Written Problems

Problem Set 2
Graham Schmidt
schmidtg@gmail.com
Harvard ID: 30825489

1. Sorting Practice

    a. Selection Sort after third pass:

    ```
    2 3 12 13 34 24 50 27
    ```

    b. Insertion Sort:

    ```
    inner do…while loop would be skipped 3 times
    ```

    c. Shell sort after initial phase:

    ```
    13 3 2 24 12 27 50 34
    ```

    d. Bubble sort after fourth pass

    ```
    3 2 13 12 24 27 34 50
    ```

    e. Quicksort after initial phase

    ```
    12 3 2 13 34 27 50 24
    ```

    f. Radix sort after the initial pass

    ```
    50 02 12 03 13 24 34 27
    ```

    g. Mergesort after completion of the fourth call to the merge() method

    ```
    3 13 24 27 2 34 50 12
    ```

2. Comparing two algorithms

    Algorithm A: sequential, keeps track of largest element
    Algorithm B: sorts array using most efficient sort, the reports last element as largest

    Algorithm A must look at each element in the array 'n' times. Because it must always do this to determine the largest element, it's best and worst time efficiency is O(n).

Algorithm B must first sort the unsorted array. Using the most efficient sorting algorithm (like Quick or Merge) it could only expect a best time efficiency of O(logn) which is slower than O(n). It also has to manipulate or look at every element in the array, adding overhead any time it must make a move or comparison on the sorting part itself.

3. Counting Comparisons

   a. Selection sort: 15
   In the case of a sorted array, the number of comparisons is equal to the sum of arithmetic sequence: n(n-1)/2. So with 6 elements, this becomes 15.

   b. Insertion sort: 5
   An already sorted array is the best case for insertion sort. It only needs to compare each element to it's previous element. So therefore it's n-1 times, or 5 in this example.

   c. Merge sort: 9
   Since the array is sorted, it will make 9 comparisons total. These occur as the merge compares each valued returned from it's base case of single elements. Since not items are out of order, it does not need to recursively check subarrays again, it'll simply merge those subarrays once.

4. Swap Sort

   a. Best Case
   Sorted array: Same as selection or bubble sort, it's $O(n^2)$, since it needs to compare each element with the remaining elements in the list. However, if each element is already sorted, then it will not have to move any elements.

   Comparisons: $C(n) = n(n-1)/2 = O(n^2)$

   Moves: $M(n) = 0$

   Overall time efficiency: $C(n) + M(N) = O(n^2)$

   b. Worst Case
   Reverse sorted array: In this case each element will still be compared to the ones after it, giving it a $O(n^2)$ for the comparisons. The moves however, will be significant as it'll need to swap each element n times too.

   Comparisons: $C(n) = n(n-1)/2 = O(n^2)$

   Moves: $M(n) = n(n-1)/2 = O(n^2)$

   Overall time efficiency: $C(n) + M(N) = O(n^2)$

5. Mode finder

a. Exact formula

$C(n) = n(n - 1)/2 = n^2/2 - n/2$

b. The time efficiency of the method as a function of the length of the array (n) is $O(n^2)$ since the largest growing component is $n^2/2$.

c. Alternative algorithm for findMode()

```
public static int findModeFast(int[] arr) {
      int mode = -1;
      int modeFreq = 0;

      quickSort(arr);

      int freq = 1;
      for (int i = 1; i < arr.length; i++) {

          // Compare each element to its previous
          if (arr[i] == arr[i - 1]) {
              freq++;
          } else {
              freq = 1;
          }

          if (freq > modeFreq || (freq == modeFreq && arr[i]
< mode)) {
              mode = arr[i];
              modeFreq = freq;
          }
      }

      return mode;
  }
```

d. By sorting the array via Quicksort, we're giving it a worst case of O(nlogn). Once the array is in sorted order, it can easily perform a single pass through the array to check for the frequency of all values, comparing each value to the previous value. The time efficiency becomes:

$C(n) = n\log n + n$
$= O(n\log n)$

6.

a.

