数据结构与算法——总结

Part 1：线性表

线性表（List）的定义：零个或多个数据元素的**有限**序列。

线性表的数据集合为{a_1, a_2……a_n}，该序列有唯一的头元素和尾元素，除了头元素外，每个元素都有唯一的前驱元素，除了尾元素外，每个元素都有唯一的后继元素。

线性表中的元素属于相同的数据类型，即每个元素所占的空间相同。

框架：

一、顺序表

python中的顺序表就是列表，元素在内存中连续存放，每个元素都有唯一序号(下标），且根据序号访问（包括读取和修改）元素的时间复杂度是O(1)的（**随机访问**）。

```
class SequentialList:
    def __init__(self, n):
        self.data = list(range(n))

    def is_empty(self):
        return len(self.data) == 0

    def length(self):
        return len(self.data)

    def append(self, item):
        self.data.append(item)

    def insert(self, index, item):
        self.data.insert(index, item)

    def delete(self, index):
        if 0 <= index < len(self.data):
            del self.data[index]
        else:
            return IndexError('Index out of range')

    def get(self, index):
        if 0 <= index < len(self.data):
            return self.data[index]
        else:
            return IndexError('Index out of range')

    def set(self, index, target):
        if 0 <= index < len(self.data):
            self.data[index] = target
        else:
            return IndexError('Index out of range')
```

```
    def display(self):
        print(self.data)

lst = SequentialList(n)
```

关于线性表的时间复杂度：

生成、求表中元素个数、表尾添加/删除元素、返回/修改对应下标元素，均为O(1)；

而查找、删除、插入元素，均为O(n)。

线性表的优缺点：

优点：1、无须为表中元素之间的逻辑关系而增加额外的存储空间；

2、可以快速的存取表中任一位置的元素。

缺点：1、插入和删除操作需要移动大量元素；

2、当线性表长度较大时，难以确定存储空间的容量；

3、造成存储空间的“碎片”。

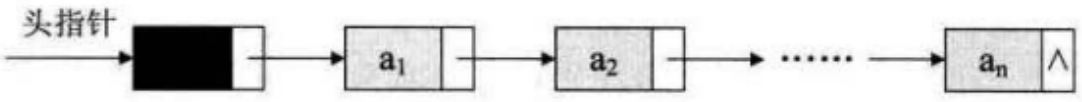
二、链表

1、单链表

在链式结构中，除了要存储数据元素的信息外，还要存储它的后继元素的存储地址。

因此，为了表示**每个数据元素**a_i**与其直接后继元素**a_i+1**之间的逻辑关系，对数据**a_i**来说，除了存储其本身的信息之外，还需要存储一个指示其直接后继的信息（即直接后继的存储位置）。我们把存储数据元素信息的域称为数据域，把存储直接后继位置的域称为指针域。指针域中存储的信息称做指针或链。这两部分信息组成数据元素**a_i**的存储映像，称为结点（Node）。**

我们把链表中第一个结点的存储位置叫做头指针。有时为了方便对链表进行操作，会在单链表的第一个结点前附设一个节点，称为头结点，此时头指针指向的结点就是头结点。



空链表，头结点的直接后继为空。


```
class LinkList:
    class Node:
        def __init__(self, data, next=None):
```

```
            self.data, self.next = data, next

    def __init__(self):
        self.head = self.tail = LinkList.Node(None, None)
        self.size = 0

    def print(self):
        ptr = self.head
        while ptr is not None:
            print(ptr.data, end=',')
            ptr = ptr.next

    def insert(self, p, data):
        nd = LinkList.Node(data, None)
        if self.tail is p:
            self.tail = nd
        nd.next = p.next
        p.next = nd
        self.size += 1

    def delete(self, p):
        if self.tail is p.next:
            self.tail = p
        p.next = p.next.next
        self.size -= 1

    def popFront(self):
        if self.head is None:
            raise Exception("Popping front for Empty link list.")
        else:
            self.head = self.head.next
            self.size -= 1
            if self.size == 0:
                self.head = self.tail = None

    def pushFront(self, data):
        nd = LinkList.Node(data, self.head)
        self.head = nd
        self.size += 1
        if self.size == 1:
            self.tail = nd

    def pushBack(self, data):
        if self.size == 0:
            self.pushFront(data)
        else:
            self.insert(self.tail, data)

    def clear(self):
        self.head = self.tail = None
        self.size = 0

    def __iter__(self):
        self.ptr = self.head
        return self
```

```
    def __next__(self):
        if self.ptr is None:
            raise StopIteration()
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data
```

