Solution 1.

1. $T(n) = 3T(n/2) + n^2$

MasterTheorem : In the recurrence relation, we know that b=2 and a=3, therefore $\log_b a = \log_2 3$. We have $f(n) = n^2 = \Omega(n^{\log_2 3+\epsilon})$ for some $\epsilon > 0$, and $\frac{3n^2}{4} \leq \delta n^2$ for some $\delta < 1$. This fulfills case 3 of the Master Theorem, we can then derive that $T(n) = \Theta(n^2)$.

2. $T(n) = 4T(n/2) + n$

MasterTheorem : In the recurrence relation, we know that b=2 and a=4, therefore $\log_b a = 2$. We have $f(n) = n = O(n^{2-\epsilon})$ for some $\epsilon > 0$. This fulfills case 1 of the Master Theorem, we can then derive that $T(n) = \Theta(n^2)$.

3. $T(n) = T(n-1) + n$

Unrolling : This recurrence uses linear effort (proportional to size), reduces the amount of data by one, solves the reduced problem recursively, and then does constant work to adjust the result of the recursive call to give the result on the full input.

Which solves to $c * n + c * (n-1) + c * (n-2) + ... + c * 1 = c(1 + 2 + 3 + ...n) = \Theta(n^2)$

4. $T(n) = 2T(n/2) + n \log n$

MasterTheorem : In the recurrence relation, we know that b=2 and a=2, therefore $\log_b a = 1$. We have $f(n) = n \log n = \Theta(n^1 \log^k n)$ for $k = 1$, which is ≥ 0, This fulfills case 2 of the Master Theorem, we can then derive that $T(n) = \Theta(n \log^2 n)$.

5.$T(n) = \sqrt{2}T(n/2) + \log n$

MasterTheorem : In the recurrence relation, we know that $b = 2$ and $a = \sqrt{2}$, therefore $\log_b a = \log_2 \sqrt{2}$. We have $f(n) = \log n = O(n^{\log_2 \sqrt{2}-\epsilon})$ for some $\epsilon > 0$. This fulfills case 1 of the Master Theorem, we can then derive that $T(n) = O(n^{\log_2 \sqrt{2}}) = \Theta(\sqrt{n})$.

6.$T(n) = 3T(n/3) + \sqrt{n}$

MasterTheorem : In the recurrence relation, we know that $b = 3$ and $a = 3$, therefore $\log_b a = 1$. We have $f(n) = \sqrt{n} = O(n^{1-\epsilon})$ for some $\epsilon > 0$. This fulfills case 1 of the Master Theorem, we can then derive that $T(n) = \Theta(n)$.

7. $T(n) = 7T(n/3) + n^2$

MasterTheorem : In the recurrence relation, we know that $b = 3$ and $a = 7$, therefore $\log_b a = \log_3 7$. We have $f(n) = n^2 = \Omega(n^{\log_3 7+\epsilon})$ for some $\epsilon > 0$, and $\frac{7n^2}{9} \leq \delta n^2$ for some $\delta < 1$. This fulfills case 3 of the Master Theorem, we can then derive that $T(n) = \Theta(n^2)$.

8. $T(n) = T(\sqrt{n}) + \log n$

MasterTheorem : First we know $n = 2^m$. From this recurrence relation, we can get $T(2^m) = T(2^{m/2}) + m$.

Assume we have $B(m) = T(2^m)$, we get $B(m) = B(m/2) + m$.

For the new recurrence relation $B(m)$, we know that $b = 2$ and $a = 1$, therefore $\log_b a = 0$. We have $f(m) = m = \Omega(m^{0+\epsilon})$ for some $\epsilon > 0$, and $m/2 \leq \delta m$ for some $\delta < 1$. This fulfills case 3 of the Master Theorem, we can then derive that $B(m) = \Theta(m)$.

Then we have $T(2^m) = B(m) = \Theta(m)$. Because $n = 2^m$, we can get $m = \log n$, therefore $T(n) = \Theta(\log n)$

Solution 2.

Question 1

a)

We are given two top skeletons, and we store them in two doubly linked list consisting of triples, for better understaning, we name the two lists A and B, and we have pointers pointing at A and B called *pointerA* and *pointerB*. We created one empty linked list S.

From the question we know the first element of the triple is the left x coordinate of the segment, the second element of the triple is the right x coordinate of the segment and the third element of the triple is the y coordinate of the segment.

We compare the triple *pointerA* and *pointerB* pointed to. We call the value *pointerA* pointed to S1 and the value *pointerB* pointed to S2. The principle is to find out which one dominates (has the higher height) and solve overlaps accordingly.

If the first element of S1 < the first element of S2, and if the third element of S1 < the third element of S2, then we check if the second element of S1 > the first element of S2 and if the second element of S1 $\leq$ the second element of S2 , if the condition is satisfied, we change the second element of S1 to the first element of S2, then we append S1 to list S and we move *pointerA* to right by one and continue processing. If the second element of S1 < the first element of S2 we just append S1 to S and we move *pointerA* to right by one and continue processing. If the second element of S1 > the first element of S2 and if the second element of S1 > the second element if S2, we first make a copy of S1 and change the second element of the copy to the first element of S2, and we append this copy into S, then we append S2 into S, next we change the first element of the S1 to the second element of S2, then we move *pointerB* to right by one and continue processing.

