Chapter 1: Introduction and Analysis of Algorithms

1.1 What is an Algorithm?

An **algorithm** is a well-defined set of instructions designed to perform a task or solve a specific problem. It is a sequence of computational steps that takes input, processes it, and produces an output.

Properties of Algorithms

1. **Finiteness**: An algorithm must always terminate after a finite number of steps.
2. **Definiteness**: Each step of the algorithm should be clearly defined and unambiguous.
3. **Input**: An algorithm can accept zero or more inputs.
4. **Output**: An algorithm must produce one or more outputs.
5. **Effectiveness**: Each operation in the algorithm should be basic enough to be performed in a finite amount of time.

Types of Algorithms

- **Brute Force**: Involves trying all possible solutions to find the best one.
- **Divide and Conquer**: Breaks down a problem into smaller subproblems, solves each recursively, and combines their solutions.
- **Greedy Algorithms**: Makes a series of choices that look best at the moment, without considering future consequences.
- **Dynamic Programming**: Solves problems by breaking them down into simpler subproblems and storing their solutions.

1.2 Algorithm Analysis

1.2.1 Importance of Algorithm Analysis

Algorithm analysis helps us understand the efficiency of an algorithm in terms of time and space. It allows us to predict how the algorithm will perform as the size of input increases.

1.2.2 Time Complexity

Time complexity measures the amount of time an algorithm takes to complete as a function of the size of the input. It is often expressed using Big O notation.

- **Best Case**: The minimum time required for an algorithm to complete.
- **Worst Case**: The maximum time required for an algorithm to complete.
- **Average Case**: The expected time required for an algorithm based on the distribution of inputs.

1.2.3 Space Complexity

Space complexity measures the amount of memory space required by an algorithm to execute as a function of the size of the input.

1.2.4 Asymptotic Notations

- **Big O Notation (O)**: Describes an upper bound on the time complexity, indicating the worst-case scenario.
- **Big Omega Notation (Ω)**: Describes a lower bound on the time complexity, indicating the best-case scenario.
- **Big Theta Notation (Θ)**: Represents a tight bound on the complexity, indicating that the algorithm behaves the same in both upper and lower limits.

1.2.5 Analyzing Control Statements

Control statements (if-else, loops) impact the complexity of algorithms. Understanding how these statements affect time and space complexity is essential.

1.2.6 Recurrences

Recurrences are equations that define a function in terms of its value at smaller inputs. Common methods for solving recurrences include:

- **Substitution Method**: Guess the solution and prove it by induction.
- **Recursion Tree Method**: Visualize the recursion tree and compute costs.
- **Master Theorem**: A formula that provides an easy way to analyze the time complexity of divide-and-conquer algorithms.

1.2.7 Sorting Techniques

Sorting is a fundamental operation in computer science. Common sorting algorithms include:

1. **Bubble Sort**
    - **Description**: Repeatedly swaps adjacent elements if they are in the wrong order.
    - **Time Complexity**: O(n2)O(n^2)O(n2) in the worst case.

python

CODE

```
def bubble_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
```

2. **Selection Sort**
    - **Description**: Divides the array into sorted and unsorted regions, repeatedly selecting the minimum from the unsorted region.

- **Time Complexity**: O(n2)O(n^2)O(n2) in the worst case.

python

CODE

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

3. **Insertion Sort**
    - **Description**: Builds a sorted array one element at a time.
    - **Time Complexity**: O(n2)O(n^2)O(n2) in the worst case, but O(n)O(n)O(n) in the best case.

python

CODE

```
def insertion_sort(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
```

Chapter 2: Divide & Conquer Algorithms

2.1 What is Divide and Conquer?

Divide and conquer is a strategy for solving complex problems by breaking them down into simpler subproblems. Each subproblem is solved independently, and the solutions are combined to solve the original problem.

