**Question 1:**

**1. Prove that the optimal substructure property exists:**

Let's assume that there is an optimal solution to the problem that does not exhibit the optimal substructure property. This means there exists a partitioning of the string such that it can be further optimized by splitting one of the partitions. However, this contradicts the assumption that the initial partitioning was optimal. Hence, the optimal substructure property holds for palindrome partitioning.

**2. What will be the smaller sub-problems?**

The smaller sub-problems can be defined as finding the minimum number of partitions required for substrings of the given string. For example, if the given string is "aab", the smaller sub-problems would be finding the minimum number of partitions for substrings "a", "aa", and "aab".

**3. Recursive definition/formula:**

Let's define a function `minPalPartition(s)` that returns the minimum number of partitions required for the string `s`.

- Base case: If the string `s` is empty or a palindrome itself, no partitions are required. So, `minPalPartition(s) = 0`.

- Recursive case: For each possible partition point `i` in the string, we can split the string into two parts: `s[0:i]` and `s[i+1:]`. The minimum number of partitions required for the string `s` can be calculated as follows:

**`minPalPartition(s) = min(1 + minPalPartition(s[0:i]) + minPalPartition(s[i+1:]))**`, for all valid `i` values.

**4. Dynamic programming algorithm for the problem that gives the optimal value as well as the optimal solution:**

function minPalPartition(s):

n = length(s)

dp = array of size (n + 1) initialized with 0

cuts = array of n + 1 empty arrays

for i = 1 to n:

dp[i] = i

for j = 0 to i:

if isPalindrome(s[j:i]):

if 1 + dp[j] < dp[i]:

dp[i] = 1 + dp[j]

cuts[i] = cuts[j] + [s[j:i]]

return dp[n],cuts[n]

The `isPalindrome(s[i:j])` function can be implemented to check if the substring `s[i:j]` is a palindrome.

**5. Space and time complexity of the solution:**

**- Time complexity:** The nested loops iterate over all possible substrings, resulting in a time complexity of **O(n^2)**, where n is the length of the input string or **O(n^3)** if isPalindrome is also considered.

**- Space complexity**: The DP table `dp` of size (n+1) is used to store intermediate results, resulting in a space complexity of **O(n)**.

**Question 2:**

**1. Optimal substructure property:**

To prove the optimal substructure property for this problem, we can assume that we have an optimal solution to the problem of bursting balloons from the ith to the jth index. Let k be the index of the last balloon to burst in this optimal solution. We can then assume that the optimal solution for the subproblems of bursting balloons from the ith to the k-1th index and from the k+1th to the jth index is also included in this optimal solution. We can prove this by contradiction: assume that one of these subproblems has a better solution than the one included in the optimal solution for the larger problem. We can then replace this sub problem's solution with the better one, and the resulting solution for the larger problem would also be better, which contradicts our assumption that the original solution was optimal. Therefore, the optimal solution for the larger problem must include the optimal solutions for the subproblems.

**2. Smaller sub-problems:**

The smaller sub-problems would be to find the optimal order to burst balloons between two indices i and j, where i and j are arbitrary indices within the given list of balloons.

**3. Recursive definition/formula:**

We can define a recursive formula to solve this problem as follows:

**-** Let dp[i][j] be the maximum number of coins that can be obtained by bursting balloons from the ith to the jth index (inclusive).

**-** We can then find the maximum number of coins that can be obtained by bursting balloons from the ith to the jth index (inclusive) by trying all possible last balloons to burst in this range. For each k in the range i ≤ k ≤ j, we can calculate the maximum number of coins that can be obtained by bursting balloons from the ith to the k-1th index and from the k+1th to the jth index (inclusive), and adding the coins obtained from bursting balloon k. The formula for this is:

