

Python ICPC Cheatsheet

Comprehensive Reference for Competitive Programming

November 3, 2025

Contents

1	Input/Output	4
2	Basic Data Structures	5
2.1	List Operations	5
2.2	Deque (Double-ended Queue)	5
2.3	Heap (Priority Queue)	6
2.4	Dictionary & Counter	6
2.5	Set Operations	7
3	String Operations	8
3.1	KMP Pattern Matching	8
3.2	Z-Algorithm	9
3.3	Rabin-Karp (Rolling Hash)	10
4	Mathematics	11
4.1	Basic Math Operations	11
4.2	Combinatorics	11
5	Number Theory	13
5.1	Modular Arithmetic	13
5.2	Sieve of Eratosthenes	13
5.3	Prime Factorization	13
5.4	Chinese Remainder Theorem	14
5.5	Euler's Totient Function	14
5.6	Fast Exponentiation with Matrix	15
6	Graph Algorithms	16
6.1	Graph Representation	16
6.2	BFS (Breadth-First Search)	16
6.3	DFS (Depth-First Search)	17
6.4	Strongly Connected Components (SCC)	19
6.5	Bridges and Articulation Points	19
6.6	Lowest Common Ancestor (LCA)	20
7	Shortest Path Algorithms	22
7.1	Dijkstra's Algorithm	22
7.2	Bellman-Ford Algorithm	23
7.3	Floyd-Warshall Algorithm	23

7.4	Minimum Spanning Tree	23
7.4.1	Kruskal's Algorithm	23
7.4.2	Prim's Algorithm	24
8	Topological Sort	26
8.1	Kahn's Algorithm (BFS-based)	26
8.2	DFS-based Topological Sort	26
9	Union-Find (Disjoint Set Union)	28
10	Binary Search	30
10.1	Template for Finding First/Last Position	30
11	Dynamic Programming	32
11.1	Longest Increasing Subsequence	32
11.2	0/1 Knapsack	33
11.3	Edit Distance (Levenshtein Distance)	33
11.4	Longest Common Subsequence (LCS)	34
11.5	Coin Change	34
11.6	Palindrome Partitioning	35
11.7	Subset Sum	35
12	Array Techniques	37
12.1	Prefix Sum	37
12.2	Difference Array	37
12.3	Sliding Window	38
13	Advanced Data Structures	39
13.1	Segment Tree	39
13.2	Fenwick Tree (Binary Indexed Tree)	40
13.3	Trie (Prefix Tree)	40
14	Bit Manipulation	42
15	Matrix Operations	44
15.1	Matrix Multiplication	44
15.2	Matrix Exponentiation	44
16	Miscellaneous Tips	46
16.1	Python-Specific Optimizations	46
16.2	Useful Libraries	46
16.3	Common Patterns	46
16.4	Common Pitfalls	47
16.5	Time Complexity Reference	48

17 Computational Geometry	49
17.1 Basic Geometry	49
17.2 Convex Hull	49
17.3 Point in Polygon	50
17.4 Closest Pair of Points	51
18 Network Flow	52
18.1 Maximum Flow - Edmonds-Karp (BFS-based Ford-Fulkerson)	52
18.2 Dinic's Algorithm (Faster)	53
18.3 Min Cut	54
18.4 Bipartite Matching	54

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 newline)
15 s = input().strip()
16 words = input().split()
17
18 # Multiple test cases pattern
19 t = int(input())
20 for _ in range(t):
21     # process each test case
22
23 # Print without newline
24 print(x, end=' ')
25
26 # Formatted output with precision
27 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)] # Correct way!
4
5 # List comprehension - concise and efficient
6 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])) # custom
13 sorted_arr = sorted(arr) # returns new list
14
15 # Binary search in sorted array
16 from bisect import bisect_left, bisect_right
17 idx = bisect_left(arr, x) # leftmost position
18 idx = bisect_right(arr, x) # rightmost position
19
20 # Common operations
21 arr.append(x) #  $O(1)$  amortized
22 arr.pop() #  $O(1)$  - remove last
23 arr.pop(0) #  $O(n)$  - remove first (slow!)
24 arr.reverse() #  $O(n)$  - in-place
25 arr.count(x) #  $O(n)$  - count occurrences
26 arr.index(x) #  $O(n)$  - first 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
21         # Remove smaller elements (not useful)
22         while dq and arr[dq[-1]] < arr[i]:
```