Expression	Address	Value
x	0x128	0x840
x.ch	0x840	h'
y.prev	0x846	null
y.next.prev	0x666	0x320
y.prev.next	0x402	0x320
y.prev.next.next	0x322	null

b. Java code fragment

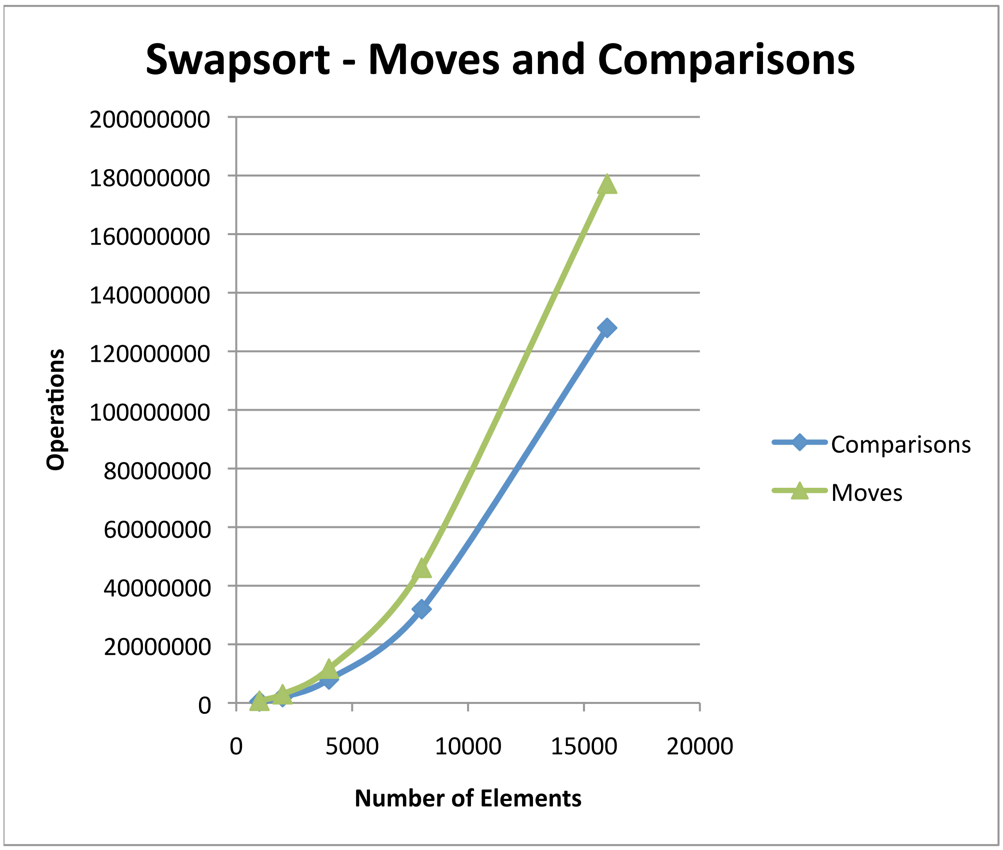
```
y.prev.next = x;
x.next = y;
x.prev = y.prev;
y.prev = x;
```

Experimental Analysis - swapSort

n	num runs	comparisons r	moves r	comparisons f	moves f
1000	10	499500	750493	499500	0
2000	10	1999000	2973191	1999000	0
4000	10	7998000	11725676	7998000	0
8000	10	31996000	46131770	31996000	0
16000	10	127992000	177230525	127992000	0

Graph Data

n	comparisons	moves
1000	499500	750493
2000	1999000	2973191
4000	7998000	11725676
8000	31996000	46131770
16000	127992000	177230525



According to my analysis, swapSort belongs to the big-O efficiency class of O(n2). As n doubles, the number of comparisons will always go up quadratically, in both the best and worst cases. In the best case where the array is already sorted, the algorithm must still compare the element in position to the remaining (n - 2) positions. Even with a slight optimization to avoid comparing the last element (as it will have already been swapped when comparing the element before it), the algorithm is still slow. In the average case, where the data is random, the number of moves goes up quadratically, i.e. when n doubles, the number of moves goes up by 4. When n quadruples, moves goes up by a factor of 16.

