

# Python ICPC Cheatsheet

Comprehensive Reference for Competitive Programming

## Contents

		7.4.1 Kruskal's Algorithm . . . . .	10
		7.4.2 Prim's Algorithm . . . . .	11
<b>1</b>	<b>Input/Output</b>	<b>2</b>	
<b>2</b>	<b>Basic Data Structures</b>	<b>2</b>	
2.1	List Operations . . . . .	2	
2.2	Deque (Double-ended Queue) . . . . .	2	
2.3	Heap (Priority Queue) . . . . .	3	
2.4	Dictionary & Counter . . . . .	3	
2.5	Set Operations . . . . .	3	
<b>3</b>	<b>String Operations</b>	<b>3</b>	
3.1	KMP Pattern Matching . . . . .	3	
3.2	Z-Algorithm . . . . .	4	
3.3	Rabin-Karp (Rolling Hash) . . . . .	4	
<b>4</b>	<b>Mathematics</b>	<b>5</b>	
4.1	Basic Math Operations . . . . .	5	
4.2	Combinatorics . . . . .	5	
<b>5</b>	<b>Number Theory</b>	<b>5</b>	
5.1	Modular Arithmetic . . . . .	5	
5.2	Sieve of Eratosthenes . . . . .	5	
5.3	Prime Factorization . . . . .	5	
5.4	Chinese Remainder Theorem . . . . .	6	
5.5	Euler's Totient Function . . . . .	6	
5.6	Fast Exponentiation with Matrix . . . . .	6	
<b>6</b>	<b>Graph Algorithms</b>	<b>6</b>	
6.1	Graph Representation . . . . .	6	
6.2	BFS (Breadth-First Search) . . . . .	7	
6.3	DFS (Depth-First Search) . . . . .	7	
6.4	Strongly Connected Components (SCC) . . . . .	8	
6.5	Bridges and Articulation Points . . . . .	8	
6.6	Lowest Common Ancestor (LCA) . . . . .	9	
<b>7</b>	<b>Shortest Path Algorithms</b>	<b>9</b>	
7.1	Dijkstra's Algorithm . . . . .	9	
7.2	Bellman-Ford Algorithm . . . . .	10	
7.3	Floyd-Warshall Algorithm . . . . .	10	
7.4	Minimum Spanning Tree . . . . .	10	
		8 Topological Sort . . . . .	11
		8.1 Kahn's Algorithm (BFS-based) . . . . .	11
		8.2 DFS-based Topological Sort . . . . .	11
		9 Union-Find (Disjoint Set Union) . . . . .	11
		10 Binary Search . . . . .	12
		10.1 Template for Finding First/Last Position . . . . .	12
		11 Dynamic Programming . . . . .	13
		11.1 Longest Increasing Subsequence . . . . .	13
		11.2 0/1 Knapsack . . . . .	13
		11.3 Edit Distance (Levenshtein Distance) . . . . .	13
		11.4 Longest Common Subsequence (LCS) . . . . .	14
		11.5 Coin Change . . . . .	14
		11.6 Palindrome Partitioning . . . . .	14
		11.7 Subset Sum . . . . .	15
		12 Array Techniques . . . . .	15
		12.1 Prefix Sum . . . . .	15
		12.2 Difference Array . . . . .	15
		12.3 Sliding Window . . . . .	15
		13 Advanced Data Structures . . . . .	16
		13.1 Segment Tree . . . . .	16
		13.2 Fenwick Tree (Binary Indexed Tree) . . . . .	16
		13.3 Trie (Prefix Tree) . . . . .	17
		14 Bit Manipulation . . . . .	17
		15 Matrix Operations . . . . .	18
		15.1 Matrix Multiplication . . . . .	18
		15.2 Matrix Exponentiation . . . . .	18
		16 Miscellaneous Tips . . . . .	19
		16.1 Python-Specific Optimizations . . . . .	19

16.2 Useful Libraries . . . . .	19	17.3 Point in Polygon . . . . .	21
16.3 Common Patterns . . . . .	19	17.4 Closest Pair of Points . . . . .	21
16.4 Common Pitfalls . . . . .	19		
16.5 Time Complexity Reference . . . . .	20	<b>18 Network Flow</b>	<b>21</b>
<b>17 Computational Geometry</b>	<b>20</b>	18.1 Maximum Flow - Edmonds-Karp (BFS-based Ford-Fulkerson) . . . . .	21
17.1 Basic Geometry . . . . .	20	18.2 Dinic's Algorithm (Faster)	22
17.2 Convex Hull . . . . .	20	18.3 Min Cut . . . . .	23
		18.4 Bipartite Matching . . . . .	23

## 1 Input/Output

**Description:** Efficient input/output is crucial in competitive programming, especially for problems with large datasets. Using `sys.stdin.readline` is significantly faster than the default `input()` function.

```

1 # Fast I/O - Essential for large inputs
2 import sys
3 input = sys.stdin.readline
4
5 # Read single integer
6 n = int(input())
7
8 # Read multiple integers on one line
9 a, b = map(int, input().split())
10
11 # Read array of integers
12 arr = list(map(int, input().split()))
13
14 # Read strings (strip to remove trailing
15 # newline)
16 s = input().strip()
17 words = input().split()
18
19 # Multiple test cases pattern
20 t = int(input())
21 for _ in range(t):
22     # process each test case
23
24 # Print without newline
25 print(x, end=' ')
26
27 # Formatted output with precision
28 print(f"{x:.6f}") # 6 decimal places

```

## 2 Basic Data Structures

### 2.1 List Operations

**Description:** Python lists are dynamic arrays with  $O(1)$  amortized append and  $O(n)$  insert/delete at arbitrary positions.

```

1 # Initialize lists
2 arr = [0] * n # n zeros
3 matrix = [[0] * m for _ in range(n)] #
4 # Correct way!
5
6 # List comprehension - concise and
7 # efficient
8 squares = [x**2 for x in range(n)]

```

```

7 evens = [x for x in arr if x % 2 == 0]
8
9 # Sorting - O(n log n)
10 arr.sort() # in-place, modifies arr
11 arr.sort(reverse=True) # descending
12 arr.sort(key=lambda x: (x[0], -x[1])) #
13 # custom
14 sorted_arr = sorted(arr) # returns new
15 # list
16
17 # Binary search in sorted array
18 from bisect import bisect_left,
19 # bisect_right
20 idx = bisect_left(arr, x) # leftmost
21 # position
22 idx = bisect_right(arr, x) # rightmost
23 # position
24
25 # Common operations
26 arr.append(x) # O(1) amortized
27 arr.pop() # O(1) - remove last
28 arr.pop(0) # O(n) - remove first
29 # (slow!)
30 arr.reverse() # O(n) - in-place
31 arr.count(x) # O(n) - count
32 # occurrences
33 arr.index(x) # O(n) - first
34 # occurrence

```

### 2.2 Deque (Double-ended Queue)

**Description:** Deque provides  $O(1)$  append and pop from both ends, making it ideal for sliding window problems and implementing queues/stacks efficiently.

```

1 from collections import deque
2 dq = deque()
3
4 # O(1) operations on both ends
5 dq.append(x) # add to right
6 dq.appendleft(x) # add to left
7 dq.pop() # remove from right
8 dq.popleft() # remove from left
9
10 # Sliding window maximum - O(n)
11 # Maintains decreasing order of elements
12 def sliding_max(arr, k):
13     dq = deque() # stores indices
14     result = []
15
16     for i in range(len(arr)):
17         # Remove indices outside window
18         while dq and dq[0] < i - k + 1:
19             dq.popleft()

```

```

20     # Remove smaller elements (not
21     useful)
22     while dq and arr[dq[-1]] < arr[i
23 ]:
24         dq.pop()
25
26     dq.append(i)
27     if i >= k - 1:
28         result.append(arr[dq[0]])
29
30     return result

```

## 2.3 Heap (Priority Queue)

**Description:** Python's `heapq` implements a min-heap. For max-heap, negate values. Useful for finding k-th largest/smallest, Dijkstra's algorithm, and scheduling problems.

```

1  import heapq
2
3  # Min heap (default)
4  heap = []
5  heapq.heappush(heap, x)      # O(log n)
6
7  min_val = heapq.heappop(heap) # O(log n)
8
9  # Max heap - negate values
10  heapq.heappush(heap, -x)
11  max_val = -heapq.heappop(heap)
12
13  # Convert list to heap in-place - O(n)
14  heapq.heapify(arr)
15
16  # K largest/smallest - O(n log k)
17  k_largest = heapq.nlargest(k, arr)
18  k_smallest = heapq.nsmallest(k, arr)
19
20  # Custom comparator using tuples
21  # Compares first element, then second,
22  # etc.
23  heapq.heappush(heap, (priority, item))

```

## 2.4 Dictionary & Counter

**Description:** Hash maps with  $O(1)$  average case insert/lookup. Counter is specialized for counting occurrences.

```

1  from collections import defaultdict,
2      Counter
3
4  # defaultdict - provides default value
5  graph = defaultdict(list) # empty list
6      default
7
8  count = defaultdict(int) # 0 default
9
10 # Counter - count elements efficiently
11 cnt = Counter(arr)
12 cnt['x'] += 1
13 most_common = cnt.most_common(k) # k
14     most frequent
15
16 # Dictionary operations
17 d = {}
18 d.get(key, default_val)

```

```

15 d.setdefault(key, default_val)
16 for k, v in d.items():
17     pass

```

## 2.5 Set Operations

**Description:** Hash sets provide  $O(1)$  membership testing and set operations.

```

1  s = set()
2  s.add(x)      # O(1)
3  s.remove(x)   # O(1), KeyError if not
4      exists
5  s.discard(x)  # O(1), no error if not
6      exists
7
8  # Set operations - all O(n)
9  a | b         # union
10 a & b         # intersection
11 a - b         # difference
12 a ^ b         # symmetric difference
13
14 # Ordered set workaround
15 from collections import OrderedDict
16 oset = OrderedDict.fromkeys([])

