Problem set 2
Due before lecture on Wednesday, October 5

I. Written problem

1. Sorting Practice (14 points)

Given array:

24	3	27	13	34	2	50	12

a. **Selection sort, after 3rd pass**

2	3	12	13	34	24	50	27

b. **Insertion sort**
The do…while() loop would be skipped **3 times**, for 27, 34 and 50.

c. **Shell sort (increment 3) after initial phase**

13	3	2	24	12	27	50	34

d. **Bubble sort, after 4th pass**

3	2	13	12	24	27	34	50

e. **Quick sort, after initial partitioning phase**
Pivot: 13

12	3	2	13	34	27	50	24

f. **Radix sort, after initial pass**

5**0**	0**2**	1**2**	0**3**	1**3**	2**4**	3**4**	2**7**

g. Merge sort, after 4th call to merge()

3	13	24	27	2	34	50	12

2. Comparing two algorithms (5 points)

Algorithm A	Algorithm B
$\boldsymbol{C(n) = n}$ $\boldsymbol{M(n) = n}$ **Time eff.:** $\boldsymbol{2n = O(n)}$	$C(n) = n * \log(n)$ $M(n) = n * \log(n)$ **Time eff.:** $2n * \log(n) = O(n * \log(n))$

The most efficient comparison based algorithm is O(n*log(n)).

O(n) <= O(n*log(n)) so the **Algorithm A** would be **more time efficient** in this case.

3. Counting comparisons (6 points)

Given an already sorted array, how many comparisons would each algorithm perform?

a. Selection sort
For each iteration, we compare the current element to all the elements on the right.

$$C(n) = \frac{(n-1)n}{2} = O(n^2)$$

In our case, n=6: there would be 15 comparisons.

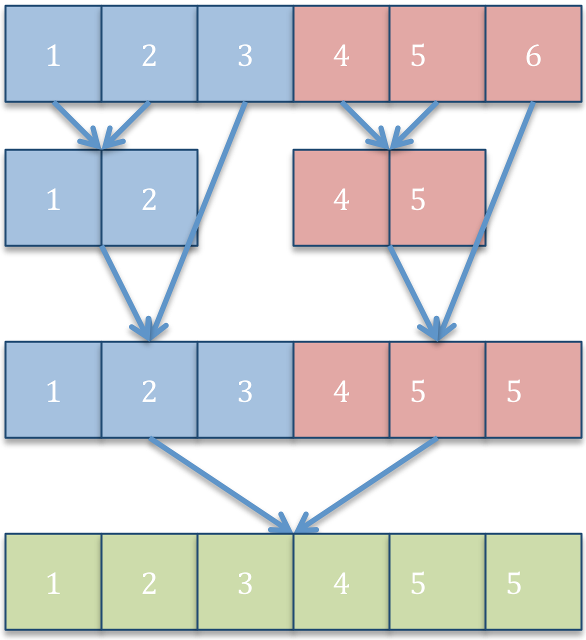
b. Insertion sort
For each iteration, we only compare to the previous element.

$$C(n) = n - 1 = O(n)$$

In our case, n=6: there would be 5 comparisons.

c. Merge Sort

For the given array:



1- compare 1 and 2: 1 comparison
2- compare [12] and 3 : 2 comparisons
3- compare 4 and 5: 1 comparison
4- compare [45] and 6: 2 comparisons
5- compare [123] and [456]: 3 comparisons

So there would be 9 comparisons. Note that to compare , we always take advantage of the fact that we compare **SORTED** items.

4. Swap sort (10 points)

a. Best case

Already sorted array: (smallest to biggest value)

We always have to do $\frac{(n-1)n}{2}$ comparisons.

$$C(n) = \frac{(n-1)n}{2} = O(n^2)$$

In the best case, the algorithm is already sorted so we don't have to do any swap.

