并查集：

```
parent = list(range(n+1))
rank = [1]*(n+1)
size = [1]*(n+1)  # 新增：记录集合大小
def find(x):
    while parent[x] != x:
        parent[x] = parent[parent[x]]
        x = parent[x]
    return x
def union(x, y):
    root_x = find(x)
    root_y = find(y)
    if root_x == root_y:
        return
    # 按秩合并，小秩挂到大秩上
    if rank[root_x] < rank[root_y]:
        parent[root_x] = root_y
        size[root_y] += size[root_x]  # 合并大小
    else:
        parent[root_y] = root_x
        size[root_x] += size[root_y]  # 合并大小
        if rank[root_x] == rank[root_y]:
            rank[root_x] += 1
# 查询 x 所在集合的大小
def get_size(x):
    return size[find(x)]  # 必须通过根节点获取
另外一个变式  # 统计连通分量个数（即宗教数上限）
roots = set()
for i in range(1, n + 1):
roots.add(find(i))  # 必须再次 find 确保路径压缩后的根节点
```

经典例子：食物链问题（A 吃 B、B 吃 C、C 吃 A）

定义 weight[x] 表示 x 相对于根节点的「类别」：0 = 和根同类，1 = 吃根，2 = 被根吃；

合并 A 和 B 时，若已知「A 吃 B」，则可推导 root_A 和 root_B 的关系，更新 weight[root_A]；

查询 A 和 B 的关系时，通过 (weight[A] - weight[B]) % 3 判断（0 = 同类，1=A 吃 B，2=B 吃 A）。

```
parent = list(range(n+1))
weight = [0]*(n+1)  # x 到 parent[x]的关系值
def find(x):
    if parent[x] != x:
        # 递归查找根节点，先更新父节点的权值（保证传递性）
        orig_parent = parent[x]
        parent[x] = find(parent[x])  # 路径压缩：x 直接指向根
```

```
            weight[x] += weight[orig_parent]   # x 到根的关系 = x 到原父的关系 + 原父到根的关系
        return parent[x]
# 已知 x 和 y 的目标关系（如 x 到 y 的关系为 val），合并两个集合
def union(x, y, val):
        root_x = find(x)
        root_y = find(y)
        if root_x == root_y:
            return
        parent[root_x] = root_y
        # 推导 root_x 到 root_y 的关系：weight[root_x] = weight[y] + val - weight[x]
        weight[root_x] = weight[y] + val - weight[x]
```

最大上升子序列

```
n = int(input())
a = list(map(int, input().split()))
# 初始化 dp 数组：每个元素自身是长度为 1 的上升子序列
dp = [1] * n
# 遍历每个元素（从第 2 个开始，索引 1）
for i in range(1, n):
    # 回顾 i 之前的所有元素
    for j in range(i):
        # 若 a[j] < a[i]，可接在后面形成更长子序列
        if a[j] < a[i]:
            dp[i] = max(dp[i], dp[j] + 1)
# dp 数组的最大值即为答案
print(max(dp))
```

约瑟夫环

```
while 1:
    n,p,m=map(int,input().split())
    pos=p-1
    if n==p==m==0:
        break
    else:
        li=list(range(1,n+1))
        ans=[]
        while len(li)>0:
            pos=(pos+m-1)%n
            ans.append(li.pop(pos))
            n-=1
        print(",".join(map(str,ans)))
```

栈：

```
while True:
    try:
        s = input()
```

代码模板（右侧最近更小）

```python
def rightSmaller(nums):
    n = len(nums)
    right = [n] * n
    stack = []

    for i in range(n):
        while stack and nums[stack[-1]] > nums[i]:
            right[stack.pop()] = i
        stack.append(i)

    return right
```

求根

```
# 验证初始区间是否满足介值定理（两端异号）
if f(left) * f(right) >= 0:
    raise ValueError("初始区间两端函数值需异号（f(left)*f(right) < 0）")

# 二分迭代：缩小区间直到满足精度
while right - left > eps:
    mid = (left + right) / 2  # 区间中点
    f_mid = f(mid)

    # 若中点函数值接近 0，直接返回（提前终止）
    if abs(f_mid) < eps:
        return mid

    # 根在左半区间：f(mid) 与 f(left) 异号
    if f(left) * f_mid < 0:
        right = mid
    # 根在右半区间：f(mid) 与 f(right) 异号
    else:
        left = mid

# 返回最终区间中点（近似根）
return (left + right) / 2
```

```
        n = len(s)
        res = [' '] * n
        stack = []
        for idx, char in enumerate(s):
            if char == '(':
                stack.append(idx)
            elif char == ')':
                if stack:
                    stack.pop()
                else:
                    res[idx] = '?'
        for idx in stack:
            res[idx] = '$'
        print(s)
        print(''.join(res))
    except EOFError:
        break
```

二分查找:

```
while left <= right:  # 循环条件：区间不为空（left == right 时仍需检查）
    # 计算中间索引（避免 left + right 溢出，等价于 (left + right) // 2）
    mid = left + (right - left) // 2

    if nums[mid] == target:
        return mid  # 找到目标，直接返回索引
    elif nums[mid] < target:
        left = mid + 1  # 目标在右半区间，左边界右移（mid 已排除）
    else:
        right = mid - 1  # 目标在左半区间，右边界左移（mid 已排除）
```

递归：

```
from functools import lru_cache

@lru_cache(maxsize=None)  # 无限缓存
def factorial(n):
```

```
while left <= right:
    mid = left + (right - left) // 2
    if nums[mid] == target:
        return mid

    # 左半段有序（nums[left] <= nums[mid]，因数组无重复元素）
    if nums[left] <= nums[mid]:
        # target 在左半段：缩小右边界
        if nums[left] <= target < nums[mid]:
            right = mid - 1
        # target 在右半段：缩小左边界
        else:
            left = mid + 1
    # 右半段有序
    else:
        # target 在右半段：缩小左边界
        if nums[mid] < target <= nums[right]:
            left = mid + 1
        # target 在左半段：缩小右边界
        else:
            right = mid - 1
```

双指针：

```
i=0
j=len(l)-1
while i<j:
    if l[i]+l[j]<target:
        i+=1
    elif l[i]+l[j]>target:
        j-=1
    if l[i]+l[j]==target:
        print(l[i],l[j])
```

```
字典.get(key, default=None)
```

参数说明：

- `key`：**必填**，要查找的字典键（可以是任意可哈希类型，如字符串、数字、元组）。
- `default`：**可选**，当 `key` 不在字典中时，返回的默认值（默认值为 `None`）。

哈希查找：

```
hash_dict = {}  # 键：数值，值：索引
for idx, num in enumerate(nums):
    complement = target - num
    # 检查互补数是否已在哈希表中
    if complement in hash_dict:
        return [hash_dict[complement], idx]
    # 存入当前数和索引（避免重复使用同一个元素）
    hash_dict[num] = idx
```

```
if len(nums1) > len(nums2):
    nums1, nums2 = nums2, nums1

freq1 = Counter(nums1)
res = []

for num in nums2:
    if freq1.get(num, 0) > 0:
        res.append(num)
        freq1[num] -= 1  # 减少计
        if freq1[num] == 0:
            del freq1[num]  # 优

return res
```

