Backtracking

1. N-Queens Problem

Detailed Problem Explanation

The N-Queens problem is a classic algorithmic problem that asks for the placement of N chess queens on an N×N chessboard such that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal. The objective is to find all distinct solutions to this problem.

Input and Output Example

```
# Example Input
N = 4

# Example Output
[
    [".Q..",  # Solution 1
     "...Q",
     "Q...",
     "..Q."],

    ["..Q.",  # Solution 2
     "Q...",
     "...Q",
     ".Q.."]
]
```

Core Logic Behind the Solution

The solution utilizes backtracking to explore all possible configurations of placing queens on the board. Here's how it works:

1. **Recursive Function**: We define a recursive function that attempts to place queens row by row.
2. **Base Case**: If the row index equals N, a valid configuration has been found, and we add it to our results.
3. **Check Validity**: For each column in the current row, we check if placing a queen there is valid by ensuring that no other queens can attack it (checking the same column and both diagonals).

4. **Backtrack**: If placing a queen leads to a solution, we move to the next row; otherwise, we remove the queen and try the next column.

This process continues until all possible configurations have been explored.

Optimized Code in Python

```
def solveNQueens(N):
    def backtrack(row, cols, diag1, diag2, board):
        if row == N:
            result.append([''.join(r) for r in board])
            return

        for col in range(N):
            if col in cols or (row - col) in diag1 or (row + col) in diag2:
                continue

            # Place the queen
            board[row][col] = 'Q'
            cols.add(col)
            diag1.add(row - col)
            diag2.add(row + col)

            # Move to the next row
            backtrack(row + 1, cols, diag1, diag2, board)

            # Remove the queen
            board[row][col] = '.'
            cols.remove(col)
            diag1.remove(row - col)
            diag2.remove(row + col)

    result = []
    board = [['.' for _ in range(N)] for _ in range(N)]
    backtrack(0, set(), set(), set(), board)
    return result

# Example usage
print(solveNQueens(4))
```

---

2. Sudoku Solver

Detailed Problem Explanation

The Sudoku Solver problem requires filling in a partially filled 9x9 Sudoku board with digits from 1 to 9. The Sudoku board is valid if each row, each column, and each of the nine 3x3 sub-boxes contain all of the digits from 1 to 9 without repetition. The goal is to fill the board such that it satisfies these conditions.

Input and Output Example

```
# Example 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"]
]

# Example 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","2","3","5","8","7","4"],
    ["2","8","7","4","1","9","6","3","5"],
    ["3","4","5","6","8","7","2","9","1"]
]
```

Core Logic Behind the Solution

The solution uses a backtracking algorithm to fill the Sudoku board:

1. **Find Empty Cell**: Iterate through the board to find an empty cell (represented by '.').
2. **Try Valid Numbers**: For each empty cell, attempt to place digits from 1 to 9.
3. **Check Validity**: Before placing a digit, check if it's valid by ensuring that the digit does not already exist in the current row, column, or 3x3 sub-box.
4. **Recursion**: If a digit can be placed, recursively attempt to fill the next empty cell. If successful, return true.

5. **Backtrack**: If placing a digit does not lead to a solution, reset the cell and try the next digit.

This continues until the entire board is filled or all possibilities are exhausted.

Optimized Code in Python

```python
def solveSudoku(board):
    def is_valid(board, row, col, num):
        for x in range(9):
            if board[row][x] == num or board[x][col] == num:
                return False
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
                if board[i + start_row][j + start_col] == num:
                    return False
        return True

    def backtrack(board):
        for i in range(9):
            for j in range(9):
                if board[i][j] == '.':
                    for num in map(str, range(1, 10)):
                        if is_valid(board, i, j, num):
                            board[i][j] = num
                            if backtrack(board):
                                return True
                            board[i][j] = '.'
                    return False
        return True

    backtrack(board)

# Example usage
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"]
]

solveSudoku(board)
print(board)
```

---

3. Permutations of a String

Detailed Problem Explanation

The problem of generating permutations of a string involves finding all possible arrangements of the characters in the string. Given a string, the goal is to produce a list of all unique permutations.

Input and Output Example

```
# Example Input
s = "abc"

# Example Output
["abc", "acb", "bac", "bca", "cab", "cba"]
```

Core Logic Behind the Solution

The solution employs a backtracking approach:

1. **Recursive Function**: Define a recursive function that takes the current permutation and tracks which characters have been used.
2. **Base Case**: When the length of the current permutation matches the original string's length, add it to the results.
3. **Explore Options**: For each character in the string, if it hasn't been used, add it to the current permutation and mark it as used.
4. **Backtrack**: After exploring one path, remove the character and mark it as unused to explore other permutations.

This approach generates all possible permutations by systematically exploring all character placements.

Optimized Code in Python

```python
def permute(s):
    def backtrack(start=0):
        if start == len(s):
            result.append(''.join(s))
            return
        for i in range(start, len(s)):
            s[start], s[i] = s[i], s[start]  # Swap
            backtrack(start + 1)
            s[start], s[i] = s[i], s[start]  # Backtrack

    result = []
    s = list(s)  # Convert string to list for mutability
    backtrack()
    return result

# Example usage
print(permute("abc"))
```

---

4. Combination Sum

Detailed Problem Explanation

The Combination Sum problem involves finding all unique combinations of numbers that sum up to a target value. Given a list of candidate numbers and a target number, each number can be used multiple times.

Input and Output Example

```python
# Example Input
candidates = [2, 3, 6, 7]
target = 7

# Example Output
[[2, 2, 3], [7]]
```

Core Logic Behind the Solution

This problem can also be solved using backtracking:

1. **Recursive Function**: Define a recursive function that takes the current combination and the remaining target value.
2. **Base Case**: If the remaining target is zero, add the current combination to the results.

3. **Explore Options**: Iterate through the candidate numbers, and for each candidate, add it to the current combination and subtract it from the target.
4. **Backtrack**: Continue exploring until the target is met or exceeded, then backtrack to explore other combinations.

This allows exploration of all possible combinations that can yield the target sum.

Optimized Code in Python

```
def combinationSum(candidates, target):
    def backtrack(start, path, target):
        if target == 0:
            result.append(path)
            return
        for i in range(start, len(candidates)):
            if candidates[i] > target:
                continue
            backtrack(i, path + [candidates[i]], target - candidates[i])  # Not incrementing
start allows reuse

    result = []
    backtrack(0, [], target)
    return result

# Example usage
print(combinationSum([2, 3, 6, 7], 7))
```

---

5. Word Search

Detailed Problem Explanation

The Word Search problem requires determining if a given word can be formed by a sequence of adjacent letters in a 2D grid. Letters must be connected horizontally or vertically, and the same letter cell cannot be used more than once in a single word formation.

