

# Python ICPC Cheatsheet

Comprehensive Reference for Competitive Programming

## Contents

<b>1</b>	<b>Input/Output</b>	<b>2</b>	7.4.1	Kruskal's Algo- rithm . . . . .	11
			7.4.2	Prim's Algorithm .	11
<b>2</b>	<b>Basic Data Structures</b>	<b>2</b>	<b>8</b>	<b>Topological Sort</b>	<b>11</b>
2.1	List Operations . . . . .	2	8.1	Kahn's Algorithm (BFS- based) . . . . .	11
2.2	Deque (Double-ended Queue) . . . . .	2	8.2	DFS-based Topological Sort . . . . .	11
2.3	Heap (Priority Queue) . .	3	<b>9</b>	<b>Union-Find (Disjoint Set Union)</b>	<b>12</b>
2.4	Dictionary & Counter . .	3	<b>10</b>	<b>Binary Search</b>	<b>12</b>
2.5	Set Operations . . . . .	3	10.1	Template for Finding First/Last Position . . . .	12
<b>3</b>	<b>String Operations</b>	<b>3</b>	<b>11</b>	<b>Dynamic Programming</b>	<b>13</b>
3.1	KMP Pattern Matching .	4	11.1	Longest Increasing Sub- sequence . . . . .	13
3.2	Z-Algorithm . . . . .	4	11.2	0/1 Knapsack . . . . .	13
3.3	Rabin-Karp (Rolling Hash)	4	11.3	Edit Distance (Leven- shtein Distance) . . . . .	14
<b>4</b>	<b>Mathematics</b>	<b>5</b>	11.4	Longest Common Subse- quence (LCS) . . . . .	14
4.1	Basic Math Operations . .	5	11.5	Coin Change . . . . .	14
4.2	Combinatorics . . . . .	5	11.6	Palindrome Partitioning .	14
<b>5</b>	<b>Number Theory</b>	<b>5</b>	11.7	Subset Sum . . . . .	15
5.1	Modular Arithmetic . . .	5	<b>12</b>	<b>Array Techniques</b>	<b>15</b>
5.2	Sieve of Eratosthenes . . .	6	12.1	Prefix Sum . . . . .	15
5.3	Prime Factorization . . .	6	12.2	Difference Array . . . . .	15
5.4	Chinese Remainder The- orem . . . . .	6	12.3	Sliding Window . . . . .	16
5.5	Euler's Totient Function .	6	<b>13</b>	<b>Advanced Data Structures</b>	<b>16</b>
5.6	Fast Exponentiation with Matrix . . . . .	7	13.1	Segment Tree . . . . .	16
<b>6</b>	<b>Graph Algorithms</b>	<b>7</b>	13.2	Fenwick Tree (Binary In- dexed Tree) . . . . .	17
6.1	Graph Representation . .	7	13.3	Trie (Prefix Tree) . . . . .	17
6.2	BFS (Breadth-First Search) . . . . .	7	13.4	Treap (Randomized Bal- anced BST) . . . . .	17
6.3	DFS (Depth-First Search)	7	<b>14</b>	<b>Bit Manipulation</b>	<b>18</b>
6.4	Strongly Connected Components (SCC) . . . .	8	<b>15</b>	<b>Matrix Operations</b>	<b>19</b>
6.5	Bridges and Articulation Points . . . . .	9	15.1	Matrix Multiplication . .	19
6.6	Lowest Common Ances- tor (LCA) . . . . .	9	15.2	Matrix Exponentiation . .	19
<b>7</b>	<b>Shortest Path Algorithms</b>	<b>10</b>	<b>16</b>	<b>Miscellaneous Tips</b>	<b>20</b>
7.1	Dijkstra's Algorithm . . .	10			
7.2	Bellman-Ford Algorithm .	10			
7.3	Floyd-Warshall Algorithm	10			
7.4	Minimum Spanning Tree .	11			

16.1 Python-Specific Optimizations . . . . .	20	17.3 Point in Polygon . . . . .	23
16.2 Useful Libraries . . . . .	20	17.4 Closest Pair of Points . . . . .	23
16.3 Common Patterns . . . . .	20		
16.4 Common Pitfalls . . . . .	20	<b>18 Network Flow</b>	<b>23</b>
16.5 Time Complexity Reference . . . . .	21	18.1 Maximum Flow - Edmonds-Karp (BFS-based Ford-Fulkerson) . . . . .	23
<b>17 Computational Geometry</b>	<b>22</b>	18.2 Dinic's Algorithm (Faster)	24
17.1 Basic Geometry . . . . .	22	18.3 Min Cut . . . . .	25
17.2 Convex Hull . . . . .	22	18.4 Bipartite Matching . . . . .	25

## 1 Input/Output

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

```

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

```

## 2 Basic Data Structures

### 2.1 List Operations

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

```

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

```

```

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.

```

1 from collections import deque
2 dq = deque()
3
4 # O(1) operations on both ends
5 dq.append(x) # add to right
6 dq.appendleft(x) # add to left
7 dq.pop() # remove from right
8 dq.popleft() # remove from left
9

```

```

10 # Sliding window maximum - O(n)
11 # Maintains decreasing order of elements
12 def sliding_max(arr, k):
13     dq = deque() # stores indices
14     result = []
15
16     for i in range(len(arr)):
17         # Remove indices outside window
18         while dq and dq[0] < i - k + 1:
19             dq.popleft()
20
21         # Remove smaller elements (not
22         # useful)
23         while dq and arr[dq[-1]] < arr[i]:
24             dq.pop()
25
26         dq.append(i)
27         if i >= k - 1:
28             result.append(arr[dq[0]])
29
30     return result

```

### 2.3 Heap (Priority Queue)

**Description:** Python's heapq 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.