求交集

```
from collections import Counter
```

```
if len(s) != len(t):
    return False
# 统计字符频率并比较
return Counter(s) == Counter(t)
```

字母出现频率是否一样

滑动窗口：

```
left = 0
right = 0
window_sum = 0  # 窗口内元素和（根据题目替换为
max_result = -float('inf')  # 存储最优结果（根

# 第一步：初始化窗口（先让右指针移动 k 步，构建
while right < k:
    window_sum += nums[right]
    right += 1
# 初始窗口的结果
max_result = max(max_result, window_sum)

# 第二步：滑动窗口（左右指针同步移动，维持窗口大
while right < n:
    # 移除左指针离开窗口的元素
    window_sum -= nums[left]
    left += 1
    # 加入右指针进入窗口的新元素
    window_sum += nums[right]
    right += 1
    # 更新最优结果
    max_result = max(max_result, window_sum)

return max_result
```

```
n = len(nums)
left = 0
right = 0
window_sum = 0  # 窗口内统计量（根据题目替换）
max_len = 0  # 最长窗口长度

while right < n:
    # 1. 扩大窗口：加入当前右指针元素，更新窗口统计量
    window_sum += nums[right]
    right += 1

    # 2. 缩小窗口：当窗口不满足条件时，左指针右移（直到满足条件）
    # 条件根据题目调整（如 window_sum > target 时不满足，需缩小）
    while window_sum > target:
        window_sum -= nums[left]
        left += 1

    # 3. 更新最长窗口长度（此时窗口已满足条件）
    current_len = right - left
    if current_len > max_len:
        max_len = current_len

return max_len
```

```
n = len(nums)
left = 0
right = 0
window_sum = 0  # 窗口内统计量（根据题目替换）
min_len = float('inf')  # 最短窗口长度（初始化为无穷大）

while right < n:
    # 1. 扩大窗口：加入当前右指针元素，更新窗口统计量
    window_sum += nums[right]
    right += 1

    # 2. 缩小窗口：当窗口满足条件时，尝试左指针右移，缩小窗口到最
    # 条件根据题目调整（如 window_sum >= target 时满足，需缩小）
    while window_sum >= target:
        current_len = right - left
        # 更新最短窗口长度
        if current_len < min_len:
            min_len = current_len
        # 移除左指针元素，缩小窗口
        window_sum -= nums[left]
        left += 1
```

```
n = len(s)
left = 0
max_len = 0
window_set = set()  # 存储窗口内的字符（无重复）

for right in range(n):
    # 缩小窗口：当前字符重复，左指针右移并移除字符
    while s[right] in window_set:
        window_set.remove(s[left])
        left += 1
    # 扩大窗口：加入当前字符
    window_set.add(s[right])
    # 更新最长长度
    current_len = right - left + 1
    max_len = max(max_len, current_len)

return max_len
```

```python
def min_window(s: str, t: str) -> str:  1 usage
    # 1. 统计 t 中每个字符需要的数量：need
    need = {}
    for ch in t:
        need[ch] = need.get(ch, 0) + 1

    window = {}                # 当前窗口中各字符的计数
    required = len(need)       # 需要满足条件的字符种类数
    valid = 0                  # 当前窗口中已经满足 need 的字符种类数

    # 结果：记录当前最短窗口的长度和左右端点
    min_len = float("inf")
    res_l, res_r = 0, 0

    L = 0  # 左指针

    # 2. 右指针 R 扫描 s
    for R in range(len(s)):
        ch = s[R]

        # 把 s[R] 加入窗口
        if ch in need:
            window[ch] = window.get(ch, 0) + 1
            if window[ch] == need[ch]:
                valid += 1

        # 3. 当窗口已经满足所有需求时，开始收缩左边
        while valid == required:
            # 更新答案
            if R - L + 1 < min_len:
                min_len = R - L + 1
                res_l, res_r = L, R

            # 准备把 s[L] 移出窗口
            left_char = s[L]
            if left_char in need:
                window[left_char] -= 1
                # 一旦数量少于 need，就不再满足，对应的种类减一
                if window[left_char] < need[left_char]:
                    valid -= 1

            L += 1  # 左指针右移，窗口缩小

    # 如果 min_len 没被更新，说明不存在这样的子串
    if min_len == float("inf"):
        return ""
    else:
        return s[res_l:res_r + 1]

s = "ADOBECODEBANCABC"
t = "ABCC"
print(min_window(s, t))  # 输出 "CABC"
```

Q:给定字符串 s 和 t，在 s 中找一个最短的子串，使得其中包含 t 中的所有字符，同时这些字符的数目也要满足要求。

Dp:

场景：

硬币有使用次数限制（如硬币 1 最多用 3 次），凑成目标金额的最少硬币数 / 方案数。

状态定义：

`dp[i]` = 凑成金额 `i` 的最少硬币数（或组合数）。

转移方程（以最少硬币数为例）：

python

```python
for coin, cnt in zip(coins, counts):  # counts 是对应硬币的最大使用次数
    for j in range(amount, coin - 1, -1):  # 逆序遍历（类似 01 背包）
        # 尝试使用 1~cnt 枚当前硬币
        for k in range(1, cnt + 1):
            if j >= k * coin:
                dp[j] = min(dp[j], dp[j - k * coin] + k)
```

最长公共子序列

python

```python
if s1[i-1] == s2[j-1]:
    # 字符匹配，长度 = 前 i-1 和 j-1 的长度 + 1
    dp[i][j] = dp[i-1][j-1] + 1
else:
    # 字符不匹配，取「前 i-1 个与 j 个」或「前 i 个与 j-1 个」的最大值
    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

回文子串

```python
if s[i] == s[j]:
    if j - i <= 1:  # 长度为 1（i=j）或 2（j=i+1），直
        dp[i][j] = True
    else:  # 长度 >2，取决于内部子串是否为回文
        dp[i][j] = dp[i+1][j-1]
else:
    dp[i][j] = False
```

状态定义：

`dp[i][j]` = 字符串 `s[i..j]`

初始化：

`dp[i][i] = True`（单个字符是回文）。

状态定义：

`dp[i][j]` = 将 `word1[0..i-1]` 转换成 `word2[0..j-1]` 的最少操作数。

转移方程：

python

```python
if word1[i-1] == word2[j-1]:
    # 字符匹配，无需操作，直接继承前一状态
    dp[i][j] = dp[i-1][j-1]
else:
    # 不匹配，取「插入、删除、替换」的最小值 +1
    dp[i][j] = min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1]) + 1
```

初始化：

`dp[i][0] = i`（`word2` 为空，需删除 `i` 个字符），`dp[0][j] = j`（`word1` 为空，需插入 `j` 个字符）

场景：

从网格左上角到右下角，只能向右或向下走，求路径上所有数字的最小和。

状态定义：

`dp[i][j]` = 到达 `(i,j)` 的最小路径和。

转移方程：

python