```

## 3 String Operations

**Description:** Strings in Python are immutable. For building strings, use `list` and `join` for  $O(n)$  complexity instead of repeated concatenation which is  $O(n^2)$ .

```

1  # Common string methods
2  s.lower(), s.upper()
3  s.strip() # remove whitespace both
4      ends
5  s.lstrip() # remove left whitespace
6  s.rstrip() # remove right whitespace
7  s.split(delimiter)
8  delimiter.join(list)
9  s.replace(old, new)
10 s.startswith(prefix)
11 s.endswith(suffix)
12 s.isdigit(), s.isalpha(), s.isalnum()
13
14 # String building - EFFICIENT O(n)
15 result = []
16 for x in data:
17     result.append(str(x))
18 s = ''.join(result)
19
20 # String concatenation - SLOW O(n^2)
21 # s = ""
22 # for x in data:
23 #     s += str(x) # Don't do this!
24
25 # ASCII values
26 ord('a') # 97
27 chr(97) # 'a'
28
29 # String to character array (for
30     mutations)
31 chars = list(s)
32 chars[0] = 'x'
33 s = ''.join(chars)

```

### 3.1 KMP Pattern Matching

**Description:** Find all occurrences of pattern in text. Time:  $O(n+m)$ .

```

1 def kmp_search(text, pattern):
2     # Build LPS (Longest Proper Prefix
3     # which is Suffix)
4     def build_lps(pattern):
5         m = len(pattern)
6         lps = [0] * m
7         length = 0 # Length of previous
8         # longest prefix
9         i = 1
10
11         while i < m:
12             if pattern[i] == pattern[
13                 length]:
14                 length += 1
15                 lps[i] = length
16                 i += 1
17             else:
18                 if length != 0:
19                     length = lps[length
20                     - 1]
21                 else:
22                     lps[i] = 0
23                     i += 1
24
25         return lps
26
27     n, m = len(text), len(pattern)
28     lps = build_lps(pattern)
29
30     matches = []
31     i = j = 0 # Indices for text and
32     # pattern
33
34     while i < n:
35         if text[i] == pattern[j]:
36             i += 1
37             j += 1
38
39         if j == m:
40             matches.append(i - j)
41             j = lps[j - 1]
42         elif i < n and text[i] !=
43             pattern[j]:
44             if j != 0:
45                 j = lps[j - 1]
46             else:
47                 i += 1
48
49     return matches

```

### 3.2 Z-Algorithm

**Description:** Compute Z-array where  $Z[i]$  = length of longest substring starting from  $i$  that matches prefix. Time:  $O(n)$ .

```

1 def z_algorithm(s):
2     n = len(s)
3     z = [0] * n
4     l, r = 0, 0
5
6     for i in range(1, n):
7         if i <= r:
8             z[i] = min(r - i + 1, z[i -
9             1])
10
11         while i + z[i] < n and s[z[i]]
12             == s[i + z[i]]:
13             z[i] += 1
14
15         if i + z[i] - 1 > r:

```

```

14         l, r = i, i + z[i] - 1
15
16     return z
17
18 # Pattern matching using Z-algorithm
19 def z_search(text, pattern):
20     # Concatenate pattern + $ + text
21     s = pattern + '$' + text
22     z = z_algorithm(s)
23
24     matches = []
25     m = len(pattern)
26
27     for i in range(m + 1, len(s)):
28         if z[i] == m:
29             matches.append(i - m - 1)
30
31     return matches

```

### 3.3 Rabin-Karp (Rolling Hash)

**Description:** Fast pattern matching using hashing. Average:  $O(n+m)$ , Worst:  $O(nm)$ .

```

1 def rabin_karp(text, pattern):
2     MOD = 10**9 + 7
3     BASE = 31 # Prime base for hashing
4
5     n, m = len(text), len(pattern)
6     if m > n:
7         return []
8
9     # Compute hash of pattern
10    pattern_hash = 0
11    power = 1
12    for i in range(m):
13        pattern_hash = (pattern_hash *
14        BASE +
15        ord(pattern[i]))
16        % MOD
17        if i < m - 1:
18            power = (power * BASE) % MOD
19
20    # Rolling hash
21    text_hash = 0
22    matches = []
23
24    for i in range(n):
25        # Add new character
26        text_hash = (text_hash * BASE +
27        ord(text[i])) % MOD
28
29        # Remove old character if window
30        # full
31        if i >= m:
32            text_hash = (text_hash -
33            ord(text[i - m])
34            * power) % MOD
35            text_hash = (text_hash + MOD
36            ) % MOD
37
38        # Check match
39        if i >= m - 1 and text_hash ==
40            pattern_hash:
41            # Verify actual match (avoid
42            # hash collision)
43            if text[i - m + 1:i + 1] ==
44                pattern:
45                matches.append(i - m +
46                1)

```

```
39     return matches
```

## 4 Mathematics

### 4.1 Basic Math Operations

```
1 import math
2
3 # Common functions
4 math.ceil(x), math.floor(x)
5 math.gcd(a, b) # Greatest common
   divisor
6 math.lcm(a, b) # Python 3.9+
7 math.sqrt(x)
8 math.log(x), math.log2(x), math.log10(x)
9
10 # Powers
11 x ** y
12 pow(x, y, mod) # (x^y) % mod -
   efficient modular exp
13
14 # Infinity
15 float('inf'), float('-inf')
16
17 # Custom GCD using Euclidean algorithm -
   O(log min(a,b))
18 def gcd(a, b):
19     while b:
20         a, b = b, a % b
21     return a
22
23 def lcm(a, b):
24     return a * b // gcd(a, b)
```

### 4.2 Combinatorics

**Description:** Compute combinations and permutations. For modular arithmetic, compute factorial arrays and use modular inverse.

```
1 from math import factorial, comb, perm
2
3 # nCr (combinations) - "n choose r"
4 comb(n, r) # Built-in Python 3.8+
5
6 # nPr (permutations)
7 perm(n, r) # Built-in Python 3.8+
8
9 # Manual nCr implementation
10 def ncr(n, r):
11     if r > n: return 0
12     r = min(r, n - r) # Optimization: C
   (n,r) = C(n,n-r)
13     num = den = 1
14     for i in range(r):
15         num *= (n - i)
16         den *= (i + 1)
17     return num // den
18
19 # Precompute factorials with modulo
20 MOD = 10**9 + 7
21 def modfact(n):
22     fact = [1] * (n + 1)
23     for i in range(1, n + 1):
24         fact[i] = fact[i-1] * i % MOD
25     return fact
26
27 # Modular combination using precomputed
   factorials
28 # First precompute inverse factorials
29 def compute_inv_factorials(n, mod):
```

```
30     fact = modfact(n)
31     inv_fact = [1] * (n + 1)
32     inv_fact[n] = pow(fact[n], mod - 2,
   mod)
33     for i in range(n - 1, -1, -1):
34         inv_fact[i] = inv_fact[i + 1] *
   (i + 1) % mod
35     return fact, inv_fact
36
37 def modcomb(n, r, fact, inv_fact, mod):
38     if r > n or r < 0: return 0
39     return fact[n] * inv_fact[r] % mod *
   inv_fact[n-r] % mod
```

## 5 Number Theory

**Description:** Essential algorithms for problems involving primes, modular arithmetic, and divisibility.

### 5.1 Modular Arithmetic

```
1 # Modular inverse using Fermat's Little
   Theorem
2 # Only works when mod is prime
3 # a^(-1) = a^(mod-2) (mod p)
4 def modinv(a, mod):
5     return pow(a, mod - 2, mod)
6
7 # Extended Euclidean Algorithm
8 # Returns (gcd, x, y) where ax + by =
   gcd(a,b)
9 # Can find modular inverse for any
   coprime a,mod
10 def extgcd(a, b):
11     if b == 0:
12         return a, 1, 0
13     g, x1, y1 = extgcd(b, a % b)
14     x = y1
15     y = x1 - (a // b) * y1
16     return g, x, y
```

### 5.2 Sieve of Eratosthenes

**Description:** Find all primes up to n in  $O(n \log \log n)$  time. Memory:  $O(n)$ .

```
1 def sieve(n):
2     is_prime = [True] * (n + 1)
3     is_prime[0] = is_prime[1] = False
4
5     for i in range(2, int(n**0.5) + 1):
6         if is_prime[i]:
7             # Mark multiples as
   composite
8             for j in range(i*i, n + 1, i
   ):
9                 is_prime[j] = False
10
11     return is_prime
12
13 # Get list of primes
14 primes = [i for i in range(n+1) if
   is_prime[i]]
```

### 5.3 Prime Factorization

**Description:** Decompose n into prime factors in  $O(\sqrt{n})$  time.

```
1 def factorize(n):
```

```

2 factors = []
3 d = 2
4
5 # Check divisors up to sqrt(n)
6 while d * d <= n:
7     while n % d == 0:
8         factors.append(d)
9         n //= d
10        d += 1
11
12 # If n > 1, it's a prime factor
13 if n > 1:
14     factors.append(n)
15
16 return factors
17
18 # Get prime factors with counts
19 from collections import Counter
20 def prime_factor_counts(n):
21     return Counter(factorize(n))
22
23 # Count divisors
24 def count_divisors(n):
25     count = 0
26     i = 1
27     while i * i <= n:
28         if n % i == 0:
29             count += 1 if i * i == n
30             else 2
31             i += 1
32     return count
33
34 # Sum of divisors
35 def sum_divisors(n):
36     total = 0
37     i = 1
38     while i * i <= n:
39         if n % i == 0:
40             total += i
41             if i != n // i:
42                 total += n // i
43             i += 1
44     return total

```

## 5.4 Chinese Remainder Theorem

**Description:** Solve system of congruences  $x \equiv a_1 \pmod{m_1}$ ,  $x \equiv a_2 \pmod{m_2}$ , ... Time:  $O(n \log M)$  where  $M$  is product of moduli.

```

1 def chinese_remainder(remainders, moduli):
2     # Solve x = remainders[i] (mod moduli[i])
3     # Assumes moduli are pairwise coprime
4
5     def extgcd(a, b):
6         if b == 0:
7             return a, 1, 0
8         g, x1, y1 = extgcd(b, a % b)
9         return g, y1, x1 - (a // b) * y1
10
11    total = 0
12    prod = 1
13    for m in moduli:
14        prod *= m
15
16    for r, m in zip(remainders, moduli):
17        p = prod // m

```

```

18        _, inv, _ = extgcd(p, m)
19        total += r * inv * p
20
21    return total % prod

