1 Number Theory

1.1 Chinese Remainder Theorem (CRT)

Definition: Provides a solution to a system of congruences with pairwise coprime moduli.

Usage: Use when solving simultaneous congruences or when combining results modulo different moduli.

Theorem: If $n_1, n_2, \dots, n_k$ are pairwise coprime positive integers, then the system:

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \vdots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

has a unique solution modulo $N = n_1 n_2 \dots n_k$.

```
def chinese_remainder_theorem(a_list, n_list):
    from functools import reduce
    def mul_inv(a, b):
        b0 = b
        x0, x1 = 0, 1
        if b == 1: return 1
        while a > 1:
            q = a // b
            a, b = b, a % b
            x0, x1 = x1 - q * x0, x0
        return x1 + b0 if x1 < 0 else x1
    N = reduce(lambda x, y: x * y, n_list)
    result = 0
    for a_i, n_i in zip(a_list, n_list):
        p = N // n_i
        result += a_i * mul_inv(p, n_i) * p
    return result % N
```

1.2 Euler's Totient Function

Definition: Denoted $\phi(n)$, it counts the positive integers up to n that are relatively prime to n.

Usage: Use in problems involving modular inverses, Fermat's little theorem generalization, and RSA encryption.

Formula: For prime p and integer k:

$$\phi(p^k) = p^k - p^{k-1}$$

For general n with prime factors p_i:

$$\phi(n) = n \prod_{p_i \mid n} \left(1 - \frac{1}{p_i}\right)$$

```python
def euler_totient(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p
        p += 1
    if n > 1:
        result -= result // n
    return result
```

1.3 Miller-Rabin Primality Test

Definition: A probabilistic primality test to check if a number is a probable prime.
Usage: Use for large numbers where deterministic primality tests are too slow.
Algorithm: Based on the properties of strong probable primes.

```python
import random

def is_prime(n, k=5):
    if n <= 1:
        return False
    elif n <= 3:
        return True
    elif n % 2 == 0:
        return False
    # Write n-1 as 2^s * d
    s, d = 0, n - 1
    while d % 2 == 0:
        d //= 2
        s += 1
    # Witness loop
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(s - 1):
            x = x * x % n
            if x == n - 1:
                break
        else:
            return False
    return True
```

1.4 Pollard's Rho Algorithm

Definition: An efficient probabilistic integer factorization algorithm.
Usage: Use to factor large integers when prime factorization is needed.

```
import math
import random

def pollards_rho(n):
    if n % 2 == 0:
        return 2
    x = random.randrange(2, n)
    y = x
    c = random.randrange(1, n)
    d = 1
    while d == 1:
        x = (x * x + c) % n
        y = (y * y + c) % n
        y = (y * y + c) % n
        d = math.gcd(abs(x - y), n)
        if d == n:
            return pollards_rho(n)
    if is_prime(d):
        return d
    else:
        return pollards_rho(d)
```

2 Graph Algorithms

2.1 Breadth-First Search (BFS)

Usage: Finding the shortest path in unweighted graphs.

```
from collections import deque

def bfs(graph, start):
    visited = set()
    distance = {start: 0}
    queue = deque([start])
    visited.add(start)
    while queue:
        vertex = queue.popleft()
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                distance[neighbor] = distance[vertex] + 1
                queue.append(neighbor)
    return distance
```

2.2 Depth-First Search (DFS)

Usage: Topological sorting, cycle detection, and traversal.

```
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)
```

2.3 Dijkstra's Algorithm for Weighted Graphs

Usage: Finding the shortest path in weighted graphs without negative edges.

```
import heapq

def dijkstra(graph, start):
    heap = [(0, start)]
    distance = {vertex: float('inf') for vertex in graph}
    distance[start] = 0
    while heap:
        dist_u, u = heapq.heappop(heap)
        if dist_u > distance[u]:
            continue
        for v, weight in graph[u]:
            if distance[u] + weight < distance[v]:
                distance[v] = distance[u] + weight
                heapq.heappush(heap, (distance[v], v))
    return distance
```

2.4 Lowest Common Ancestor (LCA)

Problem: Find the lowest common ancestor of two nodes in a tree.