2、双链表

双向链表（Double Link List）是在单链表的每个结点中，再设置一个指向其前驱结点的指针域。所以在双向链表中的结点都有两个指针域，一个指向直接后继，另一个指向直接前驱。

```
class DoubleLinkList:
    class Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev, next

    class Iterator:
        def __init__(self, p):
            self.ptr = p

        def get(self):
            return self.ptr.data

        def set(self, data):
            self.ptr.data = data

        def __iter__(self):
            self.ptr = self.ptr.next
            if self.ptr is None:
                return None
            else:
                return DoubleLinkList.Iterator(self.ptr)

        def prev(self):
            self.ptr = self.ptr.prev
            return DoubleLinkList.Iterator(self.ptr)

    def __init__(self):
        self.head = self.tail = DoubleLinkList.Node(None, None, None)
        self.size = 0

    def _insert(self, p, data):
        nd = DoubleLinkList.Node(data, p, p.next)
        if self.tail is p:
            self.tail = nd
        if p.next:
            p.next.prev = nd
        p.next = nd
        self.size += 1
```

```
    def _delete(self, p):
        if self.size == 0 or p is self.head:
            return Exception("Illegal deleting.")
        else:
            p.prev.next = p.next
            if p.next:
                p.next.prev = p.prev
            if self.tail is p:
                self.tail = p.prev
            self.size -= 1

    def clear(self):
        self.tail = self.head
        self.head.next = self.head.prev = None
        self.size = 0

    def begin(self):
        return DoubleLinkList.Iterator(self.head.next)

    def end(self):
        return None

    def insert(self, i, data):
        self._insert(i.ptr, data)

    def delete(self, i):
        self._delete(i.ptr)

    def pushFront(self, data):
        self._insert(self.head, data)

    def pushBack(self, data):
        self._insert(self.tail, data)

    def popFront(self):
        self._delete(self.head.next)

    def popBack(self):
        self._delete(self.tail)

    def __iter__(self):
        self.ptr = self.head.next
        return self

    def __next__(self):
        if self.ptr is None:
            raise StopIteration()
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data

    def find(self, val):
        ptr = self.head.next
        while ptr is not None:
```

```
            if ptr.data == val:
                return DoubleLinkList.Iterator(ptr)
            ptr = ptr.next
        return self.end()

    def printList(self):
        ptr = self.head.next
        while ptr is not None:
            print(ptr.data, end=',')
            ptr = ptr.next
```

3、循环链表

将单链表中终端节点的指针端由空指针改为指向头结点，就使整个单链表形成一个环，这种头尾相接的单链表称为单循环链表，简称循环链表。

然而这样会导致访问最后一个结点时需要O(n)的时间，所以我们可以写出**仅设尾指针的循环链表。**

```
class CircleLinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self):
        self.tail = None
        self.size = 0

    def is_empty(self):
        return self.size == 0

    def pushFront(self, data):
        nd = CircleLinkList.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
            self.tail.next = nd
        self.size += 1

    def pushBack(self, data):
        self.pushFront(data)
        self.tail = self.tail.next

    def popFront(self):
        if self.is_empty():
            return None
        else:
            nd = self.tail.next
            self.size -= 1
            if self.size == 0:
```

```
                self.tail = None
            else:
                self.tail.next = nd.next
        return nd.data

    def popBack(self):
        if self.is_empty():
            return None
        else:
            nd = self.tail.next
            while nd.next != self.tail:
                nd = nd.next
            data = self.tail.data
            nd.next = self.tail.next
            self.tail = nd
            return data

    def printList(self):
        if self.is_empty():
            print('Empty!')
        else:
            ptr = self.tail.next
            while True:
                print(ptr.data, end=',')
                if ptr == self.tail:
                    break
                ptr = ptr.next
            print()
```

三、一些题目

1、约瑟夫问题

http://cs101.openjudge.cn/practice/02746/

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircleLinkList:
    def __init__(self):
        self.tail = None
        self.size = 0

    def append(self, data):
        nd = Node(data)
        if self.size == 0:
            self.tail = nd
            nd.next = self.tail
```

