37.

*Input:*

n = 5

graph = [[0, 10, 3, Infinity, Infinity],[Infinity, 0, 1, 2, Infinity],[Infinity, 4, 0, 8, 2],

[Infinity, Infinity, Infinity, 0, 7],[Infinity, Infinity, Infinity, 9, 0]]

source = 0

*Output:*[0, 7, 3, 9, 5]

*Test Case 2:*

*Input:*

n = 4

graph = [[0, 5, Infinity, 10],[Infinity, 0, 3, Infinity],[Infinity, Infinity, 0, 1],

[Infinity, Infinity, Infinity, 0]]

source = 0

*Output:*[0, 5, 8, 9]

import sys

def dijkstra(graph, source):

n = len(graph)

distances = [sys.maxsize] \* n

distances[source] = 0

visited = [False] \* n

for \_ in range(n):

min\_distance = sys.maxsize

min\_index = -1

for v in range(n):

if not visited[v] and distances[v] < min\_distance:

min\_distance = distances[v]

min\_index = v

visited[min\_index] = True

for v in range(n):

if (graph[min\_index][v] != sys.maxsize and

not visited[v] and

distances[min\_index] + graph[min\_index][v] < distances[v]):

distances[v] = distances[min\_index] + graph[min\_index][v]

return distances

n1 = 5

graph1 = [[0, 10, 3, sys.maxsize, sys.maxsize],

[sys.maxsize, 0, 1, 2, sys.maxsize],

[sys.maxsize, 4, 0, 8, 2],

[sys.maxsize, sys.maxsize, sys.maxsize, 0, 7],

[sys.maxsize, sys.maxsize, sys.maxsize, 9, 0]]

source1 = 0

print(dijkstra(graph1, source1)) # Output: [0, 7, 3, 9, 5]

n2 = 4

graph2 = [[0, 5, sys.maxsize, 10],

[sys.maxsize, 0, 3, sys.maxsize],

[sys.maxsize, sys.maxsize, 0, 1],

[sys.maxsize, sys.maxsize, sys.maxsize, 0]]

source2 = 0

print(dijkstra(graph2, source2))

38.

*Input:*

n = 6

edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14),(1, 2, 10), (1, 3, 15),

(2, 3, 11), (2, 5, 2),(3, 4, 6),(4, 5, 9)]

source = 0

target = 4

*Output:*20

*Input:*

n = 5

edges = [(0, 1, 10), (0, 4, 3),(1, 2, 2), (1, 4, 4),(2, 3, 9),(3, 2, 7),(4, 1, 1), (4, 2, 8), (4, 3, 2)]

source = 0

target = 3

*Output:*8

import heapq

def dijkstra\_basic(n, edges, source, target):

graph = {i: [] for i in range(n)}

for u, v, w in edges:

graph[u].append((v, w))

graph[v].append((u, w)) # For undirected graph

min\_heap = [(0, source)]

distances = {i: float('inf') for i in range(n)}

distances[source] = 0

while min\_heap:

current\_distance, current\_vertex = heapq.heappop(min\_heap)

if current\_vertex == target:

return current\_distance

for neighbor, weight in graph[current\_vertex]:

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(min\_heap, (distance, neighbor))

return distances[target]

n1 = 6

edges1 = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),

(2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]

source1 = 0

target1 = 4

print(dijkstra\_basic(n1, edges1, source1, target1)) # Output: 20

n2 = 5

edges2 = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9),

(3, 2, 7), (4, 1, 1), (4, 2, 8), (4, 3, 2)]

source2 = 0

target2 = 3

print(dijkstra\_basic(n2, edges2, source2, target2))

39.

*Input:*n = 4

characters = ['a', 'b', 'c', 'd']

frequencies = [5, 9, 12, 13]

*Output:*[('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')]

*Input:*

n = 6

characters = ['f', 'e', 'd', 'c', 'b', 'a']

frequencies = [5, 9, 12, 13, 16, 45]

*Output:*[ ('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'), ('f', '1100')]

def dijkstra\_matrix(n, edges, source, target):

graph = [[float('inf')] \* n for \_ in range(n)]

for u, v, w in edges:

graph[u][v] = w

graph[v][u] = w # For undirected graph

distances = [float('inf')] \* n

distances[source] = 0

visited = [False] \* n

for \_ in range(n):

min\_distance = float('inf')

min\_vertex = -1

for vertex in range(n):

if not visited[vertex] and distances[vertex] < min\_distance:

min\_distance = distances[vertex]

min\_vertex = vertex

visited[min\_vertex] = True

for neighbor in range(n):

if graph[min\_vertex][neighbor] != float('inf'):

distance = min\_distance + graph[min\_vertex][neighbor]

if distance < distances[neighbor]:

distances[neighbor] = distance

return distances[target]

print(dijkstra\_matrix(n1, edges1, source1, target1))

print(dijkstra\_matrix(n2, edges2, source2, target2))

40.