Input and Output Example

```
# Example Input
board = [
    ["A","B","C","E"],
    ["S","F","C","S"],
    ["A","D","E","E"]
]
word = "ABCCED"

# Example Output
True
```

Core Logic Behind the Solution

The solution uses Depth-First Search (DFS) to explore the grid:

1. **DFS Function**: Define a DFS function that explores adjacent cells for matching letters.
2. **Base Cases**: If the entire word is found, return true. If the letter does not match or out of bounds, return false.
3. **Mark Visited Cells**: Temporarily mark cells as visited to prevent revisiting.
4. **Explore Adjacent Cells**: Recursively call DFS for each of the four adjacent cells.
5. **Backtrack**: Unmark the visited cell after exploring all paths.

This approach effectively searches for the word while ensuring cells are not reused within the same path.

Optimized Code in Python

```
def exist(board, word):
    rows, cols = len(board), len(board[0])

    def dfs(r, c, index

):
        if index == len(word):
            return True
        if r < 0 or r >= rows or c < 0 or c >= cols or board[r][c] != word[index]:
            return False

        temp = board[r][c]
        board[r][c] = "#"  # Mark as visited

        found = (dfs(r + 1, c, index + 1) or
                 dfs(r - 1, c, index + 1) or
                 dfs(r, c + 1, index + 1) or
                 dfs(r, c - 1, index + 1))

        board[r][c] = temp  # Unmark
        return found

    for i in range(rows):
        for j in range(cols):
            if dfs(i, j, 0):
                return True
    return False

# Example usage
board = [
    ["A","B","C","E"],
    ["S","F","C","S"],
    ["A","D","E","E"]
]
print(exist(board, "ABCCED"))
```

---

6. Subsets (Power Set)

Detailed Problem Explanation

The Subsets problem involves generating all possible subsets of a given set. This includes the empty subset and the set itself, resulting in a total of (2^n) subsets for a set of size (n).

Input and Output Example

```
# Example Input
nums = [1, 2, 3]

# Example Output
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
```

Core Logic Behind the Solution

The solution uses backtracking to explore all subsets:

1. **Recursive Function**: Define a recursive function that builds the current subset.
2. **Base Case**: If the current index reaches the end of the list, add the current subset to the results.
3. **Include or Exclude**: For each number, either include it in the current subset or exclude it and proceed to the next index.
4. **Backtrack**: This process continues until all possibilities are explored.

This generates all combinations of numbers, resulting in the power set.

Optimized Code in Python

```python
def subsets(nums):
    def backtrack(start, path):
        result.append(path)
        for i in range(start, len(nums)):
            backtrack(i + 1, path + [nums[i]])

    result = []
    backtrack(0, [])
    return result

# Example usage
print(subsets([1, 2, 3]))
```

---

7. Generate Parentheses

Detailed Problem Explanation

The Generate Parentheses problem involves generating all combinations of well-formed parentheses given a number (n), which indicates the number of pairs of parentheses.

Input and Output Example

```
# Example Input
n = 3

# Example Output
["((()))", "(()())", "(())()", "()(())", "()()()"]
```

Core Logic Behind the Solution

The solution uses backtracking to build valid parentheses combinations:

1. **Recursive Function**: Define a recursive function that takes the current combination, counts of opened and closed parentheses.
2. **Base Case**: If the current combination's length equals (2n), add it to the results.
3. **Add Parentheses**: If the count of opened parentheses is less than (n), add an opened parenthesis. If the count of closed parentheses is less than opened, add a closed parenthesis.
4. **Backtrack**: Continue exploring until all combinations are generated.

This ensures that only valid combinations are formed.

Optimized Code in Python

```
def generateParenthesis(n):
    def backtrack(current, open_count, close_count):
        if len(current) == 2 * n:
            result.append(current)
            return
        if open_count < n:
            backtrack(current + "(", open_count + 1, close_count)
        if close_count < open_count:
            backtrack(current + ")", open_count, close_count + 1)

    result = []
    backtrack("", 0, 0)
    return result

# Example usage
print(generateParenthesis(3))
```

8. Partition Equal Subset Sum

Detailed Problem Explanation

The Partition Equal Subset Sum problem involves determining whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is equal. This is a classic subset sum problem.

Input and Output Example

```
# Example Input
nums = [1, 5, 11, 5]

# Example Output
True  # [1, 5, 5] and [11] both sum to 11
```

Core Logic Behind the Solution

The solution employs dynamic programming to solve the problem:

1. **Calculate Total Sum**: First, check if the total sum of the array is odd. If it is, return false since it can't be partitioned into two equal subsets.
2. **Target Sum**: Calculate the target sum as half of the total sum.
3. **DP Array**: Use a boolean DP array where `dp[j]` indicates if a subset with sum (j) can be formed.
4. **Update DP Array**: Iterate through each number and update the DP array from the back (to avoid using the same number multiple times).
5. **Final Check**: Return the value of `dp[target]` to determine if the partition is possible.

Optimized Code in Python

```python
def canPartition(nums):
    total_sum = sum(nums)
    if total_sum % 2 != 0:
        return False
    target = total_sum // 2
    dp = [False] * (target + 1)
    dp[0] = True  # Zero sum is always possible

    for num in nums:
        for j in range(target, num - 1, -1):
            dp[j] = dp[j] or dp[j - num]

    return dp[target]

# Example usage
print(canPartition([1, 5, 11, 5]))
```

---

9. M Coloring Problem

Detailed Problem Explanation

The M Coloring Problem involves assigning colors to vertices of a graph such that no two adjacent vertices share the same color. The goal is to determine if it's possible to color the graph using at most (m) colors.

Input and Output Example

```python
# Example Input
graph = [
    [0, 1, 1, 1],
    [1, 0, 0, 1],
    [1, 0, 0, 1],
    [1, 1, 1, 0]
]
m = 3

# Example Output
True  # Graph can be colored with 3 colors
```

Core Logic Behind the Solution

The solution employs backtracking to explore color assignments:

1. **Recursive Function**: Define a function that attempts to color each vertex.
2. **Base Case**: If all vertices are colored, return true.
3. **Check Validity**: For each vertex, try each color. If a color can be assigned (i.e., no adjacent vertex has the same color), recursively attempt to color the next vertex.
4. **Backtrack**: If a color assignment leads to a solution, continue; otherwise, reset and try the next color.

This process continues until a valid coloring is found or all options are exhausted.

Optimized Code in Python

```python
def isSafe(graph, node, color, c):
    for i in range(len(graph)):
        if graph[node][i] == 1 and color[i] == c:
            return False
    return True

def mColoringUtil(graph, m, color, node):
    if node == len(graph):
        return True

    for c in range(1, m + 1):
        if isSafe(graph, node, color, c):
            color[node] = c
            if mColoringUtil(graph, m, color, node + 1):
                return True
            color[node] = 0  # Backtrack

    return False

def mColoring(graph, m):
    color = [0] * len(graph)
    return mColoringUtil(graph, m, color, 0)

# Example usage
graph = [
    [0, 1, 1, 1],
    [1, 0, 0, 1],
    [1, 0, 0, 1],
    [1, 1, 1, 0]
]
print(mColoring(graph, 3))
```

10. Letter Combinations of a Phone Number

Detailed Problem Explanation

The Letter Combinations of a Phone Number problem requires generating all possible letter combinations that a given digit string could represent on a phone keypad. Each digit maps to a set of letters, and the goal is to generate all combinations of these letters.

