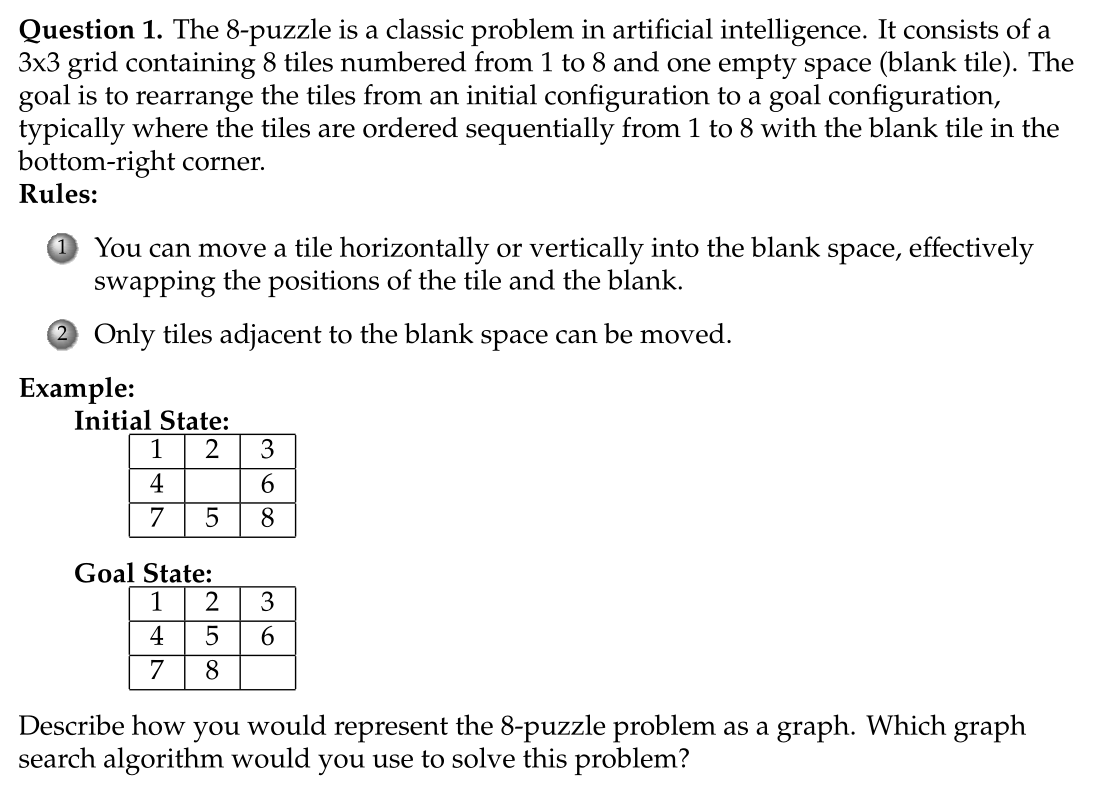
Algorithms and Data Structures  
Homework 2

**Student Name:  
ID:**This homework is divided into two parts:

# Part 1 Graph Algorithms

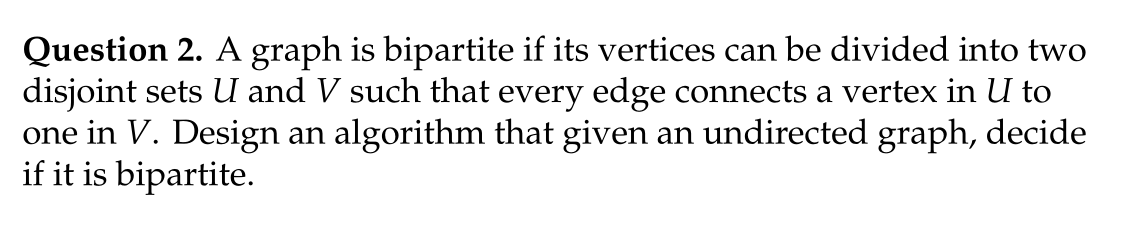


**Solution:** Graph Representation

1. Nodes: Each node represents a configuration of the 3x3 grid, including the positions of the tiles and the blank space. There are in total 9! = 362,880 nodes.
2. Edges: An edge represents a valid move of a tile into the blank space, transitioning from one configuration to another.
3. Initial and Goal States: The given configurations correspond to the start and target nodes.