2.1.1 Structure of Divide and Conquer Algorithms

1. **Divide**: Split the problem into smaller subproblems.
2. **Conquer**: Solve each subproblem recursively.
3. **Combine**: Merge the solutions of the subproblems to form a solution to the original problem.

2.1.2 Common Examples

1. **Binary Search**
    - **Description**: A search algorithm that finds the position of a target value within a sorted array by repeatedly dividing the search interval in half.
    - **Time Complexity**: O(log⁡n)O(\log n)O(logn).

python

CODE

```
def binary_search(arr, target):
  left, right = 0, len(arr) - 1
  while left <= right:
    mid = left + (right - left) // 2
    if arr[mid] == target:
      return mid
    elif arr[mid] < target:
      left = mid + 1
    else:
      right = mid - 1
  return -1
```

2. **Merge Sort**
    - **Description**: Divides the array into halves, sorts each half, and merges them back together.
    - **Time Complexity**: O(nlog⁡n)O(n \log n)O(nlogn).

python

CODE

```
def merge_sort(arr):
```

```
if len(arr) > 1:
    mid = len(arr) // 2
    L = arr[:mid]
    R = arr[mid:]

    merge_sort(L)
    merge_sort(R)

    i = j = k = 0
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
return arr
```

3. **Quick Sort**
    - **Description**: Selects a pivot element, partitions the array around the pivot, and recursively sorts the subarrays.

- **Time Complexity**: Average case O(nlog⁡n)O(n \log n)O(nlogn), worst case O(n2)O(n^2)O(n2).

python

CODE

```
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]
  return quick_sort(left) + middle + quick_sort(right)
```

4. **Strassen's Matrix Multiplication**
    - **Description**: An efficient algorithm for multiplying two matrices.
    - **Time Complexity**: O(nlog⁡27)≈O(n2.81)O(n^{\log_2 7}) \approx O(n^{2.81})O(nlog27)≈O(n2.81).

python

CODE

```
def strassen(A, B):
  # Implementation of Strassen's algorithm for matrix multiplication
  pass  # Placeholder for the detailed implementation
```

Chapter 3: Greedy Algorithms

3.1 Introduction to Greedy Algorithms

Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit. They are used when local optimization leads to global optimization.

3.1.1 Characteristics of Greedy Algorithms

- **Greedy Choice Property**: A local optimum leads to a global optimum.
- **Optimal Substructure**: An optimal solution contains optimal solutions to its subproblems.

3.1.2 Common Examples

1. **Minimum Spanning Tree**
    - **Kruskal's Algorithm**: Finds the minimum spanning tree for a connected weighted graph.

python

CODE

```
class UnionFind:
    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):
        rootU = self.find(u)
        rootV = self.find(v)
        if rootU != rootV:
            if self.rank[rootU] > self.rank[rootV]:
                self.parent[rootV] = rootU
            elif self.rank[rootU] < self.rank[rootV]:
                self.parent[rootU] = rootV
            else:
                self.parent[rootV] = rootU
                self.rank[rootU] += 1

def kruskal(vertices, edges):
    uf = UnionFind(len(vertices))
    mst = []
    edges.sort(key=lambda x: x[2])  # Sort by weight

    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
```

```
            uf.union(u, v)
            mst.append((u, v, weight))

    return mst
```

- **Prim's Algorithm**: Another method to find the minimum spanning tree, using a priority queue.

python

CODE

```
import heapq

def prim(graph, start):
    visited = set()
    min_heap = [(0, start)]
    mst_cost = 0
    mst_edges = []

    while min_heap:
        weight, u = heapq.heappop(min_heap)
        if u in visited:
            continue
        visited.add(u)
        mst_cost += weight
        mst_edges.append(u)

        for v, w in graph[u].items():
            if v not in visited:
                heapq.heappush(min_heap, (w, v))

    return mst_cost, mst_edges
```

2. **Dijkstra’s Algorithm**

- **Description**: Finds the shortest path from a source vertex to all other vertices in a weighted graph.

