Python ICPC Cheatsheet

Comprehensive Reference for Competitive Programming

Contents

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.

```python
# Fast I/O - Essential for large inputs
import sys
input = sys.stdin.readline

# Read single integer
n = int(input())

# Read multiple integers on one line
a, b = map(int, input().split())

# Read array of integers
arr = list(map(int, input().split()))

# Read strings (strip to remove trailing newline)
s = input().strip()
words = input().split()

# Multiple test cases pattern
```

```python
t = int(input())
for _ in range(t):
    # process each test case

# Print without newline
print(x, end=' ')

# Formatted output with precision
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.

```python
# Initialize lists
arr = [0] * n  # n zeros
matrix = [[0] * m for _ in range(n)]  # Correct way!

# List comprehension - concise and efficient
squares = [x**2 for x in range(n)]
evens = [x for x in arr if x % 2 == 0]

# Sorting - O(n log n)
arr.sort()  # in-place, modifies arr
arr.sort(reverse=True)  # descending
arr.sort(key=lambda x: (x[0], -x[1]))  # custom
sorted_arr = sorted(arr)  # returns new list

# Binary search in sorted array
from bisect import bisect_left, bisect_right
idx = bisect_left(arr, x)   # leftmost position
idx = bisect_right(arr, x)  # rightmost position

# Common operations
arr.append(x)       # O(1) amortized
arr.pop()           # O(1) - remove last
arr.pop(0)          # O(n) - remove first (slow!)
arr.reverse()       # O(n) - in-place
arr.count(x)        # O(n) - count occurrences
arr.index(x)        # O(n) - first occurrence
```

2.2 Deque (Double-ended Queue)

Description: Deque (pronounced ”deck”) provides O(1) append and pop operations from both ends, unlike lists which have O(n) for operations at the front. Essential for BFS, sliding window problems, implementing efficient queues/stacks, and maintaining monotonic queues. Use when you need fast insertions/deletions at both ends.

```python
from collections import deque
dq = deque()

# O(1) operations on both ends
dq.append(x)        # add to right
dq.appendleft(x)    # add to left
dq.pop()            # remove from right
dq.popleft()        # remove from left

# Sliding window maximum - O(n)
# Maintains decreasing order of elements
def sliding_max(arr, k):
    dq = deque()  # stores indices
    result = []

    for i in range(len(arr)):
        # Remove indices outside window
        while dq and dq[0] < i - k + 1:
            dq.popleft()

        # Remove smaller elements (not useful)
        while dq and arr[dq[-1]] < arr[i]:
            dq.pop()

        dq.append(i)
        if i >= k - 1:
            result.append(arr[dq[0]])

    return result
```

2.3 Heap (Priority Queue)

Description: Python's heapq module implements a min-heap (smallest element always at index 0). Provides O(log n) insert and extract-min operations, O(n) heapify, and O(1) peek. For max-heap, negate values before insertion. Critical for Dijkstra's algorithm, Prim's MST, k-th largest/smallest problems, merge k sorted lists, and any problem requiring repeated access to minimum/maximum elements. More efficient than sorting when you only need partial ordering.

```python
import heapq

# Min heap (default)
heap = []
heapq.heappush(heap, x)        # O(log n)
min_val = heapq.heappop(heap)  # O(log n)
min_val = heap[0]              # O(1) peek

# Max heap - negate values
heapq.heappush(heap, -x)
max_val = -heapq.heappop(heap)

# Convert list to heap in-place - O(n)
heapq.heapify(arr)

# K largest/smallest - O(n log k)
k_largest = heapq.nlargest(k, arr)
k_smallest = heapq.nsmallest(k, arr)

# Custom comparator using tuples
# Compares first element, then second, etc.
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.

```python
from collections import defaultdict, Counter

# defaultdict - provides default value
graph = defaultdict(list)  # empty list default
count = defaultdict(int)   # 0 default

# Counter - count elements efficiently
cnt = Counter(arr)
cnt['x'] += 1
most_common = cnt.most_common(k)  # k most frequent

# Dictionary operations
d = {}
d.get(key, default_val)
d.setdefault(key, default_val)
for k, v in d.items():
    pass
```

2.5 Set Operations

Description: Hash sets provide O(1) average-case membership testing, insertion, and deletion. Unlike lists, sets store only unique elements (no duplicates) and are unordered. Essential for removing duplicates, fast membership queries, and mathematical set operations (union, intersection, difference). Use when element uniqueness matters or you need fast lookups without caring about order. For sorted sets, consider using sorted containers or maintaining a sorted list separately.

```python
s = set()
s.add(x)         # O(1)
s.remove(x)      # O(1), KeyError if not exists
s.discard(x)     # O(1), no error if not exists

# Set operations - all O(n)
a | b    # union
a & b    # intersection
a - b    # difference
a ^ b    # symmetric difference

# Ordered set workaround
from collections import OrderedDict
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 $O(n^2)$.

```python
# Common string methods
s.lower(), s.upper()
s.strip()   # remove whitespace both ends
s.lstrip()  # remove left whitespace
s.rstrip()  # remove right whitespace
s.split(delimiter)
delimiter.join(list)
s.replace(old, new)
s.startswith(prefix)
s.endswith(suffix)
s.isdigit(), s.isalpha(), s.isalnum()

# String building - EFFICIENT O(n)
result = []
for x in data:
    result.append(str(x))
s = ''.join(result)

# String concatenation - SLOW O(n^2)
# s = ""
# for x in data:
#     s += str(x)  # Don't do this!

# ASCII values
ord('a')   # 97
chr(97)    # 'a'

# String to character array (for mutations)
chars = list(s)
chars[0] = 'x'
s = ''.join(chars)
```

3.1 KMP Pattern Matching

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

```python
def kmp_search(text, pattern):
    # Build LPS (Longest Proper Prefix which is Suffix)
    def build_lps(pattern):
        m = len(pattern)
        lps = [0] * m
        length = 0  # Length of previous longest prefix
        i = 1

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

        return lps

    n, m = len(text), len(pattern)
    lps = build_lps(pattern)

    matches = []
    i = j = 0  # Indices for text and pattern

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

        if j == m:
            matches.append(i - j)
            j = lps[j - 1]
        elif i < n and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    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).

```python
def z_algorithm(s):
    n = len(s)
    z = [0] * n
    l, r = 0, 0

    for i in range(1, n):
        if i <= r:
            z[i] = min(r - i + 1, z[i - l])

        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1

        if i + z[i] - 1 > r:
            l, r = i, i + z[i] - 1

    return z

# Pattern matching using Z-algorithm
def z_search(text, pattern):
    # Concatenate pattern + $ + text
    s = pattern + '$' + text
    z = z_algorithm(s)

    matches = []
    m = len(pattern)

    for i in range(m + 1, len(s)):
        if z[i] == m:
            matches.append(i - m - 1)

    return matches
```

3.3 Rabin-Karp (Rolling Hash)

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

```python
def rabin_karp(text, pattern):
    MOD = 10**9 + 7
    BASE = 31  # Prime base for hashing

    n, m = len(text), len(pattern)
    if m > n:
        return []

    # Compute hash of pattern
    pattern_hash = 0
    power = 1
    for i in range(m):
        pattern_hash = (pattern_hash * BASE +
                        ord(pattern[i])) % MOD
        if i < m - 1:
            power = (power * BASE) % MOD

    # Rolling hash
    text_hash = 0
    matches = []

    for i in range(n):
        # Add new character
        text_hash = (text_hash * BASE +
                     ord(text[i])) % MOD

        # Remove old character if window full
        if i >= m:
            text_hash = (text_hash -
                         ord(text[i - m]) * power) % MOD
            text_hash = (text_hash + MOD) % MOD

        # Check match
        if i >= m - 1 and text_hash == pattern_hash:
            # Verify actual match (avoid hash collision)
            if text[i - m + 1:i + 1] == pattern:
                matches.append(i - m + 1)

    return matches
```

