Math 590 Homework 5 Fall 2019

Due: Thursday, October 17

1. **Order Statistics**

   For the following, indicate True or False:

   (a) $2n^3 - 8n^2 + 32n + 9 \in O(n^3)$ TRUE

   (b) $2n^3 - 8n^2 + 32n + 9 \in o(n^3)$ FALSE

   (c) $2n^3 - 8n^2 + 32n + 9 \in \Omega(n^3)$ TRUE

   (d) $n^p \in O(e^n)$, where $p \in \mathbb{R}$ and $p \geq 0$ TRUE

   (e) $n^p \in o(e^n)$, where $p \in \mathbb{R}$ and $p \geq 0$ TRUE

   (f) $e^n \in O(n^p)$, where $p \in \mathbb{R}$ and $p \geq 0$ FALSE

   (g) $\sqrt{n} \in O(n)$ TRUE

   (h) $\sqrt{n} \in O(1)$ (side note: $O(1)$ is called 'constant time') FALSE

   (i) $\log n \in o(n^p)$, where $p \in \mathbb{R}$ and $p > 0$ TRUE

   (j) $n^p \in o(\log n)$, where $p \in \mathbb{R}$ and $p > 0$ FALSE

   It may help to know an additional way of defining little-o: We say $f \in o(g)$ if

   $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

2. **Logarithms**

   What is the relationship between $\log(n)$ and $\log(n^2)$? Indicate which of the following are True (provide a sentence explaining why your choices are correct):

   (a) $\log(n^2) \in O(\log n)$ TRUE

   (b) $\log(n^2) \in o(\log n)$ FALSE

   (c) $\log(n^2) \in \Omega(\log n)$ TRUE

   (d) $\log(n^2) \in \omega(\log n)$ FALSE

   (e) $\log(n^2) \in \Theta(\log n)$ TRUE

3. **Factorials**

True or False:

$$(n+1)! \in \Theta(n!)$$

You must provide an explanation of your answer.

First, if we use the limit definition, we can easily see that

$$\lim_{n\to\infty} \frac{n!}{(n+1)!} = \lim_{n\to\infty} \frac{1}{n+1} = 0,$$

and so it must be that $(n+1)!$ grows faster than $n!$, i.e., $(n+1)! \in \omega(n!)$. So the statement is false.

If we wish to use the definition with k_1, k_2, and N, then let's write a proof by contradiction to show that the statement is false. Assume by way of contradiction that $(n+1)! \in \Theta(n!)$. Then it must be the case that there exists $k_1, k_2 > 0$ and $N \in \mathbb{N}$ such that

$$k_1(n+1)! \leq n! \leq k_2(n+1)!, \quad \text{for all} \quad n \geq N.$$

Let's focus on that first part of the inequality,

$$k_1(n+1)! \leq n!, \quad \text{for all} \quad n \geq N.$$

Now it's time for our malicious adversary. Say we pick some value for k_1 in an attempt to make this expression true. What value of N would our devil pick to break the inequality (in this proof, the devil is helping us find the contradiction)?

Well, note that our inequality is equivalent to

$$k_1(n+1) \leq 1, \quad \text{for all} \quad n \geq N.$$

So if we set a value of k_1, then we can solve for n to find that

$$k_1(N+1) = 1 \quad \Longrightarrow \quad N = \frac{1}{k_1} - 1.$$

So for any $n > N$, the inequality will fail. This is the contradiction, which proves

$$(n+1)! \notin \Theta(n!).$$

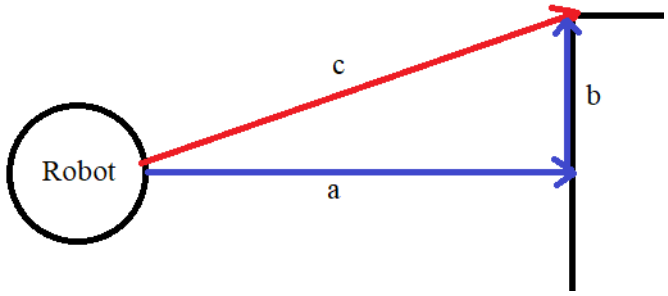
4. **Robots and Triangles**

When assigning robots to perform automated tasks, an important problem that often needs to be addressed is how to route the robot around obstacles in its path. There are two broad approaches to this problem:

Approach #1: scan the room in advance, and plot a route that goes around the obstacles (a smart-car approach).