```

23         dq.pop()
24
25     dq.append(i)
26     if i >= k - 1:
27         result.append(arr[dq[0]])
28
29     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  min_val = heapq.heappop(heap)     # O(log n)
7  min_val = heap[0]                 # O(1) peek
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, etc.
22 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, Counter
2
3  # defaultdict - provides default value
4  graph = defaultdict(list) # empty list default
5  count = defaultdict(int)  # 0 default
6
7  # Counter - count elements efficiently
8  cnt = Counter(arr)
9  cnt['x'] += 1
10 most_common = cnt.most_common(k) # k most frequent
11
12 # Dictionary operations
13 d = {}
14 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 exists
4 s.discard(x)      # O(1), no error if not exists
5
6 # Set operations - all O(n)
7 a | b             # union
8 a & b             # intersection
9 a - b             # difference
10 a ^ b            # symmetric difference
11
12 # Ordered set workaround
13 from collections import OrderedDict
14 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 ends
4 s.lstrip() # remove left whitespace
5 s.rstrip() # remove right whitespace
6 s.split(delimiter)
7 delimiter.join(list)
8 s.replace(old, new)
9 s.startswith(prefix)
10 s.endswith(suffix)
11 s.isdigit(), s.isalpha(), s.isalnum()
12
13 # String building - EFFICIENT  $O(n)$ 
14 result = []
15 for x in data:
16     result.append(str(x))
17 s = ''.join(result)
18
19 # String concatenation - SLOW  $O(n^2)$ 
20 # s = ""
21 # for x in data:
22 #     s += str(x) # Don't do this!
23
24 # ASCII values
25 ord('a') # 97
26 chr(97) # 'a'
27
28 # String to character array (for mutations)
29 chars = list(s)
30 chars[0] = 'x'
31 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 which is Suffix)
3     def build_lps(pattern):
4         m = len(pattern)
5         lps = [0] * m
6         length = 0 # Length of previous longest prefix
7         i = 1
8
9         while i < m:
10             if pattern[i] == pattern[length]:
11                 length += 1
12                 lps[i] = length
13                 i += 1
14             else:
15                 if length != 0:
16                     length = lps[length - 1]
17                 else:
18                     lps[i] = 0
19                     i += 1
20
```

```

21         return lps
22
23     n, m = len(text), len(pattern)
24     lps = build_lps(pattern)
25
26     matches = []
27     i = j = 0 # Indices for text and pattern
28
29     while i < n:
30         if text[i] == pattern[j]:
31             i += 1
32             j += 1
33
34         if j == m:
35             matches.append(i - j)
36             j = lps[j - 1]
37         elif i < n and text[i] != pattern[j]:
38             if j != 0:
39                 j = lps[j - 1]
40             else:
41                 i += 1
42
43     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 - 1])
9
10             while i + z[i] < n and s[z[i]] == s[i + z[i]]:
11                 z[i] += 1
12
13             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 * BASE +
14                        ord(pattern[i])) % MOD
15        if i < m - 1:
16            power = (power * BASE) % MOD
17
18    # Rolling hash
19    text_hash = 0
20    matches = []
21
22    for i in range(n):
23        # Add new character
24        text_hash = (text_hash * BASE +
25                    ord(text[i])) % MOD
26
27        # Remove old character if window full
28        if i >= m:
29            text_hash = (text_hash -
30                        ord(text[i - m]) * power) % MOD
31            text_hash = (text_hash + MOD) % MOD
32
33        # Check match
34        if i >= m - 1 and text_hash == pattern_hash:
35            # Verify actual match (avoid hash collision)
36            if text[i - m + 1:i + 1] == pattern:
37                matches.append(i - m + 1)
38
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 #  $a^{-1} = a^{(mod-2)} \pmod{p}$ 
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 else 2
30         i += 1
31     return count
32
33 # Sum of divisors
34 def sum_divisors(n):
35     total = 0
36     i = 1
37     while i * i <= n:
38         if n % i == 0:
39             total += i
40             if i != n // i:
41                 total += n // i
42         i += 1
43     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, weight) tuples
11 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[node] + 1
14
15     return dist
16
17 # Grid BFS - common in maze/path problems
18 def grid_bfs(grid, start):
19     n, m = len(grid), len(grid[0])
20     visited = [[False] * m for _ in range(n)]
21     queue = deque([start])
22     visited[start[0]][start[1]] = True
23
24     # 4 directions: right, down, left, up
25     dirs = [(0,1), (1,0), (0,-1), (-1,0)]
26
27     while queue:
28         x, y = queue.popleft()
29
30         for dx, dy in dirs:
31             nx, ny = x + dx, y + dy
32
33             # Check bounds and validity
34             if (0 <= nx < n and 0 <= ny < m
35                 and not visited[nx][ny]
36                 and grid[nx][ny] != '#'):
37
38                 visited[nx][ny] = True
```

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 visited:
22                     stack.append(neighbor)
23
24 # Cycle detection in undirected graph
25 def has_cycle(graph, n):
26     visited = [False] * n
27
28     def dfs(node, parent):
29         visited[node] = True
30
31         for neighbor in graph[node]:
32             if not visited[neighbor]:
33                 if dfs(neighbor, node):
34                     return True
35             # Back edge to non-parent = cycle
36             elif neighbor != parent:
37                 return True
38
39     return False
40
41 # Check all components
42 for i in range(n):
43     if not visited[i]:
44         if dfs(i, -1):
45             return True
46
47 return False
48
49 # Cycle detection in directed graph
50 def has_cycle_directed(graph, n):
51     WHITE, GRAY, BLACK = 0, 1, 2
52     color = [WHITE] * n
53
54     def dfs(node):

