模拟插板，但是二分

```
n, m = map(int, input().split())
L = list(int(input()) for x in range(n))


def check(x):
    num, cut = 1, 0
    for i in range(n):
        if cut + L[i] > x:
            num += 1
            cut = L[i]   # 在 L[i]左边插一个板，L[i]属于新的 fajo 月
        else:
            cut += L[i]
    return num <= m


maxmax = sum(L)
minmax = max(L)
while minmax < maxmax:
    middle = (maxmax + minmax) // 2
    if check(middle):   # 表明这种插法可行，那么看看更小的插法可不可以
        maxmax = middle
    else:
        minmax = middle + 1   # 这种插法不可行，改变 minmax 看看下一种插法可不可以
print(maxmax)
```

OJ08210:河中跳房子

http://cs101.openjudge.cn/2024sp_routine/08210/

排序

归并排序（Merge Sort）

基础知识

时间复杂度：

-

最坏情况: O(nlogn)

-
-

平均情况: O(nlogn)

-
-

最优情况: O(nlogn)

-
-

空间复杂度: O(n) — 需要额外的内存空间来存储临时数组。

稳定性: 稳定 — 相同元素的相对顺序在排序后不会改变。

应用

计算逆序对数：在一个数组中，如果前面的元素大于后面的元素，则这两个元素构成一个逆序对。归并排序可以在排序过程中修改并计算逆序对的总数。这通过在归并过程中，每当右侧的元素先于左侧的元素被放置到结果数组时，记录左侧数组中剩余元素的数量来实现。

排序链表：归并排序在链表排序中特别有用，因为它可以实现在链表中的有效排序而不需要额外的空间，这是由于链表的节点可以通过改变指针而不是实际移动节点来重新排序。

代码示例

对链表进行排序

```
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next


def split_list(head):
    if not head or not head.next:
        return head

    # 使用快慢指针找到中点
    slow = head
    fast = head.next   # fast 从 head.next 开始确保分割平均
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # 分割链表为两部分
    mid = slow.next
    slow.next = None

    return head, mid


def merge_sort(head):
    if not head or not head.next:
        return head

    left, right = split_list(head)
    left = merge_sort(left)
```

```
    right = merge_sort(right)
    return merge_lists(left, right)


# 创建链表：4 -> 2 -> 1 -> 3
head = ListNode(4, ListNode(2, ListNode(1, ListNode(3))))

# 排序链表
sorted_head = merge_sort(head)

# 打印排序后的链表
current = sorted_head
while current:
    print(current.value, end=" -> ")
    current = current.next
print("None")
```

OJ02299:Ultra-QuickSort

http://cs101.openjudge.cn/2024sp_routine/02299/

与 20018: 蚂蚁王国的越野跑（http://cs101.openjudge.cn/2024sp_routine/20018/）类似。

算需要交换多少次来得到一个排好序的数组，其实就是算逆序对。

```
d = 0


def merge(arr, l, m, r):
    """对 l 到 m 和 m 到 r 两段进行合并"""
    global d
    n1, n2 = m - l + 1, r - m    # L1 和 L2 的长
    L1, L2 = arr[l:m + 1], arr[m + 1:r + 1]
    # L1 和 L2 均为有序序列
    i, j, k = 0, 0, l    # i 为 L1 指针，j 为 L2 指针，k 为 arr
指针
    '''双指针法合并序列'''
    while i < n1 and j < n2:
        if L1[i] <= L2[j]:
            arr[k] = L1[i]
            i += 1
        else:
            arr[k] = L2[j]
            d += n1 - i    # 精髓所在
            j += 1
        k += 1
    while i < n1:
        arr[k] = L1[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = L2[j]
        j += 1
        k += 1


def mergesort(arr, l, r):
    """对 arr 的 l 到 r 一段进行排序"""
    if l < r:    # 递归结束条件，很重要
        m = (l + r) // 2
        mergesort(arr, l, m)
        mergesort(arr, m + 1, r)
        merge(arr, l, m, r)


while True:
    n = int(input())
    if n == 0:
        break
    array = []
    for b in range(n):
        array.append(int(input()))
    d = 0
    mergesort(array, 0, n - 1)
    print(d)
```

快速排序（Quick Sort）

时间复杂度

最坏情况: O(n2) — 通常发生在已经排序的数组或基准选择不佳的情况下。

平均情况: O(nlogn)

最优情况: O(nlogn) — 适当的基准可以保证分割平衡。

空间复杂度: O(logn) — 主要是递归的栈空间。

稳定性: 不稳定 — 基准点的选择和划分过程可能会改变相同元素的相对顺序。

应用：k-th 元素

代码示例

普通快排

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[-1]
        less = [x for x in arr[:-1] if x <= pivot]
        greater = [x for x in arr[:-1] if x > pivot]
```

```
        return quicksort(less) + [pivot] + quicksort(greater)

# 示例数组
array = [10, 7, 8, 9, 1, 5]
sorted_array = quicksort(array)
print(sorted_array)
```

在无序列表中选择第 k 大

```
def partition(nums, left, right):
    pivot = nums[right]
    i = left
    for j in range(left, right):
        if nums[j] > pivot:   # 注意这里是寻找第 k 大，所以使用大于号
            nums[i], nums[j] = nums[j], nums[i]
            i += 1
    nums[i], nums[right] = nums[right], nums[i]
    return i

def quickselect(nums, left, right, k):
    if left == right:
        return nums[left]
    pivot_index = partition(nums, left, right)
    if k == pivot_index:
        return nums[k]
    elif k < pivot_index:
        return quickselect(nums, left, pivot_index - 1, k)
    else:
        return quickselect(nums, pivot_index + 1, right, k)

