HW4

1. **Order Statistics**
   a. True.
   b. False. $\lim_{n\to\infty}\frac{8n^2+32n+9}{n^2}=8\neq 0$
   c. True.
   d. False. d~f L'Hôpital.
   e. True.
   f. True.
   g. False. Obvious incremental.
   h. True.
   i. False.
   j. True.
2. **Logarithms**
   a. True. Since $\log\left(n^2\right)=2\log\left(n\right)$, we can always calculate the limits for each statement.
   b. False. $\lim_{n\to\infty}\frac{\log\left(n^2\right)}{\log\left(n\right)}=2\neq 0$
   c. True. Same as a.
   d. False. $\lim_{n\to\infty}\frac{\log\left(n\right)}{\log\left(n^2\right)}=0.5\neq 0$
   e. True. Since a and c are true, then e is true.
3. **Resizing an Array**
   a. Allocate each time

   For an allocate-per-increase approach, when we already have k numbers in our array, for each coming in number, we need to do the following steps:

   I. 1: Allocate one spot for the new number.

   II. 2*k: Copy the existing k numbers from old array and paste to new array.

   III. 1: write a new number to the new spot.

   It takes 2+2k operations in total for each step. Therefore, the overall complexity is:

$$\sum_{k=1}^{n}2k+2=n\left(n+3\right)\sim \mathrm{O}\left(n^2\right) \ ,$$

   Which result in a non-constant time.

   b. Allocate every 100 times

   This method is pretty similar to question a, despite that for this time we allocate 100 spots each time. We can just view every 100 numbers as a chunk, and do it like what we've done in a. Assume we already have 1+100*k numbers in our array. The steps describe as follow:

I. 1: Allocate 1+100*(k+1) spot for the new chunk.
II. 2+200*k: Copy and paste the existing 1+100*k numbers into the new array.
III. 100: Write the following 100 numbers chunk into the newly allocated spots.

It takes 103+200*k operations for each step. Therefore, the overall complexity is:

$$\sum_{k=0}^{\frac{n}{100}} 200k + 103 = \frac{n(n+103)}{100} \sim \mathrm{O}(n^2)$$

As we can see above, although we reduced our constant to 1%, the big-o complexity does not change. Furthermore, the 100-length chunk might waste some space when allocating.

c. Doubled allocation

In order to make our expression easier, we can just assume the worst case, where we just got one more element before we double our array size. Therefore, we can assume a total n=2^m+1. Assume we are at the k step with a fully filled 2^k-length array, then we need to do the following steps to enlarge the array

I. 1: Allocate a 2^{k+1} size array.
II. 4^k: Copy and paste the 2^k elements into the new array.
III. 2^k: Write the following 2^k elements into the newly allocated spots and k++ for next step.

It takes 3*2^k+1 operations for each step. However, since we only have floor($\log_2$(n))+1 steps, the complexity is:

$$\sum_{k=0}^{floor(\log_2(n))+1} 3\cdot 2^k + 1 = 3\cdot\frac{1-2^{\log_2(n)}}{1-2} \sim \mathrm{O}(n)$$

This time our algorithm efficiently reached what we've expected: the average constant time.

4. Car Finding

a. If we magically figure out the direction (e.g. the car has a flashlight that beam through fog), we can just walk n distance directly towards the car.

b. The best case is n as in a. But the worst case is ∞, since if we walk towards the wrong direction, we'll never reach it. This method does not always work, only have half-half chance to work.

c. For this method, if we want to check k cars for current step and return to the office door, we'll need to walk 4*k cars each step. That is: (if left first) left k, right k to the office, right k, left k to the office. Thus before we can reach the n spot, the steps needed in total are:

$$\sum_{k=1}^{n} 4k = n(2n+2) \sim \mathrm{O}(n^2)$$

Although we might reach our car on the left and save 3*n steps, it does not affect the overall complexity of big-o notation.

d. There is a similar way to find our car to the method in Problem 3. For the details please check the description in section f. Suppose we go to left first. First go left for 1 car, then right for 1+2 cars, then left for 2+4 cars, then right for 4+8 cars, etc. For step k>1 we go to the opposite direction with 2^{k-1}+2^k steps, until we reached our car. The best case for this method is to reach our car when we reach our car on the left first place after the repetitive cars, and the worst lies on the next place of a furthermost place of a step.

e. Since we will always go 3*2^{k-1} cars towards each side, and $\lim_{k\to\infty} 2^{k-1} \to \infty$. We'll at least reach the n-th place of each side giving large enough k. Therefore, we can find the car with this algorithm.

f. As the algorithm described above, we will reach our car either on $\log_2(n)$ or $\log_2(n)+1$ step. The constant here does not affect the final big-o complexity. The total at most steps that we'll walk through are

$$1+\sum_{k=1}^{\log_2(n)+1} 2^{k-1}+2^{k} =1+\sum_{k=1}^{\log_2(n)+1} 3\cdot 2^{k-1} \approx 3\cdot\frac{1-2^{\log_2(n)+1}}{1-2} \sim \mathrm{O}(n)$$

We'll find that although we still walked through a lot of repetitive cars, we somehow take advantage of the 'doubled' approach to improve the efficiency of this algorithm, just like in Problem 3.