Python ICPC Cheatsheet

Comprehensive Reference for Competitive Programming

October 31, 2025

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
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.

```
# 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 provides O(1) append and pop from both ends, making it ideal for sliding window problems and implementing queues/stacks efficiently.

```
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]:
```

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

```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) membership testing and set operations.

```
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).

```
# 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)
```

4 Mathematics

4.1 Basic Math Operations

```python
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.

```python
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
```

```python
def modcomb(n, r, fact, modinv_fact):
    if r > n: return 0
    return fact[n] * modinv_fact[r] % MOD * modinv_fact[n-r] % MOD
```

5 Number Theory

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

5.1 Modular Arithmetic

```python
# 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).

```python
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.

```python
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)

```

```python
    return factors

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

6 Graph Algorithms

6.1 Graph Representation

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

```
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).

```
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
```

```python
                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()

        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
```

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

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
    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:
```

```python
        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).

```
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).

```
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]
```

12 Array Techniques

12.1 Prefix Sum

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

```
# 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.

```
# 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)
```

```python
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

# 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)
```

```python

    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))
        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
```

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

```

```python
# 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)
```

```python
    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)

    # 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
    state = init[::-1]  # Reverse order

    if n < k:
        return init[n]

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

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

    return result
```

16 Miscellaneous Tips

16.1 Python-Specific Optimizations

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

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

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

16.2 Useful Libraries

```
# 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

```
# 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
```

```python

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
```

```python

# 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
```