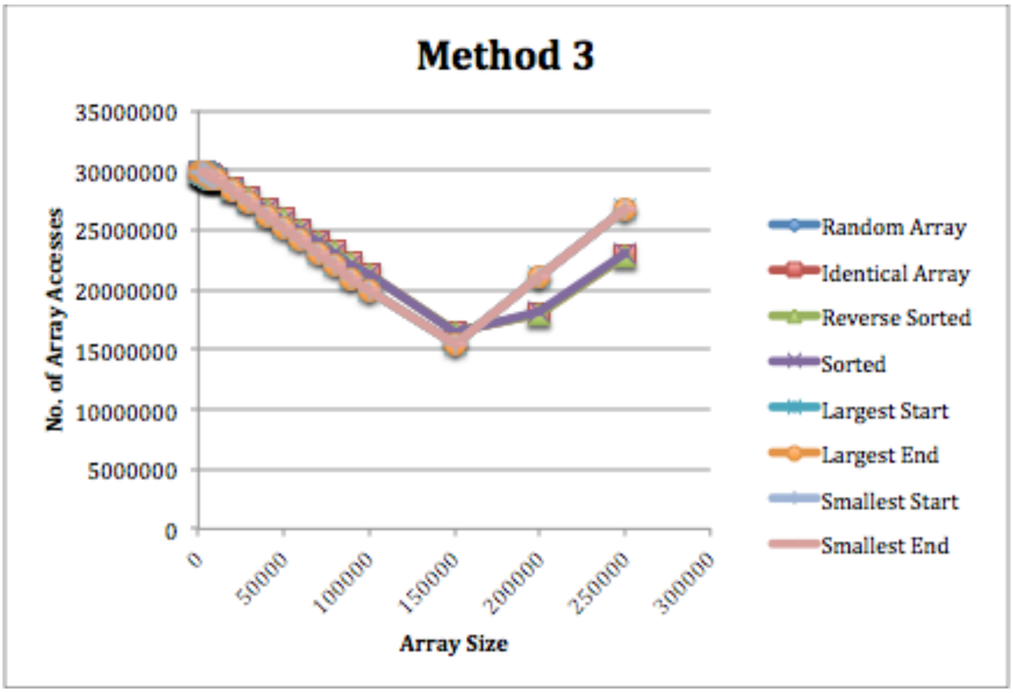

Figure 6: Graph of method 1 that shows the number of array accesses as array size increased.

Method 4: Based on the pattern of figures 7 and 8, its big-O notation is O(n^2). All the arrays had a very similar number of array accesses and the all the array types took around the same time to sort. From the image we can see a concave up curve, so we can make a conjecture that it's quadratic. We can prove this by looking at the numbers: For identical arrays, when we double the size of the array from 1000 to 2000, the output goes from 505496 to 2010996, like the second method the output quadrupled. The pattern of the time and array access plots are quite similar for method four, the small differences may just be due to the rounding off of milliseconds in the code and are trivial.

