CheatSheet

数据结构

1. 链表

```
class LinkNode():
    def __init__(self, node_data):
        self.data = node_data
        self.next : LinkNode = None
class LinkList():
    def __init__(slef):
        self.head : LinkNode = None
        self.length = 0
    def insert(self, index: int, value):
        if index < 0 or index > self.length:
            raise IndexError(index)
        node = LinkNode(value)
        cur_node, position = self.head, 0
        while position < index:
            cur_node = cur.node.nest
            position += 1
        node.nest = cur_node.next
        cur_node.next = node
        self.length += 1
```

2. KMP算法

```
# 计算Next数组，复杂度O(m)
def make_next(pattern):
    Next = [None] * len(pattern)
    i, k = 0, -1
    Next[0] = -1
    while i < len(pattern) - 1:
        if i == 0:
            Next[i] = -1
        while k >= 0 and pattern[i] != pattern[k]:
            k = Next[k]
        k += 1
        i += 1
        Next[i] = k
    return Next
# KMP 复杂度O(n)
def match(target, pattern, Next):
    i, j = 0, 0
    while i < len(pattern) and j < len(target):
        if pattern[i] == target[j] or i == -1:
            i += 1
            j += 1
        else:
            i = Next[i]
    if i == len(pattern):
        return j - len(pattern)
    return -1
# 总复杂度O(m+n)
#改进next数组
def improved_make_next(pattern):
    Next = [None] * len(pattern)
    i, k = 0, -1
    while i < len(pattern) - 1:
        if i == 0:
            Next[i] = -1
        while k >= 0 and pattern[i] != pattern[k]:
            k = Next[k]
        k += 1
        i += 1

        if pattern[i] == pattern[k]:
            Next[i] = Next[k]
        else:
            Next[i] = k
    return Next
```

3.二叉树

```
# 顺序表二叉树定义
class BinaryTree:
    def __init__(self, maxnodes = 100):
        self.nodes = [None] * maxnodes
        self.size = 0

    def insert(self, value):
        if self.size < len(self.nodes):
            self.nodes[self.size] = value
            self.size += 1

    def parent(self, index):
        return self.nodes[(index - 1) // 2] if index > 0 else None

    def left_child(self, index):
        left_index = 2 * index + 1
        return self.nodes[left_index] if left_index <self.size else None

    def right_child(self, index):
        right_index = 2 * index + 2
        return self.nodes[right_index] if right_index < self.size else None
```

```
# 链式存储
class BinaryTree:
    def __init__(self, rootobj):
        self.key = rootobj
        self.left_child = None
        self.right_child = None

    def insert_left(self, new_node):
        if self.left_child == None:
            self.left_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.left_child = self.left_child
            self.left_child = t

    def insert_right(self, new_node):
        if self.right_child == None:
            self.right_child = BinaryTree(new_node)
```

```python
        else:
            t = BinaryTree(new_node)
            t.right_child = self.right_child
            self.right_child = t

    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_val(self, obj):
        self.key = obj

    def get_root_val(self):
        return self.key
```

```python
# 根据中序、后序遍历输出前序遍历
mid = list(map(int, input().split()))
last = list(map(int, input().split()))

def build_tree(mid,last):
    if mid == []:
        return []

    root_value = last.pop()
    root_index = mid.index(root_value)

    right_tree = build_tree(mid[root_index + 1:], last)
    left_tree = build_tree(mid[:root_index], last)

    return [root_value] + left_tree + right_tree

ans = build_tree(mid, last)

print(' '.join([str(e) for e in ans]))
```

```python
# k叉书，根据中序、后序遍历判定k叉树可能的个数
from math import comb
def count_possible_trees(pre, post):
    if not pre:
        return 1
    root = pre[0]
    if len(pre) == 1:
        return 1

    root_post_index = post.index(root)

    pre_subtrees = pre[1:]
    post_subtrees = post[:root_post_index]

    count = 1
    subtree = []
    while pre_subtrees:
        root_subtree = pre_subtrees[0]
        root_subtree_post_index = post_subtrees.index(root_subtree)

        subtree_size = root_subtree_post_index + 1
        subtree.append(count_possible_trees(pre_subtrees[:subtree_size], post_subtrees[:subtree_size]))

        pre_subtrees = pre_subtrees[subtree_size:]
        post_subtrees = post_subtrees[subtree_size:]

    k = len(subtree)
    for i in subtree:
        count *= i
    count *= comb(int(m), k)
    return count

while True:
    s = input()
    if s == '0':
        break
    m, pre, post = s.split()
    print(count_possible_trees(pre, post))
```

4. 哈夫曼树

```
class HuffmanNode:
    def __init__(self, weight, char = None, parent = None):
        self.weight = weight
        self.char = char
        self.parent = parent
        self.left = None
        self.right = None
    def build_huffman_tree(weights, chars):
        nodes = [HuffmanNode(weights[i], chars[i]) for i in range(len(weights))]
        while len(nodes) > 1:
            nodes.sort(key = lambda x: x.weight)
            left = nodes.pop(0)
            right = nodes.pop(0)
            new_node = HuffmanNode(left.weight + right.weight)
            new_node.left = left
            new_node.right = right
            left.parent = new_node
            right.parent = new_node
            nodes.append(new_node)
        return nodes[0]
```

5. 二叉搜索树

```
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()

    def put(self, key, value):
        if self.root:
            self._put(key, value, self.root)
        else:
            self.root = TreeNode(key, value)
        self.size += 1

    def _put(self, key, value, current_node):
        if key < current_node.key:
            if current_node.has_left_child():
                self._put(key, value, current_node.left_child)
            else:
                current_node.left_child = TreeNode(key, value, parent=current_node)
        else:
            if current_node.has_right_child():
                self._put(key, value, current_node.right_child)
            else:
                current_node.right_child = TreeNode(key, value, parent=current_node)

    def __setitem__(self, key, value):
        self.put(key, value)

    def get(self, key):
        if self.root:
            result = self._get(key, self.root)
            if result:
                return result.value
        return None

    def _get(self, key, current_node):
        if not current_node:
            return None
        elif key == current_node.key:
            return current_node
        elif key < current_node.key:
            return self._get(key, current_node.left_child)
        else:
            return self._get(key, current_node.right_child)

    def __getitem__(self, key):
        return self.get(key)

    def __contains__(self, key):
        return bool(self._get(key, self.root))

    def __delitem__(self, key):
        self.delete(key)

    def delete(self, key):
        if self.size > 1:
            node_to_delete = self._get(key, self.root)
            if node_to_delete:
                self._delete(node_to_delete)
                self.size -= 1
            else:
                raise KeyError('Error, key not in tree')
        elif self.size == 1 and self.root.key == key:
            self.root = None
            self.size -= 1
        else:
            raise KeyError('Error, key not in tree')

    def _delete(self, current_node):
        if current_node.is_leaf():
            if current_node.is_left_child():
                current_node.parent.left_child = None
            else:
                current_node.parent.right_child = None
        elif current_node.has_both_children():
            successor = current_node.right_child.find_min()
            successor.splice_out()
            current_node.key = successor.key
            current_node.value = successor.value
        else:
            if current_node.has_left_child():
                if current_node.is_left_child():
                    current_node.left_child.parent = current_node.parent
                    current_node.parent.left_child = current_node.left_child
                elif current_node.is_right_child():
                    current_node.left_child.parent = current_node.parent
                    current_node.parent.right_child = current_node.left_child
                else:
                    current_node.replace_node_value(current_node.left_child.key,
                                                    current_node.left_child.value,
                                                    current_node.left_child.left_child,
                                                    current_node.left_child.right_child)
            else:
                if current_node.is_left_child():
                    current_node.right_child.parent = current_node.parent
                    current_node.parent.left_child = current_node.right_child
                elif current_node.is_right_child():
                    current_node.right_child.parent = current_node.parent
                    current_node.parent.right_child = current_node.right_child
                else:
                    current_node.replace_node_value(current_node.right_child.key,
                                                    current_node.right_child.value,
                                                    current_node.right_child.left_child,

 current_node.right_child.right_child)


class TreeNode:
    def __init__(self, key, value, left=None, right=None, parent=None):
        self.key = key
        self.value = value
        self.left_child = left
        self.right_child = right
        self.parent = parent

    def has_left_child(self):
        return self.left_child

    def has_right_child(self):
        return self.right_child

    def is_left_child(self):
        return self.parent and self.parent.left_child == self

    def is_right_child(self):
        return self.parent and self.parent.right_child == self

    def is_root(self):
        return not self.parent
    def is_leaf(self):
        return not (self.left_child or self.right_child)

    def has_any_children(self):
        return self.left_child or self.right_child

    def has_both_children(self):
        return self.left_child and self.right_child

    def replace_node_value(self, key, value, left, right):
        self.key = key
        self.value = value
        self.left_child = left
        self.right_child = right
        if self.has_left_child():
            self.left_child.parent = self
        if self.has_right_child():
            self.right_child.parent = self
```

