Name That Sort Solution Guide

Note that the order of the sorts is randomized based on your student id. These solutions are from student id 0000.

First we seek to identify the quadratic and super-quadratic sort algorithms.

Hypothesis: For randomly ordered input, if we increase the input size by a factor of 10, we would expect the running time of quadratic algorithms to increase by a factor that gets closer and closer to 100, as we increase the input size, because of their worst case running time is bounded above by a constant times n^2. Similarly, we would expect Stooge's running time to increase by a factor of about 513, since it's running time is O(n^2.71), and we would expect that Bogo would not be able to run to completion in a reasonable amount of time given input that contains more than 20 elements, since it runs in O(n!) time.

Table 1: Running Time in Milliseconds of Sort Algorithms Given Unique, Randomly Ordered Inputs of Various Sizes

		Array Size			Factor Increase	
		1,000	10,000	100,000	1,000 -> 10,000	10,000 -> 100,000
Sort Number	1	8	194	23424	24.3	120.7
	2	1	3	26	3.0	8.7
	3					
	4	3	13	48	4.3	3.7
	5	7	34	78	4.9	2.3
	6	1	10	31	10.0	3.1
	7	6	102	6869	17.0	67.3
	8	7	252	32191	36.0	127.7
	9	204	141122	Hours	691.8	
	10	8	131	98.9	16.4	98.9
	11	2	15	2.0	7.5	2.0
	12	1	15	2.5	15.0	2.5
	13	4	8	5.8	2.0	5.8

Based on these results, our conclusions are presented in Table 2.

Table 2: Conclusions After First Round of Tests

		Sort	ACRT
Sort Number	1		θ(n^2)
	2		o(n^2)
	3	Bogo	θ(n!)
	4		o(n^2)
	5		o(n^2)
	6		o(n^2)
	7		θ(n^2)
	8		θ(n^2)
	9	Stooge	O(n^2.71)
	10		θ(n^2)
	11		o(n^2)
	12		o(n^2)
	13		o(n^2)

Of the 10 sorts that remain (which we know about), there are 6 stable sorts (Cocktail Shaker, Bucket, Merge, Insertion, Bubble, Radix), and 4 unstable sorts (Quick, Randomized Quick, Selection, Heap).

Hypothesis: If we provide the inputs described below, and we end up with four sorts that rearrange the order of the first and second keys, then we will be able to identify all four unstable sorts. We can also guess at the stability of sort 13.

Input File 1:

```
3
2 A
2 B
1 C
```

Input File 2 contains 10 keys that are all identical, with distinct string descriptors.

Results: There were exactly four sorts that rearranged the input. Interesting sort 2 put the last element in file 2 first, keeping the rest in order, which is enough evidence to conclude that this is Quick, but we will confirm it shortly. Also, sort 6 did not reorder input file 1 but it did reorder input file 2. Since both heap and selection would definitely reorder input file 1, we could conclude already that sort 6 is Randomized Quick, but we will hold off on this. We will conclude that sort 10 is Selection, since it is a quadratic, unstable sort.

Table 3: Conclusions after Tests for Stability

		Sort	ACRT	Stability
Sort Number	1		θ(n^2)	Stable
	2		o(n^2)	Unstable
	3	Bogo	θ(n!)	Stable
	4		o(n^2)	Stable
	5		o(n^2)	Stable
	6		o(n^2)	Unstable
	7		θ(n^2)	Stable
	8		θ(n^2)	Stable
	9	Stooge	O(n^2.71)	Stable
	10	Selection	θ(n^2)	Unstable
	11		o(n^2)	Stable
	12		o(n^2)	Unstable
	13		o(n^2)	Stable

Next we will look to distinguish between the remaining unidentified quadratic sorts.
Hypothesis: If we provide input which is sorted, except that minimum element appears last, bubble sort will run much more slowly than cocktail shaker or insertion sort, since bubble sort will take O(n^2) time while the other two will run in linear time. Notes: after each iteration of the loop in bubble sort this minimum element will shift by exactly 1 position; there are *n* inversions so insertion sort will run in O(n) time; cocktail shaker sort will take two iterations of the outer loop, so it will also run in O(n) time.

Input File: 10,000 unique elements which are sorted except that the minimum element appears last.

Table 4: Running Time in Milliseconds on Almost Sorted Input (n Inversions with Min Element Last)

		Time
Sort Number	1	2
	7	1
	8	193

Based on these results, it seems pretty clear that sort 8 must be bubble sort.

Hypothesis: If we provide input which has the first sqrt(n) elements in reverse order, but the remaining elements in the correct sorted order, then cocktail shaker sort will be much slower than insertion sort, since there will be *n* inversions and insertion sort will run in linear time, but cocktail shaker sort will take O(n^3/2) time.

Input Files: 10,000 and 1,000,000 unique elements which are sorted except that the first sqrt(n) elements are in reverse order.

Table 5: Running Time in Milliseconds on Almost Sorted Input (n Inversions with first sqrt(n) elements reversed)

		n=10,000	n=1,000,000
Sort Num	1	11	4806
	7	1	17

From this test we conclude that sort 1 is Cocktail Shaker and sort 7 is Insertion.