4 Mathematics

4.1 Basic Math Operations

```
import math

# Common functions
math.ceil(x), math.floor(x)
math.gcd(a, b)      # Greatest common divisor
math.lcm(a, b)      # Python 3.9+
math.sqrt(x)
math.log(x), math.log2(x), math.log10(x)

# Powers
x ** y
pow(x, y, mod)  # (x^y) % mod - efficient modular exp

# Infinity
float('inf'), float('-inf')

# Custom GCD using Euclidean algorithm - O(log min(a,b))
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def lcm(a, b):
    return a * b // gcd(a, b)
```

4.2 Combinatorics

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

```
from math import factorial, comb, perm

# nCr (combinations) - "n choose r"
comb(n, r)  # Built-in Python 3.8+

# nPr (permutations)
perm(n, r)  # Built-in Python 3.8+

# Manual nCr implementation
def ncr(n, r):
    if r > n: return 0
    r = min(r, n - r)  # Optimization: C(n,r) = C(n,n-r)
    num = den = 1
    for i in range(r):
        num *= (n - i)
        den *= (i + 1)
    return num // den

# Precompute factorials with modulo
MOD = 10**9 + 7
def modfact(n):
    fact = [1] * (n + 1)
    for i in range(1, n + 1):
        fact[i] = fact[i-1] * i % MOD
    return fact

# Modular combination using precomputed factorials
# First precompute inverse factorials
def compute_inv_factorials(n, mod):
    fact = modfact(n)
    inv_fact = [1] * (n + 1)
    inv_fact[n] = pow(fact[n], mod - 2, mod)
    for i in range(n - 1, -1, -1):
        inv_fact[i] = inv_fact[i + 1] * (i + 1) % mod
    return fact, inv_fact

def modcomb(n, r, fact, inv_fact, mod):
    if r > n or r < 0: return 0
    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

```
# Modular inverse using Fermat's Little Theorem
# Only works when mod is prime
# a^(-1) = a^(mod-2) (mod p)
def modinv(a, mod):
    return pow(a, mod - 2, mod)

# Extended Euclidean Algorithm
# Returns (gcd, x, y) where ax + by = gcd(a,b)
# Can find modular inverse for any coprime a,mod
def extgcd(a, b):
    if b == 0:
        return a, 1, 0
    g, x1, y1 = extgcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    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).

```
def sieve(n):
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False

    for i in range(2, int(n**0.5) + 1):
        if is_prime[i]:
            # Mark multiples as composite
            for j in range(i*i, n + 1, i):
                is_prime[j] = False

    return is_prime

# Get list of primes
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.

```
def factorize(n):
    factors = []
    d = 2

    # Check divisors up to sqrt(n)
    while d * d <= n:
        while n % d == 0:
            factors.append(d)
            n //= d
        d += 1

    # If n > 1, it's a prime factor
    if n > 1:
        factors.append(n)

    return factors

# Get prime factors with counts
from collections import Counter
def prime_factor_counts(n):
    return Counter(factorize(n))

# Count divisors
def count_divisors(n):
    count = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            count += 1 if i * i == n else 2
        i += 1
    return count

# Sum of divisors
def sum_divisors(n):
    total = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            total += i
            if i != n // i:
                total += n // i
        i += 1
    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.

```python
def chinese_remainder(remainders, moduli):
    # Solve x = remainders[i] (mod moduli[i])
    # Assumes moduli are pairwise coprime

    def extgcd(a, b):
        if b == 0:
            return a, 1, 0
        g, x1, y1 = extgcd(b, a % b)
        return g, y1, x1 - (a // b) * y1

    total = 0
    prod = 1
    for m in moduli:
        prod *= m

    for r, m in zip(remainders, moduli):
        p = prod // m
        g, inv, _ = extgcd(p, m)
        # inv may be negative, normalize it
        inv = (inv % m + m) % m
        total += r * inv * p

    return total % prod
```

5.5 Euler's Totient Function

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

```python
def euler_phi(n):
    result = n
    p = 2

    while p * p <= n:
        if n % p == 0:
            # Remove factor p
            while n % p == 0:
                n //= p
            # Multiply by (1 - 1/p)
            result -= result // p
        p += 1

    if n > 1:
        result -= result // n

    return result

# Phi for range [1, n] using sieve
def phi_sieve(n):
    phi = list(range(n + 1))  # phi[i] = i initially

    for i in range(2, n + 1):
        if phi[i] == i:  # i is prime
            for j in range(i, n + 1, i):
                phi[j] = phi[j] // i * (i - 1)

    return phi
```

5.6 Fast Exponentiation with Matrix

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

```python
# Modular exponentiation
def mod_exp(base, exp, mod):
    result = 1
    base %= mod

    while exp > 0:
        if exp & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1

    return result
```

6 Graph Algorithms

6.1 Graph Representation

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

```python
from collections import defaultdict, deque

# Unweighted graph
graph = defaultdict(list)
for _ in range(m):
    u, v = map(int, input().split())
    graph[u].append(v)
    graph[v].append(u)  # for undirected

# Weighted graph - store (neighbor, weight) tuples
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).

```python
def bfs(graph, start):
    visited = set([start])
    queue = deque([start])
    dist = {start: 0}

    while queue:
        node = queue.popleft()

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                dist[neighbor] = dist[node] + 1

    return dist

# Grid BFS - common in maze/path problems
def grid_bfs(grid, start):
    n, m = len(grid), len(grid[0])
    visited = [[False] * m for _ in range(n)]
    queue = deque([start])
    visited[start[0]][start[1]] = True

    # 4 directions: right, down, left, up
    dirs = [(0,1), (1,0), (0,-1), (-1,0)]

    while queue:
        x, y = queue.popleft()

        for dx, dy in dirs:
            nx, ny = x + dx, y + dy

            # Check bounds and validity
            if (0 <= nx < n and 0 <= ny < m
                and not visited[nx][ny]
                and grid[nx][ny] != '#'):

                visited[nx][ny] = True
                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).

```python
# Recursive DFS
def dfs(graph, node, visited):
    visited.add(node)

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Iterative DFS using stack
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
```

```python

        if node not in visited:
            visited.add(node)

            for neighbor in graph[node]:
                if neighbor not in visited:
                    stack.append(neighbor)

# Cycle detection in undirected graph
def has_cycle(graph, n):
    visited = [False] * n

    def dfs(node, parent):
        visited[node] = True

        for neighbor in graph[node]:
            if not visited[neighbor]:
                if dfs(neighbor, node):
                    return True
            # Back edge to non-parent = cycle
            elif neighbor != parent:
                return True

        return False

    # Check all components
    for i in range(n):
        if not visited[i]:
            if dfs(i, -1):
                return True

    return False

# Cycle detection in directed graph
def has_cycle_directed(graph, n):
    WHITE, GRAY, BLACK = 0, 1, 2
    color = [WHITE] * n

    def dfs(node):
        color[node] = GRAY

        for neighbor in graph[node]:
            if color[neighbor] == GRAY:
                return True  # Back edge = cycle
            if color[neighbor] == WHITE:
                if dfs(neighbor):
                    return True

        color[node] = BLACK
        return False

    for i in range(n):
        if color[i] == WHITE:
            if dfs(i):
                return True
    return False

# Connected components count
def count_components(graph, n):
    visited = [False] * n
    count = 0

    def dfs(node):
        visited[node] = True
        for neighbor in graph[node]:
            if not visited[neighbor]:
                dfs(neighbor)

    for i in range(n):
        if not visited[i]:
            dfs(i)
            count += 1

    return count

# Bipartite check (2-coloring)
def is_bipartite(graph, n):
    color = [-1] * n

    def bfs(start):
        from collections import deque
        queue = deque([start])
        color[start] = 0

        while queue:
            node = queue.popleft()

            for neighbor in graph[node]:
                if color[neighbor] == -1:
                    color[neighbor] = 1 - color[node]
                    queue.append(neighbor)
                elif color[neighbor] == color[node]:
                    return False

        return True

    for i in range(n):
        if color[i] == -1:
            if not bfs(i):
                return False

    return True
```