```

## 5.5 Euler's Totient Function

**Description:**  $\phi(n)$  = count of numbers  $\leq n$  coprime to  $n$ . Time:  $O(\sqrt{n})$ .

```

1 def euler_phi(n):
2     result = n
3     p = 2
4
5     while p * p <= n:
6         if n % p == 0:
7             # Remove factor p
8             while n % p == 0:
9                 n //= p
10            # Multiply by (1 - 1/p)
11            result -= result // p
12            p += 1
13
14    if n > 1:
15        result -= result // n
16
17    return result
18
19 # Phi for range [1, n] using sieve
20 def phi_sieve(n):
21     phi = list(range(n + 1)) # phi[i] = i initially
22
23    for i in range(2, n + 1):
24        if phi[i] == i: # i is prime
25            for j in range(i, n + 1, i):
26                phi[j] = phi[j] // i * (i - 1)
27
28    return phi

```

## 5.6 Fast Exponentiation with Matrix

**Description:** Already covered in matrix section, but useful pattern.

```

1 # Modular exponentiation
2 def mod_exp(base, exp, mod):
3     result = 1
4     base %= mod
5
6     while exp > 0:
7         if exp & 1:
8             result = (result * base) % mod
9         base = (base * base) % mod
10        exp >>= 1
11
12    return result

```

## 6 Graph Algorithms

### 6.1 Graph Representation

**Description:** Adjacency list is most common for sparse graphs. Use defaultdict for convenience.

```

1 from collections import defaultdict, deque

```

```

2
3 # Unweighted graph
4 graph = defaultdict(list)
5 for _ in range(m):
6     u, v = map(int, input().split())
7     graph[u].append(v)
8     graph[v].append(u) # for undirected
9
10 # Weighted graph - store (neighbor,
11    weight) tuples
12 graph[u].append((v, weight))

```

## 6.2 BFS (Breadth-First Search)

**Description:** Explores graph level by level. Finds shortest path in unweighted graphs. Time:  $O(V+E)$ , Space:  $O(V)$ .

```

1 def bfs(graph, start):
2     visited = set([start])
3     queue = deque([start])
4     dist = {start: 0}
5
6     while queue:
7         node = queue.popleft()
8
9         for neighbor in graph[node]:
10             if neighbor not in visited:
11                 visited.add(neighbor)
12                 queue.append(neighbor)
13                 dist[neighbor] = dist[
14                     node] + 1
15
16     return dist
17
18 # Grid BFS - common in maze/path
19    problems
20 def grid_bfs(grid, start):
21     n, m = len(grid), len(grid[0])
22     visited = [[False] * m for _ in
23                range(n)]
24     queue = deque([start])
25     visited[start[0]][start[1]] = True
26
27     # 4 directions: right, down, left,
28    up
29     dirs = [(0,1), (1,0), (0,-1), (-1,0)]
30
31     while queue:
32         x, y = queue.popleft()
33
34         for dx, dy in dirs:
35             nx, ny = x + dx, y + dy
36
37             # Check bounds and validity
38             if (0 <= nx < n and 0 <= ny
39                < m
40                and not visited[nx][ny]
41                and grid[nx][ny] != '#'):
42
43                 visited[nx][ny] = True
44                 queue.append((nx, ny))

```

## 6.3 DFS (Depth-First Search)

**Description:** Explores as far as possible along each branch. Used for connectivity, cycles, topological sort. Time:

$O(V+E)$ , Space:  $O(V)$ .

```

1 # Recursive DFS
2 def dfs(graph, node, visited):
3     visited.add(node)
4
5     for neighbor in graph[node]:
6         if neighbor not in visited:
7             dfs(graph, neighbor, visited)
8
9 # Iterative DFS using stack
10 def dfs_iterative(graph, start):
11     visited = set()
12     stack = [start]
13
14     while stack:
15         node = stack.pop()
16
17         if node not in visited:
18             visited.add(node)
19
20             for neighbor in graph[node]:
21                 if neighbor not in
22                    visited:
23                     stack.append(
24                         neighbor)
25
26 # Cycle detection in undirected graph
27 def has_cycle(graph, n):
28     visited = [False] * n
29
30     def dfs(node, parent):
31         visited[node] = True
32
33         for neighbor in graph[node]:
34             if not visited[neighbor]:
35                 if dfs(neighbor, node):
36                     return True
37
38             # Back edge to non-parent =
39            cycle
40             elif neighbor != parent:
41                 return True
42
43     return False
44
45 # Check all components
46 for i in range(n):
47     if not visited[i]:
48         if dfs(i, -1):
49             return True
50
51 return False
52
53 # Cycle detection in directed graph
54 def has_cycle_directed(graph, n):
55     WHITE, GRAY, BLACK = 0, 1, 2
56     color = [WHITE] * n
57
58     def dfs(node):
59         color[node] = GRAY
60
61         for neighbor in graph[node]:
62             if color[neighbor] == GRAY:
63                 return True # Back edge
64             elif color[neighbor] == WHITE:
65                 if dfs(neighbor):
66                     return True
67
68         color[node] = BLACK
69         return False
70
71 for i in range(n):
72     if color[i] == WHITE:

```

```

69         if dfs(i):
70             return True
71     return False
72
73     # Connected components count
74     def count_components(graph, n):
75         visited = [False] * n
76         count = 0
77
78         def dfs(node):
79             visited[node] = True
80             for neighbor in graph[node]:
81                 if not visited[neighbor]:
82                     dfs(neighbor)
83
84         for i in range(n):
85             if not visited[i]:
86                 dfs(i)
87                 count += 1
88
89         return count
90
91     # Bipartite check (2-coloring)
92     def is_bipartite(graph, n):
93         color = [-1] * n
94
95         def bfs(start):
96             from collections import deque
97             queue = deque([start])
98             color[start] = 0
99
100             while queue:
101                 node = queue.popleft()
102
103                 for neighbor in graph[node]:
104                     if color[neighbor] ==
105                         -1:
106                             color[neighbor] = 1
107                             - color[node]
108                             queue.append(
109                                 neighbor)
110                             elif color[neighbor] ==
111                                 color[node]:
112                                     return False
113
114             return True
115
116         for i in range(n):
117             if color[i] == -1:
118                 if not bfs(i):
119                     return False
120
121         return True

```

## 6.4 Strongly Connected Components (SCC)

**Description:** Find all SCCs in directed graph using Tarjan's algorithm. Time:  $O(V+E)$ .

```

1 def tarjan_scc(graph, n):
2     index_counter = [0]
3     stack = []
4     lowlink = [0] * n
5     index = [0] * n
6     on_stack = [False] * n
7     index_initialized = [False] * n
8     sccs = []
9
10    def strongconnect(v):
11        index[v] = index_counter[0]
12        lowlink[v] = index_counter[0]

```

```

13        index_counter[0] += 1
14        index_initialized[v] = True
15        stack.append(v)
16        on_stack[v] = True
17
18        for w in graph[v]:
19            if not index_initialized[w]:
20                strongconnect(w)
21                lowlink[v] = min(lowlink
22                                [v], lowlink[w])
23            elif on_stack[w]:
24                lowlink[v] = min(lowlink
25                                [v], index[w])
26
27        if lowlink[v] == index[v]:
28            scc = []
29            while True:
30                w = stack.pop()
31                on_stack[w] = False
32                scc.append(w)
33                if w == v:
34                    break
35            sccs.append(scc)
36
37    for v in range(n):
38        if not index_initialized[v]:
39            strongconnect(v)
40
41    return sccs

```

## 6.5 Bridges and Articulation Points

**Description:** Find critical edges (bridges) and vertices (articulation points). Time:  $O(V+E)$ .

```

1 def find_bridges(graph, n):
2     visited = [False] * n
3     disc = [0] * n
4     low = [0] * n
5     parent = [-1] * n
6     time = [0]
7     bridges = []
8
9     def dfs(u):
10        visited[u] = True
11        disc[u] = low[u] = time[0]
12        time[0] += 1
13
14        for v in graph[u]:
15            if not visited[v]:
16                parent[v] = u
17                dfs(v)
18                low[u] = min(low[u], low
19                            [v])
20
21            # Bridge condition
22            if low[v] > disc[u]:
23                bridges.append((u, v
24                                ))
25
26            elif v != parent[u]:
27                low[u] = min(low[u],
28                            disc[v])
29
30    for i in range(n):
31        if not visited[i]:
32            dfs(i)
33
34    return bridges
35
36 def find_articulation_points(graph, n):
37     visited = [False] * n

```

<pre> 34 disc = [0] * n 35 low = [0] * n 36 parent = [-1] * n 37 time = [0] 38 ap = set() 39 40 def dfs(u): 41     children = 0 42     visited[u] = True 43     disc[u] = low[u] = time[0] 44     time[0] += 1 45 46     for v in graph[u]: 47         if not visited[v]: 48             children += 1 49             parent[v] = u 50             dfs(v) 51             low[u] = min(low[u], low 52 [v]) 53 54             # Articulation point 55             conditions 56             if parent[u] == -1 and 57 children &gt; 1: 58                 ap.add(u) 59                 if parent[u] != -1 and 60 low[v] &gt;= disc[u]: 61                     ap.add(u) 62                 elif v != parent[u]: 63                     low[u] = min(low[u], 64 disc[v]) 65 66 for i in range(n): 67     if not visited[i]: 68         dfs(i) 69 70 return list(ap) </pre>	<pre> 26 != -1: 27     self.parent[i][j] = 28     self.parent[ 29         self.parent[i][j 30 -1]][j-1] 31 32 def lca(self, u, v): 33     # Make u deeper 34     if self.depth[u] &lt; self.depth[v 35 ]: 36         u, v = v, u 37 38     # Bring u to same level as v 39     diff = self.depth[u] - self. 40 depth[v] 41     for i in range(self.LOG): 42         if (diff &gt;&gt; i) &amp; 1: 43             u = self.parent[u][i] 44 45     if u == v: 46         return u 47 48     # Binary search for LCA 49     for i in range(self.LOG - 1, -1, 50 -1): 51         if self.parent[u][i] != self 52 .parent[v][i]: 53             u = self.parent[u][i] 54             v = self.parent[v][i] 55 56     return self.parent[u][0] 57 58 def dist(self, u, v): 59     # Distance between two nodes 60     l = self.lca(u, v) 61     return self.depth[u] + self. 62 depth[v] - 2 * self.depth[l] </pre>
---	---

