35+15+10= Total:60. grades out of 10

Total : 10

"If code files are missing, a deduction of 2 points will be made overall for Q1 and Q2. In the future, more points will be deducted, as it is highly time-consuming to extract the code from the solution file, run it, and test it otherwise."

Q1)

Correct program and algorithm: 4 points

Analysis: 6 points

```
   int count=0;
   for(int i=0; i<n;i++) //where n is the size of the array
      { for(int j=i+1; j<n; j++)
          { if(a[i]==a[j])
           { count++;
            break;}}
      }
cout<<count;
```

Analysis needs three main parts: (3*2 points)

Discussion of Big-O(), Discussion of Omega(), discussion of whether and why Theta holds.

O(n) for the first loop * O(n) for the second loop * O(1) for the operations inside the nested loops= O(n^2)

They can have a cost table (similar to what we learned for linear search and binary search) and come up with the cost function of $f(n)=C_1n^2+ C_2n+ C_3$

They then can lower bound that explaining C and n0, and showing f(n)=omega(n^2). Therefore, Theta(n). We have done this lower bound and upper bound analysis in the video lectures, asymptotic notations part.

Note: Due to theta (n^2) specified in the question, students should not provide something more efficient than quadratic time complexity for this question. They, however, can make it more efficient in the next question. – 5 points if this is not followed as the meaning of Theta is not well understood/followed.

Total : 15

Q2)

Correct program with an efficient algorithm: 5 points

If the algorithm is not as efficient as O(n) then partial credit for algorithm efficiency.

Analysis: 5 points.

5 points improvement and its justification as written below.

Assuming that we have a sorted list, we can use the order to compare elements and find the number of duplicates. In the following part of the code we find the number of unique duplicate numbers

```
int dupCount = 0;
int previous = -1; (assuming all numbers are positive)
for (int i=0; i < n; ++i) {
  if (a[i] == previous)
        ++dupCount;
  else
     previous = a[i];
  }
cout<<dupCount;
```

(5 point) The functions runs in O(n) in the worst case as it goes through a loop of O(n) and each time runs constant amount of calculation (O(1) per loop) , so the total would be O(1)*O(n) which is O(n).

(5 point) Yes, We could improve the worst-case time complexity of the algorithm. You should argue this: for example you may say: That's because Q1 has f(n)=O(n^2) as the time complexity while Q2 is g(n)=O(n) and we can prove that g(n)=O(f(n)):
Here is a proof sketch:
Let C=1 and n0=1. We claim that for all n>=1, we have n<= n^2 .
Lets start by n>=1. Multiply both sides by n we will get n^2>= n and the claim is proved.
(You could use induction or other methods to prove as well.)
, in another word, f(n) is an upper bound for g(n) or g(n) grows slower as n grows comparing to f(n).

Total 34- (add 1 extra point to round = that is 35)

Question 3)

Consider the following functions as time complexities of some algorithms.

(a) First write the worst-case runtime of each algorithm in Big-O notation.

Each case 3 points: total 3*9

f1(n)=O(n^3),
f2(n)=O(nlogn),
f3(n)=O(3^n),
f4(n)=O(n) ,
f5(n)=O(1),
f6(n)=O(logn),
f7(n)=O(n^n),
f8(n)=O(n^2),
f9(n)=O(log n) note that log (a^b)=b log a

(b)

Total 7 points. 1 points for every two consecutive ones

f5<= f6 and f9 in either order<=f4<=f2<=f8<=f1<=f3<=f7
O(1)<=O(log n)<= O(n)<=O(n log n)<=O(n^2)<= O(n^3) <= O(3^n)<=O(n^n)

Proof part: Students are not required to provide the whole parts below. They should pick only two cases and prove it.

NOTE: if you want to prove f(n)=n is O(n^2), you can't just re-say that since n grows slower than n^2, therefore the claim holds. You are asked to formally prove that, so don't just re-read the claim. Use mathematical facts and operations and prove it.
For this example, you can simple assume n>=1, multiply both sides by n, that results in n<= n^2, so for c=1, n0=1, we get to n<=cn^2 for all n>=n0, and the proof follows.
Similar justification could be used for Q2 once you want to discuss why your improved algorithm with O(n) is more efficient than the one in Q1 with O(n^2).
NOTE: To prove that f(n)=n is O(n^2), you cannot simply state that n grows slower than n^ 2 and therefore the claim is valid. This is the meaning of the claim (the meaning of Big-O here). You are required to formally prove it, so avoid merely repeating the claim. Use proper mathematical facts and operations to establish the proof.
In this case, you can assume n≥1, then multiply both sides by n, which gives n≤n^2 . Therefore, for c=1 and n0 =1, we have n≤cn^2 for all n≥n0 , and the proof follows.

A similar justification can be applied to Question 2 when explaining why your improved algorithm with O(n) is more efficient than the one in Question 1 with O(n^2).

Note: For many of such cases, induction is a simple approach to use.

1) f5=O(f6 or f9) i.e., 1<=O(log n). pretty clear but here is the formal proof
   Let c=1, n>=2, let base for log be 2 through this proof.
   Considering n>=2 take log base 2 from both sides, log n>=1 that is 1<= 1* logn for all n>=2.
   Therefore assuming c=1 and n_0=2, f5<= O(log n) . Done.

2) f6 or f9 <=f4, that is log n= O(n)

There are different ways to approach this part. One way is to plot both functions and discuss how they behave. Using the plot, they can discuss that for all n>=4, $\log_2 n$ grows slower than that of function n. A solid way is to use the limit functions and show lim n/log n is infinity as n grows to infinity. Another way is to use this relation in calculus that $e^n \geq 1+n>n$. Since log is an increasing function, applying it to each side of this inequality results in $n>\log n$,

3) f4<=f2 that is n<= O(n log n)

   let n>=2, take log from both sides. Since log is an increasing function we will have
   log n>= 1, multiply both sides by n where n>=2 results in
   n log n>= n so for c=1 and n_0=2 proof follows…

4) f2<=f8 that is n log n =O(n^2)
   We showed that log n=O(n) for constants c and n_0=2. That means log n<= c n, for all n>= 2. Now multiply both sides by n results in n log n=O(n^2) for mentioned c and n_0=2, proving the claim.

5) f8<=f1 that is n^2= O(n^3)
   Let n>=1 and we know that n^2 >=1. Multiply both sides of n>=1 by n^2, resulting in n^3>= 1*n^2, therefore n^2=O(n^3) for c=1 and n_0=1, Done.

6) f1<=f3 that is n^3= O(3^n)

We want to prove n^3 $=\mathrm{O}(3^{\mathrm{n}})$:
Let *c = 1* and n_0 =4. We need to prove that
   $n^3 <= 3^n$ for all *n* > =9.
Take $\log_3$ from both sides. $log(n^3) <= log(3^n)$ and by logarithm rules $3\cdot log(n) <= n\cdot log_3(3)=n <= 3 n$
Let n = 9, we get $3\cdot log_3(9) <2^9$ and since *n* grows faster than *log(n)* as we proved ABOVE, we can claim that *log(n)* < *n* holds for all *n* > 9 (students will justify that as we did in part 2 above). Therefore, let c=3 and n_0=9, n^3 $=\mathrm{O}(3^{\mathrm{n}})$.

7) f3<=f7 that is 3^n= O(n^n)

Let c=1, n>=4…Use formal definition. discuss similar to what we did in the video lecture. Ensure you specify WHY and FOR ALL n>=n0 as you justify…Straightforward