Algorithm: Binary Lifting or Euler Tour with RMQ.

```
class Tree:
    def __init__(self, n):
        self.n = n
        self.LOGN = n.bit_length()
        self.parent = [[-1]*self.LOGN for _ in range(n)]
        self.depth = [0]*n
        self.graph = [[] for _ in range(n)]

```

```
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, u, p):
        self.parent[u][0] = p
        for i in range(1, self.LOGN):
            if self.parent[u][i-1] != -1:
                self.parent[u][i] = self.parent[self.parent[u][i
                   -1]][i-1]
        for v in self.graph[u]:
            if v != p:
                self.depth[v] = self.depth[u] + 1
                self.dfs(v, u)

    def lca(self, u, v):
        if self.depth[u] < self.depth[v]:
            u, v = v, u
        for i in range(self.LOGN - 1, -1, -1):
            if self.parent[u][i] != -1 and self.depth[self.parent[u
               ][i]] >= self.depth[v]:
                u = self.parent[u][i]
        if u == v:
            return u
        for i in range(self.LOGN -1, -1, -1):
            if self.parent[u][i] != self.parent[v][i]:
                u = self.parent[u][i]
                v = self.parent[v][i]
        return self.parent[u][0]
```

2.5 Bellman-Ford Algorithm

Definition: Computes shortest paths from a single source vertex to all other vertices in a weighted digraph, which may have negative weight edges.

Usage: Use when graph contains negative weight edges and you need to detect negative cycles.

```
def bellman_ford(graph, V, E, source):
    dist = [float('inf')] * V
    dist[source] = 0
    for _ in range(V - 1):
        for u, v, w in E:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    # Check for negative-weight cycles
    for u, v, w in E:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
```

```
            return None  # Negative cycle detected
    return dist
```

2.6 Floyd-Warshall Algorithm

Definition: A dynamic programming algorithm to find shortest paths between all pairs of vertices in a weighted graph.

Usage: Use when needing all-pairs shortest paths and the graph has negative weights but no negative cycles.

```
def floyd_warshall(graph):
    V = len(graph)
    dist = [row[:] for row in graph]
    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

2.7 Kruskal's Algorithm

Definition: An algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected graph.

Usage: Use to minimize the total weight connecting all vertices.

```
def kruskal(V, edges):
    uf = UnionFind(V)
    mst_weight = 0
    mst_edges = []
    edges.sort(key=lambda x: x[2])  # Sort by weight
    for u, v, w in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst_weight += w
            mst_edges.append((u, v, w))
    return mst_weight, mst_edges
```

2.8 Tarjan's Algorithm for Strongly Connected Components

Definition: Finds all strongly connected components (SCCs) in a directed graph.

Usage: Use to identify cycles, condensation graphs, and analyze connectivity.

```
def tarjans_scc(graph):
    index = 0
    indices = {}
```

```python
    lowlink = {}
    stack = []
    on_stack = set()
    sccs = []

    def strongconnect(v):
        nonlocal index
        indices[v] = lowlink[v] = index
        index += 1
        stack.append(v)
        on_stack.add(v)
        for w in graph[v]:
            if w not in indices:
                strongconnect(w)
                lowlink[v] = min(lowlink[v], lowlink[w])
            elif w in on_stack:
                lowlink[v] = min(lowlink[v], indices[w])
        if lowlink[v] == indices[v]:
            scc = []
            while True:
                w = stack.pop()
                on_stack.remove(w)
                scc.append(w)
                if w == v:
                    break
            sccs.append(scc)
    for v in graph:
        if v not in indices:
            strongconnect(v)
    return sccs
```

2.9 Dinic's Algorithm for Maximum Flow

Definition: An efficient algorithm for computing the maximum flow in a flow network.

Usage: Use in network flow problems, bipartite matching, and circulation problems.