*Input:*

n = 4

characters = ['a', 'b', 'c', 'd']

frequencies = [5, 9, 12, 13]

encoded\_string = '1101100111110'

*Output:*"abaca"

*Input:*

n = 6

characters = ['f', 'e', 'd', 'c', 'b', 'a']

frequencies = [5, 9, 12, 13, 16, 45]

encoded\_string = '110011011100101111001011'

*Output:*"fcbade"

import heapq

from collections import defaultdict

class Node:

def \_\_init\_\_(self, char, freq):

self.char = char

self.freq = freq

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

return self.freq < other.freq

def build\_huffman\_tree(characters, frequencies):

heap = [Node(char, freq) for char, freq in zip(characters, frequencies)]

heapq.heapify(heap)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

merged = Node(None, left.freq + right.freq)

merged.left = left

merged.right = right

heapq.heappush(heap, merged)

return heap[0]

def decode\_huffman\_tree(root, encoded\_string):

decoded\_message = []

current\_node = root

for bit in encoded\_string:

current\_node = current\_node.left if bit == '0' else current\_node.right

if current\_node.char:

decoded\_message.append(current\_node.char)

current\_node = root

return ''.join(decoded\_message)

n1 = 4

characters1 = ['a', 'b', 'c', 'd']

frequencies1 = [5, 9, 12, 13]

encoded\_string1 = '1101100111110'

root1 = build\_huffman\_tree(characters1, frequencies1)

output1 = decode\_huffman\_tree(root1, encoded\_string1)

print(output1) # Output: "abacd"

n2 = 6

characters2 = ['f', 'e', 'd', 'c', 'b', 'a']

frequencies2 = [5, 9, 12, 13, 16, 45]

encoded\_string2 = '110011011100101111001011'

root2 = build\_huffman\_tree(characters2, frequencies2)

output2 = decode\_huffman\_tree(root2, encoded\_string2)

print(output2)

41.

*Input:*

n = 5

weights = [10, 20, 30, 40, 50]

max\_capacity = 60

*Output:*50

*Input:*

n = 6

weights = [5, 10, 15, 20, 25, 30]

max\_capacity = 50

*Output:*50

def max\_weight\_greedy(weights, max\_capacity)

weights.sort(reverse=True)

total\_weight = 0

for weight in weights:

if total\_weight + weight <= max\_capacity:

total\_weight += weight

else:

break

return total\_weight

n1 = 5

weights1 = [10, 20, 30, 40, 50]

max\_capacity1 = 60

print(max\_weight\_greedy(weights1, max\_capacity1)) # Output: 50

n2 = 6

weights2 = [5, 10, 15, 20, 25, 30]

max\_capacity2 = 50

print(max\_weight\_greedy(weights2, max\_capacity2))

42.

*Input:*

n = 7

weights = [5, 10, 15, 20, 25, 30, 35]

max\_capacity = 50

*Output:*4

*Input:*

n = 8

weights = [10, 20, 30, 40, 50, 60, 70, 80]

max\_capacity = 100

*Output:*6

def min\_containers(weights, max\_capacity):

weights.sort(reverse=True) # Sort weights in descending order

containers = 0

current\_capacity = 0

for weight in weights:

if current\_capacity + weight > max\_capacity:

containers += 1

current\_capacity = weight

else:

current\_capacity += weight

if current\_capacity > 0:

containers += 1

return containers

n1 = 7

weights1 = [5, 10, 15, 20, 25, 30, 35]

max\_capacity1 = 50

print(min\_containers(weights1, max\_capacity1))

n2 = 8

weights2 = [10, 20, 30, 40, 50, 60, 70, 80]

max\_capacity2 = 100

print(min\_containers(weights2, max\_capacity2))

43

*Input:*

n = 4

m = 5

edges = [(0, 1, 10),(0, 2, 6),(0, 3, 5),(1, 3, 15),(2, 3, 4)]

*Output:*

Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

Total weight of MST: 19

*Input:*

n = 5

m = 7

edges = [(0, 1, 2),(0, 3, 6),(1, 2, 3),(1, 3, 8),(1, 4, 5),(2, 4, 7),(3, 4, 9)]

