**Greedy Algorithm**

Problem: The Coin Change Problem

Description: Given a set of coin denominations (e.g., {1, 5, 10, 25}) and an amount to make change for, find the minimum number of coins needed to make that amount.

**Pseudo-Code:**

function greedyCoinChange(coins, amount):

sort coins in descending order // Sort the coins from largest to smallest

coinCount = 0 // Initialize a variable to count the number of coins used

currentIndex = 0 // Initialize an index to track the current coin

while amount > 0:

if coins[currentIndex] <= amount:

// If the current coin is less than or equal to the remaining amount

// Use as many of these coins as possible

numCoins = floor(amount / coins[currentIndex])

coinCount += numCoins

amount -= numCoins \* coins[currentIndex]

else:

// If the current coin is too large, move to the next smaller coin

currentIndex += 1

return coinCount

**Dynamic Programming**

Problem: The Fibonacci Sequence

Description: Given a positive integer n, find the nth Fibonacci number.

**Pseudo-Code (Recursive Solution):**

function fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n - 1) + fibonacci(n - 2)

**Pseudo-Code (Dynamic Programming with Memoization):**

function fibonacci(n, memo={}):

if n in memo:

return memo[n]

if n <= 1:

memo[n] = n

else:

memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)

return memo[n]

**Traversal Graph Algorithm**

Question: Given an undirected graph and a starting vertex, find all vertices reachable from the starting vertex.

**Breadth-First Search (BFS) Pseudo-Code:**

function BFS(graph, startVertex):

visited = {} // Create a set to track visited vertices

queue = [] // Create a queue for BFS

queue.enqueue(startVertex) // Enqueue the start vertex

while queue is not empty:

vertex = queue.dequeue() // Dequeue a vertex from the queue

if vertex is not visited:

visited.add(vertex) // Mark the vertex as visited

process(vertex) // Process the vertex (e.g., print or perform an action)

// Enqueue adjacent vertices that haven't been visited

for neighbor in graph.adjacentVertices(vertex):

if neighbor is not visited:

queue.enqueue(neighbor)

**Depth-First Search (DFS) Pseudo-Code:**

function DFS(graph, startVertex):

visited = {} // Create a set to track visited vertices

function recursiveDFS(vertex):

if vertex is not visited:

visited.add(vertex) // Mark the vertex as visited

process(vertex) // Process the vertex (e.g., print or perform an action)

// Recursively visit adjacent vertices that haven't been visited

for neighbor in graph.adjacentVertices(vertex):

if neighbor is not visited:

recursiveDFS(neighbor)

recursiveDFS(startVertex)

**Topological Graph Algorithm:**

Problem: Given a directed acyclic graph (DAG), perform a topological sort on the graph. Topological sorting is an ordering of the vertices of the graph such that for every directed edge (u, v), vertex u comes before vertex v in the ordering.

Pseudo-Code for Topological Sort (using Depth-First Search, DFS):

function topologicalSort(graph):

stack = [] // Create an empty stack to store the sorted vertices

visited = {} // Create a set to track visited vertices

function recursiveDFS(vertex):

if vertex is not visited:

visited.add(vertex) // Mark the vertex as visited

// Recursively visit adjacent vertices

for neighbor in graph.adjacentVertices(vertex):

if neighbor is not visited:

recursiveDFS(neighbor)

stack.push(vertex) // Push the current vertex onto the stack after all its neighbors are visited

for each vertex in graph:

if vertex is not visited:

recursiveDFS(vertex)

return stack.reverse() // Reverse the stack to get the topological ordering

**Shortest path graph algorithm**

Problem: Given a weighted, directed or undirected graph, find the shortest path from a specified source vertex to all other vertices in the graph.

**Pseudo-Code for Dijkstra's Algorithm:**

function dijkstra(graph, source):

distances = {} // Create a dictionary to store the shortest distances

previous = {} // Create a dictionary to store the previous vertex in the shortest path

unvisited = set() // Create a set of unvisited vertices

// Initialize distances and previous vertices

for vertex in graph.vertices:

distances[vertex] = infinity // Set initial distance to infinity

previous[vertex] = None // Initialize previous vertex as None

unvisited.add(vertex)

distances[source] = 0 // Distance from source to itself is 0

while unvisited is not empty:

// Find the vertex with the smallest distance

current = vertex in unvisited with smallest distances[vertex]

// Mark current vertex as visited

unvisited.remove(current)

// Update distances to neighbors through the current vertex

for neighbor in graph.neighbors(current):

tentative\_distance = distances[current] + edge\_weight(current, neighbor)

if tentative\_distance < distances[neighbor]:

distances[neighbor] = tentative\_distance

previous[neighbor] = current

return distances, previous

**Biconnected component graph algorithm**

Problem: Given an undirected graph, find its biconnected components.

