

Python基础语法

- 1.列表
- 2.Deque
- 3.字典
 - 字典推导器
- 4.map映射函数
5. 自定义Set规则
6. 技巧
- 7.优先队列 / 堆
8. 有序列表 / 有序集合

字符串

- 字符串排序
- Z函数 (扩展KMP)
- 判断子序列
- 字符串API

合并区间

回溯/递归

离散化

二分查找

- 1.多维二分
 - 2.二分答案
 3. 朴素二分
- 自定义比较规则

单调结构 / 滑动窗口

- 滑动窗口
- 单调栈
- 单调栈优化dp
- 单调队列
- 单调队列优化dp

前缀/差分

- 1.二维差分
- 2.二维前缀

数学

取整函数性质

- (1). 上下取整转换
- (2). 取余性质
- (3). 幂等律

素数

- (1). 埃氏筛
- (2). 欧拉筛 / 线性筛
- (3). 分解质因子
- (4). 素数计数函数近似值

约数

1. 试除法求所有约数
2. 乘积数的约数个数
3. 乘积数的所有约数之和
4. 欧几里得算法

欧拉函数

- 筛法求欧拉函数

容斥原理

数学公式

- 排序不等式
- 区间递增k个数
- 平均数不等式
- 求和公式
- 取模性质

组合数学

- 二项式定理
- 卡特兰数

矩阵乘法/矩阵快速幂/快速幂

高等数学

- (1). 调和级数

数据结构

并查集

字典树

- 26叉字典树
- 哈希字典树

线段树

- 动态开点 + lazy 线段树

线段树优化DP问题

递归动态开点 (无lazy) 线段树

树状数组

离散化树状数组 + 还原

ST表 / 可重复贡献问题

图论

建图

Floyd

Dijkstra

1. 朴素Dijkstra

2. 堆优化Dijkstra

3. 堆优化Dijkstra (字典写法)

4. 最短路与子序列 和/积 问题

5. 动态修改边权

最小生成树

Prim

二分图

连通块问题

树论

倍增LCA

树上差分

树形DP(换根DP)

树上异或

树上直径

位运算

位运算与集合

拆位试填法

动态规划

背包问题

0 - 1 背包

完全背包

多重背包

分组背包

线性dp

最长上升子序列

最长公共子序列

编辑距离

区间dp

最长回文子序列

最长回文子串

数位dp

状态机dp

状压dp / 状态压缩dp

贪心

多维贪心 + 排序

反悔贪心

1. 反悔堆

2. 尝试反悔 + 反悔栈

贡献法

计算几何

旋转与向量

距离

曼哈顿距离转切比雪夫

切比雪夫转曼哈顿距离

Python基础语法

1.列表

int 转 list

```
num = 123
nums = list(map(int, str(num)))
```

list(int) 转 int

```
nums = [1, 2, 3]
num = int(''.join(map(str, nums)))

def lst_int(nums):
    return int(''.join(map(str, nums)))
```

列表特性

比较大小的时候，不管长度如何，依次比较到第一个元素不相等的位置

比如[1, 2, 3] < [2, 3] 因为在比较1 < 2的时候就终止。

嵌套列表推导：展平二维数组

```
nums = [e for row in matrix for e in row]
```

2.Deque

```
from collections import deque
list1 = [0, 1, 2, 3]
q=deque(list1)
q.append(4)      # 向右侧加
q.appendleft(-1) # 向左侧加
q.extend(可迭代元素) # 向右侧添加可迭代元素
q.extendleft(可迭代元素)
q=q.pop()      # 移除最右端并返回元素值
l=q.popleft()   # 移除最左端
q.count(1)      # 统计元素个数    1
```

```
# 返回string指定范围中str首次出现的位置
string.index(str, beg=0, end=len(string))
string.index(" ")
list(map(s.index,s)) # 返回字符索引数组，如"abcba" -> [0,1,2,1,0]
```

3.字典

```
d.pop(key) # 返回key对应的value值，并在字典中删除这个键值对
d.get(key,default_value) # 获取key对应的值，如果不存在返回default_value
d.keys()   # 键构成的可迭代对象
d.values() # 值构成的可迭代对象
d.items()  # 键值对构成的可迭代对象
d = defaultdict(list) # 指定了具有默认值空列表的字典
```

字典推导器

字母表对应下标

```
dic = {chr(i) : i - ord('a') + 1 for i in range(ord('a'), ord('z') + 1)}
```

也可以使用zip初始化dic

[2606. 找到最大开销的子字符串 - 力扣 \(LeetCode\)](#)

```
dic = dict(zip(chars, vals))
for x in s:
    y = dic.get(x, ord(x) - ord('a') + 1)
```

4.map映射函数

用法:

```
map(function, iterable, ...)
```

```
def square(x) :           # 计算平方数
    return x ** 2

map(square, [1,2,3,4,5])  # 计算列表各个元素的平方
# [1, 4, 9, 16, 25]

map(lambda x: x ** 2, [1, 2, 3, 4, 5]) # 使用 lambda 匿名函数
# [1, 4, 9, 16, 25]

# 提供了两个列表，对相同位置的列表数据进行相加
map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
# [3, 7, 11, 15, 19]
```

5. 自定义Set规则

```
class MySet(set):
    def add(self, element):
        sorted_element = tuple(sorted(element))
        if not any(sorted_element == e for e in self):
            super().add(sorted_element)
```

```
s = MySet()
s.add((2, 1, 1))
s.add((1, 2, 1))
print(s) # 输出: {(1, 1, 2)}
```

6. 技巧

快读快写

```
import sys
sys.setrecursionlimit(1000000)
input=lambda:sys.stdin.readline().strip()
write=lambda x:sys.stdout.write(str(x)+'\n')
```

7. 优先队列 / 堆

```
from heapq import heapify, heappop, heappush
heapify(nums)
score = heappop(nums)
heappush(nums, val)

# 注意:
# python中堆默认且只能是小顶堆
```

```
nums = []
heapq.heappush(nums, val) # 插入
heapq.heappop(nums)      # 弹出顶部
```

8. 有序列表 / 有序集合

SortedList

SortedList 相当于 multiset

添加元素: $O(\log n)$; `s.add(val)`

添加一组可迭代元素: $O(k \log n)$; `s.upadte(*iterable*)`

查找元素: $O(\log n)$; `s.count(val)`, 返回元素的个数

字符串

字符串排序

```
sorted(str) #返回按照字典序排序后的列表，如"eda"->['a','d','e']
s_sorted=''.join(sorted(str)) #把字符串列表组合成一个完整的字符串
```

Z函数 (扩展KMP)

对于字符串 s ，函数 $z[i]$ 表示 s 和 $s[i:]$ 的最长公共前缀(LCP)的长度。特别的，定义 $z[0] = 0$ 。即

$$z[i] = \text{len}(LCP(s, s[i:]))$$

例如， $z(abacaba) = [0, 0, 1, 0, 3, 0, 1]$

可视化: [Z Algorithm \(JavaScript Demo\) \(utdallas.edu\)](#)

```
# s = 'aabcaabxaaaz'
n = len(s)
z = [0] * n
l = r = 0
for i in range(1, n):
    if i <= r: # 在Z-box范围内
        z[i] = min(z[i - l], r - i + 1)
    while i + z[i] < n and s[z[i]] == s[i + z[i]]:
        l, r = i, i + z[i]
        z[i] += 1
# print(z) # [0, 1, 0, 0, 3, 1, 0, 0, 2, 2, 1, 0]
```

判断子序列

判断 p 在删除 ss 中下标元素后，是否仍然满足 s 是 p 的子序列。

例如：
 $s = \text{"abcacb"} , p = \text{"ab"} , \text{removable}[:2] = [3, 1]$
解释：在移除下标 3 和 1 对应的字符后， "abcacb" 变成 "accb" 。
 "ab" 是 "accb" 的一个子序列。

```
ss = set(removable[:x])
i = j = 0
n, m = len(s), len(p)
while i < n and j < m:
    if i not in ss and s[i] == p[j]:
        j += 1
    i += 1
return j == m
```

字符串API

- $s1.\text{startswith}(s2, \text{beg} = 0, \text{end} = \text{len}(s2))$
用于检查字符串 $s1$ 是否以字符串 $s2$ 开头。是则返回True。如果指定 beg 和 end ，则在 $s1[\text{beg}:\text{end}]$ 范围内查找。
- 使用 `ascii_lowercase`遍历26个字母。

```
from string import ascii_lowercase
cnt = {ch: 0 for ch in ascii_lowercase}
```

合并区间

先排序。

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort()
        res = []
        l, r = intervals[0][0], intervals[0][1]
        for interval in intervals:
            il, ir = interval[0], interval[1]
            if il > r:
                res.append([l, r])
                l = il
            r = max(r, ir)
        res.append([l, r])
        return res
```

[2580. 统计将重叠区间合并成组的方案数 - 力扣 \(LeetCode\)](#)

```
def countways(self, ranges: List[List[int]]) -> int:
    ranges.sort(key = lambda x: x[0])
    l, r = ranges[0][0], ranges[0][1]
    nranges = []
    for il, ir in ranges:
        if il > r:
            nranges.append([l, r])
            l = il
        r = max(ir, r)
```

回溯/递归

套路：

1. 当前子问题？
2. 当前操作？
3. 下一个子问题？

[LCR 086. 分割回文串 - 力扣 \(LeetCode\)](#)

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        res = []
        cur = []
        n = len(s)
        # 枚举当前位置
        def dfs(i):
            # 当前子问题：从下标 >= i 中构造分割串
            # 当前操作：遍历 j \in [i + 1, n], 枚举s[i: j] 是否是回文串
            # 下一个子问题：从下标 >= j 中构成回文子串
            if i == n:
                res.append(cur.copy())
                return
            for j in range(i + 1, n + 1):
                t = s[i: j]
                if t == t[::-1]:
                    cur.append(s[i: j])
                    dfs(j)
                    cur.pop()
        dfs(0)
        return res
```

离散化

二分写法

```
sorted_nums = sorted(set(nums))
nums = [bisect.bisect_left(sorted_nums, x) + 1 for x in nums]
```

字典写法

```
sorted_nums = sorted(set(nums))
mp = {x: i + 1 for i, x in enumerate(sorted_nums)}
nums = [mp[x] for x in nums]
```

二分 + 还原

```
tmp = nums.copy()
sorted_nums = sorted(set(nums))
nums = [bisect.bisect_left(sorted_nums, x) + 1 for x in nums]
mp_rev = {i: x for i, x in zip(nums, tmp)}
```

二分查找

```
from bisect import *
l = [1,1,1,3,3,3,4,4,4,5,5,5,8,9,10,10]
print(len(l)) # 16

print(bisect(l, 10)) # 相当于upper_bound, 16
print(bisect_right(l, 10))

print(bisect_left(l, 10)) # 14
```

1. 多维二分

```
a = [(1, 20), (2, 19), (4, 15), (7, 12)]
idx = bisect_left(a, (2, ))
```

2. 二分答案

正难则反思想，二分答案一般满足两个条件：

- 当发现问需要的最少/最多时间时
- 答案具有单调性。例如问最少的时候，你发现取值越大越容易满足条件。

check(x) 函数对单调x 进行检验。

```
y = 27
def check(x):
    if x > y:
        return True
    return False
left = a
res = left + bisect.bisect_left(range(left, mx), True, key = check))
```

[3048. 标记所有下标的最早秒数 I - 力扣 \(LeetCode\)](#)

求“至少”问题

```
n, m = len(nums), len(changeIndices)
def check(mx): # 给mx天是否能顺利考完试
    last_day = [-1] * n
    for i, x in enumerate(changeIndices[:mx]):
        last_day[x - 1] = i + 1
    #如果给mx不能完成，等价于有i遍历到考试日期的考试
    if -1 in last_day:
        return False
    less_day = 0
    for i, x in enumerate(changeIndices[:mx]):
        if last_day[x - 1] == i + 1: # 到了考试日期
            if less_day >= nums[x - 1]:
                less_day -= nums[x - 1]
                less_day -= 1 #抵消当天不能复习
            else:
                return False #寄了
        less_day += 1
```

```

    return True
left = sum(nums) + n # 至少需要的天数，也是二分的左边界
res = left + bisect.bisect_left(range(left, m + 1), True, key = check)
return -1 if res > m else res

```

求“最多”问题

[1642. 可以到达的最远建筑 - 力扣 \(LeetCode\)](#)

```

def furthestBuilding(self, heights: List[int], bricks: int, ladders: int) -> int:
    n = len(heights)
    d = [max(0, heights[i + 1] - heights[i]) for i in range(n - 1)]
    def check(x):
        t = d[:x]
        t.sort(reverse = True)
        return not (ladders >= x or sum(t[ladders: ]) <= bricks)
    return bisect.bisect_left(range(n), True, key = check) - 1

```

3. 朴素二分

在 闭区间[a, b]上二分

```

lo, hi = a, b # [a, b]
while lo < hi:
    mid = (lo + hi) // 2
    if check(mid):
        hi = mid
    else:
        lo = mid + 1
return lo

```

自定义比较规则

```

class node():
    def __init__(self, need, get, idx):
        self.need = need
        self.get = get
        self.idx = idx
    def __lt__(self, other):
        return self.need < other.need

```

单调结构 / 滑动窗口

滑动窗口

[2009. 使数组连续的最少操作数 - 力扣 \(LeetCode\)](#)

定长滑动窗口 + 正难则反：需要操作最少次数 = n - 能够不操作的最多的数字。这些数字显然是不重复的，所以首先去重。对于去重完的元素，每一个左边界 $nums[left]$ ，在去重数组中， $[nums[left], nums[left] + n - 1]$ 区间在数组中出现的次数即为当前可以保留的数字的个数。

```

def minOperations(self, nums: List[int]) -> int:
    n = len(nums)
    nums = sorted(set(nums))
    res = left = 0
    for i, x in enumerate(nums):
        while x > nums[left] + n - 1:
            left += 1
        res = max(res, i - left + 1)
    return n - res

```

单调栈


```
def trap(self, height: List[int]) -> int:
    # 单调栈: 递减栈
    stk, n, res = deque(), len(height), 0
    for i in range(n):
        # 1. 单调栈不为空、且违反单调性
        while stk and height[i] > height[stk[-1]]:
            # 2. 出栈
            top = stk.pop()
            # 3. 特判
            if not stk:
                break
            # 4. 获得左边界、宽度
            left = stk[-1]
            width = i - left - 1
            # 5. 计算
            res += (min(height[left], height[i]) - height[top]) * width
        # 6. 入栈
        stk.append(i)
    return res
```

[84. 柱状图中最大的矩形 - 力扣 \(LeetCode\)](#)

矩形面积求解: 维护单调增栈, 同时首尾插入哨兵节点。

```
def largestRectangleArea(self, heights: List[int]) -> int:
    heights.append(-1)
    stk = [-1]
    res = 0
    for i, h in enumerate(heights):
        while len(stk) > 1 and h < heights[stk[-1]]:
            cur = stk.pop()
            l = stk[-1]
            width = i - l - 1
            s = width * heights[cur]
            res = max(res, s)
        stk.append(i)
    return res
```

[1793. 好子数组的最大分数 - 力扣 \(LeetCode\)](#)

矩形面积求解问题变形: 求 $\min(nums[i], \dots, nums[j]) \times (j - i + 1)$, 并对 i, j 做了范围约束。

```
def maximumScore(self, nums: List[int], k: int) -> int:
    stk = [-1]
    nums.append(-1)
    res = 0
    for i, h in enumerate(nums):
        while len(stk) > 1 and h < nums[stk[-1]]:
            cur = stk.pop()
            l = stk[-1]
            if not (l + 1 <= k and i - 1 >= k): continue # 约束范围
            width = i - l - 1
            res = max(res, width * nums[cur])
        stk.append(i)
    return res
```

单调栈优化dp

[2617. 网格图中最少访问的格子数 - 力扣 \(LeetCode\)](#)

暴力dp转移做法

```

class Solution:
    def minimumVisitedCells(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        f = [[inf] * n for _ in range(m)]
        f[-1][-1] = 0
        for i in range(m - 1, -1, -1):
            for j in range(n - 1, -1, -1):
                g = grid[i][j]
                for k in range(1, min(g + 1, m - i)):
                    f[i][j] = min(f[i][j], f[i + k][j] + 1)
                for k in range(1, min(g + 1, n - j)):
                    f[i][j] = min(f[i][j], f[i][j + k] + 1)
        return f[0][0] + 1 if f[0][0] != inf else -1

```

单调栈 + 二分 优化dp

倒序枚举 i, j

$$f[i][j] = \min \left\{ \min_{k=j+1}^{j+g} f[i][k], \min_{k=i+1}^{i+g} f[k][j] \right\} + 1$$

可以发现左边界 i 是递减的，右边界 $j + g$ 是不确定的。联想到滑动窗口最值问题，维护一个向左增长的栈，栈元素自左向右递减。

由于栈中元素有序，每次查找只需要二分即可找出最值。

```

def minimumVisitedCells(self, grid: List[List[int]]) -> int:
    m, n = len(grid), len(grid[0])
    stky = [deque() for _ in range(n)] # 列上单调栈
    f = 0 # 行上单调栈
    for i in range(m - 1, -1, -1):
        stkx = deque()
        for j in range(n - 1, -1, -1):
            g, stky = grid[i][j], stky[j]
            f = 1 if i == m - 1 and j == n - 1 else inf
            if g > 0:
                if stkx and j + g >= stkx[0][1]:
                    mnj = bisect_left(stkx, j + g + 1, key = lambda x: x[1]) - 1
                    f = stkx[mnj][0] + 1
                if stky and i + g >= stky[0][1]:
                    mni = bisect_left(stky, i + g + 1, key = lambda x: x[1]) - 1
                    f = min(f, stky[mni][0] + 1)
            if f < inf:
                while stkx and f <= stkx[0][0]:
                    stkx.popleft()
                stkx.appendleft((f, j))
                while stky and f <= stky[0][0]:
                    stky.popleft()
                stky.appendleft((f, i))
    return f if f != inf else -1

```

单调队列

滑窗最大值 ~ 维护递减小队列；滑窗最小值 ~ 维护递增队列

[239. 滑动窗口最大值 - 力扣 \(LeetCode\)](#)

```

def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
    n = len(nums)
    res = []
    q = deque()
    for i, x in enumerate(nums):
        # 1. 入，需要维护单调递减队列的有序性
        while q and x >= nums[q[-1]]:
            q.pop()
        q.append(i)

        # 2. 出，当滑动窗口区间长度大于k的时候，弹出去左端的
        if i - q[0] + 1 > k:
            q.popleft()

```

```

        # 记录元素
        if i >= k - 1:
            res.append(nums[q[0]])
    return res

```

2398. 预算内的最多机器人数目 - 力扣 (LeetCode)

单调队列 + 滑动窗口

```

def maximumRobots(self, chargeTimes: List[int], runningCosts: List[int], budget: int) -> int:
    n = len(chargeTimes)
    res = 0
    s = l = 0 # 滑窗的和 / 窗口左边界
    q = deque() # 单调队列维护最大值
    # 滑动窗口
    for i, x in enumerate(chargeTimes):
        while q and x >= chargeTimes[q[-1]]:
            q.pop()
        q.append(i)
        s += runningCosts[i]
        while i - l + 1 > 0 and s * (i - l + 1) + chargeTimes[q[0]] > budget:
            s -= runningCosts[l]
            l += 1
            if l > q[0]:
                q.popleft()
        res = max(res, i - l + 1)
    return res

```

单调队列优化dp

2944. 购买水果需要的最少金币数 - 力扣 (LeetCode)

暴力做法: $O(n^2)$

```

def minimumCoins(self, prices: List[int]) -> int:
    n = len(prices)
    # f[i] 表示获得 i 及其以后的所有水果的最少开销
    f = [inf] * (n + 1)
    for i in range(n, 0, -1):
        # [i + 1, 2 * i] 免费
        if 2 * i >= n:
            f[i] = prices[i - 1]
        else:
            for j in range(i + 1, 2 * i + 2):
                f[i] = min(f[i], f[j] + prices[i - 1])
    return f[1]

```

注意到 i 递减, 区间 $[i + 1, 2 * i + 1]$ 是一个长度为 $i + 1$ 的滑动窗口, 转移成滑动窗口最值问题。

```

def minimumCoins(self, prices: List[int]) -> int:
    n = len(prices)
    # f[i] 表示获得 i 及其以后的所有水果的最少开销
    f = [inf] * (n + 1)
    q = deque()
    for i in range(n, 0, -1):
        # i 递减, 区间 [i + 1, 2 * i + 1] 是一个定长为 i + 1 的滑动窗口
        while q and q[-1][1] - (i + 1) + 1 > i + 1:
            q.pop()
        if 2 * i >= n:
            f[i] = prices[i - 1]
        else:
            f[i] = q[-1][0] + prices[i - 1]
            while q and f[i] <= q[0][0]:
                q.popleft()
            q.appendleft((f[i], i))
    return f[1]

