# Chapter 2:

## 2.5.1

Create an empty stack.
Iterate through each element in the given array.
Push each element onto the stack.
Iterate through each element in the given array again.
Pop elements from the stack and place them back into the array, in reverse order.
Return the modified array.
The running time of this algorithm is linear or O(n), where 'n' represents the length of the array. This is because both for-loops iterate through the array once, performing constant-time operations for each element. Therefore, the time complexity of the algorithm is proportional to the size of the input array.

## 2.5.2

Create an empty queue.
Iterate through each element in the given array.
Enqueue each element into the queue.
Iterate through each element in the given array again, starting from the end.
Dequeue elements from the queue and place them back into the array, in reverse order.
Return the modified array.
The running time of this algorithm is linear or O(n), where 'n' represents the length of the array. This is because both for-loops iterate through the array once, performing constant-time operations for each element. Thus, the time complexity of the algorithm is proportional to the size of the input array.

## 2.5.8

(a) To prove that for every node in a binary tree, the level number of the node is less than or equal to the number of nodes in the tree, we can use induction. The base case is when the node is the root, where the level number is 1 and the number of nodes is at least 1, satisfying the inequality. For the inductive step, assuming the statement holds for a binary tree with k nodes, we consider a binary tree with k+1 nodes. By splitting the tree into left and right subtrees, we can observe that the maximum level number of the root node is 1 + max(left subtree level number, right subtree level number). Since the maximum level number of both subtrees is less than or equal to k, it follows that the maximum level number of the root node is less than or equal to k+1. Therefore, the statement holds for all nodes in the binary tree.

b)
An example of a binary tree with at least five nodes that achieves the upper bound on the maximum level number for some node is:

```
     1
    / \
   2   3
  / \
 4   5
```

In this tree, the maximum level number occurs at the root node, which is level 1. The number of nodes in the tree is also 5, matching the upper bound on the maximum level number.

**2.5.14**

If the current permutation has reached length n, print it as a valid permutation.
Otherwise, for each number i from 1 to n, do the following:
If i is not already used in the current permutation:
Add i to the current permutation.
Recursively call the permutation algorithm with the updated permutation.
Remove i from the current permutation to backtrack and explore other possibilities.
The running time of this algorithm is O(n!), where n is the total number of elements. This is because there are n! permutations of n elements, and the algorithm explores each possible permutation exactly once. The recursion depth is n, and at each level, we have n choices to consider. Hence, the total number of recursive calls is n * n * ... * n (n times), resulting in a time complexity of O(n!).

**2.5.32**

Start from the root of the tree.
If the root is either employee p or employee q, return the root as the LCA.
Recursively search for employee p and employee q in the left subtree of the root.
Recursively search for employee p and employee q in the right subtree of the root.
If employee p is found in the left subtree and employee q is found in the right subtree, or vice versa, return the root as the LCA.
If both employee p and employee q are found in the left subtree, recursively call the algorithm on the left subtree.
If both employee p and employee q are found in the right subtree, recursively call the algorithm on the right subtree.
The running time of this algorithm is O(n), where n is the number of nodes in the organizational tree. In the worst case, we may have to visit all nodes in the tree to find the LCA. The algorithm traverses the tree in a depth-first manner, visiting each node once.

**2.5.33**

Initialize a global variable maxDistance to 0.
Define a recursive function height(node) that calculates and returns the height of the subtree rooted at node.
If node is null, return 0.

Calculate the height of the left subtree by recursively calling height(node.left).
Calculate the height of the right subtree by recursively calling height(node.right).
Update the maxDistance by calculating the maximum of the sum of the heights of the left and right subtrees plus 2 (to account for the distance between node.left and node.right) and the current maxDistance.
Return the maximum height between the left and right subtrees plus 1.
Call the height(root) function to calculate the heights of the subtrees and update the maxDistance.
The value of maxDistance will now contain the diameter of the organizational tree.
The running time of this algorithm is O(n), where n is the number of nodes in the organizational tree. The algorithm performs a depth-first traversal of the tree, visiting each node once, and computes the height of each subtree in a bottom-up manner.

# Chapter 3:

## 3.5.8

Professor Amongus is incorrect in claiming that the order of element insertion does not matter in a binary search tree. The resulting tree can indeed vary based on the order in which elements are inserted. To demonstrate this, let's consider the example with the elements [2, 1, 3]. We will insert these elements into a binary search tree in two different orders and observe the resulting trees.
[2,1,3] and [2,3,1]

```
   2
  / \
 1   3
```

## 3.5.18

Base case: For n = 1, the tree consists of a single node, and the height is 0. Since log(1+1) = log(2) = 1, the base case holds. Inductive hypothesis: Assume that for a binary search tree containing k items, the height is at least log(k+1). Inductive step: Consider a binary search tree with n = k+1 items. We want to show that the height of this tree is at least log(n+1). In a binary search tree, the height is determined by the longest path from the root to a leaf node. Let's divide the tree into two subtrees, the left subtree, and the right subtree, with a root node in the middle. The number of items in the left subtree can range from 0 to n-1, and similarly for the right subtree. To maximize the height, we want the number of items in the left and right subtrees to be as close to each other as possible. The worst-case scenario for height occurs when the left and right subtrees have an equal number of elements. Assuming both subtrees contain k/2 elements each, by the inductive hypothesis, the height of each subtree is at least log((k/2)+1). Therefore, the height of the entire tree with n = k+1 items is at least log((k/2)+1) + 1. Simplifying the expression, log((k/2)+1) + 1 equals log((k+1)+1).

**3.5.32**

Create an empty list or array called "sequence" to store the resulting sequence of surviving soldiers.

Initialize a variable 'start' as 0 to keep track of the starting position in the circle.

Iterate from 1 to n:

Calculate the position of the soldier to be eliminated as 'elimination_position' using the formula: elimination_position = (start + k - 1) % i.

Add the soldier at 'elimination_position' to the 'sequence' list.

Update 'start' to 'elimination_position' + 1 for the next iteration.

Return the 'sequence' list containing the resulting sequence of surviving soldiers.

The time complexity of this algorithm is O(n) because we iterate through the 'n' soldiers once, performing constant-time operations at each step. Thus, the algorithm has a linear time complexity with respect to the number of soldiers.

# Chapter 4:

**4.7.21**

The left chain is a specific binary tree configuration where each internal node has a left child, and each right child (if present) is an external node (null).To prove this, we can use the following steps:

Start with the original binary tree.

Identify the rightmost internal node in the tree.

Perform a right rotation at this internal node.

Repeat steps 2 and 3 until the entire tree is transformed into a left chain.

Let's assume we have a binary tree with n nodes. Each rotation can transform one internal node into a left chain configuration, reducing the total number of internal nodes by one. Since the initial binary tree has n internal nodes, performing n-1 rotations will transform the entire tree into a left chain, as each rotation eliminates one internal node until only the root remains.

**4.7.27**

Consider an AVL tree where the root has a key of 9. The left subtree of the root contains nodes with keys 5, 3, 2, 4, 7, 6, and 8, arranged in a balanced manner. The right subtree of the root contains nodes with keys 12 and 15. Now, suppose we want to remove a leaf node with a key of 2 from this AVL tree. After removing the node, the tree becomes unbalanced, violating the height-balance property of AVL trees. To restore the height-balance property, a sequence of trinode restructurings (rotations) from the leaf to the root is required. In this case, the exact sequence of rotations will depend on the specific implementation rules of the AVL tree. However, in general, the trinode restructurings involve performing rotations at specific nodes along the path from the removed leaf to the root. These rotations help balance the tree and ensure that the height-balance property is satisfied.

**4.7.42**

Adding a Dog:
When a new dog is added to the website, create a new web page for the dog.
Determine the correct position for the dog based on its age.
Traverse the doubly linked list to find the appropriate location where the dog should be inserted.
Insert the new dog's web page into the doubly linked list, maintaining the age-based ordering.
Update the links of the neighboring nodes to maintain the proper links between adjacent dogs.
Removing a Dog:
When a dog is placed in a loving home and no longer waiting for adoption, locate its web page in the doubly linked list.
Update the links of the neighboring nodes to bypass the dog's web page.
Remove the dog's web page from the doubly linked list, effectively removing it from the website.
By using a doubly linked list, both adding and removing dogs can be done in O(1) time complexity. Inserting a dog at the correct position involves updating a few pointers, and removing a dog simply requires adjusting the links of the neighboring nodes.
The running time of these methods in terms of the number of dogs currently displayed on the website (n) is O(1) because the operations do not depend on the total number of dogs.