## 6.6 Lowest Common Ancestor (LCA)

**Description:** Find LCA of two nodes in a tree. Binary lifting preprocessing:  $O(n \log n)$ , Query:  $O(\log n)$ .

```

1 class LCA:
2     def __init__(self, graph, root, n):
3         self.n = n
4         self.LOG = 20 # log2(n) + 1
5         self.parent = [[-1] * self.LOG
6 for _ in range(n)]
7         self.depth = [0] * n
8
9         # DFS to set parent and depth
10        visited = [False] * n
11
12        def dfs(node, par, d):
13            visited[node] = True
14            self.parent[node][0] = par
15            self.depth[node] = d
16
17            for neighbor in graph[node]:
18                if not visited[neighbor]:
19                    dfs(neighbor, node,
20 d + 1)
21
22        dfs(root, -1, 0)
23
24        # Binary lifting preprocessing
25        for j in range(1, self.LOG):
26            for i in range(n):
27                if self.parent[i][j-1]

```

## 7 Shortest Path Algorithms

### 7.1 Dijkstra's Algorithm

**Description:** Finds shortest paths from a source to all vertices in weighted graphs with non-negative edges. Time:  $O((V+E) \log V)$  with heap.

```

1 import heapq
2
3 def dijkstra(graph, start, n):
4     # Initialize distances to infinity
5     dist = [float('inf')] * n
6     dist[start] = 0
7
8     # Min heap: (distance, node)
9     heap = [(0, start)]
10
11    while heap:
12        d, node = heapq.heappop(heap)
13
14        # Skip if already processed with
15        # better distance
16        if d > dist[node]:
17            continue
18
19        # Relax edges
20        for neighbor, weight in graph[
21 node]:
22            new_dist = dist[node] +
23 weight
24            if new_dist < dist[neighbor]

```

```

23         ]:
24             dist[neighbor] =
25                 new_dist
26             heapq.heappush(heap, (
27                 new_dist, neighbor))
28
29     return dist
30
31 # Path reconstruction
32 def dijkstra_with_path(graph, start, n):
33     dist = [float('inf')] * n
34     parent = [-1] * n
35     dist[start] = 0
36     heap = [(0, start)]
37
38     while heap:
39         d, node = heapq.heappop(heap)
40         if d > dist[node]:
41             continue
42
43         for neighbor, weight in graph[
44             node]:
45             new_dist = dist[node] +
46                 weight
47             if new_dist < dist[neighbor]
48             ]:
49                 dist[neighbor] =
50                     new_dist
51                 parent[neighbor] = node
52                 heapq.heappush(heap, (
53                     new_dist, neighbor))
54
55     return dist, parent
56
57 def reconstruct_path(parent, target):
58     path = []
59     while target != -1:
60         path.append(target)
61         target = parent[target]
62     return path[::-1]

```

## 7.2 Bellman-Ford Algorithm

**Description:** Finds shortest paths with negative edges. Detects negative cycles. Time:  $O(VE)$ .

```

1 def bellman_ford(edges, n, start):
2     # edges = [(u, v, weight), ...]
3     dist = [float('inf')] * n
4     dist[start] = 0
5
6     # Relax edges n-1 times
7     for _ in range(n - 1):
8         for u, v, w in edges:
9             if dist[u] != float('inf')
10             and \
11                 dist[u] + w < dist[v]:
12                 dist[v] = dist[u] + w
13
14     # Check for negative cycles
15     for u, v, w in edges:
16         if dist[u] != float('inf') and \
17             dist[u] + w < dist[v]:
18             return None # Negative
19             cycle exists
20
21     return dist

```

## 7.3 Floyd-Warshall Algorithm

**Description:** All-pairs shortest paths. Works with negative edges (no negative cycles). Time:  $O(V^3)$ .

```

1 def floyd_warshall(n, edges):
2     # Initialize distance matrix
3     dist = [[float('inf')] * n for _ in
4             range(n)]
5
6     for i in range(n):
7         dist[i][i] = 0
8
9     for u, v, w in edges:
10         dist[u][v] = min(dist[u][v], w)
11
12     # Dynamic programming
13     for k in range(n): # Intermediate
14         vertex
15         for i in range(n):
16             for j in range(n):
17                 dist[i][j] = min(dist[i]
18                                 ][j],
19                                 dist[i][
20                                 k] + dist[k][j])
21
22     return dist
23
24 # Check for negative cycle
25 def has_negative_cycle(dist, n):
26     for i in range(n):
27         if dist[i][i] < 0:
28             return True
29     return False

```

## 7.4 Minimum Spanning Tree

### 7.4.1 Kruskal's Algorithm

**Description:** MST using Union-Find. Sort edges by weight. Time:  $O(E \log E)$ .

```

1 def kruskal(n, edges):
2     # edges = [(weight, u, v), ...]
3     edges.sort() # Sort by weight
4
5     uf = UnionFind(n)
6     mst_weight = 0
7     mst_edges = []
8
9     for weight, u, v in edges:
10         if uf.union(u, v):
11             mst_weight += weight
12             mst_edges.append((u, v,
13                               weight))
14
15     return mst_weight, mst_edges
16
17 class UnionFind:
18     def __init__(self, n):
19         self.parent = list(range(n))
20         self.rank = [0] * n
21
22     def find(self, x):
23         if self.parent[x] != x:
24             self.parent[x] = self.find(
25                 self.parent[x])
26         return self.parent[x]
27
28     def union(self, x, y):
29         px, py = self.find(x), self.find

```

```

28     (y)
29     if px == py:
30         return False
31     if self.rank[px] < self.rank[py]:
32         px, py = py, px
33         self.parent[py] = px
34         if self.rank[px] == self.rank[py]:
35             self.rank[px] += 1
36     return True

```

## 7.4.2 Prim's Algorithm

**Description:** MST using heap. Good for dense graphs. Time:  $O(E \log V)$ .

```

1 import heapq
2
3 def prim(graph, n):
4     # graph[u] = [(v, weight), ...]
5     visited = [False] * n
6     min_heap = [(0, 0)] # (weight, node)
7
8     mst_weight = 0
9
10    while min_heap:
11        weight, u = heapq.heappop(min_heap)
12
13        if visited[u]:
14            continue
15
16        visited[u] = True
17        mst_weight += weight
18
19        for v, w in graph[u]:
20            if not visited[v]:
21                heapq.heappush(min_heap, (w, v))
22
23    return mst_weight

```

## 8 Topological Sort

**Description:** Linear ordering of vertices in a DAG (Directed Acyclic Graph) such that for every edge  $u \rightarrow v$ ,  $u$  comes before  $v$ . Used for task scheduling, course prerequisites, build systems. Time:  $O(V+E)$ .

### 8.1 Kahn's Algorithm (BFS-based)

**Advantages:** Detects cycles, can process nodes level by level.

```

1 from collections import deque
2
3 def topo_sort(graph, n):
4     # Count incoming edges for each node
5     indegree = [0] * n
6     for u in range(n):
7         for v in graph[u]:
8             indegree[v] += 1
9
10    # Start with nodes having no dependencies

```

```

11    queue = deque([i for i in range(n)
12                  if indegree[i] == 0])
13    result = []
14
15    while queue:
16        node = queue.popleft()
17        result.append(node)
18
19        # Remove this node from graph
20        for neighbor in graph[node]:
21            indegree[neighbor] -= 1
22
23        # If neighbor has no more dependencies
24        if indegree[neighbor] == 0:
25            queue.append(neighbor)
26
27    # If not all nodes processed, cycle exists
28    return result if len(result) == n else []

```

## 8.2 DFS-based Topological Sort

**Advantages:** Simpler code, uses less space.

```

1 def topo_dfs(graph, n):
2     visited = [False] * n
3     stack = []
4
5     def dfs(node):
6         visited[node] = True
7
8         # Visit all neighbors first
9         for neighbor in graph[node]:
10            if not visited[neighbor]:
11                dfs(neighbor)
12
13        # Add to stack after visiting all descendants
14        stack.append(node)
15
16    # Process all components
17    for i in range(n):
18        if not visited[i]:
19            dfs(i)
20
21    # Reverse stack gives topological order
22    return stack[::-1]

```

## 9 Union-Find (Disjoint Set Union)

**Description:** Efficiently tracks disjoint sets and supports union and find operations. Used for Kruskal's MST, connected components, cycle detection. Time:  $O(\alpha(n)) \approx O(1)$  per operation with path compression and union by rank.

**Applications:**

- Kruskal's minimum spanning tree
- Detecting cycles in undirected graphs
- Finding connected components

- Network connectivity problems

```

1 class UnionFind:
2     def __init__(self, n):
3         # Each node is its own parent
4         # initially
5         self.parent = list(range(n))
6         # Rank for union by rank
7         # optimization
8         self.rank = [0] * n
9
10    def find(self, x):
11        # Path compression: point
12        # directly to root
13        if self.parent[x] != x:
14            self.parent[x] = self.find(
15                self.parent[x])
16        return self.parent[x]
17
18    def union(self, x, y):
19        # Find roots
20        px, py = self.find(x), self.find(
21            y)
22
23        # Already in same set
24        if px == py:
25            return False
26
27        # Union by rank: attach smaller
28        # tree under larger
29        if self.rank[px] < self.rank[py]:
30            px, py = py, px
31
32        self.parent[py] = px
33
34        # Increase rank if trees had
35        # equal rank
36        if self.rank[px] == self.rank[py]:
37            self.rank[px] += 1
38
39        return True
40
41    def connected(self, x, y):
42        return self.find(x) == self.find(
43            y)
44
45    # Count number of disjoint sets
46    def count_sets(self):
47        return len(set(self.find(i)
48            for i in range(len(
49                self.parent))))
50
51    # Example: Detect cycle in undirected
52    # graph
53    def has_cycle_uf(edges, n):
54        uf = UnionFind(n)
55        for u, v in edges:
56            if uf.connected(u, v):
57                return True # Cycle found
58            uf.union(u, v)
59        return False

```

## 10 Binary Search

**Description:** Search in  $O(\log n)$  time. Works on monotonic functions (sorted arrays, or functions where condition transitions from false to true exactly once).