```

前缀/差分

1.二维差分

```
d = [[0] * (n + 2) for _ in range(m + 2)]
# 对矩阵中执行操作,使得左上角为(i, j),右下角为(x, y)的矩阵都加k,等价于如下操作
d[i + 1][j + 1] += k
d[x + 2][y + 2] += k
d[i + 1][y + 2] -= k
d[x + 2][j + 1] -= k

# 还原差分时,直接原地还原
for i in range(m):
    for j in range(n):
        d[i + 1][j + 1] += d[i][j + 1] + d[i + 1][j] - d[i][j]
```

2.二维前缀

[3070. 元素和小于等于 k 的子矩阵的数目 - 力扣 \(LeetCode\)](#)

```
class PreSum2d:
    # 二维前缀和(支持加法和异或),只能离线使用,用n*m时间预处理,用O1查询子矩阵的和;op=0是加法,op=1是异或
    def __init__(self, g, op=0):
        m, n = len(g), len(g[0])
        self.op = op
        self.p = [[0] * (n + 1) for _ in range(m + 1)]
        if op == 0:
            for i in range(m):
                for j in range(n):
                    p[i + 1][j + 1] = p[i][j + 1] + p[i + 1][j] - p[i][j] + g[i][j]
        elif op == 1:
            for i in range(m):
                for j in range(n):
                    p[i + 1][j + 1] = p[i][j + 1] ^ p[i + 1][j] ^ p[i][j] ^ g[i][j]
    # O(1)时间查询闭区间左上(a,b),右下(c,d)矩形部分的数字和。
    def sum_square(self, a, b, c, d):
        if self.op == 0:
            return self.p[c + 1][d + 1] + self.p[a][b] - self.p[a][d + 1] - self.p[c + 1][b]
        elif self.op == 1:
            return self.p[c + 1][d + 1] ^ self.p[a][b] ^ self.p[a][d + 1] ^ self.p[c + 1][b]

class NumMatrix:
    def __init__(self, mat: List[List[int]]):
        self.pre = PreSum2d(mat)
    def sumRegion(self, row1: int, col1: int, row2: int, col2: int) -> int:
        # pre = self.pre
        return self.pre.sum_square(row1, col1, row2, col2)

class Solution:
    def countSubmatrices(self, grid: List[List[int]], k: int) -> int:
        n = len(grid)
        m = len(grid[0])
        res = 0
        p = NumMatrix(grid)
        for i in range(n):
            for j in range(m):
                if p.sumRegion(0, 0, i, j) <= k:
                    res += 1
        return res
```

`pre[i + 1][j + 1]` 是左上角为(0, 0)右下角为(i, j)的矩阵的元素和。

如果是前缀异或是:

`p[i + 1][j + 1] = p[i][j + 1] ^ p[i + 1][j] ^ p[i][j] ^ g[i][j]`

```
def countSubmatrices(self, grid: List[List[int]], k: int) -> int:
    m, n = len(grid), len(grid[0])
    pre = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m):
        for j in range(n):
            pre[i + 1][j + 1] = pre[i][j + 1] + pre[i + 1][j] - pre[i][j] + grid[i][j]
    res = 0
    for i in range(m):
        for j in range(n):
            if pre[i + 1][j + 1] <= k:
                res += 1
    return res
```

前缀异或 / 自定义前缀操作

```
pre = list(accumulate(nums, xor, initial = 0))
```

数学

取整函数性质

(1). 上下取整转换

$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n-1}{m} \right\rfloor + 1 = \left\lfloor \frac{n+m-1}{m} \right\rfloor$$

(2). 取余性质

$$n \bmod m = n - m \cdot \left\lfloor \frac{n}{m} \right\rfloor$$

(3). 幂等律

$$\begin{aligned} \lfloor \lfloor x \rfloor \rfloor &= \lfloor x \rfloor \\ \lceil \lceil x \rceil \rceil &= \lceil x \rceil \end{aligned}$$

素数

(1). 埃氏筛

时间复杂度: $O(n \log \log n)$

```
primes = []
is_prime = [True] * (n + 1) # MX为最大可能遇到的质数 + 1
is_prime[1] = is_prime[0] = False

for i in range(2, int(math.sqrt(n)) + 1): # i * i <= n
    if is_prime[i]:
        for j in range(i * i, n + 1, i):
            is_prime[j] = False
for i in range(2, n + 1):
    if is_prime[i]: primes.append(i)
```

时间复杂度证明

对于2, 要在数组中筛大约 $\frac{n}{2}$ 个数, 同理对于素数 p , 约要筛去 $\frac{n}{p}$ 个数。

$$\text{故有 } O\left(\sum_{k=1}^{\pi(n)} \frac{n}{p_k}\right) = O\left(n \sum_{k=1}^{\pi(n)} \frac{1}{p_k}\right) = O(n \log \log n) \text{ (Mertens第二定理)}$$

切片优化

```
primes = []
is_prime = [True] * (n + 1)
is_prime[0] = is_prime[1] = False
for i in range(2, int(math.sqrt(n)) + 1):
    if is_prime[i]:
        is_prime[i * i::i] = [False] * ((n - i * i) // i + 1)
for i in range(2, n + 1):
    if is_prime[i]: primes.append(i)
```

(2). 欧拉筛 / 线性筛

基本思想：每一个合数一定存在最小的质因子。确保每一个合数只被他的最小质因子筛去。

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

正确性证明：

1. 每个合数不会被筛超过一次：

枚举 i 从小到大的所有质数，在 $i \% p = 0$ 出现之前， p 一定小于 i 的所有质因子， $p \cdot i$ 的质因子的前缀与 i 的质因子前缀相同，故 p 一定是 $i \cdot p$ 的最小质因子，筛去；在出现 $i \% p = 0$ 时， p 恰好是 i 的最小质因子，同理，然后 break。保证每个合数只会被最小的质因子筛去。

2. 每个合数都会被筛最少一次：

每个合数 x 一定存在最小质因子 p ，和对应的 x/p 。在 i 枚举到 x/p 的时候，一定会筛去 x

由于保证每个合数一定被筛一次，所以是 $O(n)$

(3). 分解质因子

试除法。复杂度不超过 $O(\sqrt{n})$ ，实际上是 $O(\log n) \sim O(\sqrt{n})$

对于一个数 x ，最多有一个大于等于 \sqrt{n} 的质因子。（可以用反证法，证明）

所以只需要进行特判，在遍历完 $[2, \text{int}(\sqrt{n})]$ 区间后，如果 x 比 1 大，则 x 就等于那最后一个质因子。

```
def solve(x):
    for i in range(2, int(math.sqrt(x)) + 1): # i = 2; i * i <= x
        if x % i == 0:
            s = 0
            while x % i == 0:
                s += 1
                x //= i
            print(f'{i} {s}') # i 是质因子，s 表示幂次
    if x > 1:
        print(f'{x} 1')
    print()
```

Oi Wiki 风格：

```
def breakdown(N):
    result = []
    for i in range(2, int(sqrt(N)) + 1):
        if N % i == 0: # 如果 i 能够整除 N，说明 i 为 N 的一个质因子。
            while N % i == 0:
                N //= i
            result.append(i)
    if N != 1: # 说明再经过操作之后 N 留下了一个素数
        result.append(N)
    return result
```

(4). 素数计数函数近似值

小于等于 x 的素数个数记为 $\pi(x)$, $\pi(x)$ 近似于 $\frac{x}{\ln x}$ 。

约数

1. 试除法求所有约数

复杂度为: $O(\sqrt{n})$

```
def solve(x):
    res = []
    for i in range(2, int(math.sqrt(x)) + 1):
        if x % i == 0:
            res.append(i)
            if i != x // i:
                res.append(x // i)
    res.sort()
```

2. 乘积数的约数个数

对于一个以标准分解式给出的数 $N = \prod_{i=1}^k p_i^{\alpha_i}$, 其约数个数为 $\prod_{i=1}^k (\alpha_i + 1)$

例如 $N = 2^5 \cdot 3^1$, 约数个数为 $(5 + 1) \times (1 + 1) = 12$

3. 乘积数的所有约数之和

对于一个以标准分解式给出的数 $N = \prod_{i=1}^k p_i^{\alpha_i}$, 其约数之和为 $\prod_{i=1}^k (\sum_{j=0}^{\alpha_i} p_i^j)$

例如 $N = 2^5 \cdot 3^1$, 约数个数为 $(2^0 + 2^1 + \dots + 2^5) \times (3^0 + 3^1)$ 。展开结果实际上, 各个互不相同, 每一项都是一个约数, 总个数就是约数个数。

4. 欧几里得算法

算法原理: $\gcd(a, b) = \gcd(b, a \bmod b)$

证明:

- 对于任意一个能整除 a 且能整除 b 的数 d , $a \bmod b$ 可以写成 $a - k \cdot b$, 其中 $k = a // b$, 所以 d 一定能够整除 $b, a \bmod b$;
- 对于任意一个能整除 b 且能整除 $a - k \cdot b$ 的数 d , 一定能整除 $a - k \cdot b + k \cdot b = a$, 所以二者的公约数的集合是等价的。
- 所以二者的最大公约数等价

```
def gcd(a, b):
    return gcd(b, a % b) if b else a
```

时间复杂度: $O(\log(\max(a, b)))$

证明:

引理1: $a \bmod b \in [0, b - 1]$ 。例如, $38 \bmod 13 = 12$

引理2: 取模, 余数至少折半。

如果 $b > a // 2$, $a \bmod b = a - b < a // 2$ 。例如, $a = 9, b = 5, a \bmod b = 9 - 5 = 4$

如果 $b \leq a // 2$, $a \bmod b \leq b - 1 \leq a // 2 - 1$ 。

情况1: 当每次执行 gcd 时, 如果 $a < b$, 则交换; 情况2: 否则 $a \geq b$, 一定发生引理2的情况, 即对 a 取模, 一定会让 a 折半。最坏情况下, 每两次让 a 折半, 所以时间复杂度为:

$$O(T) = O(T/2) + 2 = O(T/4) + 4 = O(\frac{T}{2^k}) + k \times 2 = 2 \log k, \text{ 即 } O(\log(\max(a, b)))$$

欧拉函数

定义: $\phi(n)$ 表示 $1 \sim n$ 中与 n 互质 (最大公约数为1) 的数的个数。

时间复杂度: $O(\sqrt{n})$, 同质因数分解。

对于一个以标准分解式给出的数 $N = \prod_{i=1}^k p_i^{\alpha_i}$ ，满足：

$$\phi(N) = N \cdot \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right)$$

证明方法：

- 容斥原理。
- 减去 p_1, p_2, \dots, p_k 的所有倍数的个数，这一步会多筛一些数。例如一个数既是 p_1 ，又是 p_2 的倍数，会删去两次。

$$N - \sum_{i=1}^k \frac{N}{p_i}$$

- 加上所有 $p_i \cdot p_j$ 的倍数

$$N - \sum_{i=1}^k \frac{N}{p_i} + \sum_{i,j \in [0,k] \text{ 且 } i < j} \frac{N}{p_i \cdot p_j}$$

- 减去所有 $p_i \cdot p_j \cdot p_u$ 的倍数，以此类推。

$$N - \sum_{i=1}^k \frac{N}{p_i} + \sum_{i,j \in [0,k] \text{ 且 } i < j} \frac{N}{p_i \cdot p_j} - \sum_{i,j,u \in [0,k] \text{ 且 } i < j < u} \frac{N}{p_i \cdot p_j \cdot p_u} + \dots = N \cdot \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right)$$

最后一步，可以通过观察系数的角度来证明。例如 $\frac{1}{p_i}$ 项的系数是 -1。

证明方法二：

$$\phi(N) = \phi\left(\prod_{i=1}^k p_i^{\alpha_i}\right) = \prod_{i=1}^k \phi(p_i^{\alpha_i}) = \prod_{i=1}^k p_i^{\alpha_i} \left(1 - \frac{1}{p_i}\right) = N \cdot \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right)$$

性质：

- 积性函数：对于互质的 p, q ， $\phi(p \times q) = \phi(p) \times \phi(q)$ 。特别的，对于奇数 p ， $\phi(2p) = \phi(p)$

证明：互质的数，质因子分解的集合无交集。 $\phi(2) = 1$

- 对于质数 p ， $\phi(p^k) = p^k - \frac{p^k}{p} = p^k - p^{k-1}$

证明：减去是 p 的倍数的数，得到不是 p 的倍数的数的个数一定和 p 互质。

```
def solve(n):
    res = n
    for i in range(2, int(sqrt(n)) + 1):
        if n % i == 0:
            res = res * (i - 1) // i
            while n % i == 0:
                n //= i
    if n > 1:
        res = res * (n - 1) // n
    return res
```

筛法求欧拉函数

对于 N 的最小质因子 p_1 ， $N' = \frac{N}{p_1}$ ，我们希望筛法中， N 通过 $N' \cdot p_1$ 筛掉。

考虑两种情况：

- $N' \bmod p_1 = 0$ ，则 N' 包含了 N 的所有质因子。

$$\phi(N) = N \times \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right) = N' \cdot p_1 \times \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right) = p_1 \times \phi(N')$$

- $N' \bmod p_1 \neq 0$ ，则 N' 与 p_1 互质（证明：质数是因子只有1和本身，因此最大公约数是1，互质）。

由欧拉函数的积性性质，互质的数质因子分解无交集：

$$\phi(N) = \phi(N' \times p_1) = \phi(N') \times \phi(p_1) = \phi(N') \times (p_1 - 1)$$

在筛质数的同时筛出欧拉函数。

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



```

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

```

容斥原理

[2652. 倍数求和 - 力扣 \(LeetCode\)](#)

给你一个正整数 n ，请你计算在 $[1, n]$ 范围内能被 3 或者 5 或者 7 整除的所有整数之和。

返回一个整数，用于表示给定范围内所有满足约束条件的数字之和。

利用等差数列求和公式： $1 \sim n$ 中能被 x 整除的数之和 $= (x + 2 \cdot x + \dots + n // x \cdot x) = x \cdot (1 + n // x) * (n // x) // 2$

因而，

```

class Solution:
    def sumOfMultiples(self, n: int) -> int:
        # 定义 f(x) 为能被 x 整除的数字之和
        def f(x):
            return x * (1 + n // x) * (n // x) // 2
        return f(3) + f(5) + f(7) - f(15) - f(21) - f(35) + f(105)

```

数学公式

排序不等式

结论：对于两个有序数组的乘积和，顺序和 \geq 乱序和 \geq 倒序和。

对于 $a_1 \leq a_2 \leq \dots \leq a_n$, $b_1 \leq b_2 \leq \dots \leq b_n$, 并有 c_1, c_2, \dots, c_n 是 b_1, b_2, \dots, b_n 的乱序排列。有如下关系：

$$\sum_{i=1}^n a_i b_{n+1-i} \leq \sum_{i=1}^n a_i c_i \leq \sum_{i=1}^n a_i b_i。$$

当且仅当 $a_i = a_j$ 或者 $b_i = b_j$ ($1 \leq i, j \leq n$) 时，等号成立。

区间递增k个数

结论：对于 $i_0 = a$ ，每次递增 k ，在区间 $[a, b]$ 内的个数是：

$$(b - a - 1) // k + 1$$

平均数不等式

$$x_1, x_2, \dots, x_n \in \mathbb{R}_+ \Rightarrow \frac{n}{\sum_{i=1}^n \frac{1}{x_i}} \leq \sqrt[n]{\prod_{i=1}^n x_i} \leq \frac{\sum_{i=1}^n x_i}{n} \leq \sqrt{\frac{\sum_{i=1}^n x_i^2}{n}}$$

当且仅当 $x_1 = x_2 = \dots = x_n$, 等号成立。

即：调和平均数，几何平均数，算术平均数，平方平均数（调几算方）

应用：

例如当算术平均数为定值， x_i 分布越接近，平方平均数越小，因此可以进行贪心算法：

[3081. 替换字符串中的问号使分数最小 - 力扣 \(LeetCode\)](#)

各个字母之间的出现次数的差异越小，越均衡，最终结果越小。可以基于贪心 + 堆进行维护，每次取出出现次数最小中字典序最小的字符。

```
def minimizeStringValue(self, s: str) -> str:
    cnt = Counter(s)
    hq = [(cnt[ch], ch) for ch in string.ascii_lowercase]
    heapq.heapify(hq)
    alp = []
    res = list(s)
    for i in range(s.count('?')):
        v, k = heapq.heappop(hq)
        v += 1
        alp.append(k)
        heapq.heappush(hq, (v, k))
    alp.sort(reverse = True)
    for i, x in enumerate(res):
        if res[i] == '?':
            res[i] = alp.pop()
    return ''.join(res)
```

求和公式

$$\sum_1^n n^2 = \frac{n \cdot (n + 1) \cdot (2n + 1)}{6}$$

取模性质

模运算与基本四则运算有些相似，但是除法例外。其规则如下：

$(a + b) \% p = (a \% p + b \% p) \% p$

$(a - b) \% p = (a \% p - b \% p) \% p$

$(a * b) \% p = (a \% p * b \% p) \% p$

$a^b \% p = ((a \% p)^b) \% p$

结合律：

$((a+b) \% p + c) \% p = (a + (b+c) \% p) \% p$

$((ab) \% p * c) \% p = (a * (bc) \% p) \% p$

交换律：

$(a + b) \% p = (b+a) \% p$

$(a * b) \% p = (b * a) \% p$

分配律：

$(a+b) \% p = (a \% p + b \% p) \% p$

$((a + b) \% p * c) \% p = ((a * c) \% p + (b * c) \% p) \% p$

组合数学

$$A_m^n = \frac{m!}{n!}, C_m^n = \frac{m!}{n!(m-n)!}$$

$$C_m^n = C_m^{m-n}$$

$$C_m^n = C_{m-1}^n + C_{m-1}^{n-1}$$

$$C_n^0 + C_n^1 + \dots + C_n^n = 2^n$$

二项式定理

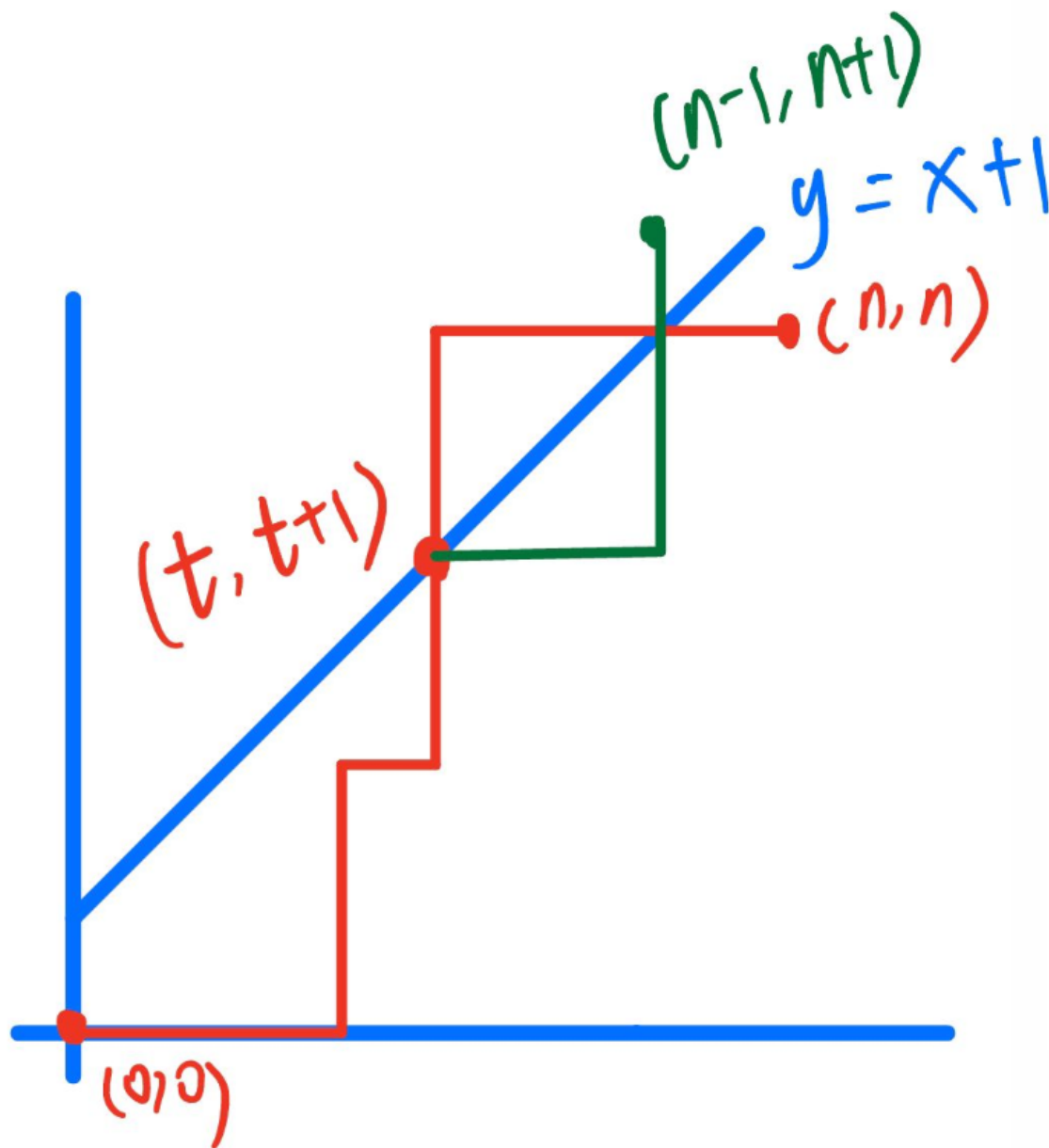
$$(a + b)^n = \sum_{i=0}^n C_n^i a^i b^{n-i}$$