6. 并查集

```
from heapq import heappop, heappush

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

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

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)
        if x_root == y_root:
            return
        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
        else:
            self.parent[y_root] = x_root
            if self.rank[x_root] == self.rank[y_root]:
                self.rank[x_root] += 1
dsu = DSU(26)
```

7. 排序算法

(1) Shell 排序

```
def shell_sort(alist):
    sublist_count = len(alist) // 2
    while sublist_count > 0:
        for start_position in range(sublist_count):
            gap_insertion_sort(alist, start_position, sublist_count)
        print("After increments of size", sublist_count, "The list is", alist)
        sublist_count = sublist_count // 2
```

```
def gap_insertion_sort(alist, start, gap):
    for i in range(start + gap, len(alist), gap):  # 从start+gap开始，以gap为增量遍历
        current_value = alist[i]  # 当前待插入元素
        position = i  # 当前位置
        while position >= gap and alist[position - gap] > current_value:
            alist[position] = alist[position - gap]  # 元素后移
            position = position - gap  # 向前比较
        alist[position] = current_value  # 插入元素
```

(2) 堆排序

```
def heap_sort(alist):
    def sift_down(start, end):
        root = start  # 当前子树的根节点
        while True:
            child = 2 * root + 1  # 左子节点位置
            if child > end:  # 如果没有子节点
                break
            if child + 1 <= end and alist[child] < alist[child + 1]:
                child += 1  # 如果有右子节点且右子节点更大，选择较大的子节点
            if alist[root] < alist[child]:
                alist[root], alist[child] = alist[child], alist[root]
                root = child  # 继续向下检查
            else:
                break  # 堆性质已满足，退出

    for start in range((len(alist) - 2) // 2, -1, -1):  # 从最后一个非叶子节点开始
        sift_down(start, len(alist) - 1)
    for end in range(len(alist) - 1, 0, -1):
        alist[0], alist[end] = alist[end], alist[0]  # 将最大值（堆顶）移到末尾
        sift_down(0, end - 1)  # 对剩余元素重新构建最大堆
```

(3) 冒泡排序

```
def bubble_sort(alist):
    for i in range(len(alist) - 1, 0, -1):  # 外层循环控制排序轮数，i表示每轮比较的最后一个元素位置
        no_swap = True
        for j in range(i):  # 内层循环进行相邻元素比较
            if alist[j] > alist[j + 1]:
                alist[j], alist[j + 1] = alist[j + 1], alist[j]
                no_swap = False
        if no_swap:
            break  # 如果没有发生交换，说明数组已有序，提前结束
```

(4) 快速排序

```python
def partition(start, end):
    pivot = alist[start]
    low = start
    high = end
    while low < high:
        while low < high and alist[high] >= pivot:
            high -= 1
        alist[low] = alist[high]
        while low < high and alist[low] <= pivot:
            low += 1
        alist[high] = alist[low]
    alist[low] = pivot
    return low
```

(5) 二路归并排序

```python
def merge_sort(alist):
    def merge(left, right):  # 1. 定义合并两个有序列表的函数
        result = []  # 存储合并结果
        i = j = 0  # 初始化左右子列表的序号
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result += left[i:]  # 将剩余元素直接追加到result末尾
        result += right[j:]
        return result

    # 2. 递归终止条件：子列表长度为0或1时直接返回
    if len(alist) <= 1:
        return alist
    mid = len(alist) // 2
    left = merge_sort(alist[:mid])  # 递归排序左半部分
    right = merge_sort(alist[mid:])  # 递归排序右半部分
    return merge(left, right)
```

```python
# 求逆序数
def mergeSort(arr):
    if len(arr) <= 1:
        return arr, 0

    mid = len(arr) // 2
    left, inv_count_left = mergeSort(arr[:mid])
    right, inv_count_right = mergeSort(arr[mid:])

    merged, inv_count = merge(left, right)
    inv_count += inv_count_left + inv_count_right

    return merged, inv_count

def merge(left, right):
    merged = []
    inv_count = 0
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            inv_count += len(left) - i

    merged += left[i:]
    merged += right[j:]

    return merged, inv_count
while True:

    n = int(input())
    if n == 0:
        break
    arr = list(map(int, input().split()))

    sorted_arr, inv_count = mergeSort(arr)
    print(inv_count)
```

8. 最小生成树

```
# Kruskal 算法
from heapq import heappop, heappush
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.count = n

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

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)
        if x_root == y_root:
            return
        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
        else:
            self.parent[y_root] = x_root
            if self.rank[x_root] == self.rank[y_root]:
                self.rank[x_root] += 1
        self.count -= 1

N = int(input())
location = []
for i in range(N):
    x, y = map(int, input().split())
    location.append((x, y))
path = []
```