```python
dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]  # 取上方或左方的最小值 + 当前格子值
```

数组 `nums` 代表房屋金额，不能偷相邻房屋，求能偷的最大金额。

状态定义：

`dp[i]` = 前 `i` 个房屋能偷的最大金额。

转移方程：

`dp[0] = 0`（0 个房屋），`dp[1] = nums[0]`（1 个房

python

```python
dp[i] = max(dp[i-1], dp[i-2] + nums[i-1])  # 不偷第 i 个，或偷第 i 个（不偷第 i-1 个
```

2. 打家劫舍 II（环形房屋，LeetCode 213）

场景：

房屋围成一圈（第一个和最后一个相邻），不能偷相邻房屋，求最大金额。

核心思路：

拆分为两个线性问题：「不偷第一个房屋」和「不偷最后一个房屋」，取两者最大值。

转移方程：

复用打家劫舍 I 的转移方程，分别计算两个子问题的结果，最终返回 `max(rob1, rob2)`。

栈的一些其他问题：

```python
bracket_map = {')': '(', '}': '{', ']': '['}
stack = []  # 存储左括号

for char in s:
    if char in bracket_map:
        # 遇到右括号：栈空或栈顶不匹配 → 无效
        top = stack.pop() if stack else '#'  # 栈空时用#填充，
        if bracket_map[char] != top:
            return False
    else:
        # 遇到左括号 → 入栈
        stack.append(char)

# 遍历结束后栈为空 → 所有左括号都闭合
return len(stack) == 0
```

```python
stack = []
current_num = 0  # 当前解析的数字
current_sign = 1  # 当前数字的符号（1:正，-1:负）
result = 0       # 累计结果

for char in s:
    if char.isdigit():
        # 解析多位数（如 "123" → 123）
        current_num = current_num * 10 + int(char
    elif char == '+':
        # 累加当前数字，更新符号
        result += current_sign * current_num
        current_sign = 1
        current_num = 0
    elif char == '-':
        # 累加当前数字，更新符号
        result += current_sign * current_num
        current_sign = -1
        current_num = 0
    elif char == '(':
        # 入栈：当前结果和符号，重置状态计算括号内
        stack.append((result, current_sign))
        result = 0
        current_sign = 1
    elif char == ')':
        # 累加括号内最后一个数字，弹出栈顶合并结果
        result += current_sign * current_num
        prev_result, prev_sign = stack.pop()
        result = prev_result + prev_sign * result
        current_num = 0

# 累加最后一个数字
result += current_sign * current_num
return result
```

题目描述

给定 `n` 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。求在该柱状图中，能够勾勒出来的矩形的最大面积。

解题思路

- 核心：每个柱子的最大矩形面积 = 高度 × 宽度（宽度 = 右边界 - 左边界 - 1）；
- 单调栈：维护栈内柱子索引 **单调递增**，用于找每个柱子的「左边界」（栈顶索引，小于当前柱子高度）和「右边界」（当前索引，小于当前柱子高度）；
- 技巧：在数组首尾添加 0（哨兵），避免处理栈空和数组末尾的边界情况。

代码实现

python 运行

```python
def largestRectangleArea(heights: list[int]) -> int:
    # 首尾添加哨兵（0），简化边界处理
    heights = [0] + heights + [0]
    stack = []
    max_area = 0

    for idx in range(len(heights)):
        # 当前柱子高度 < 栈顶柱子高度 → 计算栈顶柱子的最大面积
        while stack and heights[idx] < heights[stack[-1]]:
            h = heights[stack.pop()]  # 栈顶柱子的高度
            w = idx - stack[-1] - 1  # 宽度 = 右边界（idx） - 左边界（新栈顶） - 1
            max_area = max(max_area, h * w)
        # 当前索引入栈
        stack.append(idx)

    return max_area
```

排序：

```python
def selection_sort(a):
    n = len(a)
    for i in range(n):
        min_pos = i
        for j in range(i + 1, n):
            if a[j] < a[min_pos]:
                min_pos = j
        a[i], a[min_pos] = a[min_pos], a[i]
```

```python
def bubble_sort(a):
    n = len(a)
    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swapped = True
        if not swapped:   # 提前结束
            break
```

Greedy：

合并区间

```
    sorted_acts = sorted(activities, key=lambda x: x[1])
    if not sorted_acts:  # 空活动列表，直接返回
        return []

    # 步骤2：初始化贪心解G，选择第一个活动（结束时间最早的g1，对应证明中的第一步）
    selected = [sorted_acts[0]]

    # 步骤3：遍历剩余活动，依次选择「不与已选最后一个活动冲突」且「结束最早」的活动
    for act in sorted_acts[1:]:
        # 冲突判断：当前活动的开始时间 ≥ 已选最后一个活动的结束时间
        if act[0] >= selected[-1][1]:
            selected.append(act)  # 加入贪心解G

    return selected
```

```
sorted_intervals = sorted(intervals, key=lambda x: x[0])
merged = [sorted_intervals[0]]

for curr in sorted_intervals[1:]:
    last = merged[-1]
    if curr[0] <= last[1]:  # 重叠/相邻，合并
        last[1] = max(last[1], curr[1])
    else:  # 不重叠，加入结果
        merged.append(curr)
```

```
max_reach = 0  # 当前能到达的最远位置
n = len(nums)

for i in range(n):
    if i > max_reach:  # 当前位置无法到达，直接返回False
        return False
    # 更新最远可达位置
    max_reach = max(max_reach, i + nums[i])
    if max_reach >= n - 1:  # 提前终止：已能到达终点
        return True

return max_reach >= n - 1
```

```
n = len(nums)
if n <= 1:
    return 0

step = 0  # 跳跃次数
cur_end = 0  # 当前跳跃的边界
max_reach = 0  # 下一步能到达的最远位置

for i in range(n - 1):  # 无需遍历到最后一位（已到达）
    max_reach = max(max_reach, i + nums[i])
    if i == cur_end:  # 遍历完当前边界，必须跳一步
        step += 1
        cur_end = max_reach  # 更新边界为下一步最远位置
        if cur_end >= n - 1:  # 提前终止
            break

return step
```

```
n = len(ratings)
candies = [1] * n  # 每个孩子至少1颗

# 左→右：处理比左边评分高的情况
for i in range(1, n):
    if ratings[i] > ratings[i-1]:
        candies[i] = candies[i-1] + 1

# 右→左：处理比右边评分高的情况（取最大值）
for i in range(n-2, -1, -1):
    if ratings[i] > ratings[i+1]:
        candies[i] = max(candies[i], candies[i+1] + 1)

return sum(candies)
```

```
profit = 0
for i in range(1, len(prices)):
    # 核心：累加所有正差价
    if prices[i] > prices[i-1]:
        profit += prices[i] - prices[i-1]
return profit
```

