Joo-nam Kim

README – Squaresort

- This assignment was somewhat unpredictable in terms of amount of time it would take me to finish as finding the right algorithm may be more time consuming than actually implementing it.
    - It took me full 6 and a half days, spending on average three to four hours a day.

Goal : In a 2D array, every element needs to get sorted such that every element to its right and below has to be either greater if sorting numbers or alphabetically sorted if sorting strings.

ALGORITHM

I applied the well-known bubble sort algorithm.

1. Sort all the elements in the arrays within an outer array (rows) in an ascending order.
2. Sort vertically column by column.

How this works :

When we first sort rows, we are guaranteed to have the smallest values increasing from left to right. And, when we sort columns of the row-sorted array, out of the smallest values, they are once again sorted from top to bottom, which guarantees us to always have the smallest value at the top-left position and the largest value at the bottom-right position.

Example)
An unsorted array (5 x 5)
```
14 | 11 | 2 | 4 | 22 |
16 | 21 | 21 | 17 | 20 |
10 | 18 | 6 | 21 | 10 |
13 | 7 | 11 | 25 | 18 |
15 | 5 | 11 | 25 | 23 |
```

1) Sort every row in the 2D array using bubble sort.
```
2 | 4 | 11 | 14 | 22 |
16 | 17 | 20 | 21 | 21 |
6 | 10 | 10 | 18 | 21 |
7 | 11 | 13 | 18 | 25 |
5 | 11 | 15 | 23 | 25 |
```

2) Sort columns

```
2 | 4 | 10 | 14 | 21 |
5 | 10 | 11 | 18 | 21 |
6 | 11 | 13 | 18 | 22 |
7 | 11 | 15 | 21 | 25 |
16 | 17 | 20 | 23 | 25 |
```

IMPLEMENTATION AND LOOP INVARIANTS

Applying bubble sort, I made two separate blocks of triple nested loops.

1) For sorting elements in every row :

```
for (i = 0; i < people.length; i++){
      for (j = 0; j < people[0].length - 1; j++){
            for(k = 0; k < people[0].length - 1; k++){
                  if (comp.compare(people[i][k], people[i][k + 1]) > 0){
                        Person temp = people[i][k];
                        people[i][k] = people[i][k + 1];
                        people[i][k+1] = temp;
```

- repeatedly, compare and swap as necessary.

Invariant for the inner-most loop :

- Comparing and necessary swapping are expected to be done.
- for all row indices (first) < i and for column indices (second) < k, every element at [first][second] <= [first][second + 1]

Invariant for the second loop :

- Row(s) < j must have been sorted.
- for all l < I, m < j, element at [l][m] <= [l][m + 1]

Invariant for the last loop :

- All the elements in every row must have been sorted horizontally.
- for all i < array.length and for all j < array[0].length - 1, array[i][j] <= array[i][j + 1]

Exit condition : i must be done looping; assert i >= array.length

2) For sorting elements in each column :

```
for (i = 0; i < people[0].length; i++){
      for (j = 0; j < people.length; j++){
            for(k = 0; k < people.length - 1; k++){
                  if (comp.compare(people[k][i], people[k + 1][i]) > 0){
                        Person temp = people[k][i];
                        people[k][i] = people[k + 1][i];
                        people[k + 1][i] = temp;
```

Joo-nam Kim

Invariant for the inner-most loop :

- Column-wise comparing and necessary swapping are expected to be done.
- For all indices (first) < k and for all indices (second) < i, people[first][second] <= people[first][second + 1]

Invariant for the second loop :

- We are expecting all the columns be sorted up to this loop's last iteration.
- for all m < k, l < I people[m][l] <= people[m + 1][l]

Invariant for the last loop :

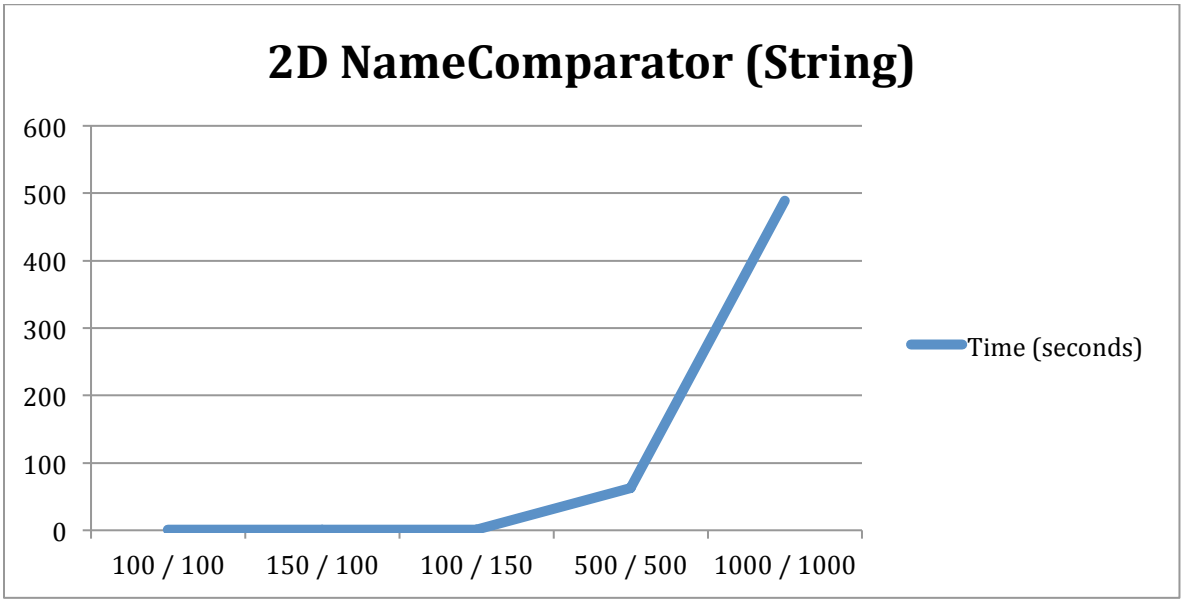
- All the elements in every column must have been sorted vertically.
- for all i < column and for all j < row, array[i][j] <= array[i + 1][j]