```python
from collections import deque

class Edge:
    def __init__(self, to, rev, capacity):
        self.to = to
        self.rev = rev
        self.capacity = capacity

class MaxFlow:
    def __init__(self, N):
        self.size = N
```

```python
        self.graph = [[] for _ in range(N)]
    def add(self, fr, to, capacity):
        forward = Edge(to, len(self.graph[to]), capacity)
        backward = Edge(fr, len(self.graph[fr]), 0)
        self.graph[fr].append(forward)
        self.graph[to].append(backward)
    def bfs_level(self, s, t, level):
        queue = deque([s])
        level[s] = 0
        while queue:
            v = queue.popleft()
            for e in self.graph[v]:
                if e.capacity > 0 and level[e.to] < 0:
                    level[e.to] = level[v] + 1
                    queue.append(e.to)
        return level[t] != -1
    def dfs_flow(self, level, iter, v, t, upTo):
        if v == t:
            return upTo
        for i in range(iter[v], len(self.graph[v])):
            e = self.graph[v][i]
            if e.capacity > 0 and level[v] < level[e.to]:
                d = self.dfs_flow(level, iter, e.to, t, min(upTo, e.capacity))
                if d > 0:
                    e.capacity -= d
                    self.graph[e.to][e.rev].capacity += d
                    return d
            iter[v] += 1
        return 0
    def max_flow(self, s, t):
        flow = 0
        level = [-1] * self.size
        INF = float('inf')
        while True:
            level = [-1] * self.size
            if not self.bfs_level(s, t, level):
                break
            iter = [0] * self.size
            while True:
                f = self.dfs_flow(level, iter, s, t, INF)
                if f == 0:
                    break
                flow += f
        return flow
```

3 Dynamic Programming

3.1 Subset Sum Algorithm

Given a set of integers $S = \{s_1, s_2, \ldots, s_n\}$ and a target integer T, determine whether there exists a subset $S' \subseteq S$ such that:

$$\sum_{x \in S'} x = T$$

```python
def subset_sum(S, T):
    n = len(S)
    dp = [[False] * (T + 1) for _ in range(n + 1)]
    dp[0][0] = True  # Base case: sum of 0 is always achievable with an empty subset

    for i in range(1, n + 1):
        for j in range(T + 1):
            dp[i][j] = dp[i - 1][j]
            if j >= S[i - 1] and dp[i - 1][j - S[i - 1]]:
                dp[i][j] = True

    return dp[n][T]
```

3.2 Longest Increasing Subsequence

Given a sequence of integers $A = \{a_1, a_2, \ldots, a_n\}$, find the length of the longest subsequence $A' \subseteq A$ where the elements of A' are in strictly increasing order.

```python
    def longest_increasing_subsequence(A):
    n = len(A)
    dp = [1] * n  # Initialize all LIS lengths to 1

    for i in range(1, n):
        for j in range(i):
            if A[j] < A[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)  # The length of the longest increasing subsequence
```

4 String Algorithms

4.1 Z-Algorithm

Definition: Computes an array Z where $Z[i]$ is the length of the longest substring starting from i that is also a prefix of the string.

Usage: Use for pattern matching, string compression, and finding repetitions.

```python
def z_algorithm(s):
    n = len(s)
    Z = [0] * n
    l, r = 0, 0
    for i in range(1, n):
        if i <= r:
            Z[i] = min(r - i + 1, Z[i - l])
        while i + Z[i] < n and s[Z[i]] == s[i + Z[i]]:
            Z[i] += 1
        if i + Z[i] - 1 > r:
            l, r = i, i + Z[i] - 1
    return Z
```

5 Geometry

5.1 Rotating Calipers

Definition: A computational geometry technique used to compute various properties of convex polygons.

Usage: Use to find the diameter of a convex polygon, the width, and solve problems involving pairs of points.