```
from functools import cmp_to_key

def largestNumber(nums):
    # 定义cmp函数：比较a+b和b+a的字符串大小
    def compare(a, b):
        s1 = str(a) + str(b)
        s2 = str(b) + str(a)
        if s1 > s2:
            return -1  # s1大则a排前面
        elif s1 < s2:
            return 1
        else:
            return 0

    # 排序并处理全0情况
    sorted_nums = sorted(nums, key=cmp_to_key(compare))
    res = ''.join(map(str, sorted_nums))
    return '0' if res[0] == '0' else res
```

Dfs:

```
def dfs(u):
    vis[u] = True
    for v in adj[u]:
        if not vis[v]
            dfs(v)
```

```
def backtrack(路径, 选择列表, 结果列表):
    if 终止条件:
        结果列表.append(路径.copy())  # 注意：需拷贝路径，避免后续修改影响结果
        return
    for 选择 in 选择列表:
        if 选择不合法（剪枝）:
            continue
        做出选择（路径.append(选择)）
        backtrack(路径, 新的选择列表, 结果列表)
        撤销选择（路径.pop()）
```

```
ans=[]
def queen(rows,cols,cn):  2 用法
    if rows==8:
        ans.append(int("".join(map(str,cn))))
        return
    for j in range(8):
        if cols[j]==0:
            check=0
            for x in range(len(cn)):
                if abs(rows-x)==abs(cn[x]-1-j):
                    check=1
                    break
            if check==0:
                cols[j]=1
                cn.append(j+1)
                queen(rows+1,cols,cn)
                cols[j]=0
                cn.pop()
li=[0]*8
queen( rows: 0,li, cn: [])
ans.sort()
n=int(input())
for _ in range(n):
    print(ans[int(input())-1])
```

```
def dfs(x,y,c):  3 用法
    ma[x][y]="#"
    di=[[0,1],[0,-1],[-1,0],[1,0]]
    for i in range(4):
        tx=x+di[i][0]
        ty=y+di[i][1]
        if ma[tx][ty]==c:
            dfs(tx,ty,c)
```

```
n,k=map(int,input().split())
num=list(map(str,input().split()))
ans=[]
def dfs(idx,temp):  3 用法
    if len(temp)==k:
        ans.append(temp[:])
        return
    if idx>n-1:
        return
    temp.append(num[idx])
    dfs(idx+1,temp)
    temp.pop()
    dfs(idx+1,temp)
dfs( idx: 0, temp: [])
for i in ans:
    print(" ".join(list(map(str,i))))
```

```
def dfs(x,y):  2 用法
    if memo[x][y]!=0:
        return memo[x][y]
    di=[(0,1),(0,-1),(1,0),(-1,0)]
    max_len=1
    for i in di:
        tx=x+i[0]
        ty=y+i[1]
        if ma[tx][ty]<ma[x][y]:
            current=1+dfs(tx,ty)
            if current>max_len:
                max_len=current
    memo[x][y]=max_len
    return max_len
ans=0
for s in range(1,r+1):
    for t in range(1,c+1):
        a=dfs(s,t)
        if a>ans:
            ans=a
print(ans)
```

Bfs:

最短路径

```
from collections import deque
dx=[0,0,-1,1]
dy=[-1,1,0,0]
n,m=map(int,input().split())
ma=[]
for _ in range(n):
    ma.append(list(map(int,input().split())))
def bfs(sx,sy):
    q=deque()
    q.append((sx,sy))
    inq=set()
    prev = [[(-1, -1) for _ in range(m)] for _ in range(n)]
    inq.add((sx,sy))
    while q:
        x,y=q.popleft()
        if x==n-1 and y==m-1:
            return prev
        for i in range(4):
            tx=x+dx[i]
            ty=y+dy[i]
            if 0 <= tx < n and 0 <= ty < m and ma[tx][ty] == 0 and (tx, ty) not in inq:
                prev[tx][ty]=(x,y)
                inq.add((tx,ty))
                q.append((tx,ty))
    return None
prevc=bfs(0,0)
path=[]
end=(n-1,m-1)
while end!=(-1,-1):
    path.append(end)
    end=prevc[end[0]][end[1]]
path.reverse()
for pos in path:
    print(pos[0]+1,pos[1]+1)
```

多终点

```
from collections import deque
dx=[0,0,-1,1]
dy=[-1,1,0,0]
n,m=map(int,input().split())
ma=[]
for _ in range(n):
    ma.append(list(map(int,input().split())))
def bfs(sx,sy):
    ans = [[(-1) for i in range(m)] for _ in range(n)]
    q=deque()
    q.append((0,sx,sy))
    inq=set()
    inq.add((sx,sy))
    while q:
        step,x,y=q.popleft()
        for i in range(4):
            tx=x+dx[i]
            ty=y+dy[i]
            if 0 <= tx < n and 0 <= ty < m and ma[tx][ty] == 0 and (tx, ty) not in inq:
                ans[tx][ty]=step+1
                inq.add((tx,ty))
                q.append((step+1,tx,ty))
    return ans
ans_=bfs(0,0)
ans_[0][0]=0
for i in range(n):
    ans_[i]=list(map(str,ans_[i]))
    print(" ".join(ans_[i]))
```

```
from collections import deque

def bfs(start, target, adj):
    """
    通用BFS模板（无向图）
    :param start: 起始节点
    :param target: 目标节点（可选，视问题而定）
    :param adj: 邻接表（表示图的连接关系，如 adj[node] = [邻接节点1, 邻接节点2...]
    :return: 最短路径长度 / 遍历结果 / 是否可达
    """
    # 1. 初始化队列和访问标记（避免重复访问）
    queue = deque()
    visited = set()  # 若节点是数字/可哈希类型，用集合；若为网格坐标，可用二维数组

    queue.append(start)
    visited.add(start)

    # 可选：记录路径长度（如最短路径问题）
    step = 0

    # 2. 循环处理队列，直到为空
    while queue:
        # 可选：层序遍历——处理当前层的所有节点（关键！）
        current_layer_size = len(queue)
        for _ in range(current_layer_size):
            # 取出队首节点
            node = queue.popleft()

            # 终止条件：找到目标节点
            if node == target:
                return step  # 或其他结果

            # 处理当前节点（如收集结果、更新状态）
            # process(node)

            # 遍历所有邻接节点
            for neighbor in adj[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

        # 每处理完一层，路径长度+1
        step += 1

    # 若未找到目标（如不可达）
    return -1  # 或返回遍历结果列表
```

```
def dijkstra(n: int, adj: List[List[Tuple[int, int]]], start: int) -> List[int]:
    """
    :param n: 节点总数（0~n-1）
    :param adj: 邻接表，adj[u] = [(v, w)] 表示u→v的边权为w
    :param start: 起点节点
    :return: 起点到各节点的最短距离（不可达为inf）
    """
    INF = float('inf')
    dist = [INF] * n  # 距离数组：dist[node] = 起点到node的最短距离
    dist[start] = 0
    heap = []
    heapq.heappush(heap, (0, start))  # (当前距离, 节点)
    visited = set()  # 标记已确定最短路径的节点（优化）

    while heap:
        curr_dist, u = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        # 松弛操作：遍历邻接节点
        for v, w in adj[u]:
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                heapq.heappush(heap, (dist[v], v))
    return dist