def find_kth_largest(nums, k):
    return quickselect(nums, 0, len(nums) - 1, k - 1)
```

堆排序（Heap Sort）

时间复杂度

-

最坏情况: O(nlogn)

-
-

平均情况: O(nlogn)

-
-

最优情况: O(nlogn)

-
-

空间复杂度: O(1) — 堆排序是原地排序算法，不需要额外的存储空间。

-
-

稳定性：不稳定 — 堆的维护过程可能会改变相同元素的原始相对顺序。

线性表

三种表达式的求值及相互转化

求值：

前序表达式（波兰表达式）：栈（最好从右向左读，但是反过来也可）

后序表达式（逆波兰表达式）：栈（从左向右读）

中序表达式：Shunting Yard Algorithm

前序表达式和后序表达式向其他任何一种转化：建树

中序表达式->后序表达式：Shunting Yard Algorithm+建树

中序表达式->前序表达式：Shunting Yard Algorithm+建树

OJ24591：中序表达式转后序表达式

http://cs101.openjudge.cn/practice/24591/

栈的经典题目，算法是 Shunting Yard Algorithm，两个栈（实际上是一个），一个运算符栈，一个输出栈（这个不用栈保存，直接输出也可）

但是用二叉树来说更好理解，叶节点是数字，非叶节点是运算符，需要把一个中序遍历转换为后序遍历

一开始想把表达式建成二叉树，但是有点麻烦，而且之前在 27637:括号嵌套二叉树等题目中练过了这种递归建树，就试着用栈写

Shunting Yard Algorithm 的理解：

从左到右遍历中序表达式，

若遇到数字，直接加到输出栈，因为后序遍历中，左右叶节点是最先的

若遇到左括号，加入运算符栈，因为左括号是要建立单独的树，是运算符优先级的一种区分

若遇到右括号，从最后一个开始，将运算符中的东西弹出并加入输出栈，直到遇到左括号，因为这代表这一整个子树的建立。

若遇到运算符，则碰到了非叶节点，弹出栈中任何优先级比当前运算符更高或与当前运算符相等（优先级更高或相等代表子树深度小，所以先输出）的运算符，并将它们添加到输出队列中，然后将自己添加到运算符栈，等待右子树

```python
operators = ['+', '-', '*', '/']


def is_num(s):
    for i in operators + ['(', ')']:
        if i in s:
            return False
    return True


def process(raw_input):
    # convert the raw input into separated sequence
    temp, ans = '', []
    for i in raw_input.strip():
        if is_num(i):
            temp += i
        else:
            if temp:
                ans.append(temp)
            ans.append(i)
            temp = ''
    if temp:
        ans.append(temp)
    return ans


def infix_to_postfix(expression):
    # Shunting Yard Algorithm
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    output_stack, op_stack = [], []
    for i in expression:
        if is_num(i):
            output_stack.append(i)
        elif i == '(':
            op_stack.append(i)
        elif i == ')':
            while op_stack[-1] != '(':
                output_stack.append(op_stack.pop())
            op_stack.pop()
        else:
            while op_stack and op_stack[-1] in operators and precedence[i] <= precedence[op_stack[-1]]:
                output_stack.append(op_stack.pop())
            op_stack.append(i)
    if op_stack:
        output_stack += op_stack[::-1]
    return output_stack


n = int(input())
for i in range(n):
    tokenized = process(input())
    print(' '.join(infix_to_postfix(tokenized)))
```

单调队列

OJ26978:滑动窗口最大值

http://cs101.openjudge.cn/2024sp_routine/26978/

```python
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        n = len(nums)
        q = collections.deque()
        for i in range(k):
            while q and nums[i] >= nums[q[-1]]:
                q.pop()
            q.append(i)

        ans = [nums[q[0]]]
        for i in range(k, n):
            while q and nums[i] >= nums[q[-1]]:
                q.pop()
            q.append(i)
            while q[0] <= i - k:
                q.popleft()
            ans.append(nums[q[0]])

        return ans
```

单调栈

OJ28203:【模板】单调栈

http://cs101.openjudge.cn/practice/28203/

给出项数为 n 的整数数列 a1...an。定义函数 f(i) 代表数列中第 i 个元素之后第一个大于 ai 的元素的下标。若不存在，则 f(i)=0。试求出 f(1...n)。

输入:第一行一个正整数 n。第二行 n 个正整数 a1...an 。

输出:一行 n 个整数表示 f(1), f(2), ..., f(n) 的值。

```python
n = int(input())
a = list(map(int, input().split()))
stack = []
for i in range(n):
    while stack and a[stack[-1]] < a[i]:
        a[stack.pop()] = i + 1
    stack.append(i)
while stack:
    a[stack[-1]] = 0
    stack.pop()
print(*a)
```

OJ04137:最小新整数

http://cs101.openjudge.cn/practice/04137/

```python
def removeKDigits(num, k):
    stack = []
    for digit in num:
        while k and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)
    while k:
        stack.pop()
        k -= 1
```

```
    return int(''.join(stack))
t = int(input())
results = []
for _ in range(t):
    n, k = input().split()
    results.append(removeKDigits(n, int(k)))
for result in results:
    print(result)
```

OJ27205:护林员盖房子 加强版

http://cs101.openjudge.cn/2024sp_routine/27205/

题解：https://zhuanlan.zhihu.com/p/162834671

```
def maximalRectangle(matrix) -> int:
    if (rows := len(matrix)) == 0:
        return 0

    cols = len(matrix[0])
    # 存储每一层的高度
    height = [0 for _ in range(cols + 1)]
    res = 0

    for i in range(rows):  # 遍历以哪一层作为底层
        stack = [-1]
        for j in range(cols + 1):
            # 计算 j 位置的高度，如果遇到 1 则置为 0，否则递增
            h = 0 if j == cols or matrix[i][j] == '1' else height[j] + 1
            height[j] = h
            # 单调栈维护长度
            while len(stack) > 1 and h < height[stack[-1]]:
                res = max(res, (j - stack[-2] - 1) * height[stack[-1]])
                stack.pop()
            stack.append(j)
    return res


rows, _ = map(int, input().split())
a = [input().split() for _ in range(rows)]

print(maximalRectangle(a))
```

辅助栈

OJ22067:快速堆猪

http://cs101.openjudge.cn/2024sp_routine/22067/

```
a = []
m = []

while True:
    try:
        s = input().split()

        if s[0] == "pop":
            if a:
                a.pop()
                if m:
                    m.pop()
        elif s[0] == "min":
            if m:
                print(m[-1])
        else:
            h = int(s[1])
            a.append(h)
            if not m:
                m.append(h)
            else:
                k = m[-1]
                m.append(min(k, h))
    except EOFError:
        break
```

散列表

OJ17968:整型关键字的散列映射

http://cs101.openjudge.cn/2024sp_routine/17968/

```
import sys
input = sys.stdin.read

data = input().split()
index = 0
N = int(data[index])
index += 1
M = int(data[index])
index += 1


k = [0.5] * M
l = list(map(int, data[index:index + N]))

ans = []
for u in l:
    t = u % M
    i = t
    while True:
        if k[i] == 0.5 or k[i] == u:
            ans.append(i)
            k[i] = u
            break
        i = (i + 1) % M

print(*ans)
```

OJ17975:用二次探查法建立散列表

http://cs101.openjudge.cn/2024sp_routine/17975/

