Solution 1.

1. $T(n) = 3T(\frac{n}{2}) + n^2$
   $a = 3$, $b = 2$, $f(n) = n^2$, $log_b a \to log_2 3 \approx 1.585$; so $f(n) = \Omega(n^{\log_b a + \epsilon})$(case 3).$T(n) \in \Theta(n^2)$

2. $T(n) = 4T(\frac{n}{2}) + n$
   $a = 4$, $b = 2$, $f(n) = n$, $log_b a \to log_2 4 = 2$; so $f(n) = O(n^{\log_b a - \epsilon})$(case 1).$T(n) \in \Theta(n^2)$

3. $T(n) = T(n-1) + n$
   Known that $T(1) = 1$,we have:

   - $T(2) = T(1) + 2 = 1 + 2$
   - $T(3) = T(2) + 3 = 1 + 2 + 3$
   - $T(4) = T(3) + 4 = 1 + 2 + 3 + 4$
   - .....
   - $T(n) = T(n-1) + n = 1 + 2 + 3 + 4 + ... + (n-1) + n$

   So that $T(n) = \sum_{i=1}^{n} i = \frac{(1+n)n}{2} = \frac{n^2+n}{2} = \Theta(n^2)$

4. $T(n) = 2T(\frac{n}{2}) + nlog(n)$
   $a = 2$, $b = 2$, $f(n) = nlog(n)$, $log_b a \to log_2 2 = 1$; so $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$(case 2).$T(n) \in \Theta(nlog^2(n))$

5. $T(n) = \sqrt{2}T(\frac{n}{2}) + \log(n)$
   $a = \sqrt{2}$, $b = 2$, $f(n) = \log(n)$, $log_b a \to log_2 \sqrt{2} = \frac{1}{2}$; so $f(n) = O(n^{\log_b a - \epsilon})$(case 1).$T(n) \in \Theta(\sqrt{n})$

6. $T(n) = 3T(\frac{n}{3}) + \sqrt{n}$
   $a = 3$, $b = 3$, $f(n) = \sqrt{n}$, $log_b a \to log_3 3 = 1$; so $f(n) = O(n^{\log_b a - \epsilon})$(case 1).$T(n) \in \Theta(n)$

7. $T(n) = 7T(\frac{n}{3}) + n^2$
   $a = 7$, $b = 3$, $f(n) = n^2$, $log_b a \to log_3 \sqrt{7} = 1.771$; so $f(n) = \Omega(n^{\log_b a + \epsilon})$(case 3).$T(n) \in \Theta(n^2)$

8. $T(n) = T(\sqrt{n}) + \log(n)$
   Let $n = 2^m$,we have:

   $$T(2^m) = T(2^{m/2}) + m$$

   Let $S(m) = T(2^m)$,we have:

   $$S(m) = S(m/2) + m$$

   $a = 1$, $b = 2$, $f(m) = m$, $log_b a \to log_2 1 = 0$; so so $f(m) = \Omega(m^{\log_b a + \epsilon})$(case 3).$T(n) = \Theta(T(m)) = \Theta(m) = \Theta(\log n)$

Solution 2.

1

(a)

The general idea is to use two pointers to keep track of H_1 and H_2. Gradually traverse the line segments of H_1 and H_2 and merge them from left to right while handling potential overlapping parts.

We first create an empty list H to store the combined top skeleton and set two pointers, i and j, to point to the start of H_1 and H_2, respectively.

Then we begin traversing H_1 and H_2: compare the left endpoints l_i of $H_1[i]$ and l_j of $H_2[j]$, and choose the segment with the smaller left endpoint to add to H:

- if $l_i < l_j$, select the segment $H_1[i]$ and increment pointer i by 1.
- if $l_i > l_j$, select the segment $H_2[j]$ and increment pointer j by 1.
- if $l_i = l_j$, select the segment with $\max(h_i, h_j)$ and increment the corresponding pointer by 1.

Next, merge the potential overlapping parts:

- If the currently added segment H_k overlaps with the last segment H_{k-1} in the x-range , and they have the same y-coordinate, merge these segments into one segment. If the y-coordinates are different, handle the overlap by:
  - If the height of H_k is greater than H_{k-1}, truncate the overlapping part of H_{k-1} and then add H_k to H.
  - If the height of H_k is less than H_{k-1}, add the non-overlapping part of the H_k to H.
- After one list is fully traversed, we add the remaining segments from the other list directly to H one by one. During this process, check for overlaps with the last segment in H and handle them as described above.

Finally, return the combined top skeleton list H.