# 示例调用
if __name__ == "__main__":
    # 邻接表：节点0→1(1)、0→2(4)；节点1→2(2)、1→3(5)；节点2→3(1)
    adj = [[(1,1), (2,4)], [(2,2), (3,5)], [(3,1)], []]
    n = 4
    start = 0
    print(dijkstra(n, adj, start))  # [0, 1, 3, 4]
```

```
    网格版Dijkstra：求(start→end)的最小代价（边权=高度差绝对值）
    :param heights: 网格高度矩阵
    :param start: 起点坐标(x1,y1)
    :param end: 终点坐标(x2,y2)
    :return: 最小代价（不可达返回-1）
    """
    rows, cols = len(heights), len(heights[0])
    dirs = [(-1,0), (1,0), (0,-1), (0,1)]  # 上下左右
    INF = float('inf')
    # 距离矩阵：dist[x][y] = 起点到(x,y)的最小代价
    dist = [[INF]*cols for _ in range(rows)]
    sx, sy = start
    dist[sx][sy] = 0
    heap = []
    heapq.heappush(heap, (0, sx, sy))  # (当前代价, x, y)
    visited = set()

    while heap:
        curr_cost, x, y = heapq.heappop(heap)
        if (x, y) == end:
            return curr_cost  # 提前终止，返回结果
        if (x, y) in visited:
            continue
        visited.add((x, y))
        # 遍历四个方向
        for dx, dy in dirs:
            nx, ny = x+dx, y+dy
            if 0<=nx<rows and 0<=ny<cols:
                # 计算当前边权（根据题目调整，此处为高度差绝对值）
                cost = abs(heights[x][y] - heights[nx][ny])
                new_cost = max(curr_cost, cost)  # 体力消耗取路径最大边权
                if new_cost < dist[nx][ny]:
                    dist[nx][ny] = new_cost
                    heapq.heappush(heap, (new_cost, nx, ny))
    return -1  # 不可达

# 示例调用
heights = [[1,2,2],[3,8,2],[5,3,5]]
start = (0,0)
end = (2,2)
print(dijkstra_grid(heights, start, end))  # 2
```

埃氏筛：

```python
def sieve_of_eratosthenes(n):
    # 初始化布尔数组，假设所有数是素数
    is_prime = [True] * (n + 1)
    is_prime[0], is_prime[1] = False, False  # 0 和 1 不是素数

    # 从 2 开始筛选，直到 sqrt(n)
    for i in range(2, int(n**0.5) + 1):
        if is_prime[i]:
            # 标记 i 的所有倍数为非素数
            for j in range(i * i, n + 1, i):
                is_prime[j] = False

    # 返回所有素数
    primes = [i for i in range(2, n + 1) if is_prime[i]]
    return primes

# 示例：查找 30 以下的所有素数
print(sieve_of_eratosthenes(30))
```

欧拉筛：

```python
def sieve_of_euler(n):
    # 初始化布尔数组，假设所有数都是素数
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False  # 0 和 1 不是素数
    primes = []  # 存储素数

    # 遍历从 2 到 n 的所有数
    for i in range(2, n + 1):
        if is_prime[i]:  # 只有是素数才继续筛选
            primes.append(i)  # 将当前素数加入素数列表

        # 遍历已找到的素数并标记它们的倍数
        for p in primes:
            # 如果 i * p > n，就跳出循环
            if i * p > n:
                break
            is_prime[i * p] = False  # 标记倍数为合数
            if i % p == 0:  # 只标记最小素因子的倍数
                break

    return primes
```

约数筛

```python
d = [0] * (N + 1)
for i in range(1, N + 1):
    for j in range(i, N + 1, i):
        d[j] += 1
```

分解质因数

```python
def factor(n: int) -> list[tuple[int, int]]:
    """返回 n 的质因数分解 [(p, a), ...]"""
    res = []
    d = 2
    while d * d <= n:
        if n % d == 0:
            cnt = 0
            while n % d == 0:
                n //= d
                cnt += 1
            res.append((d, cnt))
        d += 1
    if n > 1:
        res.append((n, 1))
    return res
```

math 库里的 math.gcd(a,b)和 math.lcm(a,b)

堆：

源代码

```
import heapq
n=int(input())
stop=[]
for _ in range(n):
    l,p=map(int,input().split())
    stop.append((l,p))
l,p=map(int,input().split())
solutions=[(l-d,f) for d,f in stop]
solutions.append((l,0))
solutions.sort()
heap=[]
s=0
count=0
check=1
for x,y in solutions:
    need=x-s
    while p<need:
        if not heap:
            check=0
            break
        max_f=-heapq.heappop(heap)
        p+=max_f
        count+=1
    if check==0:
        break
    p-=need
    s=x
    heapq.heappush(heap,-y)
print(count if check==1 else -1)
```

©2002-2022 POJ 京ICP备20010980号-1

`heapq.heappush(heap, item)`	向堆 `heap` 中插入元素 `item`	插入后保持小顶堆特性
`heapq.heappop(heap)`	弹出堆顶元素（最小值）	弹出后重新调整堆结构
`heapq.heapify(x)`	将列表 `x` 原地转为小顶堆	直接构造堆（优于逐个 push）

```
max_heap = []
# 插入元素（如5）：存-5
heapq.heappush(max_heap, -5)
# 弹出最大值：-heappop(max_heap) → 5
top = -heapq.heappop(max_heap)
```

滑动窗口最大值：

```
n = len(nums)
if n == 0 or k == 0:
    return []
q = deque()  # 存储索引，对应nums值单调递减
res = []

for i in range(n):
    # 1. 维护单调递减：弹出尾部≤当前值的索引
    while q and nums[i] >= nums[q[-1]]:
        q.pop()
    # 2. 当前索引入队
    q.append(i)
    # 3. 清理队首：索引超出窗口左边界（i-k+1）
    while q[0] < i - k + 1:
        q.popleft()
    # 4. 从第k-1个元素开始，记录窗口最大值
    if i >= k - 1:
        res.append(nums[q[0]])
```

```
    带中转限制的Dijkstra：求src→dst最多k次中转的最小代价
    :param n: 城市数
    :param flights: 航班列表[(u, v, price)]
    :param src: 起点
    :param dst: 终点
    :param k: 最大中转次数（中转k次=最多k+1段航班）
    :return: 最小代价（不可达返回-1）
    """
    # 构建邻接表
    adj = [[] for _ in range(n)]
    for u, v, w in flights:
        adj[u].append((v, w))

