Determining Time Complexities of the algorithms

Determining Time Complexity of Shell Sort algorithm

```
/**
 * Sorts the specified array of integers using the Shell
 * sort algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
void shellSort(T[] data)
{
    int gap = data.length / 2, comparisons = 0, swaps = 0;
    boolean swapflag;
    do // Time complexity of O(n)
    {
        swapflag = true;
        do
        {
            swapflag = false;
            for (int i = 0; i < data.length - gap; i++) // Time complexity of
                                                          O(log_2_n)
            {
                if (data[i].compareTo(data[i + gap]) > 0) // Time complexity of O(1)
                {
                    swap(data, i, i + gap); // Time Complexity of O(1)
                    swapflag = true;
                    swaps++;

                    //for (T obj : data)
                    //{
                    //    System.out.print (obj + " ");
                    //}
                    //System.out.println();
                }
                comparisons++;
            }
        }
        while (swapflag == true);
        gap = gap / 2;
    }
    while (gap > 0);
    System.out.println ("\nTotal comparisons: " + comparisons +
                        "\nTotal swaps: " + swaps);
}
```

Innermost loop time complexity is O(1).
For loop that terminates when i >= data.length - gap time complexity is O(log_2_n)
Outermost do loop that ensures at least one pass is made when gap = 1 is O(n)

Total time complexity is O(1 * n * log_2_n) = O(n log n)

Determining the Time Complexity of Bubble Sort Algorithm

```
/**
 * Sorts the specified array of objects using a bubble sort
 * algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
void bubbleSort(T[] data)
{
    int position, scan, comparisons = 0, swaps = 0;

    for (position = data.length - 1; position >= 0; position--) // Time complexity of O(n/2)
    {
        for (scan = 0; scan <= position - 1; scan++) // Time complexity of O(n/2)
        {
            if (data[scan].compareTo(data[scan + 1]) > 0) // Time complexity of O(1)
            {
                swap(data, scan, scan + 1); // Time Complexity of O(1)
                swaps++;
            }
            comparisons++;
        }
        //for (T obj : data)
        //{
        //    System.out.print (obj + " ");
        //}
        //System.out.println();
    }
    System.out.println ("\nTotal comparisons: " + comparisons + "\nTotal swaps: " + swaps);
}
```

Innermost body of code has time complexity of O(1)
Inner for loop has time complexity of O(n)
Outer for loop has time complexity of O(n)

Total time complexity for the Bubble Sort algorithm is O(n^2)

Determining the Time Complexity of Bubble Sort2 Algorithm

```
/**
 * Sorts the specified array of objects using a bubble sort
 * algorithm. Improves on the efficiency of the original bubbleSort
 * algorithm by avoiding a final pass if during the previous
 * pass no elements were exchanged (i.e. the list is sorted).
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
void bubbleSort2(T[] data)
{
    int scan, position = data.length - 1, comparisons = 0, swaps = 0;
    boolean swapflag = true;

    while (swapflag == true) // Time complexity of O(n)
    {
        swapflag = false;

        // prints each pass through the array
        //for (T obj : data)
        //    System.out.print (obj + " ");
        //System.out.println();

        for (scan = 0; scan <= position - 1; scan++) // Time Complexity O(n)
        {
            if (data[scan].compareTo(data[scan + 1]) > 0) // Time complexity O(1)
            {
                swap(data, scan, scan + 1); // Time complexity O(1)
                swapflag = true;
                swaps++;
            }
            comparisons++;
        }
        position--;
    }
    System.out.println ("\nTotal comparisons: " + comparisons + "\nTotal swaps: " +
                        swaps);
}
```

Time complexity of innermost code: O(1)
Time complexity of inner for loop: O(n)
Time complexity of outer while loop is very variable. O(1) at best, O(n) at worst, O(n/2) on average

Total time complexity for the Bubble Sort 2 algorithm: O(n^2)

Spreadsheet Data: test cases, comparisons, swaps, execution time analysis.

