CHEAT SHEET

栈

波兰表达式-前缀表达式求值

```
list_1 = reversed(input().split())
stack = []
for i in list_1:
    if i in '+-*/':
        a, b = stack.pop(), stack.pop()
        stack.append(str(eval(a + i + b)))
    else:
        stack.append(i)
```

逆波兰-后缀表达式求值

```
stack=[]
for t in s:
    if t in '+-*/':
        b,a=stack.pop(),stack.pop()
        stack.append(str(eval(a+t+b)))
    else:
        stack.append(t)
```

Shunting Yard算法-中序表达式转后序表达式

```
def infix_to_postfix(expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    stack = []  # 栈顶元素优先级更高，优先压入postfix栈表示先进行该运算
    postfix = []
    number = ''
    for char in expression:  # 若有括号，从第一个右括号所在算式优先级最高
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''  # 当前数压入栈
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <=
precedence[stack[-1]]:
                    postfix.append(stack.pop())  # 栈中还有元素且没有碰到括号，仅依靠算
符优先级判断运算顺序
                stack.append(char)  # 栈顶算符优先级不高于自身时可以入栈
            elif char == '(':
                stack.append(char)  # 作为标志传入
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
```

```
                stack.pop()
    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)
    while stack:
        postfix.append(stack.pop())
    return ' '.join(str(x) for x in postfix)
```

单调栈-找到后方第一个比自己大/小的元素

```
for i in range(n):
    while mono_stack and arr[mono_stack[-1]] < arr[i]:
        f[mono_stack.pop()] = i + 1  # f[i]表示后方第一个比自己大/小元素的下标
    mono_stack.append(i)  # 单调栈中储存下标
```

树

为括号嵌套表达式建树

```
def build_parse_tree(fpexp):  # 构建括号表达式的解析树过程
    pstack = Stack()
    fplist = ''.join(fpexp.split())
    root = Node('')
    pstack.push(root)
    current_node = root
    for i in fplist:
        if i == '(':  # 左括号则为本节点添加空白左子节点并且下沉至该节点
            current_node.insert_left('')  # 本节点有左子树则将左子树连接在新节点左边，将新节点连于当前节点左边
            pstack.push(current_node)
            current_node = current_node.get_left()
        elif '0' <= i <= '9':  # 数字则将其设置为本节点的值并且返回至父节点
            current_node.set_value(int(i))
            current_node = pstack.pop()
        elif i in '+-*/':  # 运算符则将其设置为本节点的值并且为本节点添加右子节点并且下沉至该节点
            current_node.set_value(i)   # 赋值
            current_node.insert_right('')
            pstack.push(current_node)
            current_node = current_node.get_right()  # 获得右子树
        elif i == ')':  # 右括号则返回至上一级
            current_node = pstack.pop()
    return root

def evaluate(root):  # 根据解析树计算算式的值
    import operator
    ops = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
    left = root.get_left()
    right = root.get_right()
    return ops[root.value](evaluate(left), evaluate(right)) if left and right else root.value
```

并查集

```
def find(self, x):  # 查询
    if self.pa[x] == x:
        return x
    else:
        self.pa[x] = self.find(self.pa[x])
        return self.pa[x]
```

```
def union(self, x, y):  # 合并
    self.pa[self.find(x)] = self.find(y)
```

Huffman编码树

```
import heapq  # priority queue


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

    def __lt__(self, other):  # 定义大小关系
        return self.freq < other.freq


def huffman_encode(char_freq):  # 根据字典建立Huffman Tree
    heap = [Node(char, freq) for char, freq in char_freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]


def external_path_length(node, depth=0):  # 计算带权外部路径长度
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth * node.freq
    return external_path_length(node.left, depth + 1) +
external_path_length(node.right, depth + 1)
```

字典树-Trie Tree

```
class TrieTree:  # 用于查找当前字符串在字符串集中是否出现过
    def __init__(self):
        self.root = TrieNode()  # TrieNode.children = dict()

    def insert(self, word):
        cur = self.root
        for char in word:
            if cur.children.get(char) is None:
                cur.children[char] = TrieNode()
            cur = cur.children[char]

    def find(self, word):  # 此处查找是否以完整word作为已出现的字符串的子串；
        cur = self.root  # 若要查找是否存在完全相等两串，只要判断最终节点是否为叶节点。
        for char in word:
            if cur.children.get(char) is None:
                return False
            cur = cur.children[char]
        return True
```