Input and Output Example

```
# Example Input
digits = "23"

# Example Output
["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

Core Logic Behind the Solution

The solution uses backtracking to generate combinations:

1. **Mapping Digits to Letters**: Define a mapping of digits to their corresponding letters.
2. **Recursive Function**: Define a recursive function that builds the current combination based on the current index of the digits string.
3. **Base Case**: If the current combination length equals the length of the

   input digits, add it to the results.
4. **Explore Options**: For each digit, iterate through its mapped letters and recursively add to the current combination.
5. **Backtrack**: Continue building until all combinations are generated.

This effectively explores all possible combinations of letters corresponding to the input digits.

Optimized Code in Python

```python
def letterCombinations(digits):
    if not digits:
        return []

    phone_map = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }

    def backtrack(index, path):
        if index == len(digits):
            result.append(''.join(path))
            return
        letters = phone_map[digits[index]]
        for letter in letters:
            path.append(letter)
            backtrack(index + 1, path)
            path.pop()  # Backtrack

    result = []
    backtrack(0, [])
    return result

# Example usage
print(letterCombinations("23"))
```

Gready

---

1. Activity Selection Problem

Detailed Problem Explanation

The Activity Selection problem involves selecting the maximum number of activities that don't overlap in time. Each activity has a start time and an end time, and the goal is to choose activities such that the selected activities do not overlap and the total number of activities is maximized.

Input and Output Example

```
# Example Input
activities = [(1, 3), (2, 5), (4, 6), (5, 7), (8, 9)]

# Example Output
[(1, 3), (4, 6), (8, 9)]
```

Core Logic Behind the Solution

The solution uses a greedy approach:

1. **Sort Activities**: First, sort the activities based on their finish times.
2. **Select Activities**: Initialize the first activity as selected. For each subsequent activity, check if its start time is greater than or equal to the finish time of the last selected activity.
3. **Add to Selected List**: If the condition is met, select the activity and update the last selected activity's finish time.

This ensures that we always select the earliest finishing activity possible, maximizing the number of selected activities.

Optimized Code in Python

```
def activity_selection(activities):
    # Sort activities by finish time
    activities.sort(key=lambda x: x[1])

    selected = [activities[0]]  # Select the first activity
    last_finish_time = activities[0][1]

    for i in range(1, len(activities)):
        if activities[i][0] >= last_finish_time:  # If the start time is >= last finish time
            selected.append(activities[i])
            last_finish_time = activities[i][1]  # Update last finish time

    return selected

# Example usage
print(activity_selection([(1, 3), (2, 5), (4, 6), (5, 7), (8, 9)]))
```

---

2. Coin Change Problem (Minimum Coins)

Detailed Problem Explanation

The Minimum Coins problem involves finding the minimum number of coins needed to make a given amount from a set of coin denominations. The goal is to minimize the total number of coins used to achieve the target amount.

Input and Output Example

```
# Example Input
coins = [1, 2, 5]
amount = 11

# Example Output
3  # (11 = 5 + 5 + 1)
```

Core Logic Behind the Solution

The solution employs dynamic programming:

1. **DP Array**: Create a DP array where `dp[i]` represents the minimum number of coins needed to make amount `i`. Initialize the array with a large value (infinity) and set `dp[0]` to 0.
2. **Update DP Array**: For each coin, iterate through the DP array and update `dp[i]` as the minimum of its current value or `dp[i - coin] + 1`.
3. **Return Result**: The value at `dp[amount]` gives the minimum coins needed. If it remains infinity, return -1 to indicate that it's not possible to make the amount.

Optimized Code in Python

```
def coinChange(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # 0 coins needed to make amount 0

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

# Example usage
print(coinChange([1, 2, 5], 11))
```

---

3. Fractional Knapsack Problem

Detailed Problem Explanation

The Fractional Knapsack problem involves maximizing the value of items placed in a knapsack of a certain capacity. Unlike the 0/1 Knapsack problem, we can take fractions of an item. Each item has a weight and a value, and we aim to maximize the total value in the knapsack.

Input and Output Example

```
# Example Input
weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50

# Example Output
240.0  # (Take 20 weight of item 2 and all of item 3)
```

Core Logic Behind the Solution

The solution uses a greedy strategy:

1. **Calculate Value-to-Weight Ratio**: For each item, calculate its value-to-weight ratio and sort the items based on this ratio in descending order.
2. **Select Items**: Initialize total value as 0 and iterate through the sorted items. For each item, if it can fit entirely in the knapsack, add its full value; otherwise, take the fraction of the remaining capacity.
3. **Return Total Value**: Continue until the knapsack is full or all items are processed.

Optimized Code in Python

```python
def fractional_knapsack(weights, values, capacity):
    items = sorted(zip(weights, values), key=lambda x: x[1] / x[0], reverse=True)
    total_value = 0.0

    for weight, value in items:
        if capacity == 0:
            break
        if weight <= capacity:
            total_value += value
            capacity -= weight
        else:
            total_value += value * (capacity / weight)
            capacity = 0  # Knapsack is full

    return total_value

# Example usage
print(fractional_knapsack([10, 20, 30], [60, 100, 120], 50))
```

---

4. Job Sequencing Problem

Detailed Problem Explanation

The Job Sequencing problem involves scheduling jobs with deadlines to maximize the total profit. Each job has a deadline and a profit associated with it. The goal is to select jobs that maximize the total profit while meeting their deadlines.

Input and Output Example

```python
# Example Input
jobs = [(1, 20), (2, 15), (3, 10), (4, 5), (5, 1)]  # (profit, deadline)

# Example Output
35  # (Select job 1 and job 2)
```

Core Logic Behind the Solution

The solution uses a greedy approach:

1. **Sort Jobs**: Sort jobs by profit in descending order.
2. **Track Slots**: Create an array to track free time slots up to the maximum deadline.

3. **Select Jobs**: Iterate through the sorted jobs and try to place each job in its latest available slot before its deadline.
4. **Calculate Total Profit**: Sum the profits of the scheduled jobs.

Optimized Code in Python

```python
def job_sequencing(jobs):
    jobs.sort(key=lambda x: x[1], reverse=True)  # Sort by profit
    max_deadline = max(job[0] for job in jobs)
    slots = [-1] * max_deadline  # Initialize slots
    total_profit = 0

    for profit, deadline in jobs:
        for j in range(min(deadline, max_deadline) - 1, -1, -1):
            if slots[j] == -1:  # If slot is free
                slots[j] = profit
                total_profit += profit
                break

    return total_profit

# Example usage
print(job_sequencing([(20, 1), (15, 2), (10, 3), (5, 4), (1, 5)]))
```

---

5. Huffman Coding

Detailed Problem Explanation

Huffman Coding is an algorithm for lossless data compression. It assigns variable-length codes to input characters based on their frequencies, with shorter codes assigned to more frequent characters. The goal is to minimize the total number of bits used for encoding.

Input and Output Example

