常用数据结构API

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

KMP / 模式匹配

暴力匹配所有起始位置

时间复杂度：$O(mn)$

```
    for i in range(len_s - len_p + 1):
        ii, j = i, 0
        while j < len_p:
            if s[ii] == p[j]: ii, j = ii + 1, j + 1
            else: break
        if j == len_p: res.append(i)
```

前缀函数 / next数组

时间复杂度：$O(n)$，在线算法

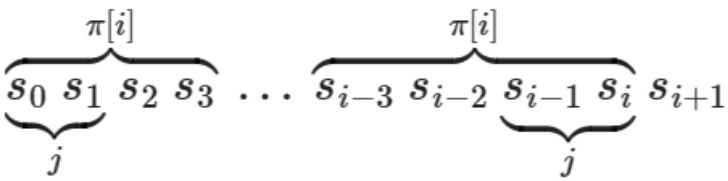
对于一个长度为 n 的字符串，其前缀函数是一个长度为 n 的数组 π，其中 $\pi(i)$ 定义：子串 $s[0]\sim s[i]$ 中存在的、相等的最长真前缀和真后缀的长度。如果不存在则为0。规定：$\pi[0]=0$，因为其不存在真前后缀。

> 例如：'aabaaab' 的 π 数组为 [0, 1, 0, 1, 2, 2, 3]

求解前缀函数：

- 相邻的前缀函数值，至多 + 1。$\pi(i-1)$表示着前一个状态匹配的最长真前后缀，也是下一个待匹配真前缀的最右元素下标。当且仅当 $s[i]=s[\pi(i-1)]$，有$\pi(i)=\pi(i-1)+1$。
- 考虑 $s[i]\neq s[\pi(i-1)]$，失配时，希望找到 $s[0]\sim s[i-1]$ 中，仅次于 $\pi[i-1]$ 的第二长度 j，使得在位置 $i-1$ 的前后缀性质仍然保持，即 $s[0]\sim s[j-1]=s[i-j]\sim s[i-1]$。

实际上，第二长真后缀也完整存在于 当前真前缀 $s[0]\sim s[j-1]$ 中，即有转移方程：$j^{(n-1)}=\pi(j^n-1)$。所以如此往复，要么直到 $s[i]=s[j']$，然后转移到第一种情况；要么直到 $j'=0$。两种情况，通过判断 $s[i]$ 是否 $s[j']$ 来确定要不要让 $j'+1$ 统一，最后 $s[i]=s[j']$。




```
def get_pi(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
        j = pi[i - 1]
        while j > 0 and s[i] != s[j]:
            j = pi[j - 1]
        if s[i] == s[j]: j += 1
        pi[i] = j
    return pi
```

KMP算法：找出 p 在 s 中的所有出现

时间复杂度：$O(n+m)$，其中 $m=len(p)$，$n=len(s)$

构造字符串 $t=p\#s$，计算其前缀函数 π。考虑前缀函数 $\pi[m+1]\sim\pi[n+m]$，其中 $\pi(i)=m$ 的地方，一定完成对模式串 p 的匹配。此时，i 位于 t 中s 的最后位置，所以原始位置为 $i-m+1-m-1=i-2*m$。

```
def kmp(p, s):
    res = []
    m, n = len(p), len(s)
    pi = get_pi(p + '#' + s)
    for i in range(m + 1, len(pi)):
        if pi[i] == m: res.append(i - 2 * m)
    return res
```

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

$$z[i]=len(LCP(s,s[i:]))$$

> 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 = "abcacb", p = "ab", 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
```

字符串哈希

49. 字母异位词分组 - 力扣（LeetCode）

2430. 对字母串可执行的最大删除数 - 力扣（LeetCode）

字符串API

- s1.startswith(s2, beg = 0, end = 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[beg: 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}
```

区间问题

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)
```

452. 用最少数量的箭引爆气球 - 力扣（LeetCode）

等价于区间选点问题：905. 区间选点 - AcWing题库

首先按照左端点排序。实际上，当前区间左端点是不需要维护的，因为选点总是贪心的放在区间右端点上。

当且仅当新区间左端点 il 大于当前区间右端点 r 时，需要新的选点，同时产生新的区间右端点（区间左端点介于 旧r 和 新 il 之间，但是我们并不关心）；否则说明左端点在当前区间内部，只需要更新区间右端点为更小的即可（意味着选点跟着移动）。

```
def findMinArrowShots(self, nums: List[List[int]]) -> int:
        nums.sort()
        r = nums[0][1]
        res = 0
        for il, ir in nums:
            if il > r:
                res += 1
                r = ir
            else:
                r = min(r, ir)
        return res + 1
```

区间交集

Problem - C - Codeforces

Lo, Hi 记录当前可变温度区间。每次来到新时刻，更新为 $[Lo-dt,\ Hi+dt]$。判断该区间是否和当前 $[lo, hi]$ 相交。是则求其交集。

```
def solve():
    n, m = map(int, input().split())
    tem = [(0, m, m)]
    for _ in range(n):
        at, lo, hi = map(int, input().split())
        tem.append((at, lo, hi))
    Lo = Hi = m
    for i in range(1, n + 1):
        at, lo, hi = tem[i]
        dt = at - tem[i - 1][0]

        Lo, Hi = Lo - dt, Hi + dt
        if Lo > hi or Hi < lo: return 'NO'
        Lo, Hi = max(Lo, lo), min(Hi, hi)
    return 'YES'
```

回溯 / 递归 / dfs

子集型回溯

枚举子集，$O(n\cdot 2^n)$

78. 子集 - 力扣（LeetCode）

回溯方法1：选 / 不选

```
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        res, path = [], []
        def dfs(i):
            if i == n:
                res.append(path.copy())
                return
            path.append(nums[i])
            dfs(i + 1)
            path.pop()
            dfs(i + 1)
        dfs(0)
        return res
```

方回溯法2：枚举选哪个数 + 记录可以选的范围

$dfs(i)$ 表示当前已经有选择了 $path$ 后，$path$ 下一个元素可以从 i 及其往后选。每一个 dfs 状态都是合法状态，需要记录。

```
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res, path = [], []
        n = len(nums)
        def dfs(i):
            res.append(path.copy())
            for j in range(i, n):
                path.append(nums[j])
                dfs(j + 1)
                path.pop()
        dfs(0)
        return res
```

位运算写法：

```python
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        s = (1 << n) - 1
        res = [[]]
        sub = s
        while sub:
            res.append([nums[j] for j in range(n) if ((sub >> j) & 1)])
            sub = (sub - 1) & s
        return res
```

组合型回溯

枚举所有长度为 k 的组合

77. 组合 - 力扣（LeetCode）

回溯方法1：选 / 不选方法

> 时间复杂度：$O(n \cdot 2^n)$

```python
    def combine(self, n: int, k: int) -> List[List[int]]:
        nums = list(range(1, n + 1))
        res, path = [], []
        def dfs(i):
            if i == n:
                if len(path) == k: res.append(path.copy())
                return
            # 不选
            dfs(i + 1)
            # 选
            path.append(nums[i])
            dfs(i + 1)
            path.pop()
        dfs(0)
        return res
```

回溯方法2：枚举当前选哪个数，以及记录可以选择的范围，每一个状态的合法情况需要记录。

剪枝操作 (1)：确保所有子集长度不会超过 k。剪枝操作 (2)：要确保枚举当前选择的数的位置，不会使得最终整个子集长度达不到 k。通过两个剪枝操作，确保只会得到长度恰好为 k 的子集。这里倒序 / 正序枚举在当前选择的数的范围影响下界 / 上界。

倒序枚举时，$dfs(i)$ 表示当前选择范围为 $nums[0] \sim nums[i]$，含有 $i+1$ 个数。由于下一个状态是 $j-1$，含有j 个数，根据 $j \geq k - len(path) - 1$ 计算下界。

> 时间复杂度：$O(k \cdot C(n,k))$。因为总共组合状态个数有 $C(n,k)$个，每个状态记录的长度不超过k。

```python
# 倒叙枚举
    def combine(self, n: int, k: int) -> List[List[int]]:
        nums = list(range(1, n + 1))
        res, path = [], []
        def dfs(i):
            if k == len(path):
                res.append(path.copy())
                return
            for j in range(i, k - len(path) - 2, -1):
                path.append(nums[j])
                dfs(j - 1)
                path.pop()
        dfs(n - 1)
        return res
```

位运算写法 + Gosper's Hack：

> 时间复杂度：$O(n \cdot C(n,k))$

```python
    def combine(self, n: int, k: int) -> List[List[int]]:
        nums = list(range(1, n + 1))
        s = (1 << n) - 1
        sub = (1 << k) - 1
```

```
        res = []
        def next_sub(x):
            lb = x & -x
            left = x + lb
            right = ((left ^ x) >> 2) // lb
            return left | right
        while sub <= s:
            res.append([nums[i] for i in range(n) if (sub >> i) & 1])
            sub = next_sub(sub)
        return res
```

完全背包型组合

每个元素可以无限重复选择，需要找出目标值等于 $target$ （或小于等于 $target$ ）的所有可行组合。

先排序，利于提前剪枝优化跳出循环。枚举当前选哪个 + 记录可以选择的范围型回溯，记录当前的和。由于可以重复选择，所以当前选择 j 以后，下一次的可以选择范围仍然是 j。

39. 组合总和 - 力扣（LeetCode）

```
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        candidates.sort()
        n, res, path = len(candidates), [], []
        # 枚举当前选哪个，以及记录可以选择的范围；以及当前的和
        def dfs(i, s):
            if s == target:
                res.append(path.copy())
                return
            for j in range(i, n):
                x = candidates[j]
                if x + s <= target:
                    path.append(x)
                    dfs(j, s + x)   # 体现可重复选择
                    path.pop()
                else: break
        dfs(0, 0)
        return res
```

括号生成问题：带限制组合型回溯

22. 括号生成 - 力扣（LeetCode）

选 / 不选型回溯：枚举当前左 / 右括号，记录当前左括号的个数。记 lc 表示左括号个数， rc 表示右括号个数。

限制1：lc 不能小于 rc。

限制2：lc 不能超过 $n/2$。

限制3：当 $lc = rc$，只能回溯左括号。

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        # 枚举当前左 / 右括号，记录当前左括号的个数
        n <<= 1
        path = [None] * n
        res = []
        def dfs(i, lc):
            rc = i - lc
            if rc > lc or lc > n // 2: return
            if i == n:
                res.append(''.join(path))
                return
            path[i] = '('
            dfs(i + 1, lc + 1)
            if rc < lc:  # 可以选右
                path[i] = ')'
                dfs(i + 1, lc)
        dfs(0, 0)
        return res
```