python

CODE

```
import heapq


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


  while min_heap:
    current_distance, current_vertex = heapq.heappop(min_heap)


    if current_distance > distances[current_vertex]:
      continue


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


  return distances
```

3. **Knapsack Problem**
    - **0/1 Knapsack Problem**: A classic optimization problem where you aim to maximize value without exceeding weight capacity.

python

CODE

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

4. **Activity Selection Problem**
    - **Description**: Select the maximum number of activities that don't overlap.

python

CODE

```
def activity_selection(start, finish):
    activities = sorted(zip(start, finish), key=lambda x: x[1])
    selected = []
    last_finish = -1

    for s, f in activities:
        if s >= last_finish:
            selected.append((s, f))
            last_finish = f

    return selected
```

Chapter 4: Dynamic Programming

4.1 What is Dynamic Programming?

Dynamic Programming (DP) is an optimization technique used to solve problems by breaking them down into simpler subproblems and storing the results to avoid redundant computations. It is particularly useful for problems that exhibit overlapping subproblems and optimal substructure.

4.1.1 Characteristics of Dynamic Programming

- **Optimal Substructure**: An optimal solution can be constructed efficiently from optimal solutions of its subproblems.

- **Overlapping Subproblems**: The problem can be broken down into smaller subproblems, which are reused several times.

4.1.2 Common Examples

1. **0/1 Knapsack Problem** (DP Approach)
    - **Description**: Given weights and values, maximize the total value without exceeding the capacity.

python

CODE

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

2. **Making Change Problem**
    - **Description**: Find the minimum number of coins needed to make a given amount.

python

CODE

```
def make_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

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

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

3. **Longest Common Subsequence**
    - **Description**: Find the length of the longest subsequence present in both strings.

python

CODE

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

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif X[i - 1] == Y[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]
```

4. **All-Pairs Shortest Paths**
    - **Floyd-Warshall Algorithm**: A dynamic programming approach to find the shortest paths between all pairs of vertices in a weighted graph.

python

CODE

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

    for i in range(n):
        for j in range(n):
```

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

    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
```

Chapter 5: Exploring Graphs

5.1 Introduction to Graphs

Graphs are data structures that consist of a set of vertices (or nodes) connected by edges. They are used to represent various real-world problems such as networks, relationships, and paths.

5.1.1 Types of Graphs

- **Undirected Graph**: Edges have no direction.
- **Directed Graph**: Edges have a direction.
- **Weighted Graph**: Edges have weights representing costs or distances.

5.1.2 Graph Traversal

Graph traversal is the process of visiting all the vertices in a graph in a systematic manner. The two primary methods are:

1. **Depth-First Search (DFS)**
    - **Description**: Explores as far as possible along each branch before backtracking.

python

CODE

```
def dfs(graph, vertex, visited=None):
    if visited is None:
        visited = set()
    visited.add(vertex)

    for neighbor in graph[vertex]:
        if neighbor not in visited:
```

```
        dfs(graph, neighbor, visited)

    return visited
```

2. **Breadth-First Search (BFS)**
    - **Description**: Explores all neighbor nodes at the present depth before moving on to nodes at the next depth level.

python

CODE

```
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
```

3. **Topological Sort**
    - **Description**: Orders the vertices of a directed acyclic graph (DAG) such that for every directed edge u→vu \rightarrow vu→v, vertex uuu comes before vvv.

python

CODE

```
def topological_sort(graph):
    visited = set()
    stack = []
```

```
    def dfs(v):
        visited.add(v)
        for neighbor in graph[v]:
            if neighbor not in visited:
                dfs(neighbor)
        stack.append(v)

    for vertex in graph:
        if vertex not in visited:
            dfs(vertex)

    return stack[::-1]  # Return reversed stack
```

Chapter 6: Backtracking and Branch & Bound

6.1 Backtracking

Backtracking is a problem-solving technique that involves searching through all possible configurations of a problem to find a solution. It incrementally builds candidates for the solution and abandons those that fail to satisfy the constraints.

