1.Word break problem

Program:

def wordBreak(s, wordDict):

word\_set = set(wordDict)

dp = [False] \* (len(s) + 1)

dp[0] = True

for i in range(1, len(s) + 1):

for j in range(i):

if dp[j] and s[j:i] in word\_set:

dp[i] = True

break

return dp[-1]

s = "leetcode"

wordDict = ["leet", "code"]

print(wordBreak(s, wordDict))

s = "applepenapple"

wordDict = ["apple", "pen"]

print(wordBreak(s, wordDict))

2.word trap problem

Program:

def wordTrap(s, wordDict):

word\_set = set(wordDict)

memo = {}

def backtrack(start, used\_words):

if start == len(s):

return True

if start in memo:

return memo[start]

for end in range(start + 1, len(s) + 1):

word = s[start:end]

if word in word\_set and word not in used\_words:

used\_words.add(word)

if backtrack(end, used\_words):

memo[start] = True

return True

used\_words.remove(word)

memo[start] = False

return False

return backtrack(0, set())

s = "leetcode"

wordDict = ["leet", "code"]

print(wordTrap(s, wordDict)) # Output: True

s = "applepenapple"

wordDict = ["apple", "pen"]

print(wordTrap(s, wordDict)) # Output: False (because "apple" would be used twice)

3.optimal binary search tree

Program:

def optimal\_bst(keys, freq):

n = len(keys)

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

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

for i in range(n):

dp[i][i] = freq[i]

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

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

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

j = i + length - 1

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

total\_freq = sum(freq[i:j + 1])

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

c = (dp[i][r - 1] if r > i else 0) + (dp[r + 1][j] if r < j else 0) + total\_freq

if c < dp[i][j]:

dp[i][j] = c

cost[i][j] = r

return dp[0][n - 1], dp, cost

# Example usage

keys = [10, 12, 20]

freq = [34, 8, 50]

min\_cost, dp\_table, root\_table = optimal\_bst(keys, freq)

print("Minimum cost of OBST:", min\_cost)

print("DP Table:")

for row in dp\_table:

print(row)

print("Root Table:")

for row in root\_table:

print(row)

4.floyd algotithm

Program:

def floyd\_warshall(graph):

n = len(graph)

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

# Initialize the distance array with given graph weights

for i in range(n):

for j in range(n):

if i == j:

dist[i][j] = 0

elif graph[i][j] != 0:

dist[i][j] = graph[i][j]

# Floyd-Warshall algorithm

for k in range(n):

for i in range(n):

for j in range(n):

if dist[i][j] > dist[i][k] + dist[k][j]:

dist[i][j] = dist[i][k] + dist[k][j]

return dist

# Example usage

graph = [

[0, 3, float('inf'), 5],

[2, 0, float('inf'), 4],

[float('inf'), 1, 0, float('inf')],

[float('inf'), float('inf'), 2, 0]

]

distances = floyd\_warshall(graph)

print("Shortest distances between every pair of vertices:")

for row in distances:

print(row)