> 时间复杂度：由于状态个数是卡特兰数列，即 $O(C_n) \sim O(\frac{4^n}{n^{\frac{3}{2}} \cdot \sqrt{\pi}})$

排列型回溯

全排列：排列元素无重复

46. 全排列 - 力扣（LeetCode）

写法1：$dfs(i, S)$ 表示枚举到 第 i 位，没有枚举过的集合为 S。外层 $path$ 表示当前回溯的路径。

其中 $path$ 可以使用 $path[i] = j$ 的写法，覆盖当前走到哪一步；也可以使用 $append/pop$ 写法，覆盖和恢复现场。但是不可以在外层增加哈希集合维护没有枚举过的，这是因为集合添加操作的乱序性，外层的集合无法正确恢复现场（恢复后遍历顺序不正确）。

时间复杂度：当有 N 个数时，所有状态个数 $M = A_N^N + A_N^{N-1} + \cdots + A_N^0 = \sum_{k=0}^{N} \frac{N!}{k!} = N! \cdot \sum_{k=0}^{N} \frac{1}{k!} = e \cdot N!$。（麦克劳林展开）。每个状态时间复杂度，可以将集合的复制下方到下一个状态，所以是 $O(n)$。故总复杂度：$O(N \cdot N!)$

```python
    def permute(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        path = [0] * n
        res = []
        # 当前枚举到 位置 i，没有枚举过的集合为 S
        def dfs(i, S):
            if i == n:
                res.append(path.copy())
                return
            for j in S:
                path[i] = j
                dfs(i + 1, S - {j})
        dfs(0, set(nums))
        return res
```

写法2：更偏向于回溯。外层$path$ 表示当前回溯的路径，外层 on_path 维护节点是否已经出现在回溯的路径中。

```python
    def permute(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        path = []
        on_path = [False] * n
        res = []
        # 当前枚举到 位置 i，on_path 记录是否已经出现在回溯路径path中
        def dfs(i):
            if i == n:
                res.append(path.copy())
                return
            for pj, on in enumerate(on_path):
                if not on:
                    on_path[pj] = True
                    path.append(nums[pj])
                    dfs(i + 1)
                    on_path[pj] = False
                    path.pop()
        dfs(0)
        return res
```

全排列：排列元素有重复：只能用on_path 回溯 / 位运算压缩

47. 全排列 II - 力扣（LeetCode）

相同元素，在 i处视为一个，加一个集合维护已经出现过的数字。

```python
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        n, res = len(nums), []
        path, on_path = [0] * n, [0] * n
        def dfs(i):
            if i == n:
                res.append(path.copy())
                return
            S = set()       # 相同元素，在i 处视为一个
            for j, on in enumerate(on_path):
                if not on and nums[j] not in S:
                    S.add(nums[j])
```

```
                    path[i] = nums[j]
                    on_path[j] = 1
                    dfs(i + 1)
                    on_path[j] = 0
        dfs(0)
        return res
```

[2850. 将石头分散到网格图的最少移动次数 - 力扣（LeetCode）](#)

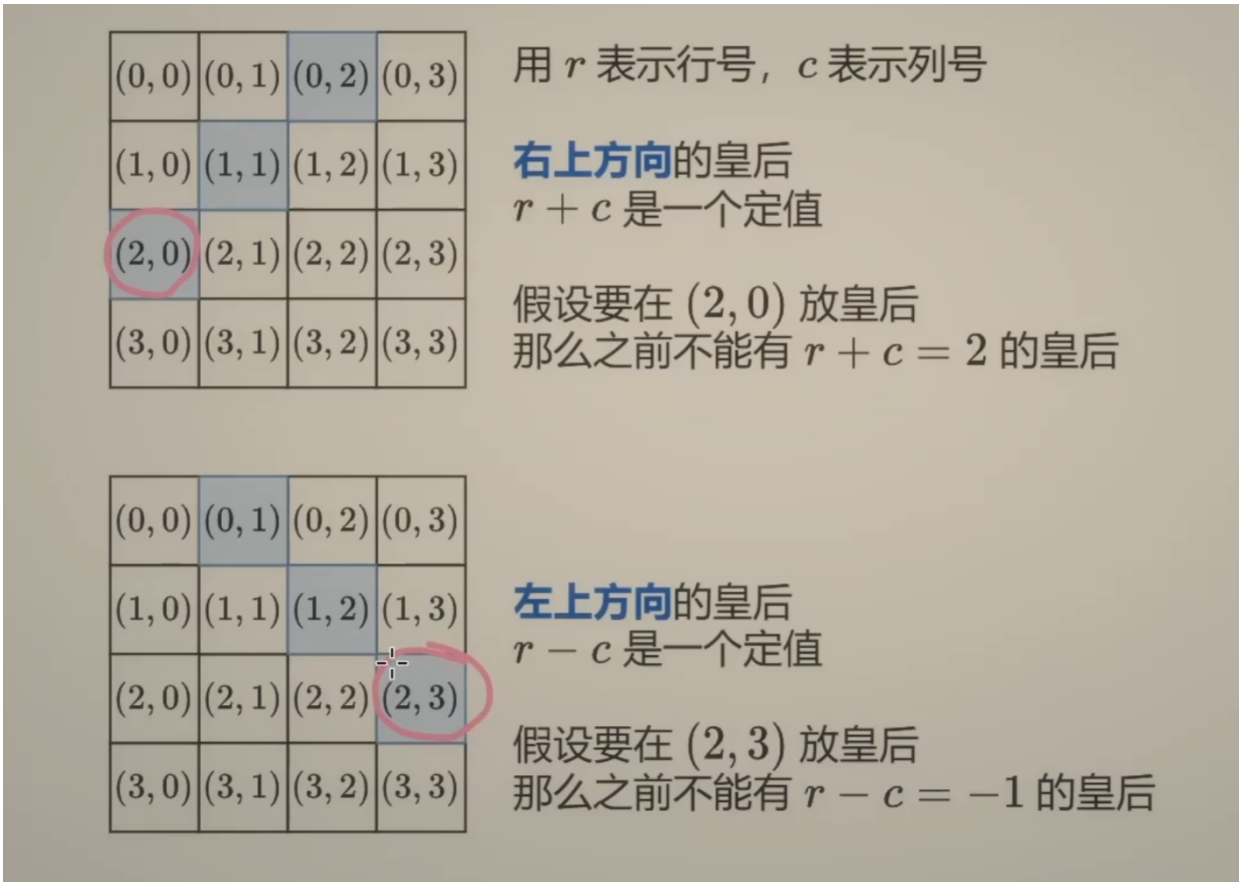
暴力枚举可重复全排列匹配 + 位运算压缩。用石头个数大于1 和 没有石头的位置，构造两个列表，进行全排列暴力匹配。

```
def minimumMoves(self, grid: List[List[int]]) -> int:
        frm, to = [], []
        for i, row in enumerate(grid):
            for j, x in enumerate(grid[i]):
                if x == 0: to.append((i, j))
                elif x > 1: frm.extend((i, j) for _ in range(x - 1))
        res = inf
        n = len(frm)
        path = [None] * n
        def dfs(i, S):
            nonlocal res
            if i == n:
                cst = sum(abs(x1 - x2) + abs(y1 - y2) for (x1, y1), (x2, y2) in zip(path, to))
                res = min(res, cst)
                return
            for j in range(n):
                if (S >> j) & 1:
                    path[i] = frm[j]
                    dfs(i + 1, S ^ (1 << j))
        dfs(0, (1 << n ) - 1)
        return res
```

N皇后问题

皇后之间不同行，不同列，且不能在同一斜线。如果只满足不同行不同列，等价于每行每列恰好一个皇后。如果用 col 表示皇后的位置，$col[i]$ 表示 第 i 行的皇后在第$col[i]$ 列，则 "每行每列恰好一个皇后" 等价于 枚举 col 的全排列。

加上斜线上不能有皇后的条件，如果从上往下枚举，则左上方向、右上方向不能有皇后。所以问题变成，当前枚举到 第 i 行，可以枚举的列号的集合 S。枚举列$j \in S$，合法情况即在 $\forall r \in [0, i-1]$，其列值 $c = col[r]$ 都不满足 $i + j = r + c$ 或者 $i - j = r - c$。



写法1：$dfs(i,S)$ 枚举当前到第 i 行（选第i 个数），可以选择的列号的集合是 S （没选择过的数字集合S）

```
    def solveNQueens(self, n: int) -> List[List[str]]:
        res = []
        path = [0] * n
        # 当前枚举到第 i 行，可以继续枚举的列号集合是 s
        def valid(i, j):
            for r in range(i):
                c = path[r]
                if r + c == i + j or r - c == i - j:
                    return False
            return True
        def dfs(i, s):
            if i == n:
                res.append(['.' * j + 'Q' + (n - j - 1) * '.' for j in path])
                return
            for j in s:
                if valid(i, j):
                    path[i] = j
                    dfs(i + 1, s - {j})
        dfs(0, set(range(n)))
        return res
```

写法2：回溯全排列 + 位运算 + 集合优化$O(1)$ 判断斜线方向

由于判断 $i+j$ 和 $i-j$ 是否在之前回溯中出现过需要$O(n)$的时间，实际上只需要用集合记录出现过的 $i+j$ 和 $i-j$ 即可。对于出现过$i+j$ 和 $i-j$ 分别（防止相互干扰）放进集合 lu 和ru 中（由于 位运算中 $i-j$ 可能出现负值，所以存放的元素改成 $i-j+10$）。

```
    def solveNQueens(self, n: int) -> List[List[str]]:
        res = []
        path = [0] * n
        lu = ru = 0
        # 当前枚举到第 i 行，可以继续枚举的列号集合是 s
        def dfs(i, s):
            nonlocal lu, ru
            if i == n:
                res.append(['.' * j + 'Q' + (n - j - 1) * '.' for j in path])
                return
            for j in range(n):
```

```
                if (S >> j) & 1 and (lu >> (i + j)) & 1 == 0 and (ru >> (i - j + 10)) & 1 == 0:
                        path[i] = j
                        lu, ru = lu | (1 << (i + j)), ru | (1 << (i - j + 10))
                        dfs(i + 1, S & ~(1 << j))
                        lu, ru = lu ^ (1 << (i + j)), ru ^ (1 << (i - j + 10))
        dfs(0, (1 << n) - 1)
        return res
```

