Cheatsheet

白晨旭，2300011172

一、基础算法

表达式

中序表达式转后序

```python
pre = {'+': 1, '-': 1, '*': 2, '/': 2}
t = int(input())
for _ in range(t):
    expr = input()
    ans = []
    ops = []
    for char in expr:
        if char.isdigit() or char == '.':
            ans.append(char)
        elif char == '(':
            ops.append(char)
        elif char == ')':
            while ops and ops[-1] != '(':
                ans.append(ops.pop())
            ops.pop()
        else:
            while ops and ops[-1] != '(' and pre[ops[-1]] >= pre[char]:
                ans.append(ops.pop())
            ops.append(char)
    while ops:
        ans.append(ops.pop())
    print(''.join(ans))
```

逆波兰表达式求值

```python
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = []
        for token in tokens:
            if token in '+-*/':
                a, b = stack.pop(), stack.pop()
                stack.append(self.evaluate(b, a, token))
            else:
                stack.append(int(token))
        return int(stack[0])

    def evaluate(self, num1, num2, op):
        if op == "+":
            return num1 + num2
        elif op == "-":
            return num1 - num2
        elif op == "*":
            return num1 * num2
        elif op == "/":
            return int(int(num1) / int(num2))
```

滑动窗口

例题：大小为k的平均值大于等于阈值的连续子数组数目

```
# 数组为arr，窗口大小为k，阈值为threshold
def numOfSubarrays(arr, k, threshold):
    ans = s = 0
    for i, x in enumerate(arr):
        s += x
        if i < k - 1:
            continue
        if s >= k * threshold:
            ans += 1
        s -= arr[i - k + 1]
    return ans
```

归并排序

```
def merge_sort(arr):
    if len(arr) <= 1:
        return a,0
    mid = len(arr) // 2
    l, l_cnt = merge_sort(arr[:mid])
    r, r_cnt = merge_sort(arr[mid:])
    merged, merge_cnt = merge(l, r)
    return merged, l_cnt + r_cnt + merge_cnt

def merge(l, r):
    merged = []
    l_idx, r_idx = 0, 0
    inverse_cnt = 0
    while l_idx < len(l) and r_idx < len(r):
        if l[l_idx] <= r[r_idx]:
            merged.append(l[l_idx])
            l_idx += 1
        else:
            merged.append(r[r_idx])
            r_idx += 1
            inverse_cnt += len(l) - l_idx
    merged.extend(l[l_idx:])
    merged.extend(r[r_idx:])
    return merged, inverse_cnt
```

搜索

```python
def binary_search(nums, target):
    l, r = 0, len(nums) - 1
    while l <= r:
        m = (l + r) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            l = mid + 1
        else:
            r = mid - 1
    return -1
```

```python
def bfs(graph, start_node):
    queue = deque([start_node])
    vis = set()
    vis.add(start_node)

    while queue:
        node = queue.popleft()

        ########################
        #   process the node   #
        ########################

        for neighbor in graph[node]:
            if neighbor not in vis:
                vis.add(neighbor)
                queue.append(neighbor)
```

```python
def dfs(graph, node, vis=None):
    if vis is None:
        vis = set()
    vis.add(node)

    ########################
    #   process the node   #
    ########################

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

二、线性数据结构

链表

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

```

```

 class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def delete(self, value):
        if self.head is None:
            return
        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.value == value:
                    current.next = current.next.next
                    break
                current = current.next
    def display(self):
        current = self.head
        while current:
            print(current.value, end=" ")
            current = current.next
        print()
```

stack

```
stack = [] # 直接使用列表即可

class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[-1]
    def size(self):
        return len(self.items)
```

单调栈

特点与性质

1. **维护数据单调**：栈中存储的元素（或其下标）满足某种单调关系（递增或递减）。
2. **常见应用**：找每个元素右侧（或左侧）第一个比它大（或小）的值；直方图中最大矩形等。
3. **时间复杂度**：元素只会被压入或弹出至多一次，所以整体多为 O(n)。

