CIT594 HW2 **Jincheng Cao**

1. How long did the assignment take?

I was expecting 4-6 hours of work. The assignment actually took me 4 hours for coding, and it took me much longer to finish this readme file.

2. Algorithm-selection sort:

In my squaresort method, I used three loops to sort each row in an ascending order. For each row, a[i] is exchanged with smallest entry in a[i+1….n], i is from 0 to n-1. This part uses ~n^2/2 compares and ~n exchanges to sort an array of length n. There are m rows, so running time is O(m*n^2). Then I used three loops to sort each column in an ascending order. This process is similar to sorting rows. So running time is O(n*m^2) since there are n rows and each row uses ~m^2 compares. The running time for squaresort method is O(m*n^2 + n*m^2). Assertions are commented out when measuring running time. The algorithm should work because after sorting each row, a[i][k] < a[i][k+1] holds for i = 0…n. When I started sorting columns, new a[i][k] value could only be smaller compared to old values. Let's use a 2*2 sub matrix as example, after sorting rows: the matrix is:

a[i][k] a[i][k+1] (a[i][k] <a[i][k+1])
a[i+1][k] a[i+1][k+1] (a[i+1][k]< a[i+1][k+1])

There could only be 4 possibilities after sorting columns, which are:

1. matrix stays the same
2. a[i+1][k] a[i][k+1]
   a[i][k] a[i+1][k+1]
   Since a[i+1][k] < a[i][k] and a[i][k]< a[i][k+1], a[i+1][k] < a[i][k+1] holds, since a[i][k+1] < a[i+1][k+1] and a[i][k] < a[i][k+1], a]i][k] < a[i+1][k+1] also holds. So the matrix is correctly sorted.
3. a[i][k] a[i+1][k+1]
   a[i+1][k] a[i][k+1]
   The proof is similar to part2
4. a[i+1][k] a[i+1][k+1]
   a[i][k] a[i][k+1]
   We've already known that a[i][k] < a[i][k+1] and a[i+1][k]< a[i+1][k+1]

3. Loop invariants:

I used loop invariants in the process of sorting each row and each column. For example, when sorting each row, loop invariant is:

`for(int m = 0; m < i - 1; m++) assert comp.compare(people[t][m],people[t][m+1]) <= 0;` This invariant guarantees that a[0]~a[i-1] is sorted, which means the program runs well up to a[i]. When finished sorting each row, I use "`assert I >= N`" as exit condition. Invariant + exit is:

`for(int m = 0; m < N - 1; m++) assert comp.compare(people[t][m], people[t][m+1]) <= 0;` to guarantee the sorting algorithm for the entire row works well. Loop invariants for sorting columns are similar. After sorting rows and columns, I use two loops to make sure the sorting algorithm works for the 2D array:

```
for(int i = 0; i < people.length -1; i++){
        for(int k = 0; k < people[0].length -1; k++){
                if(people.length >= 2 && people[0].length >= 2)
```

```
                                        assert (comp.compare(people[i][k],
people[i+1][k]) <= 0)&& (comp.compare(people[i][k], people[i][k+1]) <= 0);
                                    }
                }
```

4. Big-O analysis

As I mentioned before, the algorithm took O(mn^2 + nm^2) running time to finish. I measured some running time data for pair of m, n using three comparators.

m	n	ncomp	pcomp	ecomp
1	1	4000	4000	4000
10	1	41000	26000	25000
1	10	39000	26000	26000
2	3	23000	17000	18000
3	2	29000	21000	19000
5	5	59000	34000	34000
6	2	38000	24000	23000
7	3	61000	33000	31000
3	7	57000	32000	30000
10	10	432000	232000	215000
10	15	723000	357000	350000
15	10	798000	384000	343000
10	3	83000	46000	46000
3	10	83000	46000	47000
9	5	129000	69000	67000
5	9	124000	63000	63000
8	8	243000	95000	89000
10	6	249000	99000	87000
6	10	237000	92000	85000
5	8	116000	58000	56000
8	5	115000	55000	54000
7	7	184000	70000	69000

A. Even when m and n stay the same, the elapsed time for each run varies a lot (±10000 in general), the data presented is about the average elapsed time for each run.
B. NameComparator usually takes longer time than the other two comparators(about twice). It's because in NameComparator there's one more if condition to check if last names are the same. PayGradComparator and EmployeeIdComparator usually consume similar running time because they both contain one return statement, sometimes EmployeeIdComparator takes even less time.
C. O(m*n^2 + n*m^2) = O(mn(m+n)). So I picked (m,n) and (n,m) pairs, such as (2,3) and (3,2), (7,3) and(3,7), (10,15) and(15,10), (10,3) and(3,10), (9,5) and (5,9), (6,10) and (10,6), (5,8) and (8,5). Each pair should take similar

time using each comparator. In the experiment, most pairs take similar time. Considering elapsed time for each run varies a lot, the running time difference for each pair is still within reasonable range.