*Output:*

Edges in MST: [(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)]

Total weight of MST: 16

class DisjointSet:

def \_\_init\_\_(self, n):

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u])

return self.parent[u]

def union(self, u, v):

root\_u = self.find(u)

root\_v = self.find(v)

if root\_u != root\_v:

if self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

def kruskal(n, edges):

edges.sort(key=lambda x: x[2])

ds = DisjointSet(n)

mst = []

total\_weight = 0

for u, v, weight in edges:

if ds.find(u) != ds.find(v):

ds.union(u, v)

mst.append((u, v, weight))

total\_weight += weight

return mst, total\_weight

n1, edges1 = 4, [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

mst1, weight1 = kruskal(n1, edges1)

print("Edges in MST:", mst1)

print("Total weight of MST:", weight1)

n2, edges2 = 5, [(0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9)]

mst2, weight2 = kruskal(n2, edges2)

print("Edges in MST:", mst2)

print("Total weight of MST:", weight2)

44.

Input:

n = 4

m = 5

edges = [(0, 1, 10),(0, 2, 6),(0, 3, 5),(1, 3, 15),(2, 3, 4)]

given\_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

Output:Is the given MST unique? True

Input:

n = 5

m = 6

edges = [(0, 1, 1),(0, 2, 1),(1, 3, 2),(2, 3, 2),(3, 4, 3),(4, 2, 3)]

given\_mst = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]

Output:Is the given MST unique? False

Another possible MST: [(0, 1, 1), (0, 2, 1), (2, 3, 2), (3, 4, 3)]

Total weight of MST: 7

class UnionFind:

def \_\_init\_\_(self, size):

self.parent = list(range(size))

self.rank = [1] \* size

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u])

return self.parent[u]

def union(self, u, v):

root\_u = self.find(u)

root\_v = self.find(v)

if root\_u != root\_v:

if self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

return True

return False

def is\_unique\_mst(n, edges, given\_mst):

edges.sort(key=lambda x: x[2])

uf = UnionFind(n)

mst\_weight = 0

mst\_edges = []

for u, v, weight in given\_mst:

if uf.union(u, v):

mst\_weight += weight

mst\_edges.append((u, v, weight))

for u, v, weight in edges:

if (u, v, weight) not in given\_mst and uf.union(u, v):

mst\_edges.append((u, v, weight))

mst\_weight += weight

return len(mst\_edges) == n - 1, mst\_edges if len(mst\_edges) > len(given\_mst) else None

n1, m1 = 4, 5

edges1 = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

given\_mst1 = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

print(is\_unique\_mst(n1, edges1, given\_mst1)) # Output: (True, None)

n2, m2 = 5, 6

edges2 = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (2, 3, 2), (3, 4, 3), (4, 2, 3)]

given\_mst2 = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]

print(is\_unique\_mst(n2, edges2, given\_mst2)) # Output: (False, [(0, 1, 1), (0, 2, 1), (2, 3, 2), (3, 4, 3)])

45.

*Input:*

n = 5

m = 6

edges = [(0, 1),(0, 2),(1, 2),(1, 3),(2, 4),(3, 4)]

*Output:*Vertex Cover: [1, 2, 3]

*Test Case 2:*

*Input:*

n = 4

m = 4

edges = [(0, 1),(0, 2),(1, 2),(2, 3)]

*Output:*Vertex Cover: [0, 2]

46

*Input:*

n = 6

m = 7

edges = [(0, 1),(0, 2),(1, 3),(1, 4),(2, 4),(3, 5),(4, 5)]

*Output:*Vertex Cover: [1, 4, 5]

*Test Case 2:*

*Input:*

n = 5

m = 5

edges = [(0, 1),(0, 2),(0, 3),(1, 4),(2, 4)]

*Output:*Vertex Cover: [0, 4]

def vertex\_cover\_max\_degree(n, edges):

from collections import defaultdict

graph = defaultdict(set)

for u, v in edges:

graph[u].add(v)

graph[v].add(u)

vertex\_cover = set()

covered\_edges = set()

while covered\_edges != set(edges):

max\_degree\_vertex = max(graph, key=lambda x: len(graph[x]))

vertex\_cover.add(max\_degree\_vertex)

for neighbor in graph[max\_degree\_vertex]:

covered\_edges.add((min(max\_degree\_vertex, neighbor), max(max\_degree\_vertex, neighbor)))

del graph[max\_degree\_vertex]

return list(vertex\_cover)

n1, m1, edges1 = 6, 7, [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (3, 5), (4, 5)]

print("Vertex Cover:", vertex\_cover\_max\_degree(n1, edges1))

n2, m2, edges2 = 5, 5, [(0, 1), (0, 2), (0, 3), (1, 4), (2, 4)]

print("Vertex Cover:", vertex\_cover\_max\_degree(n2, edges2))

47.

