**Problem set 2**

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

**I. Written problem**

**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.:** |

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?

1. **Selection sort**

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

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

1. **Insertion sort**

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

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

1. **Merge Sort**

For the given array:

6

5

4

3

2

1

5

4

2

1

5

5

4

3

2

1

5

5

4

3

2

1

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)

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**

|  |
| --- |
| 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;  } |

1. **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)

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;

…

**II. Programming problem**

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

Unordered arrays:

|  |  |
| --- | --- |
| 1000 items | 2000 items |
| Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 5.54.15 PM.png | Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 5.54.45 PM.png |
| 4000 items | 8000 items |
| Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 5.55.01 PM.png | Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 5.55.18 PM.png |
| 16000 items |  |
| Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 5.59.34 PM.png |  |

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(n2) 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 |
| Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 6.04.29 PM.png | Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 6.04.49 PM.png |
| 4000 items | 8000 items |
| Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 6.05.06 PM.png | Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 6.05.22 PM.png |
| 16000 items |  |
| Untitled:Users:nr52:Desktop:Screen Shot 2011-10-04 at 6.05.44 PM.png |  |

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 n2/2 algorithm, therefore it is a O(n2) 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.