```
import sys
input = sys.stdin.read
data = input().split()
index = 0
n = int(data[index])
index += 1
m = int(data[index])
index += 1
num_list = [int(i) for i in data[index:index+n]]
mylist = [0.5] * m
```

```
def generate_result():
    for num in num_list:
        pos = num % m
        current = mylist[pos]
        if current == 0.5 or current == num:
            mylist[pos] = num
            yield pos
        else:
            sign = 1
            cnt = 1
            while True:
                now = pos + sign * (cnt ** 2)
                current = mylist[now % m]
                if current == 0.5 or current == num:
                    mylist[now % m] = num
                    yield now % m
                    break
                sign *= -1
                if sign == 1:
                    cnt += 1
result = generate_result()
print(*result)
```

树

树的四种遍历及其相互转化

给出二叉树的前序、中序、后序遍历中的两种，建树或者求出另外一种，算法实现麻烦但是思路简单。

树的 bfs 遍历（或者叫层次遍历，但是我不喜欢这个叫法）和队列有奇妙的联系，见例题

OJ22158:根据二叉树前中序序列建树

http://cs101.openjudge.cn/2024sp_routine/22158/

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    def post_order(self):
        post = ''
        if self.left:
            post += self.left.post_order()
        if self.right:
            post += self.right.post_order()
        return post + self.val


def build(pre_order, in_order):
    if not pre_order or not in_order:
        return None
    #print(pre_order, in_order)
    if len(pre_order) == 1:
        return Node(pre_order[0])
    root_val = pre_order[0]
    div = 0
    while in_order[div] != root_val:
        div += 1
    left_in_order = in_order[: div]
    right_in_order = in_order[div+1:]
    div = 1
    while pre_order[div] in left_in_order:
        div += 1
        if div >= len(pre_order):
            div = len(pre_order)
            break
    return Node(root_val, left=build(pre_order[1: div],
left_in_order), right=build(pre_order[div:], right_in_order))


while True:
    try:
        p, i = input(), input()
        tree = build(p, i)
        print(tree.post_order())
    except EOFError:
        break
```

OJ24750:根据二叉树中后序序列建树

http://cs101.openjudge.cn/dsapre/24750/

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    def post_order(self):
        post = ''
        if self.left:
            post += self.left.post_order()
        if self.right:
            post += self.right.post_order()
        return post + self.val


def build(pre_order, in_order):
    if not pre_order or not in_order:
        return None
    #print(pre_order, in_order)
    if len(pre_order) == 1:
        return Node(pre_order[0])
    root_val = pre_order[0]
    div = 0
    while in_order[div] != root_val:
        div += 1
    left_in_order = in_order[: div]
    right_in_order = in_order[div+1:]
    div = 1
    while pre_order[div] in left_in_order:
        div += 1
        if div >= len(pre_order):
            div = len(pre_order)
            break
    return Node(root_val, left=build(pre_order[1: div],
left_in_order), right=build(pre_order[div:], right_in_order))


def build_tree(inorder, postorder):
    if not inorder or not postorder:
        return []
```

```
    root_val = postorder[-1]
    root_index = inorder.index(root_val)

    left_inorder = inorder[:root_index]
    right_inorder = inorder[root_index + 1:]

    left_postorder = postorder[:len(left_inorder)]
    right_postorder = postorder[len(left_inorder):-1]

    root = [root_val]
    root.extend(build_tree(left_inorder, left_postorder))
    root.extend(build_tree(right_inorder, right_postorder))

    return root


def main():
    inorder = input().strip()
    postorder = input().strip()
    preorder = build_tree(inorder, postorder)
    print(''.join(preorder))


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

OJ25140:根据后序表达式建立表达式树

http://cs101.openjudge.cn/2024sp_routine/25140/

```
class Node:
    def __init__(self, name, left=None, right=None):
        self.name = name
        self.left = left
        self.right = right


def build(s):
    stack = []
    for i in s:
        if ord(i) > ord('Z'):
            stack.append(Node(i))
        else:
            r, l = stack.pop(), stack.pop()
            stack.append(Node(i, l, r))
    return stack[0]


for _ in range(int(input())):
    s = input()
    tree = build(s)
    bfs = [tree]
    ans = ''
    while bfs:
        now = bfs.pop(0)
        ans += now.name
        if now.left:
            bfs.append(now.left)
        if now.right:
            bfs.append(now.right)
    print(ans[::-1])
```

并查集

并查集（Union-Find 或 Disjoint Set Union，简称 DSU）是一种处理不交集合的合并及查询问题的数据结构。它支持两种操作：

- Find：确定某个元素属于哪一个子集。这个操作可以用来判断两个元素是否属于同一个子集。
- Union：将两个子集合并成一个集合。

使用场景

并查集常用于处理一些元素分组情况，可以动态地连接和判断连接，广泛应用于网络连接、图的连通分量、最小生成树等问题。

核心思想

并查集通过数组或者特殊结构存储每个元素的父节点信息。初始时，每个元素的父节点是其自身，表示每个元素自成一个集合。通过路径压缩和按秩合并等优化策略，可以提高并查集的效率。

- 路径压缩：在执行 Find 操作时，使得路径上的所有点直接指向根节点，这样可以减少后续操作的时间复杂度。
- 按秩合并：在执行 Union 操作时，总是将较小的树连接到较大的树的根节点上，这样可以避免树过深，影响操作效率。

代码示例

```
class UnionFind:
    # 初始化
    def __init__(self, size):
        # 将每个节点的上级设置为自己
        self.parent = list(range(size))
        # 每个节点的秩都是 0
        self.rank = [0] * size

    # 查找
    def find(self, p):
        if self.parent[p] != p:
            # 这一步进行了路径压缩。
            # 如果不进行路径压缩，这一步是 return
```

```
self.find(self.parent[p])
            self.parent[p] = self.find(self.parent[p])
        return self.parent[p]

    # 合并
    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP != rootQ:
            # 按秩合并，总是将较小的树连接到较大的树的根节点上
            if self.rank[rootP] > self.rank[rootQ]:
                self.parent[rootQ] = rootP
            elif self.rank[rootP] < self.rank[rootQ]:
                self.parent[rootP] = rootQ
            else:
                # 如果两个节点的秩相等，就无所谓
                self.parent[rootQ] = rootP
                # 但这时需要把连接后较大的节点的秩+1
                self.rank[rootP] += 1

    # 是否属于同一集合
    def connected(self, p, q):
        return self.find(p) == self.find(q)
```