### 10.1 Template for Finding First/Last Position

```

1 # Find FIRST position where check(mid)
2 # is True
3 def binary_search_first(left, right,
4     check):
5     while left < right:
6         mid = (left + right) // 2
7
8         if check(mid):
9             right = mid # Could be
10            answer, search left
11        else:
12            left = mid + 1 # Not answer
13            , search right
14
15    return left
16
17 # Find LAST position where check(mid) is
18 # True
19 def binary_search_last(left, right,
20     check):
21     while left < right:
22         mid = (left + right + 1) // 2 #
23         Round up!
24
25         if check(mid):
26             left = mid # Could be
27            answer, search right
28        else:
29            right = mid - 1 # Not
30            answer, search left
31
32    return left
33
34 # Example: Integer square root
35 def sqrt_binary(n):
36     left, right = 0, n
37
38     while left < right:
39         mid = (left + right + 1) // 2
40
41         if mid * mid <= n:
42             left = mid # mid might be
43            answer
44        else:
45            right = mid - 1
46
47    return left
48
49 # Binary search on answer - common
50 # pattern
51 def min_days_to_make_bouquets(bloomDay,
52     m, k):
53     # Can we make m bouquets in 'days'
54     # days?
55     def can_make(days):
56         bouquets = consecutive = 0
57         for bloom in bloomDay:
58             if bloom <= days:
59                 consecutive += 1
60                 if consecutive == k:
61                     bouquets += 1
62                     consecutive = 0
63             else:
64                 consecutive = 0
65         return bouquets >= m
66
67     if len(bloomDay) < m * k:
68         return -1
69
70     # Binary search on number of days
71     return binary_search_first(

```

```

59 min(bloomDay), max(bloomDay),
   can_make)

```

## 11 Dynamic Programming

**Description:** Solve problems by breaking them into overlapping sub-problems. Store results to avoid recomputation.

### 11.1 Longest Increasing Subsequence

**Description:** Find length of longest strictly increasing subsequence. Time:  $O(n \log n)$  using binary search.

```

1 def lis(arr):
2     from bisect import bisect_left
3
4     # dp[i] = smallest ending value of
5     # LIS of length i+1
6     dp = []
7
8     for x in arr:
9         # Find position to place x
10        idx = bisect_left(dp, x)
11
12        if idx == len(dp):
13            dp.append(x) # Extend LIS
14        else:
15            dp[idx] = x # Better
16                           ending for this length
17
18    return len(dp)
19
20 # LIS with actual sequence
21 def lis_with_sequence(arr):
22     from bisect import bisect_left
23
24     n = len(arr)
25     dp = []
26     parent = [-1] * n
27     dp_idx = [] # indices in dp
28
29     for i, x in enumerate(arr):
30         idx = bisect_left(dp, x)
31
32         if idx == len(dp):
33             dp.append(x)
34             dp_idx.append(i)
35         else:
36             dp[idx] = x
37             dp_idx[idx] = i
38
39     if idx > 0:
40         parent[i] = dp_idx[idx - 1]
41
42     # Reconstruct sequence
43     result = []
44     idx = dp_idx[-1]
45     while idx != -1:
46         result.append(arr[idx])
47         idx = parent[idx]
48
49     return result[::-1]

```

### 11.2 0/1 Knapsack

**Description:** Maximum value with weight capacity. Each item can be taken 0 or 1 time. Time:  $O(n \times \text{capacity})$ , Space:  $O(n \times \text{capacity})$ .

```

1 def knapsack(weights, values, capacity):
2     n = len(weights)
3     # dp[i][w] = max value using first i
4     # items, weight <= w
5     dp = [[0] * (capacity + 1) for _ in
6           range(n + 1)]
7
8     for i in range(1, n + 1):
9         for w in range(capacity + 1):
10            # Don't take item i-1
11            dp[i][w] = dp[i-1][w]
12
13            # Take item i-1 if it fits
14            if weights[i-1] <= w:
15                dp[i][w] = max(
16                    dp[i][w],
17                    dp[i-1][w - weights[
18                        i-1]] + values[i-1]
19                )
20
21    return dp[n][capacity]
22
23 # Space-optimized O(capacity)
24 def knapsack_optimized(weights, values,
25                          capacity):
26     dp = [0] * (capacity + 1)
27
28     for i in range(len(weights)):
29         # Iterate backwards to avoid
30         # using updated values
31         for w in range(capacity, weights
32                        [i] - 1, -1):
33             dp[w] = max(dp[w],
34                         dp[w - weights[i]
35                        ] + values[i])
36
37    return dp[capacity]

```

### 11.3 Edit Distance (Levenshtein Distance)

**Description:** Minimum operations (insert, delete, replace) to transform  $s_1$  to  $s_2$ . Time:  $O(m \times n)$ , Space:  $O(m \times n)$ .

```

1 def edit_dist(s1, s2):
2     m, n = len(s1), len(s2)
3     # dp[i][j] = edit distance of s1[:i]
4     # and s2[:j]
5     dp = [[0] * (n + 1) for _ in range(m
6           + 1)]
7
8     # Base cases: empty string
9     # transformations
10    for i in range(m + 1):
11        dp[i][0] = i # Delete all
12    for j in range(n + 1):
13        dp[0][j] = j # Insert all
14
15    for i in range(1, m + 1):
16        for j in range(1, n + 1):
17            if s1[i-1] == s2[j-1]:
18                # Characters match, no
19                # operation needed

```

```

16         dp[i][j] = dp[i-1][j-1]
17     else:
18         dp[i][j] = 1 + min(
19             Delete from s1    dp[i-1][j],      #
20             Insert into s1    dp[i][j-1],      #
21             Replace in s1     dp[i-1][j-1]     #
22         )
23
24     return dp[m][n]

```

## 11.4 Longest Common Subsequence (LCS)

**Description:** Longest subsequence common to two sequences. **Time:**  $O(m \times n)$ .

```

1  def lcs(s1, s2):
2      m, n = len(s1), len(s2)
3      dp = [[0] * (n + 1) for _ in range(m + 1)]
4
5      for i in range(1, m + 1):
6          for j in range(1, n + 1):
7              if s1[i-1] == s2[j-1]:
8                  dp[i][j] = dp[i-1][j-1] + 1
9              else:
10                 dp[i][j] = max(dp[i-1][j], dp[i][j-1])
11
12     return dp[m][n]
13
14 # Reconstruct LCS
15 def lcs_string(s1, s2):
16     m, n = len(s1), len(s2)
17     dp = [[0] * (n + 1) for _ in range(m + 1)]
18
19     for i in range(1, m + 1):
20         for j in range(1, n + 1):
21             if s1[i-1] == s2[j-1]:
22                 dp[i][j] = dp[i-1][j-1] + 1
23             else:
24                 dp[i][j] = max(dp[i-1][j], dp[i][j-1])
25
26     # Backtrack
27     result = []
28     i, j = m, n
29     while i > 0 and j > 0:
30         if s1[i-1] == s2[j-1]:
31             result.append(s1[i-1])
32             i -= 1
33             j -= 1
34         elif dp[i-1][j] > dp[i][j-1]:
35             i -= 1
36         else:
37             j -= 1
38
39     return ''.join(reversed(result))

```

## 11.5 Coin Change

**Description:** Minimum coins to make amount, or count ways. **Time:**  $O(n \times \text{amount})$ .

```

1  # Minimum coins
2  def coin_change_min(coins, amount):
3      dp = [float('inf')] * (amount + 1)
4      dp[0] = 0
5
6      for coin in coins:
7          for i in range(coin, amount + 1):
8              :
9              dp[i] = min(dp[i], dp[i - coin] + 1)
10
11     return dp[amount] if dp[amount] != float('inf') else -1
12
13 # Count ways
14 def coin_change_ways(coins, amount):
15     dp = [0] * (amount + 1)
16     dp[0] = 1
17
18     for coin in coins:
19         for i in range(coin, amount + 1):
20             :
21             dp[i] += dp[i - coin]
22
23     return dp[amount]

```

## 11.6 Palindrome Partitioning

**Description:** Minimum cuts to partition string into palindromes. **Time:**  $O(n^2)$ .

```

1  def min_palindrome_partition(s):
2      n = len(s)
3
4      # is_pal[i][j] = True if s[i:j+1] is palindrome
5      is_pal = [[False] * n for _ in range(n)]
6
7      # Every single character is palindrome
8      for i in range(n):
9          is_pal[i][i] = True
10
11     # Check all substrings
12     for length in range(2, n + 1):
13         for i in range(n - length + 1):
14             j = i + length - 1
15             if s[i] == s[j]:
16                 is_pal[i][j] = (length == 2 or
17                                is_pal[i+1][j-1])
18
19     # dp[i] = min cuts for s[0:i+1]
20     dp = [float('inf')] * n
21
22     for i in range(n):
23         if is_pal[0][i]:
24             dp[i] = 0
25         else:
26             for j in range(i):
27                 if is_pal[j+1][i]:
28                     dp[i] = min(dp[i], dp[j] + 1)
29
30     return dp[n-1]

```

## 11.7 Subset Sum

**Description:** Check if subset sums to target. Time:  $O(n \times \text{sum})$ .

```
1 def subset_sum(arr, target):
2     n = len(arr)
3     dp = [[False] * (target + 1) for _
4           in range(n + 1)]
5
6     # Base case: sum 0 is always
7     # achievable
8     for i in range(n + 1):
9         dp[i][0] = True
10
11    for i in range(1, n + 1):
12        for s in range(target + 1):
13            # Don't take arr[i-1]
14            dp[i][s] = dp[i-1][s]
15
16            # Take arr[i-1] if possible
17            if s >= arr[i-1]:
18                dp[i][s] = dp[i][s] or
19                dp[i-1][s - arr[i-1]]
20
21    return dp[n][target]
22
23 # Space optimized
24 def subset_sum_optimized(arr, target):
25     dp = [False] * (target + 1)
26     dp[0] = True
27
28     for num in arr:
29         for s in range(target, num - 1,
30                        -1):
31             dp[s] = dp[s] or dp[s - num]
32
33    return dp[target]
```

```
21 return prefix
```

```
22
23 # Rectangle sum from (x1,y1) to (x2,y2)
24 # inclusive
25 def rect_sum(prefix, x1, y1, x2, y2):
26     return (prefix[x2+1][y2+1] -
27            prefix[x1][y2+1] -
28            prefix[x2+1][y1] +
29            prefix[x1][y1])
```