卡特兰数

[5. 卡特兰数 \(Catalan\) 公式、证明、代码、典例. c n = n+1!\(n2n\)-CSDN博客](#)

给定 n 个0和 n 个1，排序成长度为 $2n$ 的序列。其中任意前缀中0的个数都不少于1的个数的序列的数量为：

$$H(n) = C_{2n}^n - C_{2n}^{n-1} = \frac{C_{2n}^n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$



证明方法:

看成从从 $(0, 0)$ 到 右上角 (n, n) , 每次只能向右或者向上, 向上的次数不超过向右的次数的路径数。

对于不合法的情况, 超过 $y = x$, 即一定触碰 $y = x + 1$, 取路径与 $y = x + 1$ 交点中, 距离原点最近的点, 将路径远离原点的部分关于 $y = x + 1$ 翻转。由于原来的终点 (n, n) 关于 $y = x + 1$ 翻转的点是 $(n - 1, n + 1)$, 所以不合法的路径数是 C_{2n}^{n-1}

递推公式1:

$$H(n+1) = H(0) \cdot H(n) + H(1) \cdot H(n-1) + \cdots + H(n) \cdot H(0) = \sum_{i=0}^n H(i) \cdot H(n-i)$$

证明方法: 从 $(0, 0)$ 到 $(n+1, n+1)$ 的路径数可以看成成分三步:

首先从 $(0, 0)$ 走到 (i, i) , 其方案数为 $H(i)$; 然后从 (i, i) 走到 (n, n) 方案数为 $H(n-i)$; 最后从 (n, n) 走到 $(n+1, n+1)$ 其方案数为 $H(1) = 1$ 。

递推公式2:

$$H(n) = H(n-1) \cdot \frac{2n(2n-1)}{(n+1)n} = H(n-1) \cdot \frac{(4n-2)}{(n+1)}$$

推论:

前几项: 1, 1, 2, 5, 14, 42, 132, 429, 1430

- n 个节点可以构造的不同的二叉树的个数。(证明: $F(n)$ 为有 n 个节点的二叉树的所有根节点个数。其左子树的可能情况为 $F(i)$, $i \in [0, n]$, 对应右子树的情况为 $F(n-i)$, 乘积求和形式即为卡特兰数列的递推式。
- 从 $(0, 0)$ 到 右上角 (n, n) , 每次只能向右或者向上, 向上的次数不超过向右的次数的路径数。(即不超过 $y = x$)
- 一个无穷大栈, 进栈顺序为 $1, 2, \dots, n$ 的出栈顺序数

- n 个左括号和 n 个右括号构成的括号序列，能够构成的有效括号序列个数。

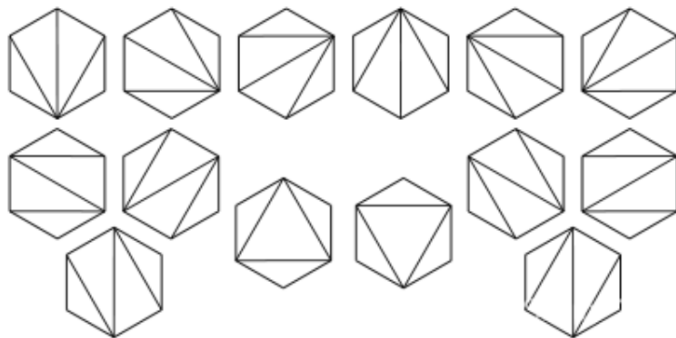
凸多边形划分问题

在一个 n 边形中，通过不相交于 n 边形内部的对角线，把 n 边形拆分为若干个三角形，问有多少种拆分方案？

如五边形有如下5种拆分方案：



如六边形有如下14种拆分方案：



以凸多边形的一边为基，设这条边的2个顶点为A和B。从剩余顶点中选1个，可以将凸多边形分成三个部分，中间是一个三角形，左右两边分别是两个凸多边形，然后求解左右两个凸多边形。

2. 设问题的解 $f(n)$ ，其中 n 表示顶点数，那么 $f(n) = f(2)f(n-1) + f(3)f(n-2) + \dots + f(n-2)f(3) + f(n-1)f(2)$ 。

其中， $f(2)f(n-1)$ 表示：三个相邻的顶点构成一个三角形，另外两个部分的顶点数分别为2（一条直线两个点）和 $n-1$ 。

其中， $f(3)f(n-2)$ 表示：将凸多边形分为三个部分，左右两边分别是一个有3个顶点的三角形和一个有 $n-2$ 个顶点的多边形。

3. 设 $f(2) = 1$ ，那么 $f(3) = 1$ ， $f(4) = 2$ ， $f(5) = 5$ 。结合递推式，不难发现 $f(n)$ 等于 $H(n-2)$ 。

矩阵乘法/矩阵快速幂/快速幂

矩阵乘法时间复杂度： $O(n^3)$

矩阵乘法

```
moder = 10**9 + 7

def mul(a, b):
    m_a, n_a = len(a), len(a[0])
    m_b, n_b = len(b), len(b[0])
    c = n_a # 可以加一个n_a和m_b的判等
    res = [[0]*n_b for _ in range(m_a)]
    for i in range(m_a):
        for j in range(n_b):
            tmp = 0
            for k in range(c):
                # tmp = (tmp + (a[i][k] * b[k][j]) % moder) % moder # 如果需要取模
                tmp += a[i][k] * b[k][j]
            res[i][j] = tmp
    return res
```

矩阵快速幂

```
moder = 10**9 + 7

def mul(a, b):
    m_a, n_a = len(a), len(a[0])
    m_b, n_b = len(b), len(b[0])
```

```

c = n_a # 可以加一个n_a和m_b的判等
res = [[0]*n_b for _ in range(m_a)]
for i in range(m_a):
    for j in range(n_b):
        tmp = 0
        for k in range(c):
            # tmp = (tmp + (a[i][k] * b[k][j]) % moder) % moder # 如果需要取模
            tmp += a[i][k] * b[k][j]
        res[i][j] = tmp
    return res

def pow(a, n):
    res = [ # 其他形状的改成nxn的E矩阵
        [1, 0],
        [0, 1]
    ]
    while n:
        if n & 1:
            res = mul(res, a)
        a = mul(a, a)
        n >>= 1
    return res

```

快速幂

```

def pow(a, n, moder):
    res = 1
    while n:
        if n & 1: res = (res * a) % moder
        n >>= 1
        a = (a * a) % moder
    return res

```

高等数学

(1). 调和级数

$$\sum_{i=1}^n \frac{1}{k} \text{ 是调和级数, 其发散率表示为 } \sum_{i=1}^n \frac{1}{k} = \ln n + C$$

经典应用：求一个数的约数的个数期望值

- 考虑 $1 \sim n$ 所有的数的约数个数。
- 从筛法的角度来看，拥有约数2的所有的数，是 $1 \sim n$ 中所有2的倍数，大约是 $n // 2$ 个。
- 所以 $1 \sim n$ 所有的数的约数个数和 可以看成 所有的倍数的个数 $= n/1 + n/2 + n/3 + \dots + n/n = n \sum_{i=1}^n \frac{1}{i} = n \ln n$ 。
- 所以=, 从期望角度来讲，一个数 n 的约束个数的期望约是 $\ln n$

数据结构

并查集

合并和查询的时间复杂度：近似 $O(1)$

`find(u) == find(v)` 表示 u, v 在同一集合。

路径压缩

递归写法

```

fa = list(range(n))
# 查找x集合的根
def find(x):
    if fa[x] != x:
        fa[x] = find(fa[x])
    return fa[x]

# v并向u中Z
def union(u, v):
    if find(u) != find(v):
        fa[find(v)] = find(u)

```

迭代写法

```

fa = list(range(n))

def find(x):
    root = x
    while fa[root] != root:
        root = fa[root]
    while fa[x] != x: # 路径压缩
        x, fa[x] = fa[x], root
    return root

def union(u, v):
    root_u = find(u)
    root_v = find(v)
    if root_u != root_v:
        fa[root_v] = root_u

```

常用拓展：

- 记录每个集合大小：绑定到根节点
- 记录每个点到根节点的距离：绑定到每一个节点上

并查集维护连通分量

[1998. 数组的最大公因数排序 - 力扣 \(LeetCode\)](#)

质因子分解 + 并查集判断连通分量。

将所有数看成一个图中的节点。任意两个数 u, v ，如果不互质 ($\gcd > 1$) 说明存在一条边 $u \sim v$ 。显然一种做法是用 $O(n^2)$ 的时间维护所有节点对应的连通块。然而，实际上只需要对每个数 x 和它的所有质因子进行合并，这样可以保证有相同质因子的两个元素，他们可以在同一个连通分量。

记数组中最大值 $m = \max(nums)$ ，可以看成有一个 m 个节点的图。每次质因子分解的时间复杂度是 $O(\sqrt{x})$ ，所以从 $O(n^2)$ 优化到 $O(n\sqrt{m})$ 。最后，将排序好的数组和原数组对应位置上的元素进行对比。判断两个元素是否同属于一个连通分量即可。

时间复杂度： $O\left(n(\sqrt{m} \cdot \alpha(m)) + n \log n\right)$

```

def gcdSort(self, nums: List[int]) -> bool:
    n = len(nums)
    fa = list(range(max(nums) + 1))
    def find(x): # x 压缩到 fa[x] 中
        if fa[x] != x:
            fa[x] = find(fa[x])
        return fa[x]
    def union(u, v): # u 合并到 v 中
        if find(u) != find(v):
            fa[find(u)] = find(v)

    for i, x in enumerate(nums):
        xx = x
        for j in range(2, int(sqrt(x)) + 1):
            if x % j == 0:
                union(j, xx)
                while x % j == 0:

```

```

        x //= j
    if x > 1:
        union(x, xx)
sorted_nums = sorted(nums)
for u, v in zip(nums, sorted_nums):
    if u == v: continue
    # 不在位元素, 需要看是否在同一连通分量
    if find(u) != find(v): return False
return True

```

[952. 按公因数计算最大组件大小 - 力扣 \(LeetCode\)](#)

```

def largestComponentSize(self, nums: List[int]) -> int:
    n = len(nums)
    m = max(nums)
    fa = list(range(m + 1))
    def find(x):
        if fa[x] != x:
            fa[x] = find(fa[x])
        return fa[x]
    def union(u, v):
        if find(u) != find(v):
            fa[find(u)] = find(v)
    for x in nums:
        xx = x
        for j in range(2, int(sqrt(x)) + 1):
            if x % j == 0:
                union(xx, j)
                while x % j == 0:
                    x //= j
        if x > 1:
            union(xx, x)
    for x in nums:
        find(x)
    cnt = Counter()
    for x in nums: cnt[fa[x]] += 1
    return max(cnt.values())

```

并查集维护集合 / 连通块大小

[2334. 元素值大于变化阈值的子数组 - 力扣 \(LeetCode\)](#)

$O(n \log n)$

并查集: 让每个子数组的默认树根是最右侧节点 i 再 + 1, 即 $i + 1$ 。

对 $nums$ 中每一个元素和对应的下标 按照降序排序, 每次向右合并, 当前的 x 一定是子数组中最小的。返回合并后并查集大小 $k - 1$ (减一使得不包含哨兵根节点)。不断向右侧合并直到出现符合的。在返回并查集大小时, 只需要保证集合大小和树根节点进行绑定即可。

```

def validSubarraySize(self, nums: List[int], threshold: int) -> int:
    n = len(nums)
    fa = list(range(n + 1)) # n是哨兵树根
    siz = [1] * (n + 1)
    def find(x):
        if fa[x] != x: fa[x] = find(fa[x])
        return fa[x]
    def union(u, v): # u 合并进 v
        if find(u) != find(v):
            siz[find(v)] += siz[find(u)]
            fa[find(u)] = find(v)
        return siz[find(v)]
    for x, i in sorted(zip(nums, range(n)), reverse = True):
        k = union(i, i + 1) - 1
        if x > (threshold // k):
            return k
    return -1

```

[2867. 统计树中的合法路径数目 - 力扣 \(LeetCode\)](#)

并查集维护所有非质数子连通块的大小。

```
def countPaths(self, n: int, edges: List[List[int]]) -> int:
    primes = []
    N = n + 10
    is_prime = [True] * N
    is_prime[0] = is_prime[1] = False
    for i in range(2, N):
        if is_prime[i]:
            primes.append(i)
            for p in primes:
                if i * p >= N:
                    break
                is_prime[i * p] = False
                if i % p == 0:
                    break
    e = [[] for _ in range(n + 1)]
    fa = list(range(n + 1))
    siz = [1] * (n + 1)
    def find(x):
        if fa[x] != x:
            fa[x] = find(fa[x])
        return fa[x]
    def union(u, v): # u 合并到 v
        if find(u) != find(v):
            siz[find(v)] += siz[find(u)]
            fa[find(u)] = find(v)

    for u, v in edges:
        e[u].append(v)
        e[v].append(u)
        if not is_prime[u] and not is_prime[v]:
            union(u, v)

    res = 0
    vis = [False] * (n + 1)
    for u in range(1, n + 1):
        if not vis[u] and is_prime[u]:
            # 遍历 u 的所有非质数连通块
            vis[u] = True
            cur_siz = 0
            for v in e[u]:
                if not is_prime[v]:
                    sz = siz[find(v)]
                    res += sz + sz * cur_siz
                    cur_siz += sz

    return res
```

并查集维护连通块按位与的值

[100244. 带权图里旅途的最小代价 - 力扣 \(LeetCode\)](#)

```
def minimumCost(self, n: int, edges: List[List[int]], query: List[List[int]]) -> List[int]:
    fa = list(range(n))
    cc_and = [-1] * n
    def find(x):
        if fa[x] != x:
            fa[x] = find(fa[x])
        return fa[x]
    def union(u, v, w): # v 合并到 u 中
        if find(u) != find(v):
            cc_and[find(u)] &= cc_and[find(v)]
            fa[find(v)] = find(u)

    for u, v, w in edges:
        # 各自连通块内更新，只要更新其一即可
        cc_and[find(u)] &= w
        union(u, v, w)

    return [0 if u == v else (-1 if find(u) != find(v) else cc_and[find(u)]) for u, v in query]
```


字典树

26叉字典树

```
class Trie:

    def __init__(self):
        self.is_end = False
        self.next = [None] * 26

    def insert(self, word: str) -> None:
        node = self
        for ch in word:
            idx = ord(ch) - ord('a')
            if not node.next[idx]:
                node.next[idx] = Trie()
            node = node.next[idx]
        node.is_end = True

    def search(self, word: str) -> bool:
        node = self
        for ch in word:
            idx = ord(ch) - ord('a')
            if not node.next[idx]:
                return False
            node = node.next[idx]
        return node.is_end

    def startswith(self, prefix: str) -> bool:
        node = self
        for ch in prefix:
            idx = ord(ch) - ord('a')
            if not node.next[idx]:
                return False
            node = node.next[idx]
        return True
```

哈希字典树

```
def countPrefixSuffixPairs(self, words: List[str]) -> int:
    class Node:
        __slots__ = 'children', 'cnt'
        def __init__(self):
            self.children = {} # 用字典的字典树
            self.cnt = 0
    res = 0
    root = Node() # 树根
    for word in words:
        cur = root
        for p in zip(word, word[::-1]): # (p[i], p[n - i - 1])
            if p not in cur.children:
                cur.children[p] = Node()
            cur = cur.children[p]
            res += cur.cnt
        cur.cnt += 1
    return res
```

```
class Trie:

    def __init__(self):
        self.end = False
        self.next = {}

    def insert(self, word: str) -> None:
        p = self
        for ch in word:
            if ch not in p.next:
                p.next[ch] = Trie()
```

```

        p = p.next[ch]
        p.end = True

    def search(self, word: str) -> bool:
        p = self
        for ch in word:
            if ch not in p.next:
                return False
            p = p.next[ch]
        return p.end

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

# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startswith(prefix)

```

线段树

动态开点 + lazy 线段树

```

# https://leetcode.cn/problems/range-module/
class Node:
    __slots__ = ['l', 'r', 'lazy', 'val']
    def __init__(self):
        self.l = None
        self.r = None
        self.lazy = 0
        self.val = False

class SegmentTree:
    __slots__ = ['root']
    def __init__(self):
        self.root = Node()

    def do(self, node, val):
        node.val = val
        node.lazy = 1

    # 下放lazy标记。如果是孩子为空，则动态开点
    def pushdown(self, node):
        if node.l is None:
            node.l = Node()
        if node.r is None:
            node.r = Node()

    # 根据lazy标记信息，更新左右节点，然后将lazy信息清除
    if node.lazy:
        self.do(node.l, node.val)
        self.do(node.r, node.val)
        node.lazy = 0

    def query(self, L, R, node = None, l = 1, r = int(1e9)):

        # 查询默认从根节点开始
        if node is None:
            node = self.root

        if L <= l and r <= R:

```

```

        return node.val

# 下放标记、根据标记信息更新左右节点，然后清除标记
self.pushdown(node)

mid = (l + r) >> 1

vl = vr = True

if L <= mid:
    vl = self.query(L, R, node.l, l, mid)
if R > mid:
    vr = self.query(L, R, node.r, mid + 1, r)
return vl and vr

def update(self, L, R, val, node = None, l = 1, r = int(1e9)):

    # 查询默认从根节点开始
    if node is None:
        node = self.root

    if L <= l and r <= R:
        self.do(node, val)
        return

    mid = (l + r) >> 1

    # 下放标记、根据标记信息更新左右节点，然后清除标记
    self.pushdown(node)

    if L <= mid:
        self.update(L, R, val, node.l, l, mid)
    if R > mid:
        self.update(L, R, val, node.r, mid + 1, r)

    # node.val 为 True 表示这个节点所在区间，均被“跟踪”
    node.val = bool(node.l and node.l.val and node.r and node.r.val)

class RangeModule:

    def __init__(self):
        self.tree = SegmentTree()

    def addRange(self, left: int, right: int) -> None:
        self.tree.update(left, right - 1, True)

    def queryRange(self, left: int, right: int) -> bool:
        return self.tree.query(left, right - 1)

    def removeRange(self, left: int, right: int) -> None:
        self.tree.update(left, right - 1, False)

# Your RangeModule object will be instantiated and called as such:
# obj = RangeModule()
# obj.addRange(left,right)
# param_2 = obj.queryRange(left,right)
# obj.removeRange(left,right)

```

线段树优化DP问题

[2617. 网格图中最少访问的格子数 - 力扣 \(LeetCode\)](#)

单点修改 + 区间查询

```

class SegmentTree:
    def __init__(self, n: int):
        self.n = n
        self.tree = [inf] * (4 * n)