**dp[i][j] = max(dp[i][j], dp[i][k-1] + dp[k+1][j] + nums[k-1] \* nums[k] \* nums[k+1])**

**4. Dynamic programming algorithm:**

maxCoins(nums):

n = length of nums

nums = [1] + nums + [1]

dp = create a matrix of size (n+2) x (n+2) initialized with zeros

burst\_order = create a matrix of size (n+2) x (n+2) initialized with zeros

for len\_range = 1 to n+1:

for i = 1 to n - len\_range + 2:

j = i + len\_range - 1

for k = i to j:

coins = dp[i][k-1] + dp[k+1][j] + nums[k-1] \* nums[k] \* nums[k+1]

if coins > dp[i][j]:

dp[i][j] = coins

burst\_order[i][j] = k

return dp[1][n], getBurstOrder(burst\_order, 1, n, [])

getBurstOrder(burst\_order, i, j, order):

if i > j:

return order

k = burst\_order[i][j]

order.append(k)

order = getBurstOrder(burst\_order, i, k-1, order)

order = getBurstOrder(burst\_order, k+1, j, order)

return order

**5. Space and time complexity:**

The time complexity of this algorithm is **O(n^3)**, where n is the length of the `nums` list, due to the three nested loops. The space complexity is also **O(n^2)**, because we need to store the values of `dp[i][j]` for each i and j.  
 **Question 3:**  
  
**1. Optimal Substructure Property:**

To prove the optimal substructure property for the Egg Dropping Problem, let's assume that there exists a solution that is not optimal, contradicting the assumption that the problem possesses the optimal substructure.

Suppose we have an optimal solution for a given number of 'eggs' and 'floors' that does not adhere to the optimal substructure property. This means that there exists an optimal solution where the chosen attempt, such as dropping an egg from a particular floor 'i', does not lead to the overall optimal solution.

Let's assume that 'k' is the floor where we performed the suboptimal attempt in the optimal solution. By dropping the egg from floor 'k', we have two possible outcomes:

* The egg breaks:

In this case, we would have one less egg remaining, and the problem reduces to finding the critical floor with 'eggs - 1' eggs on the floors below 'k'.

* The egg does not break:

In this case, we can eliminate the floors below 'k' since the egg did not break, and the problem reduces to finding the critical floor with 'eggs' eggs on the remaining floors above 'k'.

Now, if the chosen attempt from floor 'k' in the suboptimal solution did not lead to the optimal solution, it implies that there exists another floor 'j' (1 ≤ j ≤ floors) that would have yielded a better or equal optimal solution. This implies that either the subproblem with 'eggs - 1' eggs and 'j - 1' floors or the subproblem with 'eggs' eggs and 'floors - j' floors would have resulted in a smaller or equal number of attempts.

However, this contradicts our assumption that the initial solution was optimal. If there were a better or equal solution when choosing a different floor 'j' instead of 'k', the original solution would not have been optimal.

Since we have reached a contradiction, our assumption that there exists a solution violating the optimal substructure property must be false.

**2. Smaller Sub-Problems:**

The smaller sub-problems involve finding the minimum number of attempts needed to determine the critical floor for a given number of eggs and a smaller number of floors. This allows us to break down the original problem into simpler subproblems.

**3. Recursive Definition/Formula:**

Let's define the recursive formula for the Egg Dropping Problem:

* `dp(eggs, floors)`: Represents the minimum number of attempts needed to determine the critical floor for a given number of `eggs` and `floors`.
* We consider the worst-case scenario for each attempt. For a given attempt, we drop an egg from a specific floor `i`, where `i` can range from 1 to `floors`.
* If the egg breaks, we need to recursively solve the subproblem of determining the critical floor with one less egg on the floors below `i`. This can be represented as `1 + dp(eggs - 1, i - 1)`.
* If the egg does not break, we need to recursively solve the subproblem of determining the critical floor with the same number of eggs on the floors above `i`. This can be represented as `1 + dp(eggs, floors - i)`.
* We take the maximum of these two cases since we want to consider the worst-case scenario. Therefore, the recursive formula becomes:

**`dp(eggs, floors) = min(max(1 + dp(eggs - 1, i - 1), 1 + dp(eggs, floors - i)))`**

for all `i` from 1 to `floors`.