```

```

55     color[node] = GRAY
56
57     for neighbor in graph[node]:
58         if color[neighbor] == GRAY:
59             return True # Back edge = cycle
60         if color[neighbor] == WHITE:
61             if dfs(neighbor):
62                 return True
63
64     color[node] = BLACK
65     return False
66
67     for i in range(n):
68         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] == -1:
105                    color[neighbor] = 1 - color[node]
106                    queue.append(neighbor)
107                elif color[neighbor] == color[node]:
108                    return False
109
110        return True
111
112     for i in range(n):
113         if color[i] == -1:
114             if not bfs(i):
115                 return False

```

```

116
117     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[v], lowlink[w])
22             elif on_stack[w]:
23                 lowlink[v] = min(lowlink[v], index[w])
24
25         if lowlink[v] == index[v]:
26             scc = []
27             while True:
28                 w = stack.pop()
29                 on_stack[w] = False
30                 scc.append(w)
31                 if w == v:
32                     break
33             sccs.append(scc)
34
35     for v in range(n):
36         if not index_initialized[v]:
37             strongconnect(v)
38
39     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[v])
19
20             # Bridge condition
21             if low[v] > disc[u]:
22                 bridges.append((u, v))
23         elif v != parent[u]:
24             low[u] = min(low[u], disc[v])
25
26     for i in range(n):
27         if not visited[i]:
28             dfs(i)
29
30     return bridges
31
32 def find_articulation_points(graph, n):
33     visited = [False] * n
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[v])
52
53                 # Articulation point conditions
54                 if parent[u] == -1 and children > 1:
55                     ap.add(u)
56                 if parent[u] != -1 and low[v] >= disc[u]:
57                     ap.add(u)
58             elif v != parent[u]:
59                 low[u] = min(low[u], disc[v])
60
61     for i in range(n):
62         if not visited[i]:
63             dfs(i)
64
65     return list(ap)

```

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 #  $\log_2(n) + 1$ 
5         self.parent = [[-1] * self.LOG for _ in range(n)]
6         self.depth = [0] * n
7
8         # DFS to set parent and depth
9         visited = [False] * n
10
11        def dfs(node, par, d):
12            visited[node] = True
13            self.parent[node][0] = par
14            self.depth[node] = d
15
16            for neighbor in graph[node]:
17                if not visited[neighbor]:
18                    dfs(neighbor, node, d + 1)
19
20        dfs(root, -1, 0)
21
22        # Binary lifting preprocessing
23        for j in range(1, self.LOG):
24            for i in range(n):
25                if self.parent[i][j-1] != -1:
26                    self.parent[i][j] = self.parent[
27                        self.parent[i][j-1]][j-1]
28
29        def lca(self, u, v):
30            # Make u deeper
31            if self.depth[u] < self.depth[v]:
32                u, v = v, u
33
34            # Bring u to same level as v
35            diff = self.depth[u] - self.depth[v]
36            for i in range(self.LOG):
37                if (diff >> i) & 1:
38                    u = self.parent[u][i]
39
40            if u == v:
41                return u
42
43            # Binary search for LCA
44            for i in range(self.LOG - 1, -1, -1):
45                if self.parent[u][i] != self.parent[v][i]:
46                    u = self.parent[u][i]
47                    v = self.parent[v][i]
48
49            return self.parent[u][0]
50
51        def dist(self, u, v):
52            # Distance between two nodes
53            l = self.lca(u, v)
54            return self.depth[u] + self.depth[v] - 2 * self.depth[l]

```

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 better distance
15         if d > dist[node]:
16             continue
17
18         # Relax edges
19         for neighbor, weight in graph[node]:
20             new_dist = dist[node] + weight
21
22             if new_dist < dist[neighbor]:
23                 dist[neighbor] = new_dist
24                 heapq.heappush(heap, (new_dist, neighbor))
25
26     return dist
27
28 # Path reconstruction
29 def dijkstra_with_path(graph, start, n):
30     dist = [float('inf')] * n
31     parent = [-1] * n
32     dist[start] = 0
33     heap = [(0, start)]
34
35     while heap:
36         d, node = heapq.heappop(heap)
37         if d > dist[node]:
38             continue
39
40         for neighbor, weight in graph[node]:
41             new_dist = dist[node] + weight
42             if new_dist < dist[neighbor]:
43                 dist[neighbor] = new_dist
44                 parent[neighbor] = node
45                 heapq.heappush(heap, (new_dist, neighbor))
46
47     return dist, parent
48
49 def reconstruct_path(parent, target):
50     path = []
51     while target != -1:
52         path.append(target)
53         target = parent[target]
54     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') and \
10                dist[u] + w < dist[v]:
11                 dist[v] = dist[u] + w
12
13     # Check for negative cycles
14     for u, v, w in edges:
15         if dist[u] != float('inf') and \
16            dist[u] + w < dist[v]:
17             return None # Negative cycle exists
18
19     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 range(n)]
4
5     for i in range(n):
6         dist[i][i] = 0
7
8     for u, v, w in edges:
9         dist[u][v] = min(dist[u][v], w)
10
11     # Dynamic programming
12     for k in range(n): # Intermediate vertex
13         for i in range(n):
14             for j in range(n):
15                 dist[i][j] = min(dist[i][j],
16                                   dist[i][k] + dist[k][j])
17
18     return dist
19
20 # Check for negative cycle
21 def has_negative_cycle(dist, n):
22     for i in range(n):
23         if dist[i][i] < 0:
24             return True
25     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, weight))
13
14     return mst_weight, mst_edges
15
16 class UnionFind:
17     def __init__(self, n):
18         self.parent = list(range(n))
19         self.rank = [0] * n
20
21     def find(self, x):
22         if self.parent[x] != x:
23             self.parent[x] = self.find(self.parent[x])
24         return self.parent[x]
25
26     def union(self, x, y):
27         px, py = self.find(x), self.find(y)
28         if px == py:
29             return False
30         if self.rank[px] < self.rank[py]:
31             px, py = py, px
32         self.parent[py] = px
33         if self.rank[px] == self.rank[py]:
34             self.rank[px] += 1
35         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     mst_weight = 0
8
9     while min_heap:
10         weight, u = heapq.heappop(min_heap)
11
12         if visited[u]:
13             continue
14
15         visited[u] = True
16         mst_weight += weight
17
18         for v, w in graph[u]:
19             if not visited[v]:
20                 heapq.heappush(min_heap, (w, v))
21