## 12.2 Difference Array

**Description:** Efficiently perform range updates.  $O(1)$  per update,  $O(n)$  to reconstruct.

```
1 # Initialize difference array
2 diff = [0] * (n + 1)
3
4 # Add 'val' to range [l, r]
5 def range_update(diff, l, r, val):
6     diff[l] += val
7     diff[r + 1] -= val
8
9 # After all updates, reconstruct array
10 def reconstruct(diff):
11     result = []
12     current = 0
13     for i in range(len(diff) - 1):
14         current += diff[i]
15         result.append(current)
16     return result
17
18 # Example: Multiple range updates
19 diff = [0] * (n + 1)
20 for l, r, val in updates:
21     range_update(diff, l, r, val)
22 final_array = reconstruct(diff)
```

## 12 Array Techniques

### 12.1 Prefix Sum

**Description:** Precompute cumulative sums for  $O(1)$  range queries. Time:  $O(n)$  preprocessing,  $O(1)$  query.

```
1 # 1D prefix sum
2 prefix = [0] * (n + 1)
3 for i in range(n):
4     prefix[i + 1] = prefix[i] + arr[i]
5
6 # Range sum query [l, r] inclusive
7 range_sum = prefix[r + 1] - prefix[l]
8
9 # 2D prefix sum - for rectangle sum
10 # queries
11 def build_2d_prefix(matrix):
12     n, m = len(matrix), len(matrix[0])
13     prefix = [[0] * (m + 1) for _ in
14              range(n + 1)]
15
16     for i in range(1, n + 1):
17         for j in range(1, m + 1):
18             prefix[i][j] = (matrix[i-1][
19                             j-1] +
20                             prefix[i-1][j]
21                             +
22                             prefix[i][j-1]
23                             -
24                             prefix[i-1][j-1])
```

### 12.3 Sliding Window

**Description:** Maintain a window of elements while traversing. Time:  $O(n)$ .

```
1 # Fixed size window
2 def max_sum_window(arr, k):
3     window_sum = sum(arr[:k])
4     max_sum = window_sum
5
6     # Slide window: add right, remove
7     # left
8     for i in range(k, len(arr)):
9         window_sum += arr[i] - arr[i - k]
10
11     max_sum = max(max_sum,
12                  window_sum)
13
14    return max_sum
15
16 # Variable size window - two pointers
17 def min_subarray_sum_geq_target(arr,
18                                target):
19     left = 0
20     current_sum = 0
21     min_len = float('inf')
22
23     for right in range(len(arr)):
24         current_sum += arr[right]
25
26         # Shrink window while condition
27         # holds
28         while current_sum >= target:
```

```

24         min_len = min(min_len, right
25             - left + 1)
26         current_sum -= arr[left]
27         left += 1
28     return min_len if min_len != float('
29         inf') else 0
30 # Longest substring with at most k
31     distinct chars
32 def longest_k_distinct(s, k):
33     from collections import defaultdict
34
35     left = 0
36     char_count = defaultdict(int)
37     max_len = 0
38
39     for right in range(len(s)):
40         char_count[s[right]] += 1
41
42         # Shrink if too many distinct
43         while len(char_count) > k:
44             char_count[s[left]] -= 1
45             if char_count[s[left]] == 0:
46                 del char_count[s[left]]
47             left += 1
48
49         max_len = max(max_len, right -
50             left + 1)
51
52     return max_len

```

## 13 Advanced Data Structures

### 13.1 Segment Tree

**Description:** Supports range queries and point updates in  $O(\log n)$ . Can be modified for range updates with lazy propagation.

```

1 class SegmentTree:
2     def __init__(self, arr):
3         self.n = len(arr)
4         # Tree size: 4n is safe upper
5         bound
6         self.tree = [0] * (4 * self.n)
7         self.build(arr, 0, 0, self.n -
8             1)
9
10    def build(self, arr, node, start,
11        end):
12        if start == end:
13            # Leaf node
14            self.tree[node] = arr[start]
15        else:
16            mid = (start + end) // 2
17            # Build left and right
18            subtrees
19            self.build(arr, 2*node+1,
20                start, mid)
21            self.build(arr, 2*node+2,
22                mid+1, end)
23            # Combine results (sum in
24            this case)
25            self.tree[node] = (self.tree
26                [2*node+1] +
27                    self.tree
28                    [2*node+2])
29
30    def update(self, node, start, end,

```

```

22    idx, val):
23        if start == end:
24            # Leaf node - update value
25            self.tree[node] = val
26        else:
27            mid = (start + end) // 2
28            if idx <= mid:
29                # Update left subtree
30                self.update(2*node+1,
31                    start, mid, idx, val)
32            else:
33                # Update right subtree
34                self.update(2*node+2,
35                    mid+1, end, idx, val)
36            # Recompute parent
37            self.tree[node] = (self.tree
38                [2*node+1] +
39                    self.tree
40                    [2*node+2])
41
42    def query(self, node, start, end, l,
43        r):
44        # No overlap
45        if r < start or end < l:
46            return 0
47
48        # Complete overlap
49        if l <= start and end <= r:
50            return self.tree[node]
51
52        # Partial overlap
53        mid = (start + end) // 2
54        left_sum = self.query(2*node+1,
55            start, mid, l, r)
56        right_sum = self.query(2*node+2,
57            mid+1, end, l, r)
58        return left_sum + right_sum
59
60    # Public interface
61    def update_val(self, idx, val):
62        self.update(0, 0, self.n-1, idx,
63            val)
64
65    def range_sum(self, l, r):
66        return self.query(0, 0, self.n
67            -1, l, r)

```

### 13.2 Fenwick Tree (Binary Indexed Tree)

**Description:** Simpler than segment tree, supports prefix sum and point updates in  $O(\log n)$ . More space efficient.

```

1 class FenwickTree:
2     def __init__(self, n):
3         self.n = n
4         # 1-indexed for easier
5         implementation
6         self.tree = [0] * (n + 1)
7
8     def update(self, i, delta):
9         # Add delta to position i (1-
10         indexed)
11         while i <= self.n:
12             self.tree[i] += delta
13             # Move to next node: add LSB
14             i += i & (-i)
15
16     def query(self, i):
17         # Get prefix sum up to i (1-
18         indexed)
19         s = 0

```

```

17         while i > 0:
18             s += self.tree[i]
19             # Move to parent: remove LSB
20             i -= i & (-i)
21         return s
22
23     def range_query(self, l, r):
24         # Sum from l to r (1-indexed)
25         return self.query(r) - self.
26             query(l - 1)
27
28     # Usage example
29     bit = FenwickTree(n)
30     for i, val in enumerate(arr, 1):
31         bit.update(i, val)
32
33     # Range sum [l, r] (1-indexed)
34     result = bit.range_query(l, r)
35
41         if char not in node.children
42         :
43             return []
44         node = node.children[char]
45
46         # DFS to collect all words
47         words = []
48         def dfs(n, path):
49             if n.is_end:
50                 words.append(prefix +
51                     path)
52             for char, child in n.
53                 children.items():
54                 dfs(child, path + char)
55
56         dfs(node, "")
57         return words

```

### 13.3 Trie (Prefix Tree)

**Description:** Tree for storing strings, enables fast prefix searches. Time:  $O(m)$  for operations where  $m$  is string length.

```

1 class TrieNode:
2     def __init__(self):
3         self.children = {} # char ->
4             TrieNode
5         self.is_end = False # End of
6             word marker
7
8 class Trie:
9     def __init__(self):
10         self.root = TrieNode()
11
12     def insert(self, word):
13         # Insert word - O(len(word))
14         node = self.root
15         for char in word:
16             if char not in node.children
17             :
18                 node.children[char] =
19                     TrieNode()
20                 node = node.children[char]
21             node.is_end = True
22
23     def search(self, word):
24         # Exact word search - O(len(word))
25         node = self.root
26         for char in word:
27             if char not in node.children
28             :
29                 return False
30             node = node.children[char]
31         return node.is_end
32
33     def starts_with(self, prefix):
34         # Prefix search - O(len(prefix))
35         node = self.root
36         for char in prefix:
37             if char not in node.children
38             :
39                 return False
40             node = node.children[char]
41         return True
42
43     # Find all words with given prefix
44     def words_with_prefix(self, prefix):
45         node = self.root
46         for char in prefix:
47
48             if char not in node.children
49             :
50                 return []
51             node = node.children[char]
52
53             # DFS to collect all words
54             words = []
55             def dfs(n, path):
56                 if n.is_end:
57                     words.append(prefix +
58                         path)
59                 for char, child in n.
60                     children.items():
61                     dfs(child, path + char)
62
63             dfs(node, "")
64             return words

```

### 14 Bit Manipulation

**Description:** Efficient operations using bitwise operators. Useful for sets, flags, and optimization.

```

1 # Check if i-th bit (0-indexed) is set
2 is_set = (n >> i) & 1
3
4 # Set i-th bit to 1
5 n |= (1 << i)
6
7 # Clear i-th bit (set to 0)
8 n &= ~(1 << i)
9
10 # Toggle i-th bit
11 n ^= (1 << i)
12
13 # Count set bits (popcount)
14 count = bin(n).count('1')
15 count = n.bit_count() # Python 3.10+
16
17 # Get lowest set bit
18 lsb = n & -n # Also n & (~n + 1)
19
20 # Remove lowest set bit
21 n &= (n - 1)
22
23 # Check if power of 2
24 is_pow2 = n > 0 and (n & (n - 1)) == 0
25
26 # Check if power of 4
27 is_pow4 = n > 0 and (n & (n-1)) == 0 and
28     (n & 0x55555555) != 0
29
30 # Iterate over all subsets of set
31     represented by mask
32 mask = (1 << n) - 1 # All bits set
33 submask = mask
34 while submask > 0:
35     # Process submask
36     submask = (submask - 1) & mask
37
38 # Iterate through all k-bit masks
39 def iterate_k_bits(n, k):
40     mask = (1 << k) - 1
41     while mask < (1 << n):
42         # Process mask
43         yield mask
44         # Gosper's hack
45         c = mask & -mask
46         r = mask + c
47         mask = ((r ^ mask) >> 2) // c
48         | r