例题

OJ02524:宗教信仰

http://cs101.openjudge.cn/dsapre/02524/

最基本的应用，只是最后多了一步看看有多少个集合。

```
class UnionFind:
    def __init__(self, size):
        self.parent = [i for i in range(size + 1)]
        self.rank = [0] * (size + 1)

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

    def union(self, x, y):
        x_parent = self.find(x)
        y_parent = self.find(y)
        if x_parent != y_parent:
            if self.rank[x_parent] > self.rank[y_parent]:
                self.parent[y_parent] = x_parent
            elif self.rank[x_parent] < self.rank[y_parent]:
                self.parent[x_parent] = y_parent
            else:
                self.parent[y_parent] = x_parent
                self.rank[x_parent] += 1


n_case = 0
while True:
    n_case += 1
    n, m = map(int, input().split())
    if m == 0 and n == 0:
        break
    uf = UnionFind(n)
    for i in range(m):
        a, b = map(int, input().split())
        uf.union(a, b)
    cnt = set([uf.find(i) for i in uf.parent])   # 这一步是多的
    print(f'Case {n_case}:', len(cnt) - 1)
```

OJ18250:冰阔落 I

http://cs101.openjudge.cn/2024sp_routine/18250/

这题一开始 WA，后来检查，发现原因是按秩合并时，parent[x]不一定更新了。虽然最后用 self.find(x)又压缩了一次，仍然可能指向的不是最深的节点。好在此题数据小，无需按秩合并。

```
class DJS:
    def __init__(self, size):
        self.parent = [i for i in range(size + 1)]
        self.rank = [0 for _ in range(size + 1)]

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

    def union(self, a, b):
        root_a = self.find(a)
        root_b = self.find(b)
        self.parent[root_b] = root_a
        """
        if root_b != root_a:
            if self.rank[root_b] == self.rank[root_a]:
                self.parent[root_b] = root_a
                self.rank[root_a] += 1
            elif self.rank[root_b] > self.rank[root_a]:
                self.parent[root_a] = root_b
            else:
                self.parent[root_b] = root_a
        """

    def check(self, a, b):
        if self.find(a) == self.find(b):
            print('Yes')
        else:
            print('No')


while True:
    try:
        n, m = map(int, input().split())
    except EOFError:
        break
    d = DJS(n)
    for _ in range(m):
        x, y = map(int, input().split())
        d.check(x, y)
        d.union(x, y)
    cnt = 0
    ans = []
    for i in range(1, n + 1):
        if d.find(i) == i:
```

```
        cnt += 1
        ans.append(i)
    print(len(ans))
    print(*ans)
```

OJ01703:发现它，抓住它

这题一开始没想出来，因为给出的条件是某两个节点属于不同的集合，而非相同的集合。但是由于一共只有两个集合，所以可以创建一个长度为 2n 的数组，parent[x]是和 x 同类的，parent[x+n]是和 x 不同的。

思路很新颖，值得学习。

http://cs101.openjudge.cn/2024sp_routine/01703/

```
class DJS:
    def __init__(self, size):
        self.parent = [i for i in range(size + 1)]
        self.rank = [0 for _ in range(size + 1)]

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

    def union(self, a, b):
        root_a = self.find(a)
        root_b = self.find(b)
        if root_b != root_a:
            if self.rank[root_b] == self.rank[root_a]:
                self.parent[root_b] = root_a
                self.rank[root_a] += 1
            elif self.rank[root_b] > self.rank[root_a]:
                self.parent[root_a] = root_b
            else:
                self.parent[root_b] = root_a

    def check(self, a, b):
        if self.find(a) == self.find(b):
            print('Yes')
        else:
            print('No')


for _ in range(int(input())):
    n, m = map(int, input().split())
    d = DJS(2 * n)
    for _ in range(m):
        info = input().split()
        a, b = map(int, info[1:])
        if info[0] == 'A':
            if d.find(a) == d.find(b) or d.find(a + n) ==
d.find(b + n):
                print('In the same gang.')
            elif d.find(a + n) == d.find(b) or d.find(a) ==
d.find(b + n):
                print('In different gangs.')
            else:
                print('Not sure yet.')
        else:
            d.union(a, b + n)
            d.union(a + n, b)
```

OJ01182:食物链

http://cs101.openjudge.cn/2024sp_routine/01182/

和上一题很像

```
n, k = map(int, input().split())
cnt = 0
ds = []   #  本身，被 x 吃，吃 x

for i in range(3 * n + 1):
    ds.append(i)


def find(a):
    # print(a)
    if ds[a] != a:
        ds[a] = find(ds[a])
    return ds[a]


def union(a, b):
    root_a, root_b = find(a), find(b)
    ds[root_a] = root_b


def check(d, a, b):
    if d == 1:
        return find(a + n) == find(b) or find(b + n) == find(a)
    else:
        return find(a) == find(b) or find(b) == find(a + 2 * n)


for _ in range(k):
    d, x, y = map(int, input().split())
    if x > n or y > n or check(d, x, y):
        cnt += 1
        continue
    if d == 1:
        for i in range(3):
            union(x + i * n, y + i * n)
    elif d == 2:
        union(y, x + n)
        union(x, y + 2 * n)
        union(x + 2 * n, y + n)
print(cnt)
```

Trie（字典树）

OJ04089:电话号码

```
def build_trie(s, parent: dict):
    if s[0] in parent.keys():
        if len(s) == 1:
            return False
        if not parent[s[0]]:
            return False
        return build_trie(s[1:], parent[s[0]])
    parent[s[0]] = {}
    if len(s) == 1:
        return True
    return build_trie(s[1:], parent[s[0]])
t = int(input())
```

```
for i in range(t):
    trie = {}
    n = int(input())
    flag = False
    for j in range(n):
        number = input()
        if flag:
            continue
        if not build_trie(number, trie):
            print('NO')
            flag = True
    if not flag:
        print('YES')
```

堆

堆（Heap）是一种特别的完全二叉树。所有的节点都大于等于（最大堆）或小于等于（最小堆）每个它的子节点。本文将主要讨论最小堆的实现，其中每个父节点的值都小于或等于其子节点的值。

最小堆的实现原理

最小堆通常可以用一个数组来实现，利用数组的索引来模拟树结构：

- 根节点位于数组的第一个位置，即 `index = 0`。
- 对于任意位于 index = i 的节点：
  - 其左子节点的位置是 `2 * i + 1`
  - 其右子节点的位置是 `2 * i + 2`
  - 其父节点的位置是 `(i - 1) / 2`（这里的除法为整数除法）