**4.7.43**

Data Structure:
Use a balanced binary search tree (such as a self-balancing binary search tree like AVL tree or Red-Black tree) to maintain the set of employees in alphabetical order based on their names.
Each node in the binary search tree should contain the employee's name, address, and the number of shares of stock promised by the CEO.
Insertion and Removal:
To insert an employee, perform a standard insertion operation in the binary search tree, which takes O(log n) time, where n is the number of employees in FastCo.
To remove an employee, perform a standard deletion operation in the binary search tree, also taking O(log n) time.
Listing Employees in Alphabetical Order:
Perform an inorder traversal of the binary search tree, which visits nodes in ascending order based on names.
During the traversal, print the employee's information, including the number of shares promised by the CEO.
This listing operation takes O(n) time, where n is the number of employees in FastCo, as we need to visit each node once.
Processing Friday Promises:
Maintain a global variable to store the total number of shares promised by the CEO.
When processing a Friday promise, traverse the entire binary search tree, incrementing the number of shares promised to each employee by the Friday's share value.
This update operation takes O(n) time, as we need to update the shares for each employee in FastCo.

# Chapter 5:

## 5.7.8

In a max heap, the item with the largest key is stored at the root of the heap. The root node of a max heap always contains the maximum element in the heap. This property is known as the max heap property, which states that for any node 'i' in the heap, the key of 'i' is greater than or equal to the keys of its children. By maintaining the max heap property, the maximum element is always present at the root, making it easily accessible with a time complexity of O(1). This allows for efficient retrieval of the item with the largest key in the heap. It is important to note that the other elements in the heap do not have any specific order among themselves. They are arranged based on the max heap property, where each parent node is greater than or equal to its children. However, the relative ordering among the other elements may vary.

## 5.7.12

Σ(log i) from i=2 to n = log 2 + log 3 + log 4 + ... + log n
Since log i is an increasing function, we can lower bound each term of the sum by log(n/2). This is because for every i from 2 to n, we have i ≥ n/2.
Therefore, we have:
Σ(log i) from i=2 to n ≥ (n - 1) * log(n/2)
Now, let's analyze the term (n - 1) * log(n/2) further:
(n - 1) * log(n/2) = (n - 1) * (log n - log 2) = (n - 1) * log n - (n - 1) * log 2
The term (n - 1) * log n dominates the term (n - 1) * log 2. Hence, we can disregard the latter term in the lower bound.
Therefore, we have:
Σ(log i) from i=2 to n ≥ (n - 1) * log n

## 5.7.20

To show that, for any n, there is a sequence of insertions in a heap that requires Ω(nlogn) time to process, we can construct a specific example and analyze its time complexity. Consider a heap that is initially empty. We will insert n/2 elements with values equal to n, followed by n/2 elements with values equal to 1. This sequence ensures that each insertion will cause a sift-up operation to maintain the heap property. Let's analyze the time complexity of this sequence of insertions: For the n/2 elements with values n, each insertion requires log(n) operations to sift up. Therefore, the total number of operations for these insertions is (n/2) * log(n). Similarly, for the n/2 elements with values 1, each insertion also requires log(n) operations to sift up. Hence, the total number of operations for these insertions is also (n/2) * log(n). Therefore, the total number of operations for all the insertions is (n/2) * log(n) + (n/2) * log(n) = n * log(n), which is Ω(nlogn).

**5.7.27**
Initialize an empty max heap (leftHeap) and an empty min heap (rightHeap). When an insert operation is called with a value 'x':
Compare 'x' with the current median.
If 'x' is less than the median, insert 'x' into the leftHeap.
If 'x' is greater than or equal to the median, insert 'x' into the rightHeap.
Ensure that the leftHeap has the lower half of the numbers, and the rightHeap has the upper half (or equal number of elements if n is even).
Balance the heaps to maintain the property that the difference in size between the two heaps is at most 1. If needed, move the root of the larger heap to the other heap until the balance condition is satisfied.
When the median operation is called:
If the number of elements in the leftHeap is greater than the rightHeap, return the root of the leftHeap as the median.
If the number of elements in the rightHeap is greater than the leftHeap, return the root of the rightHeap as the median.
If the number of elements in both heaps is equal, return the average of the roots of the leftHeap and rightHeap as the median.
By using two heaps and balancing them as described, the insert operation can be performed in O(log n) time as we need to maintain the balance of the heaps. The median operation can be done in O(1) time as we can directly access the root of the respective heap.

# Chapter 6:

**6.6.2**
Initialize an empty max heap (leftHeap) and an empty min heap (rightHeap).
When an insert operation is called with a value 'x':
Compare 'x' with the current median.
If 'x' is less than the median, insert 'x' into the leftHeap.
If 'x' is greater than or equal to the median, insert 'x' into the rightHeap.
Ensure that the leftHeap has the lower half of the numbers, and the rightHeap has the upper half (or equal number of elements if n is even).
Balance the heaps to maintain the property that the difference in size between the two heaps is at most 1. If needed, move the root of the larger heap to the other heap until the balance condition is satisfied.
When the median operation is called:
If the number of elements in the leftHeap is greater than the rightHeap, return the root of the leftHeap as the median.
If the number of elements in the rightHeap is greater than the leftHeap, return the root of the rightHeap as the median.
If the number of elements in both heaps is equal, return the average of the roots of the leftHeap and rightHeap as the median. By using two heaps and balancing them as described, the insert operation can be performed in O(log n) time as we need to maintain the balance of the heaps.

The median operation can be done in O(1) time as we can directly access the root of the respective heap.

**6.6.10**

Base Case: We will first verify the base case where n = 2:
T(2) = T(2/2) + 2/2 = T(1) + 1 = 1 + 1 = 2
Inductive Hypothesis: Assume that for some k ≥ 2, T(k) = T(k/2) + k/2 holds true.
Inductive Step: We will show that T(n) = T(n/2) + n/2 holds for n = 2^k, where k ≥ 2.
T(n) = 2T(n/2) + n
Using the inductive hypothesis for n = n/2, we can substitute T(n/2) with T((n/2)/2) + (n/2)/2:
T(n) = 2(T(n/4) + (n/2)/2) + n
Simplifying further:
T(n) = 2T(n/4) + (n/2) + n
= 2T(n/4) + (3n/2)
We can observe that the term 2T(n/4) + (3n/2) is equivalent to T(n/2) + n/2.
Therefore, we can rewrite the equation as:
T(n) = T(n/2) + n/2

**6.6.38**

Base Case: We will first verify the base case where n = 1:
T(1) = T(1-1) + 2^1 = T(0) + 2 = 2
Inductive Hypothesis: Assume that for some k ≥ 1, T(k) = T(k-1) + 2^k holds true.
Inductive Step: We will show that T(n) = T(n-1) + 2^n holds for n ≥ 2.
T(n) = T(n-1) + 2^n
Using the inductive hypothesis for n = n-1, we can substitute T(n-1) with T((n-1)-1) + 2^(n-1):
T(n) = T(n-2) + 2^(n-1) + 2^n
Simplifying further:
T(n) = T(n-2) + 2^(n-1) + 2^n
= T(n-2) + 2^(n-1) + 2^(n-1) * 2
= T(n-2) + 2^(n-1) + 2^n
= T(n-2) + 2^n
We can observe that the term T(n-2) + 2^n is equivalent to T(n-1) + 2^n.
Therefore, we can rewrite the equation as:
T(n) = T(n-1) + 2^n

**6.6.57**

Initialize a variable count to 0, which will keep track of the total count of 1's.
For each row row in the array:
Perform a modified binary search to find the first occurrence of 0 in the row.
Update the count by adding the index of the first occurrence of 0 to it. Since the elements before this index are all 1's, this index represents the count of 1's in that row. The modified binary search can be implemented as follows:

Initialize two pointers, left and right, to the start and end of the row.
While left is less than right:
Calculate the middle index as (left + right) / 2.
If the value at the middle index is 0, update right to be the middle index.
If the value at the middle index is 1, update left to be the middle index + 1.
After the loop, left will point to the first occurrence of 0 in the row.
By applying this modified binary search for each row, we count the number of 1's efficiently. The time complexity of this method is O(n), where n is the total number of elements in the array. Since we only traverse each row once and perform binary search on each row, the overall time complexity is linear.

## 6.6.74

Initialize two variables, missing1 and missing2, to 0. These variables will store the two missing integers. Iterate through each element num in the array A: Calculate the absolute value of num and store it in the variable index (to handle negative numbers, if any).
Set the value at index in the array A to negative to mark its presence (e.g., A[index] = -A[index]).
Iterate through each index in the range from 1 to n: If the value at index in A is positive, it means that index is missing in the array. Update missing1 if missing1 is 0, else update missing2.
Reset the value at index to its absolute value to restore the original array. Return missing1 and missing2 as the two integers that are not present in the array A. This approach works based on the principle that we can use the sign of the array elements as a marker to keep track of the presence or absence of certain integers. By negating the value at a specific index, we can mark its presence, and if a value at an index remains positive after scanning the array, it indicates that the corresponding index is missing.