6.4 Strongly Connected Components (SCC)

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

```python
def tarjan_scc(graph, n):
    index_counter = [0]
    stack = []
    lowlink = [0] * n
    index = [0] * n
    on_stack = [False] * n
    index_initialized = [False] * n
    sccs = []

    def strongconnect(v):
        index[v] = index_counter[0]
        lowlink[v] = index_counter[0]
        index_counter[0] += 1
        index_initialized[v] = True
        stack.append(v)
        on_stack[v] = True

        for w in graph[v]:
            if not index_initialized[w]:
                strongconnect(w)
                lowlink[v] = min(lowlink[v], lowlink[w])
            elif on_stack[w]:
                lowlink[v] = min(lowlink[v], index[w])

        if lowlink[v] == index[v]:
            scc = []
            while True:
                w = stack.pop()
                on_stack[w] = False
                scc.append(w)
                if w == v:
                    break
            sccs.append(scc)

    for v in range(n):
        if not index_initialized[v]:
            strongconnect(v)

    return sccs
```

6.5 Bridges and Articulation Points

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

```python
def find_bridges(graph, n):
    visited = [False] * n
    disc = [0] * n
    low = [0] * n
    parent = [-1] * n
    time = [0]
    bridges = []

    def dfs(u):
        visited[u] = True
        disc[u] = low[u] = time[0]
        time[0] += 1

        for v in graph[u]:
            if not visited[v]:
                parent[v] = u
                dfs(v)
```

```python
                low[u] = min(low[u], low[v])

                # Bridge condition
                if low[v] > disc[u]:
                    bridges.append((u, v))
            elif v != parent[u]:
                low[u] = min(low[u], disc[v])

    for i in range(n):
        if not visited[i]:
            dfs(i)

    return bridges

def find_articulation_points(graph, n):
    visited = [False] * n
    disc = [0] * n
    low = [0] * n
    parent = [-1] * n
    time = [0]
    ap = set()

    def dfs(u):
        children = 0
        visited[u] = True
        disc[u] = low[u] = time[0]
        time[0] += 1

        for v in graph[u]:
            if not visited[v]:
                children += 1
                parent[v] = u
                dfs(v)
                low[u] = min(low[u], low[v])

                # Articulation point conditions
                if parent[u] == -1 and children > 1:
                    ap.add(u)
                if parent[u] != -1 and low[v] >= disc[u]:
                    ap.add(u)
            elif v != parent[u]:
                low[u] = min(low[u], disc[v])

    for i in range(n):
        if not visited[i]:
            dfs(i)

    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).

```python
class LCA:
    def __init__(self, graph, root, n):
        self.n = n
        self.LOG = 20  # log2(n) + 1
        self.parent = [[-1] * self.LOG for _ in range(n)]
        self.depth = [0] * n

        # DFS to set parent and depth
        visited = [False] * n

        def dfs(node, par, d):
            visited[node] = True
            self.parent[node][0] = par
            self.depth[node] = d

            for neighbor in graph[node]:
                if not visited[neighbor]:
                    dfs(neighbor, node, d + 1)

        dfs(root, -1, 0)

        # Binary lifting preprocessing
        for j in range(1, self.LOG):
            for i in range(n):
                if self.parent[i][j-1] != -1:
                    self.parent[i][j] = self.parent[
                        self.parent[i][j-1]][j-1]

    def lca(self, u, v):
        # Make u deeper
        if self.depth[u] < self.depth[v]:
            u, v = v, u

        # Bring u to same level as v
        diff = self.depth[u] - self.depth[v]
        for i in range(self.LOG):
            if (diff >> i) & 1:
                u = self.parent[u][i]

        if u == v:
            return u

        # Binary search for LCA
        for i in range(self.LOG - 1, -1, -1):
            if self.parent[u][i] != self.parent[v][i]:
                u = self.parent[u][i]
                v = self.parent[v][i]

        return self.parent[u][0]

    def dist(self, u, v):
        # Distance between two nodes
        l = self.lca(u, v)
        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.

```python
import heapq

def dijkstra(graph, start, n):
    # Initialize distances to infinity
    dist = [float('inf')] * n
    dist[start] = 0

    # Min heap: (distance, node)
    heap = [(0, start)]

    while heap:
        d, node = heapq.heappop(heap)

        # Skip if already processed with better distance
        if d > dist[node]:
            continue

        # Relax edges
        for neighbor, weight in graph[node]:
            new_dist = dist[node] + weight

            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                heapq.heappush(heap, (new_dist, neighbor))

    return dist

# Path reconstruction
def dijkstra_with_path(graph, start, n):
    dist = [float('inf')] * n
    parent = [-1] * n
    dist[start] = 0
    heap = [(0, start)]

    while heap:
        d, node = heapq.heappop(heap)
        if d > dist[node]:
            continue

        for neighbor, weight in graph[node]:
            new_dist = dist[node] + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                parent[neighbor] = node
                heapq.heappush(heap, (new_dist, neighbor))

    return dist, parent

def reconstruct_path(parent, target):
    path = []
    while target != -1:
        path.append(target)
```

```python
        target = parent[target]
    return path[::-1]
```

7.2 Bellman-Ford Algorithm

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

```python
def bellman_ford(edges, n, start):
    # edges = [(u, v, weight), ...]
    dist = [float('inf')] * n
    dist[start] = 0

    # Relax edges n-1 times
    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] != float('inf') and \
               dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Check for negative cycles
    for u, v, w in edges:
        if dist[u] != float('inf') and \
           dist[u] + w < dist[v]:
            return None  # Negative cycle exists

    return dist
```

7.3 Floyd-Warshall Algorithm

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

```python
def floyd_warshall(n, edges):
    # Initialize distance matrix
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        dist[i][i] = 0

    for u, v, w in edges:
        dist[u][v] = min(dist[u][v], w)

    # Dynamic programming
    for k in range(n):  # Intermediate vertex
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j],
                                dist[i][k] + dist[k][j])

    return dist

# Check for negative cycle
def has_negative_cycle(dist, n):
    for i in range(n):
        if dist[i][i] < 0:
            return True
    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).