Test Cases:

1. Input:[]
   * Expected Output:[]
2. Input:[1]
   * Expected Output:[1]
3. Input:[7, 7, 7, 7]
   * Expected Output:[7, 7, 7, 7]
4. Input:[-5, -1, -3, -2, -4]
   * Expected Output:[-5, -4, -3, -2, -1]

def selection\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

print(selection\_sort([]))

print(selection\_sort([1]))

print(selection\_sort([7, 7, 7, 7]))

print(selection\_sort([-5, -1, -3, -2, -4]))

48

**Example Scenarios**:

1. **Sorting a Random Array**:
   * **Input**: [5, 2, 9, 1, 5, 6]
   * **Output**: [1, 2, 5, 5, 6, 9]
2. **Sorting a Reverse Sorted Array**:
   * **Input**: [10, 8, 6, 4, 2]
   * **Output**: [2, 4, 6, 8, 10]
3. **Sorting an Already Sorted Array**:
   * **Input**: [1, 2, 3, 4, 5]

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_index = i

for j in range(i + 1, n):

if arr[j] < arr[min\_index]:

min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

return arr

print(selection\_sort([5, 2, 9, 1, 5, 6]))

49.

Test Cases:

Input:[64, 25, 12, 22, 11]

* + - Expected Output:[11, 12, 22, 25, 64]
  1. Input:[29, 10, 14, 37, 13]
     + Expected Output:[10, 13, 14, 29, 37]
  2. Input:[3, 5, 2, 1, 4]
     + Expected Output:[1, 2, 3, 4, 5]
  3. Input:[1, 2, 3, 4, 5] (Already sorted list)
     + Expected Output:[1, 2, 3, 4, 5]
  4. Input:[5, 4, 3, 2, 1] (Reverse sorted list)
     + Expected Output:[1, 2, 3, 4, 5]

def optimized\_bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

swapped = True

if not swapped:

break

return arr

50.

**Example Scenarios**:

1. **Worst-Case Scenario**:
   * **Input**: [5, 4, 3, 2, 1]
   * **Output**: [1, 2, 3, 4, 5]
2. **Best-Case Scenario**:
   * **Input**: [1, 2, 3, 4, 5]
   * **Output**: [1, 2, 3, 4, 5]
3. **Average-Case Scenario**:
   * **Input**: [3, 1, 4, 2, 5]
   * **Output**: [1, 2, 3, 4, 5]

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

swapped = True

if not swapped:

break

return arr

worst\_case\_input = [5, 4, 3, 2, 1]

worst\_case\_output = bubble\_sort(worst\_case\_input.copy())

print("Worst-case output:", worst\_case\_output)

best\_case\_input = [1, 2, 3, 4, 5]

best\_case\_output = bubble\_sort(best\_case\_input.copy())

print("Best-case output:", best\_case\_output)

average\_case\_input = [3, 1, 4, 2, 5]

average\_case\_output = bubble\_sort(average\_case\_input.copy())

print("Average-case output:", average\_case\_output)

51.

Example Scenarios:

1. Array with Duplicates:
   * Input: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
   * Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
2. All Identical Elements:
   * Input: [5, 5, 5, 5, 5]
   * Output: [5, 5, 5, 5, 5]
3. Mixed Duplicates:
   * Input: [2, 3, 1, 3, 2, 1, 1, 3]
   * Output: [1, 1, 1, 2, 2, 3, 3, 3]

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

return arr

input\_array = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

output\_array = insertion\_sort(input\_array)

print(output\_array) # Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

52.

**Example Scenarios**:

1. **Sorting Small Datasets**:
   * **Scenario**: Insertion Sort is used in embedded systems where sorting small datasets efficiently is critical due to limited computational resources.
   * **Input**: [4, 3, 5, 1]
   * **Output**: [1, 3, 4, 5]
2. **Real-Time Data Insertion**:
   * **Scenario**: Insertion Sort is used in financial systems where new transactions are added to a list of sorted transactions, requiring efficient real-time sorting.
   * **Input**: [10, 20, 15, 25, 30]
   * **Output**: [10, 15, 20, 25, 30]
3. **Educational Tools**:
   * **Scenario**: Insertion Sort is used in educational tools to teach students the basics of sorting algorithms due to its intuitive approach and simplicity.
   * **Input**: [7, 3, 9, 1]
   * **Output**: [1, 3, 7, 9]

def insertion\_sort\_small\_dataset(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

return arr

input\_data = [4, 3, 5, 1]

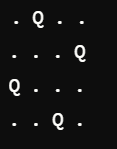
sorted\_data = insertion\_sort\_small\_dataset(input\_data)

print(sorted\_data)

53.