```

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

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

```
55     return -1
56
57     # Binary search on number of days
58     return binary_search_first(
59         min(bloomDay), max(bloomDay), can_make)
```

11 Dynamic Programming

Description: Solve problems by breaking them into overlapping subproblems. 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 LIS of length i+1
5     dp = []
6
7     for x in arr:
8         # Find position to place x
9         idx = bisect_left(dp, x)
10
11        if idx == len(dp):
12            dp.append(x) # Extend LIS
13        else:
14            dp[idx] = x # Better ending for this length
15
16    return len(dp)
17
18 # LIS with actual sequence
19 def lis_with_sequence(arr):
20     from bisect import bisect_left
21
22     n = len(arr)
23     dp = []
24     parent = [-1] * n
25     dp_idx = [] # indices in dp
26
27     for i, x in enumerate(arr):
28         idx = bisect_left(dp, x)
29
30         if idx == len(dp):
31             dp.append(x)
32             dp_idx.append(i)
33         else:
34             dp[idx] = x
35             dp_idx[idx] = i
36
37         if idx > 0:
38             parent[i] = dp_idx[idx - 1]
39
40     # Reconstruct sequence
41     result = []
42     idx = dp_idx[-1]
43     while idx != -1:
44         result.append(arr[idx])
45         idx = parent[idx]
46
47     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 items,
4     #           weight <= w
5     dp = [[0] * (capacity + 1) for _ in range(n + 1)]
6
7     for i in range(1, n + 1):
8         for w in range(capacity + 1):
9             # Don't take item i-1
10            dp[i][w] = dp[i-1][w]
11
12            # Take item i-1 if it fits
13            if weights[i-1] <= w:
14                dp[i][w] = max(
15                    dp[i][w],
16                    dp[i-1][w - weights[i-1]] + values[i-1]
17                )
18
19     return dp[n][capacity]
20
21 # Space-optimized O(capacity)
22 def knapsack_optimized(weights, values, capacity):
23     dp = [0] * (capacity + 1)
24
25     for i in range(len(weights)):
26         # Iterate backwards to avoid using updated values
27         for w in range(capacity, weights[i] - 1, -1):
28             dp[w] = max(dp[w],
29                         dp[w - weights[i]] + values[i])
30
31     return dp[capacity]
```

11.3 Edit Distance (Levenshtein Distance)

Description: Minimum operations (insert, delete, replace) to transform s1 to s2. 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] and s2[:j]
4     dp = [[0] * (n + 1) for _ in range(m + 1)]
5
6     # Base cases: empty string transformations
7     for i in range(m + 1):
8         dp[i][0] = i # Delete all
9     for j in range(n + 1):
10        dp[0][j] = j # Insert all
11
12    for i in range(1, m + 1):
13        for j in range(1, n + 1):
14            if s1[i-1] == s2[j-1]:
15                # Characters match, no operation needed
16                dp[i][j] = dp[i-1][j-1]
17            else:
18                dp[i][j] = 1 + min(
19                    dp[i-1][j], # Delete from s1
```

```

20         dp[i][j-1],          # Insert into s1
21         dp[i-1][j-1]         # Replace in s1
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         dp[i] = min(dp[i], dp[i - coin] + 1)
9
10    return dp[amount] if dp[amount] != float('inf') else -1
11
12    # Count ways
13    def coin_change_ways(coins, amount):
14        dp = [0] * (amount + 1)
15        dp[0] = 1
16
17        for coin in coins:
18            for i in range(coin, amount + 1):
19                dp[i] += dp[i - coin]
20
21    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 _ in range(n + 1)]
4
5        # Base case: sum 0 is always achievable
6        for i in range(n + 1):
7            dp[i][0] = True