```

```

def op(self, a, b):
    return min(a, b)

def update(self, ul, ur, val, idx = 1, l = 1, r = None):
    if r is None: r = self.n
    if ul <= l and r <= ur:
        self.tree[idx] = val
        return
    mid = (l + r) >> 1
    if ul <= mid: self.update(ul, ur, val, idx * 2, l, mid)
    if ur > mid: self.update(ul, ur, val, idx * 2 + 1, mid + 1, r)
    self.tree[idx] = self.op(self.tree[idx * 2], self.tree[idx * 2 + 1]) # 更新当前节点的值

def query(self, ql, qr, idx = 1, l = 1, r = None):
    if r is None: r = self.n
    if ql <= l and r <= qr:
        return self.tree[idx]
    mid = (l + r) >> 1
    ans1, ansr = inf, inf
    if ql <= mid: ans1 = self.query(ql, qr, idx * 2, l, mid)
    if qr > mid: ansr = self.query(ql, qr, idx * 2 + 1, mid + 1, r)
    return self.op(ans1, ansr)

class Solution:
    def minimumVisitedCells(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        treey = [SegmentTree(m) for _ in range(n)]
        # treey[j] 是第j 列的线段树
        for i in range(m - 1, -1, -1):
            treex = SegmentTree(n)
            for j in range(n - 1, -1, -1):
                if i == m - 1 and j == n - 1:
                    treex.update(j + 1, j + 1, 1)
                    treey[j].update(i + 1, i + 1, 1)
                    continue
                g = grid[i][j]
                if g == 0: continue
                mnx = treex.query(j + 1 + 1, min(g + j, n - 1) + 1) if j < n - 1 else inf
                mny = treey[j].query(i + 1 + 1, min(g + i, m - 1) + 1) if i < m - 1 else inf
                mn = min(mnx, mny) + 1
                treex.update(j + 1, j + 1, mn)
                treey[j].update(i + 1, i + 1, mn)
            res = treey[0].query(1, 1)
            return res if res != inf else -1

```

最值查询朴素无更新线段树:

```

class Solution:
    def subArrayRanges(self, nums: List[int]) -> int:
        class SegmentTree:
            def __init__(self, n, flag):
                self.n = n
                self.tree = [inf * flag] * (4 * n)
                self.flag = flag
            def op(self, a, b):
                if self.flag == 1: return min(a, b)
                elif self.flag == -1: return max(a, b)
            def build(self, idx = 1, l = 1, r = None):
                if not r: r = self.n
                if l == r:
                    self.tree[idx] = nums[l - 1]
                    return
                mid = (l + r) >> 1
                self.build(idx * 2, l, mid)
                self.build(idx * 2 + 1, mid + 1, r)
                self.tree[idx] = self.op(self.tree[idx * 2], self.tree[idx * 2 + 1])
            def query(self, ql, qr, idx = 1, l = 1, r = None):
                if not r: r = self.n
                if ql <= l and r <= qr:
                    return self.tree[idx]

```

```

        ans1, ansr = inf * self.flag, inf * self.flag
        mid = (l + r) >> 1
        if ql <= mid: ans1 = self.query(ql, qr, idx * 2, l, mid)
        if qr > mid: ansr = self.query(ql, qr, idx * 2 + 1, mid + 1, r)
        return self.op(ans1, ansr)

n = len(nums)
mxtr, mntr = SegmentTree(n, -1), SegmentTree(n, 1)
res = 0
mxtr.build()
mntr.build()
for i in range(n):
    for j in range(i + 1, n):
        res += mxtr.query(i + 1, j + 1) - mntr.query(i + 1, j + 1)
return res

```

递归动态开点（无lazy）线段树

区间覆盖统计问题，区间覆盖不需要重复操作，不需要进行lazy传递

但是数据范围较大，需要动态开点

```

# https://leetcode.cn/problems/count-integers-in-intervals
class CountIntervals:
    __slots__ = 'left', 'right', 'l', 'r', 'val'

    def __init__(self, l = 1, r = int(1e9)):
        self.left = self.right = None
        self.l, self.r, self.val = l, r, 0

    def add(self, l: int, r: int) -> None:

        # 覆盖区间操作，不需要重复覆盖，饱和区间无需任何操作
        if self.val == self.r - self.l + 1:
            return

        if l <= self.l and self.r <= r: # self 已被区间 [l,r] 完整覆盖，不再继续递归
            self.val = self.r - self.l + 1
            return

        mid = (self.l + self.r) >> 1

        # 动态开点
        if self.left is None:
            self.left = CountIntervals(self.l, mid) # 动态开点

        if self.right is None:
            self.right = CountIntervals(mid + 1, self.r) # 动态开点

        if l <= mid:
            self.left.add(l, r)
        if mid < r:
            self.right.add(l, r)

        # self.val 的值，表示区间[self.l, self.r] 中被覆盖的点的个数
        self.val = self.left.val + self.right.val

    def count(self) -> int:
        return self.val

```

lazy线段树（点区间赋值）

```

class SegmentTree:
    __slots__ = ['node', 'lazy']
    def __init__(self, n: int):
        self.node = [0] * (4 * n)

```

```

self.lazy = [0] * (4 * n)

def build(self, i, l, r):
    if l == r:
        self.node[i] = Nums[l - 1]
        return
    mid = (l + r) >> 1

    self.build(i * 2, l, mid)
    self.build(i * 2 + 1, mid + 1, r)

    self.node[i] = self.node[i * 2] + self.node[i * 2 + 1]

# 更新节点值, 设置lazy标记
def do(self, i, l, r, val):
    self.node[i] = val * (l - r + 1)
    self.lazy[i] = val

# 检查标记, 根据标记根据子节点信息, 下放标记, 清除标记
def pushdown(self, i, l, r):
    if self.lazy[i]:
        val = self.lazy[i]
        mid = (l + r) >> 1
        self.do(i * 2, l, mid, val)
        self.do(i * 2 + 1, mid + 1, r, val)
        self.lazy[i] = 0

def update(self, i, l, r, L, R, val):
    if L <= l and r <= R:
        # 区间更新
        self.do(i, l, r, val)
        return

    # 检查lazy标记
    self.pushdown(i, l, r)

    # 左右递归更新
    mid = (l + r) >> 1
    if L <= mid:
        self.update(i * 2, l, mid, L, R, val)
    if R > mid:
        self.update(i * 2 + 1, mid + 1, r, L, R, val)

    # 更新节点值: 区间和
    self.node[i] = self.node[i * 2] + self.node[i * 2 + 1]

def query(self, i, l, r, L, R) -> int:
    if L <= l and r <= R:
        return self.node[i]

    # 检查lazy标记
    self.pushdown(i, l, r)

    mid = (l + r) >> 1

    vl, vr = 0, 0
    if L <= mid:
        vl = self.query(i * 2, l, mid, L, R)
    if R > mid:
        vr = self.query(i * 2 + 1, mid + 1, r, L, R)
    return vl + vr

```

lazy 线段树 (01翻转)

```

class Solution:
    def handleQuery(self, nums1: List[int], nums2: List[int], queries: List[List[int]]) -> List[int]:
        n = len(nums1)
        node = [0] * (4 * n)
        # 懒标记: True表示该节点代表的区间被曾经被修改, 但是其子节点尚未更新
        lazy = [False] * (4 * n)

```

```

# 初始化线段树
def build(i = 1, l = 1, r = n):
    if l == r:
        node[i] = nums1[l - 1]
        return
    mid = (l + r) >> 1
    build(i * 2, l, mid)
    build(i * 2 + 1, mid + 1, r)
    # 维护区间 [l, r] 的值
    node[i] = node[i * 2] + node[i * 2 + 1]

# 更新节点值, 并设置lazy标记
def do(i, l, r):
    node[i] = r - l + 1 - node[i]
    lazy[i] = not lazy[i]

# 区间更新: 本题中更新区间[l, r] 相当于做翻转
def update(L, R, i = 1, l = 1, r = n):
    if L <= l and r <= R:
        do(i, l, r)
        return

    mid = (l + r) >> 1
    if lazy[i]:
        # 根据标记信息更新p的两个左右子节点, 同时为子节点增加标记
        # 然后清除当前节点的标记
        do(i * 2, l, mid)
        do(i * 2 + 1, mid + 1, r)
        lazy[i] = False

    if L <= mid:
        update(L, R, i * 2, l, mid)
    if R > mid:
        update(L, R, i * 2 + 1, mid + 1, r)

    # 更新节点值
    node[i] = node[i * 2] + node[i * 2 + 1]

build()

res, s = [], sum(nums2)
for op, L, R in queries:
    if op == 1:
        update(L + 1, R + 1)
    elif op == 2:
        s += node[1] * L
    else:
        res.append(s)
return res

```

树状数组

```

# 下标从1开始
class FenwickTree:
    def __init__(self, length: int):
        self.length = length
        self.tree = [0] * (length + 1)
    def lowbit(self, x: int) -> int:
        return x & (-x)

    # 更新自底向上
    def update(self, idx: int, val: int) -> None:
        while idx <= self.length:
            self.tree[idx] += val
            idx += self.lowbit(idx)

    # 查询自顶向下
    def query(self, idx: int) -> int:

```

```

        res = 0
        while idx > 0:
            res += self.tree[idx]
            idx -= self.lowbit(idx)
        return res

class NumArray:

    def __init__(self, nums: List[int]):
        n = len(nums)
        self.nums = nums
        self.tree = FenwickTree(n)
        for i, x in enumerate(nums):
            self.tree.update(i + 1, x)

    def update(self, index: int, val: int) -> None:
        # 因为这里是更新为val, 所以节点增加的值应为val - self.nums[index]
        # 同时需要更新nums[idx]
        self.tree.update(index + 1, val - self.nums[index])
        self.nums[index] = val

    def sumRange(self, left: int, right: int) -> int:
        r = self.tree.query(right + 1)
        l = self.tree.query(left)
        return r - l

# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# obj.update(index,val)
# param_2 = obj.sumRange(left,right)

```

离散化树状数组 + 还原

```

class FenwickTree:
    def __init__(self, length: int):
        self.length = length
        self.tree = [0] * (length + 1)

    def lowbit(self, x: int) -> int:
        return x & (-x)

    # 更新自底向上
    def update(self, idx: int, val: int) -> None:
        while idx <= self.length:
            self.tree[idx] += val
            idx += self.lowbit(idx)

    # 查询自顶向下
    def query(self, idx: int) -> int:
        res = 0
        while idx > 0:
            res += self.tree[idx]
            idx -= self.lowbit(idx)
        return res

class Solution:

    def resultArray(self, nums: List[int]) -> List[int]:
        # 离散化 nums
        sorted_nums = sorted(nums)
        tmp = nums.copy()
        nums = [bisect.bisect_left(sorted_nums, x) + 1 for x in nums]
        # 还原
        mp_rev = {i: x for i, x in zip(nums, tmp)}
        n = len(nums)
        t1 = FenwickTree(n)
        t2 = FenwickTree(n)
        a = [nums[0]]
        b = [nums[1]]

```



```

t1.update(nums[0], 1)
t2.update(nums[1], 1)
for i in range(2, len(nums)):
    x = nums[i]
    c = len(a) - t1.query(x)
    d = len(b) - t2.query(x)
    if c > d or c == d and len(a) <= len(b):
        a.append(x)
        t1.update(x, 1)
    else:
        b.append(x)
        t2.update(x, 1)
# 还原为原始数据: i 为离散化秩, x 为还原值
return [mp_rev[i] for i in a] + [mp_rev[i] for i in b]

```

ST表 / 可重复贡献问题

可重复贡献问题: 指对于运算 opt , 满足 $x \text{ opt } x = x$ 。例如区间最值问题, 区间GCD问题。

ST表思想基于倍增, 不支持修改操作。

预处理: $O(n \log n)$

$f(i, j)$ 表示 $[i, i + 2^j - 1]$ 区间的最值, 则将其分为两半, $left = [i, i + 2^{j-1} - 1], right = [i + 2^{j-1}, i + 2^j - 1]$ 。

$$\text{则 } f(i, j) = \text{opt}(f(i, j-1), f(i + 2^{j-1}, j-1))$$

初始化时, $f(i, 0) = a[i]$;

对于 j 的上界需要满足 $i + 2^j - 1$ 能够取到 $n - 1$, 即 2^j 能够取到 n 。

所以外层循环条件 $j \in [1, \text{ceil}(\log_2^j) + 1)$ 。

对于 i 的上界需要满足 $i + 2^j - 1 < n$, 即 $i \in [0, n - 2^k + 1)$ 。

例如, 对于 $f(4, 3) = \text{opt}(f(4, 2), f(8, 2))$

```

lenj = math.ceil(math.log(n, 2)) + 1
f = [[0] * lenj for _ in range(n)]
for i in range(n):
    f[i][0] = a[i]
for j in range(1, lenj):
    # i + 2 ^ j < n + 1
    for i in range(n + 1 - (1 << j)):
        f[i][j] = opt(f[i][j - 1], f[i + (1 << (j - 1))][j - 1])

```

单次询问: $O(1)$

例如, 对于 $qry(5, 10)$, 区间长度为6, $\text{int}(\log_2^6) = 2$, 只需要 $k = 2^2$ 的两个区间一定可以覆盖整个区间。

即 $\text{opt}(5, 10) = \text{opt}(\text{opt}(5, 8), \text{opt}(7, 10))$, 即分别是 $(l, l + 2^k - 1)$ 和 $(r - 2^k + 1, r)$

$$\text{qry}(l, r) = \text{opt}(\text{qry}(l, k), \text{qry}(r - 2^k + 1, k))$$

```

def qry(l, r):
    k = log[r - l + 1]
    return opt(f[l][k], f[r - (1 << k) + 1][k])

```

可以提取预处理一个对数数组。例如 $\text{int}(\log(7)) = \text{int}(\log(3)) + 1 = \text{int}(\log(1)) + 1 + 1$

```

log = [0] * (n + 1)
for i in range(2, n + 1):
    log[i] = log[i >> 1] + 1

```

模板

```

import math
import sys

```

```

input=lambda:sys.stdin.readline().strip()
write=lambda x:sys.stdout.write(str(x)+'\n')
n, m = map(int, input().split())
a = list(map(int, input().split()))
# 2 ^ j
def opt(a, b):
    return max(a, b)
lenj = math.ceil(math.log(n, 2)) + 1
f = [[0] * lenj for _ in range(n)]
log = [0] * (n + 1)
for i in range(2, n + 1):
    log[i] = log[i >> 1] + 1
for i in range(n):
    f[i][0] = a[i]
for j in range(1, lenj):
    # i + 2 ^ j < n + 1
    for i in range(n + 1 - (1 << j)):
        f[i][j] = opt(f[i][j - 1], f[i + (1 << (j - 1))][j - 1])
def qry(l, r):
    k = log[r - l + 1]
    return opt(f[l][k], f[r - (1 << k) + 1][k])
for _ in range(m):
    l, r = map(int, input().split())
    # 调用write
    write(qry(l - 1, r - 1))

```

[2104. 子数组范围和 - 力扣 \(LeetCode\)](#)

```

def subArrayRanges(self, nums: List[int]) -> int:
    # f[i][j] 表示 [i, i + 2^j - 1] 的最值
    n = len(nums)
    lenj = ceil(math.log(n, 2)) + 1
    log = [0] * (n + 1)
    for i in range(2, n + 1):
        log[i] = log[i // 2] + 1

    class ST:
        def __init__(self, n, flag):
            self.flag = flag
            f = [[inf * flag] * lenj for _ in range(n)]
            for i in range(n):
                f[i][0] = nums[i]
            for j in range(1, lenj):
                for i in range(n + 1 - (1 << j)):
                    f[i][j] = self.op(f[i][j - 1], f[i + (1 << (j - 1))][j - 1])
            self.f = f
        def op(self, a, b):
            if self.flag == 1: return min(a, b)
            return max(a, b)
        def query(self, l, r):
            k = log[(r - l + 1)]
            return self.op(self.f[l][k], self.f[r - (1 << k) + 1][k])

    n = len(nums)
    mxtr, mntr = ST(n, -1), ST(n, 1)
    res = 0
    for i in range(n):
        for j in range(i + 1, n):
            res += mxtr.query(i, j) - mntr.query(i, j)
    return res

```

图论

建图

邻接矩阵

```

g = [[inf] * n for _ in range(n)]
for u, v, w in roads:
    g[u][v] = g[v][u] = w
    g[u][u] = g[v][v] = 0

```

邻接表

```

e = [[] for _ in range(n)]
for u, v, w in roads:
    e[u].append((v, w))
    e[v].append((u, w))

```

去重边建图

[100244. 带权图里旅途的最小代价 - 力扣 \(LeetCode\)](#)

这道题需要在建图的时候取AND运算的最小值。

```

e = [defaultdict(lambda: -1) for _ in range(n)]
for u, v, w in edges:
    e[v][u] = e[u][v] = e[u][v] & w

```

Floyd

```

mp = [[inf] * n for _ in range(n)]
for u, v, w in edges:
    mp[u][v] = mp[v][u] = w
    mp[u][u] = mp[v][v] = 0
for k in range(n):
    for u in range(n):
        for v in range(n):
            mp[u][v] = min(mp[u][v], mp[u][k] + mp[k][v])

```

Dijkstra

1. 朴素Dijkstra

适用于稠密图，时间复杂度： $O(n^2)$

```

g = [[inf] * n for _ in range(n)]
for u, v, w in roads:
    g[u][v] = g[v][u] = w
    g[u][u] = g[v][v] = 0
d = [inf] * n      # dist数组，d[i] 表示源点到i 的最短路径长度
d[0] = 0
v = [False] * n    # 节点访问标记
for _ in range(n - 1):
    x = -1
    for u in range(n):
        if not v[u] and (x < 0 or d[u] < d[x]):
            x = u
    v[x] = True
    for u in range(n):
        d[u] = min(d[u], d[x] + g[u][x])

```

[1976. 到达目的地的方案数 - 力扣 \(LeetCode\)](#)

最短路Dijkstra + 最短路Dp: 求源点0到任意节点i 的最短路个数。

```

def countPaths(self, n: int, roads: List[List[int]]) -> int:
    g = [[inf] * n for _ in range(n)]
    moder = 10 ** 9 + 7
    for u, v, w in roads:
        g[u][v] = g[v][u] = w
        g[u][u] = g[v][v] = 0
    d = [inf] * n      # dist数组，d[i] 表示源点到i 的最短路径长度

```

```

d[0] = 0
v = [False] * n      # 节点访问标记
mn, res = inf, 0
f = [0] * n # f[i] 表示源点到i节点的最短路个数
f[0] = 1
for _ in range(n - 1):
    x = -1
    for u in range(n):
        if not v[u] and (x < 0 or d[u] < d[x]):
            x = u
    v[x] = True
    for u in range(n):
        a = d[x] + g[x][u]
        if a < d[u]:      # 到u的最短路个数 = 经过x到u的个数 = 到x的最短路的个数
            d[u], f[u] = a, f[x]
        elif a == d[u] and u != x: # 路径一样短, 追加
            f[u] = (f[u] + f[x]) % moder
    return f[n - 1]

```

[743. 网络延迟时间 - 力扣 \(LeetCode\)](#)

有向图 + 邻接矩阵最短路

```

def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
    g = [[inf] * (n + 1) for _ in range(n + 1)]
    for u, v, w in times:
        g[u][v] = w
        g[u][u] = g[v][v] = 0
    d = [inf] * (n + 1)
    d[k] = 0
    v = [False] * (n + 1)
    for _ in range(n - 1):
        x = -1
        for u in range(1, n + 1):
            if not v[u] and (x < 0 or d[u] < d[x]):
                x = u
        v[x] = True
        for u in range(1, n + 1):
            d[u] = min(d[u], d[x] + g[x][u])
    res = max(d[1: ])
    return res if res != inf else -1

```

[2662. 前往目标的最小代价 - 力扣 \(LeetCode\)](#)

将有向图路径 转换为 节点。不需要建图，但是需要首先对 d 数组进行预处理。

```

def minimumCost(self, start: List[int], target: List[int], specialRoads: List[List[int]]) -> int:
    # 把路径(a, b) -> (c, d) 简化成 (c, d)
    t, s = tuple(target), tuple(start)
    d, v = defaultdict(lambda: inf), set()
    d[s] = 0
    def g(p, q):
        return abs(p[0] - q[0]) + abs(p[1] - q[1])
    # 补充start 和 target 节点
    specialRoads.append([s[0], s[1], t[0], t[1], g(s, t)])
    specialRoads.append([s[0], s[1], s[0], s[1], 0])
    while True:
        x = None
        # 找到距离 start最近的 且 未计算过的节点
        for x1, y1, x2, y2, w in specialRoads:
            u = (x2, y2)
            if u not in v and (not x or d[u] < d[x]):
                x = u
        v.add(x)
        if x == t:
            return d[t]
        for x1, y1, x2, y2, w in specialRoads:
            u0, u = (x1, y1), (x2, y2)
            # 两种情况, 1. start 经过 x 到达 u
            # 2. start 经过 x 再到 u0 从路径到达 u

```

```

d1 = d[x] + g(x, u)
d2 = d[x] + g(x, u0) + w
d[u] = min(d[u], d1, d2)