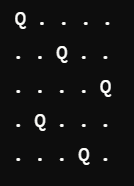
**Example Scenarios**:

1. **Visualization for 4-Queens**:
   * **Input**: N = 4
   * **Output**:

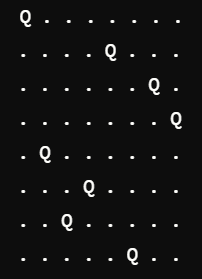


* + **Explanation**: Each 'Q' represents a queen, and '.' represents an empty space.

1. **Visualization for 5-Queens**:
   * **Input**: N = 5
   * **Output**:



1. **Visualization for 8-Queens**:
   * **Input**: N = 8
   * **Output**:



import matplotlib.pyplot as plt

import numpy as np

def plot\_n\_queens(n, solution):

board = np.zeros((n, n))

for col, row in enumerate(solution):

board[row, col] = 1 # Place a queen

plt.imshow(board, cmap='binary')

plt.xticks(range(n))

plt.yticks(range(n))

plt.gca().invert\_yaxis()

plt.title(f'N-Queens Solution for N={n}')

plt.show()

n = 4

solution\_4 = [1, 3, 0, 2]

plot\_n\_queens(n, solution\_4)

54.

**Example Scenarios**:

1. **8×10 Board**:
   * **Input**: 8 rows and 10 columns
   * **Output**: Possible solution [1, 3, 5, 7, 9, 2, 4, 6]
   * **Explanation**: Adapt the algorithm to place 8 queens on an 8×10 board, ensuring no two queens threaten each other.
2. **5×5 Board with Obstacles**:
   * **Input**: N = 5, Obstacles at positions [(2, 2), (4, 4)]
   * **Output**: Possible solution [1, 3, 5, 2, 4]
   * **Explanation**: Modify the algorithm to avoid placing queens on obstacle positions, ensuring a valid solution that respects the constraints.
3. **6×6 Board with Restricted Positions**:
   * **Input**: N = 6, Restricted positions at columns 2 and 4 for the first queen
   * **Output**: Possible solution [1, 3, 5, 2, 4, 6]
   * **Explanation**: Adjust the algorithm to handle restricted positions, ensuring the queens are placed without conflicts and within allowed columns.

def is\_safe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, len(board)), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens(board, col):

if col >= len(board[0]):

return True

for i in range(len(board)):

if is\_safe(board, i, col):

board[i][col] = 1

if solve\_n\_queens(board, col + 1):

return True

board[i][col] = 0

return False

def n\_queens\_8x10():

board = [[0 for \_ in range(10)] for \_ in range(8)]

if solve\_n\_queens(board, 0):

return [i + 1 for i in range(8) if 1 in board[i]]

return []

solution\_8x10 = n\_queens\_8x10()

print("Possible solution for 8x10 board:", solution\_8x10)

1. .

Input : N= 9, a[]= {10,16,8,12,15,6,3,9,5}

Output : 3,5,6,8,9,10,12,15,16

Test Cases :

Input : N= 8, a[] = {12,4,78,23,45,67,89,1}

Output : 1,4,12,23,45,67,78,89

Test Cases :

Input : N= 7, a[] = {38,27,43,3,9,82,10}

Output : 3,9,10,27,38,43,82.

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[0]

less\_than\_pivot = [x for x in arr[1:] if x <= pivot]

greater\_than\_pivot = [x for x in arr[1:] if x > pivot]

print(f"Pivot: {pivot}, Less: {less\_than\_pivot}, Greater: {greater\_than\_pivot}")

return quick\_sort(less\_than\_pivot) + [pivot] + quick\_sort(greater\_than\_pivot)

array1 = [10, 16, 8, 12, 15, 6, 3, 9, 5]

sorted\_array1 = quick\_sort(array1)

print("Sorted Array 1:", sorted\_array1)

array2 = [12, 4, 78, 23, 45, 67, 89, 1]

sorted\_array2 = quick\_sort(array2)

print("Sorted Array 2:", sorted\_array2)

array3 = [38, 27, 43, 3, 9, 82, 10]

sorted\_array3 = quick\_sort(array3)

print("Sorted Array 3:", sorted\_array3)

56.