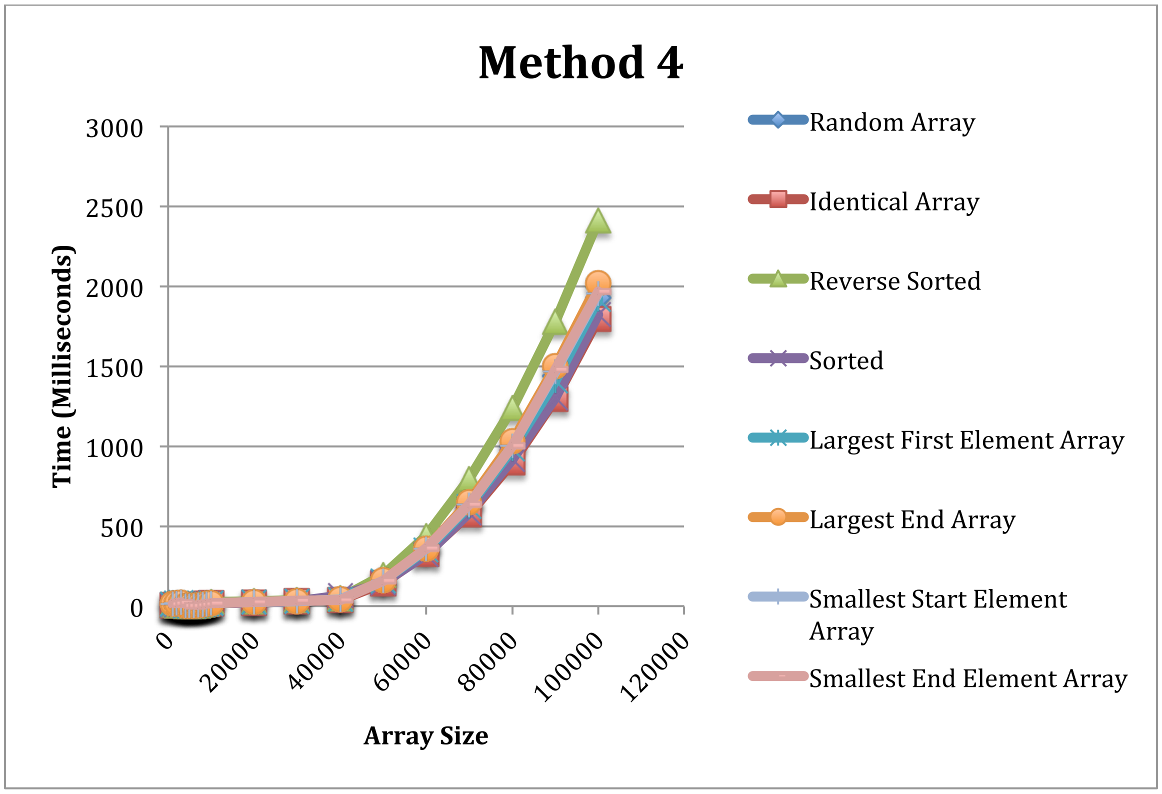

Figure 7: Graph of method 1 that shows the time as array size increased.