```
# Example Input
frequencies = {'a': 5, 'b': 9, 'c': 12, 'd': 13, 'e': 16, 'f': 45}

# Example Output
{
    'a': '1100',
    'b': '1101',
    'c': '100',
    'd': '101',
    'e': '111',
    'f': '0'
}
```

Core Logic Behind the Solution

The solution uses a priority queue:

1. **Build a Min-Heap**: Create a min-heap to store characters and their frequencies.
2. **Combine Nodes**: Continuously extract the two nodes with the smallest frequencies, create a new internal node with their sum, and insert it back into the heap.
3. **Generate Codes**: Once the heap contains a single node, traverse the tree to assign codes based on the path taken (left for '0' and right for '1').

This results in optimal codes based on the frequencies.

Optimized Code in Python

```
import heapq
from collections import defaultdict

def huffman_coding(frequencies):
    heap = [[weight, [char, '']] for char, weight in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1

:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    return sorted(heap[0][1:], key=lambda p: (len(p[-1]), p))

# Example usage
frequencies = {'a': 5, 'b': 9, 'c': 12, 'd': 13, 'e': 16, 'f': 45}
print(huffman_coding(frequencies))
```

---

6. Minimum Spanning Tree (Prim's and Kruskal's)

Detailed Problem Explanation

The Minimum Spanning Tree (MST) problem involves finding a subset of edges that connects all vertices in a graph while minimizing the total edge weight. Both Prim's and Kruskal's algorithms can be used to solve this problem.

Input and Output Example

```
# Example Input (for Prim's)
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'C': 1, 'D': 5},
    'C': {'A': 3, 'B': 1, 'D': 4},
    'D': {'B': 5, 'C': 4}
}

# Example Output (for Prim's)
[(A, B, 1), (B, C, 1), (C, D, 4)]
```

Core Logic Behind the Solution (Prim's)

1. **Initialize**: Start from any vertex and keep track of visited vertices and the edges connecting to unvisited vertices.
2. **Select Minimum Edge**: Continuously select the edge with the minimum weight that connects a visited vertex to an unvisited vertex.
3. **Repeat**: Add the selected edge to the MST and mark the newly connected vertex as visited. Repeat until all vertices are visited.

Optimized Code in Python (Prim's)

```python
import heapq

def prim(graph):
    mst_edges = []
    total_cost = 0
    start_vertex = next(iter(graph))
    visited = set([start_vertex])
    edges = [(cost, start_vertex, to) for to, cost in graph[start_vertex].items()]
    heapq.heapify(edges)

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst_edges.append((frm, to, cost))
            total_cost += cost
            for to_next, cost_next in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (cost_next, to, to_next))

    return mst_edges

# Example usage
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'C': 1, 'D': 5},
    'C': {'A': 3, 'B': 1, 'D': 4},
    'D': {'B': 5, 'C': 4}
}
print(prim(graph))
```

---

7. Earliest Finish Time

Detailed Problem Explanation

The Earliest Finish Time problem involves finding the earliest time at which a set of activities can be completed, given their start and finish times. Each activity has a start time and a finish time, and activities cannot overlap.

Input and Output Example

```
# Example Input
activities = [(1, 3), (2, 5), (4, 6), (5, 7), (8, 9)]

# Example Output
9  # The last finish time
```

Core Logic Behind the Solution

The solution uses a greedy approach:

1. **Sort Activities**: Sort activities by their finish times.
2. **Track Earliest Finish**: Initialize the earliest finish time with the finish time of the first activity. Iterate through the sorted list and update the earliest finish time based on the activities that can be completed without overlapping.
3. **Return Final Finish Time**: The last activity's finish time will be the earliest time all activities can be completed.

Optimized Code in Python

```
def earliest_finish_time(activities):
    activities.sort(key=lambda x: x[1])  # Sort by finish time
    earliest_finish = 0

    for start, finish in activities:
        if start >= earliest_finish:
            earliest_finish = finish  # Update if the activity can be scheduled

    return earliest_finish

# Example usage
print(earliest_finish_time([(1, 3), (2, 5), (4, 6), (5, 7), (8, 9)]))
```

---

8. Maximize the Number of Activities

Detailed Problem Explanation

The problem of maximizing the number of activities involves selecting the maximum number of activities that can be performed within a given timeframe. Each activity has a start time and an end time, and the goal is to select activities such that they don't overlap.

Input and Output Example

```
# Example Input
activities = [(1, 3), (2, 5), (4, 6), (5, 7), (8, 9)]

# Example Output
[(1, 3), (4, 6), (5, 7), (8, 9)]
```

Core Logic Behind the Solution

The solution uses a greedy strategy:

1. **Sort Activities**: Sort the activities based on their end times.
2. **Select Activities**: Iterate through the sorted list and select the activity if its start time is greater than or equal to the finish time of the last selected activity.
3. **Return Selected Activities**: Collect and return the selected activities.

Optimized Code in Python

```
def maximize_activities(activities):
    activities.sort(key=lambda x: x[1])  # Sort by finish time
    selected_activities = [activities[0]]  # Select the first activity
    last_finish_time = activities[0][1]

    for start, finish in activities[1:]:
        if start >= last_finish_time:
            selected_activities.append((start, finish))
            last_finish_time = finish  # Update last finish time

    return selected_activities

# Example usage
print(maximize_activities([(1, 3), (2, 5), (4, 6), (5, 7), (8, 9)]))
```

---

9. Climbing Stairs with Minimum Cost

Detailed Problem Explanation

The Climbing Stairs with Minimum Cost problem involves finding the minimum cost to reach the top of a staircase where each step has a cost associated with it. You can take either one or two steps at a time.

Input and Output Example

```
# Example Input
cost = [10, 15, 20]

# Example Output
15  # (Minimum cost to reach the top: take step 1 and step 2)
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i]` represents the minimum cost to reach step `i`. Initialize the first two values based on the cost of the first two steps.
2. **Update DP Array**: For each step from the third onward, calculate the minimum cost as the cost of the current step plus the minimum of the costs of the previous two steps.
3. **Return Result**: The result will be the minimum cost to reach the top, which is the minimum of the last two entries in the DP array.

Optimized Code in Python

```
def min_cost_climbing_stairs(cost):
    n = len(cost)
    if n == 0:
        return 0

    dp = [0] * (n + 1)
    dp[0] = 0
    dp[1] = cost[0]

    for i in range(2, n + 1):
        dp[i] = cost[i - 1] + min(dp[i - 1], dp[i - 2])

    return min(dp[n], dp[n - 1])

# Example usage
print(min_cost_climbing_stairs([10, 15, 20]))
```

---

10. Largest Number

Detailed Problem Explanation

The Largest Number problem involves arranging a list of non-negative integers in such a way that they form the largest possible number when concatenated.