D. Theoretically, 3D plot of t(x,y) = x*y^2 + y*x^2 should look like the following graph.

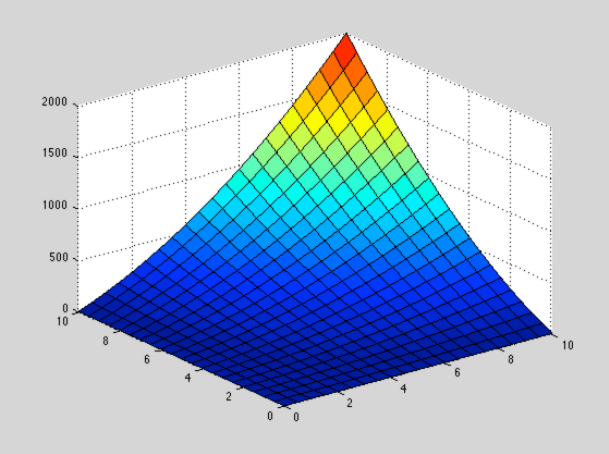

Fig1. 3D plot of function t(x,y) = x*y^2 + y*x^2

In our case, the running time vs. x and y plots for three comparator are a little different. However, I believe given enough data points, our plots should look similar to Fig1.

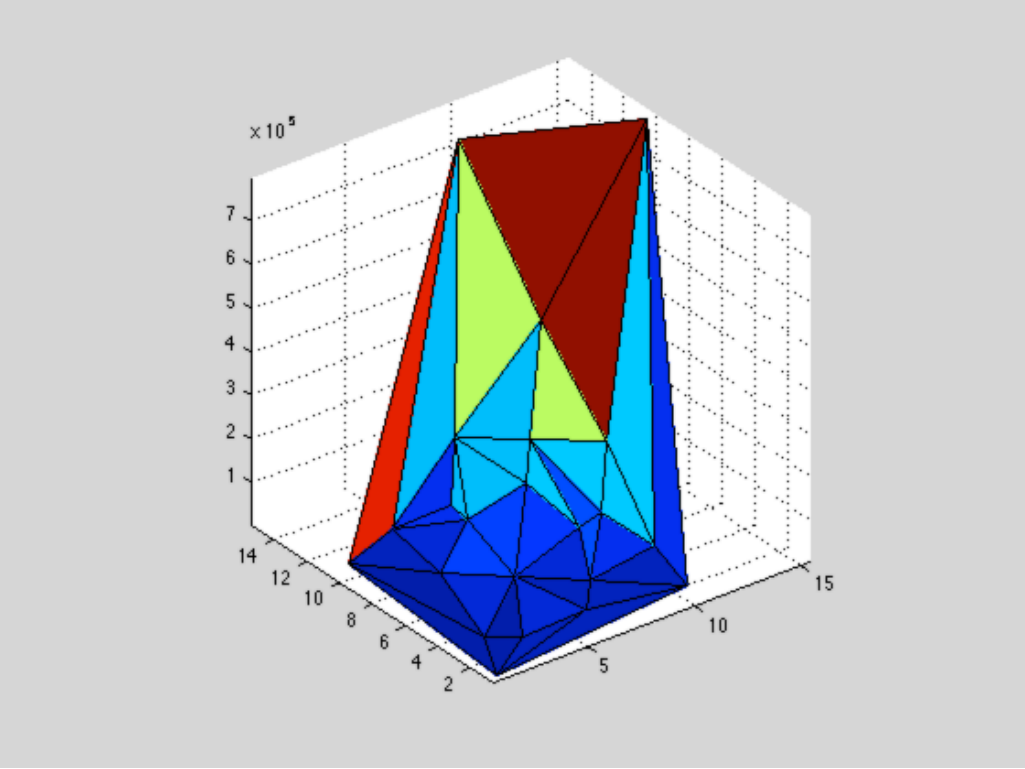

Fig 2. Running time vs. m, n 3D plot for Namecomparator

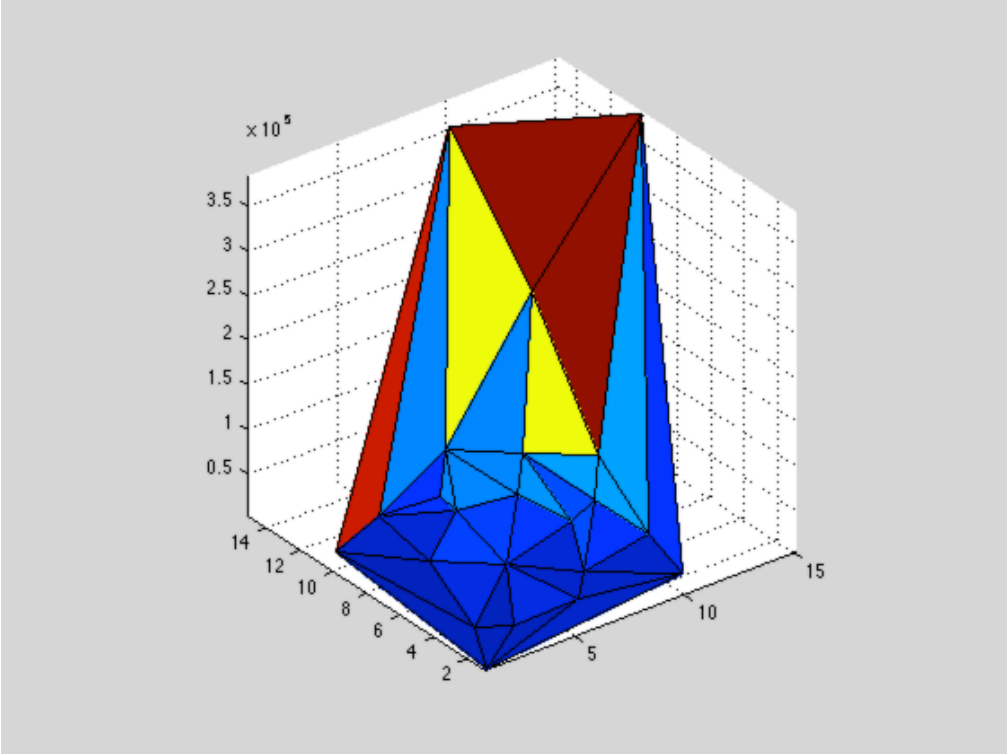

Fig 3. Running time vs. m, n 3D plot for PayGradecomparator

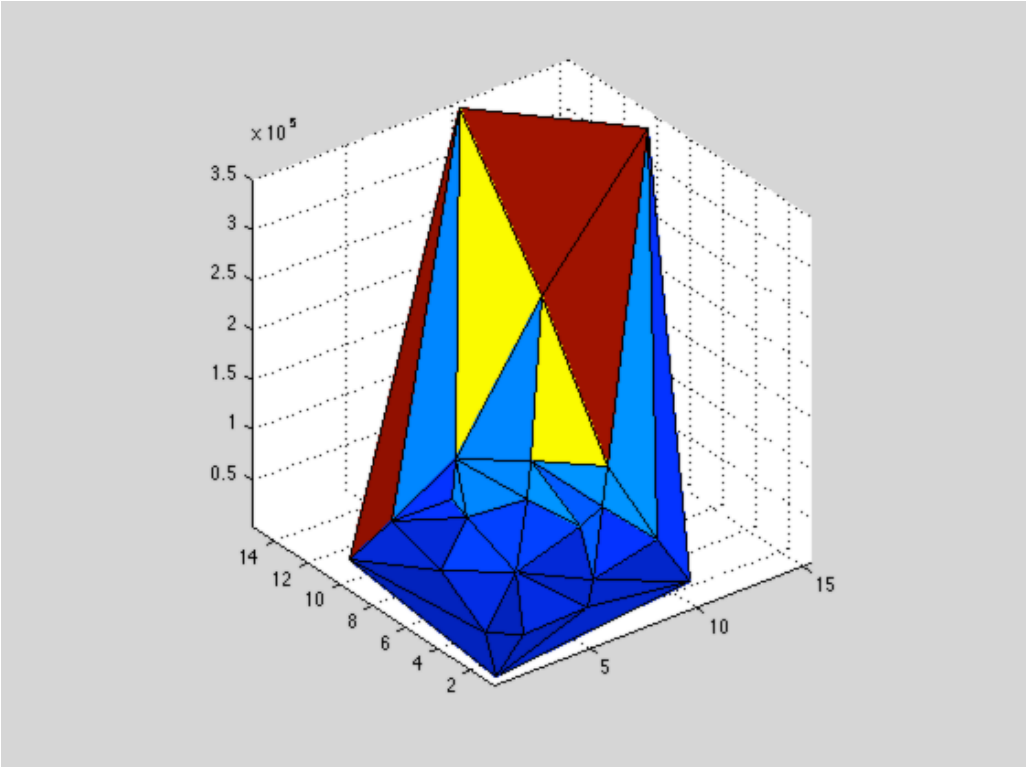

Fig 4. Running time vs. m, n 3D plot for PayGradecomparator