```

1 import heapq
2
3 # Min heap (default)
4 heap = []
5 heapq.heappush(heap, x) # O(log n)
6 min_val = heapq.heappop(heap) # O(log n)
7 min_val = heap[0] # O(1) peek
8
9 # Max heap - negate values
10 heapq.heappush(heap, -x)
11 max_val = -heapq.heappop(heap)
12
13 # Convert list to heap in-place - O(n)
14 heapq.heapify(arr)
15
16 # K largest/smallest - O(n log k)
17 k_largest = heapq.nlargest(k, arr)
18 k_smallest = heapq.nsmallest(k, arr)
19
20 # Custom comparator using tuples
21 # Compares first element, then second,
22 # etc.
23 heapq.heappush(heap, (priority, item))

```

### 2.4 Dictionary & Counter

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

```

1 from collections import defaultdict, Counter
2
3 # defaultdict - provides default value
4 graph = defaultdict(list) # empty list
5 # default
6 count = defaultdict(int) # 0 default
7
8 # Counter - count elements efficiently
9 cnt = Counter(arr)
10 cnt['x'] += 1
11 most_common = cnt.most_common(k) # k
12 # most frequent
13
14 # Dictionary operations
15 d = {}
16 d.get(key, default_val)
17 d.setdefault(key, default_val)
18 for k, v in d.items():
19     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.

```

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

```

## 3 String Operations

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

```

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

```

### 3.1 KMP Pattern Matching

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

```

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

```

```

31         i += 1
32         j += 1
33
34         if j == m:
35             matches.append(i - j)
36             j = lps[j - 1]
37         elif i < n and text[i] !=
38             pattern[j]:
39             if j != 0:
40                 j = lps[j - 1]
41             else:
42                 i += 1
43
44     return matches

```

### 3.2 Z-Algorithm

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

```

1 def z_algorithm(s):
2     n = len(s)
3     z = [0] * n
4     l, r = 0, 0
5
6     for i in range(1, n):
7         if i <= r:
8             z[i] = min(r - i + 1, z[i -
9             1])
10
11         while i + z[i] < n and s[z[i]]
12             == s[i + z[i]]:
13             z[i] += 1
14
15         if i + z[i] - 1 > r:
16             l, r = i, i + z[i] - 1
17
18     return z
19
20 # Pattern matching using Z-algorithm
21 def z_search(text, pattern):
22     # Concatenate pattern + '$' + text
23     s = pattern + '$' + text
24     z = z_algorithm(s)
25
26     matches = []
27     m = len(pattern)
28
29     for i in range(m + 1, len(s)):
30         if z[i] == m:
31             matches.append(i - m - 1)
32
33     return matches

```

### 3.3 Rabin-Karp (Rolling Hash)

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

```

1 def rabin_karp(text, pattern):
2     MOD = 10**9 + 7
3     BASE = 31 # Prime base for hashing
4
5     n, m = len(text), len(pattern)
6     if m > n:
7         return []
8

```

```

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

```

## 4 Mathematics

### 4.1 Basic Math Operations

```

1  import math
2
3  # Common functions
4  math.ceil(x), math.floor(x)
5  math.gcd(a, b) # Greatest common
6  divisor
7  math.lcm(a, b) # Python 3.9+
8  math.sqrt(x)
9  math.log(x), math.log2(x), math.log10(x)
10
11 # Powers
12 x ** y
13 pow(x, y, mod) # (x^y) % mod -
14 # efficient modular exp
15
16 # Infinity
17 float('inf'), float('-inf')
18
19 # Custom GCD using Euclidean algorithm -
20 # O(log min(a,b))
21 def gcd(a, b):
22     while b:
23         a, b = b, a % b
24     return a
25
26 def lcm(a, b):
27     return a * b // gcd(a, b)

```

## 4.2 Combinatorics

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

```

1  from math import factorial, comb, perm
2
3  # nCr (combinations) - "n choose r"
4  comb(n, r) # Built-in Python 3.8+
5
6  # nPr (permutations)
7  perm(n, r) # Built-in Python 3.8+
8
9  # Manual nCr implementation
10 def ncr(n, r):
11     if r > n: return 0
12     r = min(r, n - r) # Optimization: C
13     (n, r) = C(n, n-r)
14     num = den = 1
15     for i in range(r):
16         num *= (n - i)
17         den *= (i + 1)
18     return num // den
19
20 # Precompute factorials with modulo
21 MOD = 10**9 + 7
22 def modfact(n):
23     fact = [1] * (n + 1)
24     for i in range(1, n + 1):
25         fact[i] = fact[i-1] * i % MOD
26     return fact
27
28 # Modular combination using precomputed
29 # factorials
30 # First precompute inverse factorials
31 def compute_inv_factorials(n, mod):
32     fact = modfact(n)
33     inv_fact = [1] * (n + 1)
34     inv_fact[n] = pow(fact[n], mod - 2,
35         mod)
36     for i in range(n - 1, -1, -1):
37         inv_fact[i] = inv_fact[i + 1] *
38             (i + 1) % mod
39     return fact, inv_fact
40
41 def modcomb(n, r, fact, inv_fact, mod):
42     if r > n or r < 0: return 0
43     return fact[n] * inv_fact[r] % mod *
44         inv_fact[n-r] % mod

```

## 5 Number Theory

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

### 5.1 Modular Arithmetic

```

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

```

```

10 def extgcd(a, b):
11     if b == 0:
12         return a, 1, 0
13     g, x1, y1 = extgcd(b, a % b)
14     x = y1
15     y = x1 - (a // b) * y1
16     return g, x, y

```

```

37 while i * i <= n:
38     if n % i == 0:
39         total += i
40         if i != n // i:
41             total += n // i
42     i += 1
43 return total

```

## 5.2 Sieve of Eratosthenes

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

```

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

```

## 5.3 Prime Factorization

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

```

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

```

## 5.4 Chinese Remainder Theorem

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

```

1 def chinese_remainder(remainders, moduli
2     ):
3     # Solve x = remainders[i] (mod
4     # moduli[i])
5     # Assumes moduli are pairwise
6     # coprime
7
8     def extgcd(a, b):
9         if b == 0:
10             return a, 1, 0
11         g, x1, y1 = extgcd(b, a % b)
12         return g, y1, x1 - (a // b) * y1
13
14     total = 0
15     prod = 1
16     for m in moduli:
17         prod *= m
18
19     for r, m in zip(remainders, moduli):
20         p = prod // m
21         g, inv, _ = extgcd(p, m)
22         # inv may be negative, normalize
23         it
24         inv = (inv % m + m) % m
25         total += r * inv * p
26
27     return total % prod

```

## 5.5 Euler's Totient Function

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

```

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

```

```

23     for i in range(2, n + 1):
24         if phi[i] == i: # i is prime
25             for j in range(i, n + 1, i):
26                 phi[j] = phi[j] // i * (
27                     i - 1)
28     return phi

```

## 5.6 Fast Exponentiation with Matrix

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

```

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

```

## 6 Graph Algorithms

### 6.1 Graph Representation

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

```

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

```

### 6.2 BFS (Breadth-First Search)

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

```

1  def bfs(graph, start):
2      visited = set([start])
3      queue = deque([start])
4      dist = {start: 0}
5
6      while queue:
7          node = queue.popleft()
8
9          for neighbor in graph[node]:
10             if neighbor not in visited:
11                 visited.add(neighbor)
12                 queue.append(neighbor)
13                 dist[neighbor] = dist[