If the first element of S1 < the first element of S2, and if the third element of S1 > the third element of S2, then we check if the second element of S1 > the first element of S2 and the second element of S1 < the second element of S2, if the condition is satisfied, we change the first element of S2 to the second element of S1, then we append S1 to list S and we move *pointerA* to right by one and continue processing. If the second element of S1 > the first element of S2 and the second element of S1 $\geq$ the second element of S2, then we move *pointerB* to right by one and continue processing. If the second element of S1 < the first element of S2 we just append S1 to S and we move *pointerA* to right by on e and continue processing.

If the first element of S1 < the first element of S2, and if the third element of S1 = the third element of S2, then we check if the second element of S1 $\leq$ the first element of S2 , if the condition is satisfied, then we append S1 to list S and we move *pointerA* to right by one and continue processing, if the condition is not true, then we change the first element of S2 to the first element of S1, and we proceed to the case where the first element of S1 = the first element of S2.

If the first element of S1 = the first element of S2, and if the third element of S1 $\leq$ the third element of S2, we compare the second element of S1 and the second element of S2, if the second element of S1 $\leq$ the second element of S2. we move *pointerA* to right by one index and continue processing, if not we change the first element of S1 to the second element of S2 , we append S2 to S and then we move *pointerB* to right by one and continue processing.

If the first element of S1 = the first element of S2, and if the third element of S1 > the third element of S2, we compare the second element of S1 and the second element of S2, if the second element of S1 $\leq$ the second element of S2. we change the first element of S2 to the second element of S1 , we append S1 to S and then we move *pointerA* to right by one and continue processing, if not we move *pointerB* to right by one index and continue processing.

If the first element of S1 > the first element of S2, and if the third element of S1 < the third element of S2, then we check if the second element of S2 > the first element of S1 and the second element of S1 > the second element of S2 , if the condition is satisfied, we change the first element of S1 to the second element of S2, then we append S2 to list S and move *pointerB* to right by one and continue processing. If the second element of S2 > the first element of S1 and the second element of S1 $\leq$ the second element of S2, we move *pointerA* to right by one and continue processing. If the second element of S2 < the first element of S1, we just append S2 to S and we move *pointerB* to right by one and continue processing.

If the first element of S1 > the first element of S2, and if the third element of S1 > the third element of S2, then we check if the second element of S2 > the first element of S1 and if the second element of S2 $\leq$ the second element of S1, if the condition is satisfied, we change the second element of S2 to the first element of S1, then we append S2 to list S, and we move *pointerB* to right by one and continue processing. If the second element of S2 leq the first element of S1 we just append S2 to S, and we move *pointerB* to right by one and continue processing. if the second element of S2 > the first element of S1 and if the second element of S2 > the second element of S1, we first make a copy of S2 and change the second element of the copy to the first element of S1, and we append this copy to S, then we append S1 into S, next we change the first element of the S2 to the second element of S1, then we move *pointerA* to right by one and continue processing.

If the first element of S1 > the first element of S2, and if the third element of S1 = the third element of S2, then we check if the second element of S2 $\leq$ the first element of S1 , if the condition is satisfied, then we append S2 to list S and we move *pointerB* to right by one and continue processing, if the condition is not true, then we change the first element of S1 to the first element of S2, and we proceed to the case where the first element of S1 = the first element of S2.

We continued the aforementioned process until two pointers went out of bounds. If one of the pointers went out of bounds first, we append whatever is left in the other list that is to the right of the other pointer to S (including the one which is pointed by the other pointer). We then return S as the combined skeleton.

b)

The given input is two top skeletons sorted from left to right. This means there's no overlap in A and B, overlap can only happen when combining. We first compare the first element of each triple (the left x coordinate of the segment) to make sure we can store it sequentially, then we check if there exists an overlap by comparing the second element of the segment that has the smaller left x coordinate with the first coordinate of the other segment. If the second element of the segment that has the smaller left x coordinate is bigger than the first coordinate of the other segment, then there exists an overlap. If such overlap exists, we check which one dominates by checking the y coordinate (the third element of the triple), from the question we know the higher one dominates. We shrink the segment that has the lower y coordinate. We then append whatever is in the front (has a lower left x coordinate) to S. If the higher one completely overlaps the lower segment then we disregard the lower segment. This won't change the fact that there's no overlap in A and B as we are always shrinking the segment. The only exception is when they overlap and have the same height, I will extend the segment that has a higher left x coordinate and change the left x coordinate to the left x coordinate of the other segment. But it won't result in an overlap within its own list as all previous segments have been processed so that it will not overlap with any segment in the other list. So if I extend the segment that I am currently processing to the left x coordinate of the other segment in the other list, it won't overlap with its own list. This makes sure we only need to process overlap that happens when combining as in A and B there won't be any overlaps.