```
        else:
            nd.next = self.tail.next
            self.tail.next = nd
            self.tail = self.tail.next
        self.size += 1

    def remove(self, node):
        current = self.tail.next
        prev = self.tail
        while current != self.tail:
            if current.data == node.data:
                prev.next = current.next
                self.size -= 1
                break
            prev = current
            current = current.next
        if current == self.tail and current.data == node.data:
            prev.next = current.next
            self.tail = prev
            self.size -= 1

    def get(self, m):
        current = self.tail.next
        while self.size > 1:
            for _ in range(m-1):
                current = current.next
            nd = current.next
            self.remove(current)
            current = nd
        return self.tail.data

while True:
    n, m = map(int, input().split())
    if n == m == 0:
        break
    monkey = CircleLinkList()
    for i in range(1, n+1):
        monkey.append(i)
    win = monkey.get(m)
    print(win)
```

2、单向链表

https://www.luogu.com.cn/problem/B3631

这道题如果按照最初给的写法写，会超时，是因为每次无论是插入、查询还是删除都需要先找到相应的结点。所以可以使用字典，能够节省大量时间。

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
class LinkList:
    def __init__(self):
        self.head = Node(1)
        self.map = {1: self.head}

    def insert(self, x, y):
        nx = self.map.get(x)
        ny = Node(y)
        ny.next = nx.next
        nx.next = ny
        self.map[y] = ny

    def check(self, x):
        nx = self.map.get(x)
        if nx.next is None:
            print(0)
            return
        print(nx.next.data)

    def delete(self, x):
        nx = self.map.get(x)
        tmp = nx.next
        nx.next = tmp.next
        del self.map[tmp.data]

lst = LinkList()
for _ in range(int(input())):
    nums = list(map(int, input().split()))
    if nums[0] == 1:
        x, y = nums[1:]
        lst.insert(x, y)
    elif nums[0] == 2:
        x = nums[1]
        lst.check(x)
    else:
        x = nums[1]
        lst.delete(x)
```

3、删除链表元素

http://dsbpython.openjudge.cn/dspythonbook/P0020/

可以练习循环链表的基本知识（只是缩进有点……）。

```
class Node:
    def __init__(self, data, next=None):
        self.data, self.next = data, next
class LinkList:  #循环链表
    def __init__(self):
        self.tail = None
        self.size = 0
    def isEmpty(self):
        return self.size == 0
```

```
    def pushFront(self,data):
        nd = Node(data)
        if self.tail == None:
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
            self.tail.next = nd
        self.size += 1
    def pushBack(self,data):
        self.pushFront(data)
        self.tail = self.tail.next
    def popFront(self):
        if self.size == 0:
            return None
        else:
            nd = self.tail.next
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.tail.next = nd.next
        return nd.data
    def printList(self):
        if self.size > 0:
            ptr = self.tail.next
            while True:
                print(ptr.data,end = " ")
                if ptr == self.tail:
                    break
                ptr = ptr.next
            print("")

    def remove(self,data):
        if self.size == 0:
            return None

        if self.size == 1 and self.tail.data == data:
            self.tail = None
            self.size -= 1
            return True

        current = self.tail.next
        prev = self.tail
        while current != self.tail:
            if current.data == data:
                prev.next = current.next
                self.size -= 1
                return True
            prev = current
            current = current.next

        if current.data == data:
            prev.next = current.next
            self.tail = prev
            self.size -= 1
```

```
            return True
        return False

t = int(input())
for i in range(t):
    lst = list(map(int,input().split()))
    lkList = LinkList()
    for x in lst:
        lkList.pushBack(x)
    lst = list(map(int,input().split()))
    for a in lst:
        result = lkList.remove(a)
        if result == True:
            lkList.printList()
        elif result == False:
            print("NOT FOUND")
        else:
            print("EMPTY")
    print("----------------")
```

