**Problem set 2**

**Due before lecture on Wednesday, October 5**

**1. Sorting Practice** (14 points)

Given array:

12

50

34

2

13

27

3

24

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

27

50

13

12

24

34

3

2

1. **Insertion sort**

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

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

34

50

24

2

27

12

3

13

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

3

2

24

27

13

12

34

50

1. **Quick sort, after initial partitioning phase**

Pivot: 13

12

3

34

27

2

13

50

24

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

1**2**

0**2**

0**3**

1**3**

2**4**

3**4**

2**7**

5**0**

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

3

2

34

24

27

50

12

13

**2. Comparing two algorithms** (5 points)

|  |  |
| --- | --- |
| Algorithm A | Algorithm B |
| Time eff.: | **Time eff.:** |

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 .

1. **Selection sort**

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

1. **Insertion sort**

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

1. **Merge Sort**

The number of comparisons is always the same.

**4. Swap sort** (10 points)

1. **Best case**

Already sorted array: (smallest to biggest value).

We always have to do comparisons.

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

The overall time efficiency would be

1. **Worst case**

Array sorted in inverse order (biggest to smallest value)

We always have to do comparisons.

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

The overall time efficiency would be

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

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

First iteration: n-1 comparisons

Second iteration: n-2 comparisons

…

Last iteration: 1 comparison

1. **Time efficiency**

The number of comparisons is

The number of moves is < 2n. (the important thing here is that it’s < n2)

Therefore, the time efficiency of the method is:

1. **Alternative solution**

Merge sort: nlogn

Go through array and increase counter: n or modified merge sort

Public static int findMode(int[] arr){

Sort::mergeSort(arr);

// init the mode

Interger mode = arr[0];

Integer modeFrequence = 0;

Integer tempFrequence = 0;

for(integer i = 0; i<arr.length-1; i++ ){

if(arr[i] = arr[i+1]){

tempFrequence++;

}

elsif(tempFrequence>modeFrequence){

modeFrequence = tempFrequence;

mode = arr[i];

tempFrequence = 0;

}

else{

tempFrequence = 0;

}

}

}

1. **Time efficiency**

Nlogn

**6. Practice with reference** (10 points)

1. 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 |

1. Java code fragment

…

y.prev.next = x;

x.next = y;

x.prev = y.prev;

y.prev = x;

…