```

```

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

```

### 6.3 DFS (Depth-First Search)

**Description:** Explores as far as possible along each branch. Used for connectivity, cycles, topological sort. Time:  $O(V+E)$ , Space:  $O(V)$ .

```

1  # Recursive DFS
2  def dfs(graph, node, visited):
3      visited.add(node)
4
5      for neighbor in graph[node]:
6          if neighbor not in visited:
7              dfs(graph, neighbor, visited)
8
9  # Iterative DFS using stack
10 def dfs_iterative(graph, start):
11     visited = set()
12     stack = [start]
13
14     while stack:
15         node = stack.pop()
16
17         if node not in visited:
18             visited.add(node)
19
20             for neighbor in graph[node]:
21                 if neighbor not in
22                 visited:
23                     stack.append(
24                     neighbor)
25
26 # Cycle detection in undirected graph
27 def has_cycle(graph, n):
28     visited = [False] * n
29
30     def dfs(node, parent):
31         visited[node] = True

```

```

30         for neighbor in graph[node]:
31             if not visited[neighbor]:
32                 if dfs(neighbor, node):
33                     return True
34             # Back edge to non-parent =
35             cycle
36         elif neighbor != parent:
37             return True
38
39         return False
40
41     # Check all components
42     for i in range(n):
43         if not visited[i]:
44             if dfs(i, -1):
45                 return True
46
47     return False
48
49 # Cycle detection in directed graph
50 def has_cycle_directed(graph, n):
51     WHITE, GRAY, BLACK = 0, 1, 2
52     color = [WHITE] * n
53
54     def dfs(node):
55         color[node] = GRAY
56
57         for neighbor in graph[node]:
58             if color[neighbor] == GRAY:
59                 return True # Back edge
60             if color[neighbor] == WHITE:
61                 if dfs(neighbor):
62                     return True
63
64         color[node] = BLACK
65         return False
66
67     for i in range(n):
68         if color[i] == WHITE:
69             if dfs(i):
70                 return True
71     return False
72
73 # Connected components count
74 def count_components(graph, n):
75     visited = [False] * n
76     count = 0
77
78     def dfs(node):
79         visited[node] = True
80         for neighbor in graph[node]:
81             if not visited[neighbor]:
82                 dfs(neighbor)
83
84     for i in range(n):
85         if not visited[i]:
86             dfs(i)
87             count += 1
88
89     return count
90
91 # Bipartite check (2-coloring)
92 def is_bipartite(graph, n):
93     color = [-1] * n
94
95     def bfs(start):
96         from collections import deque
97         queue = deque([start])
98         color[start] = 0
99
100         while queue:
101             node = queue.popleft()

```

```

102         for neighbor in graph[node]:
103             if color[neighbor] ==
104             -1:
105                 color[neighbor] = 1
106             - color[node]
107             queue.append(
108             neighbor)
109             elif color[neighbor] ==
110             color[node]:
111                 return False
112
113         return True
114
115     for i in range(n):
116         if color[i] == -1:
117             if not bfs(i):
118                 return False
119
120     return True

```

## 6.4 Strongly Connected Components (SCC)

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

```

1 def tarjan_scc(graph, n):
2     index_counter = [0]
3     stack = []
4     lowlink = [0] * n
5     index = [0] * n
6     on_stack = [False] * n
7     index_initialized = [False] * n
8     sccs = []
9
10    def strongconnect(v):
11        index[v] = index_counter[0]
12        lowlink[v] = index_counter[0]
13        index_counter[0] += 1
14        index_initialized[v] = True
15        stack.append(v)
16        on_stack[v] = True
17
18        for w in graph[v]:
19            if not index_initialized[w]:
20                strongconnect(w)
21            lowlink[v] = min(lowlink
22            [v], lowlink[w])
23            elif on_stack[w]:
24                lowlink[v] = min(lowlink
25                [v], index[w])
26
27        if lowlink[v] == index[v]:
28            scc = []
29            while True:
30                w = stack.pop()
31                on_stack[w] = False
32                scc.append(w)
33                if w == v:
34                    break
35            sccs.append(scc)
36
37    for v in range(n):
38        if not index_initialized[v]:
39            strongconnect(v)
40
41    return sccs

```



## 6.5 Bridges and Articulation Points

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

```

1 def find_bridges(graph, n):
2     visited = [False] * n
3     disc = [0] * n
4     low = [0] * n
5     parent = [-1] * n
6     time = [0]
7     bridges = []
8
9     def dfs(u):
10         visited[u] = True
11         disc[u] = low[u] = time[0]
12         time[0] += 1
13
14         for v in graph[u]:
15             if not visited[v]:
16                 parent[v] = u
17                 dfs(v)
18                 low[u] = min(low[u], low
19                             [v])
20
21                 # Bridge condition
22                 if low[v] > disc[u]:
23                     bridges.append((u, v
24                                     ))
25
26             elif v != parent[u]:
27                 low[u] = min(low[u],
28                             disc[v])
29
30     for i in range(n):
31         if not visited[i]:
32             dfs(i)
33
34     return bridges
35
36 def find_articulation_points(graph, n):
37     visited = [False] * n
38     disc = [0] * n
39     low = [0] * n
40     parent = [-1] * n
41     time = [0]
42     ap = set()
43
44     def dfs(u):
45         children = 0
46         visited[u] = True
47         disc[u] = low[u] = time[0]
48         time[0] += 1
49
50         for v in graph[u]:
51             if not visited[v]:
52                 children += 1
53                 parent[v] = u
54                 dfs(v)
55                 low[u] = min(low[u], low
56                             [v])
57
58             # Articulation point
59             conditions
60             if parent[u] == -1 and
61                 children > 1:
62                 ap.add(u)
63             if parent[u] != -1 and
64                 low[v] >= disc[u]:
65                 ap.add(u)
66             elif v != parent[u]:
67                 low[u] = min(low[u],
68                             disc[v])

```

```

60 for i in range(n):
61     if not visited[i]:
62         dfs(i)
63
64 return list(ap)
65