4、双端队列

http://cs101.openjudge.cn/practice/05902/

这道题如果用deque，可以很快的做出来，但是实际上题目想考察的，是对于链表的书写和运用能力。

```
class DoubleQueue:
    class Node:
        def __init__(self, val=None):
            self.val = val
            self.next = None

    def __init__(self):
        self.tail = None
        self.size = 0

    def is_empty(self):
        return self.size == 0

    def push(self, data):
        nd = DoubleQueue.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
            self.tail.next = nd
        self.size += 1
        self.tail = self.tail.next

    def popFront(self):
        if self.is_empty():
            return
```

```
        else:
            nd = self.tail.next
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.tail.next = nd.next
            return

    def popBack(self):
        if self.is_empty():
            return
        else:
            nd = self.tail.next
            while nd.next != self.tail:
                nd = nd.next
            nd.next = self.tail.next
            self.tail = nd
            self.size -= 1
        return

    def Print(self):
        if self.is_empty():
            print('NULL')
            return
        else:
            nd = self.tail.next
            while True:
                print(nd.val, end=' ')
                if nd == self.tail:
                    break
                nd = nd.next
            print()
            return

for _ in range(int(input())):
    deque = DoubleQueue()
    for _ in range(int(input())):
        t, c = map(int, input().split())
        if t == 1:
            deque.push(c)
        else:
            if c:
                deque.popBack()
            else:
                deque.popFront()
    deque.Print()
```

四、一些基本的写法

1、反转链表

改变链表的顺序，使其反向，可以通过迭代或递归实现。

```
class Node:
    def __init__(self):
        self.val = 0
        self.next = None

# 迭代
def ReverseList(head: Node) -> Node:
    prev = None
    current = head
    while current is not None:
        tmp = current.next
        current.next = prev
        prev = current
        current = tmp
    return prev

# 递归
def reverse(head: Node) -> Node:
    if head.next is None:
        return head
    new_head = reverse(head.next)
    head.next = None
    new_head.next = head
    return new_head
```

2、检测环

判断链表是否含有环，快慢指针（Floyd的检测循环算法）是解决这个问题的一个著名方法。

```
def has_loop(head: Node) -> bool:
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

3、合并两个排序链表

给定两个已排序的链表，要求合并它们并保持排序。这个问题可以用迭代或递归解决。

P.S： 这已经涉及到部分归并排序的内容了。

```
def merge(p: Node, q: Node):
    if p is None or q is None:
        return p or q
    if p.val < q.val:
        p.next = merge(p.next, q)
        return p
    else:
        q.next = merge(p, q.next)
        return q
```

4、删除倒数第n个元素

设置两个指针，一个先走n步，之后一起走，先走的到达末尾的时候，另一个就在对应位置了。

```
def remove(head: Node, n: int) -> Node:
    dummy = Node()
    dummy.next = head
    slow, fast = dummy, dummy
    for _ in range(n+1):
        fast = fast.next
    while fast:
        slow = slow.next
        fast = fast.next
    slow.next = slow.next.next
    return dummy.next
```

5、查找中间元素

感觉和上一条差不多……如果设置了self.size的话就更好处理了，总之可以使用快慢指针。

```
def middle(head: Node) -> Node:
    slow, fast = head, head
    while fast.next is not None:
        slow = slow.next
        fast = fast.next.next
    return slow
```

6、判断是否回文链表

其实就是先找中间点，然后反转，最后进行判断即可。

```
def check(head: Node) -> bool:
    slow, fast = head, head
    while fast is not None and fast.next is not None:
        fast = fast.next.next
        slow = slow.next
    prev = None
    while slow:
        tmp = slow.next
        slow.next = prev
        prev = slow
        slow = tmp
    while prev is not None:
        if head.val != prev.val:
            return False
        head = head.next
        prev = prev.next
    return True
```

7、移除链表元素

删除链表中所有等于给定的结点，要求**一次遍历解决**。

```
def remove(head: Node, data: int) -> Node:
    dummy = Node()
    dummy.next = head
    current = dummy
    while current.next is not None:
        if current.next.val == data:
            current.next = current.next.next
        else:
            current = current.next
    return dummy.next
```

8、分割链表

根据给定值x分割链表，使得所有小于x的节点都位于大于或等于x的节点之前，保持原有元素的相对顺序。

```
def partition(head: Node, x: int) -> Node:
    before_node = Node()
    before = before_node
    after_node = Node()
    after = after_node
    while head is not None:
        if head.val < x:
```

```
            before.next = head
            before = before.next
        else:
            after.next = head
            after = after.next
        head = head.next
    after.next = None
    before.next = after_node.next
    return before_node.next
```

9、旋转链表

给定一个链表，每个元素向右移动k个位置。可以先讲单链表连通，形成环，之后再断开即可。