Approach #2: walk towards the destination, and simply go around any obstacles that the robot runs into (a roomba approach).

If the obstacle is some rectangular table in the way, then the following image shows the two different approaches (#1 in red and #2 in blue), with the distances labeled a, b, and c.



Prove that, in terms of total distance traveled, these two approaches are computationally equivalent, i.e., prove that $(a+b) \in \Theta(c)$ by finding positive real numbers k_1 and k_2 such that

$$k_1 \cdot c \leq a + b \leq k_2 \cdot c.$$

You may assume that the obstacle has flat sides, and that the robot collides with the obstacle perpendicular to its surface.

First, we use the triangle inequality, which tells us that a leg of a triangle cannot be longer than the sum of the other two legs. In other words,

$$c \leq a + b.$$

Now note that c is the longest leg of the triangle, so we know that $c \geq a$ and $c \geq b$. But that means

$$a + b \leq 2c.$$

So if we set $k_1 = 1$ and $k_2 = 2$, we have shown that

$$c \leq a + b \leq 2c, \quad \text{for all such } a,\ b,\ \text{and } c.$$

5. **Resizing an Array**

When we discuss algorithmic complexity, we assume that read/write times are constant. What we are really assuming is that the people who designed the memory management of our machines did a good job and guarantee us that all reads or writes will take constant time.

This is an easy promise to keep for fixed size data stored on the stack, but what about dynamically allocated memory stored in the heap?

- Allocating the initial array is easy, because its size is originally given by the user. This allows the machine to allocate the initial memory just like it would on the stack.
- Deallocating the final array is easy, because the size is tracked throughout computation. This allows the machine to deallocate the final memory just like it would on the stack.
- Resizing the array is more complicated...

Consider an initial array of size 1, which then has n objects appended to the end of it. For each of these writes to be an average constant time, we need the total complexity of appending n objects to be $O(n)$. Let's consider three different approaches. For each approach, calculate the complexity of appending n objects to the end of the array and determine if the approach will give us constant time read/write.

Answer: Before we begin, let's note that an append command that results in a resize for an array currently of size k will cost 1 allocation, $2k$ reads/writes for the copying, and 1 write for the new append. We should also count the 1 conditional check to see if the array has enough space, as well as the 1 arithmetic operation to calculate the new length. The cost is then $2k + 4 = 2(k + 2)$ for a resize of size k.

(a) Every time that the array runs out of space, its length is increased by 1. If the array is currently size k: we allocate space for a new array of size $k+1$, copy over the original k elements, then put the new object into the final $k + 1$ location.

When we go to do the n append operations, we will have to resize the array for each newly appending item. This means that we will resize the array at size 1, 2, 3, $\ldots$, $n - 1$. Each resize of size k costs $2(k + 1)$, resulting in the total cost of

$$2(1 + 2) + 2(2 + 2) + 2(3 + 2) + \cdots + 2(n - 1 + 2) = \sum_{k=1}^{n-1} 2(k + 2)$$

$$= 2\left(\sum_{k=1}^{n-1} k\right) + 2\left(\sum_{k=1}^{n-1} 1\right) = 2\left(\frac{(n-1)n}{2}\right) + 2(n - 1) = n^2 + n - 2 \in O(n^2).$$

This means that the average cost of an append was $O(n)$, which will not give us constant read/write times.

(b) Every time that the array runs out of space, its length is increased by 100. If the array is currently size k: we allocate space for a new array of size $k + 100$, copy over the original k elements, then put the new object into the $k+1$ location (leaving the locations $k + 2$ through $k + 100$ empty for now, but available for storing more elements in the future).

Now only $n/100$ resizes occur whenever the array's length passes a multiple of 100. So the overall cost would be (assuming that n is a multiple of 100 - if it is not, then this analysis would over-count the work by at most 99, which does not change the result)

$$2(1+2) + 2(101+2) + 2(201+2) + \cdots + 2((n-100+1)+2)$$

$$= \sum_{k=1}^{(n/100)-1} 2(100k+3) = 2\left(\frac{\left(\frac{n}{100}\right)\left(\frac{n}{100}-1\right)}{2}\right) + \frac{6n}{100} - 6 \in O(n^2),$$

which is still not enough to guarantee constant time read/writes.

(c) Every time that the array runs out of space, its length is doubled. If the array is currently size k: we allocate space for a new array of size $2k$, copy over the original k elements, then put the new object into the final $k+1$ location (leaving the locations $k + 2$ through $2k$ empty for now, but available for storing more elements in the future).