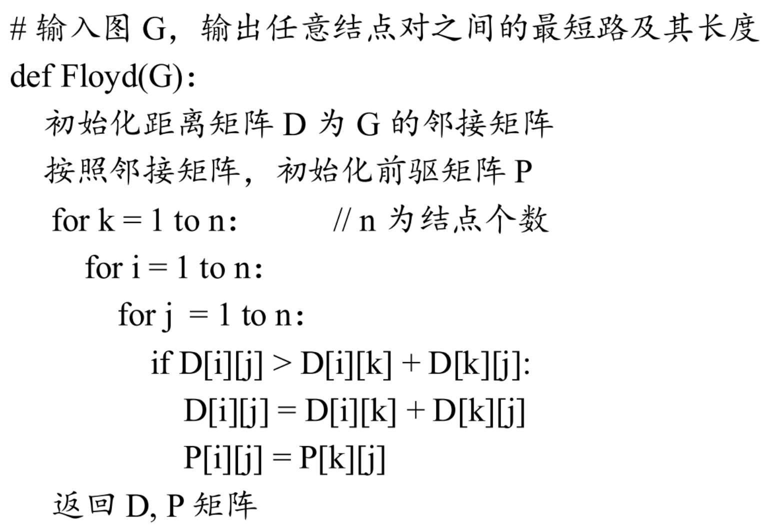
```
for i in range(N):
    for j in range(i + 1, N):
        lenth = (location[i][0] - location[j][0]) ** 2 + (location[i][1] - location[j]
[1]) ** 2
        heappush(path, (lenth, i, j))
dsu = DSU(N)

M = int(input())
for i in range(M):
    a, b= map(int, input().split())
    dsu.union(a - 1, b - 1)
root = 0
ans= []
n = 0
while path:
    #n += 1
    if dsu.count == 1:
        break
    cur_path = heappop(path)
    if dsu.find(cur_path[1]) == dsu.find(root) and dsu.find(cur_path[2]) ==
dsu.find(root) or dsu.find(cur_path[1]) == dsu.find(cur_path[2]):
        continue
    ans.append((cur_path[1] + 1, cur_path[2] + 1))
    dsu.union(cur_path[1], cur_path[2])
for i in ans:
    print(' '.join(map(str, i)))
#print(n)
# Prim
from collections import defaultdict
import heapq
# 构建图
n = int(input())
graph = defaultdict(list)
for _ in range(n - 1):
    line = input().split()
    start = line[0]
    k = int(line[1])
    for i in range(k):
        end = line[2 + 2 * i]
        side = int(line[3 + 2 * i])
        graph[start].append((end, side))
        graph[end].append((start, side))

visited = set()
ans = 0
q = []
start = 'A'  # 不妨从A开始
visited.add(start)
for nb, side in graph[start]:
    heapq.heappush(q, (side, nb))
while q and len(visited) < n:
    side, star = heapq.heappop(q)
    if star not in visited:
        visited.add(star)
        ans += side
        for v, w in graph[star]:
            if v not in visited:
                heapq.heappush(q, (w, v))

print(ans)
```

9. 最短路径问题

(1) dijkstra

```python
from heapq import heappop, heappush
P = int(input())
names = dict()
names1 = dict()
for i in range(P):
    name = input()
    names[name] = i
    names1[i] = name
graph = [[-1] * P for _ in range(P)]
Q = int(input())
for i in range(Q):
    a, b, length = input().split()
    graph[names[a]][names[b]] = int(length)
    graph[names[b]][names[a]] = int(length)
def dijkstra(start, end, graph):
    pq = []
    heappush(pq, (0, start, [start]))
    while pq:
        l, cur, path = heappop(pq)
        if cur == end:
            return path
        for i in range(P):
            if graph[cur][i] != -1 and i not in path:
                cur_path = path.copy()
                cur_path.append(i)
                heappush(pq, (l + graph[cur][i], i, cur_path))
    return
R = int(input())
for i in range(R):
    a, b = input().split()
    start = names[a]
    end = names[b]
    path = dijkstra(start, end, graph)
    ans = []
    for i in range(len(path) - 1):
        ans.append(names1[path[i]])
        ans.append('->')
        l = graph[path[i]][path[i + 1]]
        ans.append('(' + str(l) + ')')
        ans.append('->')
    ans.append(names1[path[-1]])
    print(''.join(ans))
```

(2) Floyd



```
# 输入图 G，输出任意结点对之间的最短路及其长度
def Floyd(G):
   初始化距离矩阵 D 为 G 的邻接矩阵
   按照邻接矩阵，初始化前驱矩阵 P
   for k = 1 to n:          // n 为结点个数
      for i = 1 to n:
         for j  = 1 to n:
            if D[i][j] > D[i][k] + D[k][j]:
               D[i][j] = D[i][k] + D[k][j]
               P[i][j] = P[k][j]
   返回 D, P 矩阵
```

```python
import sys

def main():
    lines = [line.strip() for line in sys.stdin if line.strip()]
    idx = 0
    case = 1
    while idx < len(lines):
        n = int(lines[idx])
        idx += 1
        if n == 0:
            break
        currencies = []
        for _ in range(n):
            currencies.append(lines[idx])
            idx += 1
        name2idx = {name: i for i, name in enumerate(currencies)}
        m = int(lines[idx])
        idx += 1
        dist = [[0.0] * n for _ in range(n)]
        for i in range(n):
            dist[i][i] = 1.0
        for _ in range(m):
            parts = lines[idx].split()
            idx += 1
            src = parts[0]
            rate = float(parts[1])
            dest = parts[2]
            src_idx = name2idx[src]
            dest_idx = name2idx[dest]
            if rate > dist[src_idx][dest_idx]:
                dist[src_idx][dest_idx] = rate
        # Floyd-Warshall relaxation
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    if dist[i][k] * dist[k][j] > dist[i][j] + 1e-8:
                        dist[i][j] = dist[i][k] * dist[k][j]
        # Check for arbitrage
        arbitrage = False
        for i in range(n):
            if dist[i][i] > 1.0 + 1e-8:
                arbitrage = True
                break
        print(f"Case {case}: {'Yes' if arbitrage else 'No'}")
        case += 1

if __name__ == "__main__":
    main()
```

10. 拓扑排序

```
from collections import defaultdict
v, a = map(int, input().split())
nodes = defaultdict(int)
path = []
for i in range(a + 1):
    path.append([])
for i in range(a):
    x1, x2 = map(int, input().split())
    nodes[x2] += 1
    path[x1].append(x2)
ans = []
n = 0
while n <= v - 1:
    for i in range(v):
        if nodes[i + 1] == 0 and i + 1 not in ans:
            n += 1
            ans.append(i + 1)
            for j in path[i + 1]:
                nodes[j] -= 1
            break
answer = []
for i in ans:
    answer.append('v' + str(i))
print(' '.join(answer))
# 检测回路中的环
from collections import deque
def dfs(start, node, ans, visited):
    if nums[node] == start:
        return ans + 1
    visited.add(nums[node])
    return dfs(start, nums[node], ans + 1, visited)
def F(nums):
    delete = set()
    n = len(nums)
    nodes = [0] * n
    for num in nums:
        if num != -1:
            nodes[num] += 1
    queue = deque()
    for i in range(n):
        if nodes[i] == 0:
            queue.append(i)
    while queue:
        node = queue.popleft()
        delete.add(node)
        nodes[nums[node]] -= 1
        if nodes[nums[node]] == 0:
            queue.append(nums[node])
    cur_nums = []
    for i in range(n):
        if i not in delete:
            cur_nums.append(i)
    if len(cur_nums) == 0:
        return 0, 0
    visited = set()
    count, max_len = 0, 0
    for start in cur_nums:
        if start not in visited:
            count += 1
            visited.add(start)
            max_len = max(max_len, dfs(start, start, 0, visited))
    return count, max_len

N = int(input())
for i in range(N):
    nums = list(map(int, input().split()))
    count, max_len = F(nums)
    print(count, max_len)
```

11. Tarjan算法

```
class TarjanSCC:
    def __init__(self, graph):
        self.graph = graph
        self.n = len(graph)
        self.index = 0
        self.stack = []
        self.on_stack = [False] * self.n
        self.indices = [-1] * self.n
        self.lowlink = [-1] * self.n
        self.sccs = []

    def run(self):
        for v in range(self.n):
            if self.indices[v] == -1:
                self.strongconnect(v)
        return self.sccs

    def strongconnect(self, v):
        self.indices[v] = self.index
        self.lowlink[v] = self.index
        self.index += 1
        self.stack.append(v)
        self.on_stack[v] = True

        for w in self.graph[v]:
            if self.indices[w] == -1:
                self.strongconnect(w)
                self.lowlink[v] = min(self.lowlink[v], self.lowlink[w])
            elif self.on_stack[w]:
                self.lowlink[v] = min(self.lowlink[v], self.indices[w])

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

# 示例用法
if __name__ == "__main__":
    # 图用邻接表表示，例如：0->1, 1->2, 2->0, 1->3
    graph = [
        [1],    # 0
        [2, 3], # 1
        [0],    # 2
        []      # 3
    ]
    tarjan = TarjanSCC(graph)
    sccs = tarjan.run()
    print("强连通分量:", sccs)
```