Input and Output Example

```
# Example Input
nums = [10, 2]

# Example Output
"210"
```

Core Logic Behind the Solution

The solution uses custom sorting:

1. **Custom Comparator**: Create a comparator that determines the order of two numbers by comparing the two possible concatenated results.
2. **Sort the List**: Sort the list of numbers based on this comparator.
3. **Concatenate Sorted Numbers**: Join the sorted list to form the final result, ensuring to handle leading zeros.

Optimized Code in Python

```
from functools import cmp_to_key

def largest_number(nums):
    def compare(x, y):
        return (y + x) > (x + y)  # Compare concatenated results

    nums = list(map(str, nums))
    nums.sort(key=cmp_to_key(compare))
    result = ''.join(nums)

    return result if result[0] != '0' else '0'  # Handle case for all zeros

# Example usage
print(largest_number([10, 2]))
```

Dynamic Programming

---

1. Fibonacci Number

Detailed Problem Explanation

The Fibonacci Number problem involves calculating the nth Fibonacci number, where each number is the sum of the two preceding ones, typically starting with 0 and 1. The sequence starts: 0, 1, 1, 2, 3, 5, 8, 13, ...

Input and Output Example

```
# Example Input
n = 5

# Example Output
5  # (The Fibonacci sequence is: 0, 1, 1, 2, 3, 5)
```

Core Logic Behind the Solution

The solution can be approached in several ways, including:

1. **Recursive**: A direct recursive function calls itself to calculate Fibonacci numbers.
2. **Dynamic Programming**: Use an array to store previously calculated Fibonacci numbers.
3. **Optimized Space**: Maintain only the last two Fibonacci numbers instead of the entire array.

The dynamic programming approach is efficient for larger values of `n`.

Optimized Code in Python

```
def fibonacci(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b

# Example usage
print(fibonacci(5))
```

---

2. Longest Increasing Subsequence

Detailed Problem Explanation

The Longest Increasing Subsequence (LIS) problem involves finding the length of the longest subsequence of a given sequence in which the elements are in sorted order. A subsequence is derived by deleting some or none of the elements without changing the order of the remaining elements.

Input and Output Example

```
# Example Input
nums = [10, 9, 2, 5, 3, 7, 101, 18]

# Example Output
4  # (The longest increasing subsequence is [2, 3, 7, 101])
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i]` represents the length of the longest increasing subsequence that ends at index `i`.
2. **Initialize**: Set each `dp[i]` to 1 since the minimum length of an increasing subsequence is 1 (the element itself).
3. **Update DP Array**: For each pair of indices, update `dp[i]` by checking all previous indices to find valid subsequences and take the maximum length.
4. **Return Result**: The final result is the maximum value in the DP array.

Optimized Code in Python

```python
def length_of_LIS(nums):
    if not nums:
        return 0

    dp = [1] * len(nums)
    for i in range(len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

# Example usage
print(length_of_LIS([10, 9, 2, 5, 3, 7, 101, 18]))
```

---

3. 0/1 Knapsack Problem

Detailed Problem Explanation

The 0/1 Knapsack problem involves selecting items with given weights and values to maximize total value without exceeding a maximum weight limit. Each item can either be included in the knapsack or not.

Input and Output Example

```python
# Example Input
weights = [1, 2, 3]
values = [10, 15, 40]
capacity = 6

# Example Output
55  # (Take items 2 and 3)
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i][j]` represents the maximum value achievable with the first `i` items and a knapsack capacity of `j`.
2. **Initialize**: Set `dp[0][j]` to 0 for all `j`, as no items means zero value.

3. **Update DP Array**: Iterate through items and capacities to either include or exclude the current item and take the maximum value possible.
4. **Return Result**: The result is found in `dp[n][capacity]`.

Optimized Code in Python

```python
def knapsack(weights, values, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

# Example usage
print(knapsack([1, 2, 3], [10, 15, 40], 6))
```

---

4. Longest Common Subsequence

Detailed Problem Explanation

The Longest Common Subsequence (LCS) problem involves finding the longest subsequence present in two sequences. The subsequence does not need to be contiguous.

Input and Output Example

```python
# Example Input
s1 = "abcde"
s2 = "ace"

# Example Output
3  # (The longest common subsequence is "ace")
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i][j]` represents the length of the LCS of the first `i` characters of `s1` and the first `j` characters of `s2`.
2. **Initialize**: Set the first row and column to 0, as an empty string has no common subsequence with any string.
3. **Update DP Array**: Iterate through both strings, and if characters match, increment the count. Otherwise, take the maximum from the previous characters.
4. **Return Result**: The result is found in `dp[m][n]`, where `m` and `n` are the lengths of the two strings.

Optimized Code in Python

```python
def longest_common_subsequence(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example usage
print(longest_common_subsequence("abcde", "ace"))
```

---

5. Edit Distance

Detailed Problem Explanation

The Edit Distance problem (Levenshtein distance) involves finding the minimum number of operations required to convert one string into another. The operations are insertions, deletions, and substitutions.

Input and Output Example

```
# Example Input
s1 = "kitten"
s2 = "sitting"

# Example Output
3  # (Operations: Substitute 'k' with 's', substitute 'e' with 'i', and insert 'g')
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i][j]` represents the edit distance between the first `i` characters of `s1` and the first `j` characters of `s2`.
2. **Initialize**: Set the first row and column based on the cost of converting from empty strings to the respective strings.
3. **Update DP Array**: Iterate through both strings and update the DP array based on the minimum of the three operations.
4. **Return Result**: The result is found in `dp[len(s1)][len(s2)]`.

Optimized Code in Python

```
def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j  # All insertions
            elif j == 0:
                dp[i][j] = i  # All deletions
            elif s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]  # No operation needed
            else:
                dp[i][j] = 1 + min(dp[i - 1][j],    # Deletion
                                   dp[i][j - 1],    # Insertion
                                   dp[i - 1][j - 1])  # Substitution

    return dp[m][n]

# Example usage
print(edit_distance("kitten", "sitting"))
```

6. Coin Change Problem (Number of Ways)

Detailed Problem Explanation

The Coin Change problem (

Number of Ways) involves finding the total number of ways to make change for a given amount using a specified set of denominations.

Input and Output Example

```
# Example Input
coins = [1, 2, 5]
amount = 5

# Example Output
4  # (Ways: [5], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1])
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i]` represents the number of ways to make change for amount `i`.
2. **Initialize**: Set `dp[0] = 1`, as there is one way to make the amount 0 (by choosing no coins).
3. **Update DP Array**: For each coin, iterate through all amounts, updating the DP array based on previously computed values.
4. **Return Result**: The result is found in `dp[amount]`.

Optimized Code in Python

```python
def coin_change_ways(coins, amount):
    dp = [0] * (amount + 1)
    dp[0] = 1  # One way to make 0 amount

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] += dp[i - coin]

    return dp[amount]

# Example usage
print(coin_change_ways([1, 2, 5], 5))
```

---

7. Minimum Path Sum in a Grid

Detailed Problem Explanation

The Minimum Path Sum problem involves finding a path from the top left to the bottom right of a grid, minimizing the sum of the weights of the cells along the path. You can only move down or right.

Input and Output Example

```python
# Example Input
grid = [[1, 3, 1],
        [1, 5, 1],
        [4, 2, 1]]

# Example Output
7  # (The path is 1 → 3 → 1 → 1 → 1)
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i][j]` represents the minimum path sum to reach cell `(i, j)`.
2. **Initialize**: Set the first cell to the value of the grid and initialize the first row and column.
3. **Update DP Array**: For each cell, update the DP value based on the minimum of the values from the top and left cells.