代码模板：找每个元素右侧第一个更大值

```python
def next_greater_element(nums):
    """
    返回数组中每个元素右侧第一个更大的元素，
    若不存在则返回 -1。
    """
    stack = []          # 存放元素下标
    res = [-1]*len(nums)
    for i, val in enumerate(nums):
        # 若当前值大于栈顶对应值，则弹栈并更新结果
        while stack and val > nums[stack[-1]]:
            idx = stack.pop()
            res[idx] = val
        stack.append(i)
    return res
```

queue

```python
queue = deque() # 直接使用collections.deque即可

class Queue:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0, item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
```

三、树

```python
def TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

遍历

前序遍历：根-左子树-右子树

```python
def preorder(root):
    res = []
    def dfs(node):
        if not node: return
        res.append(node.value)
        dfs(node.left)
        dfs(node.right)
    dfs(root)
    return res
```

中序遍历：左子树-根-右子树

```python
def preorder(root):
    res = []
    def dfs(node):
        if not node: return
        dfs(node.left)
        res.append(node.value)
        dfs(node.right)
    dfs(root)
    return res
```

后序遍历：左子树-右子树-根

```python
def preorder(root):
    res = []
    def dfs(node):
        if not node: return
        dfs(node.left)
        dfs(node.right)
        res.append(node.value)
    dfs(root)
    return res
```

层序遍历

```python
def levelorder(root):
    if not root: return []
    queue = deque([root])
    res = []
    while queue:
        node = queue.pop()
        res.append(node.value)
        if node.left: queue.append(node.left)
        if node.right: queue.append(node.right)
    return res
```

Huffman树

```python

class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    # 为了在堆中进行比较
    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(freq_map):
    """
    根据 freq_map(字符->频率) 构建 Huffman 树并返回根节点
    """
    # 使用最小堆
    min_heap = []
    for ch, freq in freq_map.items():
        heapq.heappush(min_heap, HuffmanNode(ch, freq))

    # 合并节点直到堆内只剩一个
    while len(min_heap) > 1:
        left = heapq.heappop(min_heap)
        right = heapq.heappop(min_heap)
        parent = HuffmanNode(None, left.freq + right.freq)
        parent.left = left
        parent.right = right
        heapq.heappush(min_heap, parent)

    return min_heap[0]  # 最后一个元素即 Huffman 树的根

def build_codes(root):
    """
    根据 Huffman 树 生成 字符->编码 的字典
    """
    codes = {}
    def traverse(node, prefix):
        if node.char is not None:
            # 叶子节点
            codes[node.char] = prefix
            return
        if node.left:
            traverse(node.left, prefix + "0")
        if node.right:
            traverse(node.right, prefix + "1")
    traverse(root, "")
    return codes

def huffman_encode(text, codes):
    """
    根据编码表 codes，将文本编码为二进制字符串
    """
    encoded = []
    for ch in text:
        encoded.append(codes[ch])
    return "".join(encoded)

def huffman_decode(encoded_text, root):
    """
    根据 Huffman 树，对编码的二进制串进行解码
    """
    decoded_chars = []
    current = root
    for bit in encoded_text:
        if bit == '0':
            current = current.left
        else:
            current = current.right

        if current.char is not None:
            # 到达叶子节点
            decoded_chars.append(current.char)
            current = root
    return "".join(decoded_chars)
```

最近公共祖先

```python
def lowest_common_ancestor(root, p, q):
    """
    递归寻找最近公共祖先
    :param root: TreeNode, 当前树的根节点
    :param p: TreeNode, 节点 p
    :param q: TreeNode, 节点 q
    :return: TreeNode, 最近公共祖先节点
    """
    if not root or root == p or root == q:
        return root  # 如果当前节点是空，或是 p 或 q，则返回当前节点