```

## 6.6 Lowest Common Ancestor (LCA)

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

```

1 class LCA:
2     def __init__(self, graph, root, n):
3         self.n = n
4         self.LOG = 20 # log2(n) + 1
5         self.parent = [[-1] * self.LOG
6                         for _ in range(n)]
7         self.depth = [0] * n
8
9         # DFS to set parent and depth
10        visited = [False] * n
11
12        def dfs(node, par, d):
13            visited[node] = True
14            self.parent[node][0] = par
15            self.depth[node] = d
16
17            for neighbor in graph[node]:
18                if not visited[neighbor
19                    ]:
20                    dfs(neighbor, node,
21                        d + 1)
22
23        dfs(root, -1, 0)
24
25        # Binary lifting preprocessing
26        for j in range(1, self.LOG):
27            for i in range(n):
28                if self.parent[i][j-1]
29                    != -1:
30                    self.parent[i][j] =
31                        self.parent[
32                            self.parent[i][j-1]
33                            ][j-1]
34
35        def lca(self, u, v):
36            # Make u deeper
37            if self.depth[u] < self.depth[v
38                ]:
39                u, v = v, u
40
41            # Bring u to same level as v
42            diff = self.depth[u] - self.
43            depth[v]
44            for i in range(self.LOG):
45                if (diff >> i) & 1:
46                    u = self.parent[u][i]
47
48            if u == v:
49                return u
50
51            # Binary search for LCA
52            for i in range(self.LOG - 1, -1,
53                -1):
54                if self.parent[u][i] != self
55                    .parent[v][i]:
56                    u = self.parent[u][i]
57                    v = self.parent[v][i]
58
59            return self.parent[u][0]

```

```

50 def dist(self, u, v):
51     # Distance between two nodes
52     l = self.lca(u, v)
53     return self.depth[u] + self.
54     depth[v] - 2 * self.depth[l]

```

## 7 Shortest Path Algorithms

### 7.1 Dijkstra's Algorithm

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

```

1 import heapq
2
3 def dijkstra(graph, start, n):
4     # Initialize distances to infinity
5     dist = [float('inf')] * n
6     dist[start] = 0
7
8     # Min heap: (distance, node)
9     heap = [(0, start)]
10
11     while heap:
12         d, node = heapq.heappop(heap)
13
14         # Skip if already processed with
15         # better distance
16         if d > dist[node]:
17             continue
18
19         # Relax edges
20         for neighbor, weight in graph[
21             node]:
22             new_dist = dist[node] +
23             weight
24             if new_dist < dist[neighbor
25             ]:
26                 dist[neighbor] =
27                 new_dist
28                 heapq.heappush(heap, (
29                 new_dist, neighbor))
30
31     return dist
32
33 # Path reconstruction
34 def dijkstra_with_path(graph, start, n):
35     dist = [float('inf')] * n
36     parent = [-1] * n
37     dist[start] = 0
38     heap = [(0, start)]
39
40     while heap:
41         d, node = heapq.heappop(heap)
42         if d > dist[node]:
43             continue
44
45         for neighbor, weight in graph[
46             node]:
47             new_dist = dist[node] +
48             weight
49             if new_dist < dist[neighbor
50             ]:
51                 dist[neighbor] =
52                 new_dist
53                 parent[neighbor] = node
54                 heapq.heappush(heap, (
55                 new_dist, neighbor))

```

```

46 return dist, parent
47
48 def reconstruct_path(parent, target):
49     path = []
50     while target != -1:
51         path.append(target)
52         target = parent[target]
53     return path[::-1]
54

```

### 7.2 Bellman-Ford Algorithm

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

```

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

```

### 7.3 Floyd-Warshall Algorithm

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

```

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

```

```
25     return False
```

## 7.4 Minimum Spanning Tree

### 7.4.1 Kruskal's Algorithm

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

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

### 7.4.2 Prim's Algorithm

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

```
1 import heapq
2
3 def prim(graph, n):
4     # graph[u] = [(v, weight), ...]
5     visited = [False] * n
6     min_heap = [(0, 0)] # (weight, node)
7
8     mst_weight = 0
9
10    while min_heap:
11        weight, u = heapq.heappop(
12            min_heap)
13
14        if visited[u]:
15            continue
```

```
13         continue
```

```
14
15     visited[u] = True
16     mst_weight += weight
17
18     for v, w in graph[u]:
19         if not visited[v]:
20             heapq.heappush(min_heap,
21                             (w, v))
22     return mst_weight
```

## 8 Topological Sort

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

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

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

```
1 from collections import deque
2
3 def topo_sort(graph, n):
4     # Count incoming edges for each node
5     indegree = [0] * n
6     for u in range(n):
7         for v in graph[u]:
8             indegree[v] += 1
9
10    # Start with nodes having no
11    # dependencies
12    queue = deque([i for i in range(n)
13                    if indegree[i] == 0])
14    result = []
15
16    while queue:
17        node = queue.popleft()
18        result.append(node)
19
20    # Remove this node from graph
21    for neighbor in graph[node]:
22        indegree[neighbor] -= 1
23
24    # If neighbor has no more
25    # dependencies
26    if indegree[neighbor] == 0:
27        queue.append(neighbor)
28
29    # If not all nodes processed, cycle
30    # exists
31    return result if len(result) == n
32    else []
```

### 8.2 DFS-based Topological Sort

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

```
1 def topo_dfs(graph, n):
2     visited = [False] * n
3     stack = []
4
```

```

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

```

## 9 Union-Find (Disjoint Set Union)

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

### Applications:

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

```

1  class UnionFind:
2      def __init__(self, n):
3          # Each node is its own parent
4          # initially
5          self.parent = list(range(n))
6          # Rank for union by rank
7          # optimization
8          self.rank = [0] * n
9
10     def find(self, x):
11         # Path compression: point
12         # directly to root
13         if self.parent[x] != x:
14             self.parent[x] = self.find(
15                 self.parent[x])
16         return self.parent[x]
17
18     def union(self, x, y):
19         # Find roots
20         px, py = self.find(x), self.find(
21             y)
22
23         # Already in same set
24         if px == py:
25             return False
26
27         # Union by rank: attach smaller
28         # tree under larger
29         if self.rank[px] < self.rank[py]:
30             px, py = py, px

```

```

26     self.parent[py] = px
27
28     # Increase rank if trees had
29     # equal rank
30     if self.rank[px] == self.rank[py]:
31         self.rank[px] += 1
32
33     return True
34
35     def connected(self, x, y):
36         return self.find(x) == self.find(
37             y)
38
39     # Count number of disjoint sets
40     def count_sets(self):
41         return len(set(self.find(i)
42             for i in range(len(
43                 self.parent))))
44
45     # Example: Detect cycle in undirected
46     # graph
47     def has_cycle_uf(edges, n):
48         uf = UnionFind(n)
49         for u, v in edges:
50             if uf.connected(u, v):
51                 return True # Cycle found
52             uf.union(u, v)
53         return False