**Pseudo-Code for Finding Biconnected Components:**

function findBiconnectedComponents(graph):

n = number of vertices in graph

discovery\_time = [0] \* n

low = [0] \* n

parent = [-1] \* n

visited = [False] \* n

biconnected\_components = []

def DFS(current\_vertex, current\_time):

visited[current\_vertex] = True

discovery\_time[current\_vertex] = current\_time

low[current\_vertex] = current\_time

current\_time += 1

child\_count = 0

is\_articulation\_point = False

for neighbor in graph[current\_vertex]:

if not visited[neighbor]:

child\_count += 1

parent[neighbor] = current\_vertex

DFS(neighbor, current\_time)

low[current\_vertex] = min(low[current\_vertex], low[neighbor])

if low[neighbor] >= discovery\_time[current\_vertex]:

is\_articulation\_point = True

if low[neighbor] > discovery\_time[current\_vertex]:

# (current\_vertex, neighbor) is a bridge

biconnected\_components.append((current\_vertex, neighbor))

elif neighbor != parent[current\_vertex]:

low[current\_vertex] = min(low[current\_vertex], discovery\_time[neighbor])

if parent[current\_vertex] == -1 and child\_count >= 2:

is\_articulation\_point = True

if is\_articulation\_point:

# current\_vertex is an articulation point

biconnected\_components.append((current\_vertex, -1))

for vertex in range(n):

if not visited[vertex]:

DFS(vertex, 0)

return biconnected\_components

**Strongly Directed Graph**

Problem: Given a directed graph, find its strongly connected components.

**Pseudo-Code for Finding Strongly Connected Components (Kosaraju's Algorithm):**

function findStronglyConnectedComponents(graph):

# First pass: Perform DFS and calculate finishing times

n = number of vertices in graph

visited = [False] \* n

finishing\_times = []

stack = []

def firstPassDFS(vertex):

visited[vertex] = True

for neighbor in graph[vertex]:

if not visited[neighbor]:

firstPassDFS(neighbor)

stack.append(vertex)

finishing\_times.append(vertex)

for vertex in range(n):

if not visited[vertex]:

firstPassDFS(vertex)

# Reverse the graph

reversed\_graph = reverse(graph)

# Second pass: Perform DFS in reverse order of finishing times

visited = [False] \* n

strongly\_connected\_components = []

def secondPassDFS(vertex, current\_scc):

visited[vertex] = True

current\_scc.append(vertex)

for neighbor in reversed\_graph[vertex]:

if not visited[neighbor]:

secondPassDFS(neighbor, current\_scc)

while stack is not empty:

vertex = stack.pop()

if not visited[vertex]:

current\_scc = []

secondPassDFS(vertex, current\_scc)

strongly\_connected\_components.append(current\_scc)

return strongly\_connected\_components

**Network Flow Graph Algorithm**

Problem: Given a flow network with capacities, find the maximum flow from a source vertex to a sink vertex.

**Pseudo-Code for the Ford-Fulkerson Algorithm (Edmonds-Karp Variant):**

function fordFulkerson(graph, source, sink):

residualGraph = createResidualGraph(graph) // Initialize the residual graph

maxFlow = 0

while true: // Repeat until no augmenting paths are found

parent = bfs(residualGraph, source, sink) // Find an augmenting path using BFS

if not parent:

break // If no augmenting path is found, terminate

// Find the minimum capacity along the augmenting path

minCapacity = infinity

currentVertex = sink

while currentVertex != source:

parentVertex = parent[currentVertex]

edge = residualGraph[parentVertex][currentVertex]

minCapacity = min(minCapacity, edge.capacity)

currentVertex = parentVertex

// Update the flow and residual graph along the augmenting path

currentVertex = sink

while currentVertex != source:

parentVertex = parent[currentVertex]

edge = residualGraph[parentVertex][currentVertex]

edge.flow += minCapacity

// Add the reverse edge for residual capacity

reverseEdge = residualGraph[currentVertex][parentVertex]

reverseEdge.flow -= minCapacity

currentVertex = parentVertex

maxFlow += minCapacity

return maxFlow

function bfs(graph, source, sink):

queue = [source]

visited = set()

parent = {}

while queue:

currentVertex = queue.pop(0)

visited.add(currentVertex)

if currentVertex == sink:

return parent

for neighbor, edge in graph[currentVertex].items():

if neighbor not in visited and edge.capacity - edge.flow > 0:

parent[neighbor] = currentVertex

queue.append(neighbor)

return None

function createResidualGraph(graph):

residualGraph = {}

for u in graph:

residualGraph[u] = {}

for v, edge in graph[u].items():

residualGraph[u][v] = Edge(edge.capacity, 0)

residualGraph[v][u] = Edge(0, 0) // Reverse edge with 0 flow

return residualGraph