If the higher one is overlapped by the lower one, the lower half is going to be shrinked into 2 halves respectively so that the overlap will be resolved. As mentioned above shrinking will not cause overlap within its own list.

Then we move pointers accordingly to go to the next iteration.

The program will keep executing until two pointers go out of bounds. This way we go through every segment and account for every overlap between two top skeletons, for every iteration the overlap will be dealt with correctly, and the segment that has been processed will be put in S. So in the end the algorithm will correctly return S, and we always put the one has the lower left x coordinate first, which will give us the combined top skeleton sorted from left to right.

c)

For this algorithm, at most one of the two pointers: *pointerA* and *pointerB* will move per iteration, the processing time for each iteration will cost constant time, as compare and modification of triples are all constant time. We have total of n elements so the upper bound of this algorithm is $\sum_{i=0}^{n} O(1) = O(n)$

Question 2

a)

We divide the input into two halves from the middle until we reach the base case which is until there is only one element left in each list and then we combine. We then combine it using the combination step we mentioned in Question 2.1. We then repeat this operation until done.

b)

The base case will have one element in each list because it has only one element, this guarantees there is no overlap in the list. We assume the subproblem is solved correctly. So now we need to show given two top skeletons sorted from left to right , we need to return the correct combined skeleton. According to Question 2.1, we know that given two top skeletons sorted from left to right, we can return the correct combined top skeleton. So the algorithm is correct.

c)

From the divide and conquer algorithm, we know the recurrence relation is $T(n) = 2T(n/2) + 0(n)$.

In the recurrence relation, we know that b=2 and a=2, therefore $\log_b a = 1$. We have $f(n) = n = \Theta(n^1 \log^k n)$ for $k = 0$, which is ≥ 0, This fulfills case 2 of the Master Theorem, we can then derive that $T(n) = O(n \log n)$.

Solution 3.

a)

We divide the input using the balanced splitter, which will split the whole input into three parts: A,S,B. Our base case is when A and B have less or equal to 2 verities. At the base case we know the length of the shortest directed cycle is ∞ for both A and B because such cycle doesn't exist. We use the balanced splitter to recursively divide A and B until we reach the base case. We name one group of the subproblems S_n,A_n,B_n. We keep track of the length of the shortest directed cycle in A_n, and the length of the shortest directed cycle in B_n. To calculate the length of the possible shortest directed cycle in the subgraph when combining, we do Dijkstra's algorithm on every node in S_n, After doing Dijkstra on one node, we check on every other node to see if it can be connected back to the starting point. If one node can be connected back to the starting point, the cycle length can be computed by the length of the shortest path from the starting point to that node(computed by Dijkstra) + the weight of the path from that node to the starting point. If such value exists we append this value in a list to keep track of the length. We completed checking this node when we checked the connection between this node and every other node. We do this for every vertices in S. Then we will find the smallest value in that list so we have the length of the shortest directed cycle when combining. Because we know the shortest directed cycle of A_n and B_n, we compare the three values we have so that we can find the length of the shortest directed cycle of the combined subproblems. Then we update the value to that subproblem. We repeat this operation for every subproblems until it's done.

b)

The base case is when A and B have less or equal to 2 verities. Because there are no crossing edges in the graph and this extends to its subgraph as well, from this fact we know A won't contain a cycle with $\leq$ 2 verities and B won't contain a cycle with $\leq$ 2 verities. A cycle won't exist between A and B without going through S because there are no edges with one endpoint in A and one endpoint in B. So the length of the shortest directed cycle in A and B are ∞ . So the base case is correct. We assume the subproblems are solved correctly, we have the length of the shortest cycle of A and the length of the shortest cycle of B. When combining we calculate the cycle that hasn't been calculated. That would have to be connected through S. If a cycle exists, the element in S must be contained. It can be observed that the path of the shortest directed cycle must contain the path of the Dijkstra algorithm. So we run Dijkstra on every node in S and find the length of the cycle accordingly. The reason why this works is because Dijkstra already contains the shortest distance between the starting point and each node. We query every other node to find the possible length of the shortest directed cycle and we append the every possible result into a list. When we have a list of possible shortest cycles then we find the smallest one and compare it with the length of the shortest cycle of A and the length of the shortest cycle of B, in this way we find the length of the shortest cycle for the combined subproblem. We can see the combine step also yells the right result, so the algorithm is correct.

c) The combined step of this algorithm is upper bounded by the number of times Dijkstra is run, which depends on the number of vertices in S, we know it contians $O(\sqrt{n})$ vertices.So the complexity for the combine step is $O(\sqrt{n} * n \log n)$, the querying and processing operation takes $O(\sqrt{n})$ in total, it is upper bounded by $O(\sqrt{n} * n \log n)$. Our two subproblem have at least n/3 for one of them and at most 2n/3 for the other one, so the upper bound recurrence relation is $T(n) = T(n/3) + T(2n/3) + O(n^{3/2} \log n)$.