**4. Dynamic Programming Algorithm:**

eggDrop(eggs, floors):

dp = create a matrix of size (eggs + 1) x (floors + 1) and initialize all elements to 0

floor = create a matrix of size (eggs + 1) x (floors + 1) and initialize all elements to 0

for i = 1 to eggs:

dp[i][0] = 0

dp[i][1] = 1

for j = 1 to floors:

dp[1][j] = j

for i = 2 to eggs:

for j = 2 to floors:

dp[i][j] = infinity

for k = 1 to j:

res = 1 + max(dp[i-1][k-1], dp[i][j-k])

if res < dp[i][j]:

dp[i][j] = res

floor[i][j] = k

return dp[eggs][floors], getCriticalFloors(eggs, floors, floor, dp)

getCriticalFloors(eggs, floors, floor, dp):

if eggs = 1:

return [floors]

else:

criticalFloors = []

k = floor[eggs][floors]

criticalFloors.append(k)

if dp[eggs-1][k-1] < dp[eggs][floors-k]:

criticalFloors += getCriticalFloors(eggs - 1, k - 1, floor, dp)

else:

criticalFloors += getCriticalFloors(eggs, floors - k, floor, dp)

return criticalFloors

**5. Space and time complexity:**

The time complexity of the Egg Dropping Problem algorithm is **O(eggs \* floors^2)** since we have nested loops iterating over `eggs` and `floors`, and for each iteration, we perform another loop from 1 to `j`. The space complexity is **O(eggs \* floors)** since we utilize the `dp` and `floor` matrices, each with dimensions (eggs + 1) x (floors + 1), to store the intermediate results and critical floor information.

**Question 4:**

**1. Optimal Substructure Property:**

To prove the optimal substructure property for the contiguous subsequence of the maximum sum problem, we assume the existence of a suboptimal solution that contradicts the optimal substructure.

Suppose there exists a suboptimal solution for a given array of integers where the selected contiguous subsequence does not contribute to the overall maximum sum. This implies that there must exist another contiguous subsequence within the array that yields a larger sum.

Let's assume our suboptimal solution includes a subsequence starting at index 'i' and ending at index 'j', where i ≤ j. Since this subsequence is considered suboptimal, there must exist another subsequence starting at index 'k' and ending at index 'l' (k ≤ l) such that the sum of the subsequence from 'k' to 'l' is greater than the sum of the subsequence from 'i' to 'j'.

Without loss of generality, we can assume that 'k' is the smallest index such that the subsequence from 'k' to 'l' has a larger sum. Since the subsequence from 'i' to 'j' is suboptimal, the sum of the subsequence from 'i' to 'j' must be less than or equal to the sum of the subsequence from 'k' to 'l'.

Now, consider a new subsequence starting from index 'k' and ending at index 'j'. This subsequence includes the subsequence from 'i' to 'j' but excludes the elements from 'k' to 'l'. Since the sum of the subsequence from 'k' to 'l' is greater than or equal to the sum of the subsequence from 'i' to 'j', the new subsequence starting from 'k' and ending at 'j' has a larger sum than the subsequence from 'i' to 'j'.

This contradicts our assumption that the subsequence from 'i' to 'j' is suboptimal because we have found a larger sum by considering a different subsequence. Hence, our initial assumption of a suboptimal solution is incorrect, and the problem exhibits the optimal substructure property.

**2. Smaller Sub-problems:**

The smaller subproblems involve finding the maximum sum of subsequences ending at each index of the array.

**3. Recursive Definition/Formula:**

Let's define a function maxSubsequenceSum(arr, i) that represents the maximum sum of a subsequence ending at index 'i' of the array 'arr'. The recursive formula is as follows:

**Base case:** maxSubsequenceSum(arr, 0) is simply arr[0] since there is only one element.

**maxSubsequenceSum(arr, 0) = arr[0]**

**Recursive case:** maxSubsequenceSum(arr, i) is the maximum between arr[i] and arr[i] + maxSubsequenceSum(arr, i-1).

**maxSubsequenceSum(arr, i) = max(arr[i], arr[i] + maxSubsequenceSum(arr, i-1))**

**4. Dynamic Programming Algorithm:**

function maxSubsequenceSum(arr):

n = length of arr

maxSum = arr[0]

currentSum = arr[0]

start = 0

end = 0

tempStart = 0

for i = 1 to n-1:

if currentSum < 0:

currentSum = arr[i]

tempStart = i

else:

currentSum = currentSum + arr[i]

if currentSum > maxSum:

maxSum = currentSum

start = tempStart

end = i

return maxSum, subsequence from index 'start' to 'end'

**5. Space and time complexity:**

The time complexity of the algorithm is **O(n)**, where 'n' is the size of the input array, and the space complexity is **O(1)** as we only use a few variables to store the maximum sum and indices.