```python
def rotating_calipers(points):
    # Assume points are the convex hull in counterclockwise order
    n = len(points)
    max_distance = 0
    j = 1
    for i in range(n):
        next_i = (i + 1) % n
        while True:
            next_j = (j + 1) % n
            cross = (points[next_i][0] - points[i][0]) * \
                    (points[next_j][1] - points[j][1]) - \
                    (points[next_i][1] - points[i][1]) * \
                    (points[next_j][0] - points[j][0])
            if cross < 0:
                j = next_j
            else:
                break
        dist = (points[i][0] - points[j][0]) ** 2 + \
               (points[i][1] - points[j][1]) ** 2
        max_distance = max(max_distance, dist)
    return max_distance ** 0.5
```

5.2 Closest Pair of Points

Definition: Find the pair of points with the minimum distance between them.

Usage: Use in computational geometry and clustering problems.

```
def closest_pair(points):
    def closest_pair_rec(px, py):
        if len(px) <= 3:
            min_dist = float('inf')
            for i in range(len(px)):
                for j in range(i + 1, len(px)):
                    dist = ((px[i][0] - px[j][0]) ** 2 + \
                            (px[i][1] - px[j][1]) ** 2) ** 0.5
                    min_dist = min(min_dist, dist)
            return min_dist
        mid = len(px) // 2
        Qx = px[:mid]
        Rx = px[mid:]
        midpoint = px[mid][0]
        Qy = list(filter(lambda x: x[0] <= midpoint, py))
        Ry = list(filter(lambda x: x[0] > midpoint, py))
        delta = min(closest_pair_rec(Qx, Qy), closest_pair_rec(Rx,
           Ry))
        strip = [p for p in py if abs(p[0] - midpoint) < delta]
        min_dist_strip = delta
        for i in range(len(strip)):
            for j in range(i + 1, min(i + 7, len(strip))):
                dist = ((strip[i][0] - strip[j][0]) ** 2 + \
                        (strip[i][1] - strip[j][1]) ** 2) ** 0.5
                min_dist_strip = min(min_dist_strip, dist)
        return min_dist_strip
    px = sorted(points, key=lambda x: x[0])
    py = sorted(points, key=lambda x: x[1])
    return closest_pair_rec(px, py)
```

6 Mathematical Concepts

6.1 Matrix Exponentiation

Definition: Raising a matrix to a power efficiently using exponentiation by squaring.

Usage: Use in solving linear recurrences and dynamic programming optimizations.

```
def matrix_mult(A, B):
    result = [[0]*len(B[0]) for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
```

```
                result[i][j] += A[i][k] * B[k][j]
    return result

def matrix_pow(mat, power):
    result = [[int(i == j) for j in range(len(mat))] for i in range(
        len(mat))]
    while power > 0:
        if power % 2 == 1:
            result = matrix_mult(result, mat)
        mat = matrix_mult(mat, mat)
        power //= 1
    return result
```

7 Probability and Expected Value

7.1 Expected Number of Trials

Definition: The expected value is the average number of trials needed for a random process.

Usage: Use in problems involving probabilistic expected outcomes.

$$\text{Expected Value (E)} = \sum_{i=1}^{n} x_i p_i$$

```
def expected_trials(probabilities, values):
    expected_value = sum(p * v for p, v in zip(probabilities, values
        ))
    return expected_value
```

8 Recursive Memoization

Definition: Store the results of expensive function calls and return the cached result when the same inputs occur again.

Usage: Use in problems where recursive calls have overlapping subproblems, such as Fibonacci numbers, or when optimizing recursive solutions to prevent redundant calculations.

Example: Compute the n-th Fibonacci number.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

9 Greedy Algorithms

9.1 Greedy Interval Selection

Problem: Given a set of intervals, select the minimum number of intervals to cover the entire range or maximize the number of non-overlapping intervals.

Usage: Use when local optimal choices lead to a global optimum.

```
def interval_scheduling(intervals):
    intervals.sort(key=lambda x: x[1])  # Sort by end time
    count = 0
    end = float('-inf')
    for s, e in intervals:
        if s >= end:
            end = e
            count += 1
    return count
```

10 Hierarchical Parsing

Problem: Parse and validate hierarchical structures like XML or nested parentheses.

Usage: Use when dealing with nested data structures.