    # 递归查找左右子树
    left = lowest_common_ancestor(root.left, p, q)
    right = lowest_common_ancestor(root.right, p, q)

    if left and right:
        return root  # 如果 p 和 q 分别在左右子树，则当前节点是最近公共祖先

    return left if left else right  # 否则返回非空的子树
```

二叉搜索树

```python
def bst_insert(root, val):
    if root is None:
        return TreeNode(val)
    if val < root.val:
        root.left = bst_insert(root.left, val)
    else:
        root.right = bst_insert(root.right, val)
    return root

def build_bst(arr):
    root = None
    for value in arr:
        root = bst_insert(root, value)
    return root
```

并查集

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n  # 用于启发式合并

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

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u == root_v:
            return False  # 已经在同一集合
        # 合并两个集合，使用 rank 优先
        if self.rank[root_u] < self.rank[root_v]:
            self.parent[root_u] = root_v
        elif self.rank[root_u] > self.rank[root_v]:
            self.parent[root_v] = root_u
        else:
            self.parent[root_v] = root_u
            self.rank[root_u] += 1
        return True
```

前缀树

```python
class TrieNode:
    def __init__(self):
        self.children = {}   # char -> TrieNode
        self.is_end = False  # 标记单词结尾

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

    def insert(self, word: str) -> None:
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True

    def search(self, word: str) -> bool:
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.is_end

    def startswith(self, prefix: str) -> bool:
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return True

    def is_prefix_of_other(self, word: str) -> bool:
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return bool(node.children)
# 使用示例
trie = Trie()
trie.insert("apple")
assert trie.search("apple")      # True
assert not trie.search("app")    # False
assert trie.startswith("app")    # True
```

四、图

环路检测

无向图

```
def has_cycle_undirected(n, adj):
    """
    n: 节点数
    adj: List[List[int]], 邻接表
    返回: bool, 是否存在环
    """
    visited = [False] * n

    def dfs(u, parent):
        visited[u] = True
        for v in adj[u]:
            if not visited[v]:
                if dfs(v, u):
                    return True
            elif v != parent:
                return True
        return False

    for i in range(n):
```

```python
        if not visited[i] and dfs(i, -1):
            return True
    return False
```

有向图

```python
def has_cycle_directed(n, adj):
    """
    n: 节点数
    adj: List[List[int]]，邻接表
    返回: bool，是否存在环
    """
    state = [0] * n  # 0=unvisited, 1=visiting, 2=visited

    def dfs(u):
        state[u] = 1
        for v in adj[u]:
            if state[v] == 0:
                if dfs(v):
                    return True
            elif state[v] == 1:
                return True
        state[u] = 2
        return False

    for i in range(n):
        if state[i] == 0 and dfs(i):
            return True
    return False
```

强连通分量

```python
def tarjan_scc(n, adj):
    """
    n: 节点数 (0..n-1)
    adj: List[List[int]]，有向图邻接表
    返回: List[List[int]]，各强连通分量列表
    """
    index = 0
    indices = [-1] * n
    lowlink = [0] * n
    onstack = [False] * n
    stack = []
    sccs = []

    def dfs(u):
        nonlocal index
        indices[u] = lowlink[u] = index
        index += 1
        stack.append(u)
        onstack[u] = True

        for v in adj[u]:
            if indices[v] == -1:
                dfs(v)
                lowlink[u] = min(lowlink[u], lowlink[v])
            elif onstack[v]:
                lowlink[u] = min(lowlink[u], indices[v])

        # 如果 u 是 SCC 根节点，就弹出直到 u
        if lowlink[u] == indices[u]:
            comp = []
            while True:
                w = stack.pop()
                onstack[w] = False
                comp.append(w)
                if w == u:
                    break
            sccs.append(comp)

    for i in range(n):
        if indices[i] == -1:
            dfs(i)

    return sccs
```