**Question 5:**

**1. Proof of Optimal Substructure:**

To prove the optimal substructure property for the minimum coin change problem, we assume the existence of a suboptimal solution that contradicts the optimal substructure.

Suppose there exists a suboptimal solution for a given amount of change, where the selected set of coins does not yield the minimum number of coins needed to make the change. This implies that there must exist another set of coins that can achieve the minimum number of coins for the given amount of change.

Let's assume our suboptimal solution includes a set of coins C1. Since this solution is considered suboptimal, there must exist another set of coins C2 that results in a smaller number of coins to make the change.

Without loss of generality, we can assume that C2 is the smallest set of coins that achieves the minimum number of coins for the given amount of change. Since C1 is suboptimal, the number of coins in C1 must be greater than the number of coins in C2.

Now, consider a new set of coins C3 that includes the coins from C2 and excludes the coins from C1. Since the number of coins in C1 is greater than the number of coins in C2, the number of coins in C3 is also less than the number of coins in C1.

This contradicts our assumption that C1 is suboptimal because we have found a set of coins (C3) that achieves a smaller number of coins to make the change. Hence, our initial assumption of a suboptimal solution is incorrect, and the problem exhibits the optimal substructure property.

**2. Smaller Subproblems:**

The smaller subproblems for the minimum coin change problem involve finding the minimum number of coins needed to make smaller amounts of change. We break down the original problem of making the target amount into subproblems of making smaller amounts, reducing the problem size at each step.

**3. Recursive Definition/Formula:**

Let's define a function minCoinChange(coins, amount) that represents the minimum number of coins needed to make the given 'amount' of change using the provided 'coins' denominations. The recursive formula is as follows:

**Base case**: If the 'amount' is 0, then the minimum number of coins needed is 0. We return 0 as the base case.

**minCoinChange(coins, amount) = 0** if amount = 0

**Recursive case:** For each coin denomination 'coin' in 'coins', we consider two scenarios:

* If 'coin' is greater than the 'amount', we skip it.

**minCoinChange(coins, amount) = infinity** if amount < 0

* If 'coin' is less than or equal to the 'amount', we recursively calculate the minimum number of coins needed by subtracting 'coin' from the 'amount' and adding 1 (since we use one coin) to the result.

**minCoinChange(coins, amount) = min { minCoinChange(coins, amount - coin) + 1 }**

for coin in coins

**4. Dynamic Programming Algorithm:**

function minCoinChange(coins, amount):

n = length of coins

dp = array of size amount+1 initialized with infinity

dp[0] = 0

usedCoins = array of size amount+1 initialized with empty lists

for i = 1 to amount:

for j = 0 to n-1:

if coins[j] <= i:

if 1 + dp[i - coins[j]] < dp[i]:

dp[i] = 1 + dp[i - coins[j]]

usedCoins[i] = copy of usedCoins[i - coins[j]]

usedCoins[i].append(coins[j])

return dp[amount], usedCoins[amount]

**5. Space and time complexity:**

The time complexity of the algorithm is **O(amount \* n)**, where 'amount' is the target amount and 'n' is the number of coin denominations. The space complexity is **O(amount)** as we use an array of size 'amount+1' to store the minimum number of coins.

**Question 6:**

**1. Proof of Optimal Substructure:**

To prove the optimal substructure property for the minimum number of mismatched characters problem, we assume the existence of a suboptimal solution that contradicts the optimal substructure.

Suppose there exists a suboptimal solution for comparing two strings, where the selected characters to be compared do not yield the minimum number of mismatched characters. This implies that there must exist another selection of characters that results in a smaller number of mismatched characters.

Let's assume our suboptimal solution includes a set of character pairs C1. Since this solution is considered suboptimal, there must exist another set of character pairs C2 that yields a smaller number of mismatched characters.