```
def is_valid_hierarchy(s):
    stack = []
    mapping = {')': '(', ']': '[', '}': '{'}
    for char in s:
        if char in mapping.values():
            stack.append(char)
        elif char in mapping.keys():
            if stack == [] or mapping[char] != stack.pop():
                return False
    return stack == []
```

11 Sliding Window Algorithms

11.1 Maintaining Unique Elements

Problem: Find the length of the longest substring without repeating characters.

Algorithm: Use a sliding window with a hash set.

```
def longest_unique_substring(s):
    char_set = set()
    left = result = 0
    for right in range(len(s)):
        while s[right] in char_set:
```

```python
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        result = max(result, right - left + 1)
    return result
```

11.2 Counting Connected Components (Number of Islands)

Definition: Determine the number of connected components (islands) in a grid where cells can be land or water.

Implementation Using DFS:

```python
def num_islands(grid):
    if not grid:
        return 0
    m, n = len(grid), len(grid[0])
    visited = [[False]*n for _ in range(m)]
    def dfs(x, y):
        if 0 <= x < m and 0 <= y < n and not visited[x][y] and grid[
            x][y] == '1':
            visited[x][y] = True
            dfs(x + 1, y)
            dfs(x - 1, y)
            dfs(x, y + 1)
            dfs(x, y - 1)
    count = 0
    for i in range(m):
        for j in range(n):
            if not visited[i][j] and grid[i][j] == '1':
                dfs(i, j)
                count += 1
    return count
```

11.3 2D Prefix Sums (Cumulative Sum in Grids)

Definition: Precompute a prefix sum matrix to allow efficient calculation of the sum of elements in any submatrix.

Usage: Use when needing to compute sums over submatrices multiple times efficiently.

Implementation:

```python
def compute_prefix_sums(grid):
    m, n = len(grid), len(grid[0])
    prefix_sums = [[0]*(n+1) for _ in range(m+1)]
    for i in range(m):
        for j in range(n):
            prefix_sums[i+1][j+1] = grid[i][j] + prefix_sums[i][j+1]
                + \
```

```
                                                prefix_sums[i+1][j] -
                                                    prefix_sums[i][j]
    return prefix_sums

def sum_region(prefix_sums, x1, y1, x2, y2):
    # Sum of rectangle from (x1, y1) to (x2, y2)
    return prefix_sums[x2+1][y2+1] - prefix_sums[x1][y2+1] - \
           prefix_sums[x2+1][y1] + prefix_sums[x1][y1]
```

11.4 Dynamic Programming on Grids (Unique Paths)

Problem: Count the number of unique paths from the top-left corner to the bottom-right corner of a grid, moving only down or right.

Implementation:

```
def unique_paths(m, n):
    dp = [[0]*n for _ in range(m)]
    for i in range(m):
        dp[i][0] = 1
    for j in range(n):
        dp[0][j] = 1
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]
```

11.5 Rotating and Flipping Grids

Rotating a Grid 90 Degrees Clockwise:

```
def rotate_grid(grid):
    return [list(row) for row in zip(*grid[::-1])]
```

Flipping a Grid Horizontally:

```
def flip_horizontal(grid):
    return [row[::-1] for row in grid]
```

Flipping a Grid Vertically:

```
def flip_vertical(grid):
    return grid[::-1]
```

11.6 Spiral Traversal of a Grid

Problem: Traverse a grid in a spiral order.

Implementation:

```python
def spiral_order(matrix):
    result = []
    if not matrix:
        return result
    m, n = len(matrix), len(matrix[0])
    top, bottom, left, right = 0, m - 1, 0, n - 1
    while top <= bottom and left <= right:
        for j in range(left, right + 1):
            result.append(matrix[top][j])
        top += 1
        for i in range(top, bottom + 1):
            result.append(matrix[i][right])
        right -= 1
        if top <= bottom:
            for j in range(right, left -1, -1):
                result.append(matrix[bottom][j])
            bottom -= 1
        if left <= right:
            for i in range(bottom, top -1, -1):
                result.append(matrix[i][left])
            left += 1
    return result
```