# Chapter 7

## 7.7.2

To explain why O(log m) is O(log n), we need to establish the relationship between the number of edges (m) and the number of vertices (n) in a simple connected graph. In a simple connected graph, the number of edges (m) is at most n*(n-1)/2. This is because each vertex can be connected to at most n-1 other vertices, and there are n vertices in total. So, the number of possible edges is in the range [0, n*(n-1)/2]. Now, let's consider the relationship between the logarithms of m and n:
log m = log(n*(n-1)/2) = log n + log(n-1) - log 2
In the above equation, log n dominates log(n-1) and log 2. Therefore, we can ignore the smaller terms, and we can conclude that log m is dominated by log n. Hence, O(log m) is O(log n). This means that any algorithm or function with a time complexity of O(log m) is also O(log n), as it falls within the same logarithmic growth rate.

**7.7.10**
Start with an empty list, which will eventually store the topological ordering of the vertices.
Visit a vertex that has no incoming edges (indegree = 0).
Add the visited vertex to the list.
Remove all outgoing edges from the visited vertex.
Repeat steps 2-4 until all vertices are visited.

**7.7.25**
To prove the statement, let's assume that T is a BFS tree produced for a connected graph G.
We will show that for each vertex v at level i, the path of T between the root vertex r and v has exactly i edges, and any other path of G between r and v has at least i edges.
Path of T between r and v has exactly i edges:
By the nature of the Breadth-First Search (BFS) algorithm, vertices at level i are visited after exploring all vertices at level i-1.
Since T is a BFS tree, each vertex is visited only once during the BFS traversal.
Therefore, when vertex v at level i is visited, it must have been discovered through an edge from some vertex u at level i-1.
This edge connects u and v in the BFS tree T, so the path from r to v in T consists of the edge (u, v) and the path from r to u in T.
The path from r to u in T has exactly i-1 edges, as u is at level i-1.
Therefore, the path of T between r and v has exactly i edges.
Any other path of G between r and v has at least i edges:
Suppose there exists another path P of G between r and v with fewer than i edges.
Since T is a BFS tree, the BFS algorithm explores vertices in a level-by-level manner, and vertices at level i are visited after exploring all vertices at level i-1.
If there is a path P with fewer than i edges, it implies that there is a vertex x on this path P that is at level less than i-1. However, this contradicts the property of the BFS algorithm, where vertices at level i-1 are visited before exploring vertices at level i.
Therefore, any other path of G between r and v must have at least i edges.

**7.7.35**
Check if the graph G is Eulerian. An Eulerian graph has either zero or two vertices with an odd degree. If there are zero odd-degree vertices, the graph is Eulerian, and we can start at any vertex. If there are two odd-degree vertices, we need to start at one of them.
If the graph is not Eulerian, it is not possible to design a tour that traverses each edge exactly once in each direction.
If the graph is Eulerian, we can find an Eulerian path using the Hierholzer's algorithm:
a. Initialize an empty stack and an empty tour. b. Choose a starting vertex (either the one given or one of the odd-degree vertices, if applicable) and push it onto the stack.
c. While the stack is not empty:
If the current vertex has any unvisited outgoing edges, choose one, push the destination vertex onto the stack, and remove the edge.
If the current vertex has no unvisited outgoing edges, add the current vertex to the tour.

Pop the top vertex from the stack and set it as the current vertex.
d. Reverse the tour if necessary to ensure that each edge is traversed in both directions.

## 7.7.38

Choose an arbitrary node v in the tree T as the starting point.
Perform a depth-first search (DFS) from node v to find the farthest node u in T. Keep track of the maximum distance d1 found during this search.
Start DFS from node v and maintain a distance counter initialized to 0.
Traverse the tree T using DFS, keeping track of the maximum distance from v to any visited node. Update the maximum distance whenever a longer path is found.
Once the DFS is complete, node u will be the farthest node from v, and d1 will be the maximum distance between v and u.
Perform another DFS from node u to find the farthest node w in T. Keep track of the maximum distance d2 found during this search.
Start DFS from node u and maintain a distance counter initialized to 0.
Traverse the tree T using DFS, keeping track of the maximum distance from u to any visited node. Update the maximum distance whenever a longer path is found.
Once the DFS is complete, node w will be the farthest node from u, and d2 will be the maximum distance between u and w.The diameter of the tree T will be d2, which is the maximum distance between any two nodes in T.The time complexity of this algorithm is O(|V| + |E|), where |V| is the number of vertices (nodes) in the tree T and |E| is the number of edges. Both the depth-first searches are performed once, visiting each vertex and edge once.

# Chapter 8:

## 8.5.10

Lemma 1: The two least frequent symbols in the alphabet can be assigned the two longest codewords in the Huffman code. This lemma is typically proved using a lower-bound argument. It shows that if two symbols have the lowest frequencies, assigning them the longest codewords maximizes the average codeword length and thus minimizes the total expected code length.
Lemma 2: For any two symbols with the smallest frequencies, swapping their codewords in the Huffman code results in a code with a smaller average codeword length. This lemma uses an exchange argument. It states that if two symbols with the smallest frequencies have been assigned codewords such that swapping their codewords reduces the average codeword length, then the original assignment was not optimal. Lemma 3: For any internal node in the Huffman tree, swapping the subtrees rooted at that node results in a code with a smaller average codeword length. This lemma also uses an exchange argument. It states that if swapping the subtrees rooted at an internal node in the Huffman tree leads to a decrease in the average codeword length, then the original assignment of codewords was not optimal.

**8.5.11**
Consider a knapsack with a capacity of 10 units and the following items:
Item 1: Weight = 5 units, Value = $10 (Value per unit weight = $2 per unit)
Item 2: Weight = 7 units, Value = $14 (Value per unit weight = $2 per unit)
Item 3: Weight = 9 units, Value = $20 (Value per unit weight = $2.22 per unit)
Using the greedy strategy of selecting the highest-benefit item as much as possible, we would choose: Choose Item 3 partially, taking 4 units. (Benefit = $8, Weight = 4 units, Remaining Capacity = 6 units)
Choose Item 2 completely. (Benefit = $22, Weight = 11 units, Remaining Capacity = -1 units)
The greedy strategy results in a total benefit of $22. However, the optimal solution is:
Choose Item 1 completely. (Benefit = $10, Weight = 5 units, Remaining Capacity = 5 units)
Remaining Capacity = 5 units (Not enough capacity to fit Item 2 or Item 3)
The optimal solution has a total benefit of $10, which is lower than the solution obtained through the greedy strategy.

**8.5.14**
Initialize "currentPosition" to the starting point and "remainingDistance" to the maximum distance Anatjari can walk on one bottle of water.
Iterate through the watering holes on the map, starting from the first one.
For each watering hole:
a. Calculate the distance "distanceToNextHole" between the current position and the watering hole.
b. If "distanceToNextHole" is greater than "remainingDistance", refill the water bottle at the current watering hole and update "currentPosition" and "remainingDistance".
The visited watering holes represent the optimal refill points for Anatjari to make as few stops as possible.
The algorithm is correct because it prioritizes refilling the water bottle only when the remaining distance is not enough to reach the next watering hole. This ensures Anatjari can continue traveling until the next refill point without needing extra stops. By maximizing the distance traveled before each refill, the algorithm minimizes the number of stops required.

**8.5.24**
Initialize an array "dp" of size n+1, where n is the number of words in the sequence. Each element of the array represents the penalty for breaking the words up to that point.
Initialize an array "breaks" of size n+1, where breaks[i] represents the last word index in the line ending at word i.
Set dp[0] = 0 since there are no words to break.
Iterate through the words from 1 to n:
a. Initialize a variable "lineLength" to the length of the current word.
b. Initialize a variable "minPenalty" to infinity.
c. Iterate from the current word index back to 1:

Update lineLength by adding the length of the previous word and 1 (for the space between words).
If lineLength exceeds the line length limit L, break the loop.
Calculate the penalty for this line break using the badness formula: (L - lineLength)^3.
Update minPenalty and breaks[i] if the penalty is smaller than the previous minimum.
Increment the penalty by dp[j-1] to consider the penalty of previous line breaks.
d. Set dp[i] = minPenalty.
Construct the line breaks sequence by following the breaks array from n to 1, storing the indices where line breaks occur.
To prove the optimality of the algorithm, we can use the concept of subproblems and optimal substructure. The Knuth-Plass algorithm breaks the problem into smaller subproblems, where the optimal solution to each subproblem contributes to the optimal solution of the overall problem.
The running time of the algorithm is O(n^2), where n is the number of words. This is because for each word, we iterate through all previous words to find the optimal line break, resulting in a nested loop structure.