4. **Return Result**: The result is found in `dp[m-1][n-1]`, where `m` and `n` are the dimensions of the grid.

Optimized Code in Python

```python
def min_path_sum(grid):
    if not grid:
        return 0

    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = grid[0][0]

    for i in range(1, m):
        dp[i][0] = dp[i - 1][0] + grid[i][0]

    for j in range(1, n):
        dp[0][j] = dp[0][j - 1] + grid[0][j]

    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j]

    return dp[m - 1][n - 1]

# Example usage
print(min_path_sum([[1, 3, 1], [1, 5, 1], [4, 2, 1]]))
```

---

8. House Robber Problem

Detailed Problem Explanation

The House Robber problem involves determining the maximum amount of money a robber can rob tonight without alerting the police. Adjacent houses cannot be robbed on the same night.

Input and Output Example

```python
# Example Input
nums = [2, 7, 9, 3, 1]

# Example Output
12  # (Rob houses 2, 4, and 5)
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i]` represents the maximum amount that can be robbed from the first `i` houses.
2. **Initialize**: Set the first two values based on the maximum of the first house or the sum of the first and second houses.
3. **Update DP Array**: For each house, decide to either rob it (add its value to the maximum amount robbed from two houses before) or skip it.
4. **Return Result**: The result is found in `dp[n]`.

Optimized Code in Python

```python
def house_robber(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    dp = [0] * (len(nums) + 1)
    dp[1] = nums[0]

    for i in range(2, len(nums) + 1):
        dp[i] = max(dp[i - 1], dp[i - 2] + nums[i - 1])

    return dp[-1]

# Example usage
print(house_robber([2, 7, 9, 3, 1]))
```

---

9. Palindrome Partitioning

Detailed Problem Explanation

The Palindrome Partitioning problem involves partitioning a string into the minimum number of substrings such that each substring is a palindrome.

Input and Output Example

```
# Example Input
s = "aab"

# Example Output
1  # (The palindrome partitioning is ["aa", "b"])
```

Core Logic Behind the Solution

The solution uses dynamic programming and backtracking:

1. **Check for Palindromes**: Create a 2D array to keep track of palindromic substrings.
2. **DP Array**: Use a DP array where `dp[i]` represents the minimum cuts needed for the substring `s[0:i+1]`.
3. **Update DP Array**: Iterate through the string, checking for palindromes and updating the DP array accordingly.
4. **Return Result**: The result will be found in `dp[n-1]`.

Optimized Code in Python

```python
def min_cut_palindrome_partition(s):
    n = len(s)
    if n == 0:
        return 0

    # Create a DP table for palindromes
    is_palindrome = [[False] * n for _ in range(n)]
    for i in range(n):
        is_palindrome[i][i] = True
    for length in range(2, n + 1):
        for start in range(n - length + 1):
            end = start + length - 1
            if length == 2:
                is_palindrome[start][end] = (s[start] == s[end])
            else:
                is_palindrome[start][end] = (s[start] == s[end] and is_palindrome[start + 1]
[end - 1])

    # DP array for minimum cuts
    dp = [0] * n
    for i in range(n):
        min_cuts = float('inf')
        for j in range(i + 1):
            if is_palindrome[j][i]:
                min_cuts = 0 if j == 0 else min(min_cuts, dp[j - 1] + 1)
        dp[i] = min_cuts

    return dp[n - 1]

# Example usage
print(min_cut_palindrome_partition("aab"))
```

---

10. Word Break Problem

Detailed Problem Explanation

The Word Break problem involves determining if a given string can be segmented into a space-separated sequence of one or more dictionary words.

Input and Output Example

```
# Example Input
s = "leetcode"
word_dict = ["leet", "code"]

# Example Output
True  # (The string can be segmented as "leet code")
```

Core Logic Behind the Solution

The solution uses dynamic programming:

1. **DP Array**: Create a DP array where `dp[i]` represents whether the substring `s[0:i]` can be segmented.
2. **Initialize**: Set `dp[0]` to `True`, as an empty string can always be segmented.
3. **Update DP Array**: Iterate through the string and check if any previous segment can form a word in the dictionary.
4. **Return Result**: The result is found in `dp[len(s)]`.

Optimized Code in Python

```
def word_break(s, word_dict):
    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_dict:
                dp[i] = True
                break

    return dp[len(s)]

# Example usage
print(word_break("leetcode", ["leet", "code"]))
```

Graph

---

1. Depth-First Search (DFS)

Detailed Problem Explanation

Depth-First Search (DFS) is a traversal algorithm for graph structures. It explores as far as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack.

Input and Output Example

```
# Example Input
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
start_node = 'A'

# Example Output
# DFS Traversal: A, B, D, E, F, C
```

Core Logic Behind the Solution

1. **Recursive Approach**: Start from the initial node, mark it as visited, and recursively visit all adjacent nodes.
2. **Stack Usage**: Alternatively, use a stack to keep track of nodes to explore.
3. **Backtracking**: When a node has no unvisited adjacent nodes, backtrack to explore other paths.

Optimized Code in Python

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print(dfs(graph, 'A'))  # Output: {'A', 'B', 'D', 'E', 'F', 'C'}
```

---

2. Breadth-First Search (BFS)

Detailed Problem Explanation

Breadth-First Search (BFS) is another traversal algorithm for graph structures. It explores all the neighbor nodes at the present depth before moving on to nodes at the next depth level. BFS is typically implemented using a queue.

Input and Output Example

```
# Example Input
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
start_node = 'A'

# Example Output
# BFS Traversal: A, B, C, D, E, F
```

Core Logic Behind the Solution

1. **Queue Usage**: Start from the initial node, mark it as visited, and enqueue it.
2. **Level Order Traversal**: Dequeue a node, visit all its unvisited neighbors, and enqueue them.
3. **Continue**: Repeat until the queue is empty.