拓扑排序

```
def toposort(graph):
    in_degrees = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degrees[v] += 1
    queue = deque([u for u in in_degrees if in_degrees[u] == 0])
    topo_order = []
    while queue:
        u = queue.popleft()
        topo_order.append(u)
        for v in graph[u]:
            in_degrees[v] -= 1
            if in_degrees[v] == 0:
                queue.append(v)
    return topo_order if len(topo_order) == len(graph) else []
```

最短路径

Dijkstra

```python
# 使用vis集合，邻接表 graph = {u: [(v1, w1). (v2, w2), ...], ...}
def dijkstra(start, end):
    heap = [(0, start, [start])]
    vis = set()
    while heap:
        cost, u, path = heapq.heappop(heap)
        if u in vis: continue
        vis.add(u)
        if u == end: return cost, path
        for v, weight in graph[u]:
            if v not in vis:
                heapq.heappush(heap, (cost + weight, v, path + [v]))
```

```python
# 使用dist数组，邻接表 graph = {u: [(v1, w1). (v2, w2), ...], ...}
def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    heap = [(0, start)]
    while heap:
        cur_dist, cur_node = heapq.heappop(heap)
        if cur_dist > dist[cur_node]:
            continue
        for neighbor, weight in graph[cur_node]:
            distance = cur_dist + weight
            if distance < dist[neighbor]:
                dist[neighbor] = distance
                heapq.heappush(heap, (dist, neighbor))
    return dist
```

Bellman-Ford

```python
def bellman_ford(n, edges, source):
    """
    n:     节点数（节点编号假设为 0..n-1）
    edges:列表 of (u, v, w) 边
    source: 源点编号
    返回: dist 列表或抛出 ValueError(负权环)
    """
    INF = float('inf')
    dist = [INF] * n
    dist[source] = 0
    # 1. V-1 轮松弛
    for _ in range(n - 1):
        updated = False
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                updated = True
        if not updated:
            break
    # 2. 负权环检测
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            raise ValueError("Graph contains a negative-weight cycle")

    return dist
```

最小生成树

prim

```
def prim(graph):
    n = len(graph)
    vis = [False] * n
    min_heap = []
    mst_weight = 0
    mst_edges = []
    heapq.heappush(min_heap, (0, -1, 0))

    while min_heap and len(mst_edges) < n - 1:
        weight, parent, u = heapq.heappop(min_heap)
        if vis[u]: continue
        vis[u] = True

        mst_weight += weight
        if parent != -1:
            mst_edges.append((parent, u, weight))

        for v in range(n):
            if not vis[v] and graph[u][v] != float('inf'):
                heapq.heappush(min_heap, (graph[u][v], u, v))

    if len(mst_edges) != n - 1:
        raise ValueError('Graph is not connected')

    return mst_weight, mst_edges
```

krustal

```
def krustal(graph, n):
    edges = []
    for u in graph:
        for v, w in graph[u]:
            if u < v:
                edges.append((u, v, w))
    edges.sort(key=lambda edge: edge[2])

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

    for u, v, w in edges:
        if uf.union(u, v):
            mst_edges.append((u, v, w))
            mst_weight += w
            if len(mst_weight) == n - 1:
                break
    return mst_edges, mst_weight
```

五、散列表

线性探查法

原理：所有元素存放在一个连续数组，冲突时按固定偏移线性探测下一个槽。

- 哈希函数：`h(i) = (h0 + i) % B`，其中 `h0 = key % B`，`i=0,1,2…`
- 插入：从 `i=0` 开始探测，遇空槽或“已删”标记就放入；
- 查找 / 删除：同样探测直到空槽或找到对应 key。
- 删除：将槽标记为 `DELETED`，不直接置空。

```
def insert_hash_table(keys, M):
    table = [0.5] * M  # 用 0.5 表示空位
    result = []
    for key in keys:
        index = key % M
        i = index
        while True:
            if table[i] == 0.5 or table[i] == key:
                result.append(i)
                table[i] = key
                break
            i = (i + 1) % M
    return result
```