12. Bellman-Ford 算法

```
def bellman_ford(graph, V, src):
    # graph: list of edges (u, v, w)
    # V: number of vertices
    # src: source vertex
    dist = [float('inf')] * V
    dist[src] = 0
    for _ in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    # 检查负权环
    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("图中存在负权环")
            return None

    return dist
# 示例用法
if __name__ == "__main__":
    # 顶点数
    V = 5
    # 边列表 (u, v, w)
    graph = [
        (0, 1, -1),
        (0, 2, 4),
        (1, 2, 3),
        (1, 3, 2),
        (1, 4, 2),
        (3, 2, 5),
        (3, 1, 1),
        (4, 3, -3)
    ]
    src = 0
    dist = bellman_ford(graph, V, src)
    if dist:
        print("从顶点 {} 到各顶点的最短距离:".format(src))
        for i in range(V):
            print("顶点 {} : {}".format(i, dist[i]))
```

一些算法模板

递归算法以及eval()函数

```
s = input().split()
def cal():
    cur = s.pop(0)
    if cur in "+-*/":
        return str(eval(cal() + cur + cal()))
    else:
        return cur
print("%.6f" % float(cal()))
```

辅助栈的维护

```
import sys

# 使用两个栈来实现
stack = []
min_stack = []

# 读取所有输入并按行处理
input = sys.stdin.read
data = input().strip().splitlines()

for s in data:
    if s == 'pop':
        if stack:
            stack.pop()
            min_stack.pop()
    elif s == 'min':
        if stack:
            print(min_stack[-1])
    else:
        # 假设输入格式为 "push n"
        command, value = s.split()
        if command == 'push':
            weight = int(value)
            stack.append(weight)
            if not min_stack or weight <= min_stack[-1]:
                min_stack.append(weight)
            else:
                min_stack.append(min_stack[-1])
```

区间问题

```
'''
#pypy能过,python超时
import sys
input = sys.stdin.readline

def add_interval(start, end, l, r):
    left, right = 0, len(start)
    while left < right:
        mid = (left + right) // 2
        if start[mid] > l:
            right = mid
        else:
            left = mid + 1
    start.insert(left, l)
    left, right = 0, len(end)
    while left < right:
        mid = (left + right) // 2
        if end[mid] > r:
            right = mid
        else:
            left = mid + 1
    end.insert(left, r)

def remove_interval(start, end, l, r):
    left, right = 0, len(start)
    while left < right:
        mid = (left + right) // 2
        if start[mid] == l:
            del start[mid]
            break
        elif start[mid] > l:
            right = mid
        else:
            left = mid + 1
    left, right = 0, len(end)
    while left < right:
        mid = (left + right) // 2
        if end[mid] == r:
            del end[mid]
            return
        elif end[mid] > r:
            right = mid
        else:
            left = mid + 1

def check(start, end):
    if len(start) > 1 and start[-1] > end[0]:
        return True
    return False

n = int(input())
start = []
end = []
for i in range(n):
    operater, l, r = map(str, input().strip().split())
    l , r = int(l), int(r)
    if operater == '+':
        add_interval(start, end, l, r)
    if operater == '-':
        remove_interval(start, end, l, r)
    #print(start)
    #print(end)
    if check(start, end):
        print('YES')
    else:
        print('NO')
'''

import sys
import heapq
from collections import defaultdict
```

```
input = sys.stdin.readline

minH = []
maxH = []

ldict = defaultdict(int)
rdict = defaultdict(int)

n = int(input())

for _ in range(n):
    op, l, r = map(str, input().strip().split())
    l, r = int(l), int(r)

    if op == "+":
        ldict[l] += 1
        rdict[r] += 1
        heapq.heappush(maxH, -l)
        heapq.heappush(minH, r)
    else:
        ldict[l] -= 1
        rdict[r] -= 1

    # 使用 while 循环，将最大堆 maxH 和最小堆 minH 中出现次数为 0 的边界移除。
    # 通过比较堆顶元素的出现次数，如果出现次数为 0，则通过 heappop 方法将其从堆中移除。
    while len(maxH) > 0 >= ldict[-maxH[0]]:
        heapq.heappop(maxH)
    while len(minH) > 0 >= rdict[minH[0]]:
        heapq.heappop(minH)

    # 判断堆 maxH 和 minH 是否非空，并且最小堆 minH 的堆顶元素是否小于
    # 最大堆 maxH 的堆顶元素的相反数。
    if len(maxH) > 0 and len(minH) > 0 and minH[0] < -maxH[0]:
        print("Yes")
    else:
        print("No")
```

整数划分问题dp算法

```
dp = [0] * (50 + 1)
dp[0] = 1  # 基础情况：有一种方式来分解 0
for i in range(1, 50 + 1):
    for j in range(i, 50 + 1):
        dp[j] += dp[j - i]
print(dp[-1])

# 以2的幂次为基底
N = int(input())
MOD = 10**9  # 只保留最后 9 位数字
# 初始化 dp 数组，dp[i] 表示将整数 i 进行划分的不同方案数量
dp = [0] * (N + 1)
dp[0] = 1  # 基础情况：有一种方式来分解 0
# 遍历每个 2 的幂次方项 2^j
j = 0
while (power := 2 ** j) <= N:
    for i in range(power, N + 1):
        dp[i] = (dp[i] + dp[i - power]) % MOD  # 对 10^9 取模，确保只保留最后 9 位
    j += 1
# 输出结果的最后 9 位数字
```

```
print(str(dp[N])[-9:])
```

求排列的逆序数（二分归并）

```
def mergeSort(arr):
    if len(arr) <= 1:
        return arr, 0
    mid = len(arr) // 2
    left, inv_count_left = mergeSort(arr[:mid])
    right, inv_count_right = mergeSort(arr[mid:])
    merged, inv_count = merge(left, right)
    inv_count += inv_count_left + inv_count_right
    return merged, inv_count
def merge(left, right):
    merged = []
    inv_count = 0
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            inv_count += len(left) - i
    merged += left[i:]
    merged += right[j:]
    return merged, inv_count
# 输入排列
n = int(input())
arr = list(map(int, input().split()))
# 调用归并排序函数并输出逆序数
sorted_arr, inv_count = mergeSort(arr)
print(inv_count)
```

滑动窗口解决最长子列

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        #滑动窗口
        char_map = {}
        left, max_length = 0, 0
        for right in range(len(s)):
            if s[right] in char_map and char_map[s[right]] >= left:
                left = char_map[s[right]] + 1
                char_map[s[right]] = right
            else:
                char_map[s[right]] = right
                max_length = max(max_length, right - left + 1)
        return max_length
    ans = lengthOfLongestSubstring
    print(ans)
```

小偷背包类型动态规划

```
n, t = map(int, input().split())
value = list(map(int, input().split()))
def discount(n, t, value):
    sum_value = sum(value)
    if sum_value < t:
        return 0
    dp = [0] * (sum_value + 1)
    for i in range(1, n + 1):
        for j in range(sum_value, 0, -1):
            if value[i - 1] <= j:
                dp[j] = max(dp[j], dp[j - value[i - 1]] + value[i - 1])
    for i in range(t, sum_value + 1):
        if dp[i] >= t:
            return dp[i]
ans = discount(n, t, value)
print(ans)
```