```python
def kruskal(n, edges):
    # edges = [(weight, u, v), ...]
    edges.sort()  # Sort by weight

    uf = UnionFind(n)
    mst_weight = 0
    mst_edges = []

    for weight, u, v in edges:
        if uf.union(u, v):
            mst_weight += weight
            mst_edges.append((u, v, weight))

    return mst_weight, mst_edges

class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True
```

7.4.2 Prim's Algorithm

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

```python
import heapq

def prim(graph, n):
    # graph[u] = [(v, weight), ...]
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, node)
    mst_weight = 0

    while min_heap:
        weight, u = heapq.heappop(min_heap)

        if visited[u]:
            continue

        visited[u] = True
        mst_weight += weight

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

    return mst_weight
```

8 Topological Sort

Description: Linear ordering of vertices in a DAG (Directed Acyclic Graph) such that for every edge u→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.

```python
from collections import deque

def topo_sort(graph, n):
    # Count incoming edges for each node
    indegree = [0] * n
    for u in range(n):
        for v in graph[u]:
            indegree[v] += 1

    # Start with nodes having no dependencies
    queue = deque([i for i in range(n)
                   if indegree[i] == 0])
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)

        # Remove this node from graph
        for neighbor in graph[node]:
            indegree[neighbor] -= 1

            # If neighbor has no more dependencies
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    # If not all nodes processed, cycle exists
```

```python
    return result if len(result) == n else []
```

8.2 DFS-based Topological Sort

Advantages: Simpler code, uses less space.

```python
def topo_dfs(graph, n):
    visited = [False] * n
    stack = []

    def dfs(node):
        visited[node] = True

        # Visit all neighbors first
        for neighbor in graph[node]:
            if not visited[neighbor]:
                dfs(neighbor)

        # Add to stack after visiting all descendants
        stack.append(node)

    # Process all components
    for i in range(n):
        if not visited[i]:
            dfs(i)

    # Reverse stack gives topological order
    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

```python
class UnionFind:
    def __init__(self, n):
        # Each node is its own parent initially
        self.parent = list(range(n))
        # Rank for union by rank optimization
        self.rank = [0] * n

    def find(self, x):
        # Path compression: point directly to root
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Find roots
        px, py = self.find(x), self.find(y)

        # Already in same set
        if px == py:
            return False

        # Union by rank: attach smaller tree under larger
        if self.rank[px] < self.rank[py]:
            px, py = py, px

        self.parent[py] = px

        # Increase rank if trees had equal rank
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1

        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)

    # Count number of disjoint sets
    def count_sets(self):
        return len(set(self.find(i)
                   for i in range(len(self.parent))))

# Example: Detect cycle in undirected graph
def has_cycle_uf(edges, n):
    uf = UnionFind(n)
    for u, v in edges:
        if uf.connected(u, v):
            return True  # Cycle found
        uf.union(u, v)
    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

```python
# Find FIRST position where check(mid) is True
def binary_search_first(left, right, check):
    while left < right:
        mid = (left + right) // 2

        if check(mid):
            right = mid  # Could be answer, search left
        else:
            left = mid + 1  # Not answer, search right

    return left

# Find LAST position where check(mid) is True
def binary_search_last(left, right, check):
    while left < right:
        mid = (left + right + 1) // 2  # Round up!

        if check(mid):
            left = mid  # Could be answer, search right
        else:
            right = mid - 1  # Not answer, search left

    return left

# Example: Integer square root
def sqrt_binary(n):
    left, right = 0, n

    while left < right:
        mid = (left + right + 1) // 2

        if mid * mid <= n:
            left = mid  # mid might be answer
        else:
            right = mid - 1

    return left

# Binary search on answer - common pattern
def min_days_to_make_bouquets(bloomDay, m, k):
    # Can we make m bouquets in 'days' days?
    def can_make(days):
        bouquets = consecutive = 0
        for bloom in bloomDay:
            if bloom <= days:
                consecutive += 1
                if consecutive == k:
                    bouquets += 1
                    consecutive = 0
            else:
                consecutive = 0
        return bouquets >= m

    if len(bloomDay) < m * k:
        return -1

    # Binary search on number of days
    return binary_search_first(
        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.

```python
def lis(arr):
    from bisect import bisect_left

    # dp[i] = smallest ending value of LIS of length i+1
    dp = []

    for x in arr:
        # Find position to place x
        idx = bisect_left(dp, x)

        if idx == len(dp):
            dp.append(x)  # Extend LIS
        else:
            dp[idx] = x   # Better ending for this length

    return len(dp)

# LIS with actual sequence
def lis_with_sequence(arr):
    from bisect import bisect_left

    n = len(arr)
    dp = []
    parent = [-1] * n
    dp_idx = []  # indices in dp

    for i, x in enumerate(arr):
        idx = bisect_left(dp, x)

        if idx == len(dp):
            dp.append(x)
            dp_idx.append(i)
        else:
            dp[idx] = x
            dp_idx[idx] = i

        if idx > 0:
            parent[i] = dp_idx[idx - 1]

    # Reconstruct sequence
    result = []
    idx = dp_idx[-1]
    while idx != -1:
        result.append(arr[idx])
        idx = parent[idx]

    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×capacity), Space: O(n×capacity).

```python
def knapsack(weights, values, capacity):
    n = len(weights)
    # dp[i][w] = max value using first i items,
    #            weight <= w
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            # Don't take item i-1
            dp[i][w] = dp[i-1][w]

            # Take item i-1 if it fits
            if weights[i-1] <= w:
                dp[i][w] = max(
                    dp[i][w],
                    dp[i-1][w - weights[i-1]] + values[i-1]
                )

    return dp[n][capacity]

# Space-optimized O(capacity)
def knapsack_optimized(weights, values, capacity):
    dp = [0] * (capacity + 1)

    for i in range(len(weights)):
        # Iterate backwards to avoid using updated values
        for w in range(capacity, weights[i] - 1, -1):
            dp[w] = max(dp[w],
                        dp[w - weights[i]] + values[i])

    return dp[capacity]
```

11.3 Edit Distance (Levenshtein Distance)

Description: Minimum operations (insert, delete, replace) to transform s1 to s2. Time: O(m×n), Space: O(m×n).

```python
def edit_dist(s1, s2):
    m, n = len(s1), len(s2)
    # dp[i][j] = edit distance of s1[:i] and s2[:j]
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Base cases: empty string transformations
    for i in range(m + 1):
        dp[i][0] = i  # Delete all
    for j in range(n + 1):
        dp[0][j] = j  # Insert all

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
                # Characters match, no operation needed
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(
                    dp[i-1][j],      # Delete from s1
                    dp[i][j-1],      # Insert into s1
                    dp[i-1][j-1]     # Replace in s1
                )

    return dp[m][n]
```

11.4 Longest Common Subsequence (LCS)

Description: Longest subsequence common to two sequences. Time: O(m×n).

```python
def lcs(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

# Reconstruct LCS
def lcs_string(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    # Backtrack
    result = []
    i, j = m, n
    while i > 0 and j > 0:
        if s1[i-1] == s2[j-1]:
            result.append(s1[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1

    return ''.join(reversed(result))
```

11.5 Coin Change

Description: Minimum coins to make amount, or count ways. Time: O(n×amount).