Array Size 10						
		Unsorted			Sorted	
	Trial #	Comparisons	Swaps		Comparisons	Swaps
Shell Sort	1	60	12		22	0
	2	61	15			
	3	60	14			
	4	44	5			
	5	52	13			
	average	55.4	11.8			
Bubble Sort	1	45	20		45	0
	2	45	21			
	3	45	26			
	4	45	13			
	5	45	25			
	average	45	21			
Bubble Sort 2	1	39	20		9	0
	2	35	21			
	3	42	26			
	4	39	13			
	5	45	25			
	average	40	21			

Unsorted Analysis

Here we see that Bubble Sort 2 performs more efficiently than the original Bubble Sort, not making unnecessary comparisons once the data set is sorted, despite the fact that both algorithms make the same number of swaps. It is interesting to note that Bubble Sort always makes the exact same number of comparisons due to its execution despite a data set being sorted, making approximately n^2 / 2 executions.

Shell Sort appears to on average make more comparisons than either of the Bubble Sort algorithms for small data sets; however, it makes up for some of this by making fewer swaps. It is worth mentioning that for small data sets, these differences are immaterial and any of these algorithms will do the job.

Unsorted Analysis

The original Bubble Sort makes no use of this and continues to execute as though the array were unsorted.

When fed a sorted array, Bubble Sort 2 shines, executing exactly n-1 times.

Shell Sort makes use of the fact that the data set was sorted, with an efficiency better than Bubble Sort but worse than Bubble Sort 2.

Conclusions

For small data sets, any of the algorithms will work. However, if we can, we should use Shell Sort or Bubble Sort 2.

Array Size 100						
		Unsorted			Sorted	
	Trial #	Comparisons	Swaps		Comparisons	Swaps
Shell Sort	1	2706	418		503	0
	2	2895	431			
	3	2986	428			
	4	2627	426			
	5	2508	426			
	average	2744.4	425.8			
Bubble Sort	1	4950	2376		4950	0
	2	4950	2807			
	3	4950	2315			
	4	4950	2522			
	5	4950	2388			
	average	4950	2481.6			
Bubble Sort 2	1	4947	2376		99	0
	2	4922	2807			
	3	4872	2315			
	4	4779	2522			
	5	4740	2388			
	average	4852	2481.6			

Unsorted Analysis

With the larger data set, Shell Sort begins to show its value, making significantly almost half the number of comparisons and around 1/5th the number of swaps made by the Bubble Sort algorithms.

Despite the improvement made to Bubble Sort 2, we see that with larger data sets, it performs relatively similarly to the original Bubble Sort. The Bubble Sort algorithms are both approach n^2 time complexity, while Shell Sort is showing the value of O(n log n).

Unsorted Analysis

Bubble Sort continues to perform poorly in comparison to the others with its n^2 / 2 calculations. Bubble Sort 2 on the other hand continues to shine with regards to working with sorted algorithms, once again performing only n-1 calculations. Shell Sort also performs quite well, around O(n).

Conclusions

Here we begin to see the real advantage of the Shell Sort algorithm and time complexities of O (n log n) over O(n^2). The difference between Bubble Sort and Bubble Sort 2 with regards to optimal performance also reinforces the fact that a small change to the code can make huge performance differences.

Array Size 1000							
		Unsorted				**Sorted**	
	Trial #	Comparisons	Swaps			Comparisons	Swaps
Shell Sort	1	57702	7582			8006	0
	2	56783	7490				
	3	56584	7520				
	4	61594	7950				
	5	61494	7314				
	average	58831.4	7571.2				
Bubble Sort	1	499500	245905			499500	0
	2	499500	257548				
	3	499500	250017				
	4	499500	252916				
	5	499500	249038				
	average	499500	251084.8				
Bubble Sort 2	1	498015	245905			999	0
	2	499329	257548				
	3	499455	250017				
	4	498324	252916				
	5	497547	249038				
	average	498534	251084.8				

Unsorted Analysis

Bubble Sort and Bubble Sort 2 begin to perform more and more similarly as the array size increases; they both perform O(n^2). For large arrays, Bubble Sorts seem to be an inefficient way to sort data.

Shell sort in comparison performs extremely well in comparison as the data set grows. Again we see the value in O(n log n) time complexity.

Unsorted Analysis

Shell Sort also continues to perform extremely well with a complexity of O(n). Bubble Sort 2 performs perfectly with n - 1 comparisons. Bubble Sort 2 should be avoided in this scenario as it continues to perform at O(n^2).

Conclusions

Shell sort should be utilized for medium sized data sets. Choosing to use the original Bubble Sort or Bubble Sort 2 for this size of data set makes little difference, as they perform roughly equally well.

For sorting data sets that are potentially already sorted, either Shell Sort or Bubble Sort 2 will perform quite well, but we should avoid the original Bubble Sort if we suspect a data set is unsorted.