$M(n) = 0 = O(0)$

The overall time efficiency would be $C(n) + M(n) = \frac{(n-1)n}{2} = O(n^2)$

b. Worst case

Array sorted in inverse order (biggest to smallest value)

We always have to do $\frac{(n-1)n}{2}$ comparisons.

$$C(n) = \frac{(n-1)n}{2} = O(n^2)$$

In the worst case, we have to swap element after each comparison.

$$M(n) = \frac{(n-1)n}{2} = O(n^2)$$

The overall time efficiency would be $C(n) + M(n) = n(n-1) = O(n^2)$

5. **Mode finder** (10-20 points)

a. Number of time arr[i] is compared to arr[j]

First iteration: n-1 comparisons
Second iteration: n-2 comparisons
...
Last iteration: 1 comparison

$$f(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

b. Time efficiency

The number of comparisons is $\frac{(n-1)n}{2}$

$$C(n) = \frac{(n-1)n}{2} = O(n^2)$$

The number of moves is <= 2n. (the important thing here is that it's < n^2)

$M(n) \leq 2n = O(n)$

Therefore, the time efficiency of the method is:

$C(n) + M(n) = O(n^2)$

c. Alternative solution

```
public static int modeFinder(int[] arr){
  // merge sort the array
  mergeSort(arr);
  // init the mode
  int mode = arr[0];
  int modeFrequence = 0;
  int tempFrequence = 0;
  // go through all elements 1 time!
  for(int i = 0; i<arr.length-1; i++ ){
      if(arr[i] == arr[i+1]){
        tempFrequence++;
        }
      else if(tempFrequence>modeFrequence){
        modeFrequence = tempFrequence;
        mode = arr[i];
        tempFrequence = 0;
         }
      else{
        tempFrequence = 0;
        }
  }
return mode;
}
```

d. Time efficiency

The merge sort is O(n*log(n)).
Then we go through the whole array one time, with 3 moves at maximum.
Then, the second part is <= 3n then O(n).
Therefore, the whole method is O(n*log(n)) + O(n) = O(n*log(n))

6. Practice with reference (10 points)

a. Table

Expression	Address	Value
x	0x128	0x840
x.ch	0x840	‘h’
y.prev	0x324	0x400
y.next.prev	0x664	0x320
y.prev.next	0x402	0x320
y.prev.next.next	0x322	0x660

b. Java code fragment

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

II. Programming problem

2. Practice with reference (10 points)

Unordered arrays:

1000 items	2000 items
499500 comparisons 780591 mov	1999000 comparisons 2922087 mo
499500 comparisons 768684 mov	1999000 comparisons 3037533 mo
499500 comparisons 775572 mov	1999000 comparisons 2942115 mo
499500 comparisons 754938 mov	1999000 comparisons 2878998 mo
499500 comparisons 739194 mov	1999000 comparisons 2846847 mo
499500 comparisons 728136 mov	1999000 comparisons 2873883 mo
499500 comparisons 736260 mov	1999000 comparisons 2979246 mo
499500 comparisons 764142 mov	1999000 comparisons 2938797 mo
499500 comparisons 736875 mov	1999000 comparisons 3002619 mo
499500 comparisons 780498 mov	1999000 comparisons 2986776 mo
499500 comparisons 724314 mov	1999000 comparisons 3020676 mo
499500 comparisons 757044 mov	1999000 comparisons 3021921 mo
499500 comparisons 735798 mov	1999000 comparisons 2970939 mo
499500 comparisons 751668 mov	1999000 comparisons 2951724 mo
499500 comparisons 742599 mov	1999000 comparisons 2931609 mo
499500 comparisons 757836 mov	1999000 comparisons 2941884 mo
499500 comparisons 753042 mov	1999000 comparisons 3019362 mo
499500 comparisons 775365 mov	1999000 comparisons 3032001 mo
499500 comparisons 772953 mov	1999000 comparisons 2950734 mo
499500 comparisons 765123 mov	1999000 comparisons 2956836 mo
4000 items	**8000 items**
7998000 comparisons 11637423 mo	31996000 comparisons 45600540 mov
7998000 comparisons 11811222 mo	31996000 comparisons 46596204 mov
7998000 comparisons 11713503 mo	31996000 comparisons 45646608 mov
7998000 comparisons 11767260 mo	31996000 comparisons 45694179 mov
7998000 comparisons 11542968 mo	31996000 comparisons 45763854 mov
7998000 comparisons 11781465 mo	31996000 comparisons 45564048 mov
7998000 comparisons 11756205 mo	31996000 comparisons 46561299 mov
7998000 comparisons 11684220 mo	31996000 comparisons 45906651 mov
7998000 comparisons 11672097 mo	31996000 comparisons 46333917 mov
7998000 comparisons 11678292 mo	31996000 comparisons 45567504 mov
7998000 comparisons 11737248 mo	31996000 comparisons 46396983 mov
7998000 comparisons 11843694 mo	31996000 comparisons 46445712 mov
7998000 comparisons 11660850 mo	31996000 comparisons 45989364 mov
7998000 comparisons 11640936 mo	31996000 comparisons 46450683 mov
7998000 comparisons 11557641 mo	31996000 comparisons 46110726 mov
7998000 comparisons 11855532 mo	31996000 comparisons 45796152 mov
7998000 comparisons 11716047 mo	31996000 comparisons 45604791 mov
7998000 comparisons 11671935 mo	31996000 comparisons 46083717 mov
7998000 comparisons 11969646 mo	31996000 comparisons 45665592 mov
7998000 comparisons 11746638 mo	31996000 comparisons 46305582 mov