```python
# Minimum coins
```

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

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

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

# Count ways
def coin_change_ways(coins, amount):
    dp = [0] * (amount + 1)
    dp[0] = 1

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

    return dp[amount]
```

11.6 Palindrome Partitioning

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

```python
def min_palindrome_partition(s):
    n = len(s)

    # is_pal[i][j] = True if s[i:j+1] is palindrome
    is_pal = [[False] * n for _ in range(n)]

    # Every single character is palindrome
    for i in range(n):
        is_pal[i][i] = True

    # Check all substrings
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                is_pal[i][j] = (length == 2 or
                                is_pal[i+1][j-1])

    # dp[i] = min cuts for s[0:i+1]
    dp = [float('inf')] * n

    for i in range(n):
        if is_pal[0][i]:
            dp[i] = 0
        else:
            for j in range(i):
                if is_pal[j+1][i]:
                    dp[i] = min(dp[i], dp[j] + 1)

    return dp[n-1]
```

11.7 Subset Sum

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

```python
def subset_sum(arr, target):
    n = len(arr)
    dp = [[False] * (target + 1) for _ in range(n + 1)]

    # Base case: sum 0 is always achievable
    for i in range(n + 1):
        dp[i][0] = True

    for i in range(1, n + 1):
        for s in range(target + 1):
            # Don't take arr[i-1]
            dp[i][s] = dp[i-1][s]

            # Take arr[i-1] if possible
            if s >= arr[i-1]:
                dp[i][s] = dp[i][s] or dp[i-1][s - arr[i-1]]

    return dp[n][target]

# Space optimized
def subset_sum_optimized(arr, target):
    dp = [False] * (target + 1)
    dp[0] = True

    for num in arr:
        for s in range(target, num - 1, -1):
            dp[s] = dp[s] or dp[s - num]

    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.

```python
# 1D prefix sum
prefix = [0] * (n + 1)
for i in range(n):
    prefix[i + 1] = prefix[i] + arr[i]

# Range sum query [l, r] inclusive
range_sum = prefix[r + 1] - prefix[l]

# 2D prefix sum - for rectangle sum queries
def build_2d_prefix(matrix):
    n, m = len(matrix), len(matrix[0])
    prefix = [[0] * (m + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            prefix[i][j] = (matrix[i-1][j-1] +
                           prefix[i-1][j] +
                           prefix[i][j-1] -
                           prefix[i-1][j-1])

    return prefix

# Rectangle sum from (x1,y1) to (x2,y2) inclusive
def rect_sum(prefix, x1, y1, x2, y2):
    return (prefix[x2+1][y2+1] -
            prefix[x1][y2+1] -
            prefix[x2+1][y1] +
            prefix[x1][y1])
```

12.2 Difference Array

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

```python
# Initialize difference array
diff = [0] * (n + 1)

# Add 'val' to range [l, r]
def range_update(diff, l, r, val):
    diff[l] += val
    diff[r + 1] -= val

# After all updates, reconstruct array
def reconstruct(diff):
    result = []
    current = 0
    for i in range(len(diff) - 1):
        current += diff[i]
        result.append(current)
    return result

# Example: Multiple range updates
diff = [0] * (n + 1)
for l, r, val in updates:
    range_update(diff, l, r, val)
final_array = reconstruct(diff)
```

12.3 Sliding Window

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

```python
# Fixed size window
def max_sum_window(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum

    # Slide window: add right, remove left
    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i - k]
        max_sum = max(max_sum, window_sum)

    return max_sum
```

```python
# Variable size window - two pointers
def min_subarray_sum_geq_target(arr, target):
    left = 0
    current_sum = 0
    min_len = float('inf')

    for right in range(len(arr)):
        current_sum += arr[right]

        # Shrink window while condition holds
        while current_sum >= target:
            min_len = min(min_len, right - left + 1)
            current_sum -= arr[left]
            left += 1

    return min_len if min_len != float('inf') else 0

# Longest substring with at most k distinct chars
def longest_k_distinct(s, k):
    from collections import defaultdict

    left = 0
    char_count = defaultdict(int)
    max_len = 0

    for right in range(len(s)):
        char_count[s[right]] += 1

        # Shrink if too many distinct
        while len(char_count) > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                del char_count[s[left]]
            left += 1

        max_len = max(max_len, right - left + 1)

    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.

```python
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        # Tree size: 4n is safe upper bound
        self.tree = [0] * (4 * self.n)
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, start, end):
        if start == end:
            # Leaf node
            self.tree[node] = arr[start]
        else:
            mid = (start + end) // 2
            # Build left and right subtrees
            self.build(arr, 2*node+1, start, mid)
            self.build(arr, 2*node+2, mid+1, end)
            # Combine results (sum in this case)
            self.tree[node] = (self.tree[2*node+1] +
                               self.tree[2*node+2])

    def update(self, node, start, end, idx, val):
        if start == end:
            # Leaf node - update value
            self.tree[node] = val
        else:
            mid = (start + end) // 2
            if idx <= mid:
                # Update left subtree
                self.update(2*node+1, start, mid, idx, val)
            else:
                # Update right subtree
                self.update(2*node+2, mid+1, end, idx, val)
            # Recompute parent
            self.tree[node] = (self.tree[2*node+1] +
                               self.tree[2*node+2])

    def query(self, node, start, end, l, r):
        # No overlap
        if r < start or end < l:
            return 0

        # Complete overlap
        if l <= start and end <= r:
            return self.tree[node]

        # Partial overlap
        mid = (start + end) // 2
        left_sum = self.query(2*node+1, start, mid, l, r)
        right_sum = self.query(2*node+2, mid+1, end, l, r)
        return left_sum + right_sum

    # Public interface
    def update_val(self, idx, val):
        self.update(0, 0, self.n-1, idx, val)

    def range_sum(self, l, r):
        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.

```python
class FenwickTree:
    def __init__(self, n):
        self.n = n
        # 1-indexed for easier implementation
        self.tree = [0] * (n + 1)

    def update(self, i, delta):
        # Add delta to position i (1-indexed)
        while i <= self.n:
            self.tree[i] += delta
            # Move to next node: add LSB
            i += i & (-i)

    def query(self, i):
        # Get prefix sum up to i (1-indexed)
        s = 0
        while i > 0:
            s += self.tree[i]
            # Move to parent: remove LSB
            i -= i & (-i)
        return s

    def range_query(self, l, r):
        # Sum from l to r (1-indexed)
        return self.query(r) - self.query(l - 1)

# Usage example
bit = FenwickTree(n)
for i, val in enumerate(arr, 1):
    bit.update(i, val)

# Range sum [l, r] (1-indexed)
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.

```python
class TrieNode:
    def __init__(self):
        self.children = {}  # char -> TrieNode
        self.is_end = False  # End of word marker

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        # Insert word - O(len(word))
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):
        # Exact word search - O(len(word))
```

```python
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def starts_with(self, prefix):
        # Prefix search - O(len(prefix))
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

    # Find all words with given prefix
    def words_with_prefix(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return []
            node = node.children[char]

        # DFS to collect all words
        words = []
        def dfs(n, path):
            if n.is_end:
                words.append(prefix + path)
            for char, child in n.children.items():
                dfs(child, path + char)

        dfs(node, "")
        return words
```