Here we will assume that n is a power of 2, so that $n = 2^k$. Then we will need to resize the array $k - 1$ times, which will occur when the array has lengths

$$1, 2, 4, 8, \ldots, 2^{k-1}.$$

Now we make an important observation:

$$1 + 2 + 4 + 8 + \cdots + 2^{k-1} = 2^k - 1.$$

So the total work done is

$$2(1 + 2 + 4 + 8 + \cdots + 2^{k-1}) + 4(k-1) = 2^{k+1} + 4k - 6$$

$$= 2n + 4\log_2 n - 6 \in O(n).$$

What if n wasn't a power of 2? Define m to be the smallest power of 2 that is larger than n, i.e.,

$$m = 2^k, \quad 2^{k-1} \leq n \leq 2^k.$$

We now make an important observation: $2n \geq m$. So we can see that

$$T(n) \leq T(m) = 2m + 4\log_2 m - 6$$

$$\leq 4n + 4\log_2(2n) - 6 = 4n + 4\log_2(n) + 4\log_2(2) - 6 \in O(n).$$

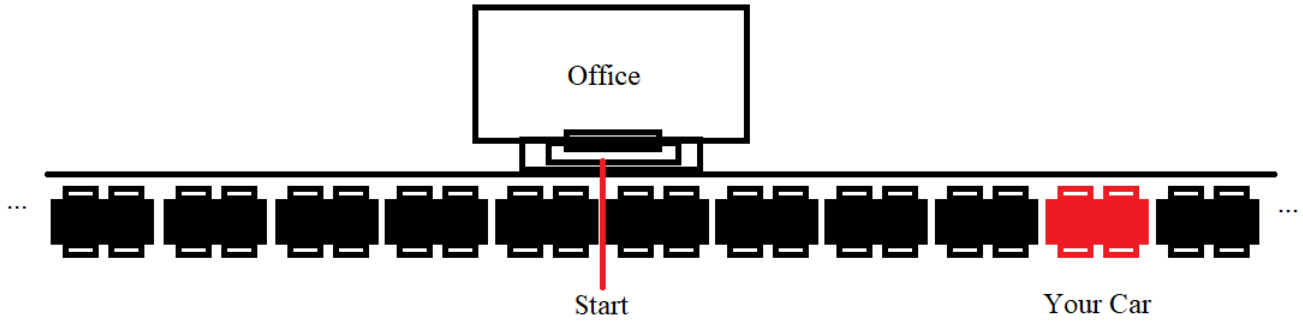
6. **Car Finding**

Let's say you work in an office building, with street parking in front of it. One day, you leave work and cannot remember where on the street you parked your car. Unfortunately, it is a foggy day and you can only see each car on the street when you are next to it. How should you find your car?

Specifics:

- You can only look at one car at a time, as you pass it.
- The cars are parked in single file along the 1D street.
- You do not know which direction your car is in.
- As far as you know, the parked cars go off to infinity in both directions.

An example image:



For the following questions, assume that your car is located exactly n cars away from your starting position. When asked how many cars you will pass, give the answer using big-O notation because we are concerned with the complexity of the different car finding algorithms.

(a) What is the minimum number of cars you would have to pass before reaching your car (assuming that your algorithm was 'perfect' and took you straight to your car)?

The car is n spots away, meaning that we will have to walk past n cars no matter what.

(b) What if you just tried walking in one direction? What is the best case? What is the worst case? Will this method always work?

The best case is that we chose the correct direction and reach our car after walking past n other cars. The worst case is that we chose wrong and never find our car. So this method fails half the time...

(c) What happens if you check 1 car to each side, then 2 cars to each side, then 3 cars to each side, etc? How many cars will you pass before reaching your car?

The last check we will make will be n cars to the correct direction. So, we will end up checking 1 car to the left, 1 to the right, 2 to the left, 2 to the right, ..., n to the left, n to the right (which may over-count by at most n). So the total cars checked is (noting that we have to walk back to the starting location before checking the other direction, which accounts for the extra factor of 2)

$$2(2+4+6+\cdots+2n) = 4\sum_{k=1}^{n} k = 2n(n+1) \in O(n^2).$$

Since we over-counted by at most n, our answer would still be $O(n^2)$.