```

## 10 Binary Search

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

### 10.1 Template for Finding First/Last Position

```

1  # Find FIRST position where check(mid)
2  # is True
3  def binary_search_first(left, right,
4  check):
5      while left < right:
6          mid = (left + right) // 2
7
8          if check(mid):
9              right = mid # Could be
10             answer, search left
11         else:
12             left = mid + 1 # Not answer
13             , search right
14
15     return left
16
17 # Find LAST position where check(mid) is
18 # True
19 def binary_search_last(left, right,
20 check):
21     while left < right:
22         mid = (left + right + 1) // 2 #
23         Round up!
24
25         if check(mid):
26             left = mid # Could be
27             answer, search right
28         else:
29             right = mid - 1 # Not
30             answer, search left

```

```

22     return left
23
24 # Example: Integer square root
25 def sqrt_binary(n):
26     left, right = 0, n
27
28     while left < right:
29         mid = (left + right + 1) // 2
30
31         if mid * mid <= n:
32             left = mid # mid might be
33             answer
34         else:
35             right = mid - 1
36
37     return left
38
39 # Binary search on answer - common
40 # pattern
41 def min_days_to_make_bouquets(bloomDay,
42                               m, k):
43     # Can we make m bouquets in 'days'
44     # days?
45     def can_make(days):
46         bouquets = consecutive = 0
47         for bloom in bloomDay:
48             if bloom <= days:
49                 consecutive += 1
50                 if consecutive == k:
51                     bouquets += 1
52                     consecutive = 0
53             else:
54                 consecutive = 0
55         return bouquets >= m
56
57     if len(bloomDay) < m * k:
58         return -1
59
60 # Binary search on number of days
61 return binary_search_first(
62     min(bloomDay), max(bloomDay),
63     can_make)

```

```

13     else:
14         dp[idx] = x # Better
15         ending for this length
16
17     return len(dp)
18
19 # LIS with actual sequence
20 def lis_with_sequence(arr):
21     from bisect import bisect_left
22
23     n = len(arr)
24     dp = []
25     parent = [-1] * n
26     dp_idx = [] # indices in dp
27
28     for i, x in enumerate(arr):
29         idx = bisect_left(dp, x)
30
31         if idx == len(dp):
32             dp.append(x)
33             dp_idx.append(i)
34         else:
35             dp[idx] = x
36             dp_idx[idx] = i
37
38         if idx > 0:
39             parent[i] = dp_idx[idx - 1]
40
41 # Reconstruct sequence
42 result = []
43 idx = dp_idx[-1]
44 while idx != -1:
45     result.append(arr[idx])
46     idx = parent[idx]
47
48 return result[::-1]

```

## 11 Dynamic Programming

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

### 11.1 Longest Increasing Subsequence

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

```

1 def lis(arr):
2     from bisect import bisect_left
3
4     # dp[i] = smallest ending value of
5     # LIS of length i+1
6     dp = []
7
8     for x in arr:
9         # Find position to place x
10        idx = bisect_left(dp, x)
11
12        if idx == len(dp):
13            dp.append(x) # Extend LIS

```

## 11.2 0/1 Knapsack

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

```

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

```

```

26     # Iterate backwards to avoid
    using updated values
27     for w in range(capacity, weights
    [i] - 1, -1):
28         dp[w] = max(dp[w],
29                     dp[w - weights[i]
    ] + values[i])
30
31     return dp[capacity]

```

### 11.3 Edit Distance (Levenshtein Distance)

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

```

1  def edit_dist(s1, s2):
2      m, n = len(s1), len(s2)
3      # dp[i][j] = edit distance of s1[:i]
    and s2[:j]
4      dp = [[0] * (n + 1) for _ in range(m
    + 1)]
5
6      # Base cases: empty string
    transformations
7      for i in range(m + 1):
8          dp[i][0] = i # Delete all
9      for j in range(n + 1):
10         dp[0][j] = j # Insert all
11
12     for i in range(1, m + 1):
13         for j in range(1, n + 1):
14             if s1[i-1] == s2[j-1]:
15                 # Characters match, no
    operation needed
16                 dp[i][j] = dp[i-1][j-1]
17             else:
18                 dp[i][j] = 1 + min(
19                     dp[i-1][j], #
    Delete from s1
20                     dp[i][j-1], #
    Insert into s1
21                     dp[i-1][j-1] #
    Replace in s1
    )
22
23     return dp[m][n]

```

### 11.4 Longest Common Subsequence (LCS)

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

```

1  def lcs(s1, s2):
2      m, n = len(s1), len(s2)
3      dp = [[0] * (n + 1) for _ in range(m
    + 1)]
4
5      for i in range(1, m + 1):
6          for j in range(1, n + 1):
7              if s1[i-1] == s2[j-1]:
8                  dp[i][j] = dp[i-1][j-1]
    + 1
9              else:
10                 dp[i][j] = max(dp[i-1][j]
    ], dp[i][j-1])
11
12     return dp[m][n]

```

```

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

```

### 11.5 Coin Change

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

```

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

```

### 11.6 Palindrome Partitioning

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

```

1  def min_palindrome_partition(s):
2      n = len(s)
3

```

```

4 # is_pal[i][j] = True if s[i:j+1] is
  palindrome
5 is_pal = [[False] * n for _ in range
  (n)]
6
7 # Every single character is
  palindrome
8 for i in range(n):
9     is_pal[i][i] = True
10
11 # Check all substrings
12 for length in range(2, n + 1):
13     for i in range(n - length + 1):
14         j = i + length - 1
15         if s[i] == s[j]:
16             is_pal[i][j] = (length
17                             == 2 or
18                             is_pal[i
19                             +1][j-1])
20
21 # dp[i] = min cuts for s[0:i+1]
22 dp = [float('inf')] * n
23
24 for i in range(n):
25     if is_pal[0][i]:
26         dp[i] = 0
27     else:
28         for j in range(i):
29             if is_pal[j+1][i]:
30                 dp[i] = min(dp[i],
31                             dp[j] + 1)
32
33 return dp[n-1]

```

## 11.7 Subset Sum

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

```

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