Exit condition : assert i >= number of columns.

TIME COMPLEXITY ANALYSIS

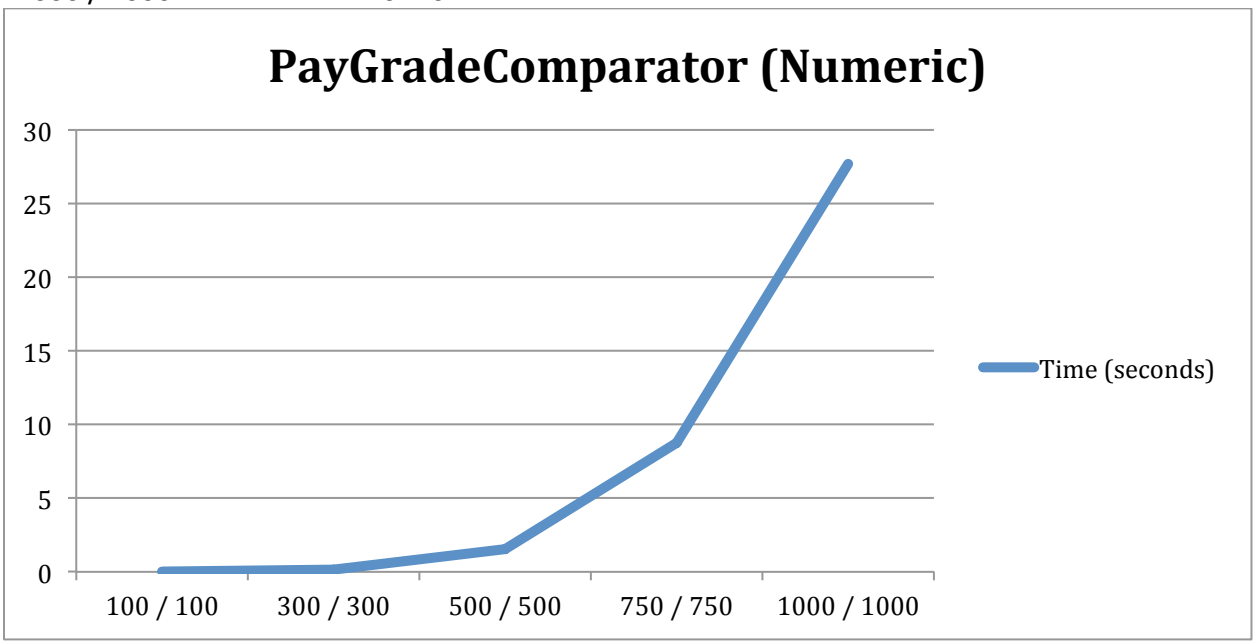
** Numbers on X-axis represent (row / columns) and Y-axis represents time taken.*

Sorting 2D array (Name Comparator	
Row / Column	**Time (seconds)**
100 / 100	0.57713975
150 / 100	0.884221
100 / 150	1.321228
500 / 500	62.458171
1000 / 1000	488.735369



Joo-nam Kim

2D array (Number Comparator)	
Row / Column	**Time (seconds)**
100 / 100	0.019009
300 / 300	0.17163
500 / 500	1.522714
750 / 750	8.752291
1000 / 1000	27.6776



- I implemented my algorithm with two *separate* blocks of triple nested loops.
  - In this case, time complexity depends on the more significant block of loops.
    - Since we can have "m x n" size arrays, if we were to express time complexity, it would depend on 'm' and 'n'.

```
for (i = 0; i < people.length; i++){
     for (j = 0; j < people[0].length - 1; j++){
          for(k = 0; k < people[0].length - 1; k++){
```

This block can be expressed as : m * n * n => **O(m*n^2)**

```
for (i = 0; i < people[0].length; i++){
     for (j = 0; j < people.length; j++){
     for(k = 0; k < people.length - 1; k++){
```

This block can be expressed as : n * m * (m – 1) => **O(n*m^2)**

Simply, if we have (n > m) the first block would dominate whereas if we have (m > n) the second block dominates, giving us O(m*n^2) and O(n*m^2) respectively.

However, assuming ‘n’ and ‘m’ are not apart by too much, we can say the algorithm has time complexity of O(n^3) and surely when m == n.