```
N, B = map(int,input().split())
value = list(map(int,input().split()))
weight = list(map(int,input().split()))
dp = [0] * (B + 1)
for i in range(1, N + 1):
    for j in range(B, 0, -1):
        if weight[i-1] <=  j:
            dp[j] = max(dp[j],dp[j - weight[i-1]] + value[i-1])
print(dp[B])

# 压缩矩阵/滚动数组 方法
N,B = map(int, input().split())
*p, = map(int, input().split())
*w, = map(int, input().split())

dp=[0]*(B+1)
for i in range(N):
    for j in range(B, w[i] - 1, -1): #此处可以到w[i]即可
        dp[j] = max(dp[j], dp[j-w[i]]+p[i])

print(dp[-1])

# 01背包，必须装满
t, n = map(int, input().split())
time, weight = [], []
for i in range(n):
    ti, wi = map(int, input().split())
    time.append(ti)
    weight.append(wi)
dp = [-1] * (t + 1)
dp[0] = 0
for i in range(n):
    for j in range(t, time[i] - 1, -1):
        if dp[j - time[i]] != -1:
            dp[j] = max(dp[j], dp[j - time[i]] + weight[i])
print(dp[-1])
```

完全背包

```
n, m = map(int, input().split())
l = list(map(int, input().split()))
dp = [float('inf')] * (m + 1)
dp[0] = 0
for i in range(1, m + 1):
    for j in l:
        if i - j >= 0:
            dp[i] = min(dp[i], dp[i - j] + 1)
if dp[-1] == float('inf'):
    print(-1)
else:
    print(dp[-1])
```

01背包变形且必须装满(NBA门票)

```
n=int(input())
tickets=list(map(int,input().split()))
price=[50,100,250,500,1000,2500,5000]
dp={0:0}
path={0:[0,0,0,0,0,0,0]}
for i in range(n):
    if i in dp:
        for k in range(7):
            if path[i][k]<tickets[k]:
                if i+price[k] in dp:
                    if dp[i]+1<dp[i+price[k]]:
                        dp[i+price[k]]=dp[i]+1
                        path[i+price[k]]=path[i][:]
                        path[i+price[k]][k]+=1
                else:
                    dp[i+price[k]]=dp[i]+1
                    path[i+price[k]]=path[i][:]
                    path[i+price[k]][k]+=1
if n in dp:
    print(dp[n])
else:
    print('Fail')
```

宝可梦皮卡丘(注释掉的是自己超时代码，没注释的是题解)

```
#N, M, K = map(int, input().split())
#l = [list(map(int, input().split())) for _ in range(K)]

#dp = [[0] * (M + 1) for _ in range(N + 1)]
#for i in range(1, K + 1):
#    for j in range(N, 0, -1):
#        for k in range(M, 0, -1):
#            if l[i - 1][0] <= j and l[i - 1][1] <= k:
#                dp[j][k] = max(dp[j][k], dp[j - l[i - 1][0]][k - l[i - 1][1]] + 1)
#target = dp[N][M]

#def find_target(dp, target, N, M):
#    ans = M + 1
#    for i in range(N + 1):
#        for j in range(M + 1):
#            if dp[i][j] == target:
#                ans = min(ans, j)
#                if ans == j:
#                    break
#    return ans

#remaining = M - find_target(dp, target, N, M)
#print(target, remaining)
N, M, K = map(int, input().split())
L = [[-1] * (M + 1) for i in range(K + 1)]
L[0][M] = N
for i in range(K):
    cost, dmg = map(int, input().split())
    for p in range(M):
        for q in range(i + 1, 0, -1):
            if p + dmg <= M and L[q - 1][p + dmg] != -1:
                L[q][p] = max(L[q][p], L[q - 1][p + dmg] - cost)

def find():
    for i in range(K, -1, -1):
        for j in range(M, -1, -1):
            if L[i][j] != -1:
                return [str(i), str(j)]

print(' '.join(find()))
```

双dp

```
# 1195C. Basketball Exercise
n = int(input())
team1 = list(map(int, input().split()))
team2 = list(map(int, input().split()))
dp1 = team1[:]
dp2 = team2[:]
for i in range(1, n):
    dp1[i] = max(dp1[i - 1], dp2[i - 1] + team1[i])
    dp2[i] = max(dp2[i - 1], dp1[i - 1] + team2[i])
print(max(dp1[-1], dp2[-1]))

# 26976:摆动序列
n = int(input())
num = list(map(int, input().split()))
dp1, dp2 = [1] * n, [1] * n
for i in range(1, n):
    if num[i] > num[i - 1]:
        dp1[i] = max(dp1[i - 1], dp2[i - 1] + 1)
        dp2[i] = dp2[i - 1]
    if num[i] < num[i - 1]:
        dp2[i] = max(dp2[i - 1], dp1[i - 1] + 1)
        dp1[i] = dp1[i - 1]
    if num[i] == num[i - 1]:
        dp1[i] = dp1[i - 1]
        dp2[i] = dp2[i - 1]
print(max(dp1[-1], dp2[-1]))

# 25573:红蓝玫瑰
s = input()
```

```python
n = len(s)
dp1, dp2 = [0] * n, [0] * n
if s[0] == 'R':
    dp2[0] = 1
if s[0] == 'B':
    dp1[0] = 1
for i in range(1, n):
    if s[i] == 'R':
        dp1[i] = dp1[i - 1]
        dp2[i] = min(dp1[i - 1], dp2[i - 1]) + 1
    if s[i] == 'B':
        dp1[i] = min(dp1[i - 1], dp2[i - 1]) + 1
        dp2[i] = dp2[i - 1]
ans = min(dp1[-1], dp2[-1] + 1)
print(ans)
```

最长先上升后下降序列

```python
n = int(input())
l = list(map(int, input().split()))
dp1 = [1] * n
dp2 = [1] * n
for i in range(1, n):
    for j in range(i):
        if l[i] > l[j]:
            dp1[i] = max(dp1[i], dp1[j] + 1)
l.reverse()
for i in range(1, n):
    for j in range(i):
        if l[i] > l[j]:
            dp2[i] = max(dp2[i], dp2[j] + 1)
dp2.reverse()
ans = 0
for i in range(n):
    ans = max(ans, dp1[i] + dp2[i] - 1)
print(n - ans)
```

接雨水

~~~~python
#动态规划解法
class Solution:
    def trap(self, height: List[int]) -> int:
        n = len(height)
        leftmax = [0] * n
        rightmax = [0] * n
        leftmax[0], rightmax[n - 1] = height[0], height[n - 1]
        for i in range(1, n):
            leftmax[i] = max(leftmax[i - 1], height[i])
        for i in range(n - 2, -1, -1):
            rightmax[i] = max(rightmax[i + 1], height[i])
        ans = 0
        for i in range(n):
            ans += min(leftmax[i], rightmax[i]) - height[i]
        return ans
    ans = trap
    print(ans)
### 单调栈解法
```python
~~~~

```
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = 0
        stack = list()
        n = len(height)

        for i, h in enumerate(height):
            while stack and h > height[stack[-1]]:
                top = stack.pop()
                if not stack:
                    break
                left = stack[-1]
                currWidth = i - left - 1
                currHeight = min(height[left], height[i]) - height[top]
                ans += currWidth * currHeight
            stack.append(i)

        return ans
# 单调栈
def find_max(heigh):
    stack = []
    heigh.append(0)
    max_area = 0
    for i, h in enumerate(heigh):
        while stack and heigh[stack[-1]] > h:
            height = heigh[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, width * height)
        stack.append(i)
    return max_area
```