Now, what if we misinterpreted the question and assumed that we would check 1 **new** car to the left, then 1 **new** car to the right, then 2 **new** cars to the left, then 2 **new** cars to the right, etc?
Well, let's say we are at a step where we want to check k new cars to the left. How many cars will we have to walk past in order to check k new cars? To have gotten to this step, we must already have checked 1 new car, then 2 new cars, then 3 new cars, ..., then $k-1$ new cars. So at this step, we have to walk past a total of

$$1+2+3+\cdots+(k-1)+k = \frac{k(k+1)}{2} = T_k$$

cars. The numbers T_k are called the triangle numbers. Now let's count how many cars we have to pass in total to reach our car located n cars away. Suppose that

$$T_{k-1} \le n \le T_k \quad \Rightarrow \quad \frac{(k-1)k}{2} \le n \le \frac{k(k+1)}{2}$$

$$\Rightarrow \quad \sqrt{1+8n} - \frac{1}{2} \le k \le \sqrt{1+8n} + \frac{1}{2} \quad \Rightarrow \quad k \in \Theta(\sqrt{n}).$$

Note that $2T_{k-1} > T_k$, and so $2n > T_k$. Then in the best case (where our car was to the left, the first side we check), we will walk a total of

$$2(2T_1 + 2T_2 + \cdots + 2T_{k-1}) + n.$$

In the worst case, we will walk a total of

$$2(2T_1 + 2T_2 + \cdots + 2T_{k-1}) + 2T_k + n.$$

Now, since we know that $n \le T_k \le 2n$, we can say that $T_k \in O(n)$, and so our algorithm will, in the best or worst case, require us to walk

$$4(T_1 + T_2 + \cdots + T_{k-1}) + O(n) = 4\sum_{j=1}^{k-1} T_j + O(n) = 2\sum_{j=1}^{k-1} j^2 + 2\sum_{j=1}^{k-1} j + O(n)$$

$$= \frac{(k-1)k(2k-1)}{3} + (k-1)k + O(n) = O(k^3) + O(n) \in O(n^{1.5}).$$

(d) Describe a faster algorithm that guarantees you will always reach your car. (Hint: think about Problem 5 - is there a way to adapt that strategy for this new problem?)

Let's double how far we walk each time. So check 1 car to left and right, then 2 cars to left and right, then 4 to left and right, then 8, etc.

A faster (though not asymptotically faster) algorithm would be to walk 1 car to the left, then (without returning to the start) walk 2 cars to the right, then (without returning to the start) walk 4 cars to the left, then walk 8 cars to the right, etc.

(e) Prove that your algorithm is correct (i.e., briefly explain why your algorithm will always find your car).

We alternately check increasingly large distances in each direction, ensuring that at some point we will check any given car. This means that our algorithm will eventually locate our car.

(f) Derive the runtime of your algorithm, i.e., how many cars will you pass before reaching your car using your algorithm?

Let's assume that n is a power of 2, i.e., $n = 2^k$.

For the first algorithm (walk 1 to left and right, then 2 to left and right, then 4 to left and right, etc), our last step would be to walk 2^k to the left and right to find our car (though in the best case, we only need to walk to the left and stop), meaning that our overall distance would be

$$4(1 + 2 + \cdots + 2^{k-1}) + 2^k \leq T(n) \leq 4(1 + 2 + \cdots + 2^{k-1}) + 3 \cdot 2^k,$$

and therefore

$$5 \cdot 2^k - 4 \leq T(n) \leq 7 \cdot 2^k - 4 \quad \Rightarrow \quad 5n - 4 \leq T(n) \leq 7n - 4 \quad \Rightarrow \quad T(n) \in \Theta(n).$$

For the second algorithm (where we do not return to the start location, but rather zig-zag while doubling the distance each time), we need to determine the size of our final step.

Note that when we make a step of size 2^m to the right, we end up less than 2^m distance to the right the start. This is because we were already some distance to the left when we began walking. When we then make a step of size 2^{m+1} back to the left, we end up less than 2^{m+1} to the left of the start. Again, this is because we were already some distance to the right when we began walking. When we then make the step of size 2^{m+2} back to the right, we therefore must end up at a location further than 2^{m+1} to the right of the start, since we began walking less than 2^{m+1} to the left of the start.

So, if we are trying to reach a location that is 2^k distance from the start, our final step must be of size between 2^k (since at this step we will not have reached our location yet) and 2^{k+3} ($k + 3$ to account for the fact that the 2^{k+2} step, which would have been far enough, could have been in the wrong direction).

