Youssef Naguib
CS261
Solutions for hw-1 questions 3-5

3.
My program is correct in finding if at least half of the total snowfall fell within a three day interval. After the input is processed into an array (a native python list), the check_snow() algorithm follows these steps:
Step 1 - calculate and store half of cumulative snowfall as a variable snowsum (constant time)
Step 2 - set a counter variable to 0 (constant time)
Step 3 - iterate through the array while the pointer is less than the #of days – 4 (linear),
and access values in array at index counter, and index of counter+3 (accessing an python list with an index is a constant time operation)
step 4 - if at any point, the value at index pointer+2 – index pointer > snowsum, print YES
- otherwise if this never occurs print NO (Constant time)
This algorithm will always produce a correct output, as it examines every possible three day interval in the array of days. Although the while loop in step 3 is a linear operation, it does iterate through n values of the array. The loop ends when the counter variable reaches (n-3),
in order to keep the algorithm from accessing an out of bounds element of the list. So we know this algorithm's time complexity can be described as (n-3) + a finite number of constant time operations. (n-3) will always be less than (n), so we can say this algorithm is tightly upper bounded by n, or in other words o(n).

4.
My approach to this problem was simple. First I wanted a way to store an instance of a person, so I created a person class. I read in the input and stored people in two groups. Group 1 can be the professors in this example, and group 2 can be the students, although it will not matter in the end which group is what. My algorithm starts now. First I run the Gale-Shapley algorithm with a professor-optimal output. The Gale-Shapley algorithm runs in O(n^2) as its worst case. I store the matchings returned in a native python list as tuples(O(1)). I repeat the Gale-Shapley algorithm on the original unmatched groups, however this time the output will be student-optimal (O(n^2)). I store the matchings in a native python list (O(1)). Lastly, I iterate through the first array of matchings (O(N)) and if any pair is found in the other array of matchings, the program prints "NO". If no identical matchings are found the program prints "YES". This algorithm's complexity is O(n^2). The Gale-Shapley algorithm will guarantee to find a stable matching for any problem instance. Since there are only 2 groups in any given problem, the algorithm can either favor the preferences of group 1 or group 2. Comparing these two stable matchings guarantees solving the problem of determining if there exists two completely different stable matchings.

5.

Uniform array size	Insertion sort time(sec)	Merge sort time(sec)	Bucket sort time(sec)
100	0.0005528926	0.0008130073	0.0002369880
1000	0.0570511817	0.0066938400	0.0087957382
10000	5.8935210704	0.0898039340	0.0197260379
100000	>180 (over 3 min)	2.6301209926	0.2389969825

Gaussian array size	Insertion sort time(sec)	Merge sort time(sec)	Bucket sort time(sec)
100	4.2915344238	0.000601053	0.0001938343
1000	0.0002958774	0.005642890	0.0018701553
10000	0.0029759407	0.077114105	0.0188741683
100000	0.0295481681	2.423306941	0.1964731216

Implementing these different sorting algorithms was very interesting and informative. It was no surprise to me that insertion sort took a longer time to run the larger the size of the uniformly distributed array, however I was a little confused as to why the contrary was true for the Gaussian distribution array. Merge sort and Bucket sort on the other hand were consistent. When it came to both types of distributions, a larger array size meant a slightly longer execution time. In the end, it is obvious that Bucket Sort is the superior sorting algorithm of the three.