```

```

8
9     for i in range(1, n + 1):
10         for s in range(target + 1):
11             # Don't take arr[i-1]
12             dp[i][s] = dp[i-1][s]
13
14             # Take arr[i-1] if possible
15             if s >= arr[i-1]:
16                 dp[i][s] = dp[i][s] or dp[i-1][s - arr[i-1]]
17
18     return dp[n][target]
19
20 # Space optimized
21 def subset_sum_optimized(arr, target):
22     dp = [False] * (target + 1)
23     dp[0] = True
24
25     for num in arr:
26         for s in range(target, num - 1, -1):
27             dp[s] = dp[s] or dp[s - num]
28
29     return dp[target]

```

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 queries
10 def build_2d_prefix(matrix):
11     n, m = len(matrix), len(matrix[0])
12     prefix = [[0] * (m + 1) for _ in range(n + 1)]
13
14     for i in range(1, n + 1):
15         for j in range(1, m + 1):
16             prefix[i][j] = (matrix[i-1][j-1] +
17                             prefix[i-1][j] +
18                             prefix[i][j-1] -
19                             prefix[i-1][j-1])
20
21     return prefix
22
23 # Rectangle sum from (x1,y1) to (x2,y2) inclusive
24 def rect_sum(prefix, x1, y1, x2, y2):
25     return (prefix[x2+1][y2+1] -
26             prefix[x1][y2+1] -
27             prefix[x2+1][y1] +
28             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)
```

12.3 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 left
7      for i in range(k, len(arr)):
8          window_sum += arr[i] - arr[i - k]
9          max_sum = max(max_sum, window_sum)
10
11     return max_sum
12
13 # Variable size window - two pointers
14 def min_subarray_sum_geq_target(arr, target):
15     left = 0
16     current_sum = 0
17     min_len = float('inf')
18
19     for right in range(len(arr)):
20         current_sum += arr[right]
21
22         # Shrink window while condition holds
23         while current_sum >= target:
24             min_len = min(min_len, right - left + 1)
25             current_sum -= arr[left]
26             left += 1
27
28     return min_len if min_len != float('inf') else 0
29
30 # Longest substring with at most k distinct chars
31 def longest_k_distinct(s, k):
32     from collections import defaultdict
33
34     left = 0
35     char_count = defaultdict(int)
36     max_len = 0
37
38     for right in range(len(s)):
39         char_count[s[right]] += 1
40
41         # Shrink if too many distinct
42         while len(char_count) > k:
43             char_count[s[left]] -= 1
44             if char_count[s[left]] == 0:
45                 del char_count[s[left]]
46             left += 1
47
48         max_len = max(max_len, right - left + 1)
49
50     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 bound
5         self.tree = [0] * (4 * self.n)
6         self.build(arr, 0, 0, self.n - 1)
7
8     def build(self, arr, node, start, end):
9         if start == end:
10             # Leaf node
11             self.tree[node] = arr[start]
12         else:
13             mid = (start + end) // 2
14             # Build left and right subtrees
15             self.build(arr, 2*node+1, start, mid)
16             self.build(arr, 2*node+2, mid+1, end)
17             # Combine results (sum in this case)
18             self.tree[node] = (self.tree[2*node+1] +
19                               self.tree[2*node+2])
20
21     def update(self, node, start, end, idx, val):
22         if start == end:
23             # Leaf node - update value
24             self.tree[node] = val
25         else:
26             mid = (start + end) // 2
27             if idx <= mid:
28                 # Update left subtree
29                 self.update(2*node+1, start, mid, idx, val)
30             else:
31                 # Update right subtree
32                 self.update(2*node+2, mid+1, end, idx, val)
33             # Recompute parent
34             self.tree[node] = (self.tree[2*node+1] +
35                               self.tree[2*node+2])
36
37     def query(self, node, start, end, l, r):
38         # No overlap
39         if r < start or end < l:
40             return 0
41
42         # Complete overlap
43         if l <= start and end <= r:
44             return self.tree[node]
45
46         # Partial overlap
47         mid = (start + end) // 2
48         left_sum = self.query(2*node+1, start, mid, l, r)
49         right_sum = self.query(2*node+2, mid+1, end, l, r)
50         return left_sum + right_sum
51
52     # Public interface
53     def update_val(self, idx, val):
54         self.update(0, 0, self.n-1, idx, val)
```

```

55
56     def range_sum(self, l, r):
57         return self.query(0, 0, self.n-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 implementation
5          self.tree = [0] * (n + 1)
6
7      def update(self, i, delta):
8          # Add delta to position i (1-indexed)
9          while i <= self.n:
10             self.tree[i] += delta
11             # Move to next node: add LSB
12             i += i & (-i)
13
14      def query(self, i):
15          # Get prefix sum up to i (1-indexed)
16          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.query(l - 1)
26
27  # Usage example
28  bit = FenwickTree(n)
29  for i, val in enumerate(arr, 1):
30      bit.update(i, val)
31
32  # Range sum [l, r] (1-indexed)
33  result = bit.range_query(l, r)