13.4 Treap (Randomized Balanced BST)

Description: Ordered set/map with expected O(log n) insert, erase, search, k-th, and rank. Combines a BST by key and a heap by random priority. Stores unique keys; for multiset, store (key, uid) or maintain a count.

```python
import random

class TreapNode:
    __slots__ = ("key", "prio", "left", "right", "size")
    def __init__(self, key):
        self.key = key
        self.prio = random.randint(1, 1 << 30)
        self.left = None
        self.right = None
        self.size = 1

def _sz(t):
    return t.size if t else 0

def _upd(t):
    if t:
        t.size = 1 + _sz(t.left) + _sz(t.right)

def _merge(a, b):
    # assumes all keys in a < all keys in b
    if not a or not b:
        return a or b
    if a.prio > b.prio:
        a.right = _merge(a.right, b)
        _upd(a)
        return a
    else:
        b.left = _merge(a, b.left)
        _upd(b)
        return b

def _split(t, key):
    # returns (l, r): l has keys < key, r has keys >= key
    if not t:
        return (None, None)
    if key <= t.key:
        l, t.left = _split(t.left, key)
        _upd(t)
        return (l, t)
    else:
        t.right, r = _split(t.right, key)
        _upd(t)
        return (t, r)

def _erase(t, key):
    if not t:
        return None
    if key == t.key:
        return _merge(t.left, t.right)
    if key < t.key:
        t.left = _erase(t.left, key)
    else:
        t.right = _erase(t.right, key)
    _upd(t)
    return t

class Treap:
    def __init__(self):
        self.root = None

    def __len__(self):
        return _sz(self.root)

    def contains(self, key):
        t = self.root
        while t:
            if key == t.key:
                return True
            t = t.left if key < t.key else t.right
        return False

    def insert(self, key):
        if self.contains(key):
            return
        node = TreapNode(key)
        l, r = _split(self.root, key)
        self.root = _merge(_merge(l, node), r)

    def remove(self, key):
        self.root = _erase(self.root, key)

    def kth_smallest(self, k):
        # 0-indexed k
        t = self.root
        while t:
            ls = _sz(t.left)
            if k < ls:
                t = t.left
            elif k == ls:
                return t.key
            else:
                k -= ls + 1
                t = t.right
        return None  # k out of range

    def count_less_than(self, key):
        # number of keys < key
        t, cnt = self.root, 0
        while t:
            if key <= t.key:
                t = t.left
            else:
                cnt += 1 + _sz(t.left)
                t = t.right
        return cnt

    def lower_bound(self, key):
        # smallest key >= key; returns None if none
        t, ans = self.root, None
        while t:
            if t.key >= key:
                ans = t.key
                t = t.left
            else:
                t = t.right
        return ans

# Usage example
T = Treap()
for x in [5, 1, 7, 3]:
    T.insert(x)
T.contains(3)            # True
T.kth_smallest(1)        # 3  (0-indexed)
T.count_less_than(6)     # 3  (1,3,5)
T.remove(5)
len(T)                   # 3
```

14 Bit Manipulation

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

```python
# Check if i-th bit (0-indexed) is set
is_set = (n >> i) & 1

# Set i-th bit to 1
n |= (1 << i)

# Clear i-th bit (set to 0)
n &= ~(1 << i)

# Toggle i-th bit
n ^= (1 << i)

# Count set bits (popcount)
count = bin(n).count('1')
count = n.bit_count()  # Python 3.10+

# Get lowest set bit
lsb = n & -n  # Also n & (~n + 1)

# Remove lowest set bit
n &= (n - 1)

# Check if power of 2
is_pow2 = n > 0 and (n & (n - 1)) == 0

# Check if power of 4
is_pow4 = n > 0 and (n & (n-1)) == 0 and (n & 0x55555555) != 0

# Iterate over all subsets of set represented by mask
mask = (1 << n) - 1  # All bits set
submask = mask
while submask > 0:
    # Process submask
    submask = (submask - 1) & mask

# Iterate through all k-bit masks
def iterate_k_bits(n, k):
    mask = (1 << k) - 1
    while mask < (1 << n):
        # Process mask
        yield mask
        # Gosper's hack
        c = mask & -mask
        r = mask + c
        mask = (((r ^ mask) >> 2) // c) | r

# XOR properties
# a ^ a = 0 (number XOR itself is 0)
# a ^ 0 = a (number XOR 0 is itself)
# XOR is commutative and associative
# Find unique element when all others appear twice:
def find_unique(arr):
    result = 0
    for x in arr:
        result ^= x
    return result

# Subset enumeration
n = 5  # Number of elements
for mask in range(1 << n):
    subset = [i for i in range(n) if mask & (1 << i)]
    # Process subset

# Check parity (odd/even number of 1s)
def parity(n):
    count = 0
    while n:
        count ^= 1
        n &= n - 1
    return count  # 1 if odd, 0 if even

# Swap two numbers without temp variable
a, b = 5, 10
a ^= b
b ^= a
a ^= b
# Now a=10, b=5
```

15 Matrix Operations

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

15.1 Matrix Multiplication

```python
# Standard matrix multiplication - O(n^3)
def matmul(A, B):
    n, m, p = len(A), len(A[0]), len(B[0])
    C = [[0] * p for _ in range(n)]

    for i in range(n):
        for j in range(p):
            for k in range(m):
                C[i][j] += A[i][k] * B[k][j]

    return C

# With modulo
def matmul_mod(A, B, mod):
    n = len(A)
    C = [[0] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] = (C[i][j] +
                           A[i][k] * B[k][j]) % mod

    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.

```python
def matpow(M, n, mod):
    size = len(M)

    # Identity matrix
    result = [[1 if i==j else 0
                for j in range(size)]
               for i in range(size)]

    # Binary exponentiation
    while n > 0:
        if n & 1:
            result = matmul_mod(result, M, mod)
        M = matmul_mod(M, M, mod)
        n >>= 1

    return result

# Example: Fibonacci using matrix exponentiation
# F(n) = [[1,1],[1,0]]^n
def fibonacci(n, mod):
    if n == 0: return 0
    if n == 1: return 1

    M = [[1, 1], [1, 0]]
    result = matpow(M, n - 1, mod)
    return result[0][0]

# Linear recurrence: a(n) = c1*a(n-1) + c2*a(n-2) + ...
# Build transition matrix and use matrix exponentiation
def linear_recurrence(coeffs, init, n, mod):
    k = len(coeffs)

    if n < k:
        return init[n]

    # Transition matrix
    # [a(n), a(n-1), ..., a(n-k+1)]
    M = [[0] * k for _ in range(k)]
    M[0] = coeffs  # First row
    for i in range(1, k):
        M[i][i-1] = 1  # Identity for shifting

    # Initial state vector [a(k-1), a(k-2), ..., a(0)]
    state = init[k-1::-1]

    # M^(n-k+1)
    result_matrix = matpow(M, n - k + 1, mod)

```

```python
    # Multiply with initial state
    result = 0
    for i in range(k):
        result = (result + result_matrix[0][i] * state[i]) % mod

    return result

# Example: Tribonacci T(n) = T(n-1) + T(n-2) + T(n-3)
def tribonacci(n, mod):
    if n == 0: return 0
    if n == 1 or n == 2: return 1

    coeffs = [1, 1, 1]
    init = [0, 1, 1]
    return linear_recurrence(coeffs, init, n, mod)
```

16 Miscellaneous Tips

16.1 Python-Specific Optimizations

```python
# Fast input for large datasets
import sys
input = sys.stdin.readline

# Increase recursion limit for deep DFS/DP
sys.setrecursionlimit(10**6)

# Threading for higher stack limit (CAUTION: use carefully)
import threading
threading.stack_size(2**26)  # 64MB
sys.setrecursionlimit(2**20)

# Deep copy (be careful with performance)
from copy import deepcopy
new_list = deepcopy(old_list)

# Fast output (for printing large results)
import sys
print = sys.stdout.write  # Only use for string output
```

16.2 Useful Libraries

```python
# Iterator tools - powerful combinations
from itertools import *

# permutations(iterable, r) - all r-length permutations
perms = list(permutations([1,2,3], 2))
# [(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)]