    INF = float('inf')
    # 二维距离数组：dist[steps][node] = 中转steps次到node的最小代价
    dist = [[INF]*n for _ in range(k+2)]
    dist[0][src] = 0
    heap = []
    heapq.heappush(heap, (0, src, 0))  # (当前代价, 节点, 中转次数
    while heap:
        curr_cost, u, steps = heapq.heappop(heap)
        if u == dst:
            return curr_cost  # 提前终止
        if steps > k:
            continue  # 超过中转限制，跳过
        if curr_cost > dist[steps][u]:
            continue  # 剪枝：当前代价已非最优
        # 遍历邻接节点
        for v, w in adj[u]:
            new_steps = steps + 1
            new_cost = curr_cost + w
            if new_cost < dist[new_steps][v]:
                dist[new_steps][v] = new_cost
                heapq.heappush(heap, (new_cost, v, new_steps))
    return -1

# 示例调用
flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]]
print(dijkstra_limit(4, flights, 0, 3, 1))  # 700
```

kadane 算法：

```
def maxSubArray(nums):
    cur = ans = nums[0]
    for x in nums[1:]:
        cur = max(x, cur + x)
        ans = max(ans, cur)
    return ans
```

思路拆解

1. 固定上边界 top
2. 向下枚举 bottom
3. 每一列累加 → 一维数组
4. 对该数组跑 Kadane

代码（理解级，不要求手写）

```
def maxSumSubmatrix(matrix):
    n, m = len(matrix), len(matrix[0])
    ans = float('-inf')

    for top in range(n):
        colSum = [0]*m
        for bottom in range(top, n):
            for j in range(m):
                colSum[j] += matrix[bottom][j]

            cur = colSum[0]
            best = colSum[0]
            for x in colSum[1:]:
                cur = max(x, cur+x)
                best = max(best, cur)

            ans = max(ans, best)

    return ans
```

公式 3：简化递推式（高效递推，强烈推荐）

$$C_0 = 1, \quad C_n = \frac{2(2n-1)}{n+1} \cdot C_{n-1} \quad (n \geq 1)$$

ASCII：

- 关键规律：
  - 数字 0-9：48~57（`ord('0')=48`）；
  - 大写 A-Z：65~90（`ord('A')=65`）；
  - 小写 a-z：97~122（`ord('a')=97`）；
  - 大小写字母 ASCII 差：32（`ord('a') - ord('A') = 32`）。

`ord(c)`	单个字符 → 对应 ASCII/Unicode 码
`chr(n)`	ASCII/Unicode 码 → 对应字符

匿名函数排序：

```python
# 数字升序
nums = [3,1,4,2]
sorted_nums = sorted(nums, key=lambda x: x)  # [1,2,3,4]

# 字符串按长度升序
strs = ["apple", "cat", "banana"]
sorted_strs = sorted(strs, key=lambda x: len(x))  # ["cat", "apple", "banana"]

# 字符串按ASCII码升序（默认）/降序
chars = ["B", "a", "C"]
sorted_chars = sorted(chars, key=lambda x: ord(x))  # ["B"(66), "C"(67), "a"(97)]
sorted_chars_desc = sorted(chars, key=lambda x: ord(x), reverse=True)  # 降序
```

```python
d = {"b": 2, "a": 1, "c": 3}
# 按键升序转字典                    : key=lambda x: x[1]（x[1]是值）
sorted_dict_by_key = dict(sorted(d.items(), key=lambda x: x[0]))
print(sorted_dict_by_key)  # {'a': 1, 'b': 2, 'c': 3}
```

```python
# 方式1：仅排序键（返回键的列表）
sorted_keys_asc = sorted(d)  # ['a', 'b', 'c']
```

二维前缀和

```python
for i in range(1, m+1):
    row_sum = 0  # 可选优化：逐行累加，减少重复计算（非必需）
    for j in range(1, n+1):
        pre_sum[i][j] = pre_sum[i-1][j] + pre_sum[i][j-1] - pre_sum[i-1][j-1] + grid[i-1][j-1]
```

石子合并：

```python
n = len(stones)
if n <= 1:
    return 0, 0

# 1. 预处理前缀和（sum[i][j] = pre[j+1] - pre[i]）
pre_sum = [0] * (n + 1)
for i in range(n):
    pre_sum[i+1] = pre_sum[i] + stones[i]

# 2. 初始化DP数组（n x n）
dp_min = [[float('inf')] * n for _ in range(n)]
dp_max = [[0] * n for _ in range(n)]

# 单个石子堆，代价为0
for i in range(n):
    dp_min[i][i] = 0
    dp_max[i][i] = 0

# 3. 按区间长度从小到大遍历（len=2到n）
for length in range(2, n+1):  # length：当前合并的区间长度
    for i in range(n - length + 1):  # i：区间起点
        j = i + length - 1  # j：区间终点
        # 枚举分割点k
        for k in range(i, j):
            # 计算区间[i,j]的石子总数
            sum_ij = pre_sum[j+1] - pre_sum[i]
            # 更新最小/最大代价
            dp_min[i][j] = min(dp_min[i][j], dp_min[i][k] + dp_min[k+1][j] + sum_ij)
            dp_max[i][j] = max(dp_max[i][j], dp_max[i][k] + dp_max[k+1][j] + sum_ij)

# 最终结果：合并[0, n-1]的代价
return dp_min[0][n-1], dp_max[0][n-1]
```

```python
n = len(stones)
if n <= 1:
    return 0, 0

# 1. 破环成链：数组翻倍
new_stones = stones + stones
m = 2 * n

# 2. 预处理前缀和
pre_sum = [0] * (m + 1)
for i in range(m):
    pre_sum[i+1] = pre_sum[i] + new_stones[i]

# 3. 初始化DP数组（m x m）
dp_min = [[float('inf')] * m for _ in range(m)]
dp_max = [[0] * m for _ in range(m)]

# 单个石子堆代价为0
for i in range(m):
    dp_min[i][i] = 0
    dp_max[i][i] = 0

# 4. 区间DP：按长度从小到大遍历
for length in range(2, n+1):  # 只需要合并长度为n的区间（环形的本质）
    for i in range(m - length + 1):
        j = i + length - 1
        for k in range(i, j):
            sum_ij = pre_sum[j+1] - pre_sum[i]
            dp_min[i][j] = min(dp_min[i][j], dp_min[i][k] + dp_min[k+1][j] + sum_ij)
            dp_max[i][j] = max(dp_max[i][j], dp_max[i][k] + dp_max[k+1][j] + sum_ij)

# 5. 遍历所有长度为n的区间，找最小/最大值
min_cost = float('inf')
max_cost = 0
for i in range(n):
    min_cost = min(min_cost, dp_min[i][i + n - 1])
    max_cost = max(max_cost, dp_max[i][i + n - 1])
```

```python
def multiple_knapsack(w, v, s, C):
    """
    多重背包（二进制拆分优化）
    :param w: 物品重量列表
    :param v: 物品价值列表
    :param s: 物品数量限制列表
    :param C: 背包容量
    :return: 最大价值
    """
    # 二进制拆分：将多重背包转为01背包
    new_w = []
    new_v = []
    for i in range(len(w)):
        cnt = s[i]  # 当前物品的数量
        k = 1       # 二进制拆分的基数（1,2,4,8...）
        while cnt > k:
            new_w.append(w[i] * k)
            new_v.append(v[i] * k)
            cnt -= k
            k *= 2
        # 剩余部分
        if cnt > 0:
            new_w.append(w[i] * cnt)
            new_v.append(v[i] * cnt)