```

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 -> TrieNode
4          self.is_end = False # End of word marker
5
6  class Trie:
7      def __init__(self):
8          self.root = TrieNode()
9
10     def insert(self, word):
11         # Insert word - O(len(word))
12         node = self.root

```

```

13     for char in word:
14         if char not in node.children:
15             node.children[char] = TrieNode()
16             node = node.children[char]
17     node.is_end = True
18
19 def search(self, word):
20     # Exact word search - O(len(word))
21     node = self.root
22     for char in word:
23         if char not in node.children:
24             return False
25         node = node.children[char]
26     return node.is_end
27
28 def starts_with(self, prefix):
29     # Prefix search - O(len(prefix))
30     node = self.root
31     for char in prefix:
32         if char not in node.children:
33             return False
34         node = node.children[char]
35     return True
36
37 # Find all words with given prefix
38 def words_with_prefix(self, prefix):
39     node = self.root
40     for char in prefix:
41         if char not in node.children:
42             return []
43         node = node.children[char]
44
45     # DFS to collect all words
46     words = []
47     def dfs(n, path):
48         if n.is_end:
49             words.append(prefix + path)
50         for char, child in n.children.items():
51             dfs(child, path + char)
52
53     dfs(node, "")
54     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 (n & 0x55555555) != 0
28
29 # Iterate over all subsets of set represented by mask
30 mask = (1 << n) - 1 # All bits set
31 submask = mask
32 while submask > 0:
33     # Process submask
34     submask = (submask - 1) & mask
35
36 # Iterate through all k-bit masks
37 def iterate_k_bits(n, k):
38     mask = (1 << k) - 1
39     while mask < (1 << n):
40         # Process mask
41         yield mask
42         # Gosper's hack
43         c = mask & -mask
44         r = mask + c
45         mask = ((r ^ mask) >> 2) // c | r
46
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 mask & (1 << i)]
62     # Process subset
63
64 # Check parity (odd/even number of 1s)
65 def parity(n):
66     count = 0
67     while n:
68         count ^= 1
69         n &= n - 1
70     return count # 1 if odd, 0 if even
71
72 # Swap two numbers without temp variable
73 a, b = 5, 10
74 a ^= b
75 b ^= a
76 a ^= b
77 # 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, M, mod)
13        M = matmul_mod(M, M, mod)
14        n >>= 1
15
16    return result
17
18 # Example: Fibonacci using matrix exponentiation
19 #  $F(n) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ 
20 def fibonacci(n, mod):
21     if n == 0: return 0
22     if n == 1: return 1
23
24     M = [[1, 1], [1, 0]]
25     result = matpow(M, n - 1, mod)
```

```

26     return result[0][0]
27
28 # Linear recurrence:  $a(n) = c_1 a(n-1) + c_2 a(n-2) + \dots$ 
29 # Build transition matrix and use matrix exponentiation
30 def linear_recurrence(coeffs, init, n, mod):
31     k = len(coeffs)
32
33     # Transition matrix
34     #  $[a(n), a(n-1), \dots, a(n-k+1)]$ 
35     M = [[0] * k for _ in range(k)]
36     M[0] = coeffs # First row
37     for i in range(1, k):
38         M[i][i-1] = 1 # Identity for shifting
39
40     # Initial state vector
41     state = init[::-1] # Reverse order
42
43     if n < k:
44         return init[n]
45
46     #  $M^{(n-k+1)}$ 
47     result_matrix = matpow(M, n - k + 1, mod)
48
49     # Multiply with initial state
50     result = 0
51     for i in range(k):
52         result = (result + result_matrix[0][i] * state[i]) % mod
53
54     return result
55
56 # Example: Tribonacci  $T(n) = T(n-1) + T(n-2) + T(n-3)$ 
57 def tribonacci(n, mod):
58     if n == 0: return 0
59     if n == 1 or n == 2: return 1
60
61     coeffs = [1, 1, 1]
62     init = [0, 1, 1]
63     return linear_recurrence(coeffs, 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/DP
6 sys.setrecursionlimit(10**6)
7
8 # Threading for higher stack limit (CAUTION: use carefully)
9 import threading
10 threading.stack_size(2**26) # 64MB
11 sys.setrecursionlimit(2**20)
12
13 # Deep copy (be careful with performance)
14 from copy import deepcopy
15 new_list = deepcopy(old_list)
16
17 # Fast output (for printing large results)
18 import sys
19 print = sys.stdout.write # Only use for string output
```

16.2 Useful Libraries