## 8.5.25

In the line breaking problem with a penalty for line breaks defined as (L - lineLength)^(3/2), a greedy strategy of scanning the sequence from beginning to end and choosing the option that minimizes the penalty based on previous choices while maintaining each line length at most L does not necessarily result in an optimal set of line breaks. A counterexample can be constructed where the greedy strategy leads to a higher penalty than an alternative set of line breaks. By carefully selecting words and line lengths, it is possible to demonstrate cases where the greedy strategy makes suboptimal choices. Therefore, it is important to note that the specific penalty formula used in this scenario does not satisfy the necessary conditions for the greedy strategy to guarantee an optimal solution. Alternative approaches or algorithms would be required to obtain the optimal set of line breaks.

# Chapter 9:

## 9.5.3

Start by initializing two empty data structures: a dictionary called dist to store the distance from the source vertex to each vertex in the graph, and another dictionary called tree to store the parent vertices in the shortest path tree.
Create a priority queue (e.g., a min-heap) to store vertices based on their tentative distances.
Set the distance of the source vertex to 0 and add it to the priority queue.
Repeat the following steps until the priority queue is empty:
Remove the vertex u with the minimum distance from the priority queue.
For each neighbor v of u, calculate the distance alt from the source vertex to v through u. If this distance is smaller than the current distance stored in dist[v], update dist[v] with alt and set tree[v] to u.

Add v to the priority queue with its updated distance.
After the algorithm finishes, the dist dictionary will contain the shortest distances from the source vertex to each vertex in the graph, and the tree dictionary will represent the shortest path tree rooted at the source vertex.
The modified Dijkstra's algorithm ensures that the paths in the tree dictionary from the source vertex to any other vertex in the graph are actually the shortest paths from the source vertex to those vertices in the original graph.

**9.5.12**
Let's assume the start vertex is A and the goal vertex is D. Using the given greedy strategy, the algorithm would proceed as follows:
Initialize path to A and VisitedVertices to {A}.
Since start = goal, the algorithm terminates and returns the path as [A].
However, the actual shortest path from A to D in this graph is [A, B, D] with a total weight of 4.
The greedy strategy fails to find this shortest path because it only considers the immediate minimum-weight edge at each step, without considering the overall path length.
The reason why the greedy strategy fails is that it lacks the ability to explore all possible paths and make informed decisions based on the cumulative path lengths. It may get trapped in suboptimal paths early on and miss the opportunity to find shorter paths.

**9.5.17**
Initialize a priority queue pq to store vertices based on their tentative costs and distances.
Initialize a dictionary dist to store the minimum cost to reach each vertex. Set the cost of the starting vertex to 0 and add it to the priority queue.
Initialize a dictionary prev to store the previous vertex in the shortest path for each vertex. Set the previous vertex of the starting vertex to None.
While the priority queue is not empty, do the following steps:
Remove the vertex u with the minimum cost from the priority queue.
For each neighbor v of u that is reachable by a left-to-right movement, calculate the cost alt to reach v via u. If alt is less than the current cost stored in dist[v], update dist[v] with alt and set prev[v] to u. Add v to the priority queue with its updated cost. Once the algorithm finishes, reconstruct the minimum-cost monotone path from the prev dictionary starting from the ending vertex and following the previous vertices until reaching the starting vertex.
The time complexity of this algorithm depends on the implementation of the priority queue. If a binary heap is used, the time complexity is typically O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph. However, with the Fibonacci heap, the time complexity can be improved to O(V log V + E).

**9.5.18**
Create a graph,

, where each station is represented as a vertex and each communication channel is represented as an edge. Assign a weight of 0 to the edges that have not been compromised and a weight of 1 to the edges that have been compromised. Define the starting station, , as the source vertex and the field station, , as the destination vertex. Find the shortest path from to in the graph using any efficient shortest-path algorithm such as Dijkstra's algorithm or Bellman-Ford algorithm. Since the weights represent the level of compromise, the shortest path from to will correspond to the path that minimizes the number of compromised edges along the way. By solving the shortest-path problem on the modified graph, we can find the path that minimizes the number of compromised edges and ensures the super-secret message is transmitted with minimal risk of interception. The efficiency of the algorithm depends on the chosen shortest-path algorithm. Dijkstra's algorithm has a time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges. The Bellman-Ford algorithm has a time complexity of O(V * E). The choice of the algorithm depends on the size and characteristics of the communication network.

**9.5.20**
Create a graph where each airport is represented as a vertex. Add directed edges between airports for each flight, considering the minimum connecting times at intermediate airports. Assign weights to the edges based on the departure and arrival times of the flights. The weight of an edge from airport A to airport B represents the time it takes to travel from A to B, including the waiting time at intermediate airports. Initialize a priority queue pq to store vertices based on their tentative arrival times. Initialize a dictionary arrival_time to store the earliest arrival time at each airport. Set the arrival time of the starting airport to the given departure time. While the priority queue is not empty, do the following steps:
Remove the airport u with the minimum tentative arrival time from the priority queue.
For each neighboring airport v of u, calculate the arrival time alt at v by considering the departure time, arrival time, and minimum connecting time of the corresponding flight.
If alt is earlier than the current arrival time stored in arrival_time[v], update arrival_time[v] with alt and add v to the priority queue with its updated arrival time.
Once the algorithm finishes, the arrival_time dictionary will contain the earliest arrival time at each airport when departing from the starting airport at or after the given time.

# Chapter 10:

**10.6.8**
Kruskal's algorithm: Kruskal's algorithm sorts the edges in ascending order of their weights and progressively adds them to the MST as long as they do not create a cycle. Negative-weight edges or cycles do not affect the correctness of Kruskal's algorithm because the algorithm only considers adding edges that do not create cycles. If a negative-weight edge or cycle exists, the algorithm simply avoids including it in the MST. As a result, the algorithm will find an MST that includes only the edges with non-negative weights.Prim's algorithm: Prim's algorithm starts with an arbitrary vertex and grows the MST by iteratively adding the shortest edge from the current

MST to a new vertex. The algorithm maintains a set of visited vertices and considers the shortest edges that connect the visited vertices to the unvisited vertices. Negative-weight edges or cycles do not affect the correctness of Prim's algorithm because the algorithm always selects the shortest edge that connects the visited vertices to the unvisited vertices. It does not matter if there are negative-weight edges or cycles; the algorithm will only consider the shortest non-negative-weight edge at each step.

**10.6.11**
Let's suppose there is an MST T that does not contain the edge (u, v).
Since T is a spanning tree, it includes a path between u and v. Let's denote this path as P in T. Removing any edge e from P will create two disjoint sets of vertices, X and Y, where u is in X and v is in Y. Since (u, v) is the smallest-weight edge in the graph, its weight is less than or equal to the weight of any other edge in P. We can replace the edge e in P with the edge (u, v) without creating a cycle since (u, v) is not in T. The resulting tree T' is still connected and acyclic. The weight of T' is equal to the weight of T minus the weight of e plus the weight of (u, v). Since the weight of (u, v) is less than or equal to the weight of e, the weight of T' is less than or equal to the weight of T. Therefore, T' is a spanning tree with a weight less than or equal to T, contradicting the assumption that T is an MST. This means that our initial assumption that there is an MST T that does not contain (u, v) is false. Hence, there exists an MST of the graph that includes the smallest-weight edge (u, v).

**10.6.22**

To find a maximum spanning tree in the given graph, where the weight of each edge represents the amount of money the CIA will pay, we can modify the Kruskal's algorithm to achieve the desired result. Here's the algorithm:
Sort the edges of the graph in descending order based on their weights (amount of money the CIA will pay). Initialize an empty set max_spanning_tree to store the maximum spanning tree. Iterate through the sorted edges from the highest weight to the lowest weight. For each edge (u, v), check if adding it to the max_spanning_tree would create a cycle. If not, add the edge to the max_spanning_tree. Continue iterating through the edges until all vertices are included in the max_spanning_tree or all edges have been considered. Return the max_spanning_tree as the maximum spanning tree of the graph. The running time of this algorithm depends on the sorting step, which takes O(E log E) time, where E is the number of edges in the graph. Additionally, the algorithm performs a union-find operation to check for cycles, which can be done in O(log V) time on average, where V is the number of vertices in the graph. Therefore, the overall running time of the algorithm is O(E log E + E log V), which simplifies to O(E log E) or O(E log V) depending on the density of the graph.

**10.6.27**
Initialize a priority queue pq to store vertices based on their tentative costs. Initialize a dictionary key to store the minimum cost to reach each vertex. Set the cost of the starting vertex to 0 and

add it to the priority queue. Initialize a dictionary prev to store the previous vertex in the minimum-cost spanning tree for each vertex. Set the previous vertex of the starting vertex to None. While the priority queue is not empty, do the following steps:
Remove the vertex u with the minimum tentative cost from the priority queue.
For each neighboring vertex v of u, calculate the cost alt to reach v by considering the cost of the cable required to connect the rooms. If alt is less than the current cost stored in key[v], update key[v] with alt and set prev[v] to u. Add v to the priority queue with its updated cost.
Once the algorithm finishes, reconstruct the minimum-cost spanning tree from the prev dictionary.