(b)

- **Invariant**: To prove the correctness of the algorithm, we use an invariant: After each step, the list H contains the correct combined segments of H_1 and H_2 up to the current pointers i and j.
- **Base case**: Initially, H is empty, and i and j point to the start of H_1 and H_2, respectively. The invariant trivially holds .
- **Inductive Step**
  - When adding a new segment from H_1 or H_2, we ensure it is correctly placed in H by comparing the left endpoints l_i and l_j.This ensures that the segments are arranged in increasing order of their left endpoints.

- If the new segment overlaps with the last segment in H, we handle the overlap by either merging the segments (if they have the same y-coordinate) or truncating the overlapping part (if the y-coordinates are different). This ensures that the segments in H remain correctly ordered and non-overlapping.
- After one list is fully traversed, the remaining segments from the other list are added directly to H. Any overlaps with the last segment in H are handled as before, ensuring the invariant continues to hold.

By maintaining the invariant at each step, we ensure that the final list H is the correct combined top skeleton of H_1 and H_2, with all segments correctly merged and overlapping parts properly handled. Therefore, the algorithm is correct.

(c)

The initialization step, which involves creating an empty list H and initializing pointers i and j to point to the start of H_1 and H_2, respectively, is a constant time operation with a time complexity of $O(1)$.

During the iteration process, the algorithm traverses both lists H_1 and H_2 using these pointers. In each iteration, it compares the current segments from H_1 and H_2 and decides which one to add to H. Since each segment from H_1 and H_2 is considered exactly once, the total number of iterations is proportional to the sum of the lengths of H_1 and H_2, resulting in a time complexity of $O(n)$.

While handling overlaps, the algorithm checks for overlaps and merges segments if necessary. Each segment is either added to H or merged with the last segment in H, with these operations being performed in constant time for each segment, also giving a time complexity of $O(n)$.

After one list is fully traversed, the algorithm directly appends the remaining segments from the other list to H, considering any potential overlaps during this process. This involves iterating over the remaining segments and checking for overlaps with the last segment in H, ensuring all segments are correctly handled, which also results in a time complexity of $O(n)$.

Therefore, the overall time complexity, which is the sum of the complexities of each step, is $O(n)$, where n is the total number of segments.

2

a

Here's the description of the algorithm:

1. **Base Case**: If the set S contains only one segment, return this segment as the top skeleton.
2. **Divide**: Split the set S into two roughly equal halves: S_1 and S_2.
3. **Conquer**: Recursively compute the top skeleton for each half. Let $H1$ be the top skeleton for S_1 and $H2$ be the top skeleton for S_2.

4. **Combine**: Use the function `CombineSkeletons(`$H1, H2$`)` in the first part to merge the two top skeletons $H1$ and $H2$ into a single top skeleton H.

5. **Return**: Return the combined top skeleton H.

(b) Correctness of the Algorithm

The base case is correct because when the input set S has only one segment, the top skeleton is the segment itself. In the divide step, the set S is split into two halves, ensuring each segment is in either S_1 or S_2. During the conquer step, recursively computing the top skeletons for S_1 and S_2 correctly finds the top skeletons H_1 and H_2 for each half. The combine step uses the function `CombineSkeletons(`H_1, H_2`)` to merge 2 two top skeletons into a single top skeleton by handling overlaps and retaining the highest segments. Given that the algorithm correctly divides the problem into smaller subproblems and combines the results accurately, and considering the correctness of the base case, by induction, the entire algorithm will compute the correct top skeleton for the entire set S.

(c) Time Complexity

- **Divide Step**: Dividing the set S into two halves takes $O(1)$ time.
- **Conquer Step**: Recursively solving the subproblems for the two halves S_1 and S_2. Each subproblem is half the size of the original problem.
- **Combine Step**: The `CombineSkeletons` function runs in $O(n)$ time, where n is the total number of segments in the two halves being combined.

The recurrence relation for the time complexity $T(n)$ of the algorithm can be expressed as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the Master Theorem for divide-and-conquer recurrences of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a = 2$, $b = 2$, and $f(n) = O(n)$:

- $a = b^c$ where $c = 1$.
- Since $f(n) = O(n^c)$, this is Case 2 of the Master Theorem.

Therefore, the time complexity $T(n)$ is $O(n \log n)$.
Thus, the algorithm runs in $O(n \log n)$ time.

Solution 3.

a

1. **Divide**:

   According to the usage of a balanced splitter above, we partition the graph into three vertex sets A, B, and S, where A and B are treated as two subproblems.