```

## 12 Array Techniques

### 12.1 Prefix Sum

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

```

1 # 1D prefix sum
2 prefix = [0] * (n + 1)
3 for i in range(n):
4     prefix[i + 1] = prefix[i] + arr[i]
5
6 # Range sum query [l, r] inclusive
7 range_sum = prefix[r + 1] - prefix[l]
8
9 # 2D prefix sum - for rectangle sum
  queries
10 def build_2d_prefix(matrix):
11     n, m = len(matrix), len(matrix[0])
12     prefix = [[0] * (m + 1) for _ in
13         range(n + 1)]
14
15     for i in range(1, n + 1):
16         for j in range(1, m + 1):
17             prefix[i][j] = (matrix[i-1][
18                 j-1] +
19                             prefix[i-1][j
20                             ] +
21                             prefix[i][j
22                             -1] -
23                             prefix[i-1][j
24                             -1])
25
26     return prefix
27
28 # Rectangle sum from (x1,y1) to (x2,y2)
  inclusive
29 def rect_sum(prefix, x1, y1, x2, y2):
30     return (prefix[x2+1][y2+1] -
31             prefix[x1][y2+1] -
32             prefix[x2+1][y1] +
33             prefix[x1][y1])

```

### 12.2 Difference Array

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

```

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

```

## 12.3 Sliding Window

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

```
1 # Fixed size window
2 def max_sum_window(arr, k):
3     window_sum = sum(arr[:k])
4     max_sum = window_sum
5
6     # Slide window: add right, remove left
7     for i in range(k, len(arr)):
8         window_sum += arr[i] - arr[i - k]
9         max_sum = max(max_sum, window_sum)
10
11     return max_sum
12
13 # Variable size window - two pointers
14 def min_subarray_sum_geq_target(arr, target):
15     left = 0
16     current_sum = 0
17     min_len = float('inf')
18
19     for right in range(len(arr)):
20         current_sum += arr[right]
21
22         # Shrink window while condition holds
23         while current_sum >= target:
24             min_len = min(min_len, right - left + 1)
25             current_sum -= arr[left]
26             left += 1
27
28     return min_len if min_len != float('inf') else 0
29
30 # Longest substring with at most k distinct chars
31 def longest_k_distinct(s, k):
32     from collections import defaultdict
33
34     left = 0
35     char_count = defaultdict(int)
36     max_len = 0
37
38     for right in range(len(s)):
39         char_count[s[right]] += 1
40
41         # Shrink if too many distinct
42         while len(char_count) > k:
43             char_count[s[left]] -= 1
44             if char_count[s[left]] == 0:
45                 del char_count[s[left]]
46             left += 1
47
48         max_len = max(max_len, right - left + 1)
49
50     return max_len
```

## 13 Advanced Data Structures

### 13.1 Segment Tree

**Description:** Supports range queries and point updates in  $O(\log n)$ . Can

be modified for range updates with lazy propagation.

```
1 class SegmentTree:
2     def __init__(self, arr):
3         self.n = len(arr)
4         # Tree size: 4n is safe upper bound
5         self.tree = [0] * (4 * self.n)
6         self.build(arr, 0, 0, self.n - 1)
7
8     def build(self, arr, node, start, end):
9         if start == end:
10             # Leaf node
11             self.tree[node] = arr[start]
12         else:
13             mid = (start + end) // 2
14             # Build left and right subtrees
15             self.build(arr, 2*node+1, start, mid)
16             self.build(arr, 2*node+2, mid+1, end)
17             # Combine results (sum in this case)
18             self.tree[node] = (self.tree[2*node+1] +
19                               self.tree[2*node+2])
20
21     def update(self, node, start, end, idx, val):
22         if start == end:
23             # Leaf node - update value
24             self.tree[node] = val
25         else:
26             mid = (start + end) // 2
27             if idx <= mid:
28                 # Update left subtree
29                 self.update(2*node+1, start, mid, idx, val)
30             else:
31                 # Update right subtree
32                 self.update(2*node+2, mid+1, end, idx, val)
33             # Recompute parent
34             self.tree[node] = (self.tree[2*node+1] +
35                               self.tree[2*node+2])
36
37     def query(self, node, start, end, l, r):
38         # No overlap
39         if r < start or end < l:
40             return 0
41
42         # Complete overlap
43         if l <= start and end <= r:
44             return self.tree[node]
45
46         # Partial overlap
47         mid = (start + end) // 2
48         left_sum = self.query(2*node+1, start, mid, l, r)
49         right_sum = self.query(2*node+2, mid+1, end, l, r)
50         return left_sum + right_sum
51
52 # Public interface
53 def update_val(self, idx, val):
54     self.update(0, 0, self.n-1, idx,
```



```

55         val)
56     def range_sum(self, l, r):
57         return self.query(0, 0, self.n
            -1, l, r)

```

### 13.2 Fenwick Tree (Binary Indexed Tree)

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

```

1  class FenwickTree:
2      def __init__(self, n):
3          self.n = n
4          # 1-indexed for easier
           implementation
5          self.tree = [0] * (n + 1)
6
7      def update(self, i, delta):
8          # Add delta to position i (1-
           indexed)
9          while i <= self.n:
10             self.tree[i] += delta
11             # Move to next node: add LSB
12             i += i & (-i)
13
14      def query(self, i):
15          # Get prefix sum up to i (1-
           indexed)
16          s = 0
17          while i > 0:
18             s += self.tree[i]
19             # Move to parent: remove LSB
20             i -= i & (-i)
21          return s
22
23      def range_query(self, l, r):
24          # Sum from l to r (1-indexed)
25          return self.query(r) - self.
           query(l - 1)
26
27      # Usage example
28      bit = FenwickTree(n)
29      for i, val in enumerate(arr, 1):
30          bit.update(i, val)
31
32      # Range sum [l, r] (1-indexed)
33      result = bit.range_query(l, r)

```

### 13.3 Trie (Prefix Tree)

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

```

1  class TrieNode:
2      def __init__(self):
3          self.children = {} # char ->
           TrieNode
4          self.is_end = False # End of
           word marker
5
6  class Trie:
7      def __init__(self):
8          self.root = TrieNode()
9
10     def insert(self, word):
11         # Insert word - O(len(word))

```

```

12     node = self.root
13     for char in word:
14         if char not in node.children
           :
15             node.children[char] =
           TrieNode()
16             node = node.children[char]
17             node.is_end = True
18
19     def search(self, word):
20         # Exact word search - O(len(word))
21         node = self.root
22         for char in word:
23             if char not in node.children
           :
24                 return False
25             node = node.children[char]
26         return node.is_end
27
28     def starts_with(self, prefix):
29         # Prefix search - O(len(prefix))
30         node = self.root
31         for char in prefix:
32             if char not in node.children
           :
33                 return False
34             node = node.children[char]
35         return True
36
37     # Find all words with given prefix
38     def words_with_prefix(self, prefix):
39         node = self.root
40         for char in prefix:
41             if char not in node.children
           :
42                 return []
43             node = node.children[char]
44
45         # DFS to collect all words
46         words = []
47         def dfs(n, path):
48             if n.is_end:
49                 words.append(prefix +
           path)
50             for char, child in n.
           children.items():
51                 dfs(child, path + char)
52
53         dfs(node, "")
54         return words