16000 items	
127992000 comparisons 175203444 mo	
127992000 comparisons 177917553 mo	
127992000 comparisons 177883962 mo	
127992000 comparisons 177387795 mo	
127992000 comparisons 177302292 mo	
127992000 comparisons 177648963 mo	
127992000 comparisons 176500962 mo	
127992000 comparisons 177378651 mo	
127992000 comparisons 178213119 mo	
127992000 comparisons 176501244 mo	
127992000 comparisons 177276813 mo	
127992000 comparisons 178464003 mo	
127992000 comparisons 177724878 mo	
127992000 comparisons 175419630 mo	
127992000 comparisons 177427983 mo	
127992000 comparisons 177436134 mo	
127992000 comparisons 176947122 mo	
127992000 comparisons 176096532 mo	
127992000 comparisons 175352547 mo	
127992000 comparisons 178608429 mo	

N = 1 000: 1 200 000 operations -> 1 000 * 1 000
N = 2 000: 5 000 000 operations -> 2 000 * 2 000
N = 4 000: 19 000 000 operations -> 4000 * 4 000
N = 8 000: 77 000 000 operations -> 8 000 * 8 000
N = 16 000: 300 000 000 operations -> 16 000 * 16 000

It is a O(n^2) algorithm.

You can no predict how many moves you will have to do: (in case some items are ordered)
n-1 * n / 2 comparisons
n-1 * n / 2 swaps (worst case)
(1 swap is 3 moves)

We can just say that for a 1000 items array,
499500 (best) < operations < 4 * 499500 (worst) will be performed.

Ordered arrays:

1000 items	2000 items
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
499500 comparisons 0 moves	1999000 comparisons 0 move
4000 items	**8000 items**
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov
7998000 comparisons 0 moves	31996000 comparisons 0 mov

16000 items	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	
127992000 comparisons 0 mov	

N = 1 000: 500 000 operations -> 1 000 * 1 000
N = 2 000: 2 000 000 operations -> 2 000 * 2 000
N = 4 000: 8 000 000 operations -> 4000 * 4 000
N = 8 000: 32 000 000 operations -> 8 000 * 8 000
N = 16 000: 130 000 000 operations -> 16 000 * 16 000

Roughly $n^2/2$ algorithm, therefore it is a $O(n^2)$ algorithm.

You can predict that 0 moves will be performed since the array is ordered.
n-1 * n / 2 comparisons
0 moves

Therefore, we can just say that for a 1000 items array, 499500 will be performed.