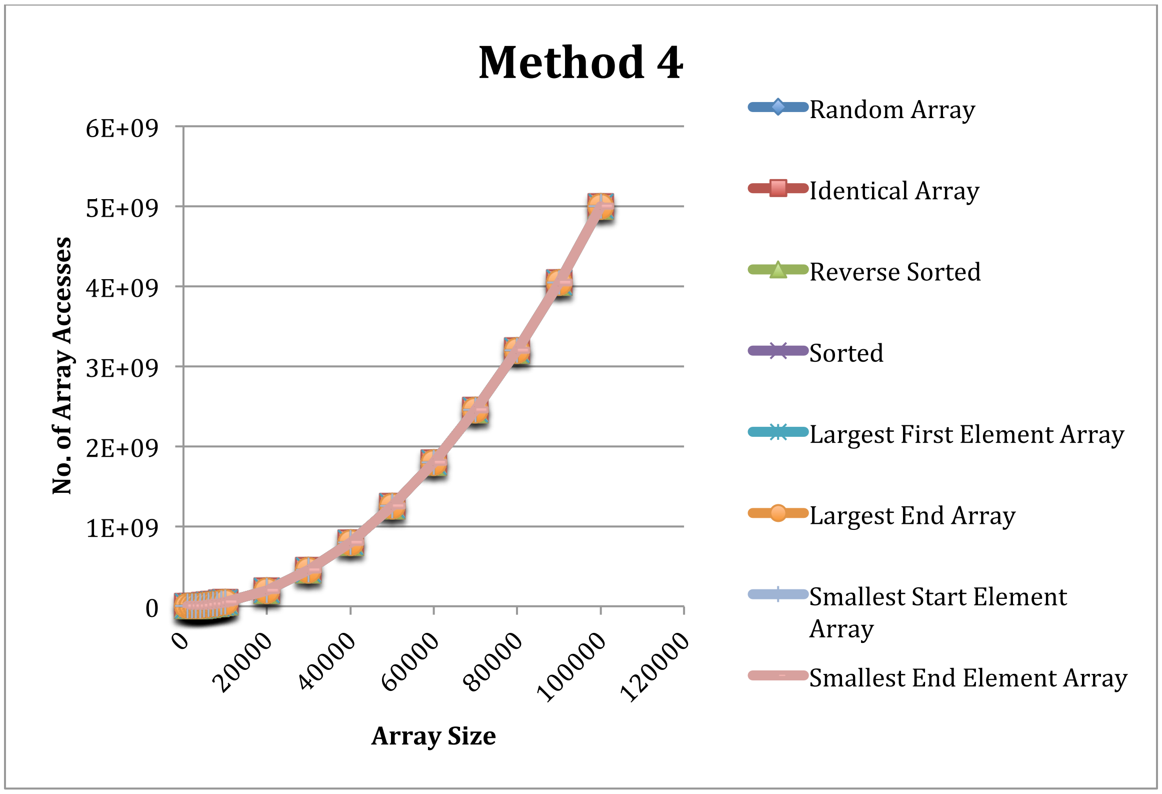

Figure 8: Graph of method 1 that shows the number of array accesses as array size increased.

Method 5: Based on the pattern of figures 9 and 10, its big-O notation is O(n). Looking at figure 9, we see that the shape of the graph is a bit strange. All the arrays had a very similar number of array accesses as we can see in figure 10 and the all the array types took around the same time to sort. The pattern between the time graph and number of array accesses graph are quite minimal, other than the huge spike in the time graph and the appearance that some arrays are being sorted faster, all the arrays have the same number of array accesses.

