**Standard 6: Recurrence relation: solve by plugging in/unrolling**

For standard 6, I failed to solve for and substitute k correctly (among other problems I will talk about in the next paragraph). This resulted with me not simplifying my unrolled solution correctly. After unrolling the recurrence, I should have solved for T(n), but since my substitution was incorrect, my answer for T(n) was also incorrect. K should have been = (n-5)/5. This could have been found by taking the first value after unrolling, 2kT(n-(5\*k)), and solving for k within the T() by suing the base case in the recurrence. By doing this, I would have solved for n-(5k) = 5. Solving for k, I would have gotten k = (n-5)/5.

Unfortunately, even if I had substituted k correctly, I did not recognize (or would not have recognized) the proper ways to simplify and solve for T(n). I should have recognized that the right-hand side addition was a geometric sequence (after the unrolling step). The right-hand side addition was a geometric sequence due to it incrementing by 2i after every addition. This geometric series would have been evaluated to (1-2k)/(1-2).

After substituting this, I would be able to reduce the problem down to the solution by recognizing which values we could exclude due to their size/growth rates. I would then remove all values left on the right side of the recurrence other than 2(n-5)/5 because that is the only value that needs to be considered. It is the only value that needs to be considered since it is the only value that exponentially grows over time left in the recurrence.

If I had been able to solve for k correctly, identify that the right-hand side addition was a geometric sequence, and simplified the remaining recurrence correctly, I would have successfully completed the problem correctly.

I now understand how to solve for k and how to plug it back into the unrolled recurrence. I also now understand how to identify a geometric sequence and substitute the geometric sequence evaluation back into the unrolled recurrence. Finally, I now understand how to solve the initial recurrence with k and the geometric sequence evaluation plugged back into the unrolled recurrence.

**Standard 9: QuickSort (worst-case vs expected behavior)**

I was incorrect in assuming that QuickSort’s worst input expected runtime was the same as its worst input worst-case runtime. I believed this to be true due to us discussing during lecture that by using randomness, QuickSort can be made to have the expected runtime O(nlogn) every time. This is not applicable to this question because the worst input worst-case runtime is O(n^2) due to how randomized QuickSort works in its worst-case runtime. In the worst-case runtime, QuickSort ends up using the worst pivot, which means that the pivot will either be set to the greatest value or smallest value in the sorted array. Due to this, the expected runtime for the worst input using QuickSort is not the same as the worst-case runtime for the worst input.

I believe that I did not fully understand what the question was asking and applied information that did not pertain to this question. The expected runtime will be O(nlogn) based on any input, but the worst-case runtime for the worst input will be O(n^2). This is due to the array ending up being split into all individual size 1 arrays before being put back together. We cannot apply the expected runtime of O(nlogn) in this case because we are explicitly declaring that we want to know worst-case runtime of QuickSort for the worst input possible.

I now understand that the expected runtime of QuickSort will be different than the worst-case runtime. I also understand now why the worst-case worst input runtime of Quicksort is O(n^2).

**Standard 10: Hash tables (load factor, when to apply vs other data structures)**

When attempting this standard on the quiz, I did not fully understand how to solve for and explain the complexity of a hash table. I also failed to speak on and apply the load factor of the hash table.

A)

For part A, I did not specify the expected collision length for the Add, Lookup, and Remove operations. The Add expected collision length for a linked list is always 1, so for this problem the complexity of adding an item is O(1). The Lookup and Remove operations have the same complexity, O(n). This is due to us having to compare and determine if the values are the same. This is not the case with Add because we can simply insert the item into the bucket.

The load factor of this hash table would be n/1, which directly applies to how Lookup and Remove will function in this hash table. N comes from us having n items and 1 comes from us having 1 bucket.

B)

In part B, I failed again to speak on the complexities of Add, Lookup, and Remove. I also again failed to speak on and apply the load factor of the hash table.

The Add expected collision length for a linked list is always 1, so for this problem the complexity of adding an item is O(1). The Lookup and Remove operations have the same expected collision length, 1. We can solve for this by solving for the load factor of the hash table. The load factor of the table will be n/O(n). O(n) comes from us having O(n) buckets and n comes from us having n items. This load factor can be reduced to 1, so the complexity of Lookup and Remove for this hash table are O(1).

C)

In part C, I failed again to speak on the complexities of Add, Lookup, and Remove. I also again failed to speak on and apply the load factor of the hash table.

The Add expected collision length for a linked list is always 1, so for this problem the complexity of adding an item is O(1). The Lookup and Remove operations have the same expected collision length, O(). We can solve for this by solving for the load factor of the table. The load factor of the table will be n/ O(). O() comes from us having O() buckets and n comes from us having n items. This load factor can be reduced to O(), so the complexity of Lookup and Remove for this hash table are O().

I now understand how to solve for and explain the complexity of a hash table by speaking about its relationship with the Add, Lookup, and Remove operations. I also now understand how to solve for the load factor of a hash table and why it is used when explaining the complexity of the hash table.