2. **Conquer**:

   We use a global variable x to record the shortest cycle length, initialized to positive infinity. We recursively compute the shortest cycle length in the subgraph $G[A]$. Similarly, we recursively compute the shortest cycle length in the subgraph $G[B]$. If the number of vertices in the graph is less than or equal to 2, it means there is no cycle within the set (because this is a non-crossing graph), and we return (base case).

3. **Combine**:

   Since the shortest cycles within $G[A]$ and $G[B]$ have already been updated in the global variable x, we need to consider whether there is a shorter cycle in $G[A \cup S \cup B]$.

   To do this, we calculate the shortest cycle length that passes through vertices in the set S and update x. We use a modified Dijkstra's algorithm for this purpose:

   For each vertex in the set S, we run Dijkstra's algorithm. Each time, we extract the vertex u with the smallest distance value from the priority queue and add it to the set. We then check whether vertex u can directly return to the starting vertex. If it can, it indicates that a cycle is formed through the vertices in set S. If the length of this cycle is shorter than x, we update x.

   After performing the above process for each vertex in G_S, we merge the three vertex sets $A \cup S \cup B$. This ensures that the shortest cycle length in $G[A \cup S \cup B]$ is reflected in x.

After completing the recursion, the value stored in x is the shortest cycle length in graph G.

b

We use an invariant to prove the correctness of the algorithm. The invariant is defined as follows:

Invariant: At the end of each recursive call, the value stored in x is the length of the shortest cycle in the current subgraph.

Base Case

When the number of vertices in the graph is less than or equal to 2, no cycle exists, thus the invariant holds.

Recursive Steps

1. **Divide**:

   The graph G is divided into subsets A, B, and S. In each recursive call, the invariant ensures that the value stored in x is the length of the shortest cycle in the subgraphs $G[A]$ and $G[B]$.

2. **Combine**:

   For the subgraphs $G[A]$ and $G[B]$, the shortest cycles have been found recursively and the global variable x has been updated.

   Next, we consider cycles passing through the vertices in the set S. By using the modified Dijkstra's algorithm, we calculate and update the shortest cycle length passing through the vertices in S, and update x.

   Running the modified Dijkstra's algorithm for each vertex s in S ensures that all possible cycles passing through S are considered. Since we start from each vertex in S and find the shortest path from that vertex and back, we cover all possible cycles passing through vertices in S. The properties of Dijkstra's algorithm ensure that we find the shortest path from the start vertex s to all other vertices. Thus, when we check if we can return from u to the start vertex s, we are looking for the shortest path that goes through the set of vertices S and returns to the starting vertex s. This ensures that the cycle we find is the shortest.

   In each recursive call, by running the algorithm on $G[A]$ and $G[B]$, we find and update the shortest cycle length in the subgraphs and update the global variable x. In the combine step, by running the modified Dijkstra's algorithm on each vertex in S, we ensure that all cycles passing through the set of vertices S are considered and the shortest one is recorded in the global variable x.

   The combine step ensures that the shortest cycle length for the entire graph G has been correctly recorded in x.

By recursively applyingabove steps, we ensure that the value stored in the global variable x is the length of the shortest cycle in the entire graph G.

Therefore, after completing all recursive calls, the value in the variable x is the length of the shortest cycle in the graph G, proving the correctness of the algorithm.

c

- **Divide Step** In the divide step, we use a balanced splitter to divide the graph into three vertex sets A, B, and S, where A and B are the two subproblems. The time for this step is given by the problem as $O(n)$.

- **Conquer Step** In the conquer step, we recursively compute the shortest cycle length in the subgraphs $G[A]$ and $G[B]$, and store the result in a global variable x. Due to the property of the balance splitter, in the worst case, one of A and B is of size $\frac{2}{3}n$ while the other is of size $\frac{1}{3}n$, resulting in the maximum tree height.

- **Combine Step** In the combine step, we need to compute the shortest cycle length passing through the vertex set S and update the global variable x. For this, we run a modified Dijkstra's algorithm on each vertex in the set S.

  Since $|E|$ is bounded by $O(n)$, with n being the total number of vertices, each run of Dijkstra's algorithm has a time complexity of $O(n \log n)$. Additionally, we need to check if the vertex u can directly return to the starting vertex, which involves checking $n-1$ vertices, resulting in a process of $O(n-1)$. Therefore, each run of Dijkstra's algorithm has a time complexity of $O(n \log n)$. Considering that the set S has $O(\sqrt{n})$ vertices, the time complexity is $O(n^{3/2} \log n)$.

- **Overall Time Complexity**

$$T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + O(n^{\frac{3}{2}} log n)$$