Algorithm: **BFS**

1. BFS explores the state-space graph level by level, ensuring that the first time the goal state is reached, the shortest solution has been found.
2. Use a queue to maintain the current state and the sequence of moves made to reach it.
3. Use a set to track visited states and avoid revisiting them, reducing redundant computation.



**Solution:** To determine if an undirected graph is bipartite, we can use a graph traversal algorithm (DFS or BFS) to attempt to color the graph with two colors. If this is possible, the graph is bipartite; otherwise, it is not.

**Idea**

1. A graph is bipartite if it is possible to color its vertices with two colors such that no two adjacent vertices share the same color.
2. Use **DFS** or **BFS** to traverse the graph and try to assign colors alternately to neighboring vertices.
3. If at any point a vertex is found to have the same color as one of its neighbors, the graph is not bipartite.

**Algorithm (Using BFS)**

**Input:**

* An undirected graph represented as an adjacency list G.

**Output:**

* True if the graph is bipartite, False otherwise.

**Steps:**

1. Initialize a dictionary color to store the color of each vertex. Use {0,1} as the two colors.
2. Iterate through all vertices in the graph to handle disconnected components.
3. For each unvisited vertex, perform a BFS:
   * Assign the starting vertex a color (e.g., 0).
   * For each vertex visited, attempt to assign its neighbors the opposite color.
   * If a neighbor already has the same color as the current vertex, the graph is not bipartite.
4. If all vertices can be colored successfully, return True.

**Pseudocode**

def is\_bipartite(graph):

# Dictionary to store color of each vertex (0 or 1)

color = {}

def bfs(start):

queue = [start]

color[start] = 0 # Assign the first color to the starting node

while queue:

current = queue.pop(0)

for neighbor in graph[current]:

if neighbor not in color:

# Assign the opposite color to the neighbor

color[neighbor] = 1 - color[current]

queue.append(neighbor)

elif color[neighbor] == color[current]:

# Conflict detected

return False

return True

# Iterate through all vertices to handle disconnected graphs

for vertex in graph:

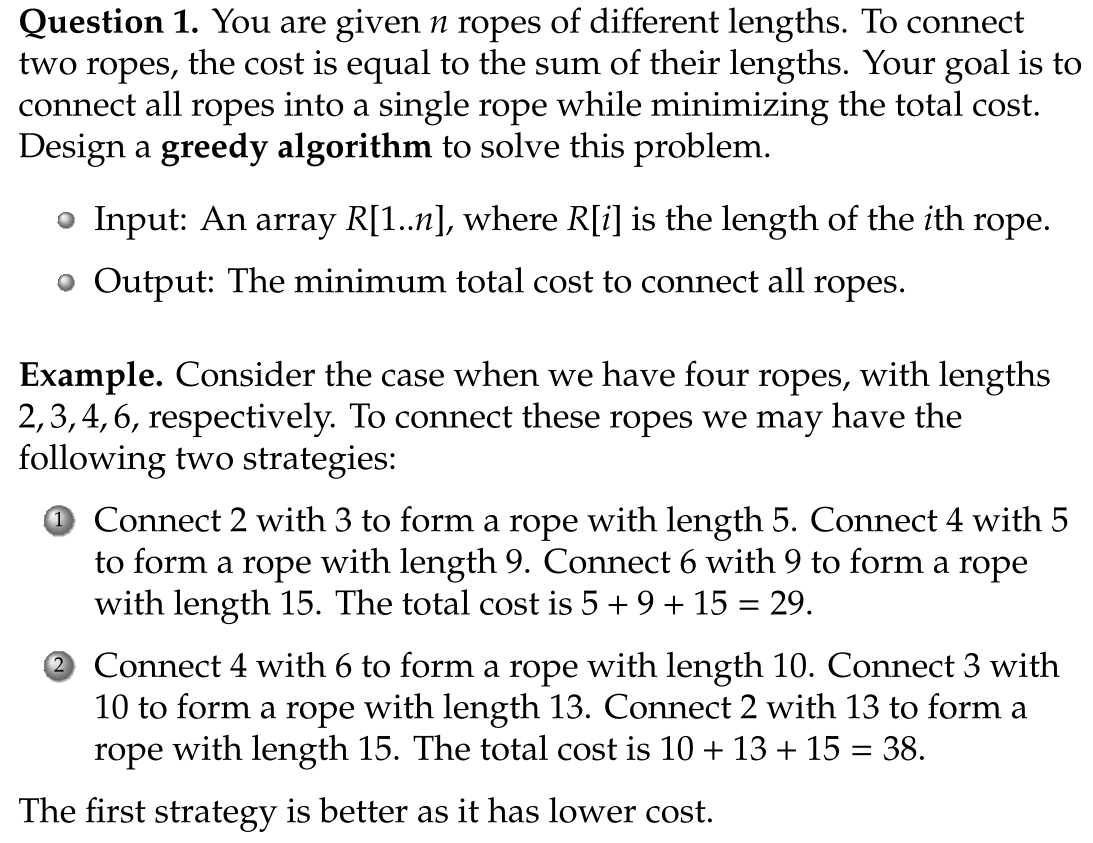
if vertex not in color:

