Python基础语法

1.列表

int 转 list

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

list(int) 转 int

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

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

列表特性

比较大小的时候，不管长度如何，依次比较到第一个元素不相等的位置

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

嵌套列表推导：展平二维数组

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

2.Deque

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

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

3.字典

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

字典推导器

字母表对应下标

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

也可以使用zip初始化dic

[2606. 找到最大开销的子字符串 - 力扣（LeetCode）](#)

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

4.map映射函数

用法:

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

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

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

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

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

5. 自定义Set规则

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

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

字符串

1.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
```

##

2.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)]

```python
# 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]
```

3. 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。
> ```

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

字符串API

- s1.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。

区间合并

```python
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
```

回溯/递归

套路：

1. 当前子问题？
2. 当前操作？
3. 下一个子问题？

LCR 086. 分割回文串 - 力扣（LeetCode）

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

离散化

二分写法

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

二分 + 还原

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

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]
```

二分查找

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

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

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

1.多维二分

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

2.二分答案

正难则反思想，二分答案一般满足两个条件：

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

check(x) 函数对单调x 进行检验。

```python
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

```python
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）]()

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

3. 朴素二分

在 闭区间[a, b]上二分

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

前缀异或

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

优先队列

```python
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
```

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

自定义比较规则

```python
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

单调栈

```python
    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
```

单调队列

[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
```

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
```

数学

1.取整函数性质

(1). 上下取整转换

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

(2). 取余性质

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

(3). 幂等律

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

2. 素数筛

埃氏筛：nloglogn

```
is_prime = [True] * MX # MX为最大可能遇到的质数 + 1
is_prime[1] = is_prime[0] = False
for i in range(2, isqrt(MX) + 1):
    if is_prime[i]:
        for j in range(i * i, MX, i):
            is_prime[j] = False
```

切片优化

```
is_prime = [True] * MX
is_prime[0] = is_prime[1] = False
for i in range(2, isqrt(MX) + 1):
    if is_prime[i]: # [i * i, n] -> [i * i, n + 1) -> [i * i, MX)
        is_prime[i * i::i] = [False] * ((MX - 1 - i * i) // i + 1)
```

3.其他简单数学

1.判断回文

```
def is_par(x: int) -> bool:
    return x == int(str(x)[::-1])
```

2.pow函数

求$a^b \mod c$:

```
pow(a, b, c)
```

3.求和公式

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

4. 取模性质

模运算与基本四则运算有些相似，但是除法例外。其规则如下：
(a + b) % p = (a % p + b % p) % p
(a - b) % p = (a % p - b % p) % p
(a * b) % p = (a % p * b % p) % p
a ^ b % p = ((a % p)^b) % p

结合律：
((a+b) % p + c) % p = (a + (b+c) % p) % p
((ab) % p * c)% p = (a * (bc) % p) % p
交换律：
(a + b) % p = (b+a) % p
(a * b) % p = (b * a) % p
分配律：
(a+b) % p = (a % p + b % p) % p
((a +b)% p * c) % p = ((a * c) % p + (b * c) % p) % p

4.数学公式

(1).排序不等式

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

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

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

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

(2). 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：

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

5.矩阵乘法/矩阵快速幂

> 矩阵乘法时间复杂度：$O(n^3)$

矩阵乘法

```
moder = 10**9 + 7

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

矩阵快速幂

```
moder = 10**9 + 7
```

```python
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
```

数据结构

26叉字典树

```python
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
```

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)
```

```python
            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)
```

```python
    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)
```

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