回溯分割字符串

记录当前切割到的位置，枚举下一个切割位置，判断切割合法性。

LCR 086. 分割回文串 - 力扣（LeetCode）

```
    def partition(self, s: str) -> List[List[str]]:
        n, path, res = len(s), [], []
        # 当前分割的位置，枚举下次分割位置
        def dfs(i):
            if i == n:
                res.append(path.copy())
                return
            for j in range(i + 1, n + 1):
                t = s[i: j]
                if t == t[::-1]:
                    path.append(t)
                    dfs(j)
                    path.pop()
        dfs(0)
        return res
```

93. 复原 IP 地址 - 力扣（LeetCode）

增加了字符串段数限制：恰好等于4。时间复杂度：$O(n \times C(n,3))$

```
    def restoreIpAddresses(self, s: str) -> List[str]:
        n, path, res = len(s), [], []
        # 记录当前分割位置，枚举下一个分割位置
        def dfs(i):
            if len(path) == 4:
                if i == n:
                    res.append('.'.join(path))
                return
            for j in range(i + 1, n + 1):
                t = s[i: j]
                if t == '0' or '0' not in t[0] and int(t) <= 255:
                    path.append(t)
                    dfs(j)
                    path.pop()
        dfs(0)
        return res
```

2698. 求一个整数的惩罚数 - 力扣（LeetCode）

判断一个数，其平方是否可能划分成若干字符串，其各段对应数字之和等于本身。例如 $36 \times 36 = 1296, 1 + 29 + 6 = 36$

```
    def check(x):
        sx = str(x * x)
        n = len(sx)
        def dfs(i, s):
            if i == n: return s == x
            t = 0
            for j in range(i + 1, n + 1):
                t = t * 10 + int(sx[j - 1])
                if t + s <= x and dfs(j, s + t):
                    return True
            return False
        return dfs(0, 0)
```

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
```

多维二分

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

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
```

实现bisect_left

注意：查找范围为$[lo, hi]$；当 $x > max(nums[lo : hi + 1])$ 时，结果 lo 值 等于 hi 。

$bisect\_left(nums, x + 1, lo, hi) - 1$ 查找闭区间$[lo, hi]$内，恰好大于 x 的首个位置。如果不存在时，$lo = hi$ ，注意需要特判。

当$hi = n$ 时，兼容了存在和不存在两种情况。当不存在时，$lo = n$。

```
def bisect_left(nums, x, lo, hi):
    def check(pos):
        return nums[pos] >= x
    while lo < hi:
        # 查找恰好比x大于等于的位置
        mid = (lo + hi) >> 1
        if check(mid):
            hi = mid
        else:
            lo = mid + 1
    return lo
print(bisect_left(nums, val, 0, len(nums)))
```

2563. 统计公平数对的数目 - 力扣（LeetCode） 同 Problem - 1538C - Codeforces

```
def countFairPairs(self, nums: List[int], lower: int, upper: int) -> int:
        n = len(nums)
        nums.sort()
        res = 0
        L, R = lower, upper
        def bisect_left1(nums, x, lo, hi):
```

```
            while lo < hi:
                mid = (lo + hi) >> 1
                if nums[mid] >= x:
                    hi = mid
                else:
                    lo = mid + 1
            return lo
        for i, x in enumerate(nums):
            res += bisect_left1(nums, R - x + 1, i + 1, n) - 1 -  bisect_left1(nums, L - x, i + 1, n) + 1
        return res
```

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 $min(nums[i], \cdots, nums[j]) \times (j-i+1)$，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
```

单调栈维护元素的左右山形边界

对于 $a[i]=x$，希望找到在左侧中的最小 l 满足 $a[l+1] \sim a[i-1] \geq$ 或者 $\leq x$;

对于 $a[i]=x$，希望找到在右侧中的最大 r 满足 $a[i+1] \sim a[r-1] \geq$ 或者 $\leq x$;

```
    stk, left = [], [-1] * n
    for i in range(n):
        x = nums[i]
        while stk and x <= nums[stk[-1]]: stk.pop()
        if stk: left[i] = stk[-1]
        stk.append(i)
    stk, right = [], [n] * n
    for i in range(n - 1, -1, -1):
        x = nums[i]
        while stk and x <= nums[stk[-1]]: stk.pop()
        if stk: right[i] = stk[-1]
        stk.append(i)
```

2334. 元素值大于变化阈值的子数组 - 力扣（LeetCode）

在山形边界开区间所夹的区间内，满足所有元素大于等于山形边界元素 x，即 $x=\min(a[l+1] \sim a[r-1])$。

```
    def validSubarraySize(self, nums: List[int], threshold: int) -> int:
        n = len(nums)
        # 单调栈解法
        stk, left = [], [-1] * n
        for i in range(n):
            x = nums[i]
            while stk and x <= nums[stk[-1]]: stk.pop()
            if stk: left[i] = stk[-1]
            stk.append(i)
        stk, right = [], [n] * n
        for i in range(n - 1, -1, -1):
            x = nums[i]
```

```
            while stk and x <= nums[stk[-1]]: stk.pop()
            if stk: right[i] = stk[-1]
            stk.append(i)
        for i, x in enumerate(nums):
            l, r = left[i], right[i]
            k = r - l - 1
            if x > (threshold / k): return k
        return -1
```

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])
        stkyy = [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], stkyy[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）

```python
    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

```python
    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：$O(n^2)$

```python
    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]是一个长度为为i + 1 的滑动窗口，转移成滑动窗口最值问题。

```python
    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 rang(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=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

数论

取整函数

上下取整转换

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

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

不等式

$$x - 1 < \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1$$

取余性质

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

幂等律

$$\lfloor \lfloor x \rfloor \rfloor = \lfloor x \rfloor$$
$$\lceil \lceil x \rceil \rceil = \lceil x \rceil$$

素数

(1). 埃氏筛

时间复杂度：$O(nloglogn)$

```
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$\frac{p}{n}$个数。

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

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晒一次，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 $O(logn) \sim O(\sqrt{n})$

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$[2, int(\sqrt{n})]$ 区间后，如果 x 比 1大，则x 就等于那最后一个质因子。

```python
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：

```python
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
```

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

试除法求所有约数

复杂度为：$O(\sqrt{n})$

```python
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()
```

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$(5 + 1) \times (1 + 1) = 12$

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$(2^0 + 2^1 + \cdots + 2^5) \times (3^0 + 3^1)$。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。

871. 约数之和 - AcWing题库

```python
from collections import Counter
from math import *
moder = 10 ** 9 + 7
res = 1
t = int(input())
cnt = Counter()
for _ in range(t):
    x = int(input())
    for i in range(2, int(sqrt(x)) + 1):
        if x % i == 0:
            c = 0
            while x % i == 0:
```

```
                c += 1
                x //= i
            cnt[i] += c
    if x > 1: cnt[x] += 1
def S(a, n):
    s0 = 1
    for _ in range(n):
        s0 = (a * s0 + 1) % moder
    return s0
for a, n in cnt.items():
    res = (res * S(a, n)) % moder
print(res % moder)
```

欧几里得算法

算法原理：$gcd(a,b)=gcd(b,a \mod b)$

证明：

- 对于任意一个能整除a 且 能整除 b 的数 d，$a \mod b$可以写成 $a-k\cdot b$，其中 $k=a//b$，所以 d一定能够整除 $b, a \mod b$;
- 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：$a \mod b \in [0,\ b-1]$。例如，$38 \mod 13=12$

引理2：取模，余数至少折半。

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

如果 $b\le a//2,\ a \mod b\le b-1\le a//2-1$。

情况1：当每次执行 gcd时，如果 $a<b$，则交换；情况2：否则$a\ge b$，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：

$O(T)=O(T/2)+2=O(T/4)+4=O(\frac{T}{2^k})+k\times 2=2\log k$，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 $p_1,p_2,\cdots,p_k$ 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

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

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

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

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：

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

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
```

1. 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：

- $N'\mod p_1=0$，则 N' 包含了N 的所有质因子。

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

- $N'\mod p_i\neq 0$，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：

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

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]
```

2. 欧拉定理

$$若a与n互质，则a^{\phi(n)} \mod n = 1$$

> 例如：$5^{\phi(6)} \mod 6 = 5^2 \mod 6 = 25 \mod 6 = 1$。

证明：考察$1 \sim n$ 中与n 互质的 $\phi(n)$ 个数：p_1，p_2，$\cdots$，$p_{\phi(n)}$。将他们乘上 a ，再逐个对 n 取模，得到另一组数 $ap_1 \mod n$，$ap_2 \mod n, \cdots$，$ap_{\phi(n)} \mod n$。

可以证明这一组数两两不相同（反证法，若$ap_i \equiv ap_j(\mod n)$，则$ap_i - ap_j \equiv 0(\mod n)$，由于 a，n 互质，则一定有 $p_i = p_j$，矛盾），同时这一组每个数都和 n 互质（因为 a 和 p_i 都与 n 互质）。

则可以得到，新的这组数集 和 原先与 n 互质的数集完全相同。有：$p_1 \cdot p_2 \cdots p_{\phi(n)} \mod n = \prod ap_i \mod n$，即：

$$a^{\phi(n)} \equiv 1 \pmod n$$

3.费马小定理

若 a 与 素数 p 互质，则 $a^{p-1} \equiv 1(\mod p)$。

乘法逆元

逆元解决除法取模

如果b 与p 互质，对于$\forall\ a$，如果a/b 是整数，则一定存在乘法逆元 x，使得 $\frac{a}{b} \equiv a \cdot x(mod\ p)$。 x 是 b 的乘法逆元，记为b^{-1}（b 模 p 的逆元）。**b 的乘法逆元存在的充要条件：b 和 p 互质。**

逆元性质

- $b \cdot b^{-1} \equiv 1(mod\ p)$（证明：对定义式两边同乘 b，得到 $a \equiv a \cdot b^{-1} \cdot b(mod\ p)$，由于 a 是 b 的倍数，且b 与 p 互质，所以a 与 p 互质，满足同余的除法原理。）
- 当模数 p 为素数时，$b^{-1}_{\mod p} = b^{p-2}$。（证明：特殊情况下，对于质数 p，由费马小定理得 $b^{p-1} \equiv 1(mod\ p)$，则可知，要求与其互质的数 b 的逆元 x 满足 $bx \equiv 1(mod\ p)$，$b^{-1} = x = b^{p-2}$，可以使用快速幂求。

裴蜀定理

对正整数 a, b，记最大公约数 $d = gcd(a, b)$，

- 对于任意 x, y，$ax + by = D$，D 是 d 的倍数，即 $d|(ax + by)$
- 一定存在 x, y 使得 $ax + by = d$ 成立。（例如，一定存在 x, y 使得 $12x + 8y = 4$）
- 推论：$gcd(a, b) = d \iff$ 存在 x, y使得$ax + by = d$;

  a, b 互质 $\iff$ 存在x, y 使得 $ax + by = 1$。

推广：对于任意n 个数 $a1, a2, \cdots, a_n$，最大公约数为 $d = gcd(a1, \cdots, a_n)$

- 一定存在 $x1, \cdots, x_n$ 使得 $\sum a_i x_i = d$ 成立。（即对于任意 $\sum a_i x_i = k$，k 一定是 d 的整数倍）
- $gcd(a1, \cdots, a_n) = d \iff$ 存在 $x_1, c \ldots, x_n$，使得 $\sum a_i x_i = d$

[1250. 检查「好数组」 - 力扣（LeetCode）]

判断是否能从原给定集合中，选出子集 $A = \{a1, \cdots, a_n\}$，存在一组$X = \{x1, \cdots, x_n\}$，使得$AX = 1$。实际只需要整个原集合的 gcd 值为1，则一定存在一个最小子集其gcd 为1，由裴蜀定理，能找到存在一组$X = \{x1, \cdots, x_n\}$，使得$AX = 1$。