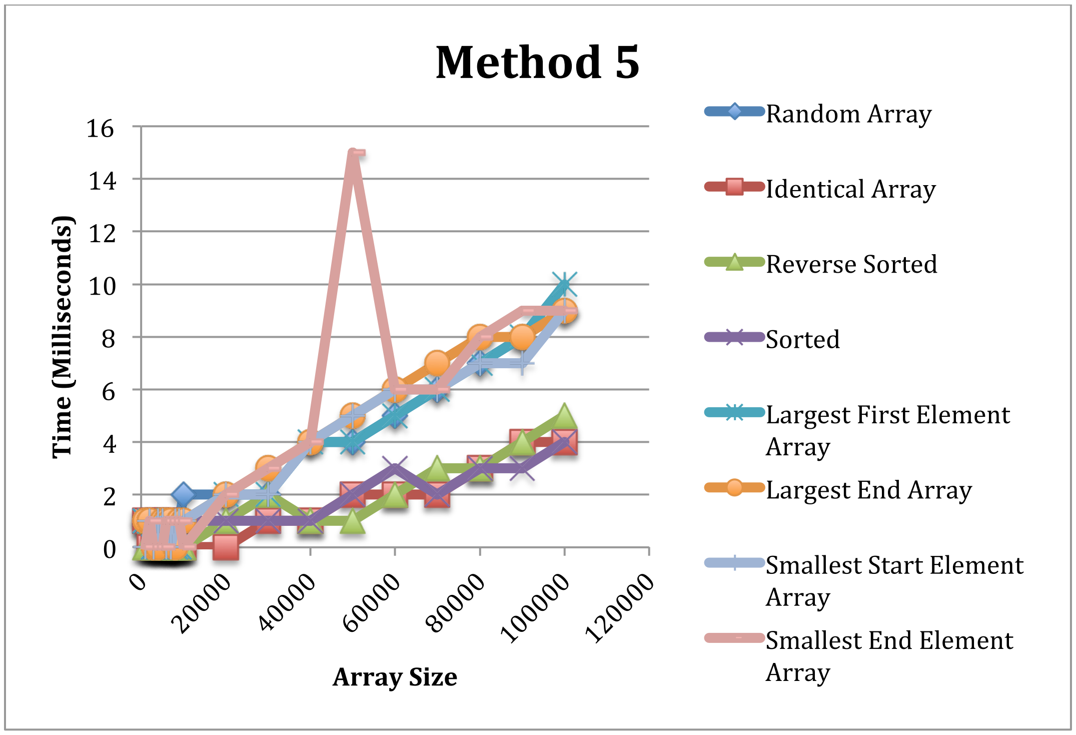

Figure 9: Graph of method 1 that shows the number of array accesses as array size increased.

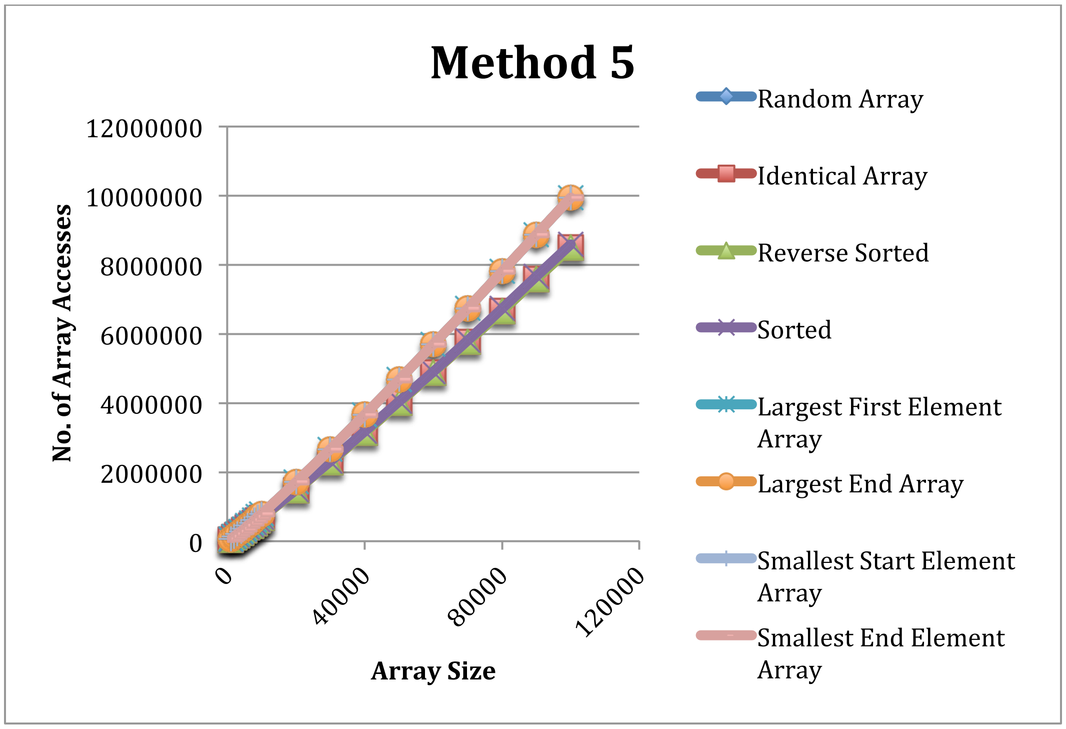

Figure 10: Graph of method 1 that shows the number of array accesses as array size increased.