Graph 及具体问题

bfs解决无权图最短路-词梯问题

```
for word in words:
    for i in range(4):  # 利用字典，将相差一个字母的两个单词储存在同一个关键字下的同一集合中，表示两个词之间有一条边相连。
        graph.setdefault(word[:i] + '_' + word[i + 1:], set()).add(word)
```

拓扑排序-Kahn算法

根据有向无环图生成一个包含所有点的序列，这个序列保证如果图中存在边$(V1,\ V2)$,序列中$V1$一定在$V2$之前。

1、计算每个顶点的入度。

2、创建队列，将入度为0的顶点加入队列。

3、创建结果列表。当队列不为空时，从队列中弹出顶点，加入结果列表中，并将该顶点相连的所有顶点入度-1.

如果相连顶点入度为0，则也将之加入队列中，重复操作。

程序终止lie时，若结果列表中含有所有顶点，则排序成功，图中没有环；否，则图中存在环。

```
def kahn():  # 此处只统计了序列长度
    cnt = 0
    while queue:
        front = queue.pop(0)
        cnt += 1
        for vertex in graph[front]:
            degree[vertex] -= 1
            if not degree[vertex]:
                queue.append(vertex)
    return cnt
```

判断图中是否有环

无向图：记录直接前驱的dfs / 并查集dfs

有向图：拓扑排序，结果列表中含有所有顶点，则排序成功，图中没有环；否，则图中存在环。

Dijkstra：非负边权图的单源最短路

初始化最短距离数组存储答案，除起点外到所有节点路径长度记录为无穷大；设置优先队列，队列元素为储存最短路径和点的元组，每一次从队列中弹出距离最小点，该点就是已经确定最短路的点。弹出后，将该点标记为已找到最短路，之后不再访问；对于该点的所有邻居，更新最短距离数组中的长度，如果小于当前值，将其加入优先队列中。**时间复杂度**：堆优化后$O(VlogE)$

```
def dijkstra(start, end):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    pq = [(0, start)]
    while pq:
        current_d, current_v = heapq.heappop(pq)
        if current_d > dist[current_v]:
            continue
        if current_v == end:
            return dist[end]
        for neighbor, d in graph[current_v].items():  # 此处键值为字符串，图结构字典嵌套
            new_dist = current_d + d
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))
    return -1
```

Floyd：负权图的多源最短路

初始化：$i=j$，为0；有边为边权，无边为无穷。**时间复杂度**：$O(V^3)$，常数较小

状态转移方程：用$dp[i][j]$表示从点i到j的最短距离，则每轮更新：
$dp[i][j]=min(dp[i][j],\ dp[i][k]+dp[k][j])$

```
 for k in range(v):
    for i in range(v):
        for j in range(v):
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j])
```

Prim

时间复杂度：堆优化后$ElogV$,适用于稠密图。

```
def prim(start):  # 从起点开始，每次选择连接已访问过的点到未访问过的点的边集中最短的一条边。
    from heapq import heappop, heappush
    ans = 0
    pq = [(0, start)]
    visited = set()
    while pq:
        front = heappop(pq)
        if front[1] not in visited:
```

```
            ans += front[0]
            visited.add(front[1])
            if len(visited) == v:
                return ans
            for d, neighbor in graph[front[1]]:
                if neighbor not in visited:
                    heappush(pq, (d, neighbor))
    return -1
```

Kruskal

时间复杂度： $O(ElogE)$，适用于稀疏图

```
Dsu = DisjointSet(n)  # 枚举最短的边，若当前边两顶点加入已访问过的点集后不会形成回路，则加入该边。遍历所有边。
edges.sort()
cnt = 0
for e in edges:
    w, v, u = e
    if Dsu.find(v) != Dsu.find(u):
        Dsu.union(v, u)
        cnt += w
```

其他DP实现

归并排序-求逆序对数

核心：数组逆序对数 = 左子串逆序对数+右子串逆序对数 + 左子串元素相对右子串逆序对数。**分治**过程完全同归并排序。、

```
def merge_sort(arr):
    if len(arr) < 2:
        return arr, 0
    mid = len(arr) // 2
    left, l_cnt = merge_sort(arr[:mid])
    right, r_cnt = merge_sort(arr[mid:])
    merged, merged_cnt = merge(left, right)
    return merged, l_cnt + r_cnt + merged_cnt


def merge(left, right):
    temp = []
    cnt = 0
    while left and right:
        if left[0] <= right[0]:
            temp.append((left.pop(0)))
        else:
            temp.append(right.pop(0))
            cnt += len(left)
    temp += left if left else right
    return temp, cnt
```