最小堆的核心操作

1. 插入操作（Add）
2.
   - 将新元素添加到数组的末尾。
   - 从这个新元素开始，向上调整堆，以保持最小堆的性质。这通常被称为“上浮”（bubble up 或 percolate up），即如果添加的元素小于其父节点，则与父节点交换位置，重复这一过程直到恢复堆的性质或者该节点成为根节点。

删除最小元素（Extract Min）

- 最小元素总是位于数组的第一个位置。
- 将数组最后一个元素移动到第一个位置，然后从根节点开始向下调整堆（下沉或 percolate down）。如果父节点大于任一子节点，则与最小的子节点交换位置，重复这一过程直到恢复堆的性质或者该节点成为叶节点。

获取最小元素（Find Min）

- 由于最小元素总是位于数组的第一位置，因此获取最小元素非常高效，时间复杂度为 O(1)。

堆化（Heapify）

- 将一个不满足最小堆性质的数组转换成最小堆。这通常通过从最后一个非叶子节点开始，依次对每个节点执行“下沉”操作来实现。非叶子节点的开始位置可以从 `n/2 - 1` 开始（`n` 是数组长度），这是因为所有更后面的节点都是叶子节点，已经满足堆的性质。

性能

插入和删除操作的时间复杂度通常是 O(log n)，因为需要在树的高度上进行操作（上浮或下沉），而树的高度与节点数的对数成正比。

-
-

堆化操作的时间复杂度是 O(n)，这是通过精心构造的下沉操作实现的，虽然看起来每个节点都要处理，但实际上更深的节点较少，处理起来也更快。

-

OJ04078:实现堆结构

手动实现最小堆

```
from math import floor as floor


class MinHeap:
    def __init__(self):
        self.value = []

    def get_min(self):
        if not self.value:
            return None
        return self.value[0]

    def swap(self, a, b):
        self.value[a], self.value[b] = self.value[b], self.value[a]

    def insert(self, x):
        self.value.append(x)
        index = len(self.value) - 1
        while True:
            parent_index = floor((index - 1) / 2)
            if index <= 0 or self.value[parent_index] <= self.value[index]:
                return
            self.swap(index, parent_index)
            index = parent_index

    def delete_min(self):
        if not self.value:
            return
        self.swap(0, -1)
        self.value.pop()
        index = 0
        while index < len(self.value):
            left_index, right_index = 2 * index + 1, 2 * index + 2
            if left_index >= len(self.value):
                return
            if right_index >= len(self.value):
                if self.value[index] > self.value[left_index]:
                    self.swap(index, left_index)
                    continue
                else:
                    return
            if self.value[index] < min(self.value[left_index], self.value[right_index]):
                return
            small = left_index if self.value[left_index] < self.value[right_index] else right_index
            self.swap(small, index)
            index = small


heap = MinHeap()
for i in range(int(input())):
    s = input()
    if s[0] == '1':
        heap.insert(int(s[2:]))
    else:
        print(heap.get_min())
        heap.delete_min()
```

Huffman 树

OJ22161:哈夫曼编码树

http://cs101.openjudge.cn/practice/22161/

```
import heapq


class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
        self.height = None

    def huff_value(self, h):
        if not self.left and not self.right:
            return h * self.val
        left_value, right_value = 0, 0
        if self.left:
            left_value = self.left.huff_value(h + 1)
        if self.right:
            right_value = self.right.huff_value(h + 1)
        return left_value + right_value

    def __lt__(self, other):
        return self.val < other.val

    def __gt__(self, other):
        return self.val > other.val


n = int(input())
nodes = []
for i in list(map(int, input().split())):
    heapq.heappush(nodes, Node(i))
while len(nodes) > 1:
    left, right = heapq.heappop(nodes), heapq.heappop(nodes)
    heapq.heappush(nodes, Node(left.val + right.val, left, right))
print(nodes[0].huff_value(0))
```

```
import heapq
class Node:
    def __init__(self, weight, char=None):
```

```
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None

    def __lt__(self, other):
        if self.weight == other.weight:
            return self.char < other.char
        return self.weight < other.weight
def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        #merged = Node(left.weight + right.weight) #note: 
合并后，char 字段默认值是空
        merged = Node(left.weight + right.weight, 
min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
def encode_huffman_tree(root):
    codes = {}
    def traverse(node, code):
        #if node.char:
        if node.left is None and node.right is None:
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')
    traverse(root, '')
    return codes
def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded
def huffman_decoding(root, encoded_string):
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.left is None and node.right is None:
            decoded += node.char
            node = root
    return decoded

# 读取输入
n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)
# 构建哈夫曼编码树
huffman_tree = build_huffman_tree(characters)
# 编码和解码
codes = encode_huffman_tree(huffman_tree)
strings = []
while True:
    try:
        line = input()
        strings.append(line)
    except EOFError:
        break
results = []
for string in strings:
    if string[0] in ('0','1'):
        results.append(huffman_decoding(huffman_tree, 
string))
    else:
        results.append(huffman_encoding(codes, string))
for result in results:
    print(result)
```

AVL 树

OJ27625:AVL 树至少有几个结点

http://cs101.openjudge.cn/2024sp_routine/27625/

图

拓扑排序及 Kahn 算法

拓扑排序是对有向无环图（DAG，Directed Acyclic Graph）的顶点进行排序的一种方法，使得对于图中的每条有向边 UV（从顶点 U 指向顶点 V），U 在排序中都出现在 V 之前。拓扑排序不是唯一的，一个有向无环图可能有多个有效的拓扑排序。

拓扑排序常用的算法包括基于 DFS（深度优先搜索）的方法和基于 BFS（广度优先搜索，也称为 Kahn 算法）的方法。

作用：检测是否有环

代码示例

```
from collections import deque, defaultdict


def topological_sort(vertices, edges):
    # 计算所有顶点的入度
    in_degree = {v: 0 for v in vertices}
    graph = defaultdict(list)

    # u->v
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1   # v 的入度+1

    # 将所有入度为 0 的顶点加入队列
    queue = deque([v for v in vertices if in_degree[v] == 0])
    sorted_order = []

    while queue:
        u = queue.popleft()
        sorted_order.append(u)
```

```
        # 对于每一个相邻顶点，减少其入度
        for v in graph[u]:
            in_degree[v] -= 1
            # 如果入度减为 0，则加入队列
            if in_degree[v] == 0:
                queue.append(v)

    if len(sorted_order) != len(vertices):
        return None  # 存在环，无法进行拓扑排序
    return sorted_order


# 示例使用
vertices = ['A', 'B', 'C', 'D', 'E', 'F']
edges = [('A', 'D'), ('F', 'B'), ('B', 'D'), ('F', 'A'), ('D', 'C')]
result = topological_sort(vertices, edges)
if result:
    print("拓扑排序结果:", result)
else:
    print("图中有环，无法进行拓扑排序")
```