Method 6: Looking at figures 11 and 12, we can make a conjecture that the big-O notation is either O(n^2) or O(2^n). Looking at the worst-case scenario in figure 11, which is the reverse sorted array type it looks like it's increasing at a faster rate than the other arrays such as the largest first and end element array, the smallest start and end element array and the random array which may possibly have big-O notation which is quadratic. The other array types are kind of irrelevant as you can see in the two figures below. We can calculate the big-O notation of method 6 by looking at the data, first lets look at the runtime for the random array:

At 1000, the number of array accesses 753998 when you double the size the number of array accesses almost quadruples to 2985017. So it's clearly O(n^2) and has a quadratic big-O notation. Knowing that the random array and those other arrays have runtime of O(n^2) and looking at figures 11 and 12, we can make a conjecture that the reverse sorted array is the worst-case time complexity for method six. Looking at the data for the reverse sorted array, when we increase the size by a thousand from 9000 to 10000, the number of array accesses increases from 12150314997 to 15000349997, if it were exponential it would increase at a much faster rate. The rate of increase is actually much faster at the beginning, increasing from 1503497 when the size is 1000 to 6006997 when the size is 2000, a quadruple increase. Therefore, we see that the rate of increase is slowing down therefore it can't have an exponential big-O runtime. Once again, the pattern of the time graph and number of array accesses graph look identical.

Vincent Chee
26/02/15
Professor Kristina Striegnitz
Data Structures

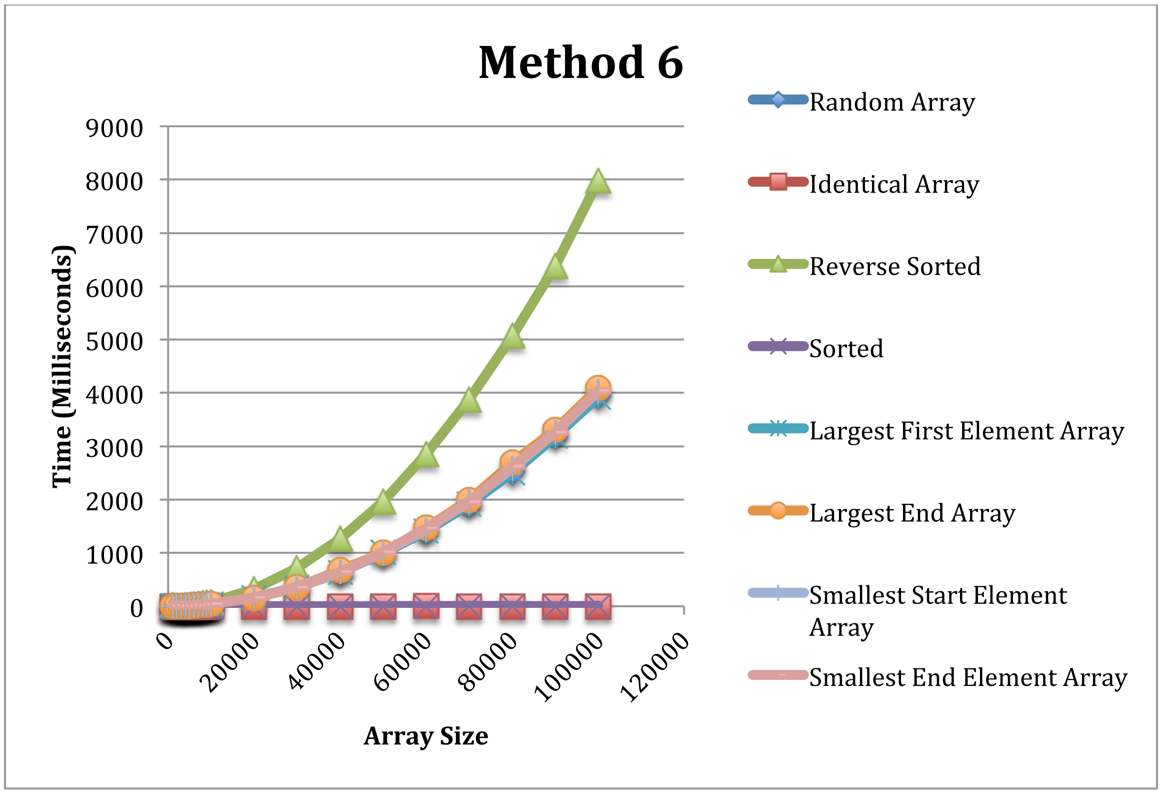

Figure 11: Graph of method 1 that shows the time as array size increased.