Without loss of generality, we can assume that C2 is the smallest set of character pairs that achieves the minimum number of mismatched characters. Since C1 is suboptimal, the number of mismatched characters in C1 must be greater than the number of mismatched characters in C2.

Now, consider a new set of character pairs C3 that includes the character pairs from C2 and excludes the character pairs from C1. Since the number of mismatched characters in C1 is greater than the number of mismatched characters in C2, the number of mismatched characters in C3 is also less than the number of mismatched characters in C1.

This contradicts our assumption that C1 is suboptimal because we have found a set of character pairs (C3) that achieves a smaller number of mismatched characters. Hence, our initial assumption of a suboptimal solution is incorrect, and the problem exhibits the optimal substructure property.

**2. Smaller Subproblems:**

The smaller subproblem in the minimum number of mismatched characters problem is considering the alignment of the prefixes of the two strings. By aligning the prefixes, we can compare the current characters and increment the mismatch count accordingly.

**3. Recursive Definition/Formula:**

Let's define C(i, j) as the minimum number of mismatched characters for aligning the prefixes of S1 up to index i and S2 up to index j. The recursive formula is as follows:

**C(i, j) =**

**if (S1[i-1] == S2[j-1]):**

**C(i-1, j-1) // Characters match, no mismatch**

**else:**

**1 + min(C(i-1, j-1), C(i-1, j), C(i, j-1))**

**4. Dynamic Programming Algorithm:**

void getMinimumAlignment(string S1, string S2) {

int m = S1.length();

int n = S2.length();

int C[n + 1][m + 1] = {0}; // table for storing optimal substructure answers

for (int i = 0; i <= m; i++) {

C[0][i] = i;

}

for (int j = 1; j <= n; j++) {

C[j][0] = j;

}

// calculating the minimum penalty

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= m; j++) {

if (S1[j - 1] == S2[i - 1]) {

C[i][j] = C[i - 1][j - 1];

} else {

C[i][j] = min(C[i - 1][j - 1] + 1, min(C[i - 1][j] + 1, C[i][j - 1] + 1));

}

}

}

// Reconstructing the solution

int len = n + m; // maximum possible length

int i = m;

int j = n;

int pos1 = len;

int pos2 = len;

string Slans(len + 1, ' ');

string S2ans(len + 1, ' ');

while (!(i == 0 && j == 0)) {

if (S1[i - 1] == S2[j - 1]) {

Slans[pos1--] = S1[i - 1];

S2ans[pos2--] = S2[j - 1];

i--;

j--;

} else if (C[i - 1][j - 1] + 1 == C[i][j]) {

Slans[pos1--] = S1[i - 1];

S2ans[pos2--] = S2[j - 1];

i--;

j--;

} else if (C[i - 1][j] + 1 == C[i][j]) {

Slans[pos1--] = S1[i - 1];

S2ans[pos2--] = '\_';

i--;

} else if (C[i][j - 1] + 1 == C[i][j]) {

Slans[pos1--] = '\_';

S2ans[pos2--] = S2[j - 1];

j--;

}

}

while (pos1 > 0) {

if (i > 0) {

Slans[pos1--] = S1[--i];

} else {

Slans[pos1--] = '\_';

}

}

while (pos2 > 0) {

if (j > 0) {

S2ans[pos2--] = S2[--j];

} else {

S2ans[pos2--] = '\_';

}

}

// Since we have assumed the answer to be n+m long,

// we need to remove the extra gaps in the starting

int id = 1;

for (i = 1; i <= len; i++) {

if (S2ans[i] != '\_' && Slans[i] != '\_') {

id = i;

break;

}

}

cout << "Minimum Penalty in aligning the strings: " << C[n][m] << endl;

cout << "The aligned strings are: " << endl;

for (i = id; i <= len; i++) {

cout << Slans[i];

}

cout << endl;

for (i = id; i <= len; i++) {

cout << S2ans[i];

}

cout << endl;

}

**5. Space and time complexity:**

The time complexity of the dynamic programming solution is **O(m \* n)**, where m and n are the lengths of the input strings S1 and S2, respectively. This is because we fill in a table of size (m+1) x (n+1) to store the optimal substructure answers. The space complexity is also **O(m \* n)** as we require a table of the same size to store the intermediate results.