例题

OJ04084:拓扑排序

http://cs101.openjudge.cn/2024sp_routine/04084/

拓扑排序，但是要求“同等条件下，编号小的顶点在前”，不得不把普通队列转换成一个优先队列了。

```
from collections import deque, defaultdict
import heapq

def topo_sort(g, nv):
    ans = []
    deg = {v: 0 for v in range(1, nv+1)}
    child = {v: [] for v in range(1, nv+1)}
    for u, v in g:
        # u->v
        if v not in deg:
            deg[v] = 1
        else:
            deg[v] += 1
        if u not in child:
            child[u] = [v]
        else:
            child[u].append(v)
    q = [v for v in deg.keys() if deg[v] == 0]
    heapq.heapify(q)
    while q:
        now = heapq.heappop(q)
        ans.append(now)
        for i in child[now]:
            deg[i] -= 1
            if deg[i] == 0:
                heapq.heappush(q, i)

    return ans
v, a = map(int, input().split())
g = []
for _ in range(a):
    x, y = map(int, input().split())
    g.append([x, y])
for i in topo_sort(g, v):
    print('v' + str(i), end=' ')
```

OJ01094:Sorting It All Out

http://cs101.openjudge.cn/dsapre/01094/

此题要求每给出一条边就进行一次拓扑排序。首先判断给出的图有没有环，若拓扑排序后有顶点入度不为 0，则有环。然后判断拓扑排序是否唯一，若同一时间队列长度大于 1，则给出的条件不足以唯一确定拓扑排序。

```
from collections import deque


def topo_sort(g, nv):
    ans = []
    deg = {chr(i): 0 for i in range(65, 65 + nv)}
    child = {chr(i): [] for i in range(65, 65 + nv)}
    for u, v in g:
        # u->v
        if v in child[u]:
            continue
        deg[v] += 1
        child[u].append(v)
    q = deque([v for v in deg.keys() if deg[v] == 0])
    not_determined = False
    while q:
        not_determined = len(q) >= 2 or not_determined
        now = q.popleft()
        ans.append(now)
        for i in child[now]:
            deg[i] -= 1
            if deg[i] == 0:
                q.append(i)
    loop = False
    for k, v in deg.items():
        if v != 0:
            loop = True
            break
    return ans, loop, not_determined


while True:
    v, a = map(int, input().split())
    if v == 0 and a == 0:
        break
    g = []
    sorted_seq = None
    end = False
    for _ in range(a):
        x, y = map(str, input().split('<'))
        if end:
            continue
        g.append([x, y])
        sorted_seq, loop, not_determined = topo_sort(g, v)
        if loop:
            print(f'Inconsistency found after {_ + 1}
relations.')
            end = True
        elif not not_determined:
            print(f'Sorted sequence determined after {_ + 1}
```

```
relations: ', end='')
                for qq in sorted_seq:
                    print(qq, end='')
                print('.')
                end = True
        if end:
            continue
        print('Sorted sequence cannot be determined.')
```

OJ09202:舰队、海域出击！

http://cs101.openjudge.cn/2024sp_routine/09202/

检测有向图有没有环，拓扑排序，也就是 Kahn 算法。

```
from collections import deque
def topo(g, deg, v):
    q = deque([x for x in deg.keys() if deg[x] == 0])
    cnt = 0
    while q:
        now = q.popleft()
        cnt += 1
        for next in g[now]:
            deg[next] -= 1
            if deg[next] == 0:
                q.append(next)
    if cnt == v:
        print('No')
    else:
        print('Yes')
for _ in range(int(input())):
    v, m = map(int, input().split())
    g = {a: [] for a in range(1, v + 1)}
    deg = {a: 0 for a in range(1, v + 1)}
    for _ in range(m):
        x, y = map(int, input().split())
        deg[y] += 1
        g[x].append(y)
    topo(g, deg, v)
```

Dijkstra 算法

代码示例

```
import heapq


def dijkstra(graph, start):
    # 初始化距离字典，所有顶点距离为无穷大，起始点距离为 0
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    # 优先队列，用于存储每个顶点及其对应的距离，并按距离自动排序
    priority_queue = [(0, start)]

    while priority_queue:
        # 获取当前距离最小的顶点
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # 遍历当前顶点的邻接点
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            # 如果计算的距离小于已知距离，更新距离
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances


# 示例图
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

# 测试算法
start_vertex = 'A'
distances = dijkstra(graph, start_vertex)
print(f"Distances from {start_vertex}: {distances}")
```

例题

OJ05443:兔子与樱花

http://cs101.openjudge.cn/dsapre/05443/

模板题目，额外的一点是需要记录路径

```
import heapq

def dijkstra(adjacency, start):
    # 初始化，将其余所有顶点到起始点的距离都设为 inf（无穷大）
    distances = {vertex: float('inf') for vertex in adjacency}
    # 初始化，所有点的前一步都是 None
    previous = {vertex: None for vertex in adjacency}
    # 起点到自身的距离为 0
    distances[start] = 0
    # 优先队列
    pq = [(0, start)]

    while pq:
        # 取出优先队列中，目前距离最小的
        current_distance, current_vertex = heapq.heappop(pq)
        # 剪枝，如果优先队列里保存的距离大于目前更新后的距离，则可以跳过
        if current_distance > distances[current_vertex]:
            continue

        # 对当前节点的所有邻居，如果距离更优，将他们放入优先队列中
        for neighbor, weight in adjacency[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
```

```
                distances[neighbor] = distance
                # 这一步用来记录每个节点的前一步
                previous[neighbor] = current_vertex
                heapq.heappush(pq, (distance, neighbor))

    return distances, previous

def shortest_path_to(adjacency, start, end):
    # 逐步访问每个节点上一步
    distances, previous = dijkstra(adjacency, start)
    path = []
    current = end
    while previous[current] is not None:
        path.insert(0, current)
        current = previous[current]
    path.insert(0, start)
    return path, distances[end]

# Read the input data
P = int(input())
places = {input().strip() for _ in range(P)}

Q = int(input())
graph = {place: {} for place in places}
for _ in range(Q):
    src, dest, dist = input().split()
    dist = int(dist)
    graph[src][dest] = dist
    graph[dest][src] = dist    # Assuming the graph is
bidirectional

R = int(input())
requests = [input().split() for _ in range(R)]

# Process each request
for start, end in requests:
    if start == end:
        print(start)
        continue

    path, total_dist = shortest_path_to(graph, start, end)
    output = ""
    for i in range(len(path) - 1):
        output +=
f"{path[i]}->({graph[path[i]][path[i+1]]})->"
    output += f"{end}"
    print(output)
```