# combinations(iterable, r) - r-length combinations
combs = list(combinations([1,2,3], 2))
# [(1,2), (1,3), (2,3)]

# product - cartesian product
prod = list(product([1,2], ['a','b']))
# [(1,'a'), (1,'b'), (2,'a'), (2,'b')]

# accumulate - running totals
acc = list(accumulate([1,2,3,4]))
# [1, 3, 6, 10]

# chain - flatten iterables
chained = list(chain([1,2], [3,4]))
# [1, 2, 3, 4]
```

16.3 Common Patterns

```python
# Lambda sorting with multiple keys
arr.sort(key=lambda x: (-x[0], x[1]))
# Sort by first desc, then second asc

# All/Any - short-circuit evaluation
all(x > 0 for x in arr)  # True if all positive
any(x > 0 for x in arr)  # True if any positive

# Zip - parallel iteration
for a, b in zip(list1, list2):
    pass

# Enumerate - index and value
for i, val in enumerate(arr):
    print(f"arr[{i}] = {val}")

# Custom comparison function
from functools import cmp_to_key

def compare(a, b):
    # Return -1 if a < b, 0 if equal, 1 if a > b
    if a + b > b + a:
        return -1
    return 1

arr.sort(key=cmp_to_key(compare))

# DefaultDict with lambda
from collections import defaultdict
d = defaultdict(lambda: float('inf'))

# Multiple assignment
a, b = b, a  # Swap
a, *rest, b = [1,2,3,4,5]  # a=1, rest=[2,3,4], b=5
```

16.4 Common Pitfalls

```python
# Integer division - floors toward negative infinity
print(7 // 3)    # 2
print(-7 // 3)   # -3 (not -2!)

# For ceiling division toward zero:
def div_ceil(a, b):
    return -(-a // b)

# Modulo with negative numbers
print((-5) % 3)  # 1 (not -2!)
print(5 % -3)    # -1

# List multiplication creates references!
matrix = [[0] * m] * n  # WRONG! All rows same object
matrix[0][0] = 1        # Changes all rows!

# Correct way
matrix = [[0] * m for _ in range(n)]

# Float comparison - don't use ==
a, b = 0.1 + 0.2, 0.3
print(a == b)  # False!

# Use epsilon comparison
eps = 1e-9
print(abs(a - b) < eps)  # True

# String immutability
s = "abc"
# s[0] = 'd'  # ERROR!
s = 'd' + s[1:]  # OK

# For many string mutations, use list
chars = list(s)
chars[0] = 'd'
s = ''.join(chars)

# Mutable default arguments - dangerous!
def func(arr=[]):  # WRONG!
    arr.append(1)
    return arr

# Each call modifies same list
print(func())  # [1]
print(func())  # [1, 1]

# Correct way
def func(arr=None):
    if arr is None:
        arr = []
    arr.append(1)
    return arr

# Generator expressions save memory
sum(x*x for x in range(10**6))  # Memory efficient
# vs
sum([x*x for x in range(10**6)])  # Creates full list

# Ternary operator
x = a if condition else b

# Dictionary get with default
count = d.get(key, 0) + 1

# Matrix rotation 90 degrees clockwise
```

```python
def rotate_90(matrix):
    return [list(row) for row in zip(*matrix[::-1])]

# Matrix transpose
def transpose(matrix):
    return [list(row) for row in zip(*matrix)]
```

16.5 Time Complexity Reference

Common time complexities (Python, rough guides for 1–2s limits):

- $O(1)$, $O(\log n)$: instant
- $O(n)$: usually fine up to $\sim 10^7$ operations (∼1s)
- $O(n \log n)$: OK for n up to several 10^5 depending on constants
- $O(n\sqrt{n})$: risky in Python (may be OK for n up to a few 10^4 with low constants)
- $O(n^2)$: often TLE for $n > 10^4$
- $O(2^n)$: TLE for $n > 20$ (unless heavy pruning/memoization)
- $O(n!)$: TLE for $n > 11$

Input size guidelines (Python-focused):

- $n \leq 12$: $O(n!)$ (brute-force permutations)
- $n \leq 20$: $O(2^n)$ (subset DP / bitmask DP)
- $n \leq 500$: $O(n^3)$ may sometimes pass for small constants
- $n \leq 5000$: $O(n^2)$ borderline; optimize heavily
- $n \leq 10^6$: $O(n \log n)$ common; $O(n)$ preferred when possible
- $n \leq 10^7$: $O(n)$ may be OK for tight loops
- $n > 10^7$: aim for $O(n)$ with very low constants, or $O(\log n)/O(1)$

Complexity examples (Python implementations)

- $O(1)$: array access, dictionary lookup, push/pop from list end.
- $O(\log n)$: binary search (bisect), heap push/pop (heapq), operations in sortedcontainers.
- $O(n)$: single-pass scans, two-pointers, prefix sums, counting frequencies (Counter).
- $O(n \log n)$: sorting (Timsort via sorted()/list.sort()), heap construction, divide-and-conquer merges.
- $O(n\sqrt{n})$: sqrt-decomposition queries, some Mo's algorithm variants (constant-sensitive).
- $O(n^2)$: nested loops for pairwise checks, naive DP on pairs (be cautious for $n > 10{,}000$).
- $O(n^3)$: triple loops (Floyd–Warshall), usually too slow unless $n \leq 200$.
- $O(2^n)$: bitmask DP, subset enumerations, recursion over subsets (recommended for $n \leq 20$).
- $O(n!)$: full permutations, exhaustive search over orderings (recommended for $n \leq 10$; occasionally up to 11).

How to use: This quick reference maps input size n (left) to typical feasible time complexities (right) for contest time limits (1–2s) targeting Python implementations. Use it to pick algorithmic approaches and to decide when to optimize or change strategy.

Notes on filling the table:

- Start by checking the problem's time limit and target language. These guidelines are Python-focused (assume roughly $\approx 10^7$ simple operations/s; actual throughput depends on implementation details and input shapes).
- Convert algorithm cost to operation count: roughly cost $= c \cdot f(n)$. If cost $>$ time_limit $\times$ ops_per_sec, it will TLE.
- When in doubt, aim one complexity class lower (e.g. prefer $O(n \log n)$ over $O(n^2)$ for n around 10^5).
- Consider memory limits—some faster algorithms use more memory (e.g. segment trees vs. Fenwick tree).
- For multivariate inputs, replace n with the product/dominant parameter (e.g. $n \cdot m$) and apply the same rules.
- If an algorithm theoretically fits but is close to the limit, try to reduce constant factors: use local variables, avoid heavy Python objects in inner loops, use built-in functions, or move hot code to PyPy/Cython if allowed.

17 Computational Geometry

17.1 Basic Geometry

Description: Fundamental geometric operations for 2D points.