扔一个dfs模板

```
#可移动方式
dx = [1, -1, 0, 0]
dy = [0, 0, 1, -1]

#维护，防止越界
def in_bound(x, y, n, m):
    return 0 <= x < n and 0 <= y < m

#定义dfs函数
def dfs(x, y, n, m, visited, matrix, path, value, last_path, last_value):

    #如果走到终点，且value大于last_value，更新last_value以及last_path
    if x == n - 1 and y == m - 1:
        if value > last_value or last_value == 0:
            last_value = value
            last_path = path[:]
        return last_path, last_value

    #递归调用与回溯
    for i in range(4):
        next_x = x + dx[i]
        next_y = y + dy[i]
        if in_bound(next_x, next_y, n, m) and not visited[next_x][next_y]:
            value += matrix[next_x][next_y]
            path.append([next_x, next_y])
            visited[next_x][next_y] = True
            last_path, last_value = dfs(next_x, next_y, n, m, visited, matrix, path,
value, last_path, last_value)
            value -= matrix[next_x][next_y]
            path.pop()
            visited[next_x][next_y] = False
    return last_path, last_value

#主函数
n, m = map(int,input().split())
visited = [[False] * m for _ in range(n)]
visited[0][0] = True
matrix = []
for i in range(n):
    matrix.append(list(map(int,input().split())))
#初始化
value, last_value = matrix[0][0], float('-inf')
path, last_path = [[0, 0]], []
last_path, last_value = dfs(0, 0, n, m, visited, matrix, path, value, last_path,
last_value)
for i in last_path:
    print(' '.join([str(e + 1) for e in i]))
```

bfs模板（以及python数据读入可能遇到的问题）

```
import sys
sys.setrecursionlimit(300000)
input = sys.stdin.read

from collections import deque

dx = [1, -1, 0, 0]
dy = [0, 0, 1, -1]

def in_bound(x, y, M, N):
    return 0 <= x < M and 0 <= y < N

def bfs(matrix, M, N, positions, x_c, y_c):
    visited = [[False] * N for _ in range(M)]
    queue = deque([])

    for x_i, y_i, heigh in positions:
        if x_i == x_c and y_i == y_c:
            return True
        visited[x_i][y_i] = True
        queue.append((x_i, y_i, heigh))

    while queue:
        cur_x, cur_y, heigh = queue.popleft()
        for i in range(4):
            next_x = cur_x + dx[i]
            next_y = cur_y + dy[i]

            if in_bound(next_x, next_y, M, N) and not visited[next_x][next_y] and
matrix[next_x][next_y] < heigh:
                if next_x == x_c and next_y == y_c:
                    return True
                visited[next_x][next_y] = True
                queue.append((next_x, next_y, heigh))
    return visited[x_c][y_c]

# 读取所有输入
data = input().split()
index = 0

ans = []
K = int(data[index])
index += 1

for i in range(K):
    M, N = map(int, data[index:index + 2])
    index += 2
    matrix = []
    for _ in range(M):
        matrix.append(list(map(int, data[index:index + N])))
        index += N
    x_c, y_c = map(int, data[index:index + 2])
    index += 2
    P = int(data[index])
    index += 1
    positions = []
    for j in range(P):
        x_i, y_i = map(int, data[index: index + 2])
        positions.append((x_i - 1, y_i - 1, matrix[x_i - 1][y_i - 1]))
        index += 2

    if bfs(matrix, M, N, positions, x_c - 1, y_c - 1):
        ans.append('Yes')
    else:
        ans.append('No')

for answer in ans:
    print(answer)
```

dijkstra 走山路

```
from heapq import heappop, heappush

dx = [1, -1, 0, 0]
dy = [0, 0, 1, -1]

def in_bound(x, y, m ,n):
    return 0 <= x < m and 0 <= y < n

def dijkstra(m, n, start, end, matrix):
    if matrix[start[0]][start[1]] == '#' or matrix[end[0]][end[1]] == '#':
        return 'NO'
    pq = []
    distence = [[float('inf')] * n for _ in range(m)]
    distence[start[0]][start[1]] = 0
    heappush(pq, (0, start[0], start[1]))

    while pq:
        value, cur_x, cur_y = heappop(pq)

        if (cur_x, cur_y) == end:
            return value
        for i in range(4):
            nx = cur_x + dx[i]
            ny = cur_y + dy[i]

            if in_bound(nx, ny, m, n) and matrix[nx][ny] != '#':

                cur_value = value + abs(int(matrix[nx][ny]) - int(matrix[cur_x][cur_y]))

                if distence[nx][ny] > cur_value:
                    heappush(pq, (cur_value, nx, ny))
                    distence[nx][ny] = cur_value
    return 'NO'

m, n, p = map(int, input().split())
matrix = [list(input().split()) for _ in range(m)]
for i in range(p):
    x1, y1, x2, y2 = map(int, input().split())
    ans = dijkstra(m, n, (x1, y1), (x2, y2), matrix)
    print(ans)
```

二维矩阵二分查找

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        n = len(matrix)
        m = len(matrix[0])
        #初始化指针
        i, j = 0, n - 1

        while i <= j:
            mid_start = (i + j) // 2
            if matrix[mid_start][0] <= target <= matrix[mid_start][-1]:
                break
            elif matrix[mid_start][0] < target:
                i = mid_start + 1
            else:
                j = mid_start - 1

        #不存在
        if i > j:
            return False

        l = matrix[mid_start]
        start, end = 0, m - 1

        while start <= end:
            mid = (start + end) // 2
            if l[mid] == target:
                return True
            elif l[mid] < target:
                start = mid + 1
            else:
                end = mid - 1
        if l[mid] == target:
            return True
        else:
            return False
    if searchMatrix:
        print('true')
    else:
        print('false')
```

二分查找（找列表中一个数开始与结束位置）

```
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        n = len(nums)
        if n == 0:
            return [-1, -1]

        #查找左边界（第一个等于target的数）
        def binary_search_left(nums, target):
            left, right = 0, n - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] < target:
                    left = mid + 1
                else:
                    right = mid - 1
            return left

        #查找右边界（第一个大于target的数）
        def binary_search_right(nums, target):
            left, right = 0, n - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] <= target:
                    left = mid + 1
                else:
                    right = mid - 1
            return right

        start = binary_search_left(nums, target)
        end = binary_search_right(nums, target)

        #判断
        if start <= end and start < n and nums[start] == target:
            return [start, end]
        else:
            return [-1, -1]
    print(searchRange)

# 逆向思维的二分问题
def count(cost, m, max_cost):
    cur_sum = 0
    x = 1
    for spend in cost:
        if cur_sum + spend <= max_cost:
            cur_sum += spend
        else:
            x += 1
            cur_sum = spend
            if x > m:
                return False
    return True
def find(cost, m):
    l, r = max(cost), sum(cost)
    while l < r:
        mid = (l + r) // 2
        if count(cost, m, mid):
            r = mid
        else:
            l = mid + 1
    return l
n, m = map(int, input().split())
cost = [int(input()) for _ in range(n)]
ans = find(cost, m)
print(ans)
```

八皇后问题（涵盖生成下一个排列）