```

2.堆优化Dijkstra

适用于稀疏图（点个数的平方远大于边的个数），复杂度为 $O(m\log m)$ ， m 表示边的个数。

使用小根堆，存放未确定最短路点集对应的 $(d[i], i)$ 。对于同一个 i 可能存放多组不同 $d[i]$ 的元组，因此堆中元素的个数最多是 m 个。

寻找最小值的过程可以用一个最小堆来快速完成。

```

e = [[] for _ in range(n)]
for u, v, w in roads:
    e[u].append((v, w))
    e[v].append((u, w))

d = [inf] * n
d[0] = 0
hq = [(0, 0)] # 小根堆，存放未确定最短路点集对应的 (d[i], i)
while hq:
    dx, x = heapq.heappop(hq)
    if dx > d[x]: continue # 跳过重复出堆，首次出堆一定是最短路
    for u, w in e[x]:
        a = d[x] + w
        if a < d[u]:
            d[u] = a # 同一个节点u的最短路 d[u] 在出堆前会被反复更新
            heapq.heappush(hq, (a, u))

```

[1976. 到达目的地的方案数 - 力扣 \(LeetCode\)](#)

```

def countPaths(self, n: int, roads: List[List[int]]) -> int:
    e = [[] for _ in range(n)]
    for u, v, w in roads:
        e[u].append((v, w))
        e[v].append((u, w))

    moder = 10 ** 9 + 7
    f = [0] * n
    d = [inf] * n
    f[0], d[0] = 1, 0
    hq = [(0, 0)] # 小根堆，存放未确定最短路点集对应的 (d[i], i)
    while hq:
        dx, x = heapq.heappop(hq)
        if dx > d[x]: continue # 之前出堆过
        for u, w in e[x]:
            a = d[x] + w
            if a < d[u]:
                d[u] = a
                f[u] = f[x]
                heapq.heappush(hq, (a, u))
            elif a == d[u]:
                f[u] = (f[u] + f[x]) % moder
    return f[n - 1]

```

[743. 网络延迟时间 - 力扣 \(LeetCode\)](#)

有向图 + 邻接矩阵最短路

```

def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
    e = [[] * (n + 1) for _ in range(n + 1)]
    for u, v, w in times:
        e[u].append((v, w))
    d = [inf] * (n + 1)
    d[k] = 0
    hq = [(0, k)]

```

```

while hq:
    dx, x = heapq.heappop(hq)
    if dx > d[x]: continue
    for u, w in e[x]:
        a = d[x] + w
        if a < d[u]:
            d[u] = a
            heapq.heappush(hq, (a, u))
res = max(d[1: ])
return res if res < inf else -1

```

[2045. 到达目的地的第二短时间 - 力扣 \(LeetCode\)](#)

使用双列表d，存放最短和次短。将等红绿灯转换为松弛条件，通过t 来判断红灯还是绿灯。

```

def secondMinimum(self, n: int, edges: List[List[int]], time: int, change: int) -> int:
    # 将 节点 (u, t) 即 (节点, 时间) 作为新的节点
    e = [[] for _ in range(n + 1)]
    for u, v in edges:
        e[u].append(v)
        e[v].append(u)
    hq = [(0, 1)]
    # (t // change) & 1 == 0 绿色
    # (x, t) -> (u, t + time)

    # (t // change) & 1 == 1 红色
    # 需要 change - t % change 时间进入下一个节点
    d, dd = [inf] * (n + 1), [inf] * (n + 1)
    d[1] = 0
    while hq:
        t, x = heapq.heappop(hq)
        if d[x] < t and dd[x] < t: # 确认最小的和次小的
            continue
        for u in e[x]:
            nt = inf
            if (t // change) & 1 == 0:
                nt = t + time
            else:
                nt = t + change - t % change + time
            if nt < d[u]:
                d[u] = nt
                heapq.heappush(hq, (nt, u))
            elif dd[u] > nt > d[u]:
                dd[u] = nt
                heapq.heappush(hq, (nt, u))
    return dd[n]

```

3. 堆优化Dijkstra (字典写法)

转换建图 + 堆Dijkstra(字典写法)

[LCP 35. 电动车游城市 - 力扣 \(LeetCode\)](#)

```

def electricCarPlan(self, paths: List[List[int]], cnt: int, start: int, end: int, charge: List[int]) -> int:
    # 将(节点, 电量) 即 (u, c) 看成新的节点
    # 将充电等效转换成图
    # 则将节点i充电消耗时间charge[u] 看成从(u, c) 到 (u, c + 1) 有 w = 1
    n = len(charge)
    e = [[] for _ in range(n)]
    for u, v, w in paths:
        e[u].append((v, w))
        e[v].append((u, w))
    hq = [(0, start, 0)]
    d = {}
    while hq:
        dx, x, c = heapq.heappop(hq)
        if (x, c) in d: # 已经加入到寻找到最短路的集合中
            continue
        d[(x, c)] = dx

```

```

for u, w in e[x]:
    if c >= w and (u, c - w) not in d:
        heapq.heappush(hq, (w + dx, u, c - w))
if c < cnt:
    heapq.heappush(hq, (charge[x] + dx, x, c + 1))
return d[(end, 0)]

```

[743. 网络延迟时间 - 力扣 \(LeetCode\)](#)

```

def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
    e = [[] * (n + 1) for _ in range(n + 1)]
    for u, v, w in times:
        e[u].append((v, w))
    d = {}
    hq = [(0, k)]
    while hq:
        dx, x = heapq.heappop(hq)
        if x in d: continue # 跳过非首次出堆
        d[x] = dx # 首次出堆一定是最短路
        for u, w in e[x]:
            a = d[x] + w
            if u not in d: # 未确定最短路
                heapq.heappush(hq, (a, u)) # 入堆, 同一个节点可能用多组
    for i in range(1, n + 1):
        if i != k and i not in d:
            return -1
    return max(d.values())

```

[2045. 到达目的地的第二短时间 - 力扣 \(LeetCode\)](#)

求解严格次短路问题: 两个d字典, 一个存放最短, 一个存放严格次短

```

def secondMinimum(self, n: int, edges: List[List[int]], time: int, change: int) -> int:
    # 将 节点 (u, t) 即 (节点, 时间) 作为新的节点
    # (t // change) & 1 == 0 绿色
    # (x, t) -> (u, t + time)

    # (t // change) & 1 == 1 红色
    # 需要 change - t % change 时间进入下一个节点
    # (x, t) -> (u, t + change - t % change + time)

    e = [[] for _ in range(n + 1)]
    for u, v in edges:
        e[u].append(v)
        e[v].append(u)
    hq = [(0, 1)]
    d, dd = {}, {} # dd 是确认次短的字典
    while hq:
        t, x = heapq.heappop(hq)
        if x not in d:
            d[x] = t
        elif t > d[x] and x not in dd:
            dd[x] = t
        else:
            continue
        for u in e[x]:
            if (t // change) & 1 == 0:
                if u not in dd:
                    heapq.heappush(hq, (t + time, u))
            else:
                if u not in dd:
                    heapq.heappush(hq, (t + change - t % change + time, u))
    return dd[n]

```

转换建图问题: 可折返图 转换成 到达时间的奇偶问题

[2577. 在网格图中访问一个格子的最少时间 - 力扣 \(LeetCode\)](#)

```

class Solution:
    def minimumTime(self, grid: List[List[int]]) -> int:

```

```

# (w, x0, x1) 表示到达(x0, x1) 时刻至少为w
if grid[0][1] > 1 and grid[1][0] > 1: return -1
m, n = len(grid), len(grid[0])
deltas = [(1, 0), (-1, 0), (0, 1), (0, -1)]
target = (m - 1, n - 1)
d = {}
hq = [(0, (0, 0))]
while hq:
    dx, x = heappop(hq)
    if x in d: continue
    d[x] = dx
    if x == target: return d[target]
    x0, x1 = x[0], x[1]
    for u0, u1 in [(x0 + dx, x1 + dy) for dx, dy in deltas]:
        if not (0 <= u0 < m and 0 <= u1 < n) or (u0, u1) in d: continue
        u, t = (u0, u1), grid[u0][u1]
        if dx + 1 >= t:
            heappush(hq, (dx + 1, u))
        else:
            # 例如 3 -> 6, 折返一次变成5 后 + 1到达 6
            du = (t - dx - 1) if (t - dx) & 1 else t - dx
            heappush(hq, (dx + du + 1, u))

```

4. 最短路与子序列 和/积 问题

求解一个数组的所有子序列的和 / 积中第 k 小(大同理)问题, 其中子序列是原数组删去一些元素后剩余元素不改变相对位置的数组。

以和为例, 可以转化为最短路问题:

将子序列看成节点 (s, idx) , s 表示序列的和, idx 表示下一个位置, 则 $idx - 1$ 表示序列最后一个元素的位置。

例如 $[1, 2, 4, 4, 5, 9]$ 的其中一个子序列 $[1, 2]$, 对应节点 $(3, 2)$ 。如果从 $idx - 1$ 位置选或不选来看, 可以转换为子序列 $[1, 2, 4]$ 和 $[1, 4]$, 则定义节点之间的边权是序列和之差, 由于有序数组, 边权一定非负。

可以将原问题看成从 $[\]$ 为源节点的, 带正权的图。只需要不断求解到源节点的最短路节点, 就可以得到所有子序列从小到大的和的值。

假设有 n 个节点, 堆中元素个数不会超过 k , 时间复杂度是 $O(k \log k)$ 。

注意, 如果采用二分答案方式求解, 即想求出恰好有 k 个元素小于等于对应子序列之和 s 的算法, 时间复杂度为 $O(k \log U)$, $U = \sum a_i$

[2386. 找出数组的第 K 大和 - 力扣 \(LeetCode\)](#)

```

def kSum(self, nums: List[int], k: int) -> int:
    res = sum(x for x in nums if x > 0)
    nums = sorted([abs(x) for x in nums])
    # (s, idx) (子序列和, 当前下标)
    hq = [(0, 0)]
    while k > 1:
        # 每一次会将最小的子序列的和pop出去
        # pop k - 1次, 堆顶就是答案
        s, idx = heappop(hq)
        # 选 idx - 1
        if idx < len(nums):
            heappush(hq, (s + nums[idx], idx + 1))
        # 不选 idx - 1
        if idx:
            heappush(hq, (s + (nums[idx] - nums[idx - 1]), idx + 1))
        k -= 1
    return res - hq[0][0]

```

5. 动态修改边权

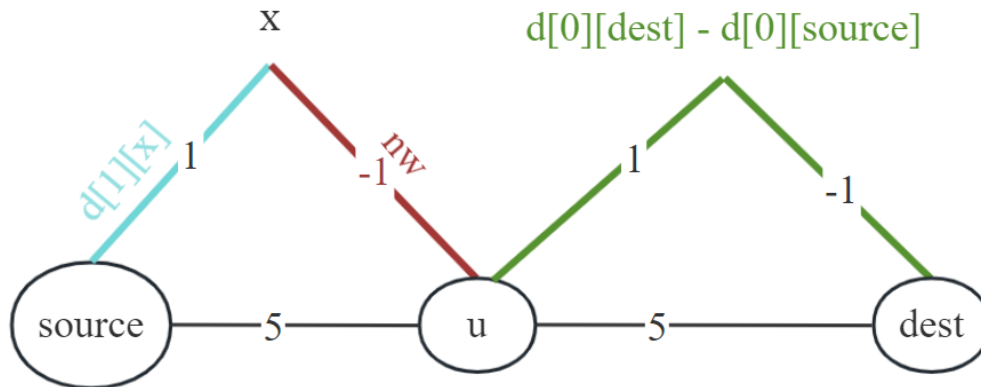
[2699. 修改图中的边权 - 力扣 \(LeetCode\)](#)

1. 在邻接表数组中记录原矩阵中边的位置, 方便修改
1. 记 $d_{signal, i}$ 表示第 $signal$ 次得到的节点 i 到源点的最短路。跑两次 `dijkstra`算法
1. 第二次修改边权时, 对于特殊边尝试修改条件:

$$d_{1,x} + nw + d_{0,dest} - d_{0,u} = target$$

$$\text{解得 } nw = target - d_{1,x} + d_{0,u} - d_{0,dest}$$

当这个值大于1时，是一个合法的边权，进行修改。



```
def modifiedGraphEdges(self, n: int, edges: List[List[int]], source: int, destination: int, target: int)
-> List[List[int]]:
    e = [[] for _ in range(n)]
    # 存放边的位置，方便在原矩阵直接修改
    for pos, (u, v, w) in enumerate(edges):
        e[u].append([v, pos])
        e[v].append([u, pos])

    total_d = [[inf] * n for _ in range(2)]
    total_d[0][source] = total_d[1][source] = 0
    def dijkstra(signal):
        d = total_d[signal] # 第signal次的最短路数组
        v = set()
        for _ in range(n - 1):
            x = -1
            for u in range(n):
                if u not in v and (x < 0 or d[u] < d[x]):
                    x = u
            v.add(x)
            for u, pos in e[x]:
                w = edges[pos][2]
                w = 1 if w == -1 else w
                # d[x] + nw + total_d[0][destination] - total_d[0][u] = target
                if signal == 1 and edges[pos][2] == -1:
                    nw = target - total_d[0][destination] + total_d[0][u] - d[x]
                    if nw > 1: # 合法修改
                        w = edges[pos][2] = nw
                d[u] = min(d[u], d[x] + w)
            return d[destination]
        if dijkstra(0) > target: return [] # 全为1也会超过target
        if dijkstra(1) < target: return [] # 最短路无法变大
        for e in edges:
            if e[2] == -1:
                e[2] = 1
        return edges
```

最小生成树

Prim

```
def solve():
    n, m = map(int, input().split())
    low_cost = [inf] * n
    g = [[] for _ in range(n)]
    for _ in range(m):
        u, v, w = map(int, input().split())
```

```

u, v = u - 1, v - 1
g[u].append((v, w))
g[v].append((u, w))

low_cost[0] = 0
res = 0
s = set()
for _ in range(n):
    dx, x = inf, -1
    for i in range(n):
        if i not in s and (x < 0 or low_cost[i] < dx):
            dx, x = low_cost[i], i
    s.add(x)
    res += dx

    for i, w in g[x]:
        if i not in s:
            low_cost[i] = min(low_cost[i], w)

if inf not in low_cost:
    print(res)
    return
print('orz')

```

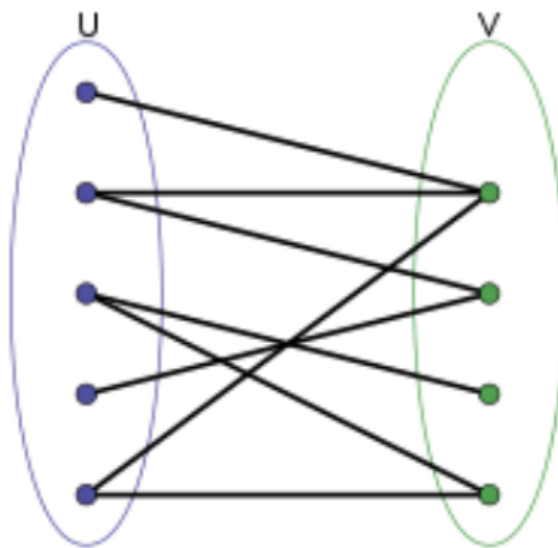
二分图

定义：无向图 $G(U, V, E)$ 中节点可以划分成互斥集合 U, V ，使得 $\forall (u, v) \in E$ 的两个端点分属于两个集合。

- 两个互斥点集中的任意两点之间都不存在边
- 任何一条边的两个端点分别来自互斥的两个点集 U, V
- 不存在奇数点的环（不存在奇数条边的环）

证明：因为走过一条边必然从一个集合走到另一个集合，要完成闭环必须走偶数条边（偶数个点）

- 可能存在孤点



[785. 判断二分图 - 力扣 \(LeetCode\)](#)

DFS染色：

```

def isBipartite(self, graph: List[List[int]]) -> bool:
    n = len(graph)
    color = [0] * n
    flag = True
    def dfs(u, c):
        nonlocal flag
        color[u] = c
        for v in graph[u]:
            if color[v] == 0:
                dfs(v, -c)

```

```

        elif color[v] == c:
            flag = False
            return
    for i in range(n):
        if color[i] == 0: dfs(i, 1)
        if not flag: return False
    return True

```

Bfs染色:

```

def isBipartite(self, graph: List[List[int]]) -> bool:
    n = len(graph)
    q = collections.deque()
    color = [0] * n
    for i in range(n):
        if not color[i]:
            q.append(i)
            color[i] = 1
    while q:
        u = q.popleft()
        c = color[u]
        for v in graph[u]:
            if not color[v]:
                color[v] = -c
                q.append(v)
            elif color[v] == c:
                return False
    return True

```

并查集做法:

维护两个并查集 U, V , 分别存储两个互斥点集。

对于每个节点 u 遍历其所有邻接节点 v 。如果遇到 u, v 在同一个并查集, 说明不满足二分图。(同一点集中出现连接的边)

否则将所有邻接节点加到另一个并查集中。

```

def isBipartite(self, graph: List[List[int]]) -> bool:
    n = len(graph)
    s = set()
    pa = list(range(n))
    def find(x):
        if pa[x] != x:
            pa[x] = find(pa[x])
        return pa[x]
    def union(u, v):
        if find(u) != find(v):
            pa[find(v)] = find(u)
    for u in range(n):
        if u not in s:
            s.add(u)
            p = None
            for v in graph[u]:
                if find(u) == find(v):
                    return False
                if p: union(p, v)
                p = v
    return True

```

连通块问题

[2867. 统计树中的合法路径数目 - 力扣 \(LeetCode\)](#)

DFS + 字典维护节点所在连通块大小

`cc_siz` 用来记录连通块的大小。`vis` 数组对质数节点进行记录, dfs的起始节点一定是质数节点的非质数子节点。

使用 `cc_node` 记录一次连通分量dfs得到的节点列表, 更新对应 `cc_siz` 的值。这样后续在遍历到已经遍历过的非质数连通块时, 可以直接得到结果。

```

def countPaths(self, n: int, edges: List[List[int]]) -> int:

```

```

primes = []
N = n + 10
is_prime = [True] * N
is_prime[0] = is_prime[1] = False
for i in range(2, N):
    if is_prime[i]:
        primes.append(i)
        for p in primes:
            if i * p >= N:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
e = [[] for _ in range(n + 1)]
for u, v in edges:
    e[u].append(v)
    e[v].append(u)
vis = [False] * (n + 1)
cc_siz = {}
cc_node = []
def dfs(u, fa):
    siz = 1
    cc_node.append(u)
    for v in e[u]:
        if v != fa and not is_prime[v]:
            siz += dfs(v, u)
    return siz
res = 0
for u in range(1, n + 1):
    if not vis[u] and is_prime[u]:
        vis[u] = True
        cur_siz = 0
        for v in e[u]:
            if is_prime[v]:
                continue
            # 对于每一个子连通分量
            if v in cc_siz:
                siz = cc_siz[v]
            else:
                cc_node.clear()
                siz = dfs(v, u)
                for node in cc_node:
                    cc_siz[node] = siz
            res += siz + siz * cur_siz
            cur_siz += siz
return res

```

DFS + 字典维护连通块的 AND 值 和 节点对应的连通块下标

[100244. 带权图里旅途的最小代价 - 力扣 \(LeetCode\)](#)

通过字典中连通块下标，判断两个节点是否在同一连通块内。

```

def minimumCost(self, n: int, edges: List[List[int]], query: List[List[int]]) -> List[int]:
    cc_and = {} # 键为节点，值为 (cc_cnt, and_ans)，即对应的连通块编号 和 连通块的and值
    cc_cnt = 0 # 计数，记录当前统计到第几个连通块
    cc_node = []
    e = [{} for _ in range(n)]
    for u, v, w in edges:
        if v not in e[u]:
            e[v][u] = e[u][v] = w
        else:
            e[v][u] = e[u][v] = e[u][v] & w
    vis = [False] * n

    def dfs(u):
        vis[u] = True
        cc_node.append(u)
        and_ans = -1
        for v in e[u]:
            w = e[u][v]

```

```

        and_ans &= w
        if not vis[v]:
            and_ans &= dfs(v)
    return and_ans
for u in range(n):
    if not vis[u]:
        and_ans = dfs(u)
        for node in cc_node:
            cc_and[node] = (cc_cnt, and_ans)
        cc_node.clear()
        cc_cnt += 1
return [0 if u == v else (cc_and[u][1] if cc_and[u][0] == cc_and[v][0] else -1)
        for u, v in query]