```

```

47 # XOR properties
48 # a ^ a = 0 (number XOR itself is 0)
49 # a ^ 0 = a (number XOR 0 is itself)
50 # XOR is commutative and associative
51 # Find unique element when all others
    appear twice:
52 def find_unique(arr):
53     result = 0
54     for x in arr:
55         result ^= x
56     return result
57
58 # Subset enumeration
59 n = 5 # Number of elements
60 for mask in range(1 << n):
61     subset = [i for i in range(n) if
62               mask & (1 << i)]
63     # Process subset
64
65 # Check parity (odd/even number of 1s)
66 def parity(n):
67     count = 0
68     while n:
69         count ^= 1
70         n &= n - 1
71     return count # 1 if odd, 0 if even
72
73 # Swap two numbers without temp variable
74 a, b = 5, 10
75 a ^= b
76 b ^= a
77 a ^= b
78 # Now a=10, b=5

```

## 15 Matrix Operations

**Description:** Matrix operations for DP optimization, graph algorithms, and recurrence relations.

### 15.1 Matrix Multiplication

```

1 # Standard matrix multiplication - O(n^3)
2 def matmul(A, B):
3     n, m, p = len(A), len(A[0]), len(B[0])
4     C = [[0] * p for _ in range(n)]
5
6     for i in range(n):
7         for j in range(p):
8             for k in range(m):
9                 C[i][j] += A[i][k] * B[k][j]
10
11     return C
12
13 # With modulo
14 def matmul_mod(A, B, mod):
15     n = len(A)
16     C = [[0] * n for _ in range(n)]
17
18     for i in range(n):
19         for j in range(n):
20             for k in range(n):
21                 C[i][j] = (C[i][j] +
22                             A[i][k] * B[k][j]) % mod
23
24     return C

```

### 15.2 Matrix Exponentiation

**Description:** Compute  $M^n$  in  $O(k^3 \log n)$  where  $k$  is matrix dimension. Used for solving linear recurrences efficiently.

```

1 def matpow(M, n, mod):
2     size = len(M)
3
4     # Identity matrix
5     result = [[1 if i==j else 0
6                 for j in range(size)]
7                for i in range(size)]
8
9     # Binary exponentiation
10    while n > 0:
11        if n & 1:
12            result = matmul_mod(result,
13                                  M, mod)
14            M = matmul_mod(M, M, mod)
15            n >>= 1
16
17    return result
18
19 # Example: Fibonacci using matrix
    exponentiation
20 # F(n) = [[1,1],[1,0]]^n
21 def fibonacci(n, mod):
22     if n == 0: return 0
23     if n == 1: return 1
24
25     M = [[1, 1], [1, 0]]
26     result = matpow(M, n - 1, mod)
27     return result[0][0]
28
29 # Linear recurrence: a(n) = c1*a(n-1) +
    c2*a(n-2) + ...
30 # Build transition matrix and use matrix
    exponentiation
31 def linear_recurrence(coeffs, init, n,
32                           mod):
33     k = len(coeffs)
34
35     # Transition matrix
36     # [a(n), a(n-1), ..., a(n-k+1)]
37     M = [[0] * k for _ in range(k)]
38     M[0] = coeffs # First row
39     for i in range(1, k):
40         M[i][i-1] = 1 # Identity for
        shifting
41
42     # Initial state vector
43     state = init[::-1] # Reverse order
44
45     if n < k:
46         return init[n]
47
48     # M^(n-k+1)
49     result_matrix = matpow(M, n - k + 1,
50                               mod)
51
52     # Multiply with initial state
53     result = 0
54     for i in range(k):
55         result = (result + result_matrix[0][i] * state[i]) % mod
56
57     return result
58
59 # Example: Tribonacci T(n) = T(n-1) + T(
    n-2) + T(n-3)
60 def tribonacci(n, mod):
61     if n == 0: return 0
62     if n == 1 or n == 2: return 1

```

```

60     coeffs = [1, 1, 1]
61     init = [0, 1, 1]
62     return linear_recurrence(coeffs,
63                               init, n, mod)

```

## 16 Miscellaneous Tips

### 16.1 Python-Specific Optimizations

```

1  # Fast input for large datasets
2  import sys
3  input = sys.stdin.readline
4
5  # Increase recursion limit for deep DFS/
6  # DP
7  sys.setrecursionlimit(10**6)
8
9  # Threading for higher stack limit (
10 # CAUTION: use carefully)
11 import threading
12 threading.stack_size(2**26) # 64MB
13 sys.setrecursionlimit(2**20)
14
15 # Deep copy (be careful with performance)
16 from copy import deepcopy
17 new_list = deepcopy(old_list)
18
19 # Fast output (for printing large
20 # results)
21 import sys
22 print = sys.stdout.write # Only use for
23 # string output

```

### 16.2 Useful Libraries

```

1  # Iterator tools - powerful combinations
2  from itertools import *
3
4  # permutations(iterable, r) - all r-
5  # length permutations
6  perms = list(permutations([1,2,3], 2))
7  # [(1,2), (1,3), (2,1), (2,3), (3,1),
8  # (3,2)]
9
10 # combinations(iterable, r) - r-length
11 # combinations
12 combs = list(combinations([1,2,3], 2))
13 # [(1,2), (1,3), (2,3)]
14
15 # product - cartesian product
16 prod = list(product([1,2], ['a','b']))
17 # [(1,'a'), (1,'b'), (2,'a'), (2,'b')]
18
19 # accumulate - running totals
20 acc = list(accumulate([1,2,3,4]))
21 # [1, 3, 6, 10]
22
23 # chain - flatten iterables
24 chained = list(chain([1,2], [3,4]))
25 # [1, 2, 3, 4]

```

### 16.3 Common Patterns

```

1  # Lambda sorting with multiple keys
2  arr.sort(key=lambda x: (-x[0], x[1]))
3  # Sort by first desc, then second asc
4
5  # All/Any - short-circuit evaluation

```

```

6  all(x > 0 for x in arr) # True if all
7  # positive
8  any(x > 0 for x in arr) # True if any
9  # positive
10
11 # Zip - parallel iteration
12 for a, b in zip(list1, list2):
13     pass
14
15 # Enumerate - index and value
16 for i, val in enumerate(arr):
17     print(f"arr[{i}] = {val}")
18
19 # Custom comparison function
20 from functools import cmp_to_key
21
22 def compare(a, b):
23     # Return -1 if a < b, 0 if equal, 1
24     # if a > b
25     if a + b > b + a:
26         return -1
27     return 1
28
29 arr.sort(key=cmp_to_key(compare))
30
31 # defaultdict with lambda
32 from collections import defaultdict
33 d = defaultdict(lambda: float('inf'))
34
35 # Multiple assignment
36 a, b = b, a # Swap
37 a, *rest, b = [1,2,3,4,5] # a=1, rest
38 # = [2,3,4], b=5

```

### 16.4 Common Pitfalls

```

1  # Integer division - floors toward
2  # negative infinity
3  print(7 // 3) # 2
4  print(-7 // 3) # -3 (not -2!)
5
6  # For ceiling division toward zero:
7  def div_ceil(a, b):
8      return -(a // b)
9
10 # Modulo with negative numbers
11 print((-5) % 3) # 1 (not -2!)
12 print(5 % -3) # -1
13
14 # List multiplication creates references
15 # !
16 matrix = [[0] * m] * n # WRONG! All
17 # rows same object
18 matrix[0][0] = 1 # Changes all
19 # rows!
20
21 # Correct way
22 matrix = [[0] * m for _ in range(n)]
23
24 # Float comparison - don't use ==
25 a, b = 0.1 + 0.2, 0.3
26 print(a == b) # False!
27
28 # Use epsilon comparison
29 eps = 1e-9
30 print(abs(a - b) < eps) # True
31
32 # String immutability
33 s = "abc"
34 # s[0] = 'd' # ERROR!
35 s = 'd' + s[1:] # OK
36
37 # For many string mutations, use list
38 chars = list(s)

```

```

35 chars[0] = 'd'
36 s = ''.join(chars)
37
38 # Mutable default arguments - dangerous!
39 def func(arr=[]): # WRONG!
40     arr.append(1)
41     return arr
42
43 # Each call modifies same list
44 print(func()) # [1]
45 print(func()) # [1, 1]
46
47 # Correct way
48 def func(arr=None):
49     if arr is None:
50         arr = []
51     arr.append(1)
52     return arr
53
54 # Generator expressions save memory
55 sum(x*x for x in range(10**6)) # Memory
    efficient
56
57 # vs
58 sum([x*x for x in range(10**6)]) #
    Creates full list
59
60 # Ternary operator
61 x = a if condition else b
62
63 # Dictionary get with default
64 count = d.get(key, 0) + 1
65
66 # Matrix rotation 90 degrees clockwise
67 def rotate_90(matrix):
68     return [list(row) for row in zip(*
        matrix[::-1])]
69
70 # Matrix transpose
71 def transpose(matrix):
    return [list(row) for row in zip(*
        matrix)]

```

## 16.5 Time Complexity Reference

```

1 # Common time complexities for n = 10^6:
2 # O(1), O(log n): instant
3 # O(n): ~1 second
4 # O(n log n): ~1-2 seconds
5 # O(n sqrt(n)): ~30 seconds (risky)
6 # O(n^2): TLE for n > 10^4
7 # O(2^n): TLE for n > 20
8 # O(n!): TLE for n > 11
9
10 # Input size guidelines:
11 # n <= 12: O(n!)
12 # n <= 20: O(2^n)
13 # n <= 500: O(n^3)
14 # n <= 5000: O(n^2)
15 # n <= 10^6: O(n log n)
16 # n <= 10^8: O(n)
17 # n > 10^8: O(log n) or O(1)