```python
import math

# Point operations
def dist(p1, p2):
    # Euclidean distance
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def cross_product(O, A, B):
    # Cross product of vectors OA and OB
    # Positive: counter-clockwise
    # Negative: clockwise
    # Zero: collinear
    return (A[0] - O[0]) * (B[1] - O[1]) - \
           (A[1] - O[1]) * (B[0] - O[0])

def dot_product(A, B, C, D):
    # Dot product of vectors AB and CD
    return (B[0] - A[0]) * (D[0] - C[0]) + \
           (B[1] - A[1]) * (D[1] - C[1])

# Check if point is on segment
def on_segment(p, q, r):
    # Check if q lies on segment pr
    return (q[0] <= max(p[0], r[0]) and
            q[0] >= min(p[0], r[0]) and
            q[1] <= max(p[1], r[1]) and
            q[1] >= min(p[1], r[1]))

# Segment intersection
def segments_intersect(p1, q1, p2, q2):
    o1 = cross_product(p1, q1, p2)
    o2 = cross_product(p1, q1, q2)
    o3 = cross_product(p2, q2, p1)
    o4 = cross_product(p2, q2, q1)

    # General case
    if o1 * o2 < 0 and o3 * o4 < 0:
        return True

    # Special cases (collinear)
    if o1 == 0 and on_segment(p1, p2, q1):
        return True
    if o2 == 0 and on_segment(p1, q2, q1):
        return True
    if o3 == 0 and on_segment(p2, p1, q2):
        return True
    if o4 == 0 and on_segment(p2, q1, q2):
        return True

    return False
```

17.2 Convex Hull

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

```python
def convex_hull(points):
    # Graham's scan algorithm
    points = sorted(points)  # Sort by x, then y

    if len(points) <= 2:
        return points

    # Build lower hull
    lower = []
    for p in points:
        while (len(lower) >= 2 and
               cross_product(lower[-2], lower[-1], p) <= 0):
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while (len(upper) >= 2 and
               cross_product(upper[-2], upper[-1], p) <= 0):
            upper.pop()
        upper.append(p)

    # Remove last point (duplicate of first)
```

```
    return lower[:-1] + upper[:-1]

# Convex hull area
def polygon_area(points):
    # Shoelace formula
    n = len(points)
    area = 0

    for i in range(n):
        j = (i + 1) % n
        area += points[i][0] * points[j][1]
        area -= points[j][0] * points[i][1]

    return abs(area) / 2
```

17.3 Point in Polygon

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

```
def point_in_polygon(point, polygon):
    # Ray casting algorithm
    x, y = point
    n = len(polygon)
    inside = False

    p1x, p1y = polygon[0]
    for i in range(1, n + 1):
        p2x, p2y = polygon[i % n]

        if y > min(p1y, p2y):
            if y <= max(p1y, p2y):
                if x <= max(p1x, p2x):
                    if p1y != p2y:
                        xinters = (y - p1y) * (p2x - p1x) / \
                                  (p2y - p1y) + p1x

                    if p1x == p2x or x <= xinters:
                        inside = not inside

        p1x, p1y = p2x, p2y

    return inside
```

17.4 Closest Pair of Points

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

```
def closest_pair(points):
    points_sorted_x = sorted(points, key=lambda p: p[0])
    points_sorted_y = sorted(points, key=lambda p: p[1])

    def closest_recursive(px, py):
        n = len(px)

        # Base case: brute force
        if n <= 3:
            min_dist = float('inf')
            for i in range(n):
                for j in range(i + 1, n):
                    min_dist = min(min_dist, dist(px[i], px[j]))
            return min_dist

        # Divide
        mid = n // 2
        midpoint = px[mid]

        pyl = [p for p in py if p[0] <= midpoint[0]]
        pyr = [p for p in py if p[0] > midpoint[0]]

        # Conquer
        dl = closest_recursive(px[:mid], pyl)
        dr = closest_recursive(px[mid:], pyr)
        d = min(dl, dr)

        # Combine: check strip
        strip = [p for p in py if abs(p[0] - midpoint[0]) < d]

        for i in range(len(strip)):
            j = i + 1
            while j < len(strip) and strip[j][1] - strip[i][1] <
      d:
                d = min(d, dist(strip[i], strip[j]))
                j += 1

        return d

    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: $O(VE^2)$.

```
from collections import deque, defaultdict

def max_flow(graph, source, sink, n):
    # graph[u][v] = capacity from u to v
    # Build residual graph
    residual = defaultdict(lambda: defaultdict(int))
    for u in graph:
        for v in graph[u]:
            residual[u][v] = graph[u][v]

    def bfs_path():
        # Find augmenting path using BFS
        parent = {source: None}
        visited = {source}
        queue = deque([source])

        while queue:
            u = queue.popleft()

            if u == sink:
                # Reconstruct path
                path = []
                while parent[u] is not None:
                    path.append((parent[u], u))
                    u = parent[u]
                return path[::-1]

            for v in range(n):
                if v not in visited and residual[u][v] > 0:
                    visited.add(v)
                    parent[v] = u
                    queue.append(v)

        return None

    max_flow_value = 0

    # Find augmenting paths
    while True:
        path = bfs_path()
        if path is None:
            break

        # Find minimum capacity along path
        flow = min(residual[u][v] for u, v in path)

        # Update residual graph
        for u, v in path:
            residual[u][v] -= flow
            residual[v][u] += flow

        max_flow_value += flow

    return max_flow_value

# Example usage
# graph[u][v] = capacity
graph = defaultdict(lambda: defaultdict(int))
graph[0][1] = 10
graph[0][2] = 10
graph[1][3] = 4
graph[1][4] = 8
graph[2][4] = 9
graph[3][5] = 10
graph[4][3] = 6
graph[4][5] = 10

n = 6  # Number of nodes
result = max_flow(graph, 0, 5, n)
```

18.2 Dinic's Algorithm (Faster)

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

```python
from collections import deque, defaultdict

class Dinic:
    def __init__(self, n):
        self.n = n
        self.graph = defaultdict(lambda: defaultdict(int))

    def add_edge(self, u, v, cap):
        self.graph[u][v] += cap

    def bfs(self, source, sink):
        # Build level graph
        level = [-1] * self.n
        level[source] = 0
        queue = deque([source])

        while queue:
            u = queue.popleft()

            for v in range(self.n):
                if level[v] == -1 and self.graph[u][v] > 0:
                    level[v] = level[u] + 1
                    queue.append(v)

        return level if level[sink] != -1 else None

    def dfs(self, u, sink, pushed, level, start):
        if u == sink:
            return pushed

        while start[u] < self.n:
            v = start[u]

            if (level[v] == level[u] + 1 and
                self.graph[u][v] > 0):

                flow = self.dfs(v, sink,
                               min(pushed, self.graph[u][v]),
                               level, start)

                if flow > 0:
                    self.graph[u][v] -= flow
                    self.graph[v][u] += flow
                    return flow

            start[u] += 1

        return 0

    def max_flow(self, source, sink):
        flow = 0

        while True:
            level = self.bfs(source, sink)
            if level is None:
                break

            start = [0] * self.n

            while True:
                pushed = self.dfs(source, sink, float('inf'),
                                 level, start)
                if pushed == 0:
                    break
                flow += pushed

        return flow
```

18.3 Min Cut

Description: Find minimum cut after computing max flow.

```python
def min_cut(graph, source, n, residual):
    # After running max_flow, residual graph is available
    # Min cut = set of reachable nodes from source
    visited = [False] * n
    queue = deque([source])
    visited[source] = True

    while queue:
        u = queue.popleft()
        for v in range(n):
            if not visited[v] and residual[u][v] > 0:
                visited[v] = True
                queue.append(v)

    # Cut edges
    cut_edges = []
    for u in range(n):
        if visited[u]:
            for v in range(n):
                if not visited[v] and graph[u][v] > 0:
                    cut_edges.append((u, v))

    return cut_edges
```

18.4 Bipartite Matching

Description: Maximum matching in bipartite graph using flow.

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

    n = left_size + right_size + 2
    source = 0
    sink = n - 1

    graph = defaultdict(lambda: defaultdict(int))

    # Source to left nodes
    for i in range(1, left_size + 1):
        graph[source][i] = 1

    # Left to right edges
    for l, r in edges:
        graph[l + 1][left_size + r + 1] = 1

    # Right nodes to sink
    for i in range(1, right_size + 1):
        graph[left_size + i][sink] = 1

    return max_flow(graph, source, sink, n)
```