# Chapter 12:

## 12.6.1

(a) T(n) = 9 * T(n/3) + n^2
In this case, we have a = 9, b = 3, and f(n) = n^2.
We can see that f(n) = n^2 is larger than n^log_b(a) = n^log_3(9) = n^2, so it falls into Case 3 of the master theorem.
Therefore, the solution to the recurrence equation (a) can be characterized as T(n) = Θ(n^2).

(b) T(n) = 2 * T(n/2) + n log n
In this case, we have a = 2, b = 2, and f(n) = n log n.
We can see that f(n) = n log n is asymptotically equal to n^log_b(a) = n^log_2(2) = n, so it falls into Case 2 of the master theorem.
Therefore, the solution to the recurrence equation (b) can be characterized as T(n) = Θ(n log^2 n).

(c) T(n) = 4 * T(n/2) + n^2 sqrt(n)
In this case, we have a = 4, b = 2, and f(n) = n^2 sqrt(n).
We can see that f(n) = n^2 sqrt(n) is larger than n^log_b(a) = n^log_2(4) = n^2, so it falls into Case 3 of the master theorem.
Therefore, the solution to the recurrence equation (c) can be characterized as T(n) = Θ(n^2 sqrt(n)).

(d) T(n) = 7 * T(n/3) + n^3
In this case, we have a = 7, b = 3, and f(n) = n^3.
We can see that f(n) = n^3 is smaller than n^log_b(a) = n^log_3(7), so it falls into Case 1 of the master theorem.
Therefore, the solution to the recurrence equation (d) can be characterized as T(n) = Θ(n^log_3(7)).

(e) T(n) = 2 * T(n/4) + √n
In this case, we have a = 2, b = 4, and f(n) = √n.

We can see that f(n) = √n is smaller than n^log_b(a) = n^log_4(2) = n^0.5, so it falls into Case 1 of the master theorem.
Therefore, the solution to the recurrence equation (e) can be characterized as T(n) = Θ(n^0.5).

**12.6.9**
Stooge-sort is correct and correctly sorts the input array in ascending order.
The running time of Stooge-sort can be characterized using the recurrence equation T(n) = 3 * T(2n/3) + O(1).
By applying the master theorem, the asymptotic bound for the running time of Stooge-sort is O(n^2.7095).

**12.6.12**
f the set S contains only one element, return that element as both the minimum and maximum. If the set S contains two elements, compare them and return the smaller element as the minimum and the larger element as the maximum. Divide the set S into two equal-sized subsets, S1 and S2. Recursively find the minimum and maximum elements of S1, let's call them min1 and max1, respectively. Recursively find the minimum and maximum elements of S2, let's call them min2 and max2, respectively. Compare min1 and min2 to determine the overall minimum, and compare max1 and max2 to determine the overall maximum. Return the overall minimum and maximum as the result.
The number of comparisons required can be calculated as follows:
For the base case of one element, no comparisons are needed.
For the base case of two elements, one comparison is needed.
In the divide step, we split the set into two halves, resulting in two recursive calls.
Therefore, the total number of comparisons is given by the recurrence relation: T(n) = 2T(n/2) + 2.

**12.6.16**
Create an empty bank of optimization registers. Insert all n numbers into the bank of registers, using the numbers as keys and assigning arbitrary values to them. Extract the key-value pairs from the bank of registers in ascending order of the keys. This can be done by repeatedly removing and returning the key-value pair with the smallest key until the bank of registers is empty. The extracted key-value pairs will be in sorted order based on the keys. The time complexity of this algorithm depends on the operations performed on the bank of registers:
Inserting a key-value pair into the bank of registers takes O(1) time.
Removing and returning the key-value pair with the smallest key from the bank of registers takes O(1) time.

**12.6.17**
If the set of sub-intervals is empty, return an empty skyline.

If there is only one sub-interval, create a skyline with two points: (a, h) and (b, 0), where a and b are the start and end points of the interval, and h is the associated height.
Divide the set of sub-intervals into two equal-sized subsets.
Recursively compute the skylines of the left and right subsets.
Merge the two skylines by iterating through the points and determining the maximum height at each x-coordinate.
Add the final point (b, 0) to the merged skyline.
Return the merged skyline.
The time complexity is O(n log n) because the recursion divides the problem into halves, and the merging step requires O(n) time.

# Chapter 13:

## 13.4.8

If the length of the input array is less than or equal to 1, it is already sorted, so return.
Divide the input array into two equal-sized subarrays. Recursively apply the merge-sort algorithm on each subarray. Merge the two sorted subarrays into the input array using the additional workspace array.
Initialize three pointers: left, right, and current.
Set the left pointer to the start of the first subarray.
Set the right pointer to the start of the second subarray.
Set the current pointer to the start of the input array.
While both subarrays have elements:
Compare the elements at the left and right pointers.
If the element at the left pointer is smaller or equal, copy it to the current position in the input array and move the left and current pointers to the next position.
Otherwise, if the element at the right pointer is smaller, copy it to the current position in the input array and move the right and current pointers to the next position.
If there are remaining elements in the first subarray, copy them to the remaining positions in the input array.
If there are remaining elements in the second subarray, copy them to the remaining positions in the input array.
Return the sorted input array.
By using this in-place merge-sort algorithm, we can sort the input array without using any additional memory other than the workspace array.
The time complexity of this algorithm remains the same as the traditional merge-sort, which is O(n log n), where n is the size of the input array.

## 13.4.9

Initialize an empty hash set, resultSet.
Iterate through each object in the collection,
. For each object, perform the following steps:

Check if the object is already present in resultSet using a hash function or comparison.
If the object is not present, add it to resultSet.
Once the iteration is complete, resultSet will contain a set of unique objects from the original collection. The running time of this method depends on the efficiency of the hash set operations, specifically the time complexity of insertion and searching. Insertion and searching operations in a hash set have an average time complexity of O(1). In the worst case scenario, if there are many collisions and hash table resizing is required, the time complexity can be O(n), where n is the number of objects in the collection. Therefore, the overall running time of this method is O(n) on average, but in the worst case, it can be O(n^2) or O(n) with some additional constant factors due to hash table resizing.

**13.4.28**
Initialize an array count of size 26 (assuming the alphabet contains only lowercase letters) to store the count of names starting with each letter. Iterate through the collection of names and increment the count of the corresponding letter based on the order array. Perform a cumulative sum on the count array to determine the starting index for each letter. Create a new array sortedNames of the same size as the collection of names. Iterate through the collection of names in reverse order and place each name in the sortedNames array at its corresponding index based on the count and cumulative sum. Return the sortedNames array.
The running time of this algorithm depends on the number of names, n, and the size of the order array, k. The initial iteration to increment the count for each name takes O(n) time. The cumulative sum on the count array takes O(k) time. The iteration to place the names in the sortedNames array takes O(n) time.
Therefore, the overall running time of this algorithm is O(n + k).

# Chapter 14:

**14.5.6**
If we were to form groups of size 3 instead of groups of size 5 in deterministic selection, the induction proof for showing that it runs in O(n) time would fail. The proof for the time complexity of deterministic selection relies on the fact that the size of the groups used for partitioning is a constant. By forming groups of size 5, we ensure a good approximation of the median within each group and a balanced partitioning of the input. This leads to efficient recursive calls and a total time complexity of O(n). If we were to form groups of size 3, the accuracy of the pivot element and the balance of the partitions would be compromised. The smaller group size increases the likelihood of outliers affecting the median calculation, resulting in a less accurate pivot element. This can lead to unbalanced partitions and potentially increase the running time of the algorithm.

**14.5.8**

Modify the comparison function to consider two elements as equal only if their keys and their original positions in the input are the same. This ensures that elements with the same key will be compared based on their original positions in the input. When comparing two elements with the same key, use an additional comparison to determine their original positions. This can be done by comparing the indices of the elements in the original input. By introducing this tie-breaking comparison based on the original positions of elements, the sorting algorithm will maintain stability. Elements with the same key will be sorted in the same relative order as they appeared in the input. This modification does not affect the asymptotic running time of the sorting algorithm. The additional comparison for tie-breaking is a constant-time operation, and the original comparison-based operations still dominate the overall time complexity, which remains unchanged.

**14.5.24**

Initialize a variable candidate to represent the current candidate with the most votes.
Initialize a counter count to keep track of the number of votes for the current candidate.
Iterate through the votes in the array votes.
If the count is 0, set the current candidate as the candidate and increment the count.
If the current vote matches the candidate, increment the count.
If the current vote is different from the candidate, decrement the count.
After the iteration, the candidate will hold the student number that potentially has the majority of votes.
Iterate through the votes array again to count the number of occurrences of the candidate and store it in a variable candidateCount.
If candidateCount is greater than n/2, the candidate has a majority of the votes. Return candidate as the winner.
If candidateCount is not greater than n/2, there is no candidate with a majority of the votes. Return null or a special value to indicate this.