Table 6: Conclusions after Tests on Quadratic Sorts

		Sort	ACRT	Stability
Sort Number	1	Cocktail Shaker	θ(n^2)	Stable
	2		o(n^2)	Unstable
	3	Bogo	θ(n!)	Stable
	4		o(n^2)	Stable
	5		o(n^2)	Stable
	6		o(n^2)	Unstable
	7	Insertion	θ(n^2)	Stable
	8	Bubble	θ(n^2)	Stable
	9	Stooge	O(n^2.71)	Stable
	10	Selection	θ(n^2)	Unstable
	11		o(n^2)	Stable
	12		o(n^2)	Unstable
	13		o(n^2)	Stable

In order to identify Quick and Randomized Quick sorts, we can provide inputs that would produce a quadratic worst case running time.

Hypothesis: If we provide input which is sorted, the running time for Quicksort will be significantly slower than the other o(n^2) average case sorts, since QuickSort has O(n^2) running time in this case.

Input Files: 10,000 and 50,000 unique elements which are sorted in reverse order.

Interestingly, sort 13 also performed very poorly here! There was a stack overflow for sorts 2 and 13 with the 50k input file. However, I used the argument "-Xss100m" to increase the stack size and get the results shown below

Table 7: Running Time in Milliseconds for Sort Given Sorted Input

		n=10,000	n=50,000
Sort Number	2	139	2972
	4	17	15
	5	22	66
	6	5	12
	11	15	26
	12	7	19
	13	365	8898

We can conclude that sort 2 must be QuickSort.

Hypothesis: If we provide input with all identical keys, the running time for Quicksort and Randomized Quicksort will be significantly slower than the other o(n^2) average case sorts, since both of these sorts have O(n^2) running time in this case.

Input Files: 10,000 and 50,000 identical elements.

Interestingly, sort 13 also performed very poorly here! There was a stack overflow for sorts 2, 6, and 13 with the 50k input file. However, I used the argument "-Xss100m" to increase the stack size and get the results shown below. We can conclude that sort 6 is Randomized Quick.

Table 8: Running TIme in Milliseconds When Given Input With Identical Key Values

		n=10,000	n=50,000
Sort Number	2	137	3206
	4	4	10
	5	15	62
	6	183	4127
	11	4	7
	12	2	5
	13	364	8980

Based on the previous results, it appears that we have three sorts with an average case running time of o(n^2) but a worst case running time that appears to be θ(n^2), including our mystery sort 13. The only remaining unstable o(n^2) sort must be heap sort. Table 9 shows our conclusions so far.

Table 9: Conclusions after Tests With Sorted/Duplicate Elements

		Sort	ACRT	Stability	WCRT
Sort Number	1	Cocktail Shaker	θ(n^2)	Stable	
	2	Quick	o(n^2)	Unstable	θ(n^2)
	3	Bogo	θ(n!)	Stable	
	4		o(n^2)	Stable	
	5		o(n^2)	Stable	
	6	Randomized Quick	o(n^2)	Unstable	θ(n^2)
	7	Insertion	θ(n^2)	Stable	
	8	Bubble	θ(n^2)	Stable	
	9	Stooge	O(n^2.71)	Stable	
	10	Selection	θ(n^2)	Unstable	
	11		o(n^2)	Stable	
	12	Heap	o(n^2)	Unstable	
	13		o(n^2)	Stable	θ(n^2)

The remaining sorts to identify are Bucket, Merge, and Radix.

Hypothesis: If we provide input that contains a single input value with maximum key, BucketSort will take much longer than Merge or Radix, since it must create an array of size equal to this maximum key.

Input: A file with a single key value equal to 100,000,000.

Table 10: Running Time in Milliseconds with a Single Large Key

		Time
Sort Number	4	Heap size error
	5	0
	11	1

Clearly sort 4 must be bucket sort – we see this not because it took a really long time to complete, but rather because there was not enough room on the heap to allocate an array with 100 million elements.

Hypothesis: If we provide two input files, one with 100,000 keys all equal to 1 and one with 100,000 keys all equal to the maximum value, merge sort will take about the same amount of time on both, but radix sort will take longer on the second, since merge sort does about the same amount of work for any input of a fixed size, but the amount of work done by radix sort is a function of the maximum value in the input file.

Table 11: Running Time in Milliseconds with Duplicate Keys

		key=1	key=100,000,000
Sort Num	5	60	60
	11	15	53

It appears then, that sort 5 is merge and sort 11 is radix.

For sort 13, we know that the sort performs extremely poorly when provided input that is in sorted order, but it appears to run in better than quadratic time given random input. The data structure that this might remind you of is a binary search tree, which when given a list of sorted keys to insert, degenerates into a linked list, with each new key appended to the end of the list. However, when given randomly ordered input, the tree would have a height of O(log n). The sequence of insert operations would take O(n^2) time if the input is sorted, and O(n log n) if the input is randomly ordered. Thus the behavior of sort 13 is consistent with an approach that adds all of the keys to a binary search tree, and then does an in-order traversal to output the items in sorted order.

Thus our final conclusions are:

Sort #1: CocktailShakerSort
Sort #2: QuickSort
Sort #3: BogoSort
Sort #4: BucketSort
Sort #5: MergeSort
Sort #6: Randomized QuickSort
Sort #7: InsertionSort
Sort #8: BubbleSort
Sort #9: StoogeSort
Sort #10: SelectionSort
Sort #11: RadixSort
Sort #12: HeapSort
Sort #13: Tree Sort