OJ07735:道路

http://cs101.openjudge.cn/practice/07735/

dijkstra，但是有点区别，加入优先队列的条件不是距离更短，而是金币够用，但是优先队列的比较仍然是用距离比的

```
import heapq

k, n, r = int(input()), int(input()), int(input())


def dij(g, s, e):
    dis = {v: float('inf') for v in range(1, n + 1)}
    dis[s] = 0
    q = [(0, s, 0)]
    heapq.heapify(q)
    while q:
        d, now, fee = heapq.heappop(q)
        if now == n:
            return d
        for neighbor, distance, c in g[now]:
            if fee + c <= k:
                dis[neighbor] = distance + d
                heapq.heappush(q, (distance + d, neighbor,
fee + c))
    return -1


g = {v: [] for v in range(1, n + 1)}
for _ in range(r):
    s, e, m, j = map(int, input().split())
    g[s].append((e, m, j))
p = dij(g, 1, n)
print(p)
```

OJ02502:Subway

乍一看有点难，但实际上就是模板题。暴力把所有点之间全连上一条路径，然后把通地铁的车站的路径的代价设置小点（实际上在我的实现中，是多连了一条代价更小的路径，但是 dijkstra 算法可以容忍这一点）

```
import heapq


def hash(x, y):
    return str(x) + ' ' + str(y)


x0, y0, x1, y1 = map(int, input().split())
g = {}
all_v = [(x0, y0), (x1, y1)]
subways = []


def d(a, b, c, d):
    return ((a - c) ** 2 + (b - d) ** 2) ** 0.5


while True:
    try:
        lst = list(map(int, input().split()))[:-2]
    except EOFError:
        break
    v = []
    for i in range(0, len(lst), 2):
        x, y = lst[i], lst[i + 1]
        v.append([x, y])
        all_v.append([x, y])
    subways.append(v)
for i in range(len(all_v) - 1):
    for j in range(i + 1, len(all_v)):
        sx, sy = all_v[i]
        ex, ey = all_v[j]
        dd = d(sx, sy, ex, ey) / (10 / 3.6) / 60
        if hash(sx, sy) in g.keys():
```

```
                    g[hash(sx, sy)].append([dd, ex, ey])
                else:
                    g[hash(sx, sy)] = [[dd, ex, ey]]
                if hash(ex, ey) in g.keys():
                    g[hash(ex, ey)].append([dd, sx, sy])
                else:
                    g[hash(ex, ey)] = [[dd, sx, sy]]
for j in subways:
        for i in range(len(j) - 1):
                sx, sy = j[i]
                ex, ey = j[i + 1]
                dd = d(sx, sy, ex, ey) / (40 / 3.6) / 60
                g[hash(sx, sy)].append([dd, ex, ey])
                g[hash(ex, ey)].append([dd, sx, sy])


def dij():
        dis = {hash(x, y): float('inf') for x, y in all_v}
        dis[hash(x0, y0)] = 0
        q = [(0, x0, y0)]
        heapq.heapify(q)
        while q:
                distance, nowx, nowy = heapq.heappop(q)
                if distance > dis[hash(nowx, nowy)]:
                        continue
                for newd, tx, ty in g[hash(nowx, nowy)]:
                        if newd + distance < dis[hash(tx, ty)]:
                                dis[hash(tx, ty)] = newd + distance
                                heapq.heappush(q, (newd + distance, tx,
ty))
        return dis[hash(x1, y1)]


print(round(dij()))
```

最小生成树（Prim 算法、Kruskal 算法）

两种算法，Prim，Kruskal

求解最小生成树（Minimum Spanning Tree, MST）的问题在图论中非常重要，尤其是在设计和优化网络、路由算法以及集群分析等领域。最小生成树是一个无向图的生成树，它包括图中所有的顶点，并且选取的边的总权重最小。以下是几种常用的求解最小生成树的算法：

1. Kruskal 算法

2.
- 基本思想：Kruskal 算法是一种基于边的贪心算法。它的核心思想是按照边的权重从小到大的顺序选择边，但在选择的同时必须保证不形成环。
-
- 步骤：
-
- 将所有边按权重从小到大排序。
-
- 初始化一个空的最小生成树。
-
- 依次考虑排序后的每条边，如果加入这条边不会与已选的边形成环（可以用并查集来检测），则将其加入到最小生成树中。
-
- 重复上一步，直到最小生成树中含有 V−1 条边，其中 V 是图中顶点的数量。
- 复杂度：如果使用优化的并查集，复杂度可以达到 O(ElogE) 或 O(ElogV)。
- Prim 算法
- 基本思想：Prim 算法是基于顶点的贪心算法。它从任意顶点开始，逐渐增加边和顶点，直到包括所有顶点。
- 步骤：
- 选择任意一个顶点作为起始点。
-
- 使用一个优先队列来维护可选择的边（根据边的权重）。
-

从优先队列中选择一条权重最小的边，如果这条边连接的顶点还未被加入最小生成树，则将此顶点及边加入树中。

6.
7.