    # 按01背包求解
    dp = [0] * (C + 1)
    for i in range(len(new_w)):
        for j in range(C, new_w[i]-1, -1):
            dp[j] = max(dp[j], dp[j - new_w[i]] + new_v[i])
    return dp[C]
```

```python
def two_dim_knapsack(w, v, val, Cw, Cv):
    """
    二维费用01背包
    :param w: 重量列表
    :param v: 体积列表
    :param val: 价值列表
    :param Cw: 最大重量
    :param Cv: 最大体积
    :return: 最大价值
    """
    # 初始化二维dp数组：dp[重量][体积]
    dp = [[0]*(Cv+1) for _ in range(Cw+1)]
    n = len(w)
    for k in range(n):  # 遍历每个物品
        # 逆序遍历重量和体积（01背包，避免重复选）
        for i in range(Cw, w[k]-1, -1):
            for j in range(Cv, v[k]-1, -1):
                dp[i][j] = max(dp[i][j], dp[i - w[k]][j - v[k]] + val[k])
    return dp[Cw][Cv]

# 测试：物品1(w=2,v=3,val=4)、物品2(w=3,v=2,val=5)，Cw=5,Cv=5
print(two_dim_knapsack([2,3], [3,2], [4,5], 5, 5))  # 输出：9（两个物品都选，重量5，体积5，价值9）
```

```python
def knapsack_count(w, v, C):
    """01背包求最大价值的方案数"""
    dp = [0]*(C+1)
    cnt = [0]*(C+1)
    cnt[0] = 1  # 初始方案数
    MOD = 10**9+7  # 题目常要求取模

    for i in range(len(w)):
        for j in range(C, w[i]-1, -1):
            if dp[j - w[i]] + v[i] > dp[j]:
                dp[j] = dp[j - w[i]] + v[i]
                cnt[j] = cnt[j - w[i]] % MOD  # 重置方案数
            elif dp[j - w[i]] + v[i] == dp[j]:
                cnt[j] = (cnt[j] + cnt[j - w[i]]) % MOD  # 累加方案数
    return dp[C], cnt[C]

# 测试：物品[w=1,v=2],[w=2,v=3]，容量3 → 最大价值5（1+2），方案数1
print(knapsack_count([1,2], [2,3], 3))  # 输出：(5, 1)
```

```python
分组背包
:param groups: 分组列表，每个元素是(重量列表，价值列表)，如[(w1,v1), (w2,v2)]
:param C: 背包容量
:return: 最大价值
"""
dp = [0] * (C + 1)
# 遍历每组
for w_list, v_list in groups:
    # 逆序遍历容量（同01背包，避免组内物品重复选）
    for j in range(C, -1, -1):
        # 遍历组内每个物品
        for i in range(len(w_list)):
            w = w_list[i]
            v = v_list[i]
            if j >= w:
                dp[j] = max(dp[j], dp[j - w] + v)
return dp[C]

# 测试：组1[(2,3),(3,4)]，组2[(1,2),(4,5)]，容量5
groups = [([2,3], [3,4]), ([1,2], [4,5])]
print(group_knapsack(groups, 5))  # 输出：7（组1选B(3,4)+组2选C(1,2)，总重量4≤5，价值6？或组2选D(4,5)+组1选A(2
```

```python
import heapq
list.sort(key=lambda x: x[0])
min_heap = [list[0][1]]
for i in range(1, n):
    if list[i][0] > min_heap[0]:
        heapq.heappop(min_heap)
    heapq.heappush(min_heap, list[i][1])
num=len(min_heap)
```

```python
def mixed_knapsack(w, v, s, C):
    """
    混合背包
    :param w: 重量列表
    :param v: 价值列表
    :param s: 数量列表（01背包s=1，完全背包s=-1，多重背包s>1）
    :param C: 容量
    :return: 最大价值
    """
    dp = [0] * (C + 1)
    n = len(w)
    for i in range(n):
        if s[i] == 1:  # 01背包物品
            for j in range(C, w[i]-1, -1):
                dp[j] = max(dp[j], dp[j - w[i]] + v[i])
        ] == -1:  # 完全背包物品
        j in range(w[i], C + 1):
        dp[j] = max(dp[j], dp[j - w[i]] + v[i])
        多重背包物品，二进制拆分
        = s[i]
        1
        e cnt > k:
        # 拆分后的物品按01处理
        for j in range(C, w[i]*k -1, -1):
            dp[j] = max(dp[j], dp[j - w[i]*k] + v[i]*k)
        cnt -= k
        k *= 2
        nt > 0:
        for j in range(C, w[i]*cnt -1, -1):
            dp[j] = max(dp[j], dp[j - w[i]*cnt] + v[i]*cnt)
```

```python
def knapsack_path(w, v, C):
    """01背包求最大价值的具体方案"""
    n = len(w)
    dp = [[0]*(C+1) for _ in range(n+1)]
    # 常规01背包DP
    for i in range(1, n+1):
        for j in range(1, C+1):
            if j < w[i-1]:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i-1]] + v[i-1])

    # 回溯找方案
    path = []
    j = C
    for i in range(n, 0, -1):
        # 若选第i个物品（i-1索引）能达到当前价值，则选
        if j >= w[i-1] and dp[i][j] == dp[i-1][j - w[i-1]] + v[i-1]:
            path.append(i-1)  # 记录物品索引
            j -= w[i-1]
    path.reverse()  # 逆序转为正序
    return dp[n][C], path

# 测试：物品[w=2,v=3],[w=3,v=4],[w=4,v=5]，容量5 → 最大价值7（选0和1号物品）
max_val, path = knapsack_path([2,3,4], [3,4,5], 5)
print(f"最大价值：{max_val}，选中物品索引：{path}")  # 输出：7，[0,1]
```

dp[i][j]：考虑前 i 种物品，背包容量为 j 时，能获得的最大价值。

- `lst.copy()`：返回列表的一个浅拷贝。
- `lst[:]`：使用切片操作创建列表的浅拷贝。

字典

创建字典

- `{}`：创建一个空字典。
- `{key1: value1, key2: value2, ...}`：使用键值对创建字典。
- `dict()`：创建一个空字典或从可迭代对象创建字典。
- `dict.fromkeys(iterable[, value])`：使用可迭代对象中的元素作为键，所有键的值为指定值（默认为 `None`）。

添加和修改项

- `d[key] = value`：添加或更新指定键的值。
- `d.update([other_dict | iterable_of_pairs])`：更新字典，可以是另一个字典或键值对的可迭代对象。

访问项

- `d[key]`：获取指定键对应的值，如果键不存在则抛出 `KeyError`。
- `d.get(key[, default])`：获取指定键对应的值，如果键不存在则返回默认值（默认为 `None`）。
- `d.setdefault(key[, default])`：如果键存在返回其值；如果不存在插入键并设置为默认值（默认为 `None`），然后返回该值。

删除项

- `del d[key]`：删除指定键的项，如果键不存在则抛出 `KeyError`。
- `d.pop(key[, default])`：移除并返回指定键的值，如果键不存在则返回默认值（或抛出 `KeyError`）。
- `d.popitem()`：移除并返回一个任意的 `(key, value)` 对，字典为空时抛出 `KeyE`
- `d.clear()`：清空字典中的所有项。

遍历字典

- `for key in d`：遍历字典的键。
- `for value in d.values()`：遍历字典的值。
- `for key, value in d.items()`：遍历字典的键值对。

检查成员关系

- `key in d`：检查键是否存在于字典中。
- `key not in d`：检查键是否不存在于字典中。

获取字典信息

- `len(d)`：获取字典中键值对的数量。
- `list(d.keys())`：返回包含所有键的列表。
- `list(d.values())`：返回包含所有值的列表。
- `list(d.items())`：返回包含所有键值对的列表（每个键值对是一个元组）

复制字典

- `d.copy()`：返回字典的一个浅拷贝。
- `dict(d)`：使用构造函数创建字典的浅拷贝。