```
# 暴力枚举
import math
def make_list(num):
    i = len(num) - 2
    while i >= 0:
        if num[i] > num[i + 1]:
            i -= 1
        else:
            break
    j = len(num) - 1
    while j >= 0:
        if num[j] < num[i]:
            j -= 1
        else:
            break
    num[i], num[j] = num[j], num[i]
    num[i + 1:] = reversed(num[i + 1:])
    return num
def check(num):
    for i in range(1, 9):
        for j in range(i + 1, 9):
            if abs(num[i - 1] - num[j - 1]) == abs(i - j):
                return False
    else:
        return True
num = [i for i in range(1, 9)]
all = ['12345678']
ans = []
k = math.factorial(8) - 1
for i in range(k):
    num = make_list(num)
    if check(num[:]):
        ans.append(''.join([str(e) for e in num[:]]))
ans.sort()
n = int(input())
answers = []
for i in range(n):
    answers.append(ans[int(input()) - 1])
for answer in answers:
    print(answer)

# 搜索回溯
def A(row, n, position, check, count, o):
    if row == n + 1:
        count += 1
        o.append(position[1:])
        return
    for col in range(1, n + 1):
        if not check[col]:
            s = True
            for pre_row in range(1, row):
                if abs(row - pre_row) == abs(col - position[pre_row]):
                    s = False
                    break
            if s:
                position[row] = col
                check[col] = True
                A(row + 1, n, position, check, count, o)
                check[col] = False
n = 8
position = [0] * (n + 1)
check = [False] * (n + 1)
count = 0
o = []
A(1, n, position, check, count, o)
answers = []
for i in o:
    i = [str(e) for e in i]
    answers.append(''.join(i))
    answers.sort()
answer = []
k = int(input())
for i in range(k):
    answer.append(answers[int(input())-1])
for i in answer:
    print(i)
```

求排列的逆序数

```
def mergeSort(arr):
    if len(arr) <= 1:
        return arr, 0

    mid = len(arr) // 2
    left, inv_count_left = mergeSort(arr[:mid])
    right, inv_count_right = mergeSort(arr[mid:])

    merged, inv_count = merge(left, right)
    inv_count += inv_count_left + inv_count_right

    return merged, inv_count

def merge(left, right):
    merged = []
    inv_count = 0
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
            inv_count += len(left) - i
    merged += left[i:]
    merged += right[j:]

    return merged, inv_count

# 输入排列
n = int(input())
arr = list(map(int, input().split()))

# 调用归并排序函数并输出逆序数
sorted_arr, inv_count = mergeSort(arr)
print(inv_count)
```

欧拉筛找素数

```
def euler_sieve(n):
    is_prime = [True] * (n + 1)
    primes = []

    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
        for prime in primes:
            if i * prime > n:
                break
            is_prime[i * prime] = False
            if i % prime == 0:
                break

    return primes
```

回文字符串

```
#注意找到状态转移方程
s = input()
n = len(s)
dp = [[0] * n for _ in range(n)]
for i in range(n - 1, -1, -1):
    for j in range(i + 1, n):
        if s[i] == s[j]:
            dp[i][j] = dp[i + 1][j - 1]
        else:
            dp[i][j] = min(dp[i + 1][j], dp[i][j - 1], dp[i + 1][j - 1]) + 1
ans = dp[0][-1]
print(ans)
```

田忌赛马问题

```
# 搜索方法，缓存位置不对会导致wa
from functools import lru_cache
import sys
sys.setrecursionlimit(1 << 30) # 设置递归深度为 2^30（不然会RE）

def compare(a, b):
    if a > b:
        return 1
    elif a == b:
        return 0
    else:
        return -1

# 要么和最大的比，要么和最小的比，返回两种情况中大的

while True:
    n = int(input())
    if n == 0:
        break
    tian = list(map(int, input().split()))
    king = list(map(int, input().split()))
    tian.sort()
    king.sort()
    @lru_cache(maxsize = 2048)
    def dfs(start, end, i):
        if i < n:
            tian_value = tian[i]
            king_value_start = king[start]
            x1 = dfs(start + 1, end, i + 1) + compare(tian_value, king_value_start)

            king_value_end = king[end]
            x2 = dfs(start, end - 1, i + 1) + compare(tian_value, king_value_end)

            x = max(x1, x2)
            return x
        else:
            return 0
    result = dfs(0, n - 1, 0)
    print(200 * result)

# 贪心算法（难点在于对平局的处理）
def max_value(n, v1, v2):
    score = 0
    v1.sort()
    v2.sort()
    i, l = 0, 0
    j, m = n-1, n-1
    while i <= j:
        if v1[i] > v2[l]:
            score += 200
            i += 1
            l += 1
        elif v1[j] > v2[m]:
            score += 200
            j -= 1
            m -= 1
        else:
            if v1[i] < v2[m]:
                score -= 200
            i += 1
            m -= 1
    return score
while True:
    n = int(input())
    if n == 0:
        break
    else:
        v1 = list(map(int, input().split()))
        v2 = list(map(int, input().split()))
        ans = max_value(n, v1, v2)
        print(ans)
```

greedy

```
# 最大非相交区间（充实的寒假生活、进程检测）
n = int(input())
l = [list(map(int, input().split())) for _ in range(n)]
l.sort(key = lambda x: (-x[0], x[1]))
last_end = l[0][0]
ans = 1
for i in range(1, n):
    if l[i][1] < last_end:
        ans += 1
        last_end = l[i][0]
print(ans)

# 以最小的区间数覆盖整个区间（世界杯只因）
N = int(input())
r = list(map(int, input().split()))
def F(r, N):
    position = [(max(1, i + 1 - point), -min(N, i + 1 + point)) for i, point in enumerate(r)]
    position.sort()
    last_position = -position[0][1]
    cur_last_position = -position[0][1]
    if last_position == N:
        return 1
    ans = 1
    for start, end in position:
        if cur_last_position == N:
            ans += 1
            break
        if start <= last_position:
            cur_last_position = max(cur_last_position, -end)
        else:
            ans += 1
            last_position = cur_last_position
    return ans
ans = F(r, N)
print(ans)

class Solution:
    def videoStitching(self, clips: List[List[int]], time: int) -> int:
        position = [(e[0], -e[1]) for e in clips]
        position.sort()
        #return position
        last_position = -position[0][1]
        cur_last_position = -position[0][1]
        if position[0][0] > 0:
            return -1
        if last_position >= time:
            return 1
        ans = 1
        for start, end in position[1:]:
            if cur_last_position >= time:
                ans += 1
                return ans
            if start <= last_position:
                cur_last_position = max(cur_last_position, -end)
            else:
                ans += 1
                last_position = cur_last_position
        if cur_last_position >= position[-1][0] and -position[-1][1] >= time:
            return ans + 1
        return -1
# 区间分组问题 给出一堆区间，问最少可以将这些区间分成多少组使得每个组内的区间互不相交。
class Solution:
    def minmumNumberOfHost(self , n , startEnd ):
        starts=[]
        ends=[]
        for start,end in startEnd:
            starts.append(start);
            ends.append(end);

        starts.sort();
        ends.sort()

        i,j,count,res=0,0,0,0
        for time in starts:
            while(i<n and starts[i]<=time):
                i+=1
                count+=1
            while(j<n and ends[j]<=time):
                j+=1
                count-=1
            if res<count:
                res=count
        return res
```

康特展开

```
# 用来求解排列在全排列中的位置
import math
n = int(input())
arr = list(map(int, input().split()))
mod = 998244353
ans = 0
for i in range(n - 1):
    cur_num = 0
    for j in range(i + 1, n):
        if arr[j] < arr[i]:
            cur_num += 1
    ans = (ans + cur_num * math.factorial(n - i - 1)) % mod
print((ans + 1) % mod)
```

Kadane's Algorithm