二次探查法

原理：与线性探测相似，但探测步长随 `i` 的平方增长，减少聚集。

- 哈希函数：`h(i) = (h0 + c1·i + c2·i²) % B`，常取 `c1=0, c2=1`。

```
mylist = [0.5] * m
 def generate_result():
    for num in num_list:
        pos = num % m
        current = mylist[pos]
        if current == 0.5 or current == num:
            mylist[pos] = num
            yield pos
        else:
            sign = 1
            cnt = 1
            while True:
                now = pos + sign * (cnt ** 2)
                current = mylist[now % m]
                if current == 0.5 or current == num:
                    mylist[now % m] = num
                    yield now % m
                    break
                sign *= -1
                if sign == 1:
                    cnt += 1
 result = generate_result()
 print(*result)
```

六、工具函数

lrucache

```python
from functools import lru_cache
@lrucache(maxsize=None)
def ...
```

bisect

```python
import bisect
# 创建一个有序列表
sorted_list = [1, 3, 4, 4, 5, 7]
# 使用bisect_left查找插入点
position = bisect.bisect_left(sorted_list, 4)
print(position)  # 输出: 2
# 使用bisect_right查找插入点
position = bisect.bisect_right(sorted_list, 4)
print(position)  # 输出: 4
# 使用insort_left插入元素
bisect.insort_left(sorted_list, 4)
print(sorted_list)  # 输出: [1, 3, 4, 4, 4, 5, 7]
# 使用insort_right插入元素
bisect.insort_right(sorted_list, 4)
print(sorted_list)  # 输出: [1, 3, 4, 4, 4, 4, 5, 7]
```

字符串操作

1. `str.lstrip() / str.rstrip()`: 移除字符串左侧/右侧的空白字符。
2. `str.find(sub)`: 返回子字符串`sub`在字符串中首次出现的索引，如果未找到，则返回-1。
3. `str.replace(old, new)`: 将字符串中的`old`子字符串替换为`new`。
4. `str.startswith(prefix) / str.endswith(suffix)`: 检查字符串是否以`prefix`开头或以`suffix`结尾。
5. `str.isalpha() / str.isdigit() / str.isalnum()`: 检查字符串是否全部由字母/数字/字母和数字组成。
6. `str.title()`: 每个单词首字母大写。

counter：计数

```python
from collections import Counter
# 创建一个Counter对象
count = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'apple'])
# 输出Counter对象
print(count)  # 输出: Counter({'apple': 3, 'banana': 2, 'orange': 1})
# 访问单个元素的计数
print(count['apple'])  # 输出: 3
# 访问不存在的元素返回0
print(count['grape'])  # 输出: 0
# 添加元素
count.update(['grape', 'apple'])
print(count)  # 输出: Counter({'apple': 4, 'banana': 2, 'orange': 1, 'grape': 1})
```

permutations：全排列

```python
from itertools import permutations
# 创建一个可迭代对象的排列
perm = permutations([1, 2, 3])
# 打印所有排列
for p in perm:
    print(p)
# 输出: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

combinations：组合

```python
from itertools import combinations
# 创建一个可迭代对象的组合
comb = combinations([1, 2, 3], 2)
# 打印所有组合
for c in comb:
    print(c)
# 输出: (1, 2), (1, 3), (2, 3)
```

reduce：累次运算

```python
from functools import reduce
# 使用reduce计算列表元素的乘积
product = reduce(lambda x, y: x * y, [1, 2, 3, 4])
print(product)  # 输出: 24
```

product：笛卡尔积

```python
from itertools import product
# 创建两个可迭代对象的笛卡尔积
prod = product([1, 2], ['a', 'b'])
# 打印所有笛卡尔积对
for p in prod:
    print(p)
# 输出：(1, 'a')，(1, 'b')，(2, 'a')，(2, 'b')
```