```python
# 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（内部）

```python
n = 0
tree, lazy = [], []
Nums = []

def build(i, l, r):
    if l == r:
        tree[i] = Nums[l - 1]
        return
    mid = (l + r) >> 1
    build(i * 2, l, mid)
    build(i * 2 + 1, mid + 1, r)
    tree[i] = tree[i * 2] + tree[i * 2 + 1]

# 节点区间赋值、打上lazy标记
def do(i, l, r, val):
    tree[i] = (l - r + 1) * val
    lazy[i] = val

# 根据标记信息，更新子节点，设置子节点标记，清空标记
def pushdown(i, l, r):
    if lazy[i]:
        val = lazy[i]
        mid = (l + r) >> 1
        do(i * 2, l, mid, val)
        do(i * 2 + 1, mid + 1, r, val)
        lazy[i] = val

def Update(L, R, val, i = 1, l = 1, r = n):
    if L <= l and r <= R:
        do(i, l, r, val)
        return

    # 检查标记
    pushdown(i, l, r)
    mid = (l + r) >> 1
    if L <= mid:
        Update(L, R, val, i * 2, l, mid)
    if R > mid:
        Update(L, R, val, i * 2 + 1, mid + 1, r)

    # 更新节点区间
    tree[i] = tree[i * 2] + tree[i * 2 + 1]


def Query(L, R, i, l, r) -> int:
    if L <= l and r <= R:
        return tree[i]

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


class NumArray:
    def __init__(self, nums: List[int]):
```

```python
        global n, tree, lazy, Nums
        n = len(nums)
        tree = [0] * (4 * n)
        lazy = [0] * (4 * n)
        Nums = nums
        build(1, 1, n)

    def update(self, index: int, val: int) -> None:
        Update(index + 1, index + 1, val, 1, 1, n)

    def sumRange(self, left: int, right: int) -> int:
        return Query(left + 1, right + 1, 1, 1, n)
```

lazy线段树（点区间赋值）

```python
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)
```

```python
        # 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）

```python
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:
```

```python
                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
```

```python
        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

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

图论/树

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))
```

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
```

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)]
```

```python
        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

```python
    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。

```python
    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]
```

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
```

```python
    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]
```

树上差分

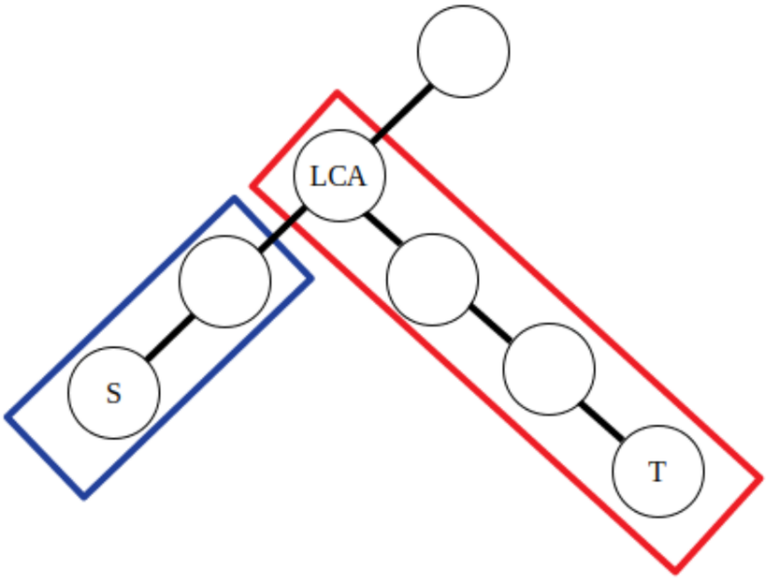
点差分：解决多路径节点计数问题。

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

```python
# 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
```

$$d_s \leftarrow d_s + 1$$
$$d_{lca} \leftarrow d_{lca} - 1$$
$$d_t \leftarrow d_t + 1$$
$$d_{f(lca)} \leftarrow d_{f(lca)} - 1$$

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



树形DP(换根DP)

[834. 树中距离之和 - 力扣（LeetCode）](#)