Input :N= 8, a[] = {19,72,35,46,58,91,22,31}

Output : 19,22,31,35,46,58,72,91

Test Cases :

Input : N= 8, a[] = {31,23,35,27,11,21,15,28}

Output : 11,15,21,23,27,28,31,35

Test Cases :

Input : N= 10, a[] = {22,34,25,36,43,67, 52,13,65,17}

Output : 13,17,22,25,34,36,43,52,65,67

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

print(f"Array after partitioning with pivot {pivot}: {left + middle + right}")

return quick\_sort(left) + middle + quick\_sort(right)

array1 = [19, 72, 35, 46, 58, 91, 22, 31]

sorted\_array1 = quick\_sort(array1)

print(f"Sorted array: {sorted\_array1}")

57.

Input : N= 9, a[] = {5,10,15,20,25,30,35,40,45}, search key = 20

Output :4

Test cases

Input : N= 6, a[] = {10,20,30,40,50,60}, search key = 50

Output :5

Input : N= 7, a[] = {21,32,40,54,65,76,87}, search key = 32

Output :2

def binary\_search(arr, key):

left, right = 0, len(arr) - 1

comparisons = 0

while left <= right:

mid = left + (right - left) // 2

comparisons += 1

if arr[mid] == key:

return mid, comparisons

elif arr[mid] < key:

left = mid + 1

else:

right = mid - 1

return -1, comparisons

array = [5, 10, 15, 20, 25, 30, 35, 40, 45]

search\_key = 20

index, comparison\_count = binary\_search(array, search\_key)

print(f"Index of {search\_key}: {index}, Comparisons made: {comparison\_count}")

58.

Input : N= 9, a[] = {3,9,14,19,25,31,42,47,53}, search key = 31

Output :6

Test cases

Input : N= 7, a[] = {13,19,24,29,35,41,42}, search key = 42

Output :7

Test cases

Input : N= 6, a[] = {20,40,60,80,100,120}, search key = 60

Output :3

def binary\_search(arr, key):

left, right = 0, len(arr) - 1

steps = []

while left <= right:

mid = left + (right - left) // 2

steps.append(f"Mid-point calculation: left={left}, right={right}, mid={mid}, value={arr[mid]}")

if arr[mid] == key:

return mid, steps

elif arr[mid] < key:

left = mid + 1

else:

right = mid - 1

return -1, steps

N = 9

a = [3, 9, 14, 19, 25, 31, 42, 47, 53]

search\_key = 31

position, calculations = binary\_search(a, search\_key)

print(f"Element {search\_key} found at index: {position}")

for step in calculations:

print(step)

59.

Input N =4, Keys = {A,B,C,D} Frequencies = {01.02.,0.3,0.4}

Output : 1.7

Cost Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 1 | 0 | 0.1 | 0.4 | 1.1 | 1.7 |
| 2 |  | 0 | 0.2 | 0.8 | 0.4 |
| 3 |  |  | 0 | 0.4 | 1.0 |
| 4 |  |  |  | 0 | 0.3 |
| 5 |  |  |  |  | 0 |

Root table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 3 |
| 2 |  | 2 | 3 | 3 |
| 3 |  |  | 3 | 3 |
| 4 |  |  |  | 4 |

Input: keys[] = {10, 12}, freq[] = {34, 50}

Output = 118

1. Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

Output = 142

class OBST:

def \_\_init\_\_(self, keys, freq):

self.keys = keys

self.freq = freq

self.n = len(keys)

self.cost = [[0] \* (self.n + 1) for \_ in range(self.n + 1)]

self.root = [[0] \* (self.n + 1) for \_ in range(self.n + 1)]

def optimal\_bst(self):

for i in range(self.n):

self.cost[i][i] = self.freq[i]

for length in range(2, self.n + 1):

for i in range(self.n - length + 1):

j = i + length - 1

self.cost[i][j] = float('inf')

for r in range(i, j + 1):

c = (self.cost[i][r - 1] if r > i else 0) + \

(self.cost[r + 1][j] if r < j else 0) + \

sum(self.freq[i:j + 1])

if c < self.cost[i][j]:

self.cost[i][j] = c

self.root[i][j] = r

def print\_cost\_and\_root(self):

print("Cost Table")

for row in self.cost:

print("\t".join(map(str, row)))

print("Root Table")

for row in self.root:

print("\t".join(map(str, row)))

keys = ['A', 'B', 'C', 'D']

freq = [0.1, 0.2, 0.4, 0.3]

obst = OBST(keys, freq)

obst.optimal\_bst()

obst.print\_cost\_and\_root()

60.