```
1 # Iterator tools - powerful combinations
2 from itertools import *
3
4 # permutations(iterable, r) - all r-length permutations
5 perms = list(permutations([1,2,3], 2))
6 # [(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)]
7
8 # combinations(iterable, r) - r-length combinations
9 combs = list(combinations([1,2,3], 2))
10 # [(1,2), (1,3), (2,3)]
11
12 # product - cartesian product
13 prod = list(product([1,2], ['a', 'b']))
14 # [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
15
16 # accumulate - running totals
17 acc = list(accumulate([1,2,3,4]))
18 # [1, 3, 6, 10]
19
20 # chain - flatten iterables
21 chained = list(chain([1,2], [3,4]))
22 # [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 positive
7 any(x > 0 for x in arr) # True if any positive
8
9 # Zip - parallel iteration
```

```

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

```

16.4 Common Pitfalls

```

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

```

```

33 # For many string mutations, use list
34 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 # vs
57 sum([x*x for x in range(10**6)]) # Creates full list
58
59 # Ternary operator
60 x = a if condition else b
61
62 # Dictionary get with default
63 count = d.get(key, 0) + 1
64
65 # Matrix rotation 90 degrees clockwise
66 def rotate_90(matrix):
67     return [list(row) for row in zip(*matrix[::-1])]
68
69 # Matrix transpose
70 def transpose(matrix):
71     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 + (p1[1] - p2[1])**2)
7
8 def cross_product(O, A, B):
9     # Cross product of vectors OA and OB
10    # Positive: counter-clockwise
11    # Negative: clockwise
12    # Zero: collinear
13    return (A[0] - O[0]) * (B[1] - O[1]) - \
14           (A[1] - O[1]) * (B[0] - O[0])
15
16 def dot_product(A, B, C, D):
17     # Dot product of vectors AB and CD
18     return (B[0] - A[0]) * (D[0] - C[0]) + \
19           (B[1] - A[1]) * (D[1] - C[1])
20
21 # Check if point is on segment
22 def on_segment(p, q, r):
23     # Check if q lies on segment pr
24     return (q[0] <= max(p[0], r[0]) and
25            q[0] >= min(p[0], r[0]) and
26            q[1] <= max(p[1], r[1]) and
27            q[1] >= min(p[1], r[1]))
28
29 # Segment intersection
30 def segments_intersect(p1, q1, p2, q2):
31     o1 = cross_product(p1, q1, p2)
32     o2 = cross_product(p1, q1, q2)
33     o3 = cross_product(p2, q2, p1)
34     o4 = cross_product(p2, q2, q1)
35
36     # General case
37     if o1 * o2 < 0 and o3 * o4 < 0:
38         return True
39
40     # Special cases (collinear)
41     if o1 == 0 and on_segment(p1, p2, q1):
42         return True
43     if o2 == 0 and on_segment(p1, q2, q1):
44         return True
45     if o3 == 0 and on_segment(p2, p1, q2):
46         return True
47     if o4 == 0 and on_segment(p2, q1, q2):
48         return True
49
50     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, then y
4
5     if len(points) <= 2:
6         return points
7
8     # Build lower hull
9     lower = []
10    for p in points:
11        while (len(lower) >= 2 and
12               cross_product(lower[-2], lower[-1], p) <= 0):
13            lower.pop()
14        lower.append(p)
15
16    # Build upper hull
17    upper = []
18    for p in reversed(points):
19        while (len(upper) >= 2 and
20               cross_product(upper[-2], upper[-1], p) <= 0):
21            upper.pop()
22        upper.append(p)
23
24    # Remove last point (duplicate of first)
25    return lower[:-1] + upper[:-1]
26
27 # Convex hull area
28 def polygon_area(points):
29     # Shoelace formula
30     n = len(points)
31     area = 0
32
33     for i in range(n):
34         j = (i + 1) % n
35         area += points[i][0] * points[j][1]
36         area -= points[j][0] * points[i][1]
37
38     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 - p1y) * (p2x - p1x) / \
16                                   (p2y - p1y) + p1x
17
18                    if p1x == p2x or x <= xinters:

```