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

1，指定某个节点为根节点。

2，第一次搜索完成预处理（如子树大小等），同时得到该节点的解。

3，第二次搜索进行换根的动态规划，由已知解的节点推出相连节点的解。

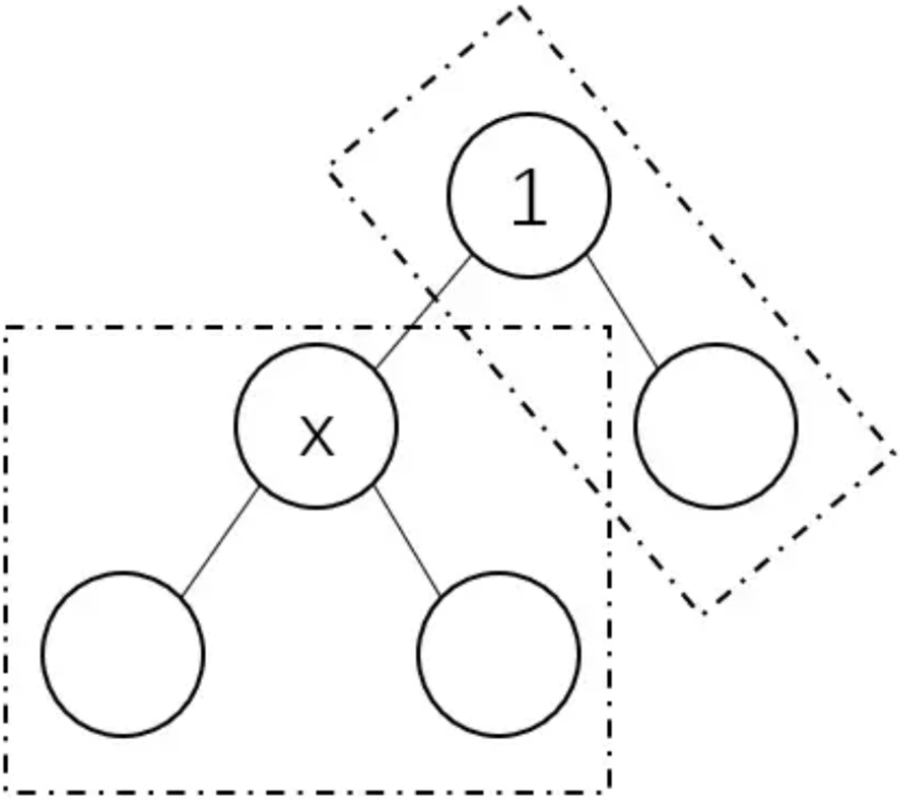
```
    def sumOfDistancesInTree(self, n: int, edges: List[List[int]]) -> List[int]:
        g = [[] for _ in range(n)]
        dep = [0] * n
        siz = [1] * n
        res = [0] * n
        for u, v in edges:
            g[u].append(v)
            g[v].append(u)

        def dfs1(u, fa):     # 预处理深度
            dep[u] = dep[fa] + 1 if fa != -1 else 0
            for v in g[u]:
                if v != fa:
                    dfs1(v, u)
                    siz[u] += siz[v]
        def dfs2(u, fa):
            for v in g[u]:
                if v != fa:
                    res[v] = res[u] - siz[v] + (n - siz[v])
                    dfs2(v, u)
        dfs1(0, -1)
```

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

u剔除v子树部分下降1，深度和增加$n - siz[v]$

v子树部分上升1，深度和减少$siz[v]$

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



并查集

`find(u) == find(v)` 表示u, v在同一集合

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

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）

```
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
```

位运算/状态压缩

1.二维矩阵 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
```

2.枚举一个二进制串的子集

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

3.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
```

4.二进制

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()
```

5.最大异或

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

6.常用位运算操作

(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)
```

7. 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。

```
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/)

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

```
        mask = res = 0
        for b in range(n, -1, -1):
            mask |= 1 << b
            ans_res = -1     # 初始值全是1
```

```python
            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
```

数位dp

```python
class Solution:
    def numberOfPowerfulInt(self, start: int, finish: int, limit: int, s: str) -> int:
        low = str(start)
        high = str(finish)
        n = len(high)
        low = '0' * (n - len(low)) + low # 补全前导0
        diff = n - len(s)

        @lru_cache(maxsize = None)
        def dfs(i, limit_low: bool, limit_high: bool) -> int:
            if i == n:
                return 1
            lo = int(low[i]) if limit_low else 0
            hi = int(high[i]) if limit_high else 9
            res = 0
            if i < diff:    # 枚举这个位填什么
                for d in range(lo, min(hi, limit) + 1):
                    res += dfs(i + 1, limit_low and d == lo, limit_high and d == hi)
            else:
                x = int(s[i - diff])
                if lo <= x <= min(hi, limit):
                    res = dfs(i + 1, limit_low and x == lo, limit_high and x == high)
            return res
        return dfs(0, True, True)
```

状态机dp

3068. 最大节点价值之和 - 力扣（LeetCode）

0 表示当前异或偶数个k，1表示当前异或奇数个k

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

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

```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]
```

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）

```python
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
```

```python
            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）](#)

```python
    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
```