```

树论

倍增LCA

$f[u][i]$ 表示 u 节点向上跳 2^i 的节点, $dep[u]$ 表示深度

```

MX = int(n.bit_length())
f = [[0] * (MX + 1) for _ in range(n)]
dep = [0] * n

def dfs(u, fa):
    # father[u] = fa
    dep[u] = dep[fa] + 1    # 递归节点深度
    f[u][0] = fa
    for i in range(1, MX + 1):    # 倍增计算向上跳的位置
        f[u][i] = f[f[u][i - 1]][i - 1]
    for v in g[u]:
        if v != fa:
            dfs(v, u)

# 假定0节点是树根
dep[0] = 1
for v in g[0]:
    dfs(v, 0)

def lca(u, v):
    if dep[u] < dep[v]:
        u, v = v, u
    # u 跳到和v 同一层
    for i in range(MX, -1, -1):
        if dep[f[u][i]] >= dep[v]:
            u = f[u][i]
    if u == v:
        return u
    # 跳到lca的下一层
    for i in range(MX, -1, -1):
        if f[u][i] != f[v][i]:
            u, v = f[u][i], f[v][i]
    return f[u][0]

```

[P3379 【模板】最近公共祖先 \(LCA\) - 洛谷 | 计算机科学教育新生态 \(luogu.com.cn\)](#)

```

from math import log
import sys
input = lambda: sys.stdin.readline().strip()
n, m, s = map(int, input().split())

# f[n][mx]
mx = int(log(n, 2))
f = [[0] * (mx + 1) for _ in range(n + 10)]
e = [[] for _ in range(n + 10)]
dep = [0] * (n + 10)

```

```

dep[s] = 1

for _ in range(n - 1):
    u, v = map(int, input().split())
    e[u].append(v)
    e[v].append(u)

def dfs(u, fa):
    dep[u] = dep[fa] + 1
    f[u][0] = fa
    for i in range(1, mx + 1):
        f[u][i] = f[f[u][i - 1]][i - 1]
    for v in e[u]:
        if v != fa:
            dfs(v, u)
for v in e[s]:
    dfs(v, s)

def lca(u, v):
    # 让u 往上跳
    if dep[u] < dep[v]: u, v = v, u
    for i in range(mx, -1, -1):
        if dep[f[u][i]] >= dep[v]:
            u = f[u][i]
    if u == v: return u
    # 一定是在lca的下一层
    # 一起跳
    for i in range(mx, -1, -1):
        if f[u][i] != f[v][i]:
            u, v = f[u][i], f[v][i]
    return f[u][0]
for _ in range(m):
    a, b = map(int, input().split())
    print(lca(a, b))

```

树上差分

点差分：解决多路径节点计数问题。

$u \rightarrow v$ 的路径转化为 $u \rightarrow lca$ 左孩子 + $lca \rightarrow v$

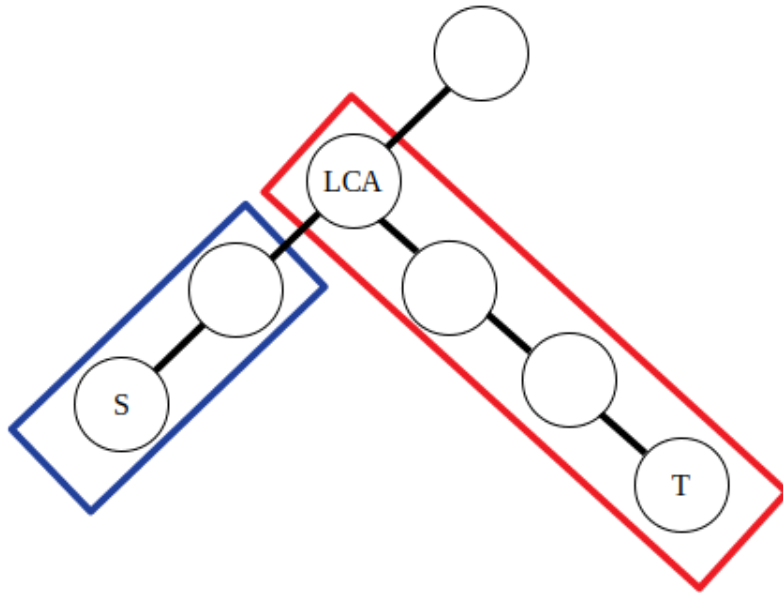
```

# 差分时左闭右开，无需考虑啊u = a的情况
for u, v in query:
    a = lca(u, v)
    diff[u] += 1
    diff[a] -= 1
    diff[v] += 1
    if father[a] != -1:
        diff[father[a]] -= 1

```

$$\begin{aligned}
 d_s &\leftarrow d_s + 1 \\
 d_{lca} &\leftarrow d_{lca} - 1 \\
 d_t &\leftarrow d_t + 1 \\
 d_{f(lca)} &\leftarrow d_{f(lca)} - 1
 \end{aligned}$$

其中 $f(x)$ 表示 x 的父亲节点, d_i 为点权 a_i 的差分数组。



树形DP(换根DP)

[834. 树中距离之和 - 力扣 \(LeetCode\)](#)

[题目详情 - Problem 4E. 最大社交深度和 - HydroOJ](#)

- 1, 指定某个节点为根节点。
- 2, 第一次搜索完成预处理（如子树大小等），同时得到该节点的解。
- 3, 第二次搜索进行换根的动态规划，由已知解的节点推出相连节点的解。

```
def sumOfDistancesInTree(self, n: int, edges: List[List[int]]) -> List[int]:
    g = [[] for _ in range(n)]
    dep = [0] * n
    siz = [1] * n
    res = [0] * n
    for u, v in edges:
        g[u].append(v)
        g[v].append(u)

    def dfs1(u, fa):
        # 预处理深度
        dep[u] = dep[fa] + 1 if fa != -1 else 0
        for v in g[u]:
            if v != fa:
                dfs1(v, u)
                siz[u] += siz[v]

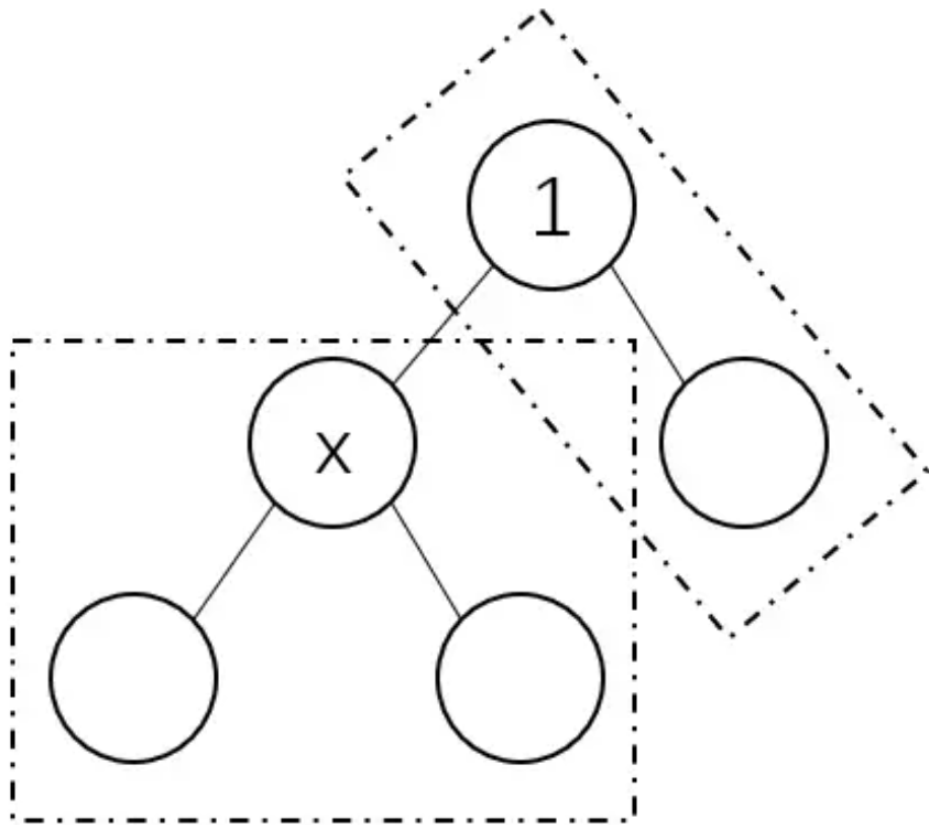
    def dfs2(u, fa):
        for v in g[u]:
            if v != fa:
                res[v] = res[u] - siz[v] + (n - siz[v])
                dfs2(v, u)

    dfs1(0, -1)
    res[0] = sum(dep)
    dfs2(0, -1)
    return res
```

u 剔除 v 子树部分下降1, 深度和增加 $n - siz[v]$

v 子树部分上升1, 深度和减少 $siz[v]$

则状态转移方程 $res[v] = res[u] - siz[v] + (n - siz[v])$



树上异或

性质1: 对树上一条路径 $u \rightarrow x_0 \rightarrow x_1 \rightarrow \dots \rightarrow v$ 进行相邻节点两两异或运算, 等价于只对路径起始节点和终止节点异或。

因而树上相邻异或 等价于 树上任意两点进行异或

性质2: 在树上任意相邻异或, 总是有偶数个节点被异或。

[3068. 最大节点价值之和 - 力扣 \(LeetCode\)](#)

```
class Solution:
    def maximumValueSum(self, nums: List[int], k: int, edges: List[List[int]]) -> int:
        res = sum(nums)
        delta = sorted([(x ^ k) - x for x in nums], reverse = True)
        for du, dv in zip(delta[:2], delta[1::2]):
            res = max(res, res + du + dv)
        return res
```

树上直径

时间复杂度: $O(n)$

定义: 树上任意两节点之间最长的简单路径即为树的「直径」。

定理:

- 对于无负边权的树, 从树的任意节点出发寻找到距离最远的节点, 一定是树直径的一个端点。 (反证)

方法一: 两次dfs

```
def treeDiameter(self, edges: List[List[int]]) -> int:
    n = len(edges) + 1
    e = [[] for _ in range(n + 1)]
    for u, v in edges:
        e[u].append(v)
        e[v].append(u)
```



```

def dfs(u, fa):
    res, mxv = 0, u
    for v in e[u]:
        if v == fa: continue
        a, b = dfs(v, u)
        if a + 1 > res:
            res, mxv = a + 1, b
    return res, mxv
_, s = dfs(0, -1)
res, _ = dfs(s, -1)
return res

```

方法二：树形DP

返回每个节点的最长路径fst 和 与最长路径没有公共边的次长路径 sec，取max(fst + sec)

```

def treeDiameter(self, edges: List[List[int]]) -> int:
    n = len(edges) + 1
    e = [[] for _ in range(n + 1)]
    for u, v in edges:
        e[u].append(v)
        e[v].append(u)
    res = 0
    def dfs(u, fa):
        nonlocal res
        # 找出节点u 为子树的最长 / 次长路径
        fst = sec = -1
        for v in e[u]:
            if v == fa: continue
            a, _ = dfs(v, u)
            if a >= fst:
                fst, sec = a, fst
            else:
                sec = max(a, sec)
        res = max(fst + sec + 2, res)
        return fst + 1, sec + 1
    dfs(0, -1)
    return res

```

[310. 最小高度树 - 力扣 \(LeetCode\)](#)

树的直径问题，最小高度树的根一定在树的直径上。

```

def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
    e = [[] for _ in range(n)]
    for u, v in edges:
        e[u].append(v)
        e[v].append(u)
    # 确定以x 为根
    pa = [-1] * n
    def dfs(u, fa):
        pa[u] = fa
        res, mxv = 0, u
        for v in e[u]:
            if v == fa:
                continue
            a, b = dfs(v, u)
            if a + 1 > res:
                res, mxv = a + 1, b
        return res, mxv
    _, x = dfs(0, -1)
    dis, y = dfs(x, -1)
    path = []
    while y != -1:
        path.append(y)
        y = pa[y]
    res = [path[dis // 2]]
    if dis & 1:
        res.append(path[dis // 2 + 1])
    return res

```

位运算

位运算与集合

集合 A, B , 最大二进制长度为 N

操作	位运算
全集	$(1 \ll N) - 1$
补集	$\sim A$
添加元素	$A (1 \ll i)$
删除元素	$A \& \sim (1 \ll i)$
删除元素 (一定在集合中)	$A \oplus (1 \ll i)$
属于 / 不属于	$(A \gg i) \& 1 = 1/0$
删除最小元素	$A \& (A - 1)$
差集	$A \& \sim B$
差集 (子集) / 对称差	$A \oplus B$

(1). 把b位置为1

通过 **或** 实现

```
mask |= 1 << b
```

(2). 把b位置清零

通过 **与非**实现

```
mask &= ~(1 << b)
```

(3). 获得一个数从高到低的每一位的值

[1261. 在受污染的二叉树中查找元素 - 力扣 \(LeetCode\)](#)

```
class FindElements:

    def __init__(self, root: Optional[TreeNode]):
        self.root = root

    def find(self, target: int) -> bool:
        target += 1
        node = self.root
        for b in range(target.bit_length() - 2, -1, -1):
            x = (target >> b) & 1
            node = node.right if x else node.left
            if not node: return False
        return True
```

二维矩阵 压缩为一维二进制串

```
num = sum((ch == '.') << i for i, ch in enumerate(s)) # 010110
```

满足 $num \gg x == s[i]$

```
s = ["#", ".", ".", "#", ".", "#"]
num = sum((ch == '.') << i for i, ch in enumerate(s)) # 010110
print(bin(num)) # 0b 010110
```

枚举一个二进制串的子集

```
s = 19
j = s
while j:
    # print(format(j, '06b'))
    j = (j - 1) & s
```

判断是否有两个连续（相邻）的1

```
(s & (s >> 1)) == 0 # 为True是表示没有两个连续的1
或者
(s & (s << 1)) == 0
```

十进制长度

```
m = int(log(n + 1, 10)) + 1
```

二进制长度

```
n = num.bit_lenght()
```

二进制中1的数量

```
cnt = num.bit_count()
```

十进制 int 转换 对应二进制的 int

```
def bin(x):
    res = 0
    i = 0
    while x:
        res = res + pow(10, i) * (x % 2)
        x >>= 1
        i += 1
    return res
```

##

最大异或

```
def findMaximumXOR(self, nums: List[int]) -> int:
    n = max(nums).bit_length()
    res = mask = 0
    for i in range(n - 1, -1, -1):
        mask |= 1 << i
        s, tmp = set(), res | (1 << i)
        for x in nums: # x ^ a = tmp -> a = tmp ^ x
            x &= mask
            if tmp ^ x in s:
                res = tmp
                break
            s.add(x)
    return res
```

拆位试填法

当发现题目要求所有元素按位运算得到的**最大值**问题时，从高位开始考虑是否能为1/0。

考虑过的状态记录在res中，不考虑的位用mask置为0表示。

```
mask = res = 0
for b in range(n, -1, -1):
    mask |= 1 << b # 蒙版
    for x in nums:
        x &= mask
    # 最大值 ...
    res |= 1 << b # 得到最大值
    mask &= ~(1 << b) # 该位自由，不用考虑
```

```
mask = res = 0
for b in range(n, -1, -1):
    mask |= 1 << b
    ans_res = -1    # 初始值全是1
    cnt = 0
    for x in nums:
        ans_res &= x & mask
        if ans_res > 0:
            cnt += 1
        else:
            ans_res = -1    # 重置初始值
    if cnt > k: # 说明这一位必然是1
        # mask这位蒙版就置为0，表示后续都不考虑这位
        mask &= ~(1 << b)
    res |= 1 << b
return res
```

动态规划

[划分型dp - 力扣 \(LeetCode\)](#)

[数位dp - 力扣 \(LeetCode\)](#)

[状压dp - 力扣 \(LeetCode\)](#)

[线性dp / 背包 - 力扣 \(LeetCode\)](#)

[状态机dp - 力扣 \(LeetCode\)](#)

[区间dp - 力扣 \(LeetCode\)](#)

背包问题

N 个物品，价值为 v_i ，重量为 w_i ，背包容量为 W 。挑选物品不超过背包容量下，总价值最大是多少。

- 0-1背包：每个物品用 0 或者 1 次。
- 完全背包：每个物品可以用 0 到 $+\infty$ 次。
- 多重背包：每个物品最多 s_i 次。
- 分组背包：物品分为若干组，每一组里面选 0 或者 1 次。

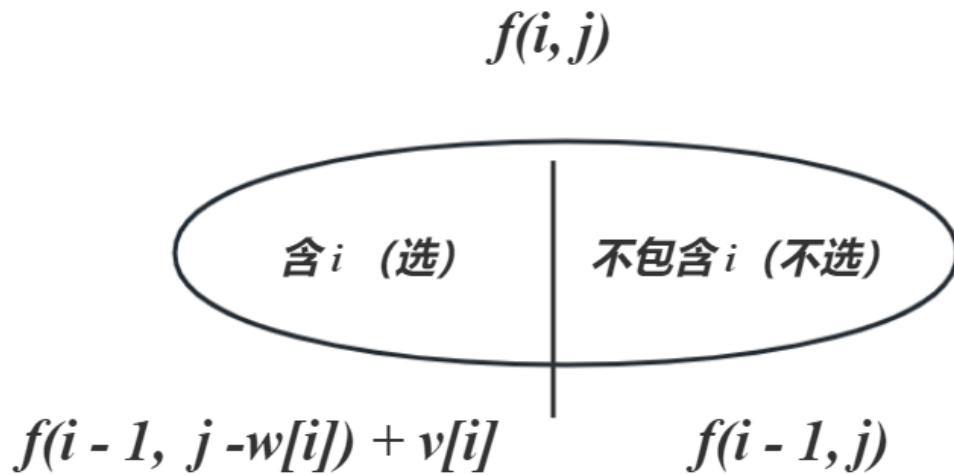
0-1 背包

状态表示： $f(i, j)$

- 集合：
 - 所有拿物品的选法
 - 条件：1. 只从前 i 个物品中选；2. 总重量 $\leq j$
- 表示的属性（一般是max, min, 个数）：所有选法的总价值的最大值（max）
最终求解的问题 $f(N, W)$ 。

状态计算：

集合的划分问题：如何将集合 $f(i, j)$ 划分成更小的可计算子集。



```
# f[i][j] 表示用前i 个物品，在总重量不超过j 的情况下，所有物品选法构成的集合中，总价值的最大值
# f[0][0] ~ f[N][0] = 0
# 考虑f[i][j] 对应集合的完备划分： 选i，其子集的最大值是f[i-1][j-w[i]] + v[i]，需要在 j-w[i] >= 0 满足
# 不选i，其子集的最大值是 f[i-1][j]。一定可以满足
for i in range(1, N+1):
    for j in range(W+1):
        f[i][j] = f[i-1][j]
        if j-w[i] >= 0:
            f[i][j] = max(f[i][j], f[i-1][j-w[i]] + v[i])
return f[N][W]
```

滚动数组优化为一维：逆序遍历

由于 $f(i, j)$ 只和 $f(i-1, j)$ 有关。如果使用滚动数组 $f(j)$ 优化，去掉第一维度，在同一个 i 下，如果正序遍历 j ，在恰好更新 $f(j)$ 前所有 $f(j' < j)$ 存放的是新值 $f(i, j')$ ，所有 $f(j'' \geq j)$ 存放的是老值 $f(i-1, j'')$ 。

由于我们希望能够得到 $f(i-1, j-w[i])$ ，所以必须逆序遍历 j ：在恰好更新 $f(j)$ 前， $f(j' \leq j)$ 都是老值，表示 $f(i-1, j')$ 。

所以 j 的枚举为 $\text{range}(W, w[i]-1, -1)$

```
f = [0] * (W+1)
for i in range(1, N+1):
    for j in range(W, w[i]-1, -1):
        f[j] = max(f[j], f[j-w[i]] + v[i])
    # 此时f[j] 就代表 f[i-1][j], f[j-w[i]] 代表f[i-1][j-w[i]]
return f[W]
```

[题目详情 - LC2431. 最大限度地提高购买水果的口味 - HydroOJ](#)

增加限制条件：不超过 k 次使用折扣券。注意， k 的遍历方向也是逆序。

```
def maxTastiness(self, price: List[int], tastiness: List[int], maxAmount: int, maxCoupons: int) -> int:
    # f[i][j][k] 从前i 个物品，不超过容量j 的情况下，不超过k张券的最大价值
    # f[i][j][k] = max(f[i-1][j][k], f[i-1][j-w][k] + v, f[i-1][j-w//2][k-1] + v)
    f = [[0] * (maxCoupons+1) for _ in range(maxAmount+1)]

    for w, v in zip(price, tastiness):
        for j in range(maxAmount, w//2-1, -1):
            for k in range(maxCoupons, -1, -1):
                if j-w >= 0:
                    f[j][k] = max(f[j][k], f[j-w][k] + v)
                if k >= 1:
                    f[j][k] = max(f[j][k], f[j-w//2][k-1] + v)
    return f[maxAmount][maxCoupons]
```

