Jonathan Tso
HW 2
CS 251

1A.

This would be $\Theta(n^2)$. This is because the double for-loop would effectively be a summation from 0 toward n. Every time we increase i, we iterate from Ø and add up to i. This iterates up to n times since the outer for loop is from Ø to n. The loop thus is a summation that approaches $n(n-1)/2$ at maximum $i=n$.

1 B.

This runs in $\Theta(\log(n))$ time. This is because the inner k is $k = k \cdot 2$ meaning we increase in powers of 2 each time. Thus, we only do more iterations as we do 2^n which is $\log(n)$ number of loops for all increasing n.

1C.

This is a mixture of 1A and 1B, where we did each inside for loop individually. Thus, we then have the runtime to be approximately the sum of each individual, where we have n^2 and then $n \log n$. Because the rate of n^2 is greater, we reduce this down to n^2, thus:
$\Theta(n^2)$

1D.

In the case of all even numbers, we run into an inner for loop such that it resembles $O(n^2)$. In all odd cases, we run $O(1)$ time because we just do x--; Then, half the time we do a double for loop and the other half we just run x--, We can extrapolate that it then will be that it is $(n/2) \cdot (n)$ which will be $\Theta(n^2)$ time.

1E.

The number of times that the i%n == 0 if statement will execute is n times, since it's effectively every multiple of n and we have n of them. Then, the inner for loop is run in $\Theta(n)$ time. Thus, our total runtime is $\Theta(n^2)$.

1F. The inner for loop will run at worst case as $O(n)$ time. The outer for loop is run on $O(\log n)$ time. When we put these together, we get $\Theta(n \log n)$ time.

1G. The inner loop here is dependant on the outer loop in that $i < n$, then run inner loop and $i = i*2$. We know that in the maximum scenario, $i < n$ and in 1F, we have considered then that the upper limit must be $\Theta(n \log n)$. Additionally, we know the outer loop mandates we are running in at least $\log(n)$ time.

Looking at an example where $n = 8$

1st loop	$i = 1$	$j = 0$	$x = x + 1$.
2nd loop	$i = 2$	$j = 0$	$x = x + 2$.
3rd loop	$i = 4$	$j = 0$	$x = x + 4$

The summation is then $x = x + 7$.

The worst case scenario is when we run into where $i = n-1$, then it would become $n \cdot \log(n)$. Then the runtime is $\Theta(n \log(n))$.

2A. Here, we know limit = 16. The inner if command only runs when we hit i==limit. In all other instances, we run in $O(1)$ time since we just do i++. When we hit the limit, we add from 0 to limit, and then multiply limit by 2. In this sense, we are only running the worst case scenario runtimes wherever we have it that $i = \text{limit} \cdot 2^{k-1}$ where $\text{limit} \cdot 2^{k} < n$.

When we look at if $n = 17$, we only hit i==limit once, and then adding limit # times. When we hit at if $n = 65$, we hit i==limit three times (16, 32, 64), adding limit # times each.

This means we iterate through the loop, it is similar to doubling the size of a dynamic array. This updates once we hit the limit and doubles in size. Then, we double only on $16 \cdot 2^{\log n}$ which, after removing constants, correlates to $\Theta(n)$.

2B. This is similar to adding a fixed amount to a dynamic array when you need more space. Unlike 1A, we are allocating more space more frequently (whenever we hit limit, which is every +8). At this point, we are constantly adding the total amount up to limit. Then, this means we add 16, then 24... which runs similarly to when we simply add up to $N(N-1)/2$. Thus, this runs in a similar fashion, which would be $\Theta(n^2)$.

Tre outside

3.

For this, we can break it down into three sections. 1, the initialization, 2, passing out all the beans, and 3, tallying up the beans per person. For the first part, we can initialize with n times. To pass all the beans out, this will take 10n times. For printing out the beans per person, we will get a total of 10n times, the number of total beans. Thus, we have 21n as run time, which corresponds to $\theta(n)$ runtime.

4. Given the set of starting and ending times, we parse it into two arrays. One for all start times, one for all end times.

Afterwards, perform a merge sort on each of the two arrays to provide a sorted start times array and sorted end times array.

Now we run comparisons starting at index =0 for each of the two arrays. If end time > start time, overlap++, and we update start index+1. If end time = start time, overlap stays the same and we update end index+1 and start index+1. If end time < start time, overlap -- and end index+1. When we run out of start times, the comparisons finish.

We have a variable maxOverlap that begins at 1. If overlap > maxOverlap, update maxOverlap to be equal to overlap. This will give us the maximum overlap of all of the intervals.

This function will be type int, returning type int maxOverlap.

This runs in O(n log(n)) time since the time separating at the beginning and the time comparing is O(n) each, while the merge sort is n log(n). Because n log(n) > n, we can omit the n for O(n log(n)) time.

5. Mystery

n	runtime
1000	1.8
2000	9.446
100	0.0
200	0.03
500	0.300
1500	4.970
3000	27.600
300	0.080
2500	17.410
5000	97.760
10000	547.370

When doubling n, such as from 5000 to 10,000, we can see an approximate rise by a factor of 5 in runtime. Similarly, this can be seen in other doubling results, such as 1000, 2000.

This leads to the approximation that doubling n results in about 5 times the runtime, and so it would be

$$\frac{n^D}{(2n)^D} = \frac{\cancel{1}}{5\cancel{1}} = \frac{1}{5}$$

$$\frac{\cancel{n^D}}{2^D\cancel{n^D}} = \frac{1}{5}$$

$$\frac{1}{2^D} = \frac{1}{5} \qquad 5 = 2^D$$

$$\log 5 = D \log 2$$

$$\frac{\log 5}{\log 2} = D$$

$$D \approx 2.32$$

6.

A.

f[]	2	4	6	8	10	12	14	16	18	20
g[]	17	16	15	14	13	12	11	10	9	8
max	17	16	15	14	13	12	14	16	18	20

↑ This is the minimum of the maximum values

we know that because f[], is smallest to largest and g[] is largest to smallest, the ends and beginnings of each array will hold the largest maximums. Thus, we are more interested in the centre pieces. We can approach in a binary-search fashion, where we are interested in only the sides that have max (f,g) to be less than our current max (that we are observing).

```
int minimax (double f[] ,double g[] ,int n) {
        int   lowIndex  =0;
        int   highIndex  = n-1;
        int   val        = (lowIndex +highIndex)/2;
    if (n<=0) {
        return Ø;
    }
    while (lowIndex <= highIndex) {
            if (f[val] == g[val]) {     return val;}
        else if (f[val]< g[val]) {
                    lowIndex     = val+1;
                  }
        else if (f[val] > g[val]) {
                    highIndex = val-1;
                  }
            val = (lowIndex + highIndex)/2;
    }
    return val;
}
```