Experimental Analysis of swapSort

n	Type	run	comparisons	moves
1000	r	1	499500	744828
1000	r	2	499500	754197
1000	r	3	499500	744519
1000	r	4	499500	758505
1000	r	5	499500	780288
1000	r	6	499500	726573
1000	r	7	499500	759657
1000	r	8	499500	762342
1000	r	9	499500	720672
1000	r	10	499500	753351
		Average	499500	750493
1000	f	1	499500	0
1000	f	2	499500	0
1000	f	3	499500	0
1000	f	4	499500	0
1000	f	5	499500	0
1000	f	6	499500	0
1000	f	7	499500	0
1000	f	8	499500	0
1000	f	9	499500	0
1000	f	10	499500	0
		Average	499500	0
2000	r	1	1999000	2955204
2000	r	2	1999000	2944047
2000	r	3	1999000	2988129
2000	r	4	1999000	2990493
2000	r	5	1999000	3003201
2000	r	6	1999000	2957853
2000	r	7	1999000	2987073
2000	r	8	1999000	2963448
2000	r	9	1999000	2991417
2000	r	10	1999000	2951049
		Average	1999000	2973191
2000	f	1	1999000	0
2000	f	2	1999000	0
2000	f	3	1999000	0
2000	f	4	1999000	0
2000	f	5	1999000	0
2000	f	6	1999000	0
2000	f	7	1999000	0
2000	f	8	1999000	0
2000	f	9	1999000	0
2000	f	10	1999000	0

	Average		1999000	0
4000 r		1	7998000	11822637
4000 r		2	7998000	11731182
4000 r		3	7998000	11687262
4000 r		4	7998000	11782992
4000 r		5	7998000	11544636
4000 r		6	7998000	11819871
4000 r		7	7998000	11800539
4000 r		8	7998000	11876292
4000 r		9	7998000	11511783
4000 r		10	7998000	11679564
	Average		7998000	11725676
4000 f		1	7998000	0
4000 f		2	7998000	0
4000 f		3	7998000	0
4000 f		4	7998000	0
4000 f		5	7998000	0
4000 f		6	7998000	0
4000 f		7	7998000	0
4000 f		8	7998000	0
4000 f		9	7998000	0
4000 f		10	7998000	0
	Average		7998000	0
8000 r		1	31996000	46207338
8000 r		2	31996000	46806399
8000 r		3	31996000	46195140
8000 r		4	31996000	46281594
8000 r		5	31996000	46162032
8000 r		6	31996000	46216110
8000 r		7	31996000	45903570
8000 r		8	31996000	45634290
8000 r		9	31996000	45845616
8000 r		10	31996000	46065606
	Average		31996000	46131770
8000 f		1	31996000	0
8000 f		2	31996000	0
8000 f		3	31996000	0
8000 f		4	31996000	0
8000 f		5	31996000	0
8000 f		6	31996000	0
8000 f		7	31996000	0
8000 f		8	31996000	0
8000 f		9	31996000	0
8000 f		10	31996000	0
	Average		31996000	0

Experimental Analysis - swapSort

16000	r		1	127992000	179256285
16000	r		2	127992000	177310698
16000	r		3	127992000	175446228
16000	r		4	127992000	176945598
16000	r		5	127992000	177127209
16000	r		6	127992000	177245973
16000	r		7	127992000	175358079
16000	r		8	127992000	177323535
16000	r		9	127992000	177728130
16000	r		10	127992000	178563519
		Average		127992000	177230525