恰好装满型 0-1背包

2915. 和为目标值的最长子序列的长度 - 力扣 (LeetCode)

$f[i][j] = \max(f[i-1][j], f[i-1][j-w] + v)$, 第二个转移的条件是 $f[i-1][j-w] > 0$ 或者 $f[i-1][j-w] = 0$ 且 $w = j$

可以通过初始值修改, 将不合法的 $f[i][j]$ 置为 $-\infty$, 合法的 $f[i][j] \geq 0$ 。则初始值 $f[0][0] = 0$

得到二维版本:

```
def lengthOfLongestSubsequence(self, nums: List[int], target: int) -> int:
    # f[i][j] 表示从前i个数中, 和为j的子序列的所有选法构成的集合中, 子序列长度的最大值
    # f[n][target]
    # f[i][j] = max(f[i-1][j], f[i-1][j-w] + 1)
    n = len(nums)
    f = [[-inf] * (target + 1) for _ in range(n + 1)]
    f[0][0] = 0
    for i in range(1, n + 1):
        w = nums[i - 1]
        for j in range(target + 1):
            f[i][j] = f[i - 1][j]
            if j - w >= 0:
                f[i][j] = max(f[i][j], f[i - 1][j - w] + 1)
    return f[n][target] if f[n][target] >= 0 else -1
```

优化: j 的上界可以优化为 $\min(\text{重量前缀}, \text{target})$

```
def lengthOfLongestSubsequence(self, nums: List[int], target: int) -> int:
    f = [0] + [-inf] * target
    pre = 0
    for w in nums:
        pre += w
        for j in range(min(pre, target), w - 1, -1):
            f[j] = max(f[j], f[j - w] + 1)
    return f[target] if f[target] >= 0 else -1
```

完全背包

状态表示: $f(i, j)$ 同 0-1 背包。

状态计算: 对于集合的划分, 按照第 i 个物品选几个 $(0, 1, \dots)$ 划分。

朴素做法: $O(N \cdot W^2)$

```
for i in range(1, N + 1):
    for j in range(w + 1):
        for k in range(j // w[i] + 1):
            f[i][j] = max(f[i][j], f[i - 1][j - k * w[i]] + k * v[i])
return f[N][w]
```

冗余优化: $O(N \cdot W)$

可以发现后面一坨的最大值等价于 $f(i, j - w)$

$$\begin{aligned} f[i, j] &= \text{Max}(f[i-1, j], & f[i-1, j-w] + v, & f[i-1, j-2w] + 2v, & f[i-1, j-3w] + 3v, & \dots) \\ f[i, j-w] &= \text{Max}(& f[i-1, j-w], & f[i-1, j-2w] + v, & f[i-1, j-3w] + 2v, & \dots) \end{aligned}$$

所以 $f(i, j) = \max(f(i-1, j), f(i, j-w) + v[i])$,

```
for i in range(1, N + 1):
    for j in range(w + 1):
        f[i][j] = f[i - 1][j]
        if j - w[i] >= 0:
            f[i][j] = max(f[i][j], f[i][j - w[i]] + v[i])
            # f[i][j - w[i]] 包含了 f[i - 1][j - k * w[i]] 的部分 (k >= 1)
return f[N][w]
```

优化为一维

```
for i in range(1, N + 1):
    for j in range(w[i], w + 1):
        f[j] = max(f[j], f[j - w[i]] + v[i])
```

518. 零钱兑换 II - 力扣 (LeetCode)

求组合方案数

$$\begin{aligned} f[i, j] &= \sum (f[i-1, j], & f[i, j-c]) \\ f[i, j-c] &= \sum (f[i-1, j-c], & f[i, j-2 \cdot c]) \end{aligned}$$

```
def change(self, amount: int, coins: List[int]) -> int:
    # f[i][j] 表示 前i 个硬币凑出 j 的方案数
    # 状态表示: 从前i 个硬币中组合出j 的所有方案的集合
    # 属性: 个数
    # 转移: 对集合进行划分。
    # f[i][j] = f[i-1][j] + f[i][j-c]
    n = len(coins)
    f = [[0] * (amount + 1) for _ in range(n + 1)]
    # f[i][0] = 1
    for i in range(n + 1): f[i][0] = 1

    for i in range(1, n + 1):
        for j in range(1, amount + 1):
            c = coins[i - 1]
            f[i][j] = f[i-1][j]
            if j - c >= 0:
                f[i][j] += f[i][j - c]
    return f[n][amount]
```

优化成一维:

```
def change(self, amount: int, coins: List[int]) -> int:
    # f[i][j] = f[i-1][j] + f[i][j-c]
    n = len(coins)
    # 从前i 个中构成 j 的方案数
    f = [0] * (amount + 1)
    f[0] = 1
    for c in coins:
        for j in range(c, amount + 1):
            f[j] += f[j - c]
    return f[amount]
```

1449. 数位成本和为目标值的最大数字 - 力扣 (LeetCode)

每个数字有一个重量，可以无限选，问恰好重量为target的最大数字。（类似题目：长度最大的字典序最小串等）

先用完全背包模型求出最长长度，然后贪心的从9~1倒序遍历逆序构造。构造的条件是 $f[target - w] + 1 = f[target]$ ，即通过长度判断是否可以转移。

```
def largestNumber(self, cost: List[int], target: int) -> str:
    # 先求出能构成的最长数串
    # 每个物品重量w，价值为1，
    # f[i][j] 表示从前i 个物品中选法中，能够构成的最大价值
    # f[i][j] = max(f[i][j], f[i][j-w])
    f = [0] + [-inf] * target
    for w in cost:
        for j in range(w, target + 1):
            f[j] = max(f[j], f[j - w] + 1)
    mxl = f[target]
    if mxl <= 0: return '0'
    res = ''
    # 贪心的构造，从高位到低位尽可能构造
    for x in range(9, -1, -1):
        w = cost[x + 1]
        while target - w >= 0 and f[target] == f[target - w] + 1:
            res += str(x)
            target -= w
    return res
```

多重背包

在完全背包的基础上，增加每个物品最多选择选择的次数限制 $s[i]$ 。

暴力做法: $O(N \cdot W^2)$

```
for i in range(1, n + 1):
    for j in range(w + 1):
        for k in range(min(c[i] + 1, j // w[i] + 1)):
            f[i][j] = max(f[i][j], f[i - 1][j - k * w[i]] + k * v[i])
```

$$f(i, j) = \max(f(i - 1, j), f(i - 1, j - w) + v, \dots, f(i - 1, j - c \cdot w) + c \cdot v)$$
$$f(i, j - w) = \max(f(i - 1, j - w), \dots, f(i - 1, j - c \cdot w) + (c - 1) \cdot v, f(i - 1, j - (c + 1) \cdot w) + c \cdot v)$$

可以发现无法借助完全背包的方法进行优化。

二进制拆分为新的包裹: $O(N \cdot W \cdot \log(\sum W))$

思路：将每一件最多能选 c 个的物品拆分成若干个包裹，大小分别是 $1, 2, \dots, 2^k, c'$ ，例如 $c = 500$ ，拆分成 $1, 2, \dots, 128, 245$ ，可以证明这些数字可以枚举出 $0 \sim 500$ 之间的所有数。将这些包裹看出是新的物品，有其对应的新的重量和价值。

可以估算，总包裹的个数不超过 $N \cdot \log_2(\sum W)$ 。

```
w, v = [], []
for _ in range(N):
    ow, ov, oc = map(int, input().split())
    k = 1
    while oc >= k: # 例如10, 拆分成1, 2, 4和3
        w, v = w + [ow * k], v + [ov * k]
        oc -= k
        k <= 1
    if oc > 0:
        w, v = w + [ow * oc], v + [ov * oc]

f = [0] * (mxw + 1)
for w, v in zip(w, v):
    for j in range(mxw, w - 1, -1):
        f[j] = max(f[j], f[j - w] + v)
print(f[mxw])
```

分组背包

有 N 组物品，容量为 mxW 的背包，每组物品最多只能选其中一个。例如，水果（苹果，香蕉，橘子）只能选一个或者不选。

$f(i, j)$ 从前 i 组选，总重量不超过 j 的所有选法方案的价值和的最大值。

状态转移：第 i 组物品一个都不选 $f(i - 1, j)$ ，第 i 组物品选第 k 个 $f(i - 1, j - w[i][k]) + v[i][k]$

```
w, v = [[0] for _ in range(N + 1)], [[0] for _ in range(N + 1)]
for i in range(1, N + 1):
    k = int(input())
    for k in range(k):
        w, v = map(int, input().split())
        w[i], v[i] = w[i] + [w], v[i] + [v]

f = [0] * (mxw + 1)
for i in range(1, N + 1):
    for j in range(mxw, -1, -1):
        for k in range(len(w[i])):
            if j - w[i][k] >= 0:
                f[j] = max(f[j], f[j - w[i][k]] + v[i][k])
```

线性dp

最长上升子序列

$O(n^2)$ 做法， $f[i]$ 表示以 $nums[i]$ 结尾的所有上升子序列中最长的长度。


```
for i, x in enumerate(nums):
    for j in range(i):
        if nums[j] < x:
            f[i] = max(f[i], f[j] + 1)
```

$O(n \log n)$ 做法, $f[i]$ 表示长度为 i 的所有上升子序列中, 子序列末尾的最小值

正序遍历 $nums$ 中每一个数 x , 二分找出 x 在 f 中的插入位置 (恰好大于 x 的位置)。

```
# f[i] 表示长度为i 的子序列的末尾元素的最小值
f = []

# 找到恰好大于x的位置
def check(x, mid):
    return f[mid] >= x
for x in nums:
    lo, hi = 0, len(f)
    while lo < hi:
        mid = (lo + hi) >> 1
        if check(x, mid):
            hi = mid
        else:
            lo = mid + 1
    if lo >= len(f):
        f.append(x)
    else:
        f[lo] = x
```

最长公共子序列

$f[i][j]$ 表示从 $s[0 : i]$ 和 $s2[0 : j]$ 中的最长公共子序列

时间复杂度: $O(mn)$

可以证明: $f(i-1, j-1) + 1 \geq \max(f(i-1, j), f(i, j-1))$

```
#
# f[n][m]
f = [[0] * (m + 1) for _ in range(n + 1)]
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if s1[i - 1] == s2[j - 1]:
            f[i][j] = f[i - 1][j - 1] + 1
        else:
            f[i][j] = max(f[i - 1][j], f[i][j - 1])
```

编辑距离

```
def getEditDist(s1, s2):
    m, n = len(s1), len(s2)
    f = [[inf] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1): f[i][0] = i
    for i in range(1, n + 1): f[0][i] = i
    f[0][0] = 0
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            a = f[i - 1][j] + 1
            b = f[i][j - 1] + 1
            c = f[i - 1][j - 1] + (1 if s1[i - 1] != s2[j - 1] else 0)
            f[i][j] = min(a, b, c)
    return f[m][n]
```

区间dp

石子合并

[AcWing 282. 石子合并 - AcWing](#)

```
s = [0] * (n + 1)
f = [[0] * n for _ in range(n)]
for i in range(n):
    s[i + 1] = s[i] + nums[i]
for l in range(2, n + 1):
    for i in range(n + 1 - l):
        j = i + l - 1
        f[i][j] = inf
        for k in range(i, j):
            f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + s[j + 1] - s[i])
```

[312. 戳气球 - 力扣 \(LeetCode\)](#)

长度统一处理: 对于 $\text{length} = 1$, $f[i][i - 1]$ 是0, $f[j + 1][j]$ 也是0。等价于没有

对于 $\text{length} = 2$, $f[i][i + 1]$ 其中一项 $f[i][i - 1] + f[i + 1][i + 1] + \dots$, 因此和长度大于等于3统一。

```
def maxCoins(self, nums: List[int]) -> int:
    nums = [1] + nums + [1]
    n = len(nums)
    f = [[0] * n for _ in range(n)]
    for l in range(1, n - 1):
        for i in range(1, n - l):
            j = i + l - 1
            for k in range(i, j + 1):
                f[i][j] = max(f[i][j], f[i][k - 1] + f[k + 1][j] + nums[k] * nums[i - 1] * nums[j + 1])
    return f[1][n - 2]
```

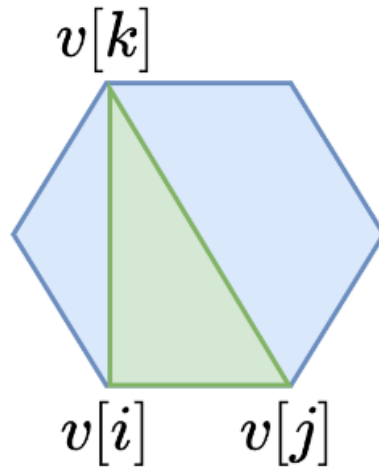
[375. 猜数字大小 II - 力扣 \(LeetCode\)](#)

$f[a, b]$ 表示从 $[a : b]$ 一定能获胜的最小金额。一定制胜的策略是当前位置一定答错, 同时选择左右两边较大区间

复杂度: $O(n^3)$

```
def getMoneyAmount(self, n: int) -> int:
    # f[a, b] 表示从 [a : b] 一定能获胜的最小金额
    # 最多取到 f[n + 1][n]
    f = [[0] * (n + 1) for _ in range(n + 2)]
    for l in range(2, n + 1):
        for i in range(1, n + 2 - l):
            j = i + l - 1
            f[i][j] = inf
            for k in range(i, j + 1):
                f[i][j] = min(f[i][j], k + max(f[i][k - 1], f[k + 1][j]))
    return f[1][n]
```

[1039. 多边形三角剖分的最低得分 - 力扣 \(LeetCode\)](#)



```
def minScoreTriangulation(self, values: List[int]) -> int:
    # f[i: j] 表示从[i: j] 的最小得分
    # f[0: n - 1]
    n = len(values)
    f = [[0] * (n + 1) for _ in range(n + 1)]
    for l in range(3, n + 1):
        for i in range(n + 1 - l):
            j = i + l - 1
            f[i][j] = inf
            for k in range(i + 1, j):
                f[i][j] = min(f[i][j], f[i][k] + f[k][j] + values[i] * values[k] * values[j])
    return f[0][n - 1]
```

95. 不同的二叉搜索树 II - 力扣 (LeetCode)

卡特兰数 + 区间dp, $f[i, j]$ 表示从 $i, i + 1, \dots, j$ 序列中构成的所有二叉搜索树的根节点 (对应的列表)。

最终问题: $f(1, n)$, 对于每个区间, 枚举中间节点 $k \in [i, j]$, 分别从左右子树对应的列表中 ($f(i, k - 1)$ 和 $f(k + 1, j)$), 利用乘法原理进行构造。

```
def generateTrees(self, n: int) -> List[Optional[TreeNode]]:
    # f[i, j] 表示用 1 .. j 构建的二叉搜索树的所有根节点列表
    # 枚举树根节点k in range(i, j + 1)
    # f[i, k - 1] 为所有左子树可能的根节点列表
    # f[k + 1, j] 为所有右子树可能的根节点列表
    f = [[None] for _ in range(n + 2)] for _ in range(n + 2)]
    for l in range(1, n + 1):
        for i in range(1, n + 2 - l):
            j = i + l - 1
            f[i][j] = []
            for k in range(i, j + 1):
                for left in f[i][k - 1]:
                    for right in f[k + 1][j]:
                        f[i][j].append(TreeNode(k, left, right))
    return f[1][n]
```

最长回文子序列

求最长回文子序列长度问题

$f[i: j]$ 表示 $s[i] \sim s[j]$ 中的最长回文子序列的长度

516. 最长回文子序列 - 力扣 (LeetCode)

```
def longestPalindromeSubseq(self, s: str) -> int:
    # f[i: j] 表示s[i] ~ s[j] 中的最长回文子序列的长度
    n = len(s)
    f = [[0] * (n + 1) for _ in range(n + 1)]
    for i in range(n):
        f[i][i] = 1
    for l in range(2, n + 1):
        for i in range(n + 1 - l):
```

```

        j = i + l - 1
        if s[i] == s[j]:
            f[i][j] = f[i + 1][j - 1] + 2
        else:
            f[i][j] = max(f[i + 1][j], f[i][j - 1])
    return f[0][n - 1]

```

推论：对于长度为 n 的字符串，其最长回文子序列长度为 L ，则最少添加 $n - L$ 个字符可以使原串变成回文串。

[1312. 让字符串成为回文串的最少插入次数 - 力扣 \(LeetCode\)](#)

[P1435 [OI2000](#)] [回文字串 - 洛谷 | 计算机科学教育新生态 \(luogu.com.cn\)](#)

```

def minInsertions(self, s: str) -> int:
    # f[i: j] 表示从s[i] ~ s[j] 的 最长回文子序列
    n = len(s)
    f = [[0] * (n + 1) for _ in range(n + 1)]
    for i in range(n):
        f[i][i] = 1
    for l in range(2, n + 1):
        for i in range(n + 1 - l):
            j = i + l - 1
            if s[i] == s[j]:
                f[i][j] = f[i + 1][j - 1] + 2
            else:
                f[i][j] = max(f[i + 1][j], f[i][j - 1])
    return n - f[0][n - 1]

```

最长回文子串

[5. 最长回文子串 - 力扣 \(LeetCode\)](#)

```

def longestPalindrome(self, s: str) -> str:
    # 定义f[i][j] 表示从 s[i] ~ s[j] 是否是回文字符串
    left = right = 0
    n = len(s)
    f = [[True] * (n + 1) for _ in range(n + 1)]
    for l in range(2, n + 1):
        for i in range(n + 1 - l):
            j = i + l - 1
            f[i][j] = s[i] == s[j] and f[i + 1][j - 1]
            if f[i][j]:
                left, right = i, j
    return s[left: right + 1]

```

数位dp

模板1：统计各位数字出现次数

统计在 $[a, b]$ 区间各个数字出现的次数。

需要实现 $count(n, x)$ 函数统计 $[1, n]$ 区间中数字 x 出现的次数

```

def count(n, x):
    # 在 1 ~ n 中x数字出现的次数
    # 上界 abcdefg
    # yyyzzzz , 考虑i位上x的出现次数
    #
    # 1.1如果x不为0 yyy 为 000 ~ abc - 1, zzz 为 000 ~ 999
    # 1.2x为0. yyy 为 001 ~ abc - 1, zzz 为 000 ~ 999
    #
    # 2. yyy 为 abc,
    #     2.1 d < x时, 0
    #     2.2 d = x 时, zzz为 000 ~ efg
    #     2.3 d > x 时, zzz为 000 ~ 999
    s = str(n)
    res = 0
    n = len(s)

```

```

for i in range(n):
    pre = 0 if i == 0 else int(s[:i])
    suf = s[i + 1:]
    if x == 0: res += (pre - 1) * pow(10, len(suf))
    else: res += pre * pow(10, len(suf))
    d = int(s[i])
    if d == x: res += (int(suf) if suf else 0) + 1
    elif d > x: res += pow(10, len(suf))
return res
def get(a, b):
    for i in range(10):
        print(count(b, i) - count(a - 1, i), end = ' ')
    print()

```

简化版:

```

def count(n, x): # 统计 1 ~ n 中 数字 x 的出现次数
    res = 0
    s = str(n)
    m = len(s)
    for i in range(m):
        pre = 0 if i == 0 else int(s[: i])
        d = int(s[i])
        sufs = s[i + 1: ]
        if x == 0: pre -= 1
        if d > x: pre += 1
        if d == x: res += (int(sufs) if sufs else 0) + 1
        res += pre * pow(10, len(sufs))
    return res
def get(a, b):
    for i in range(10):
        print(count(b, i) - count(a - 1, i), end = ' ')
    print()

```

[233. 数字 1 的个数 - 力扣 \(LeetCode\)](#)

```

def count(n, x): # 统计 1 ~ n 中 数字 x 的出现次数
    res = 0
    s = str(n)
    m = len(s)
    for i in range(m):
        pre = 0 if i == 0 else int(s[: i])
        d = int(s[i])
        sufs = s[i + 1: ]
        if x == 0: pre -= 1
        if d > x: pre += 1
        if d == x: res += (int(sufs) if sufs else 0) + 1
        res += pre * pow(10, len(sufs))
    return res
return count(n, 1)