if not bfs(vertex):

return False

return True

# Part II Greedy Algorithms and Dynamic Programming



**Sample Solution:** The **greedy algorithm** works by always combining the two smallest ropes first. This minimizes the intermediate cost at every step, ensuring that the overall cost is minimized.

**Why This Works (Justification of the Greedy Approach):**

* **Greedy Choice Property**: Combining the two shortest ropes minimizes the immediate cost and does not affect the optimality of the remaining subproblem. If you choose any other pair instead of the smallest two, the cost will increase unnecessarily.
* **Optimal Substructure**: In an optimal solution (the best way to connect all ropes), consider the last step. Suppose in the last step you connect two last remaining ropes, say x and y, into a single rope. Then in this solution, the strategy that you used to connect ropes to form x and y are also optimal, i.e., you used the least cost to form ropes x and y.

**Steps to Solve Using a Min-Heap**

1. **Insert all rope lengths into a min-heap**:
   * A min-heap allows efficient extraction of the two smallest ropes in O(log⁡n)O(\log n) time.
2. **Combine the two smallest ropes**:
   * Remove the two smallest elements from the heap, compute their sum, and add the result back to the heap.
   * Keep track of the cost incurred during this operation.
3. **Repeat until there is only one rope left in the heap**:
   * The total cost of all combinations is the final answer.

**Pseudocode**

function minimizeCost(R):

// Step 1: Insert all rope lengths into a min-heap

heap = new MinHeap()

for length in R:

heap.insert(length)

totalCost = 0

// Step 2: Combine ropes until only one rope remains

while heap.size() > 1:

// Extract the two smallest ropes

rope1 = heap.extractMin()

rope2 = heap.extractMin()