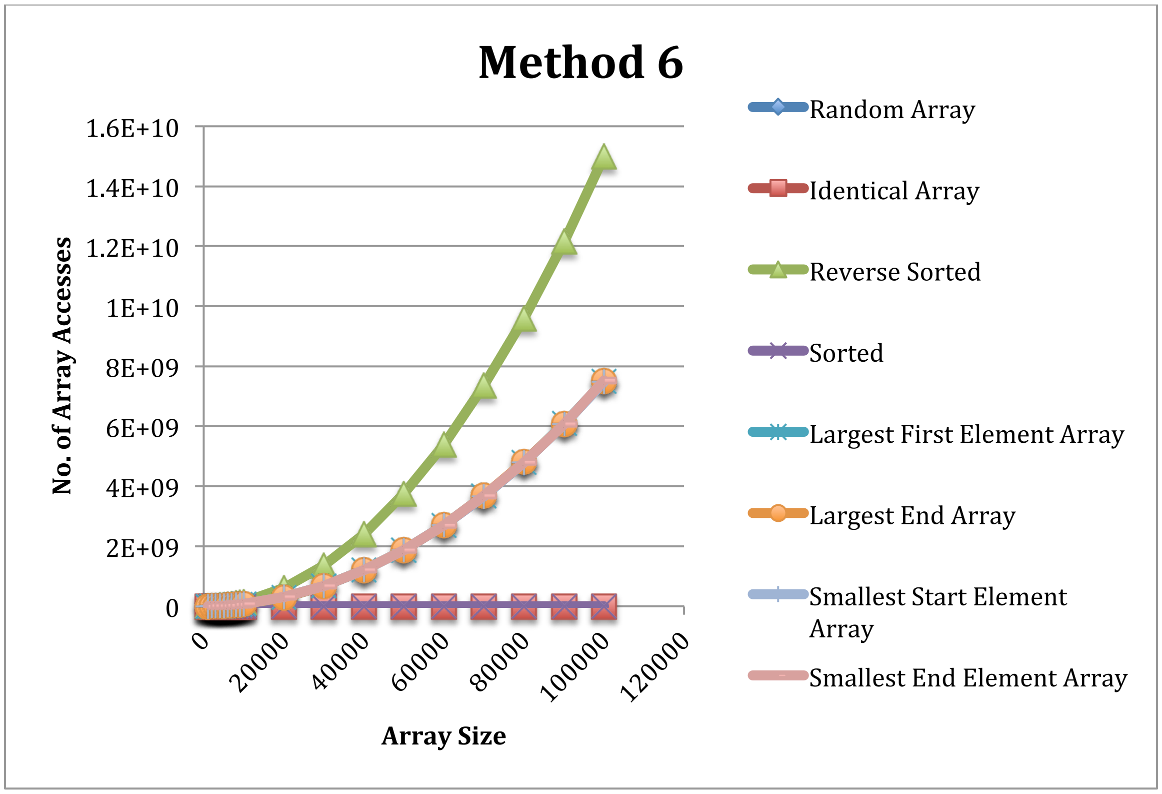

Figure 12: Graph of method 1 that shows the number of array accesses as array size increased.

The eight different array types: random arrays contain random negative integers in a random order, an identical arrays is filled with all the same integers, a reverse sorted array is an array sorted from the largest integer to the smallest, a sorted array is an array which is sorted from the smallest integer to largest, largest first element array is an array with its first element as its largest and the rest as negative integers, largest end element array is an array with its last element as its largest and the rest as negative integers, smallest start element array is an array with its first element as its smallest (Integer.MIN_VALUE) and the rest as negative integers and smallest end element array is an array with its last element as its smallest and the rest as negative integers.