```python
    def isGoodArray(self, nums: List[int]) -> bool:
        res_gcd = nums[0]
        for x in nums:
            res_gcd = gcd(res_gcd, x)
        return res_gcd == 1
```

[1625. 执行操作后字典序最小的字符串 - 力扣（LeetCode）]

暴力枚举需要 $O(nC + nC)$，其中当 b 是偶数时，$C = 10$，否则 $C = 10^2$。其中枚举轮转起点位置阶段可以使用裴蜀定理优化。

对于任意起点 $i' = (i \times b) \mod n = ib - kn$，由裴蜀定理 $\sum a_i x_i = k$，k 一定是 d 的整数倍，所以 $i' = K \times gcd(b, n)$。则在一个字符串轮转中，只需要通过扩展字符串为两倍长度，枚举起点在 $gcd(a, b)$ 的整倍数的位置即可。时间复杂度：$O(nC + \frac{n}{gcd(a,b)} \times C)$，其中当 b 是偶数时，$C = 10$，否则 $C = 10^2$。

```python
    def findLexSmallestString(self, s: str, a: int, b: int) -> str:
        res = s
        n = len(s)
        s = s + s
        s_str = set()
        e_lim = 10 if b & 1 else 1   # b为奇数才可以对偶数位置增加
        for o_cnt in range(10): # 对奇数位置增加a的次数
            for e_cnt in range(e_lim):
                tmp = list(map(int, s))
                for i in range(1, 2 * n, 2): tmp[i] = (tmp[i] + a * o_cnt) % 10
                for i in range(0, 2 * n, 2): tmp[i] = (tmp[i] + a * e_cnt) % 10
                s_str.add(''.join(map(str, tmp)))
        def _gcd(a, b):
            return _gcd(b, a % b) if b else a
        g = _gcd(n, b)
        for i in range(0, n, g):    # 裴蜀定理优化
            for ss in s_str:
                tmp = ss[i: i + n]
                if tmp < res: res = tmp
        return res
```

扩展欧几里得

线性同余方程

同余

两个整数a,b,若它们除以正整数m 所得的余数相等，则称a, b对于模m 同余， 读作a同余于b 模m,或读作a与b关于模m 同余。

同余性质

- 保持基本运算：

$$a \equiv b \pmod{m} \Rightarrow \begin{cases} an \equiv bn \pmod{m}, \forall n \in \mathbb{Z} \\ a^n \equiv b^n \pmod{m}, \forall n \in \mathbb{N}^0 \end{cases} .$$

- 除法原理：若$ka \equiv kb \pmod{m}$且k, m 互质，则$a \equiv b \pmod{m}$

线性同余方程

给定a, b, m， 求一个整数解x， 使得 $ax \equiv b (\mathrm{mod}\ m)$。

> 例如 $2x \equiv 3 (\mathrm{mod}\ 6)$，无解；$4x \equiv 3 (\mathrm{mod}\ 5)$，解的形式是$5k + 2$。

转换为 存在整数y， 使得$ax = my + b$，即 $ax + my' = b$。这个方程有解的条件是 b 能整除 $gcd(a, m)$。

中国剩余定理

引理：寻找整数 y_1 满足 y_1 除以3余1、除以5余0、除以7余0;

$y1$ 一定是 $5 \times 7 = 35$ 的倍数，设 $y1 = 35k$ ，则有 $35k \equiv 1 (\mathrm{mod}\ 3)$，此时 k 是 35 模3的逆元

[中国剩余定理（CRT） - 知乎 (zhihu.com)]

设整数 $m_1, m_2, \ldots, m_n$ **两两互质**，则对于任意的整数$a_1, a_2, \ldots, a_n$,方程组

$$\begin{cases} x \equiv a_1 (\mathrm{mod}\ m_1) \\ x \equiv a_2 (\mathrm{mod}\ m_2) \\ \cdots \\ x \equiv a_n (\mathrm{mod}\ m_n) \end{cases}$$

都存在整数解。若X, Y 都满足该方程组，则必有 $X \equiv Y (\mathrm{mod} N)$,其中 $N = \prod_{i=1}^{n} m_i$ 。

$$x \equiv \sum_{i=1}^{n} a_i \times \frac{N}{m_i} \times \left[\left(\frac{N}{m_i} \right)^{-1} \right]_{m_i} \pmod{N}。$$

离散数学

容斥

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 $= (x + 2 \cdot x + \cdots + n//x \cdot x) = x \cdot (1 + n//x) * (n//x)//2$

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)
```

3116. 单面值组合的第 K 小金额 - 力扣（LeetCode）

容斥 + 预处理最小公倍数：给定无重复集合 $coins'$，$1 \sim x$ 中，能对任意一个 $coins'$ 中元素整除的个数 为 $check(x)$。

将问题转换成，恰好能有 不少于 k 个数被任意一个 $coins'$ 中元素整除的 x 的值，使用二分答案。回溯法枚举子集，预处理所有 $coins'$ 子集的最小公倍数，所有相同长度、为l 的子集的最小公倍数存放在 $dic[l]$ 中。 对于任意一个数 y，$1 \sim x$ 中能被它整除的个数为 $int(x/y)$。

```
    def findKthSmallest(self, coins: List[int], k: int) -> int:
        coins.sort()
        if coins[0] == 1: return k
        c = set(coins)
        n = len(coins)
        for i in range(n):
            for j in range(i + 1, n):
                x, y = coins[i], coins[j]
                if y % x == 0 and y in c:
                    n -= 1
                    c.remove(y)
        coins = list(c)

        # 预处理：dic[i] 表示 从coins 中选出 i个数的子集的最小公倍数
        dic = defaultdict(list)
        dic[1] = coins

        # 回溯枚举子集
        path = []
        def dfs(i):
            l = len(path)
            if i == n:
                if l >= 2:
                    lcm_ = path[0]
                    for j in range(1, l):
                        lcm_ = lcm(lcm_, path[j])
                    dic[l].append(lcm_)
                return
            dfs(i + 1)
            path.append(coins[i])
            dfs(i + 1)
            path.pop()
        dfs(0)

        def check(x):
            # 检查 1 ~ x 中，能被任意一个c 整除的个数res和k的关系
            res = 0
            for l in range(1, n + 1):
                plus = l & 1
                for d in dic[l]:
                    res = res + (1 if plus else -1) * (x // d)
            return res >= k

        lo, hi = 0, 5 * 10 ** 10 + 10
        while lo < hi:
```

```
            mid = (lo + hi) >> 1
            if check(mid):
                hi = mid
            else:
                lo = mid + 1
        return lo
```

鸽巢原理 / 抽屉原理

常用于求解最坏情况下的解，以及证明不存在解（连最坏情况下，都不存在解，则所有情况不存在解）。

鸽巢原理定理

有 $n+1$ 只鸽子，飞入 n 个鸽子巢，则至少有一个巢里有不少于两只鸽子。（反证法：假设没有一个巢中有不少于两只鸽子，则鸽子总数不会超过 n，矛盾）。即将 $n+1$ 个物体，划分成 n 组，至少有一组有不少于两个物体。

推广：将 n 个物体，划分成 k 组，至少有一组不少于 $\lceil \frac{n}{k} \rceil$。（证明：反证法，假设所有组少于 $\lceil \frac{n}{k} \rceil$，则至多 $(\lceil \frac{n}{k} \rceil - 1) \times k < (\frac{n}{k}) \times k = n$ 个物体，矛盾）

> 例如，53个物体，分成6组，最坏情况下是 9，9，9，9，9，8。

简单应用：

- 任意11个整数中，至少有2个整数之差是10的倍数。（证明，从余数角度来看，11个数对10的余数有11个，一共有10种余数 0 ~ 9，至少有两个数对10同余，故其差也对10同余）
- 一个人骑车10小时内走完了281公里路程，已知他第一小时走了30公里，最后一小时走了17公里。证明：他一定在某相继的两小时中至少走完了58公里路程。（证明：8小时走了234公里，234个物品分到8组，最坏情况下，至少有一组是 $\lceil \frac{234}{8} \rceil = 29$公里，其余各组是 28 公里，那么第一小时和第二小时一定至少有58公里）

Ramsey定理 / 拉姆齐定理

任意 n 个人，必然有 $\lceil (n-1)/2 \rceil$ 个人相互认识 或者 相互不认识。（证明：考虑其中一个人的视角，剩下 n - 1个人需要划分成两组，$k1$ 表示与它认识，$k2$ 表示与他不认识，其中一组至少为 $\lceil (n-1)/2 \rceil$ 个人。）