```

模板2: 带限制数位dp 统计问题

通用模板 v1.0: 统计 $[1, n]$ 区间中, 符合限制条件的数字个数。

$f(i, mask, is_limit, is_num)$ 表示 前导数字集合为 $mask$, 从第 i 位开始往后填, 能满足限制条件的数字个数。

其中, is_limit 表示前导是否恰好全都取到上界。为 $True$ 时, i 的上界 $hi = int(s[i])$ 否则为 9;

is_num 表示前导是否有数字。为 $True$ 时, i 的下界从 0 开始; 否则可以继续不填数字, 或者下界从 1 开始。

```

@lru_cache(maxsize = None)
def f(i: int, mask: int, is_limit: bool, is_num: bool):
    if i == m:
        if is_num: return 1
        return 0
    res = 0
    lo, hi = 0, 9
    if not is_num:
        lo = 1
    res += f(i + 1, mask, False, False)

```

```

    if is_limit:
        hi = int(s[i])
    for j in range(lo, hi + 1):
        # j 没有在mask 的集合中出现过
        if (mask >> j) & 1 == 0:
            res += f(i + 1, mask | (1 << j), is_limit and j == hi, True)
    return res
return f(0, 0, True, False)

```

简化版本:

```

@lru_cache(None)
def f(i, mask, is_limit, is_num):
    if i == len(s): return int(is_num)
    res = 0 if is_num else f(i + 1, mask, False, False)
    lo, hi = 0 if is_num else 1, int(s[i]) if is_limit else 9
    for j in range(lo, hi + 1):
        if (mask >> j) & 1 == 0:
            res += f(i + 1, mask | (1 << j), is_limit and j == hi, True)
    return res

```

时间复杂度: 记 $D = 10$, 由于每个状态只会被计算一次, 每个状态的复杂度是 $O(D)$; 每一个 $(i, mask)$ 能够唯一确定 $(i, mask, is_limit, is_num)$ 四元组 (因此在记忆化的时候只需要 $(i, mask)$ 维度), 所以状态个数为 $m \cdot 2^D$, 其中 m 表示 n 的二进制长度。所以复杂度为: $O(D \cdot m \cdot 2^D)$

实际上某些问题中, is_num 可以被简化掉, 因为 $not(mask == 0)$ 和 is_num 是等价的。

[2376. 统计特殊整数 - 力扣 \(LeetCode\)](#)

统计 $1 \sim n$ 中各个数位都不相同的数字的个数。限制条件: $mask$ 前导中出现过的数字是不可以填的。

```

def countSpecialNumbers(self, n: int) -> int:
    s = str(n)
    @lru_cache(None)
    def f(i, mask, is_limit, is_num):
        if i == len(s): return int(is_num)
        res = 0 if is_num else f(i + 1, mask, False, False)
        lo, hi = 0 if is_num else 1, int(s[i]) if is_limit else 9
        for j in range(lo, hi + 1):
            if (mask >> j) & 1 == 0:
                res += f(i + 1, mask | (1 << j), is_limit and j == hi, True)
        return res
    return f(0, 0, True, False)

```

[788. 旋转数字 - 力扣 \(LeetCode\)](#)

统计区间: $1 \sim N$ 中的所有数字, 每个数位都被旋转。

限制条件: 旋转后不等于自身, 且合法的数字。只需要在数字中包含至少一个 $[2, 5, 6, 9]$ 且不包含 $[3, 4, 7]$ 。

```

def rotatedDigits(self, n: int) -> int:
    s = str(n)
    m = len(s)
    # 合法情况: 包含至少一个 [2, 5, 6, 9] 且 不包含 [3, 4, 7]
    nums = [0, 0, 1, -1, -1, 1, 1, -1, 0, 1]
    @lru_cache(None)
    def f(i, has_mir, is_limit, is_num):
        if i == m: return int(has_mir and is_num)
        res = 0 if is_num else f(i + 1, has_mir, False, False)
        lo, hi = 0 if is_num else 1, int(s[i]) if is_limit else 9
        for j in range(lo, hi + 1):
            if nums[j] != -1:
                res += f(i + 1, has_mir or nums[j] == 1, is_limit and j == hi, True)
        return res
    return f(0, False, True, False)

```

[902. 最大为 N 的数字组合 - 力扣 \(LeetCode\)](#)

```

def atMostNGivenDigitSet(self, digits: List[str], n: int) -> int:

```

```

s = str(n)
ss = set([int(ch) for ch in digits])
m = len(s)
@lru_cache(None)
def f(i, is_limit, is_num):
    if i == m: return int(is_num)
    res = 0 if is_num else f(i + 1, False, False)
    lo, hi = 0 if is_num else 1, int(s[i]) if is_limit else 9
    for j in range(lo, hi + 1):
        if j in ss:
            res += f(i + 1, is_limit and j == hi, True)
    return res
return f(0, True, False)

```

2827. 范围中美丽整数的数目 - 力扣 (LeetCode)

运用模运算的性质：整个数字模 k 的结果，比如 $1234 \bmod 17$ ，可以看成 $(1000 \bmod 17) + (200 \bmod 17) + (30 \bmod 17) + (4 \bmod 17)$ ，所以最后模数的结果只需要等价成不断 $mod_res \times 10 + j$ 即可。

```

def numberOfBeautifulIntegers(self, low: int, high: int, k: int) -> int:
    def cal(x):
        s = str(x)
        m = len(s)
        @lru_cache(None)
        def f(i, mod_res, odd_even_delta, is_limit, is_num):
            if i == m: return int(odd_even_delta == 0 and mod_res == 0 and is_num)
            res = 0 if is_num else f(i + 1, mod_res, odd_even_delta, False, False)
            lo, hi = 0 if is_num else 1, int(s[i]) if is_limit else 9
            for j in range(lo, hi + 1):
                res += f(i + 1, (mod_res * 10 + j) % k, odd_even_delta + (1 if j & 1 else -1),
                    is_limit and j == hi, True)
            return res
        return f(0, 0, 0, True, False)
    return cal(high) - cal(low - 1)

```

600. 不含连续1的非负整数 - 力扣 (LeetCode)

二进制数位dp。上界改为1

```

def findIntegers(self, n: int) -> int:
    def get_bin(x):
        res = i = 0
        while x:
            res = res + pow(10, i) * (x % 2)
            i += 1
            x >>= 1
        return res
    n = get_bin(n)
    s = str(n)
    m = len(s)
    @lru_cache(None)
    def f(i, pre, is_limit, is_num):
        if i == m: return int(is_num)
        res = 0 if is_num else f(i + 1, None, False, False)
        lo, hi = 0 if is_num else 1, int(s[i]) if is_limit else 1
        for j in range(lo, hi + 1):
            if pre == None or (j == 1 and pre != 1) or j == 0:
                res += f(i + 1, j, is_limit and j == hi, True)
        return res
    return f(0, None, True, False) + 1

```

状态机dp

3068. 最大节点价值之和 - 力扣 (LeetCode)

0 表示当前异或偶数个k，1表示当前异或奇数个k

$0 \rightarrow 0$ 或者 $1 \rightarrow 1$: 加上 x

$0 \rightarrow 1$ 或者 $1 \rightarrow 0$: 加上 $x \oplus k$

```
def maximumValueSum(self, nums: List[int], k: int, edges: List[List[int]]) -> int:
    n = len(nums)
    dp = [[0] * 2 for _ in range(n + 1)]
    dp[n][1] = -inf
    for i, x in enumerate(nums):
        dp[i][0] = max(dp[i - 1][0] + x, dp[i - 1][1] + (x ^ k))
        dp[i][1] = max(dp[i - 1][1] + x, dp[i - 1][0] + (x ^ k))
    return dp[n - 1][0]
```

状压dp / 状态压缩dp

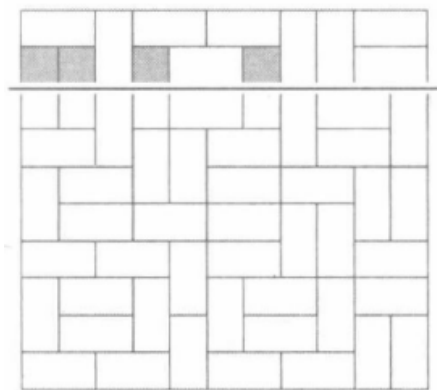
291. 蒙德里安的梦想 - AcWing题库

竖方块摆放确定时，横方块摆放一定确定（合法或者恰好填充），所以只需要看竖方块的摆放情况。对于 $N \times M$ 的方格， $f(i, j)$ 第 i 行形态为 j 时，前 i 行合法切割方案数。 j 是用十进制记录 M 位二进制数，其每位数字为1表示放竖方块上半部分，为0表示其他情况。（竖方块下半部分 / 横方块）

$f(i, j)$ 能由 $f(i - 1, k)$ 状态转移的充要条件：1. $j \& k = 0$ ，保证同列上下两行不会同时放竖方块的上半部分。2. $j \mid k$ 的所有连续的0的个数必须是偶数。 $j \mid k$ 为0当且仅当上下两行都是横方块，所以必须是偶数个。

初始状态对于 $f(0)$ ，不能对第一行产生影响，所以只有 $f(0, 0) = 1$ ，其余为0。最终目标： $f(N, 0)$ ，状态转移方程： $f(i, j) = \sum \text{valid}(f(i - 1, k))$

对于所有 M 位二进制数，预处理其是否满足所有连续0的个数是否是偶数。



```
N = M = 11
f = [[0] * (1 << M + 1) for _ in range(N + 1)]
def solve(n, m):
    # f[n][1 << m]
    # 预处理，判断 i 是否含有连续的奇数个0
    s = set()
    for i in range(1 << m):
        c = 0
        for j in range(m):
            if i >> j & 1:
                if c & 1: break
                else: c += 1
            if c & 1: s.add(i)
        f[0][0] = 1
    for i in range(1, n + 1):
        for j in range(1 << m):
            f[i][j] = 0
            for k in range(1 << m):
                if (j & k == 0 and (j | k not in s)):
                    f[i][j] += f[i - 1][k]
    return f[n][0]
```

最短哈密顿回路 / 旅行商问题

哈密顿回路：无向带权图中经过所有顶点的回路。朴素做法对于 N 个顶点，时间复杂度为 $O(n!)$ ，是 $NP - hard$ 问题（无法在多项式时间复杂度内求解）。

实际上, 设已经访问过的点集 S , 当前节点 j , 设 $f(S, j)$ 表示路径已经访问过点集 S 中的点且当前访问的 j 时的最短路径。有状态转移: $f(S, j) = \min\{f(S - j, k) + w(k, j), \forall k \in S - j\}$ 。可以用二进制来压缩已经访问的点集 S 。最终问题 $f(2^N - 1, N - 1)$, 初始值 $f(0, 0) = 0$ 。

```
def solve():
    n = int(input())
    f = [[inf] * n for _ in range(1 << n)]
    w = []
    for _ in range(n):
        w.append(list(map(int, input().split())))
    f[1][0] = 0
    for s in range(1, 1 << n):
        for j in range(n):
            if (s >> j) & 1: # j 在 s 中,
                for k in range(n):
                    if ((s ^ (1 << j)) >> k) & 1: # 且 k 在 s - j 中
                        f[s][j] = min(f[s ^ (1 << j)][k] + w[k][j], f[s][j])
    return f[(1 << n) - 1][n - 1]
```

贪心

多维贪心 + 排序

[406. 根据身高重建队列 - 力扣 \(LeetCode\)](#)

贪心: 先按照身高, 从大到小排序;

同身高内, 按照 k 从小到大排序

前缀性质: 任何一个 p 的前面的所有的 h 一定比自己大

```
def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
    # [7, 0] [7, 1] [6, 1] [5, 0] [5, 2] [4, 4]
    people.sort(key = lambda x: -x[0] * 10 ** 5 + x[1])
    res = []
    for i, p in enumerate(people):
        h, k = p[0], p[1]
        if k == i:
            res.append(p)
        elif k < i:
            res.insert(k, p)
    return res
```

反悔贪心

1. 反悔堆

- 贪心: 尽可能
- 反悔堆
- 反悔条件: 不满足原条件

[630. 课程表 III - 力扣 \(LeetCode\)](#)

反悔贪心: 按照截止日期排序, 尽可能不跳过每一个课程。反悔条件 ($cur > y$) 满足时从反悔堆反悔用时最大的课程。

```
def scheduleCourse(self, courses: List[List[int]]) -> int:
    # 按照截止日期排序
    courses.sort(key = lambda x: x[1])
    hq = []
    res, cur = 0, 0
    for x, y in courses:
        cur += x # 贪心: 尽可能不跳过每一个课程
        heapq.heappush(hq, -x) # 反悔堆: 存放所有课程耗时
        if cur > y: # 反悔条件: 超过截止日期
            cur -= heapq.heappop(hq)
    else:
        res += 1
    return res
```

[LCP 30. 魔塔游戏 - 力扣 \(LeetCode\)](#)

```
def magicTower(self, nums: List[int]) -> int:
    if sum(nums) + 1 <= 0:
        return -1
    hq = []
    res, cur = 0, 1
    for x in nums:
        cur += x    # 贪心：尽可能不使用移动
        if x < 0:    # 反悔堆
            heapq.heappush(hq, x)
        if cur <= 0:    # 反悔条件：血量不是正值
            res += 1
            cur -= heapq.heappop(hq) # 从反悔堆中，贪心回复血量
    return res
```

[1642. 可以到达的最远建筑 - 力扣 \(LeetCode\)](#)

```
def furthestBuilding(self, heights: List[int], bricks: int, ladders: int) -> int:
    n = len(heights)
    d = [max(0, heights[i] - heights[i - 1]) for i in range(1, n)]
    hq = []
    for res, x in enumerate(d):
        # ladders - len(hq) 代表剩余梯子数量
        heapq.heappush(hq, x)    # 贪心 + 反悔堆
        if ladders - len(hq) < 0: # 反悔条件：梯子不够了
            bricks -= heapq.heappop(hq)
        if bricks < 0:
            return res
    return n - 1
```

[871. 最低加油次数 - 力扣 \(LeetCode\)](#)

循环反悔贪心 + 反悔堆后置（需要贪心完成后才能加入当前值）

```
def minRefuelStops(self, target: int, startFuel: int, stations: List[List[int]]) -> int:
    stations.append([target, 0])
    n = len(stations)
    pre = 0
    res, cur = 0, startFuel
    hq = []
    for x, y in stations:
        cur -= x - pre # 贪心：尽可能耗油不加油
        pre = x
        while hq and cur < 0: # 反悔条件：剩余油不够了
            res += 1
            cur -= heapq.heappop(hq)
        if cur < 0 and not hq:
            return -1
        heapq.heappush(hq, -y) # 反悔堆：保存没加的油
    return res
```

2. 尝试反悔 + 反悔栈

也是一个二维贪心问题。尽可能优先考虑利润维度。

[2813. 子序列最大优雅度 - 力扣 \(LeetCode\)](#)

```
def findMaximumElegance(self, items: List[List[int]], k: int) -> int:
    items.sort(reverse = True)
    s = set()    # 只出现一次的种类 c
    stk = []     # 反悔栈：出现两次以上的利润 p
    res = total_profit = 0
    for i, (p, c) in enumerate(items):
        if i < k:
            total_profit += p
            if c not in s:    # 种类c首次出现，对应p一定最大，一定保留
                s.add(c)
            else:
                stk.append(p)    # 反悔栈：存放第二次及以后出现的更小的p
```

```

elif stk and c not in s:
    # 只有c没有出现在s中时，才尝试反悔一个出现两次及以上的p
    total_profit += p - stk.pop()
    s.add(c)
    # 贪心：s的长度只增不减
res = max(res, total_profit + len(s) ** 2)
return res

```

贡献法

经典问题：子数组的最小值之和，子数组的最大值之和，子数组的极差之和。

1. 套娃式定义，如子数组的子数组，子序列的子序列
2. 求某些的和，可以考虑成子问题对总问题的贡献

[2104. 子数组范围和 - 力扣 \(LeetCode\)](#)

考虑每个值对子数组最大值，最小值的贡献情况，用单调栈维护。

最大值用减小栈维护，贡献是 $(i - t) \times (t - stk[-1]) \times nums[t]$

```

def subArrayRanges(self, nums: List[int]) -> int:
    res = 0
    stk = [-1]
    total_mx = 0    # 贡献
    nums.append(-inf)
    for i, x in enumerate(nums):
        # 单调减
        while len(stk) > 1 and x >= nums[stk[-1]]:
            t = stk.pop()
            total_mx += (i - t) * (t - stk[-1]) * nums[t]
        stk.append(i)
    stk = [-1]
    nums[-1] = -inf
    total_mn = 0
    for i, x in enumerate(nums):
        # 单调增
        while len(stk) > 1 and x <= nums[stk[-1]]:
            t = stk.pop()
            total_mn += (i - t) * (t - stk[-1]) * nums[t]
        stk.append(i)
    return total_mx - total_mn

```

计算几何

旋转与向量

将点 (x, y) 顺时针旋转 α 后，新的点坐标为 $(x \cos \alpha + y \sin \alpha, y \cos \alpha - x \sin \alpha)$

证明：

点 $P(x, y)$ 表示为半径为 r ，极角为 θ 的坐标系下，
$$\begin{cases} x = r \cos \theta \\ y = r \sin \theta \end{cases}$$

顺时针旋转 α 后，
$$\begin{cases} x' = r \cos(\theta - \alpha) = x \cos \alpha + y \sin \alpha \\ y' = r \sin(\theta - \alpha) = y \cos \alpha - x \sin \alpha \end{cases}$$

距离

$A(x_1, y_1), B(x_2, y_2)$

曼哈顿距离 = $|x_1 - x_2| + |y_1 - y_2|$

切比雪夫距离 = $\max(|x_1 - x_2|, |y_1 - y_2|)$

曼哈顿距离转切比雪夫

即将所有点顺时针旋转45°后再乘 $\sqrt{2}$ 。

将 $P(x, y)$ 映射到 $P'(x + y, x - y)$ 坐标系下, $d_M = d'_Q$

对于三维点 $P(x, y, z)$ 映射到 $P''(x + y + z, -x + y + z, x - y + z, x + y - z)$ 坐标系下, $d_M = d''_Q$

当要求若干点之间的最大 d_M 时, 可以转换为

$$\begin{aligned} \forall i, j \in P, \max(|x_i - x_j| + |y_i - y_j|) &\iff \max(\max(|x'_i - x'_j|, |y'_i - y'_j|)) \\ &\iff \forall i, j \in P, \max(\max(|x'_i - x'_j|), \max(|y'_i - y'_j|)) \end{aligned}$$

[3102. 最小化曼哈顿距离 - 力扣 \(LeetCode\)](#)

```
from sortedcontainers import SortedList
class Solution:
    def minimumDistance(self, points: List[List[int]]) -> int:
        msx, msy = SortedList(), SortedList()
        for x, y in points:
            msx.add(x + y)
            msy.add(x - y)
        res = inf
        for x, y in points:
            msx.remove(x + y)
            msy.remove(x - y)
            xmx = msx[-1] - msx[0]
            ymx = msy[-1] - msy[0]
            res = min(res, max(xmx, ymx))
            msx.add(x + y)
            msy.add(x - y)
        return res
```

切比雪夫转曼哈顿距离

将 $P(x, y)$ 映射到 $P'(\frac{x+y}{2}, \frac{x-y}{2})$ 坐标系下, $d_Q = d'_M$

切比雪夫距离在计算的时候需要取max, 往往不是很好优化, 对于一个点, 计算其他点到该的距离的复杂度为O(n)

而曼哈顿距离只有求和以及取绝对值两种运算, 我们把坐标排序后可以去掉绝对值的影响, 进而用前缀和优化, 可以把复杂度降为O(1)

[P3964 TJOI2013] 松鼠聚会 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

转换成切比雪夫距离。将x, y 分离, 前缀和维护到各个xi 和 yi 的距离和, 再相加

```
def solve():
    n = int(input())
    points = []
    res = inf
    for _ in range(n):
        x, y = map(int, input().split())
        points.append(((x + y) / 2, (x - y) / 2))
    numsx = [p[0] for p in points]
    numsy = [p[1] for p in points]
    def g(nums):
        nums.sort()
        curx = nums[0]
        curd = sum(nums[i] - curx for i in range(1, n))
        dic = {nums[0]: curd}
        for i in range(1, n):
            x = nums[i]
            d = x - curx
            curd = curd + i * d - (n - i) * d
            dic[x] = curd
            curx = x
        return dic
    dicx, dicy = g(numsx), g(numsy)
    for x, y in points:
        ans = dicx[x] + dicy[y]
        res = min(res, ans)
```

```
print(int(res))
```