Optimized Code in Python

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print(bfs(graph, 'A'))  # Output: {'A', 'B', 'C', 'D', 'E', 'F'}
```

---

3. Dijkstra's Algorithm

Detailed Problem Explanation

Dijkstra's Algorithm finds the shortest path from a starting node to all other nodes in a weighted graph with non-negative weights.

Input and Output Example

```
# Example Input
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'C': 2, 'D': 5},
    'C': {'D': 1},
    'D': {}
}
start_node = 'A'

# Example Output
# Shortest paths: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

Core Logic Behind the Solution

1. **Initialization**: Set the distance to the start node as 0 and all others to infinity.
2. **Priority Queue**: Use a priority queue to select the node with the smallest distance.
3. **Relaxation**: Update the distances of adjacent nodes if a shorter path is found.
4. **Repeat**: Continue until all nodes have been processed.

Optimized Code in Python

```python
import heapq

def dijkstra(graph, start):
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'C': 2, 'D': 5},
    'C': {'D': 1},
    'D': {}
}
print(dijkstra(graph, 'A'))  # Output: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

---

4. A* Search Algorithm

Detailed Problem Explanation

A* is a pathfinding and graph traversal algorithm that uses heuristics to find the shortest path from a start node to a goal node. It combines features of Dijkstra's Algorithm and Greedy Best-First Search.

Input and Output Example

```
# Example Input
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'C': 2, 'D': 5},
    'C': {'D': 1},
    'D': {}
}
heuristics = {
    'A': 7,
    'B': 6,
    'C': 2,
    'D': 0
}
start_node = 'A'
goal_node = 'D'

# Example Output
# Path: ['A', 'B', 'C', 'D']
```

Core Logic Behind the Solution

1. **Priority Queue**: Use a priority queue to explore the lowest estimated cost (actual cost + heuristic).
2. **Heuristic Function**: Combine the cost from the start node to the current node with the estimated cost from the current node to the goal.
3. **Path Reconstruction**: Keep track of the path by storing parent nodes.

Optimized Code in Python

```python
import heapq

def a_star(graph, start, goal, heuristics):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {node: float('infinity') for node in graph}
    g_score[start] = 0
    f_score = {node: float('infinity') for node in graph}
    f_score[start] = heuristics[start]

    while open_set:
        current = heapq.heappop(open_set)[1]

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            return path[::-1]

        for neighbor, cost in graph[current].items():
            tentative_g_score = g_score[current] + cost
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + heuristics[neighbor]
                if neighbor not in [i[1] for i in open_set]:
                    heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return []

# Example usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'C': 2, 'D': 5},
    'C': {'D': 1},
    'D': {}
}
heuristics = {
    'A': 7,
    'B': 6,
    'C': 2,
    'D': 0
}
print(a_star(graph, 'A', 'D', heuristics))  # Output: ['A', 'B', 'C', 'D']
```

5. Bellman-Ford Algorithm

Detailed Problem Explanation

The Bellman-Ford algorithm finds the shortest paths from a source vertex to all other vertices in a weighted graph, allowing for negative weights. It can also detect negative weight cycles.

Input and Output Example

```
# Example Input
edges = [
    ('A', 'B', 1),
    ('B', 'C', 2),
    ('A', 'C', 4),
    ('C', 'D', 3),
    ('D', 'B', -5)
]
start_node = 'A'

# Example Output
# Shortest paths: {'A': 0, 'B': 1, 'C': 3, 'D': 6}
```

Core Logic Behind the

Solution

1. **Initialization**: Set the distance to the start node as 0 and all others to infinity.
2. **Relaxation**: For each edge, update the distance to the destination vertex if a shorter path is found.
3. **Repeat**: Perform the relaxation for `V-1` times, where `V` is the number of vertices.
4. **Negative Cycle Check**: Check for negative weight cycles by performing one more relaxation round.

Optimized Code in Python

```python
def bellman_ford(edges, start):
    distances = {node: float('infinity') for edge in edges for node in edge[:2]}
    distances[start] = 0

    for _ in range(len(distances) - 1):
        for u, v, weight in edges:
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight

    # Check for negative weight cycles
    for u, v, weight in edges:
        if distances[u] + weight < distances[v]:
            raise ValueError("Graph contains a negative weight cycle")

    return distances

# Example usage
edges = [
    ('A', 'B', 1),
    ('B', 'C', 2),
    ('A', 'C', 4),
    ('C', 'D', 3),
    ('D', 'B', -5)
]
print(bellman_ford(edges, 'A'))  # Output: {'A': 0, 'B': 1, 'C': 3, 'D': 6}
```

---

6. Floyd-Warshall Algorithm

Detailed Problem Explanation

The Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices in a weighted graph. It works with negative weights but no negative cycles.

Input and Output Example

```
# Example Input
graph = [
    [0, 5, float('inf'), 10],
    [float('inf'), 0, 3, float('inf')],
    [float('inf'), float('inf'), 0, 1],
    [float('inf'), float('inf'), float('inf'), 0]
]

# Example Output
# Shortest paths matrix:
# [[0, 5, 8, 9],
#  [float('inf'), 0, 3, 4],
#  [float('inf'), float('inf'), 0, 1],
#  [float('inf'), float('inf'), float('inf'), 0]]
```

Core Logic Behind the Solution

1. **Initialization**: Create a distance matrix initialized with the graph's weights.
2. **Update Matrix**: For each pair of vertices, check if a path through an intermediate vertex offers a shorter path.
3. **Repeat**: Iterate over all vertices as potential intermediate nodes.

Optimized Code in Python

```python
def floyd_warshall(graph):
    num_vertices = len(graph)
    dist = [[float('inf')] * num_vertices for _ in range(num_vertices)]

    for i in range(num_vertices):
        for j in range(num_vertices):
            dist[i][j] = graph[i][j]

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                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, 5, float('inf'), 10],
    [float('inf'), 0, 3, float('inf')],
    [float('inf'), float('inf'), 0, 1],
    [float('inf'), float('inf'), float('inf'), 0]
]
result = floyd_warshall(graph)
for row in result:
    print(row)
```

---

7. Topological Sorting

Detailed Problem Explanation

Topological sorting is a linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u \to v), vertex (u) comes before vertex (v).

Input and Output Example

```
# Example Input
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D'],
    'D': []
}

# Example Output
# Topological Order: ['A', 'B', 'C', 'D'] or similar valid order
```

Core Logic Behind the Solution

1. **Indegree Count**: Calculate the indegree of each vertex.
2. **Queue**: Start with vertices that have an indegree of 0.
3. **Processing**: Remove vertices from the queue, appending them to the result, and reduce the indegree of their neighbors.
4. **Continue**: Repeat until the queue is empty.

Optimized Code in Python

```python
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = {node: 0 for node in graph}
    for neighbors in graph.values():
        for neighbor in neighbors:
            indegree[neighbor] += 1

    queue = deque([node for node in graph if indegree[node] == 0])
    top_order = []

    while queue:
        current = queue.popleft()
        top_order.append(current)

        for neighbor in graph[current]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    return top_order if len(top_order) == len(graph) else "Graph has a cycle"