[P8807 蓝桥杯 2022 国 C] 取模 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

给定 n, m, 问是否存在两个不同的数 x, y 使得 $1 \le x < y \le m$ 且$n \bmod x = n \bmod y$。

考虑反面情况，当且仅当对于任意 $[1, m]$ 的数 x ，$n \bmod x$ 两两不相等，则不成立。由于对于任意的 n，$n \bmod 1 = 0$，所以 $n \bmod 2$ 只能取1，同理 $n \bmod 3$只能取2，可以得到 $n \bmod k$必须取 $k-1$。所以当且仅当 $\forall k \in [1, m]$有$n \bmod k = k - 1$恒成立，才不存在。

```
def solve():
    n, m = map(int, input().split())
    for k in range(1, m + 1):
        if n % k != k - 1:
            return 'Yes'
    return 'No'
```

由中国剩余定理，

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 $\ge$ 乱序和 $\ge$ 倒序和。

对于$a_1 \le a_2 \le \cdots \le a_n$，$b_1 \le b_2 \le \cdots \le b_n$, 并有$c1, c2, \cdots, c_n$是$b1, b2, \cdots, b_n$的乱序排列。有如下关系：

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

当且仅当$a_i = a_j$或者$b_i = b_j$ $(1 \le i, j \le n)$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

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

当且仅当 $x_1 = x_2 = \cdots = x_n$,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 = heappop(hq)
            v += 1
            alp.append(k)
            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

$$\Sigma_1^n n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

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 * (b*c) % 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

数列

等比数列求和公式

$$S_n = \frac{a_1(1-q^n)}{1-q},\ q \neq 1$$

递推方法求等比数列求和（带模运算）

希望求：$S(a,n) \mod p = (a^0 + a^1 + \cdots + a^n) \mod p$，不难发现$S(a,n) = a \cdot \big(S(a,n-1)\big) + 1$。

时间复杂度：$O(n)$

```
def S(a, n):
    s0 = 1
    for _ in range(n):
        s0 = (a * s0 + 1) % moder
    return s0
```

组合数学

排列

$A_m^n = \frac{m!}{n!}$

$\mathrm{A}_n^m = n\mathrm{A}_{n-1}^{m-1}$

递推公式：可理解为“某特定位置”先安排，再安排其余位置。

```
def A(n, m):
    if m == 0:
        return 1
    return n * A(n-1, m-1)
```

组合数学

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

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

递推公式：

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

```
def C(n, m):
    if m == 0 or n == m:
        return 1
    return C(n-1, m-1) + C(n-1, m)
```

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

二项式定理

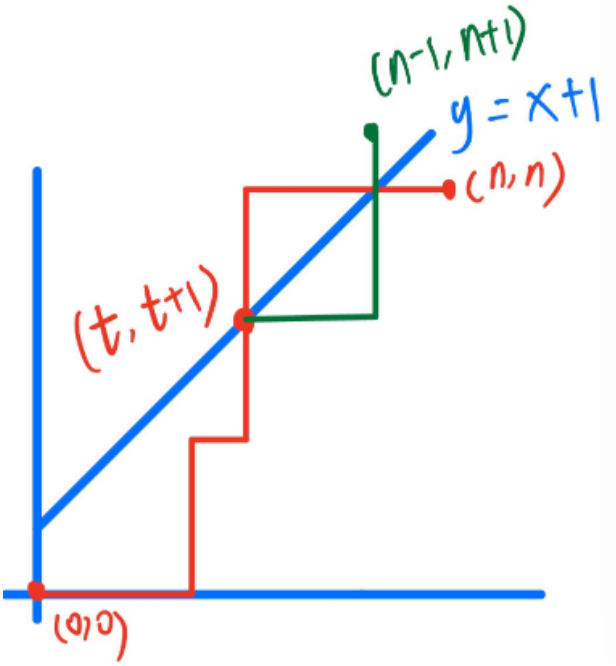
$$(a+b)^n = \sum_{i=0}^{n} C_n^i a^i b^{n-i}$$

卡特兰数

5. 卡特兰数（Catalan）公式、证明、代码、典例. c n = n+11 (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

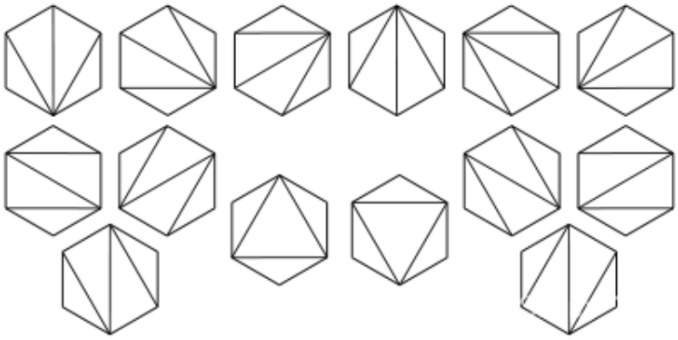
- n 个节点可以构造的不同的二叉树的个数。（证明：$F(n)$为有n个节点的二叉树的所有根节点个数。其左子树的可能情况为 $F(i), i \in [0, n]$,对应右子树的情况为 $F(n - i)$, 乘积求和形式即为卡特兰数列的递推式。
- 从 $(0,\ 0)$ 到 右上角 $(n,\ n)$，每次只能向右或者向上，向上的次数不超过向右的次数的路径数。（即不超过 $y = x$ ）
- 一个无穷大栈，进栈顺序为 $1, 2, \ldots, n$ 的出栈顺序数
- n 个左括号和 n 个右括号构成的括号序列，能够构成的有效括号序列个数。

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)+.....+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)。

快速幂

欧拉降幂 / 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
```

> 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

```python
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
```

高等数学

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~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 ~ n中所有2的倍数，大约是 n // 2个。
- 所以 1~n所有的数的约数个数和 可以看成 所有的倍数的个数 = $n/1+n/2+n/3+\cdots+n/n=n\sum_{i=1}^{n}\frac{1}{i}=n\ln n$。
- 所以=，从期望角度来讲，一个数n 的约束个数的期望约是 $\ln n$

泰勒展开式

$$f(x)=f(x_0)+\frac{f'(x_0)}{1!}(x-x_0)+\frac{f''(x_0)}{2!}(x-x_0)^2+\cdots+\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n+R_n$$

麦克劳林公式： $x_0=0$

$$f(x)=\sum_{k=0}^{n}\frac{f^{(k)}(0)}{k!}\cdot x^k+o(x^n)=f(0)+\frac{f'(0)}{1!}\cdot x+\frac{f''(0)}{2!}\cdot x^2+\cdots+\frac{f^{(n)}(0)}{n!}\cdot x^n+o(x^n)$$

常用展开：

$$e^x=1+\frac{1}{1!}\cdot x+\frac{1}{2!}\cdot x^2+\cdots+\frac{1}{n!}\cdot x^n+o(x^n)$$

所以有：

$$e=\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\cdots+\frac{1}{n!}$$

Stirling 斯特林公式

描述阶乘的近似阶：

$$n!=\sqrt{2\pi n}(\frac{n}{e})^n$$

所以可以得到卡特兰数列的近似：

$$O(C_n) \sim O(\frac{4^n}{n^{\frac{3}{2}} \cdot \sqrt{\pi}})$$

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$m = max(nums)$, 可以看成一个 有 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：$O\left(n\left(\sqrt{m} \cdot \alpha(m)\right) + n \log n\right)$

```
    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())
```

并查集维护连通块大小

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]
```

并查集维护链

把考虑过的元素串起来，链条的长度就是当前一段数的长度。关键需要高效串联两条链。

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

时间复杂度：$O(nlogn)$

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。

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

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
        ansl, ansr = inf, inf
        if ql <= mid:ansl = 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(ansl, 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]
                ansl, ansr = inf * self.flag, inf * self.flag
                mid = (l + r) >> 1
                if ql <= mid: ansl = 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(ansl, 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

```python
# 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

```python
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 $x\ opt\ x = x$。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：$O(nlogn)$

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

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

$$初始化时，\ f(i,0) = a[i];$$

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。

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

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$f(4,3) = opt(f(4,2), f(8,2))$

```
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，$int(log_2^6) = 2$，只需要$k = 2^2$的两个区间一定可以覆盖整个区间。

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

$$qry(l,r) = opt(qry(l,k), qry(r-2^k+1, k))$$

```
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$int(log(7)) = int(log(3)) + 1 = int(log(1)) + 1 + 1$

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

模板

```
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):
    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])
```

类模板

```
class ST:
    def opt(self, a, b):
        return a & b
    def __init__(self, nums):
        n = len(nums)
        log = [0] * (n + 1)
        for i in range(2, n + 1):
            log[i] = log[i >> 1] + 1
        lenj = ceil(math.log(n, 2)) + 1
        f = [[0] * 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.opt(f[i][j - 1], f[i + (1 << (j - 1))][j - 1])
        self.f = f
        self.log = log
    def qry(self, L, R):
        k = self.log[R - L + 1]
        return self.opt(self.f[L][k], self.f[R - (1 << k) + 1][k])
```

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$O(mlogm)$，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$O(klogk)$。

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$O(klogU), U=\sum a_i$

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]
```

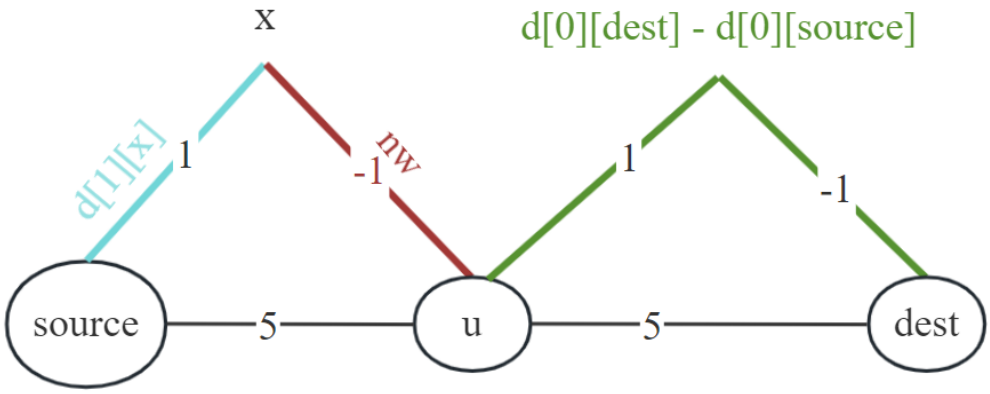
5. 动态修改边权

[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：

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

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

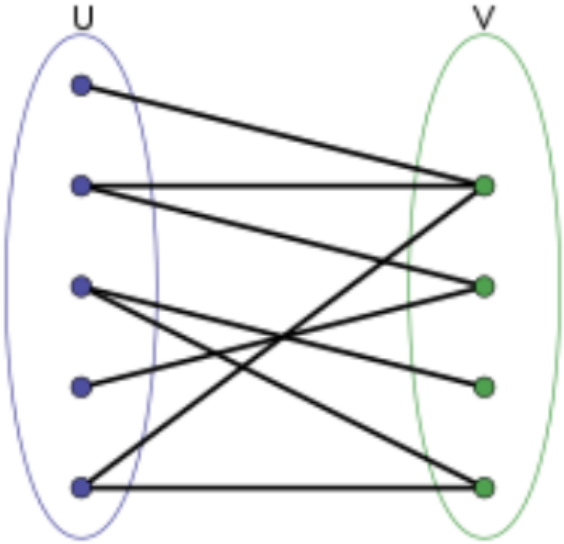
简单来说，如果图中点可以被分为两组，并且使得所有边都跨越组的边界，则这就是一个二分图。

[二分图的最大匹配、完美匹配和匈牙利算法_完美匹配图论-CSDN博客]

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



二分图判定

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
```