// Combine the two ropes

cost = rope1 + rope2

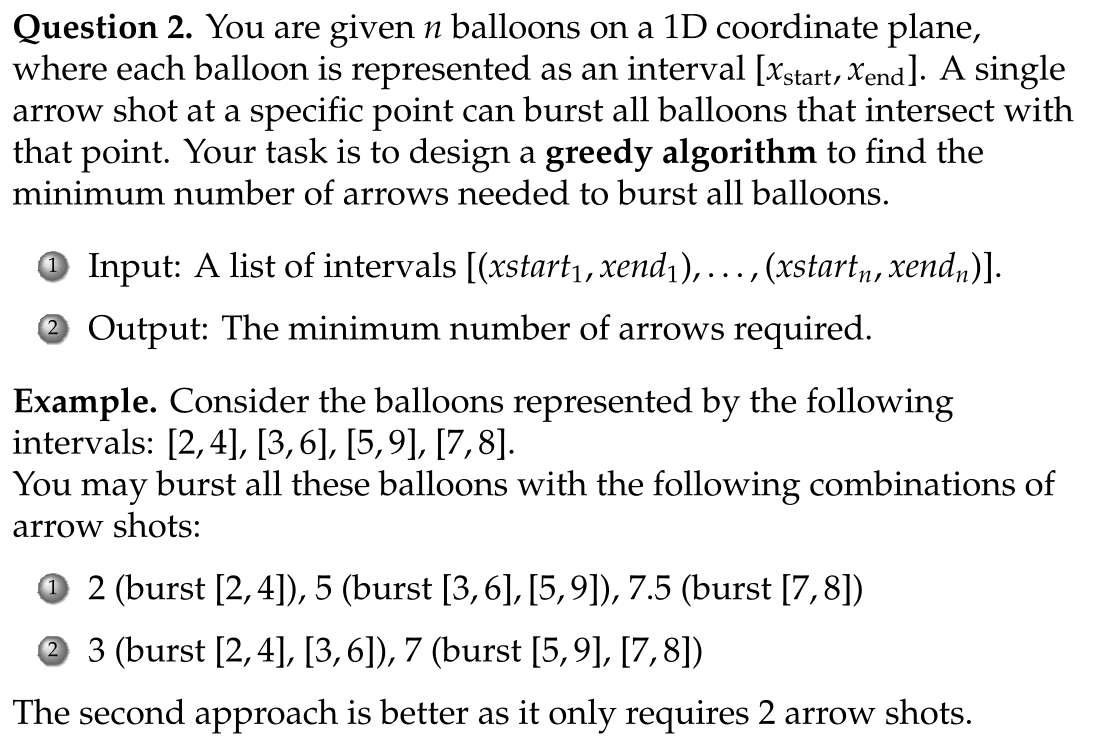
totalCost += cost

// Insert the combined rope back into the heap

heap.insert(cost)

// Step 3: Return the total cost

return totalCost



**Sample Solution:** This problem can be solved using a **greedy algorithm**.

**Key Idea:**

Sort the intervals by their end points. Place an arrow at the end of the first interval. For every subsequent interval, check if it overlaps with the current arrow position. If it does, the arrow bursts the balloon, and no new arrow is needed. Otherwise, place a new arrow at the end of the current interval.

**Why This Approach Works**

**Greedy Choice Property:**

* Placing the arrow at the end of the earliest finishing interval ensures that we burst as many balloons as possible with that arrow, minimizing the number of arrows used.

**Optimal Substructure:**

* After handling the first interval, the remaining problem is to find the minimum arrows for the remaining intervals, which follows the same logic.

**Pseudocode**

function findMinArrows(intervals):

if intervals is empty:

return 0

// Step 1: Sort intervals by their end points

sort intervals by xend

// Step 2: Initialize variables

arrows = 1

arrow\_position = intervals[0].end

// Step 3: Iterate through intervals

for interval in intervals:

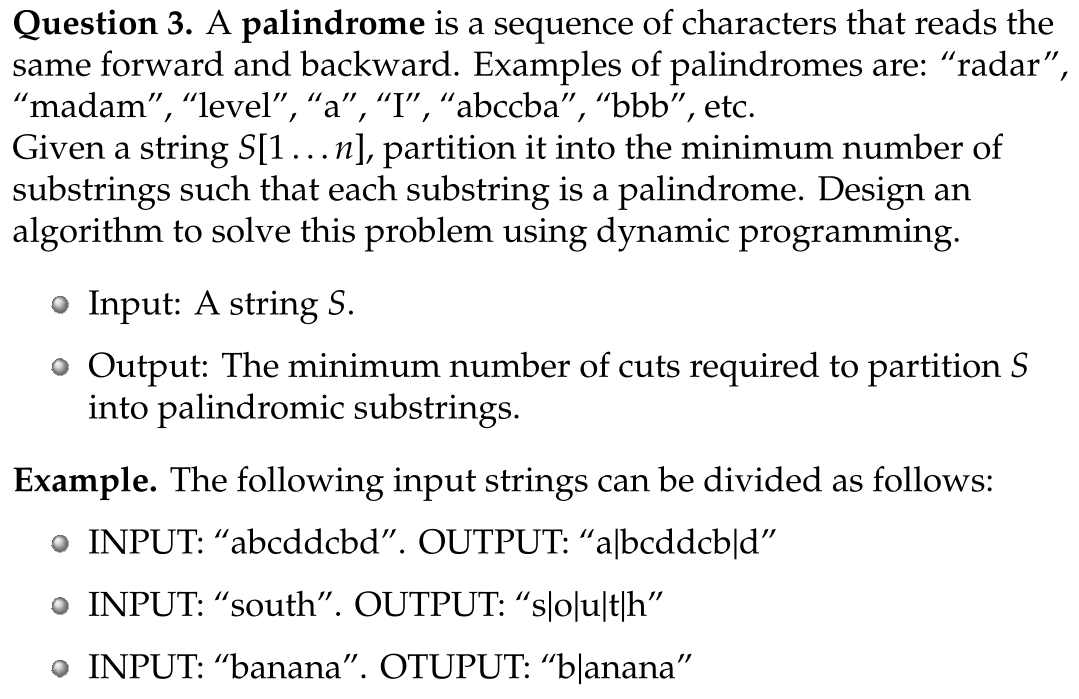
if interval.start > arrow\_position:

// Current balloon cannot be burst by the last arrow

arrows += 1

arrow\_position = interval.end

return arrows



**Sample Solution:** We solve this problem using two dynamic programming tables:

1. **Palindrome Table** :
   * if is a palindrome.
   * otherwise.
2. **Minimum Cuts Table** :
   * stores the minimum number of cuts needed to partition the substring into palindromes.

**Algorithm**

**Step 1: Build the Palindrome Table**

1. **Base Case**:
   * , because any single character is a palindrome.
   * if , because two identical adjacent characters form a palindrome.
2. **General Case**:
   * For , P[i][j]=True if:
     + , and
     + .

**Step 2: Compute the Minimum Cuts Table**

1. **Base Case**:
   * , because no cuts are needed if the string has only one character.
2. **General Case**:
   * For each , if , then .
   * Otherwise:
     + Compute

**Pseudocode**

function minCuts(S):

n = length(S)

// Step 1: Build the palindrome table

P = array of size (n+1) x (n+1) initialized to False

for i = 1 to n:

P[i][i] = True // Single character is a palindrome

if i < n and S[i] == S[i+1]:

P[i][i+1] = True // Two identical adjacent characters

for length = 3 to n: // Consider substrings of length >= 3

for i = 1 to n-length+1:

j = i + length - 1

if S[i] == S[j] and P[i+1][j-1]:

P[i][j] = True

// Step 2: Compute the minimum cuts table

C = array of size n+1 initialized to infinity

C[0] = 0 // No cuts needed for an empty string

for i = 1 to n:

if P[1][i]: // If the whole substring S[1..i] is a palindrome

C[i] = 0

else:

for k = 1 to i-1:

if P[k+1][i]: // If S[k+1..i] is a palindrome

C[i] = min(C[i], C[k] + 1)

return C[n]

**Example**

**Input:**

S = "abccbc"

**Palindrome Table (PP):**

| **i\j** | **1** | **2** | **3** | **4** | **5** | **6** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | T | F | F | F | F | F |
| 2 |  | T | F | T | F | F |
| 3 |  |  | T | F | T | F |
| 4 |  |  |  | T | F | T |
| 5 |  |  |  |  | T | F |
| 6 |  |  |  |  |  | T |

**Minimum Cuts Table (CC):**

| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| C | 0 | 0 | 1 | 2 | 1 | 2 | 3 |

**Output:**

Minimum cuts = 3

Explanation: Partition "a | bccb | c".