Sure! Here are the implementations for each of the algorithms in Python.

Dynamic Programming Algorithms

Floyd-Warshall Algorithm

```python
def floyd_warshall(graph):
    n = len(graph)
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(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]
]
print(floyd_warshall(graph))
```

Multistage Graph

```python
def multistage_graph(graph, stages):
    n = len(graph)
    cost = [float('inf')] * n
    cost[n - 1] = 0

    for i in range(n - 2, -1, -1):
        for j in range(i + 1, n):
            if graph[i][j] != float('inf'):
                cost[i] = min(cost[i], graph[i][j] + cost[j])

    return cost[0]

# Example usage
graph = [
    [0, 1, 2, float('inf')],
    [float('inf'), 0, float('inf'), 4],
    [float('inf'), 3, 0, 1],
    [float('inf'), float('inf'), float('inf'), 0]
]
stages = 4
print(multistage_graph(graph, stages))
```

0/1 Knapsack Problem

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

    for i in range(n + 1):
        for w in range(capacity + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i - 1] <= w:
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i -
1]], dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

# Example usage
weights = [1, 2, 3, 8, 7, 4]
values = [20, 5, 10, 40, 15, 25]
capacity = 10
print(knapsack(weights, values, capacity))
```

Optimal Binary Search Tree (OBST)

```
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for x in range(n)] for y in range(n)]

    for i in range(n):
        cost[i][i] = freq[i]

    for L in range(2, n + 1):
        for i in range(n - L + 1):
            j = i + L - 1
            cost[i][j] = float('inf')
            for r in range(i, j + 1):
                c = 0
                if r > i:
                    c += cost[i][r - 1]
                if r < j:
                    c += cost[r + 1][j]
                c += sum(freq[i:j + 1])
                if c < cost[i][j]:
                    cost[i][j] = c

    return cost[0][n - 1]
```

```python
# Example usage
keys = [10, 12, 20]
freq = [34, 8, 50]
print(optimal_bst(keys, freq))
```

Backtracking Algorithms

N-Queens Problem

```python
def solve_n_queens(n):
    board = [['.' for _ in range(n)] for _ in range(n)]
    res = []

    def is_safe(board, row, col):
        for i in range(row):
            if board[i][col] == 'Q':
                return False
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False
        for i, j in zip(range(row, -1, -1), range(col, n)):
            if board[i][j] == 'Q':
                return False
        return True

    def solve(row):
        if row == n:
            res.append([''.join(r) for r in board])
            return
        for col in range(n):
            if is_safe(board, row, col):
                board[row][col] = 'Q'
                solve(row + 1)
                board[row][col] = '.'

    solve(0)
    return res

# Example usage
n = 4
print(solve_n_queens(n))
```

Graph Coloring

```python
def graph_coloring(graph, m):
    n = len(graph)
```

```python
    colors = [-1] * n

    def is_safe(v, c):
        for i in range(n):
            if graph[v][i] == 1 and colors[i] == c:
                return False
        return True

    def solve(v):
        if v == n:
            return True
        for c in range(1, m + 1):
            if is_safe(v, c):
                colors[v] = c
                if solve(v + 1):
                    return True
                colors[v] = -1
        return False

    if solve(0):
        return colors
    else:
        return None

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

Sum of Subsets

```python
def sum_of_subsets(arr, target):
    res = []
    n = len(arr)

    def solve(i, curr_sum, path):
        if curr_sum == target:
            res.append(path)
            return
        if i >= n or curr_sum > target:
            return
        solve(i + 1, curr_sum + arr[i], path + [arr[i]])
        solve(i + 1, curr_sum, path)

    solve(0, 0, [])
```

```python
    return res

# Example usage
arr = [1, 2, 3, 4, 5]
target = 10
print(sum_of_subsets(arr, target))
```

Backtracking Knapsack Problem

```python
def backtracking_knapsack(weights, values, capacity):
    n = len(weights)
    max_value = [0]

    def solve(i, curr_weight, curr_value):
        if curr_weight <= capacity and curr_value > max_value[0]:
            max_value[0] = curr_value
        if i == n:
            return
        solve(i + 1, curr_weight + weights[i], curr_value + values[i])
        solve(i + 1, curr_weight, curr_value)

    solve(0, 0, 0)
    return max_value[0]

# Example usage
weights = [1, 2, 3, 8, 7, 4]
values = [20, 5, 10, 40, 15, 25]
capacity = 10
print(backtracking_knapsack(weights, values, capacity))
```

These implementations cover the algorithms you requested using Python. Each algorithm is designed to be clear and efficient.