二分图最大匹配 / 匈牙利算法

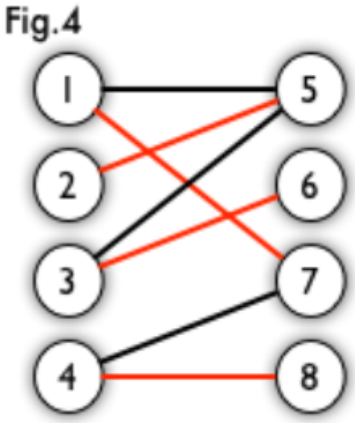
二分图的匹配

给定一个二分图G，在G的一个子图M中， M的边集 $\{E\}$ 中的**任意两条边都没有公共顶点**，则称M是一**个匹配。**

最大匹配：匹配边数最大的匹配。

完美匹配：如果一个图的某个匹配中，所有的顶点都是匹配点，那么它就是一个完美匹配。图 4 是一个完美匹配。

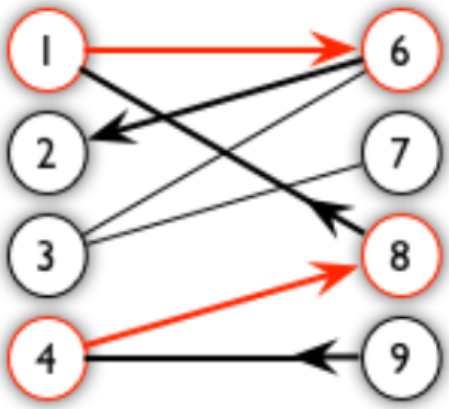
- 完美匹配一定是最大匹配（完美匹配的任何一个点都已经匹配，添加一条新的匹配边一定会冲突）
- 但并非每个图都存在完美匹配。
- 完美匹配的边数 = 左 / 右部的点数



二分图最大权完美匹配：二分图边权和最大的完美匹配。

交替路：从一个未匹配点出发(右)，依次经过非匹配边、匹配边、非匹配边...形成的路径叫交替路。

Fig.5



增广路：从一个未匹配点出发(右)，走交替路，如果途径另一个未匹配点（出发的点不算），则这条交替路称为增广路（agumenting path）。例如，图 5 中的一条增广路如图 6 所示（图中的匹配点均用红色标出）

- 特点：**非匹配边比匹配边多一条。**因此，研究增广路的意义是改进匹配。只要把增广路中的匹配边和非匹配边的身份交换即可。由于中间的匹配节点不存在其他相连的匹配边，所以这样做不会破坏匹配的性质。交换后，图中的匹配边数目比原来多了 1 条。

Fig.6

增广路定理

通过不停地找增广路来增加匹配中的匹配边和匹配点。找不到增广路时，达到最大匹配

861. 二分图的最大匹配 - AcWing题库

```
n1, n2, m = map(int, input().split())

vis = set()
match = {}
e = defaultdict(list)

def dfs(u) -> bool:
    for v in e[u]:
        if v in vis: continue
        vis.add(v)
        if v not in match or dfs(match[v]):
            match[v] = u
            return True
    return False
for _ in range(m):
    u, v = map(int, input().split())
    e[u].append(v)

for u in range(1, n1 + 1):
    vis = set()
    dfs(u)

print(len(match))
```

二分图最大权完美匹配 / KM算法

1947. 最大兼容性评分和 - 力扣（LeetCode）

暴力枚举时间复杂度：$O(m!)$，实际上是二分图的最大权完美匹配问题。复杂度：$O(m^3)$

```
class KM:
    def __init__(self, n):
        self.maxn = 300 + 10
        self.INF = float('inf')
        self.wx = [0] * (self.maxn)
        self.wy = [0] * (self.maxn)
        self.cx = [-1] * (self.maxn)
        self.cy = [-1] * (self.maxn)
        self.visx = [0] * (self.maxn)
        self.visy = [0] * (self.maxn)
        self.cntx = n
        self.cnty = n
        self.Map = [[0] * (self.maxn) for _ in range(self.maxn)]
        self.slack = [0] * (self.maxn)

    def dfs(self, u):
        self.visx[u] = 1
        for v in range(1, self.cnty + 1):
            if not self.visy[v] and self.Map[u][v] != self.INF:
                t = self.wx[u] + self.wy[v] - self.Map[u][v]
                if t == 0:
                    self.visy[v] = 1
                    if self.cy[v] == -1 or self.dfs(self.cy[v]):
                        self.cx[u] = v
                        self.cy[v] = u
                        return True
                elif t > 0:
                    self.slack[v] = min(self.slack[v], t)
        return False

    def KM(self):
        for i in range(1, self.cntx + 1):
            for j in range(1, self.cnty + 1):
                if self.Map[i][j] == self.INF:
                    continue
                self.wx[i] = max(self.wx[i], self.Map[i][j])
        for i in range(1, self.cntx + 1):
            self.slack = [self.INF] * (self.maxn)
            while True:
                self.visx = [0] * (self.maxn)
                self.visy = [0] * (self.maxn)
                if self.dfs(i):
                    break
                minz = self.INF
                for j in range(1, self.cnty + 1):
                    if not self.visy[j] and minz > self.slack[j]:
                        minz = self.slack[j]
                for j in range(1, self.cntx + 1):
                    if self.visx[j]:
                        self.wx[j] -= minz
                for j in range(1, self.cnty + 1):
                    if self.visy[j]:
                        self.wy[j] += minz
                    else:
                        self.slack[j] -= minz

        ans = 0
        for i in range(1, self.cntx + 1):
            if self.cx[i] != -1:
                ans += self.Map[i][self.cx[i]]
        return ans

    def add_edge(self, u, v, w):
        self.Map[u + 1][v + 1] = w
```

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]
```

并查集维护连通块属性

[928. 尽量减少恶意软件的传播 II - 力扣（LeetCode）]

题目问从 bad 选一个节点，删除其和其邻接的边，能得到感染后的最小数量。暴力做法枚举删除的 bad 点，需要 $O(n^3)$ 。

逆向思维：枚举所有的 $good$ 连通块，维护块大小 siz ，及其邻接的 bad 节点集合 cc_bad。恰好只有一个邻接 bad 的连通块，由于删除 bad 后整个连通块不会被感染，所以其对该 bad 节点的贡献为连通块大小。最后，返回所有 bad 节点得到的贡献和（来自恰好仅邻接该节点的 $good$ 连通块）最大、序最小的节点。

```
def minMalwareSpread(self, graph: List[List[int]], initial: List[int]) -> int:
        n = len(graph[0])
        fa = list(range(n))
        siz = [1] * n
        cc_bad = defaultdict(set)
        def find(x):
            if fa[x] != x: fa[x] = find(fa[x])
            return fa[x]
        def union(u, v):
            if find(u) != find(v):
                siz[find(u)] += siz[find(v)]
                cc_bad[find(u)] |= cc_bad[find(v)]
                fa[find(v)] = find(u)
        bad = set(initial)
        good = set(range(n)) - bad
        for u in good:
            for v, con in enumerate(graph[u]):
                if not con: continue
                if v in bad: cc_bad[find(u)].add(v)
                else: union(u, v)
        pa = set(find(u) for u in good)
        bad_siz = Counter()
        for p in pa:
            if len(cc_bad[p]) == 1:
                bad_siz[list(cc_bad[p])[0]] += siz[p]
        mx, res = 0, min(bad)
        for u, sz in bad_siz.items():
            if sz > mx: mx, res = sz, u
            if sz == mx: res = min(res, u)
        return res
```

树上问题

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

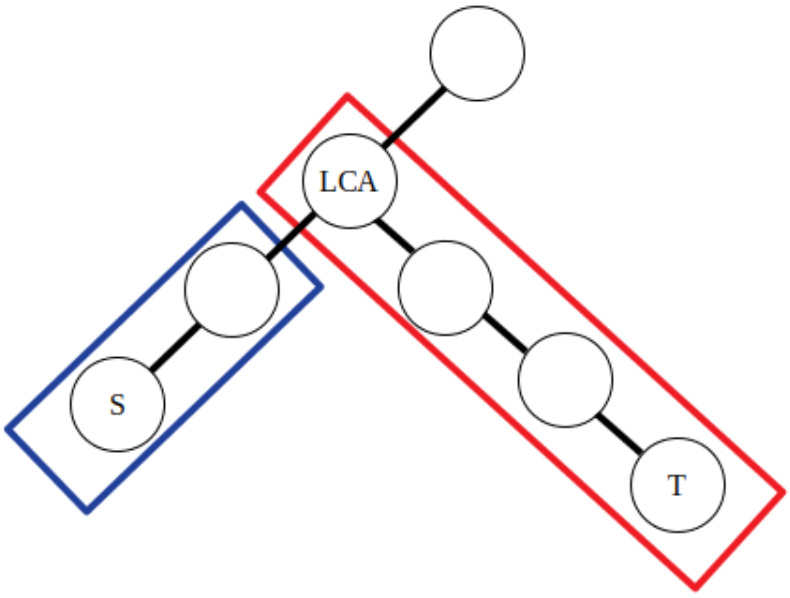
点差分：解决多路径节点计数问题。

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