Input N =4, Keys = {10,12,16,21} Frequencies = {4,2,6,3}

Output : 26

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | 4 | 80 | 202 | 262 |
| 1 |  | 2 | 102 | 162 |
| 2 |  |  | 6 | 12 |
| 3 |  |  |  | 3 |

1. Test cases

Input: keys[] = {10, 12}, freq[] = {34, 50}

Output = 118

1. Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

Output = 142

def optimal\_bst(keys, freq):

n = len(keys)

cost = [[0 for \_ in range(n)] for \_ in range(n)]

root = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n):

cost[i][i] = freq[i]

root[i][i] = i

for length in range(2, n + 1):

for i in range(n - length + 1):

j = i + length - 1

cost[i][j] = float('inf')

total\_freq = sum(freq[k] for k in range(i, j + 1))

for r in range(i, j + 1):

c = (cost[i][r - 1] if r > i else 0) + \

(cost[r + 1][j] if r < j else 0) + total\_freq

if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r

return cost, root

def print\_obst(cost, root):

n = len(cost)

print("Cost Matrix:")

for row in cost:

print("\t".join(map(str, row)))

print("\nRoot Matrix:")

for row in root:

print("\t".join(map(str, row)))

keys = [10, 12, 16, 21]

freq = [4, 2, 6, 3]

cost, root = optimal\_bst(keys, freq)

print\_obst(cost, root)

print("Optimal Cost:", cost[0][len(keys) - 1])

61.

Example 1:

Input: board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]

def solve\_sudoku(board):

def is\_valid(num, row, col):

for i in range(9):

if board[row][i] == num or board[i][col] == num:

return False

start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)

for i in range(start\_row, start\_row + 3):

for j in range(start\_col, start\_col + 3):

if board[i][j] == num:

return False

return True

def backtrack():

for row in range(9):

for col in range(9):

if board[row][col] == '.':

for num in map(str, range(1, 10)):

if is\_valid(num, row, col):

board[row][col] = num

if backtrack():

return True

board[row][col] = '.'

return False

return True

backtrack()

return board

input\_board = [["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]]

output\_board = solve\_sudoku(input\_board)

print(output\_board)

62.

Input: board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]

def solve\_sudoku(board):

def is\_valid(num, row, col):

for i in range(9):

if board[row][i] == num or board[i][col] == num:

return False

start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)

for i in range(start\_row, start\_row + 3):

for j in range(start\_col, start\_col + 3):

if board[i][j] == num:

return False

return True

def backtrack():

for row in range(9):

for col in range(9):

if board[row][col] == '.':

for num in map(str, range(1, 10)):

if is\_valid(num, row, col):

board[row][col] = num

if backtrack():

return True

board[row][col] = '.'

return False

return True

backtrack()

return board

input\_board = [["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]]

output\_board = solve\_sudoku(input\_board)

print(output\_board)

63

Input: candidates = [2,3,6,7], target = 7

Output: [[2,2,3],[7]]

Explanation:

2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.

7 is a candidate, and 7 = 7.

These are the only two combinations.

Input: candidates = [2,3,5], target = 8

Output: [[2,2,2,2],[2,3,3],[3,5]]

Input: candidates = [2], target = 1

Output: []

def combination\_sum(candidates, target):

def backtrack(remaining, combo, start):

if remaining == 0:

result.append(list(combo))

return

elif remaining < 0:

return

for i in range(start, len(candidates)):

combo.append(candidates[i])

backtrack(remaining - candidates[i], combo, i) # Not i + 1 because we can reuse the same elements

combo.pop()

result = []

backtrack(target, [], 0)

return result

print(combination\_sum([2, 3, 6, 7], 7))

print(combination\_sum([2, 3, 5], 8))

print(combination\_sum([2], 1))

64.

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8

Output:

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

]

Example 2:

Input: candidates = [2,5,2,1,2], target = 5

Output:

[

[1,2,2],

[5]

]

def combination\_sum2(candidates, target):

def backtrack(start, path, target):

if target == 0:

result.append(path)

return

for i in range(start, len(candidates)):

if i > start and candidates[i] == candidates[i - 1]:

continue

if candidates[i] > target:

break

backtrack(i + 1, path + [candidates[i]], target - candidates[i])

candidates.sort()

result = []

backtrack(0, [], target)

return result

print(combination\_sum2([10, 1, 2, 7, 6, 1, 5], 8))

print(combination\_sum2([2, 5, 2, 1, 2], 5))

65.