6.1.1 Common Backtracking Problems

1. **N-Queens Problem**
    - **Description**: Place N queens on an N×NN \times NN×N chessboard so that no two queens threaten each other.

python

CODE

```
def n_queens(n):
    def solve(board, row):
        if row == n:
            result.append(board[:])
            return
        for col in range(n):
            if is_safe(board, row, col):
                board[row] = col
```

```
            solve(board, row + 1)


    def is_safe(board, row, col):
        for i in range(row):
            if board[i] == col or \
                board[i] - i == col - row or \
                board[i] + i == col + row:
                return False
        return True


    result = []
    solve([-1] * n, 0)
    return result
```

2. **Traveling Salesman Problem (TSP)**
    - **Description**: Find the shortest possible route that visits each city exactly once and returns to the origin city.

python

CODE

```
from itertools import permutations


def traveling_salesman(graph):
    n = len(graph)
    min_path = float('inf')
    for perm in permutations(range(n)):
        cost = sum(graph[perm[i]][perm[i + 1]] for i in range(n - 1))
        cost += graph[perm[-1]][perm[0]]  # Return to start
        min_path = min(min_path, cost)
    return min_path
```

Chapter 7: String Matching & NP Completeness

7.1 String Matching Algorithms

String matching involves finding occurrences of a pattern within a text. Several algorithms exist for efficient string matching.

7.1.1 Naive String Matching

- **Description**: Checks for the pattern at every position in the text.

python

CODE

```
def naive_string_matching(text, pattern):
    n = len(text)
    m = len(pattern)
    for i in range(n - m + 1):
        if text[i:i + m] == pattern:
            print(f"Pattern found at index {i}")
```

7.1.2 Rabin-Karp Algorithm

- **Description**: Uses hashing to find any one of a set of pattern strings in a text.

python

CODE

```
def rabin_karp(text, pattern, d=256, q=101):
    m = len(pattern)
    n = len(text)
    p = 0  # hash value for pattern
    t = 0  # hash value for text
    h = 1

    for i in range(m - 1):
        h = (h * d) % q

    for i in range(m):
        p = (d * p + ord(pattern[i])) % q
        t = (d * t + ord(text[i])) % q

    for i in range(n - m + 1):
```

```
    if p == t:
        if text[i:i + m] == pattern:
            print(f"Pattern found at index {i}")
    if i < n - m:
        t = (d * (t - ord(text[i]) * h) + ord(text[i + m])) % q
        if t < 0:
            t += q
```

7.1.3 Knuth-Morris-Pratt (KMP) Algorithm

- **Description**: Preprocesses the pattern to create a longest prefix suffix (LPS) array.

python

CODE

```
def kmp_search(text, pattern):
    m = len(pattern)
    n = len(text)
    lps = [0] * m
    compute_lps(pattern, m, lps)

    i = 0
    j = 0
    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1

        if j == m:
            print(f"Pattern found at index {i - j}")
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
```

```
            i += 1


def compute_lps(pattern, m, lps):
    length = 0
    lps[0] = 0
    i = 1

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
```

7.2 NP Completeness

7.2.1 What is NP Completeness?

NP-completeness is a classification of decision problems for which no efficient solution is known. A problem is NP-complete if it is both in NP (nondeterministic polynomial time) and as hard as any problem in NP.

7.2.2 P Class Problems

- **P Class**: Problems that can be solved in polynomial time.

7.2.3 NP Class Problems

- **NP Class**: Problems for which a solution can be verified in polynomial time.

7.2.4 Hamiltonian Cycle

- **Description**: A cycle in a graph that visits every vertex exactly once. Determining if such a cycle exists is NP-complete.

7.2.5 Implications of NP Completeness

- If any NP-complete problem can be solved in polynomial time, then all NP problems can also be solved in polynomial time.