```

19         inside = not inside
20
21     p1x, p1y = p2x, p2y
22
23     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=lambda p: p[0])
3      points_sorted_y = sorted(points, key=lambda p: p[1])
4
5      def closest_recursive(px, py):
6          n = len(px)
7
8          # Base case: brute force
9          if n <= 3:
10             min_dist = float('inf')
11             for i in range(n):
12                 for j in range(i + 1, n):
13                     min_dist = min(min_dist, dist(px[i], px[j]))
14             return min_dist
15
16         # Divide
17         mid = n // 2
18         midpoint = px[mid]
19
20         pyl = [p for p in py if p[0] <= midpoint[0]]
21         pyr = [p for p in py if p[0] > midpoint[0]]
22
23         # Conquer
24         dl = closest_recursive(px[:mid], pyl)
25         dr = closest_recursive(px[mid:], pyr)
26         d = min(dl, dr)
27
28         # Combine: check strip
29         strip = [p for p in py if abs(p[0] - midpoint[0]) < d]
30
31         for i in range(len(strip)):
32             j = i + 1
33             while j < len(strip) and strip[j][1] - strip[i][1] < d:
34                 d = min(d, dist(strip[i], strip[j]))
35                 j += 1
36
37         return d
38
39     return closest_recursive(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, defaultdict
2
3 def max_flow(graph, source, sink, n):
4     # graph[u][v] = capacity from u to v
5     # Build residual graph
6     residual = defaultdict(lambda: defaultdict(int))
7     for u in graph:
8         for v in graph[u]:
9             residual[u][v] = graph[u][v]
10
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 None:
24                     path.append((parent[u], u))
25                     u = parent[u]
26                 return path[::-1]
27
28             for v in range(n):
29                 if v not in visited and residual[u][v] > 0:
30                     visited.add(v)
31                     parent[v] = u
32                     queue.append(v)
33
34     return None
35
36 max_flow_value = 0
37
38 # Find augmenting paths
39 while True:
40     path = bfs_path()
41     if path is None:
42         break
43
44     # Find minimum capacity along path
45     flow = min(residual[u][v] for u, v in path)
46
47     # Update residual graph
48     for u, v in path:
49         residual[u][v] -= flow
50         residual[v][u] += flow
51
52     max_flow_value += flow
53
54 return max_flow_value
55
```

```

56 # Example usage
57 # graph[u][v] = capacity
58 graph = defaultdict(lambda: defaultdict(int))
59 graph[0][1] = 10
60 graph[0][2] = 10
61 graph[1][3] = 4
62 graph[1][4] = 8
63 graph[2][4] = 9
64 graph[3][5] = 10
65 graph[4][3] = 6
66 graph[4][5] = 10
67
68 n = 6 # Number of nodes
69 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, defaultdict
2
3 class Dinic:
4     def __init__(self, n):
5         self.n = n
6         self.graph = defaultdict(lambda: defaultdict(int))
7
8     def add_edge(self, u, v, cap):
9         self.graph[u][v] += cap
10
11     def bfs(self, source, sink):
12         # Build level graph
13         level = [-1] * self.n
14         level[source] = 0
15         queue = deque([source])
16
17         while queue:
18             u = queue.popleft()
19
20             for v in range(self.n):
21                 if level[v] == -1 and self.graph[u][v] > 0:
22                     level[v] = level[u] + 1
23                     queue.append(v)
24
25         return level if level[sink] != -1 else None
26
27     def dfs(self, u, sink, pushed, level, start):
28         if u == sink:
29             return pushed
30
31         while start[u] < self.n:
32             v = start[u]
33
34             if (level[v] == level[u] + 1 and
35                 self.graph[u][v] > 0):
36
37                 flow = self.dfs(v, sink,
38                               min(pushed, self.graph[u][v]),
39                               level, start)
40
41                 if flow > 0:
42                     self.graph[u][v] -= flow

```

```

43         self.graph[v][u] += flow
44         return flow
45
46     start[u] += 1
47
48     return 0
49
50 def max_flow(self, source, sink):
51     flow = 0
52
53     while True:
54         level = self.bfs(source, sink)
55         if level is None:
56             break
57
58         start = [0] * self.n
59
60         while True:
61             pushed = self.dfs(source, sink, float('inf'),
62                               level, start)
63             if pushed == 0:
64                 break
65             flow += pushed
66
67     return flow

```

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

```

18.4 Bipartite Matching

Description: Maximum matching in bipartite graph using flow.

```

1 def max_bipartite_matching(left_size, right_size, edges):
2     # edges = [(left_node, right_node), ...]
3     # Add source (0) and sink (left_size + right_size + 1)

```

```

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

```