# Chapter 15:

**15.8.10**

(a) When computing C(n, k) without using memoization and assuming n is even, we can observe that the recursive calls made for each value of k from 0 to n result in a significant number of distinct computations. For each recursive call, the formula for C(n, k) involves two additional recursive calls: C(n-1, k-1) and C(n-1, k). This branching leads to a large number of distinct recursive calls being made. In the case where n is even, we can express n as 2m. As we compute C(n, k) for each value of k from 0 to n, the number of distinct recursive calls can be represented as the sum of C(n-1, k-1) and C(n-1, k) for each k.

(b) To compute C(n, k) using memoization, we can store the intermediate results in a lookup table or cache. Before computing C(n, k), we can check if it already exists in the cache. If it does, we can directly retrieve the precomputed value instead of recomputing it. If it is not present, we calculate the value using the recursive formula and store it in the cache for future use. By utilizing memoization, we significantly reduce the number of computations needed. Each unique computation is performed only once, and subsequent calls can retrieve the precomputed value from the cache in constant time. The time complexity of computing C(n, k) using memoization can be characterized as O(nk). This is because there are at most n * k unique combinations of n and k, and each computation requires a constant number of arithmetic operations.

**15.8.29**

Initialize an empty list, result, to store the valid English words.
Create a dynamic programming table, dp, of size n+1, where n is the length of the input string.
Set dp[0] to True since an empty string is always a valid word.
Iterate through the string from left to right, considering each prefix of the string.
For each prefix ending at index j, check if dp[j] is True (indicating a valid prefix).
If dp[j] is True, check if the suffix from index j to i is a valid word using the valid function.
If the suffix is valid, set dp[i] to True.
After the iteration, if dp[n] is True, it means the entire string is a valid sequence of English words.
To obtain the sequence of valid words, trace back the dynamic programming table from dp[n] to dp[0], storing the words found along the way in result.
Reverse the result list to obtain the correct word order.
The running time of this algorithm depends on the length of the input string, n, and the time complexity of the valid function, assumed to be O(1).

**15.8.34**

Speech recognition systems need to match audio streams that represent the same words spoken at different speeds. Suppose, therefore, that you are given two sequences of numbers, , representing two different audio streams that need to be matched. A mapping between and is a list, of distinct pairs, that is ordered lexicographically, such that, for each there is at least one pair, in and for each there is at least one pair, in Such a mapping is notonic if, for any and in, with coming before in we have and. For example, given one possible monotonic mapping between and would be.
The dynamic time warping problem is to find a monotonic mapping, between and that minimizes the distance, between and subject to which is defined as where this minimization is taken over all possible monotonic mappings between.

**15.8.35**

Create a dynamic programming table, dp, of size n, where n is the number of houses in the row.
Initialize dp[0] to be the value of the first house in the row.
Initialize dp[1] to be the maximum of the first two houses' values.

Iterate through the remaining houses from index 2 to n-1.
For each house at index i, calculate the maximum net value Alice can obtain if she chooses houses up to this point.
To calculate dp[i], Alice has two options: choosing the current house alone or choosing the current house with the maximum net value obtained so far.
Calculate the maximum of the following two values:
The value of the current house plus the maximum net value Alice can obtain by skipping the previous house (dp[i-2]).
The maximum net value obtained so far (dp[i-1]).
Set dp[i] to be the maximum of the two calculated values.
After the iteration, dp[n-1] will hold the maximum net value Alice can achieve by choosing houses up to the last one.
Return dp[n-1] as the maximum net value.
The running time of this algorithm is O(n) because we iterate through the houses once, performing constant-time calculations at each step.

## 15.8.36

To determine the overall probability that the Anteaters will win the World Series, we define a recursive equation:
p(n, m) = P(Anteaters win game n+m) * p(n-1, m) + P(Bears win game n+m) * p(n, m-1)
where n represents the number of games won by the Anteaters and m represents the number of games won by the Bears after playing a total of n+m games.
The special cases are:
p(n, m) = 0 when n = 0 and m = 0 (no games played yet).
p(n, m) = 1 when n = 0 and m > 0 (Bears have already won the required games).
p(n, m) = 0 when n > 0 and m = 0 (Anteaters have already won the required games).
To calculate the overall probability, we need to compute p((N+1)/2, N/2) where N is the total number of games in the series.
Using a recursive algorithm, we compute the values of p(n, m) starting from the base cases and building up the probabilities using the recursive equation.

# Chapter 16:

## 16.6.8

```
      (root)
        |
        a
        |
        b
       / \
      a   b
     /   / \
    b   a   b
```

```
       /    \
      b      a
     /
    a
   /
  c
 /
b
```

**16.6.18**

Preprocess the pattern to construct an auxiliary array, lps (longest proper prefix which is also a suffix) of size n.
Initialize lps[0] to 0.
Set len to 0 (length of the previous longest prefix suffix).
Iterate through the pattern from index 1 to n-1:
If the current character at index i matches the character at index len in the pattern:
Increment len by 1.
Set lps[i] to len.
If the characters do not match:
If len is not 0, update len using lps[len-1].
If len is 0, set lps[i] to 0.
After the iteration, lps will contain the longest proper prefix suffix values for each index in the pattern.

Perform the pattern matching using the KMP algorithm.
Initialize two pointers, i for the text and j for the pattern, both starting at 0.
Iterate until i is less than the length of the text and j is less than the length of the pattern:
If the characters at indices i and j match:
Increment both i and j by 1.
If the characters do not match:
If j is not 0, update j using lps[j-1].
If j is 0, increment i by 1.
After the iteration, if j is equal to the length of the pattern, it means the entire pattern is a substring of the text.
The longest prefix of the pattern that is a substring of the text is from index 0 to index j-1.

**16.6.28**

Initialize a Bloom filter with an array of size m and k hash functions.
The array is typically initialized with all bits set to 0.
The number of hash functions, k, should be determined based on the desired false positive rate and the expected number of elements to be stored.

The size of the array, m, should be chosen to accommodate the expected number of elements and desired false positive rate.
When the web crawler encounters a new web page:
Compute the hash values of the web page using the k hash functions.
Set the corresponding bits in the Bloom filter array to 1.
To check whether a web page has been encountered before:
Compute the hash values of the web page using the same k hash functions.
Check if all the corresponding bits in the Bloom filter array are set to 1.
If any of the bits are not set to 1, the web page has definitely not been encountered before.
If all the bits are set to 1, there is a possibility of a false positive, indicating that the web page may have been encountered before. Further verification may be required.
The time complexity of checking whether a web page has been encountered before is O(k), as it involves computing the hash values and checking the corresponding bits in the Bloom filter array. This operation takes constant time, regardless of the length of the web page.

# Chapter 19:

## 19.7.2

Initialize a Bloom filter with an array of size m and k hash functions.
The array is typically initialized with all bits set to 0. The number of hash functions, k, should be determined based on the desired false positive rate and the expected number of elements to be stored. The size of the array, m, should be chosen to accommodate the expected number of elements and desired false positive rate. When the web crawler encounters a new web page: Compute the hash values of the web page using the k hash functions. Set the corresponding bits in the Bloom filter array to 1. To check whether a web page has been encountered before: Compute the hash values of the web page using the same k hash functions. Check if all the corresponding bits in the Bloom filter array are set to 1. If any of the bits are not set to 1, the web page has definitely not been encountered before. If all the bits are set to 1, there is a possibility of a false positive, indicating that the web page may have been encountered before. Further verification may be required. The time complexity of checking whether a web page has been encountered before is O(k), as it involves computing the hash values and checking the corresponding bits in the Bloom filter array. This operation takes constant time, regardless of the length of the web page.

## 19.7.6

Assuming capacities are labeled on each edge, let's say the capacities are as follows:

(S, A): 4
(S, B): 3
(A, C): 2
(A, D): 2
(B, C): 1

(B, D): 5
(C, E): 3
(C, F): 2
(D, E): 4
(D, F): 2
(E, T): 3
(F, T): 3

To find a minimum cut, we can identify a set of vertices that, when removed, disconnects the source (S) from the sink (T) in the flow network. One such minimum cut in this case could be the set {A, C, D}.

## 19.7.28

Initialize an empty list, matched_puppies, to store the matched pairs of residents and puppies. Sort the list of residents in descending order based on their preferences. For each resident, sort their preferences of puppies in descending order.
Iterate through the sorted list of residents:
For each resident, iterate through their preferences of puppies. Check if the current puppy is available (i.e., not already assigned to another resident) and if the resident has not reached their adoption limit of one puppy. If the puppy is available and the resident has not reached their limit, assign the puppy to the resident and add the matched pair to matched_puppies.
Update the availability and adoption limit of the puppy and the resident, respectively.
Return matched_puppies containing the pairs of residents and puppies. The time complexity of this algorithm depends on the number of residents and puppies. If there are n residents and m puppies, the worst-case time complexity is O(n^2 * m). However, in practice, the algorithm often runs in much less time due to early termination when residents reach their adoption limit or when all puppies are assigned.