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D'],
    'D': []
}
print(topological_sort(graph))  # Output: ['A', 'B', 'C', 'D'] or similar valid order
```

---

8. Minimum Spanning Tree (Prim's and Kruskal's)

Detailed Problem Explanation

The Minimum Spanning Tree (MST) problem involves finding a subset of edges that connects all vertices in a graph with the minimum total edge weight. Two common algorithms for this are Prim's and Kruskal's.

Prim's Algorithm focuses on building the MST incrementally, while **Kruskal's Algorithm** works by sorting all edges and adding them one by one if they don't form a cycle.

Input and Output Example (Prim's)

```
# Example Input (Prim's)
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'C': 1, 'D': 5},
    'C': {'A': 3, 'B': 1, 'D': 2},
    'D': {'B': 5, 'C': 2}
}

# Example Output (Prim's)
# MST: [('A', 'B', 1), ('B', 'C', 1), ('C', 'D', 2)]
```

Core Logic Behind Prim's Algorithm

1. **Initialization**: Start with an arbitrary node and keep track of visited nodes.
2. **Priority Queue**: Use a priority queue to select the edge with the minimum weight that connects a visited node to an unvisited node.
3. **Building MST**: Repeat until all nodes are included in the MST.

Optimized Code in Python (Prim's)

```python
import heapq

def prim(graph):
    mst = []
    total_cost = 0
    visited = set()
    min_heap = [(0, 'A')]  # (cost, start_node)

    while min_heap:
        cost, node = heapq.heappop(min_heap)
        if node in visited:
            continue
        visited.add(node)
        total_cost += cost

        for neighbor, weight in graph[node].items():
            if neighbor not in visited:
                heapq.heappush(min_heap, (weight, neighbor))
                mst.append((node, neighbor, weight))

    return mst, total_cost

# Example usage
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'C': 1, 'D': 5},
    'C': {'A': 3, 'B': 1, 'D': 2},
    'D': {'B': 5, 'C': 2}
}
print(prim(graph))  # Output: MST and total cost
```

---

9. Strongly Connected Components (Tarjan's Algorithm)

Detailed Problem Explanation

Tarjan's Algorithm finds the strongly connected components (SCCs) of a directed graph. An SCC is a maximal subgraph where every vertex is reachable from every other vertex in the same subgraph.

Input and Output Example

```
# Example Input
graph = {
    'A': ['B'],
    'B': ['C'],
    'C': ['A', 'D'],
    'D': []
}

# Example Output
# SCCs: [['A', 'B', 'C'], ['D']]
```

Core Logic Behind the Solution

1. **DFS with Indices**: Perform a depth-first search (DFS) while maintaining an index and low-link values for each node.
2. **Stack Usage**: Use a stack to track the current path and identify SCCs when backtracking.
3. **Identify SCCs**: When the current node's index equals its low-link value, pop nodes from the stack to form an SCC.

Optimized Code in Python

```
python
def tarjans_scc(graph):
    index = [0]
    stack = []
    lowlink = {}
    on_stack = set()
    sccs = []

    def strongconnect(node):
        lowlink[node] = index[0]
        idx = index[0]
        index[0] += 1
        stack.append(node)
        on_stack.add(node)

        for neighbor in graph[node]:
            if neighbor not in lowlink:
                strongconnect(neighbor)
                lowlink[node] = min(lowlink[node], lowlink[neighbor])
            elif neighbor in on_stack:
                lowlink[node] = min(lowlink[node], lowlink[neighbor])

        if lowlink[node] == idx:
            scc = []
            while True:
                w = stack.pop()
                on_stack.remove(w)
                scc.append(w)
                if w == node:
                    break
            sccs.append(scc)

    for v in graph:
        if v not in lowlink:
            strongconnect(v)

    return sccs

# Example usage
graph = {
    'A': ['B'],
    'B': ['C'],
    'C': ['A', 'D'],
    'D': []
}
print(tarjans_scc(graph))  # Output: [['A', 'B', 'C'], ['D']]
```

10. Graph Coloring Problem

Detailed Problem Explanation

The Graph Coloring Problem involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The goal is to use the minimum number of colors.

Input and Output Example

```
# Example Input
graph = {
    0: [1, 2],
    1: [0, 2, 3],
    2: [0, 1],
    3: [1]
}

# Example Output
# Coloring: {0: 0, 1: 1, 2: 0, 3: 1} (or any valid coloring)
```

Core Logic Behind the Solution

1. **Greedy Algorithm**: Use a greedy approach to assign colors, ensuring no two adjacent vertices have the same color.
2. **Backtracking**: If a conflict occurs, backtrack and try different colors.
3. **Color Assignment**: Use an array to track assigned colors.

Optimized Code in Python

```python
def graph_coloring(graph, m):
    result = {}

    def is_safe(node, color):
        for neighbor in graph[node]:
            if neighbor in result and result[neighbor] == color:
                return False
        return True

    def color_graph(node):
        if node == len(graph):
            return True

        for color in range(m):
            if is_safe(node, color):
                result[node] = color
                if color_graph(node + 1):
                    return True
                del result[node]

        return False

    if color_graph(0):
        return result
    return "No solution"

# Example usage
graph = {
    0: [1, 2],
    1: [0, 2, 3],
    2: [0, 1],
    3: [1]
}
print(graph_coloring(graph, 2))  # Output: Coloring dictionary or "No solution"
```

How to build graph

Building a graph can be done in two common ways: using an adjacency list or an adjacency matrix. Here's how to implement both methods in Python:

1. Adjacency List

An adjacency list represents a graph as a dictionary where each key is a vertex, and its value is a list of adjacent vertices.

Example Code for Adjacency List

```
# Using a dictionary to represent an adjacency list
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)  # For undirected graph; remove for directed graph

    def display(self):
        for vertex in self.graph:
            print(f"{vertex}: {self.graph[vertex]}")

# Example usage
g = Graph()
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'D')
g.add_edge('C', 'D')
g.display()
```

Output

```
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'D']
D: ['B', 'C']
```

2. Adjacency Matrix

An adjacency matrix represents a graph as a 2D array (list of lists) where each element at position (i, j) indicates whether there is an edge between vertex (i) and vertex (j).

Example Code for Adjacency Matrix

```python
class Graph:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.adj_matrix = [[0] * num_vertices for _ in range(num_vertices)]

    def add_edge(self, u, v):
        self.adj_matrix[u][v] = 1
        self.adj_matrix[v][u] = 1  # For undirected graph; remove for directed graph

    def display(self):
        for row in self.adj_matrix:
            print(row)

# Example usage
g = Graph(4)  # Create a graph with 4 vertices (0, 1, 2, 3)
g.add_edge(0, 1)  # A <-> B
g.add_edge(0, 2)  # A <-> C
g.add_edge(1, 3)  # B <-> D
g.add_edge(2, 3)  # C <-> D
g.display()
```

Output

```
[0, 1, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 1]
[0, 1, 1, 0]
```

Summary

- **Adjacency List**: Efficient for sparse graphs; uses less space compared to the adjacency matrix.
- **Adjacency Matrix**: Simple to implement; best for dense graphs where you need to check for the presence of edges quickly. However, it uses more space, especially for large graphs.