Thus we will have to walk a distance

$$1 + 2 + \cdots + 2^k \leq T(n) \leq 1 + 2 + \cdots + 2^{k+3} \quad \Rightarrow \quad 2^{k+1} - 1 \leq T(n) \leq 2^{k+4} - 1$$

$$\Rightarrow \quad 2n - 1 \leq T(n) \leq 16n - 1 \quad \Rightarrow \quad T(n) \in \Theta(n).$$

7. **Dragon Hunting**

You are a knight defending a group of small farms spread across the country side. A very hungry dragon has arrived, and has started eating the animals at the farms. The dragon flies to the largest uneaten farm (by total weight), eats all of the animals there, burns down the farm, and then moves to the next largest remaining farm. Both you and the dragon know the total weight of animals at each farm.

Unfortunately, you are not aware of where the dragon is right now, and the dragon flies faster than your horse can gallop. Assuming that it takes the same time to travel between any two farms, in what order should you visit the farms to catch the dragon while minimizing destruction?

Specifics:

- You have a list of n farms to protect, sorted from largest to smallest.
- The dragon has already burned down the first k farms and will start moving towards the $(k+1)$st farm as you begin the hunt.
- You do not know the value of k.
- It takes a constant time to move between any two farms.
- The dragon will move slightly faster than you, but can only burn down 1 farm in the time it takes you to travel between farms.

Solution:

The idea here is that we will use a binary search. However, we have to be careful because it is possible for the dragon to slip past us while we search. After all, the dragon moves.

If we consider the example with 9 farms where the dragon starts at farm 3, we see the problem (for this example, X is destroyed but unknown, 0 is safe but unknown, x is destroyed and known, o is safe and known, D is dragon, the dotted symbol is you). Seeing the initial setup, start by visiting farm 5 to begin the binary hunt:

X	X	D	0	0	0	0	0	0
X	X	X	D	$\dot{0}$	0	0	0	0
X	X	$\dot{x}$	X	D	0	0	0	0
X	X	x	$\dot{x}$	o	D	0	0	0

The binary search terminates, but the dragon escaped past a location we had marked as safe!

There are several ways to modify our algorithm and guarantee that we can catch the dragon in the $O(\log n)$ time of the binary search (which comes from dividing the length n search range in half repetitively):

- In the first approach, let's conduct our binary search as stated before. That means it's possible that the dragon escapes us (and certain if our malicious adversary is involved). But how far past us could it have gotten? Note that the last 0 we saw (our starting position in the example above) was a safe farm at that time, call that location i. That means that the farm i was safe at the beginning of the algorithm. Our binary search took $O(\log n)$ steps, so we know that the dragon cannot be more than $\log n$ steps beyond farm i (plus or minus a few if n is not a power of 2, which is something we could keep track of easily). In other words, the dragon is between location i and $i + \log n$.

  We now could restart our binary search in this new range of size $\log n$. However, the dragon could again escape us. This search took $\log(\log n)$ time, and so our new search will be in an interval of size $\log(\log n)$.

  This continues, and gives us the runtime expression

  $$\log n + \log(\log n) + \log(\log(\log n)) + \cdots + 1 \in O(\log n).$$

- A lazier approach focuses on the statement 'in other words, the dragon is between location i and $i + \log n$.' This means that farm $i + \log n$ has not yet been burned. Moreover, it means that the dragon is at most $\log n$ distance from that farm. So the solution now is to go to farm $i + \log n$ and simply wait for the dragon. The runtime now is the $\log n$ from the initial binary search, with the additional $\log n$ from the waiting. This gives a total of $2 \log n \in O(\log n)$.

- A third approach to the problem involves modifying the binary search itself to account for the dragon's movement. Consider the following modified binary search, where i and j define the range in which the dragon is located, starting originally at $i = 1$ and $j = n$:
  - While the dragon is not caught:
    - Consider location $x = \lfloor \frac{i+j}{2} \rfloor$.
    - If x is burned, $i = x$. Else, $j = x$.
    - Increment: i++ and j++.
    - Ensure $j \leq n$.

  The first three lines perform standard binary search. But that fourth line with the increment is the key to our modified solution to catch the dragon. If the dragon could have been between i and j at iteration k, then after one move at iteration $k + 1$ it will be between $i + 1$ and $j + 1$. Now the binary search can proceed as usual, since it is moving with the dragon. So this give $O(\log n)$ time as well.