```python
import sys

lines = []
# 遍历sys.stdin逐行读取（自动到EOF终止）
for line in sys.stdin:
    stripped_line = line.strip()
    if stripped_line:  # 过滤空行（可选）
        lines.append(stripped_line)

# 处理示例：转为整数列表（算法题常用）
nums = [int(x) for x in lines]
print("转换后的整数列表：", nums)
```

```python
my_list = ['apple', 'banana', 'cherry', 'banana']
indices = [i for i, x in enumerate(my_list) if x == 'banana']
print(indices)  # 输出将是 [1, 3]，因为 'banana' 在索引1和3处。
```

数值操作

- `math.ceil(x)`：返回不小于 x 的最小整数。
- `math.floor(x)`：返回不大于 x 的最大整数。
- `math.trunc(x)`：截断 x 的小数部分，返回整数部分。
- `math.fabs(x)`：返回 x 的绝对值。
- `math.copysign(x, y)`：返回与 y 同号的 x。
- `math.gcd(a, b)`：返回 a 和 b 的最大公约数。
- `math.lcm(a, b)`：返回 a 和 b 的最小公倍数（Python 3.9+）。
- `math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`：判断两个浮点数是否接近

创建和转换

字符串

- `str(object)`：将对象转换为字符串。
- `repr(object)`：获取对象的可打印表示形式（通常用于调试）。

常用方法

- `string.upper()`：返回字符串的大写版本。
- `string.lower()`：返回字符串的小写版本。
- `string.capitalize()`：返回首字母大写的字符串。
- `string.title()`：返回每个单词首字母大写的字符串。
- `string.swapcase()`：大小写互换。

查找和替换

- `string.find(sub[, start[, end]])`：返回子串 `sub` 第一次出现的位置，找不到返回
- `string.index(sub[, start[, end]])`：类似于 `find()`，但找不到时抛出异常。
- `string.replace(old, new[, count])`：替换 `old` 为 `new`，最多替换 `count` 次。
- `string.count(sub[, start[, end]])`：统计子串 `sub` 出现的次数。

分割和连接

- `string.split([sep[, maxsplit]])`：使用 `sep` 分割字符串，默认按任意空白字符分
- `string.rsplit([sep[, maxsplit]])`：从右开始分割。
- `string.join(iterable)`：使用字符串作为分隔符，连接序列中的元素。

检查字符串内容

- `string.startswith(prefix[, start[, end]])`：检查字符串是否以指定前缀
- `string.endswith(suffix[, start[, end]])`：检查字符串是否以指定后缀结
- `string.isalpha()`：是否全是字母。
- `string.isdigit()`：是否全是数字。
- `string.isalnum()`：是否全是字母或数字。
- `string.isspace()`：是否全是空白字符。
- `string.islower()`：是否全为小写字母。
- `string.isupper()`：是否全为大写字母。
- `string.istitle()`：是否为标题格式（每个单词首字母大写）。

- `len(string)`：获取字符串长度。
- `string.partition(sep)`：将字符串分为三部分：`sep` 左边、`sep` 本身、`sep`
- `string.rpartition(sep)`：从右边开始查找 `sep` 并分割。

集合

添加和移除元素

- `s.add(element)`：添加一个元素到集合中。
- `s.update(iterable)`：添加多个元素到集合中。
- `s.remove(element)`：移除指定元素，如果元素不存在则抛出 `KeyError`。
- `s.discard(element)`：移除指定元素，如果元素不存在则不报错。
- `s.pop()`：移除并返回集合中的任意一个元素，集合为空时抛出 `KeyError`
- `s.clear()`：清空集合中的所有元素。

集合操作

- `s.union(t)` 或 `s | t`：返回两个集合的并集。
- `s.intersection(t)` 或 `s & t`：返回两个集合的交集。
- `s.difference(t)` 或 `s - t`：返回在 `s` 中但不在 `t` 中的元素。
- `s.symmetric_difference(t)` 或 `s ^ t`：返回在 `s` 或 `t` 中，但不同时在
- `s.copy()`：返回集合的一个浅拷贝。

检查成员关系

- `element in s`：检查元素是否存在于集合中。
- `element not in s`：检查元素是否不存在于集合中。

子集和超集检查

- `s.issubset(t)` 或 `s <= t`：检查 `s` 是否是 `t` 的子集。
- `s.issuperset(t)` 或 `s >= t`：检查 `s` 是否是 `t` 的超集。
- `s.isdisjoint(t)`：检查 `s` 和 `t` 是否没有交集。

获取集合信息

- `len(s)`：获取集合中元素的数量。
- `min(s)`：获取集合中的最小元素。
- `max(s)`：获取集合中的最大元素。
- `sum(s)`：计算集合中所有元素的总和（仅适用于数值类型）。

创建列表

列表

- `[]`：创建一个空列表。
- `[element1, element2, ...]`：使用方括号创建一个包含指定元素的列表。
- `list(iterable)`：从可迭代对象（如字符串、元组等）创建列表。

添加和移除元素

- `lst.append(element)`：在列表末尾添加一个元素。
- `lst.extend(iterable)`：将可迭代对象中的所有元素添加到列表末尾。
- `lst.insert(index, element)`：在指定位置插入一个元素。
- `lst.remove(element)`：移除第一个匹配的元素，如果元素不存在则抛出 `ValueE`
- `lst.pop([index])`：移除并返回指定位置的元素，默认移除最后一个元素。
- `lst.clear()`：清空列表中的所有元素。

访问和修改元素

- `lst[index]`：访问或修改指定索引处的元素。
- `lst[start:end:step]`：切片访问，获取子列表。
- `lst.index(element[, start[, end]])`：返回第一个匹配元素的索引，找不到时
- `lst.count(element)`：统计元素出现的次数。
- `lst.reverse()`：就地反转列表。
- `lst.sort(key=None, reverse=False)`：就地对列表进行排序。
- `sorted(lst, key=None, reverse=False)`：返回排序后 ↓ 下列表，不改变原列表。