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Example 2:

Input: nums = [0,1]

Output: [[0,1],[1,0]]

Example 3:

Input: nums = [1]

Output: [[1]]

def permute(nums):

def backtrack(start):

if start == len(nums):

result.append(nums[:])

return

for i in range(start, len(nums)):

nums[start], nums[i] = nums[i], nums[start]

backtrack(start + 1)

nums[start], nums[i] = nums[i], nums[start]

result = []

backtrack(0)

return result

print(permute([1, 2, 3]))

66.

Input: nums = [1,1,2]

Output:

[[1,1,2],

[1,2,1],

[2,1,1]]

Example 2:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

def permute\_unique(nums):

def backtrack(start):

if start == len(nums):

result.append(nums[:])

return

seen = set()

for i in range(start, len(nums)):

if nums[i] not in seen:

seen.add(nums[i])

nums[start], nums[i] = nums[i], nums[start]

backtrack(start + 1)

nums[start], nums[i] = nums[i], nums[start]

nums.sort()

result = []

backtrack(0)

return result

print(permute\_unique([1, 1, 2]))

print(permute\_unique([1, 2, 3]))

67.

import sys

def prims\_algorithm(graph):

num\_vertices = len(graph)

selected\_nodes = [False] \* num\_vertices

selected\_nodes[0] = True

edges = []

for \_ in range(num\_vertices - 1):

minimum = sys.maxsize

x = 0

y = 0

for i in range(num\_vertices):

if selected\_nodes[i]:

for j in range(num\_vertices):

if not selected\_nodes[j] and graph[i][j]:

if minimum > graph[i][j]:

minimum = graph[i][j]

x = i

y = j

edges.append((x, y, minimum))

selected\_nodes[y] = True

return edges

graph = [

[0, 2, 0, 6, 0],

[2, 0, 3, 8, 5],

[0, 3, 0, 0, 7],

[6, 8, 0, 0, 9],

[0, 5, 7, 9, 0]

]

mst = prims\_algorithm(graph)

print("Edges in the Minimum Spanning Tree:")

for edge in mst:

print(f"{edge[0]} - {edge[1]} with weight {edge[2]}")

68.

Input: [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

Output:19

krushkals

class DisjointSet:

def \_\_init\_\_(self, n):

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u])

return self.parent[u]

def union(self, u, v):

root\_u = self.find(u)

root\_v = self.find(v)

if root\_u != root\_v:

if self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

def kruskal(n, edges):

edges.sort(key=lambda x: x[2])

ds = DisjointSet(n)

mst = []

total\_cost = 0

for u, v, weight in edges:

if ds.find(u) != ds.find(v):

ds.union(u, v)

mst.append((u, v, weight))

total\_cost += weight

return mst, total\_cost

edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

n = 4

mst, cost = kruskal(n, edges)

print("Edges in the Minimum Spanning Tree:", mst)

print("Total cost of the Minimum Spanning Tree:", cost)

69.

boruvkas algorithm.

Input:[( 0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

Output:

Edge 0 - 3 with weight 5 included in MST

Edge 0 - 1 with weight 10 included in MST

Edge 2 - 3 with weight 4 included in MST

Total weight of MST is 19

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

self.graph.append([u, v, w])

def find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

def union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

def boruvka\_mst(self):

parent = []

rank = []

cheapest = []

num\_trees = self.V

mst\_weight = 0

for node in range(self.V):

parent.append(node)

rank.append(0)

cheapest = [-1] \* len(self.graph)

while num\_trees > 1:

for i in range(len(self.graph)):

u, v, w = self.graph[i]

set\_u = self.find(parent, u)

set\_v = self.find(parent, v)

if set\_u != set\_v:

if cheapest[set\_u] == -1 or cheapest[set\_u][2] > w:

cheapest[set\_u] = [u, v, w]

if cheapest[set\_v] == -1 or cheapest[set\_v][2] > w:

cheapest[set\_v] = [u, v, w]

for node in range(self.V):

if cheapest[node] != -1:

u, v, w = cheapest[node]

set\_u = self.find(parent, u)

set\_v = self.find(parent, v)

if set\_u != set\_v:

mst\_weight += w

self.union(parent, rank, set\_u, set\_v)

print(f"Edge {u} - {v} with weight {w} included in MST")

cheapest = [-1] \* len(self.graph)

num\_trees -= 1

print(f"Total weight of MST is {mst\_weight}")

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 4)

g.boruvka\_mst()