```
# 练习02766: 最大子矩阵
import sys
input = sys.stdin.read()
data = list(map(int, input.split()))
N = data[0]
matrix = []
for i in range(1, N ** 2 + 1, N):
    line = data[i : i + N]
    matrix.append(line)
total_max_sum = float('-inf')
for top in range(N):
    temp = [0] * N
    for bottom in range(top, N):
        cur_max_num = 0
        for i in range(N):
            temp[i] += matrix[bottom][i]
            if cur_max_num + temp[i] > temp[i]:
                cur_max_num += temp[i]
            else:
                cur_max_num = temp[i]
            total_max_sum = max(total_max_sum, cur_max_num)
print(total_max_sum)
```

Manacher 算法

```
def manacher(s):
    # 1. 预处理字符串
    t = '^#' + '#'.join(s) + '$'  # 字符间插入 #，从而对于偶数子串也可以中心扩展
    n = len(t)  # 得到新字符串长度
    P = [0] * n  # P[i] 表示以 t[i] 为中心的回文半径
    C, R = 0, 0  # C 为当前回文中心，R 为当前回文的右边界

    # 2. 计算回文半径
    for i in range(1, n - 1):
        # 如果 i 在 R 范围内，用对称位置的回文半径初始化 P[i]
        P[i] = min(R - i, P[2 * C - i]) if i < R else 0

        # 中心扩展，尝试扩展回文半径
        while t[i + P[i] + 1] == t[i - P[i] - 1]:
            P[i] += 1

        # 更新回文的中心和右边界
        if i + P[i] > R:
            C, R = i, i + P[i]

    # 3. 找到最长回文
    max_len = max(P)  # 最长回文半径
    center_index = P.index(max_len)  # 最长回文对应的中心索引

    # 原始字符串中的起始索引
    start = (center_index - max_len) // 2
    return s[start:start + max_len]
```

树形dp

```
n = int(input())
r = [0] * (n + 1)
for i in range(1, n + 1):
    r[i] = int(input())

children = [[] for _ in range(n + 1)]
parent = [0] * (n + 1)
for _ in range(n - 1):
    l, k = map(int, input().split())
    children[k].append(l)
    parent[l] = k
```

```
root = None
for i in range(1, n + 1):
    if parent[i] == 0:
        root = i
        break

dp = [[0, 0] for _ in range(n + 1)]
stack = [(root, False)]

while stack:
    node, visited = stack.pop()
    if not visited:
        stack.append((node, True))
        for child in children[node]:
            stack.append((child, False))
    else:
        sum0 = 0
        sum1 = r[node]
        for child in children[node]:
            sum0 += max(dp[child][0], dp[child][1])
            sum1 += dp[child][0]
        dp[node][0] = sum0
        dp[node][1] = sum1

print(max(dp[root][0], dp[root][1]))
```

基础语法

二分

```
import bisect

a = [1, 2, 4, 4, 8]
print(bisect.bisect_left(a, 4))  # 输出：2
print(bisect.bisect_right(a, 4))  # 输出：4
print(bisect.bisect(a, 4))  # 输出：4
```

优先队列

```
import heapq

data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapq.heapify(data)
print(data)  # 输出：[0, 1, 2, 3, 9, 5, 4, 6, 8, 7]

heapq.heappush(data, -5)
print(data)  # 输出：[-5, 0, 2, 3, 1, 5, 4, 6, 8, 7, 9]

print(heapq.heappop(data))  # 输出：-5
```

日期与时间

```python
import calendar, datetime

print(calendar.isleap(2020))  # 输出: True
print(datetime.datetime(2023, 10, 5).weekday())  # 输出: 3 (星期四)

# 天数差值
base_date = datetime.datetime(2022, 1, 7)
start_str, end_stre = input().split()
start = (datetime.datetime.strptime(f"2022-{start_str}", "%Y-%m.%d") - base_date).days
end = (datetime.datetime.strptime(f"2022-{end_str}", "%Y-%m.%d") - base_date).days
```

数据结构

```python
import collections

# deque
dq = collections.deque([1, 2, 3])
dq.append(4)
print(dq)  # 输出: deque([1, 2, 3, 4])
dq.appendleft(0)
print(dq)  # 输出: deque([0, 1, 2, 3, 4])
dq.pop()
print(dq)  # 输出: deque([0, 1, 2, 3])
dq.popleft()
print(dq)  # 输出: deque([1, 2, 3])

dd = collections.defaultdict(int)
dd['a'] += 1
print(dd)  # 输出: defaultdict(<class 'int'>, {'a': 1})

od = collections.OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od)  # 输出: OrderedDict([('a', 1), ('b', 2), ('c', 3)])

Point = collections.namedtuple('Point', ['x', 'y'])
p = Point(11, 22)
print(p)  # 输出: Point(x=11, y=22)
print(p.x, p.y)  # 输出: 11 22
```

遍历

```python
import itertools
for item in itertools.product('AB', repeat=2):
    print(item)  # 输出: ('A', 'A'), ('A', 'B'), ('B', 'A'), ('B', 'B')
```

函数

```python
import functools
print(functools.reduce(lambda x, y: x + y, [1, 2, 3, 4]))  # 输出: 10
```

分数和有理数

```python
import fractions
import decimal

frac = fractions.Fraction(1, 3)
print(frac)  # 输出: 1/3

dec = decimal.Decimal('0.1')
print(dec)  # 输出: 0.1
```

数学

```python
import math
print(math.ceil(4.2))  # 输出: 5 向上取整
print(math.floor(4.2))  # 输出: 4 向下取整
print('%.2f'%0.322) #输出0.32（保留两位小数）
abs() pow(x, y) bin(num) oct(num) hex(num) int('123', base = k)
math.factorial(n) #阶乘
math.gcd(a, b) #最大公约数
(a * b) // math.gcd(a, b) #最小公倍数
#辗转相除:
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

拷贝

```python
import copy

original = [1, 2, [3, 4]]
copied = copy.deepcopy(original)
print(copied)  # 输出: [1, 2, [3, 4]]
```

数组操作

```python
squared = list(map(lambda x: x**2, [1, 2, 3, 4]))
print(squared)  # 输出: [1, 4, 9, 16]

a = [1, 2, 3]
b = ['a', 'b', 'c']
zipped = list(zip(a, b))
print(zipped)  # 输出: [(1, 'a'), (2, 'b'), (3, 'c')]

filtered = list(filter(lambda x: x > 2, [1, 2, 3, 4]))
print(filtered)  # 输出: [3, 4]

enumerated = list(enumerate(['a', 'b', 'c']))
print(enumerated)  # 输出: [(0, 'a'), (1, 'b'), (2, 'c')]
```

排序

```
from functools import cmp_to_key
def compare(a, b):
    if str(a) + str(b) < str(b) + str(a):
        return 1
    if str(a) + str(b) == str(b) + str(a):
        return 0
    if str(a) + str(b) > str(b) + str(a):
        return -1

m = int(input())
n = int(input())
num = list(map(int, input().split()))
num.sort(key = cmp_to_key(compare))
# 冒泡排序
n = int(input())
num=input.split()
for i in range(n-1):
    for j in range(i+1,n):
        if num[i]+num[j]<num[j]+num[i]: #这里是字符串的组合，比较的是字典序
            num[i],num[j]=num[j],num[i]
```

字符串处理

```
print(input().swapcase()) #大小写互换
```