## 19.7.32

Create a flow network from the directed acyclic graph:

Introduce a source node, S, and connect it to the starting node, A, with an edge of infinite capacity. Introduce a sink node, T, and connect it to the target node, B, with an edge of infinite capacity. Assign a capacity of 1 to each edge in the graph, representing the fact that only one of the wolf or the goat can use an edge. The capacities of the remaining edges in the graph are set to infinity, indicating that both the wolf and the goat can use those edges. If there are any additional constraints or edge weights, incorporate them into the flow network. Run the Ford-Fulkerson algorithm to find the maximum flow in the network from S to T.

Check if the maximum flow is equal to 2. If it is, there exist two edge-disjoint paths for the goat and the wolf to travel from A to B, ensuring the safety of the goat. If the maximum flow is not equal to 2, there are no two edge-disjoint paths that satisfy the safety requirement. The time complexity of the Ford-Fulkerson algorithm is determined by the number of iterations required to reach the maximum flow.

## 19.7.33

Create a flow network from the directed acyclic graph:
Introduce a source node, S, and connect it to the starting node, A, with an edge of infinite capacity. Introduce a sink node, T, and connect it to the target node, B, with an edge of infinite capacity. Assign a capacity of 1 to each edge in the graph, representing the fact that only one of the wolf or the goat can use an edge. The capacities of the remaining edges in the graph are set to infinity, indicating that both the wolf and the goat can use those edges. If there are any additional constraints or edge weights, incorporate them into the flow network. Run the Ford-Fulkerson algorithm to find the maximum flow in the network from S to T.
Check if the maximum flow is equal to 2. If it is, there exist two edge-disjoint paths for the goat and the wolf to travel from A to B, ensuring the safety of the goat.
If the maximum flow is not equal to 2, there are no two edge-disjoint paths that satisfy the safety requirement. The time complexity of the Ford-Fulkerson algorithm is determined by the number of iterations required to reach the maximum flow. In the worst-case scenario, it has a time complexity of O(Ef), where E is the number of edges in the graph and f is the maximum flow value.

## 19.7.34

Create a cost matrix:

Initialize a square matrix of size n x n, where n is the number of pickup locations and limos.
Assign the cost of each limo-location pair to the corresponding cell in the matrix. The cost represents the distance between the limo and the pickup location.
Apply the Hungarian Algorithm to find the optimal assignment:
Subtract the smallest element from each row and each column of the cost matrix. This step ensures that there are no negative values in the matrix. Find the minimum number of lines (either rows or columns) required to cover all zeros in the matrix. If the number of lines equals n, move to the next step. Otherwise, adjust the matrix and lines to find additional lines until n lines are covered. If n lines are not covered, update the cost matrix and lines to create new opportunities for covering lines. Repeat the previous step until n lines are covered.
Assign the limos to pickup locations based on the covered lines, ensuring that each limo is assigned to only one pickup location and vice versa.
Calculate the total distance traveled:
Sum up the distances in the cost matrix for the assigned limo-location pairs. This total distance represents the minimum distance traveled by all limos.
The time complexity of the Hungarian Algorithm is O(n^3), where n is the number of pickup locations and limos. This makes it an efficient algorithm for solving the limousine dispatchment problem and minimizing the total distance traveled.

# Chapter 20:

## 20.8.1

No, Professor Amongus has not proven that P=NP based on the information provided.
The fact that problem A is polynomial-time reducible to an NP-complete problem B means that if we have a polynomial-time algorithm to solve B, we can use it to solve A in polynomial time. However, it does not necessarily imply that A itself can be solved in polynomial time. In this case, Professor Amongus has shown that problem A can be solved in polynomial time. However, this does not automatically prove that P=NP. It only indicates that problem A is in the complexity class P, which consists of decision problems that can be solved in polynomial time.

## 20.8.3

SAT is in the complexity class NP, meaning that given a potential solution, we can verify its correctness in polynomial time.
Given a potential assignment of truth values to the variables in a Boolean formula, we can substitute these values into the formula and evaluate it. This evaluation can be done in polynomial time. If the formula evaluates to true, we accept the solution as a satisfying assignment. Otherwise, we reject it. Therefore, SAT is in NP. SAT is NP-hard, meaning that any problem in the class NP can be reduced to SAT in polynomial time. To demonstrate this, we perform a reduction from the well-known NP-complete problem, 3-SAT, to SAT. We transform an instance of 3-SAT into an equivalent instance of SAT. This reduction can be done in polynomial time. By showing that SAT is both in NP and NP-hard, we establish its NP-completeness. This means that SAT is one of the hardest problems in the class NP, and it serves as a benchmark for the difficulty of other problems in NP.

## 20.8.9

To construct the instance of VERTEX-COVER from the given 3SAT formula, we follow the reduction from 3SAT to VERTEX-COVER. Here's an example of the construction:
Given a 3SAT formula with variables x1, x2, x3, and clauses (x1 ∨ x2 ∨ ¬x3), (¬x1 ∨ x2 ∨ x3), and (x1 ∨ ¬x2 ∨ x3), the reduction proceeds as follows:
Create a vertex for each literal and its negation:
Create vertices for x1, x2, x3, ¬x1, ¬x2, and ¬x3.
Create a vertex for each clause:
Create vertices C1, C2, and C3.
Connect vertices representing literals to the corresponding clause vertices:

Connect x1 to C1, x2 to C1, and ¬x3 to C1.
Connect ¬x1 to C2, x2 to C2, and x3 to C2.
Connect x1 to C3, ¬x2 to C3, and x3 to C3.

**20.8.24**
First, let's define the implication graph:
Create a vertex for each variable and its negation in the 2SAT instance.
For each clause (a ∨ b), add two directed edges: (¬a -> b) and (¬b -> a) in the implication graph.
Now, we can state the rule:
If there exists a variable x such that there is a path from ¬x to x and a path from x to ¬x in the implication graph, then the 2SAT instance is not satisfiable. This is because if both x and ¬x are assigned true, it creates a contradiction. Conversely, if there is no variable x that has such a path, then the 2SAT instance is satisfiable. This rule holds because in a satisfiable 2SAT instance, the implication graph must not contain any SCC where a variable and its negation are both present. If such an SCC exists, it implies a contradiction in the assignment of truth values.
To solve the 2SAT problem using this rule, we can use the Kosaraju's algorithm to find the SCCs in the implication graph. If we find an SCC that contains a variable and its negation, the 2SAT instance is not satisfiable. Otherwise, it is satisfiable.

**20.8.35**
To show that the decision version of Dr. Drama's problem is NP-complete, we need to demonstrate two things:
The problem is in the complexity class NP, meaning that given a potential solution, we can verify its correctness in polynomial time. Given a set of invited companies, we can verify if it satisfies the condition of no pair of competing companies being invited by checking the list of competitors for each invited company. If there is no pair of competing companies among the invited set, we accept the solution. This verification can be done in polynomial time, so the problem is in NP.
The problem is NP-hard, meaning that any problem in the class NP can be reduced to this problem in polynomial time. We can reduce the Maximum Independent Set (MIS) problem, a known NP-complete problem, to Dr. Drama's problem. Given an instance of MIS, we construct an instance of Dr. Drama's problem by mapping each vertex in the MIS graph to a company that could be invited. For each edge in the MIS graph, we add an entry to the competitor list of both corresponding companies in Dr. Drama's problem. The maximum number of noncompeting companies is set to be the same as the maximum independent set size.

**20.8.39**
To show that the decision version of the problem is NP-complete, we need to demonstrate two things:

The problem is in the complexity class NP, meaning that given a potential solution, we can verify its correctness in polynomial time. Given a potential collection of websites, we can verify if it satisfies the condition of being visited by all infected computers by checking each log file and confirming that every infected computer has visited all the websites in the collection. This verification can be done in polynomial time, so the problem is in NP. The problem is NP-hard, meaning that any problem in the class NP can be reduced to this problem in polynomial time. We can perform a reduction from the well-known NP-complete problem, Set Cover, to the problem of determining the smallest collection of websites visited by all infected computers. In the Set Cover problem, we are given a universe of elements and a collection of subsets of the universe. We aim to find the smallest sub-collection of subsets that covers all the elements. By mapping infected computers to elements and websites visited by each computer to subsets, we can reduce Set Cover to the problem of determining the smallest collection of websites visited by all infected computers.