另还有朴素算法，时间复杂度也为$O(nlogn)$：

```
from bisect import *
a=[]
rev=0
for _ in range(n):
    num=int(input())
    rev+=bisect_left(a,num)
    insort_left(a,num)
ans=n*(n-1)//2-rev
```

最大全0子矩阵

DP。给出主矩阵规模和障碍点坐标，求出最大的不含障碍的点子矩阵的面积。

思路：将障碍点从左到右排序，按照障碍点枚举左边界情况；对于一个障碍点确定的左边界，枚举右边界，更新上下边界直到主矩阵右边界。 再从右到左扫描一遍，最后更新左右边界为主矩阵左右边界情况。**时间复杂度**：$O(S^2)$，S为点数目

```
# 从左向右找，直到右边界
for i in range(v):
    high, low = 0, n
    for j in range(i + 1, v):
        ans = max(ans, (low - high) * (vertex[j][1] - vertex[i][1]))
        if vertex[j][0] > vertex[i][0]:
            low = min(low, vertex[j][0])
        elif vertex[j][0] < vertex[i][0]:
            high = max(high, vertex[j][0])
        else:
            low = high = vertex[i][0]
            break
    ans = max(ans, (low - high) * (m - vertex[i][1]))
# 从右向左找，直到左边界
for i in range(v - 1, -1, -1):
    high, low = 0, n
    for j in range(i - 1, -1, -1):
    ...
# 以主矩阵左右边界为左右边界
```

ST表-区间最值

时间、空间复杂度：$O(nlogn)$

创建ST表：ST表主体为二维数组$st[i][j]$，表示从原数组中第 i 项到第 $i+2^j$项 的区间最值，区间左闭右开。**状态转移方程**：

$$st[i][j] = max\{st[i][j-1],\ st[i+2^{j-1}][2^{j-1}]\}$$

初值：$st[i][0] = S[i]$

```
min_st = [[arr[x]] for x in range(len(arr))]
max_st = [[arr[x]] for x in range(len(arr))]
for j in range(1, int(math.log(len(arr), 2)) + 1):
    for i in range(len(arr) - (1 << j)):
        min_st[i].append(min(min_st[i][j - 1], min_st[i + (1 << j - 1)][j - 1]))
        max_st[i].append(max(max_st[i][j - 1], max_st[i + (1 << j - 1)][j - 1]))
```

```
left, right = map(int, input().split())  # 查询：此处查询区间左闭右开
x = int(math.log(right - left, 2))
print(min(min_st[left][x], min_st[left + (1 << x)][x]))
print(max(max_st[left][x], max_st[left + (1 << x)][x]))
```

最长公共子序列

```
for i in range(len(str_1)):  # dp(i,j) 表示 串1前i项、串2前j项能找到的最长公共子序列长度
    for j in range(len(str_2)):
        if str_1[i] == str_2[j]:
            dp[i + 1][j + 1] = dp[i][j] + 1
        else:
            dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j])
```

最长上升子序列*, O(nlogn)

```
for i in range(1, len(sequence)):  # d[i] 表示的是目前长度为i的最优子序列的末尾元素，随遍历更
    if sequence[i] > d[num - 1]:  # 符合条件，更新最长子序列
        num += 1
        d.append(sequence[i])
    else:  # 不符合条件，更新前面的子序列，使后续插入的条件更宽松
        left = 0
        right = num - 1
        while left < right:  # 二分查找
            mid = left + (right - left >> 1)
            if d[mid] > sequence[i]:
                right = mid
            else:
                left = mid + 1
        d[left] = min(d[left], sequence[i])
```

01背包

```
for i in range(n):  # ans[j]表示背包重量为j时能获得的最大价值
    for j in range(t - time[i], -1, -1):  # 从ans[t]更新到ans[time[i]]
        ans[j + time[i]] = max(ans[j + time[i]], ans[j] + value[i])
```

工具

int(str,n) 将字符串 `str` 转换为 `n` 进制的整数。

list(zip(a,b)) 将两个列表元素一一配对，生成元组的列表。

math.pow(m,n) 计算 `m` 的 `n` 次幂。

math.log(m,n) 计算以 `n` 为底的 `m` 的对数。

字符串

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

```
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})
```

product：笛卡尔积

```
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')
```