```

### 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.

```

1  import random
2
3  class TreapNode:
4      __slots__ = ("key", "prio", "left",
           "right", "size")
5      def __init__(self, key):
6          self.key = key
7          self.prio = random.randint(1, 1
           << 30)
8          self.left = None

```

```

9         self.right = None
10        self.size = 1
11
12    def _sz(t):
13        return t.size if t else 0
14
15    def _upd(t):
16        if t:
17            t.size = 1 + _sz(t.left) + _sz(t
18                .right)
19
20    def _merge(a, b):
21        # assumes all keys in a < all keys
22        in b
23        if not a or not b:
24            return a or b
25        if a.prio > b.prio:
26            a.right = _merge(a.right, b)
27            _upd(a)
28            return a
29        else:
30            b.left = _merge(a, b.left)
31            _upd(b)
32            return b
33
34    def _split(t, key):
35        # returns (l, r): l has keys < key,
36        r has keys >= key
37        if not t:
38            return (None, None)
39        if key <= t.key:
40            l, t.left = _split(t.left, key)
41            _upd(t)
42            return (l, t)
43        else:
44            t.right, r = _split(t.right, key)
45            _upd(t)
46            return (t, r)
47
48    def _erase(t, key):
49        if not t:
50            return None
51        if key == t.key:
52            return _merge(t.left, t.right)
53        if key < t.key:
54            t.left = _erase(t.left, key)
55        else:
56            t.right = _erase(t.right, key)
57            _upd(t)
58            return t
59
60    class Treap:
61        def __init__(self):
62            self.root = None
63
64        def __len__(self):
65            return _sz(self.root)
66
67        def contains(self, key):
68            t = self.root
69            while t:
70                if key == t.key:
71                    return True
72                t = t.left if key < t.key
73            else t.right
74            return False
75
76        def insert(self, key):
77            if self.contains(key):
78                return
79            node = TreapNode(key)
80            l, r = _split(self.root, key)
81            self.root = _merge(_merge(l,
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

## 14 Bit Manipulation

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

```

1  # Check if i-th bit (0-indexed) is set
2  is_set = (n >> i) & 1
3
4  # Set i-th bit to 1
5  n |= (1 << i)
6
7  # Clear i-th bit (set to 0)
8  n &= ~(1 << i)
9
10 # Toggle i-th bit
11 n ^= (1 << i)
12
13 # Count set bits (popcount)
14 count = bin(n).count('1')

```

```

15 count = n.bit_count() # Python 3.10+
16
17 # Get lowest set bit
18 lsb = n & -n # Also n & (~n + 1)
19
20 # Remove lowest set bit
21 n &= (n - 1)
22
23 # Check if power of 2
24 is_pow2 = n > 0 and (n & (n - 1)) == 0
25
26 # Check if power of 4
27 is_pow4 = n > 0 and (n & (n-1)) == 0 and
    (n & 0x55555555) != 0
28
29 # Iterate over all subsets of set
    represented by mask
30 mask = (1 << n) - 1 # All bits set
31 submask = mask
32 while submask > 0:
33     # Process submask
34     submask = (submask - 1) & mask
35
36 # Iterate through all k-bit masks
37 def iterate_k_bits(n, k):
38     mask = (1 << k) - 1
39     while mask < (1 << n):
40         # Process mask
41         yield mask
42         # Gosper's hack
43         c = mask & -mask
44         r = mask + c
45         mask = ((r ^ mask) >> 2) // c
        | r
46
47 # XOR properties
48 # a ^ a = 0 (number XOR itself is 0)
49 # a ^ 0 = a (number XOR 0 is itself)
50 # XOR is commutative and associative
51 # Find unique element when all others
    appear twice:
52 def find_unique(arr):
53     result = 0
54     for x in arr:
55         result ^= x
56     return result
57
58 # Subset enumeration
59 n = 5 # Number of elements
60 for mask in range(1 << n):
61     subset = [i for i in range(n) if
        mask & (1 << i)]
62     # Process subset
63
64 # Check parity (odd/even number of 1s)
65 def parity(n):
66     count = 0
67     while n:
68         count ^= 1
69         n &= n - 1
70     return count # 1 if odd, 0 if even
71
72 # Swap two numbers without temp variable
73 a, b = 5, 10
74 a ^= b
75 b ^= a
76 a ^= b
77 # Now a=10, b=5

```

## 15 Matrix Operations

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

recurrence relations.

### 15.1 Matrix Multiplication

```

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

```

### 15.2 Matrix Exponentiation

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

```

1 def matpow(M, n, mod):
2     size = len(M)
3
4     # Identity matrix
5     result = [[1 if i==j else 0
6         for j in range(size)]
7         for i in range(size)]
8
9     # Binary exponentiation
10    while n > 0:
11        if n & 1:
12            result = matmul_mod(result,
13                M, mod)
14            M = matmul_mod(M, M, mod)
15            n >>= 1
16
17    return result
18
19 # Example: Fibonacci using matrix
    exponentiation
20 # F(n) = [[1,1],[1,0]]^n
21 def fibonacci(n, mod):
22     if n == 0: return 0
23     if n == 1: return 1
24
25     M = [[1, 1], [1, 0]]
26     result = matpow(M, n - 1, mod)
27     return result[0][0]
28
29 # Linear recurrence: a(n) = c1*a(n-1) +
    c2*a(n-2) + ...
30 # Build transition matrix and use matrix
    exponentiation
def linear_recurrence(coeffs, init, n,

```

```

31     mod):
32     k = len(coeffs)
33
34     if n < k:
35         return init[n]
36
37     # Transition matrix
38     # [a(n), a(n-1), ..., a(n-k+1)]
39     M = [[0] * k for _ in range(k)]
40     M[0] = coeffs # First row
41     for i in range(1, k):
42         M[i][i-1] = 1 # Identity for
43         # shifting
44
45     # Initial state vector [a(k-1), a(k-2), ..., a(0)]
46     state = init[k-1::-1]
47
48     # M^(n-k+1)
49     result_matrix = matpow(M, n - k + 1,
50                             mod)
51
52     # Multiply with initial state
53     result = 0
54     for i in range(k):
55         result = (result + result_matrix
56                  [0][i] * state[i]) % mod
57
58     return result
59
60 # Example: Tribonacci T(n) = T(n-1) + T(
61 # n-2) + T(n-3)
62 def tribonacci(n, mod):
63     if n == 0: return 0
64     if n == 1 or n == 2: return 1
65
66     coeffs = [1, 1, 1]
67     init = [0, 1, 1]
68     return linear_recurrence(coeffs,
69                               init, n, mod)

```

## 16 Miscellaneous Tips

### 16.1 Python-Specific Optimizations