```
# 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。

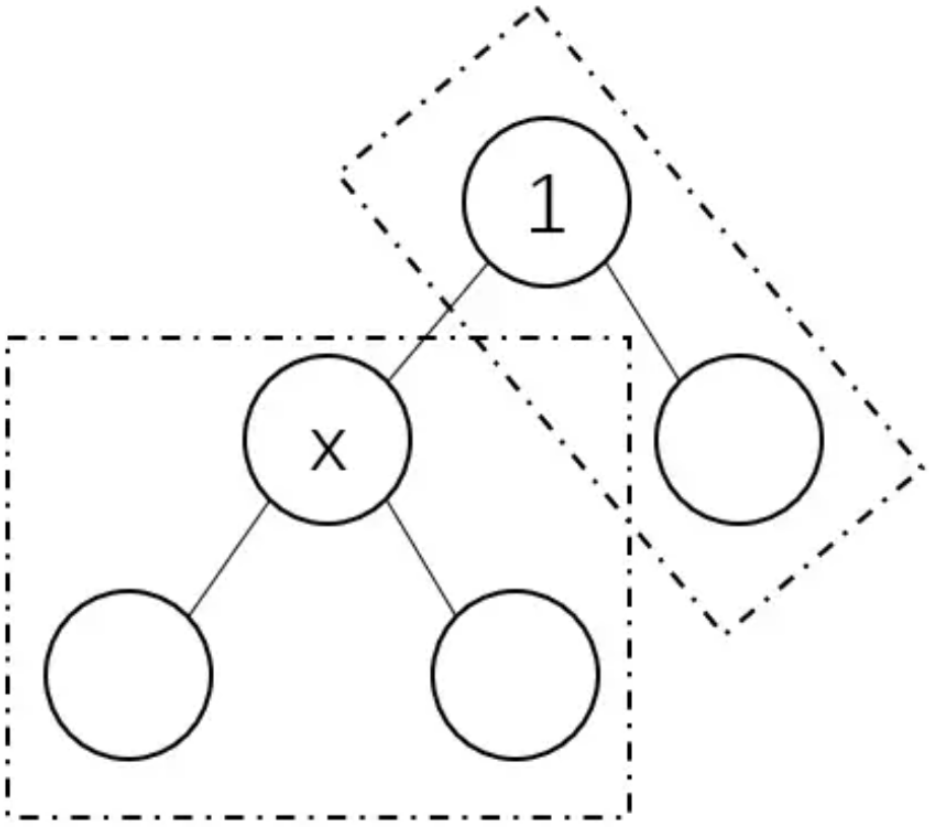
```python
    def sumOfDistancesInTree(self, n: int, edges: List[List[int]]) -> List[int]:
        g = [[] for _ in range(n)]
        dep = [0] * n
        siz = [1] * n
        res = [0] * n
        for u, v in edges:
            g[u].append(v)
            g[v].append(u)

        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 $u \to x_0 \to x_1 \to \cdots \to v$ 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）

```python
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

```python
    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

位运算与集合论

集合 A, B，最大二进制长度为 N

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

(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 $num >> x == s[i]$

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

从大到小枚举一个s 的所有非空子集

暴力做法是从 s 出发，不断减1。但是中途需要规避不是 s 子集的情况，相当于做 "压缩版" 的二进制减法：普通的二进制减法会把最低位的 1 变成 0，同时 1右边的 0 变成 1（例如 $101000 \to 100111$）；"压缩版" 的二进制减法只保留原集合中的的、右边的1，其余仍然是0。（例如 $101000 \to 100101$，假设$s = 111101$）。保留的方法，就是 $\&s$。

```
sub = s
while sub:
    # 处理 sub 的逻辑
    sub = (sub - 1) & s
```

Gosper's Hack：枚举大小恰好为 k 的子集

例如当前为 0100110，下一个大小仍然为 3 的集合是 0101001（$left = 0101000$，即 $sub + lowbit(sub)$；$right = 000001$，即$left \oplus sub = 0001111$，$right = left \oplus sub >> 2/lowbit(sub)$。

时间复杂度：$O(n \cdot C(n,k))$，实际上优化不大

```
    s = (1 << n) - 1
    sub = (1 << k) - 1
    def next_sub(x):
        lb = x & -x
        left = x + lb
        right = ((left ^ x) >> 2) // lb
        return left | right
    while sub <= s:
        # 处理 sub 逻辑
        sub = next_sub(sub)
```

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

```python
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。

```python
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
```

3022 给定操作次数内使剩余元素的或值最小

https://leetcode.cn/problems/minimize-or-of-remaining-elements-using-operations/

```python
        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这位蒙版就应置为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

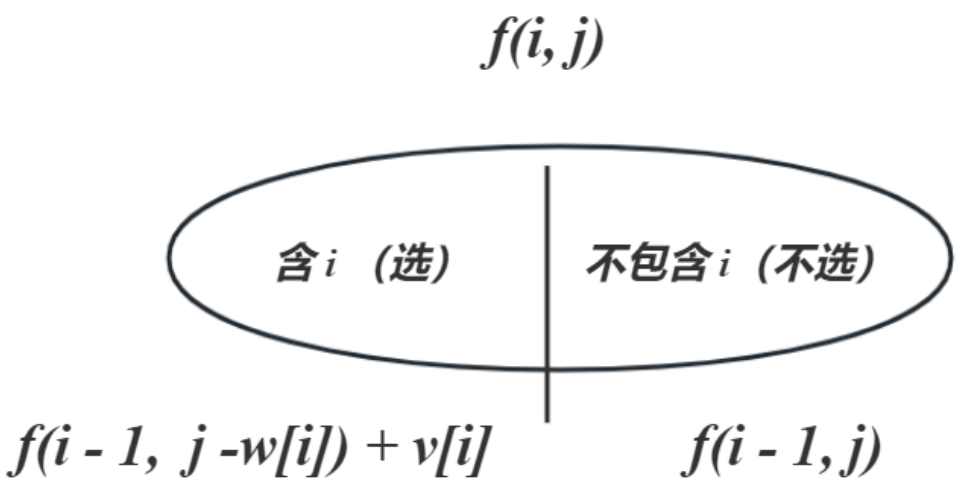
状态表示：$f(i,j)$

- 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$f(i-1,j-w[i])$，所以我们必须逆序遍历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$range(W, w[i]-1, -1)$

```
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$\min(\text{重量前缀}, target)$

```
    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
```

分割等和子集问题

给定一组数，判断是否可以分割成两个等和子集。

416. 分割等和子集 - 力扣（LeetCode）

```
    def canPartition(self, nums: List[int]) -> bool:
        s, n = sum(nums), len(nums)
        if s & 1: return False
        # f[i][j] 表示从前i个数中，分割成和为j是否可能。
        # f[n][s // 2]
        f = [1] + [0] * (s // 2)
        for x in nums:
            for j in range(s // 2, x - 1, -1):
                f[j] |= f[j - x]
        return f[s // 2] == 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 $f(i,j-w)$

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

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

```
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）

完全背包求组合方案数

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

```
    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：

```python
    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]
```

求排列方案数：伪完全背包

377. 组合总和 Ⅳ - 力扣（LeetCode）

$f(i)$ 表示找出总和为 i 的排列方案数，$f(i) = \sum f(i-w)$

```python
    def combinationSum4(self, nums: List[int], target: int) -> int:
        n = len(nums)
        f = [0] * (target + 1)
        f[0] = 1
        for i in range(1, target + 1):
            for j in range(n):
                w = nums[j]
                if i - w >= 0:
                    f[i] += f[i - w]
        return f[target]
```

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。

```python
    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：$O(N \cdot W^2)$

```python
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])
```

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

可以发现无法借助完全背包的方法进行优化。

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

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$1,2,\cdots,2^k,c'$，例如 $c=500$, 拆分成$1,2,\cdots,128,245$，可以证明这些数字可以枚举出$0\sim500$ 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

9. 分组背包问题 - AcWing题库

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)
```

$O(nlogn)$ 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 length = 1, $f[i][i-1]$ 是0，$f[j+1][j]$也是0。等价于没有

对于length = 2， $f[i][i+1]$其中一项$[i][i-1]+f[i+1][i+1]+...$，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]
```

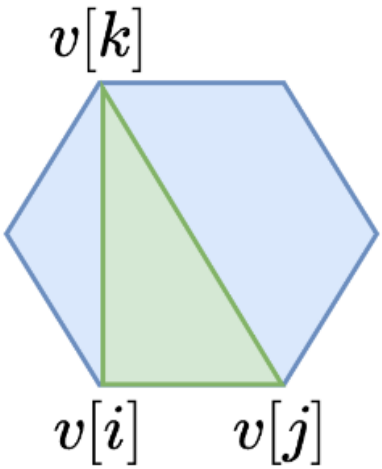
[375. 猜数字大小 II - 力扣（LeetCode）]

$f[a,b]$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）]



```python
    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 $i, i+1, \cdots, j$ 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。

```python
    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）

```python
    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 IOI2000] 回文字串 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

```python
    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
# yyyizzz , 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 $1234 \mod 17$，可以看成 $(1000 \mod 17) + (200 \mod 17) + (30 \mod 17) + (4 \mod 17)$，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

$0 \to 0$或者$1 \to 1$：加上x

$0 \to 1$或者$1 \to 0$：加上$x \oplus k$

```python
    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

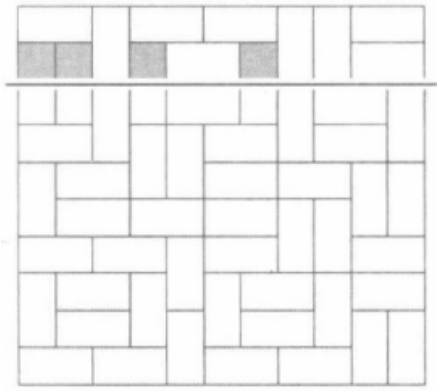
[291. 蒙德里安的梦想 - AcWing题库]

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：$f(i,j) = \sum valid(f(i-1,k))$

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。