更新优先队列，重复上述过程，直到所有顶点都被加入。

8.
•

复杂度：使用优先队列（如二叉堆），复杂度为 O(ElogV*)。

•

OJ01258:Agri-Net

http://cs101.openjudge.cn/practice/01258/

```
import heapq

while True:
    try:
        n = int(input())
    except EOFError:
        break
    visited = {0}
    m_cost = 0
    g = [list(map(int, input().split())) for _ in range(n)]
    edges = [(cost, 0, to) for to, cost in enumerate(g[0])]
    heapq.heapify(edges)
    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            m_cost += cost
            for neighbor, next_cost in enumerate(g[to]):
                if neighbor not in visited:
                    heapq.heappush(edges, (next_cost, to,
neighbor))
    print(m_cost)
```

OJ05442:兔子与星空

http://cs101.openjudge.cn/practice/05442/

prim

```
import heapq

def prim(graph, start):
    mst = []
    used = set([start])
    edges = [
        (cost, start, to)
        for to, cost in graph[start].items()
    ]
    heapq.heapify(edges)

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in used:
            used.add(to)
            mst.append((frm, to, cost))
            for to_next, cost2 in graph[to].items():
                if to_next not in used:
                    heapq.heappush(edges, (cost2, to,
to_next))

    return mst

def solve():
    n = int(input())
    graph = {chr(i+65): {} for i in range(n)}
    for i in range(n-1):
        data = input().split()
        star = data[0]
        m = int(data[1])
        for j in range(m):
            to_star = data[2+j*2]
            cost = int(data[3+j*2])
            graph[star][to_star] = cost
            graph[to_star][star] = cost
    mst = prim(graph, 'A')
    print(sum(x[2] for x in mst))

solve()
```

Kurskal

```
class DisjSet:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

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

    def union(self, x, y):
        xset, yset = self.find(x), self.find(y)
        if self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
        else:
            self.parent[xset] = yset
            if self.rank[xset] == self.rank[yset]:
                self.rank[yset] += 1


def kruskal(n, edges):
    dset = DisjSet(n)
    edges.sort(key=lambda x: x[2])
    sol = 0
    for u, v, w in edges:
        u, v = ord(u) - 65, ord(v) - 65
        if dset.find(u) != dset.find(v):
            dset.union(u, v)
            sol += w
    if len(set(dset.find(i) for i in range(n))) > 1:
        return -1
    return sol


n = int(input())
edges = []
```

```
for _ in range(n - 1):
    arr = input().split()
    root, m = arr[0], int(arr[1])
    for i in range(m):
        edges.append((root, arr[2 + 2 * i], int(arr[3 + 2 * i])))
print(kruskal(n, edges))
```

OJ01798:Truck History

http://cs101.openjudge.cn/2024sp_routine/01798/

```
import heapq

def truck_history():
    while True:
        n = int(input())
        if n == 0:
            break

        trucks = [input() for _ in range(n)]
        trucks.sort()

        graph = [[0]*n for _ in range(n)]
        for i in range(n):
            for j in range(i+1, n):
                graph[i][j] = graph[j][i] = sum(a!=b for a, b in zip(trucks[i], trucks[j]))

        visited = [False]*n
        min_edge = [float('inf')]*n
        min_edge[0] = 0
        total_distance = 0

        min_heap = [(0, 0)]
        while min_heap:
            d, v = heapq.heappop(min_heap)
            if visited[v]:
                continue
            visited[v] = True
            total_distance += d
            for u in range(n):
                if not visited[u] and graph[v][u] < min_edge[u]:
                    min_edge[u] = graph[v][u]
                    heapq.heappush(min_heap, (graph[v][u], u))

        print(f"The highest possible quality is 1/{total_distance}.")

truck_history()
```

强连通分量与 Kosaraju 算法

是一个用于寻找有向图中所有强连通分量（Strongly Connected Components，SCC）的高效算法。一个强连通分量是最大的子图，其中任何两个顶点都是双向可达的。这个算法的时间复杂度为 �(�+�)O(V+E)，其中 �V 是顶点数，�E 是边数。

算法步骤

Kosaraju's 算法包括以下几个步骤：

第一步：对原图进行深度优先搜索（DFS）

- 从任意未访问过的顶点开始，对原始图进行一次深度优先搜索。

-

- 每次完成一个顶点的 DFS 遍历后，将该顶点推入一个栈中。这样做的目的是按照完成时间的顺序保存顶点，确保在进行第二次 DFS 时，我们从一个 SCC 的源点（或接近源点）开始。

第二步：获取转置图

- 将原图的所有边反向，得到转置图。转置图的意思是如果原图中有一条从 �u 到 �v 的有向边，那么在转置图中就有一条从 �v 到 �u 的边。

第三步：对转置图进行 DFS

- 依次从栈中弹出顶点，如果该顶点未被访问过，就以该顶点为起点，对转置图进行 DFS。

-

- 每次从一个顶点开始的 DFS 可以访问到的所有顶点，都属于同一个强连通分量。

-

- 记录下每次 DFS 访问到的所有顶点，它们组成了一个强连通分量。

算法正确性的关键

第一次 DFS 帮助我们了解每个顶点的完成时间。在转置图中，我们按照原图的完成时间逆序（即最晚完成的顶点最先处理）来处理每个顶点，这样可以保证每当开始一个新的 DFS 时，我们都是从另一个 SCC 的源点开始的。

在转置图中，如果从顶点 �u 可以到达顶点 �v，那么在原图中，从 �v 也能到达 �u。因此，第二次 DFS 实际上是在追踪原图中每个 SCC 的边界。

代码示例

以下是使用 Python 实现 Kosaraju's 算法的基本框架：

```
def dfs(graph, v, visited, stack):
    visited[v] = True
    for neighbour in graph[v]:
        if not visited[neighbour]:
            dfs(graph, neighbour, visited, stack)
    stack.append(v)

def dfs_util(graph, v, visited):
    visited[v] = True
    print(v, end=' ')
    for neighbour in graph[v]:
        if not visited[neighbour]:
            dfs_util(graph, neighbour, visited)

def kosaraju(graph, vertices):
    stack = []
    visited = [False] * vertices

    # Step 1: Fill vertices in stack according to their finishing times
    for i in range(vertices):
        if not visited[i]:
            dfs(graph, i, visited, stack)

    # Step 2: Create a reversed graph
    rev_graph = [[] for _ in range(vertices)]
    for v in range(vertices):
        for neighbour in graph[v]:
            rev_graph[neighbour].append(v)

    # Step 3: Process all vertices in order defined by Stack
    visited = [False] * vertices
    while stack:
        v = stack.pop()
        if not visited[v]:
            print("SCC: ", end='')
            dfs_util(rev_graph, v, visited)
            print()

# Example usage
graph = [
    [1],        # 0 -> 1
    [2],        # 1 -> 2
    [0, 3],     # 2 -> 0, 3
    [4],        # 3 -> 4
    [5],        # 4 -> 5
    [3]         # 5 -> 3
]
kosaraju(graph, 6)
```

这段代码首先实现了一个 DFS 来填充栈，然后创建一个转置图，最后再根据栈中顶点的顺序对转置图执行 DFS 来找到所有的强连通分量。