```

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

```

## 16.2 Useful Libraries

```

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

```

## 16.3 Common Patterns

```

1 # Lambda sorting with multiple keys
2 arr.sort(key=lambda x: (-x[0], x[1]))
3 # Sort by first desc, then second asc
4
5 # All/Any - short-circuit evaluation
6 all(x > 0 for x in arr) # True if all
7 # positive
8
9 any(x > 0 for x in arr) # True if any
10 # positive
11
12 # Zip - parallel iteration
13 for a, b in zip(list1, list2):
14     pass
15
16 # Enumerate - index and value
17 for i, val in enumerate(arr):
18     print(f"arr[{i}] = {val}")
19
20 # Custom comparison function
21 from functools import cmp_to_key
22
23 def compare(a, b):
24     # Return -1 if a < b, 0 if equal, 1
25     # if a > b
26     if a + b > b + a:
27         return -1
28     return 1
29
30 arr.sort(key=cmp_to_key(compare))
31
32 # DefaultDict with lambda
33 from collections import defaultdict
34 d = defaultdict(lambda: float('inf'))
35
36 # Multiple assignment
37 a, b = b, a # Swap
38 a, *rest, b = [1,2,3,4,5] # a=1, rest
39 # = [2,3,4], b=5

```

## 16.4 Common Pitfalls

```

1 # Integer division - floors toward
2 # negative infinity

```

```

2 | print(7 // 3)      # 2
3 | print(-7 // 3)     # -3 (not -2!)
4 |
5 | # For ceiling division toward zero:
6 | def div_ceil(a, b):
7 |     return -(-a // b)
8 |
9 | # Modulo with negative numbers
10 | print((-5) % 3)    # 1 (not -2!)
11 | print(5 % -3)      # -1
12 |
13 | # List multiplication creates references
14 | matrix = [[0] * m] * n # WRONG! All
    rows same object
15 | matrix[0][0] = 1    # Changes all
    rows!
16 |
17 | # Correct way
18 | matrix = [[0] * m for _ in range(n)]
19 |
20 | # Float comparison - don't use ==
21 | a, b = 0.1 + 0.2, 0.3
22 | print(a == b)      # False!
23 |
24 | # Use epsilon comparison
25 | eps = 1e-9
26 | print(abs(a - b) < eps) # True
27 |
28 | # String immutability
29 | s = "abc"
30 | # s[0] = 'd' # ERROR!
31 | s = 'd' + s[1:]    # OK
32 |
33 | # For many string mutations, use list
34 | chars = list(s)
35 | chars[0] = 'd'
36 | s = ''.join(chars)
37 |
38 | # Mutable default arguments - dangerous!
39 | def func(arr=[]): # WRONG!
40 |     arr.append(1)
41 |     return arr
42 |
43 | # Each call modifies same list
44 | print(func())      # [1]
45 | print(func())      # [1, 1]
46 |
47 | # Correct way
48 | def func(arr=None):
49 |     if arr is None:
50 |         arr = []
51 |     arr.append(1)
52 |     return arr
53 |
54 | # Generator expressions save memory
55 | sum(x*x for x in range(10**6)) # Memory
    efficient
56 | # vs
57 | sum([x*x for x in range(10**6)]) #
    Creates full list
58 |
59 | # Ternary operator
60 | x = a if condition else b
61 |
62 | # Dictionary get with default
63 | count = d.get(key, 0) + 1
64 |
65 | # Matrix rotation 90 degrees clockwise
66 | def rotate_90(matrix):
67 |     return [list(row) for row in zip(*
    matrix[::-1])]
68 |
69 | # Matrix transpose

```

```

70 | def transpose(matrix):
71 |     return [list(row) for row in zip(*
    matrix)]

```

## 16.5 Time Complexity Reference

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 ( $\sim 1$ s)
- $O(n \log n)$ : 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  $\text{cost} = c \cdot f(n)$ . If  $\text{cost} > \text{time\_limit} \times \text{ops\_per\_sec}$ , 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.

```
1 import math
2
3 # Point operations
```

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

### 17.2 Convex Hull

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

```
1 def convex_hull(points):
2     # Graham's scan algorithm
3     points = sorted(points) # Sort by x
4     # , then y
5
6     if len(points) <= 2:
7         return points
8
9     # Build lower hull
10    lower = []
11    for p in points:
12        while (len(lower) >= 2 and
13              cross_product(lower[-2],
14                            lower[-1], p) <= 0):
```

```

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

```

### 17.3 Point in Polygon

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

```

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

```

### 17.4 Closest Pair of Points

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

```

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

```

```

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

```

## 18 Network Flow

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

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

```

1 from collections import deque,
2   defaultdict
3
4 def max_flow(graph, source, sink, n):
5     # graph[u][v] = capacity from u to v
6     # Build residual graph
7     residual = defaultdict(lambda:
8                             defaultdict(int))
9     for u in graph:
10         for v in graph[u]:
11             residual[u][v] = graph[u][v]
12
13     def bfs_path():
14         # Find augmenting path using BFS
15         parent = {source: None}

```

```

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

```

## 18.2 Dinic's Algorithm (Faster)

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

```

1 from collections import deque,
2     defaultdict

```

```

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

```



```

64         break
65         flow += pushed
66
67     return flow

```

### 18.3 Min Cut

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

```

1 def min_cut(graph, source, n, residual):
2     # After running max_flow, residual
3     graph is available
4     # Min cut = set of reachable nodes
5     from source
6     visited = [False] * n
7     queue = deque([source])
8     visited[source] = True
9
10    while queue:
11        u = queue.popleft()
12        for v in range(n):
13            if not visited[v] and
14                residual[u][v] > 0:
15                visited[v] = True
16                queue.append(v)
17
18    # Cut edges
19    cut_edges = []
20    for u in range(n):
21        if visited[u]:
22            for v in range(n):
23                if not visited[v] and
24                    graph[u][v] > 0:
25                    cut_edges.append((u,
26                                    v))
27
28

```

```

23     return cut_edges

```

### 18.4 Bipartite Matching

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

```

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

```