```python
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]
```

约束全排列型状压

对于朴素的全排列问题，$f[i][s]$ 表示考虑完全排列 $p[0:i]$，已经选择集合状态为 s 情况下的合法方案数。一般转移方程：$f[i][s] = \sum f[i-1][s-\{j\}], \forall\ valid(j)$，初始化 $f[0][0] = 1$。时间复杂度为 $O(n^2 \times 2^n)$。

优化思路：由于 s 中包含了 i 的信息，即 $bin(s).count('1')$，所以第一维度可以省略。时间复杂度 $O(n \times 2^n)$

526. 优美的排列 - 力扣（LeetCode）

时间复杂度为 $O(n^2 \times 2^n)$。

```
class Solution:
    def countArrangement(self, n: int) -> int:
        res = 0
        # f[i][s]  考虑完perm[1] ~ perm[i]，已选择状态为s
        m = (1 << n) - 1
        f = [[0] * (m + 1) for _ in range(n + 1)]
        f[0][0] = 1
        for i in range(1, n + 1):
            for s in range(m + 1):
                for j in range(n):
                    if (s >> j) & 1 and ((j + 1) % i == 0 or i % (j + 1) == 0):
                        f[i][s] += f[i - 1][s ^ (1 << j)]
        return f[n][m]
```

优化：省略前一维度。时间复杂度 $O(n \times 2^n)$

```
class Solution:
    def countArrangement(self, n: int) -> int:
        res = 0
        # f[i][s]  考虑完perm[1] ~ perm[i]，已选择状态为s
        m = (1 << n) - 1
        f = [1] + [0] * m
        for s in range(m + 1):
            i = bin(s).count('1')
            for j in range(n):
                if (s >> j) & 1 and ((j + 1) % i == 0 or i % (j + 1) == 0):
                    f[s] += f[s ^ (1 << j)]
        return f[m]
```

集合是否能划分成 k 个相等子集

698. 划分为k个相等的子集 - 力扣（LeetCode）

$f[s]$ 为在压缩状态 s 下的余数。考察每一个在集合中的元素 $nums[j]$，对于删去其的集合 $ls = f[s \oplus (1<<j)]$，当且仅当 $ls + nums[j] \le siz$ 的时候可以更新 $f[s]$，相当于枚举所有删去一个元素的子集向 $f[s]$ 转移，能否构造出整数倍的集合。

这种方法会有一定的重复，不妨反过来，对于 $f[s]$，考察其没有出现的每一个元素 $nums[j]$，更新 $f[s|nums[j]]$。这样可以大大减少重复。

> 时间复杂度：不超过 $O(n \cdot 2^n)$

```
class Solution:
    def canPartitionKSubsets(self, nums: List[int], k: int) -> bool:
        siz = sum(nums) // k
        if sum(nums) % k != 0 or any(x > siz for x in nums): return False
        # f[s] 表示 在选择状态为s的情况下，余数是多少
```

```
        n = len(nums)
        m = (1 << n) - 1
        f = [0] + [-inf] * m
        for s in range(m):
            if f[s] == -inf: continue
            for j in range(n):
                if (s >> j) & 1 == 0:
                    nx = s | (1 << j)
                    if f[nx] == 0: continue
                    if f[s] + nums[j] <= siz:
                        f[nx] = (f[s] + nums[j]) % siz
        return f[m] == 0
```

划分dp

约束划分个数

将数组分成 (恰好/至多) k 个连续子数组，计算与这些子数组有关的最优值。

类型1： $f[i][j]$ 当前考虑完前缀 $a[:i]$，且$a[:i]$ 恰好划分成 j 个连续子数组所得到的最优解。 枚举最后一个子数组的左端点 L, 从 $f[L][j-1]$ 转移到 $f[i][j]$，并考虑 $a[L:i]$ 对最优解的影响。 $f(i,j)=\min(f(L,j-1))$

类型2： $f(i,j,pre)$ 表示当前考虑到 $a[i]$， 且$a[:i]$ 的前缀中包含 j 个连续子数组所得的最优解，其中 pre 表示当前待划分的这段的状态。 考虑是否在 i 处划分，并考虑前一段状态 pre 是否允许划分。
$f(i,j,pre)=\min\{\ f(i+1,j,pre),\ f(i+1,j+1,pre')\ \}$

3117. 划分数组得到最小的值之和 - 力扣（LeetCode）

$f(i,j,pre\_and)$: 表示当前考虑到$nums[i]$，且前缀中包含 j 段，pre_and 表示当前待划分的这段的AND。

```
    def minimumValueSum(self, nums: List[int], andValues: List[int]) -> int:
        n, m = len(nums), len(andValues)
        @lru_cache(None)
        def f(i, j, pre_and):
            # 表示当前考虑到nums[i]，且前缀中包含j 段，pre_and表示当前待划分的这段的AND
            if i == n and j == m: return 0
            if i < n and j == m: return inf
            if i == n and j < m: return inf
            pre_and &= nums[i]
            # 在i处不划分，
            res = f(i + 1, j, pre_and)
            # 在i处划分，条件是这一段 pre_and == andValues[j]
            if pre_and == andValues[j]:
                res = min(res, f(i + 1, j + 1, -1) + nums[i])
            return res
        res = f(0, 0, -1)
        return res if res < inf else -1
```

时间复杂度：$O(mn\log U)$，由于 pre_and 表示当前待划分这段的按位与。记 $\log U$ 表示最大数对应的二进制位数。对于一个确定的 i，向前 AND 每次不变或者减少比特1的个数。所以不同的pre_and 数不超过 $logU$。总共有 $mn\log U$ 个状态，每个状态是 $O(1)$ 。

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

```python
    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。

```python
    def scheduleCourse(self, courses: List[List[int]]) -> int:
        # 按照截至日期排序
        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）

```python
    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）

```python
    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
```

消消乐贪心

配合哈希表 / 哈希集合，在 $O(n)$ 复杂度内，通过对乱序枚举到的每一个 x，贪心找出符合性质 / 限制的整组数据并且消除。

最长连续子序列

最长连续序列

给定一个未排序的整数数组 `nums`，找出数字连续的最长子序列。

对于任何一个数 x，向两边贪心找到相邻的这一组数，将其消除。

```
    def longestConsecutive(self, nums: List[int]) -> int:
        s = set(nums)
        res = 0
        for x in nums:
            if x not in s: continue
            cur = 1
            s.remove(x)
            y = x + 1
            while y in s:
                s.remove(y)
                cur, y = cur + 1, y + 1
            y = x - 1
```

```
            while y in s:
                s.remove(y)
                cur, y = cur + 1, y - 1
            res = max(res, cur)
        return res
```

2007. 从双倍数组中还原原数组 - 力扣（LeetCode）

对于任何一个数 x，如果是奇数则是最小出发数；否则向下贪心折半，直到得到最小出发数（奇数或者最小可达的偶数）。从最小出发数，出发，贪心删除整组数据。时间复杂度：$O(n)$

```
def findOriginalArray(self, changed: List[int]) -> List[int]:
        n = len(changed)
        if n & 1: return []
        res = []
        cnt = Counter(changed)
        for i, x in enumerate(changed):
            if cnt[x] == 0: continue
            if x == 0:
                if cnt[0] & 1: return []
                res.extend(cnt[0] //2 * [0])
                cnt[0] = 0
                continue
            while x & 1 == 0 and cnt[x // 2] > 0: x //= 2
            y = x
            while cnt[y] > 0:
                if cnt[y * 2] < cnt[y]: return []
                res.extend(cnt[y] * [y])
                cnt[y * 2] -= cnt[y]
                cnt[y] = 0
                if cnt[y * 2]: y = 2 * y
                else: y = 4 * y
        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 $(x\cos\alpha+y\sin\alpha,\quad y\cos\alpha\ -x\sin\alpha)$

证明：

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

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

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 $\sqrt{2}$。

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

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

当需要求若干点之间的最大 d_M 时，可以转换为

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

$$\iff\forall i,j\in P,\ \max(\max(|x'_i-x'_j|),\ \max(|y'_i-y'_j|))$$

3102. 最小化曼哈顿距离 - 力扣（LeetCode）

```python
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

```python
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))
```

杂项问题

$ceil$ 精度处理

同时存在除法和 $ceil$ 运算时， $ceil(a/b)$ 以及 $ceil(a/b+x)$ 操作会由于精度问题，导致偏差。

方法1：$ceil(x) = math.ceil(x - eps)$，其中 eps 是小常量，如10^{-8}

方法2：所有数乘 b，其中 $b \times ceil(a,b) = ((a-1)//b+1) \times b$，例如 $ceil(4,3) \times 3 = 2 \times 3 = 6$。

1883. 准时抵达会议现场的最小跳过休息次数 - 力扣（LeetCode）

$f(i,j) = \min\{f(i-1,j-1) + d[i]/s,\ ceil(f(i-1,j) + d[i]/s)\}$

方法1：

```
    eps = 1e-8
    def ceil(x):
        return math.ceil(x - eps)

    def minSkips(self, d: List[int], s: int, hoursBefore: int) -> int:
        n = len(d)
        if sum(d) > s * hoursBefore: return -1
        if n == 1: return 0 if d[0] <= s * hoursBefore else -1
        mx = sum(d) + n
        f = [[mx] * (n + 1) for _ in range(n + 1)]

        d = [D / s for D in d]
        f[0][0] = ceil(d[0])
        f[0][1] = d[0]
        for i in range(1, n - 1):
            for j in range(i + 2):
                f[i][j] = ceil(f[i - 1][j] + d[i])   # 不休息
                if j: f[i][j] = min(f[i][j], f[i - 1][j - 1] + d[i])    # 休息
        for k in range(n):
            if f[n - 2][k] + d[-1] <= hoursBefore:
                return k
```

方法2：

```
    def ceil(a, b):
        return ((a - 1) // b + 1) * b

    def minSkips(self, d: List[int], s: int, hoursBefore: int) -> int:
        n = len(d)
        if sum(d) > s * hoursBefore: return -1
        if n == 1: return 0 if d[0] <= s * hoursBefore else -1
```

```
        mx = sum(d) + n
        f = [[mx] * (n + 1) for _ in range(n + 1)]

        f[0][0] = ceil(d[0], s)
        f[0][1] = d[0]
        for i in range(1, n - 1):
            for j in range(i + 2):
                f[i][j] = ceil(f[i - 1][j] + d[i], s)
                if j: f[i][j] = min(f[i][j], f[i - 1][j - 1] + d[i])
        for k in range(n):
            if f[n - 2][k] + d[-1] <= hoursBefore * s:
                return k
```

灵茶の试炼 (qq.com)