**20.8.41**

To show that determining whether a division of books is possible, such that the total used-book value is equal between two individuals, is NP-complete, we need to demonstrate two things: The problem is in the complexity class NP, meaning that given a potential division of books, we can verify its correctness in polynomial time. Given a potential division of books, we can calculate the total used-book value for each individual and check if they are equal. This verification can be done by summing the values of the books in each person's set, which can be done in polynomial time. Thus, the problem is in NP. The problem is NP-hard, meaning that any problem in the class NP can be reduced to this problem in polynomial time. We can perform a reduction from the well-known NP-complete problem, Subset Sum, to the problem of dividing books. In the Subset Sum problem, we are given a set of integers and a target sum, and we need to determine if there is a subset of the integers that adds up to the target sum. To perform the reduction, we can map the integers in the Subset Sum problem to the values of the books and set the target sum to be half of the total used-book value. We aim to determine if there is a division of books between the two individuals such that the total used-book value of each set is equal to the target sum.

# Chapter 21:

**21.10.2**

In the Subset Sum problem, we are given a set of integers and a target sum, and we need to determine if there is a subset of the integers that adds up to the target sum. To perform the reduction, we can map the integers in the Subset Sum problem to the values of the books and set the target sum to be half of the total used-book value. We aim to determine if there is a division of books between the two individuals such that the total used-book value of each set is equal to the target sum. By performing this reduction, we have shown that Subset Sum can be reduced to the problem of dividing books. Since Subset Sum is known to be NP-complete, and we have shown that the book division problem is in NP and can be reduced to an NP-complete problem, we conclude that determining whether a division of books is possible with equal total used-book value is NP-complete.

**21.10.18**

Let's consider an array of size 2: [A, B]. The original Fisher-Yates algorithm would randomly select one of the two elements, say A, and swap it with itself, resulting in the same array [A, B]. The algorithm preserves the original order. In the modified algorithm, when we choose a random index for swapping, we can either select the same index or the other index. Let's analyze both cases:
Swapping with the same index: If we choose the same index, say index 0, we swap A with itself, resulting in [A, B]. This case preserves the original order, just like the original algorithm.
Swapping with the other index: If we choose the other index, say index 1, we swap A and B, resulting in [B, A]. This case changes the order of the elements. In this case, the modified algorithm generates two possible permutations: [A, B] and [B, A], but with different probabilities. Swapping with the same index occurs with a probability of 1/2, and swapping with the other index also occurs with a probability of 1/2.

**21.10.38**
(a) The probability that the th employee has a distinct PIN given that the previous PINs are distinct can be calculated as follows:
The first employee can be assigned any of the 100 million PINs, so the probability of having a distinct PIN is 1. For the second employee, there are 99,999,999 remaining possible PINs. The probability of choosing a distinct PIN for the second employee is therefore (99,999,999 / 100,000,000). Similarly, for the th employee, there are (100,000,000 - t + 1) remaining possible PINs. So the probability of choosing a distinct PIN for the th employee is ((100,000,000 - t + 1) / 100,000,000). Therefore, the probability that the th employee has a distinct PIN is the product of these probabilities for each employee: P(distinct PIN for the th employee) = 1 * (99,999,999 / 100,000,000) * ((100,000,000 - t + 1) / 100,000,000).
(b) The probability that all the PINs are distinct is the product of the probabilities for each employee having a distinct PIN. Using the fact that (1 - x) <= e^(-x) for any positive value x, we can write: (all PINs distinct) = (1 * (99,999,999 / 100,000,000) * ((100,000,000 - 1 + 1) / 100,000,000)) * (1 * (99,999,999 / 100,000,000) * ((100,000,000 - 2 + 1) / 100,000,000)) * ... * (1 * (99,999,999 / 100,000,000) * ((100,000,000 - (t-1) + 1) / 100,000,000))
= (1 - 1/100,000,000) * (1 - 2/100,000,000) * ... * (1 - (t-1)/100,000,000)
<= e^(-(1/100,000,000 + 2/100,000,000 + ... + (t-1)/100,000,000))
= e^(-(t(t-1)/(2 * 100,000,000)))

(c) We want to find the number of PINs, denoted as t, such that the probability of two employees having the same PIN is at least p. From part (b), we have the bound:
P(all PINs distinct) <= e^(-(t(t-1)/(2 * 100,000,000)))
We want this probability to be less than or equal to 1 - p:
e^(-(t(t-1)/(2 * 100,000,000))) <= 1 - p
Taking the natural logarithm on both sides, we get:
-(t(t-1)/(2 * 100,000,000)) <= ln(1 - p)
Multiplying both sides by -1, we have:
(t(t-1)/(2 * 100,000,000)) >= -ln(1 - p)
Simplifying, we get:

t(t-1) >= 2 * 100,000,000 * ln(1 - p)

# Chapter 23:

## 23.6.28

Lower bound on the greedy tour cost, G:
The greedy algorithm adds the nearest city to the current tour at each step. The cost of the tour G can be expressed as the sum of distances between consecutive cities in the tour. Upper bound on the optimal tour cost, OPT:
The optimal tour OPT is the shortest possible tour that visits all cities. Therefore, the cost of OPT must be greater than or equal to the sum of distances between consecutive cities in any valid tour. By comparing the lower bound on G and the upper bound on OPT, we establish the relationship G ≤ OPT.
Relationship between G and OPT:
Since the greedy algorithm always selects the nearest city, the distance between any two consecutive cities in G is at most half the distance between the corresponding cities in OPT. Summing up these inequalities, we get G ≤ (1/2) * OPT. Therefore, the cost of the tour produced by the nearest-neighbor greedy algorithm, G, is at most half the cost of the optimal tour, OPT:
G ≤ (1/2) * OPT

# Chapter 24:

## 24.6.4

In a linear program with a parameter "k" in the objective function, the values of "k" that result in a program with no unique solution depend on the specific structure and coefficients of the objective function and constraints. Typically, a linear program seeks to optimize an objective function (maximize or minimize) while satisfying a set of constraints. The objective function is a linear combination of decision variables, where the coefficients may be influenced by the parameter "k." When "k" takes certain values, it can introduce degeneracies or redundancies in the problem formulation, leading to multiple optimal solutions or infeasibility. For example, if the objective function has coefficients that are multiples of "k" or involve "k" in a way that cancels out certain terms, the linear program may have an infinite number of optimal solutions. This occurs when changing the value of "k" does not affect the optimal solution or alters it in a predictable manner. Conversely, if the objective function coefficients or constraints have conflicting relationships with "k," it may render the linear program infeasible, meaning there is no solution that satisfies all constraints simultaneously. To determine the values of "k" resulting in no unique solution, a careful examination of the objective function and constraints is necessary. Analyzing the specific coefficients and their dependence on "k" will help identify cases where the linear program exhibits multiple optimal solutions or becomes infeasible.

**24.6.13**
Primal:
Maximize: $3x_1 + 2x_2$
Subject to:
$x_1 + 2x_2 \leq 5$
$2x_1 + x_2 \leq 4$
$x_1, x_2 \geq 0$

The dual of this primal program would be:
Dual:
Minimize: $5y_1 + 4y_2$
Subject to:
$y_1 + 2y_2 \geq 3$
$2y_1 + y_2 \geq 2$
$y_1, y_2 \geq 0$

In the dual program, "$y_1$" and "$y_2$" are the dual variables corresponding to the constraints of the primal program, and the coefficients "c" and "b" have been transposed to form the dual program's objective function and constraint equations.

**24.6.35**

Let:
$x_1$ = number of 18L jugs of water
$x_2$ = number of tents
$x_3$ = number of packs of first aid supplies
$x_4$ = number of cases of food rations
$x_5$ = number of packs of personal supplies
We want to minimize the number of camels needed, so our objective function is:
Minimize: $x_6$
Subject to the following constraints:
Water constraint:
$x_1 \geq$ (2L/person/day) * 15 persons * (10 days + 0.5 * 10 days) = 300L

Tent constraint:
$x_2 \geq$ ceil(15 persons / 2 persons per tent) = 8 (using the ceiling function)

First aid constraint:
$x_3 \geq$ ceil((15 persons * 10 days) / 10 persons) = 15 (using the ceiling function)

Food ration constraint:
$x_4 \geq$ (1 ration/person/day) * 15 persons * (10 days + 0.5 * 10 days) / 12 rations per case

Personal supplies constraint:
$x_5 \geq 15$ persons

Camel capacity constraint:
$x_1$ * 18L + $x_2$ * weight of tent + $x_3$ * weight of first aid pack + $x_4$ * weight of ration case + $x_5$ * weight of personal supply pack ≤ $x_6$ * camel capacity

Trade goods constraint:
5 * weight of spice chest + 5 * weight of tea chest + 5 * weight of silk chest ≤ $x_6$ * camel capacity

All variables must be non-negative:
$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$

This linear program formulates the problem of determining the minimum number of camels required to transport all the necessary supplies for the journey across the desert. By solving this linear program, you can find the optimal solution that minimizes the number of camels needed for the expedition.