```

## 17 Computational Geometry

### 17.1 Basic Geometry

**Description:** Fundamental geometric operations for 2D points.

```

1 import math
2
3 # Point operations
4 def dist(p1, p2):
5     # Euclidean distance
6     return math.sqrt((p1[0] - p2[0])**2
7     + (p1[1] - p2[1])**2)
8
9 def cross_product(O, A, B):
10     # Cross product of vectors OA and OB
11     # Positive: counter-clockwise
12     # Negative: clockwise
13     # Zero: collinear
14     return (A[0] - O[0]) * (B[1] - O[1])
15     - \
16     (A[1] - O[1]) * (B[0] - O[0])
17
18 def dot_product(A, B, C, D):
19     # Dot product of vectors AB and CD
20     return (B[0] - A[0]) * (D[0] - C[0])
21     + \
22     (B[1] - A[1]) * (D[1] - C[1])
23
24 # Check if point is on segment
25 def on_segment(p, q, r):
26     # Check if q lies on segment pr
27     return (q[0] <= max(p[0], r[0]) and
28     q[0] >= min(p[0], r[0]) and
29     q[1] <= max(p[1], r[1]) and
30     q[1] >= min(p[1], r[1]))
31
32 # Segment intersection
33 def segments_intersect(p1, q1, p2, q2):
34     o1 = cross_product(p1, q1, p2)
35     o2 = cross_product(p1, q1, q2)
36     o3 = cross_product(p2, q2, p1)
37     o4 = cross_product(p2, q2, q1)
38
39     # General case
40     if o1 * o2 < 0 and o3 * o4 < 0:
41         return True
42
43     # Special cases (collinear)
44     if o1 == 0 and on_segment(p1, p2, q1):
45         return True
46     if o2 == 0 and on_segment(p1, q2, q1):
47         return True
48     if o3 == 0 and on_segment(p2, p1, q2):
49         return True
50     if o4 == 0 and on_segment(p2, q1, q2):
51         return True
52     return False

```

### 17.2 Convex Hull

**Description:** Find convex hull using Graham's scan. Time:  $O(n \log n)$ .

```

1 def convex_hull(points):
2     # Graham's scan algorithm
3     points = sorted(points) # Sort by x
4     # , then y
5
6     if len(points) <= 2:
7         return points
8
9     # Build lower hull
10    lower = []
11    for p in points:

```

```

11         while (len(lower) >= 2 and
12               cross_product(lower[-2],
13                             lower[-1], p) <= 0):
14             lower.pop()
15             lower.append(p)
16
17     # Build upper hull
18     upper = []
19     for p in reversed(points):
20         while (len(upper) >= 2 and
21               cross_product(upper[-2],
22                             upper[-1], p) <= 0):
23             upper.pop()
24             upper.append(p)
25
26     # Remove last point (duplicate of
27     # first)
28     return lower[:-1] + upper[:-1]
29
30 # Convex hull area
31 def polygon_area(points):
32     # Shoelace formula
33     n = len(points)
34     area = 0
35
36     for i in range(n):
37         j = (i + 1) % n
38         area += points[i][0] * points[j]
39         area -= points[j][0] * points[i]
40
41     return abs(area) / 2

```

### 17.3 Point in Polygon

**Description:** Check if point is inside polygon. Time:  $O(n)$ .

```

1 def point_in_polygon(point, polygon):
2     # Ray casting algorithm
3     x, y = point
4     n = len(polygon)
5     inside = False
6
7     p1x, p1y = polygon[0]
8     for i in range(1, n + 1):
9         p2x, p2y = polygon[i % n]
10
11         if y > min(p1y, p2y):
12             if y <= max(p1y, p2y):
13                 if x <= max(p1x, p2x):
14                     if p1y != p2y:
15                         xinters = (y -
16                                     p1y) * (p2x - p1x) / \
17                                     (p2y -
18                                     p1y) + p1x
19
20                     if p1x == p2x or x
21                         <= xinters:
22                             inside = not
23                             inside
24
25     p1x, p1y = p2x, p2y
26
27     return inside

```

### 17.4 Closest Pair of Points

**Description:** Find closest pair using divide and conquer. Time:  $O(n \log n)$ .

```

1 def closest_pair(points):

```

```

2     points_sorted_x = sorted(points, key
3                               =lambda p: p[0])
4     points_sorted_y = sorted(points, key
5                               =lambda p: p[1])
6
7     def closest_recursive(px, py):
8         n = len(px)
9
10        # Base case: brute force
11        if n <= 3:
12            min_dist = float('inf')
13            for i in range(n):
14                for j in range(i + 1, n):
15                    min_dist = min(
16                        min_dist, dist(px[i], px[j]))
17            return min_dist
18
19        # Divide
20        mid = n // 2
21        midpoint = px[mid]
22
23        pyl = [p for p in py if p[0] <=
24                midpoint[0]]
25        pyr = [p for p in py if p[0] >
26                midpoint[0]]
27
28        # Conquer
29        dl = closest_recursive(px[:mid],
30                                pyl)
31        dr = closest_recursive(px[mid:],
32                                pyr)
33        d = min(dl, dr)
34
35        # Combine: check strip
36        strip = [p for p in py if abs(p
37                                       [0] - midpoint[0]) < d]
38
39        for i in range(len(strip)):
40            j = i + 1
41            while j < len(strip) and
42                  strip[j][1] - strip[i][1] < d:
43                d = min(d, dist(strip[i]
44                                ], strip[j]))
45            j += 1
46
47        return d
48
49    return closest_recursive(
50        points_sorted_x, points_sorted_y)

```

## 18 Network Flow

### 18.1 Maximum Flow - Edmonds-Karp (BFS-based Ford-Fulkerson)

**Description:** Find maximum flow from source to sink. Time:  $O(VE^2)$ .

```

1 from collections import deque,
2   defaultdict
3
4 def max_flow(graph, source, sink, n):
5     # graph[u][v] = capacity from u to v
6     # Build residual graph
7     residual = defaultdict(lambda:
8                             defaultdict(int))
9     for u in graph:
10        for v in graph[u]:
11            residual[u][v] = graph[u][v]

```

```

11 def bfs_path():
12     # Find augmenting path using BFS
13     parent = {source: None}
14     visited = {source}
15     queue = deque([source])
16
17     while queue:
18         u = queue.popleft()
19
20         if u == sink:
21             # Reconstruct path
22             path = []
23             while parent[u] is not
24                 path.append((parent[
25                     u], u))
26             u = parent[u]
27             return path[::-1]
28
29         for v in range(n):
30             if v not in visited and
31                 residual[u][v] > 0:
32                 visited.add(v)
33                 parent[v] = u
34                 queue.append(v)
35
36         return None
37
38     max_flow_value = 0
39
40     # Find augmenting paths
41     while True:
42         path = bfs_path()
43         if path is None:
44             break
45
46         # Find minimum capacity along
47         # path
48         flow = min(residual[u][v] for u,
49                     v in path)
50
51         # Update residual graph
52         for u, v in path:
53             residual[u][v] -= flow
54             residual[v][u] += flow
55
56         max_flow_value += flow
57
58     return max_flow_value
59
60 # Example usage
61 # graph[u][v] = capacity
62 graph = defaultdict(lambda: defaultdict(
63     int))
64 graph[0][1] = 10
65 graph[0][2] = 10
66 graph[1][3] = 4
67 graph[1][4] = 8
68 graph[2][4] = 9
69 graph[3][5] = 10
70 graph[4][3] = 6
71 graph[4][5] = 10
72
73 n = 6 # Number of nodes
74 result = max_flow(graph, 0, 5, n)

```

## 18.2 Dinic's Algorithm (Faster)

**Description:** Faster max flow using level graph and blocking flow. Time:  $O(V^2E)$ .

```

1 from collections import deque,
2     defaultdict
3
4 class Dinic:
5     def __init__(self, n):
6         self.n = n
7         self.graph = defaultdict(lambda:
8             defaultdict(int))
9
10    def add_edge(self, u, v, cap):
11        self.graph[u][v] += cap
12
13    def bfs(self, source, sink):
14        # Build level graph
15        level = [-1] * self.n
16        level[source] = 0
17        queue = deque([source])
18
19        while queue:
20            u = queue.popleft()
21
22            for v in range(self.n):
23                if level[v] == -1 and
24                    self.graph[u][v] > 0:
25                    level[v] = level[u]
26                    + 1
27                    queue.append(v)
28
29            return level if level[sink] !=
30                -1 else None
31
32    def dfs(self, u, sink, pushed, level
33            , start):
34        if u == sink:
35            return pushed
36
37        while start[u] < self.n:
38            v = start[u]
39
40            if (level[v] == level[u] + 1
41                and
42                    self.graph[u][v] > 0):
43                flow = self.dfs(v, sink,
44                                min(
45                                    pushed, self.graph[u][v]),
46                                level,
47                                start)
48
49                if flow > 0:
50                    self.graph[u][v] -=
51                        flow
52                    self.graph[v][u] +=
53                        flow
54                    return flow
55
56            start[u] += 1
57
58            return 0
59
60    def max_flow(self, source, sink):
61        flow = 0
62
63        while True:
64            level = self.bfs(source,
65                            sink)
66            if level is None:
67                break
68
69            start = [0] * self.n
70
71            while True:
72                pushed = self.dfs(source
73                                , sink, float('inf'),

```

```

62         start)
63         if pushed == 0:
64             break
65         flow += pushed
66
67     return flow

```

```

22     level,
23     return cut_edges

```

### 18.3 Min Cut

**Description:** Find minimum cut after computing max flow.

```

1 def min_cut(graph, source, n, residual):
2     # After running max_flow, residual
3     graph is available
4     # Min cut = set of reachable nodes
5     from source
6     visited = [False] * n
7     queue = deque([source])
8     visited[source] = True
9
10    while queue:
11        u = queue.popleft()
12        for v in range(n):
13            if not visited[v] and
14            residual[u][v] > 0:
15                visited[v] = True
16                queue.append(v)
17
18    # Cut edges
19    cut_edges = []
20    for u in range(n):
21        if visited[u]:
22            for v in range(n):
23                if not visited[v] and
24                graph[u][v] > 0:
25                    cut_edges.append((u,
26                    v))

```

### 18.4 Bipartite Matching

**Description:** Maximum matching in bipartite graph using flow.

```

1 def max_bipartite_matching(left_size,
2     right_size, edges):
3     # edges = [(left_node, right_node),
4     ...]
5     # Add source (0) and sink (left_size
6     + right_size + 1)
7
8     n = left_size + right_size + 2
9     source = 0
10    sink = n - 1
11
12    graph = defaultdict(lambda:
13        defaultdict(int))
14
15    # Source to left nodes
16    for i in range(1, left_size + 1):
17        graph[source][i] = 1
18
19    # Left to right edges
20    for l, r in edges:
21        graph[l + 1][left_size + r + 1]
22        = 1
23
24    # Right nodes to sink
25    for i in range(1, right_size + 1):
26        graph[left_size + i][sink] = 1
27
28    return max_flow(graph, source, sink,
29    n)

```