```
def rotate(head: Node, k: int) -> Node:
    if head.next is None or k == 0:
        return head
    length = 1
    tail = head
    while tail.next is not None:
        tail = tail.next
        length += 1
    tail.next = head
    new_tail = head
    for _ in range(length-k % length-1):
        new_tail = new_tail.next
    new_head = new_tail.next
    new_tail.next = None
    return new_head
```

Part 2：排序

一、冒泡排序（Bouble Sort）

冒泡排序是最简单的排序算法，它通过不断交换相邻元素以实现正确的排序结果。

时间复杂度：O(n^2)；空间复杂度：O(1)。

冒泡排序是一种原地排序算法，无需额外空间，是稳定的。

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

二、选择排序（Selection Sort）

选择排序是一种简单且高效的排序算法，其工作原理是从列表的未排序部分反复选择最小(或最大)元素，并将其移动到列表的已排序部分。

时间复杂度：O(n^2)；空间复杂度：O(1)。

选择排序同样是一种原地排序算法，无需额外空间，但它在小的数据集下相对高效，在极端情况下会具有较大的时间复杂度，是不稳定的。

```python
def selection_sort(arr):
    for p in range(len(arr)-1, 0, -1):
        position = 0
        for location in range(1, p+1):
            if arr[location] > arr[position]:
                position = location
        if p != position:
            arr[p], arr[position] = arr[position], arr[p]
```

三、快速排序（Quick Sort）

快速排序是一种基于分治算法的排序算法，它选择一个元素作为基准，并通过将基准放置在已排序数组中的正确位置来围绕所选择的基准对给定数组进行分区。

时间复杂度：最好时为O(nlogn)，最差时为O(n^2)；空间复杂度：考虑递归堆栈，为O(n)，不考虑则为O(1)。

快速排序相对更适用于大数据集，在某些极端情况下会显现出较差的时间复杂度，是不稳定的。

P.S：似乎二叉搜索树的建树再加上中序遍历，就可以获得同样的结果。

```python
def quick_sort(arr, left, right):
    if left < right:
        position = partition(arr, left, right)
        quick_sort(arr, left, position-1)
```

```
        quick_sort(arr, position+1, right)
def partition(arr, left, right):
    i = left
    j = right-1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
```

四、归并排序（Merge Sort）

归并排序作为一种排序算法，其原理是将数组划分为更小的子数组，对每个子数组进行排序，然后将排序后的子数组合并在一起，形成最终的排序数组。（和快速排序同样的分治思想）

时间复杂度：O(nlogn)；空间复杂度：O(n)。

归并排序是一种天然的可并行化算法，而且稳定，因此特别适合用来处理大数据集，但它需要额外空间。

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort(left)
        merge_sort(right)
        i, j, k = 0, 0, 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j > len(right):
            arr[k] = right[j]
            j += 1
            k += 1
```

这里附两道题，求逆序数的，之前一直用树状数组.....

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

```
def merge(nums, left, mid, right):
    l_nums = nums[left:mid+1]
    r_nums = nums[mid+1:right+1]
    i, j = 0, 0
    count = 0
    for k in range(left, right+1):
        if i < len(l_nums) and (j >= len(r_nums) or l_nums[i] < r_nums[j]):
            nums[k] = l_nums[i]
            i += 1
        else:
            nums[k] = r_nums[j]
            j += 1
            count += len(l_nums)-i
    return count

def merge_sort(nums, left, right):
    if left < right:
        mid = (left+right)//2
        count = merge_sort(nums, left, mid)
        count += merge_sort(nums, mid+1, right)
        count += merge(nums, left, mid, right)
        return count
    return 0

while True:
    n = int(input())
    if n == 0:
        break
    nums = [int(input()) for _ in range(n)]
    print(merge_sort(nums, 0, n-1))
```

http://cs101.openjudge.cn/2024sp_routine/09201/

```
def merge(nums, left, mid, right):
    l_nums = nums[left: mid+1]
    r_nums = nums[mid+1: right+1]
    i, j = 0, 0
    count = 0
    for k in range(left, right+1):
        if i < len(l_nums) and (j >= len(r_nums) or l_nums[i] < r_nums[j]):
            nums[k] = l_nums[i]
            i += 1
        else:
            nums[k] = r_nums[j]
            count += len(l_nums)-i
```

```
            j += 1
    return count

def merge_sort(nums, left, right):
    if left < right:
        mid = (left+right)//2
        count = merge_sort(nums, left, mid)
        count += merge_sort(nums, mid+1, right)
        count += merge(nums, left, mid, right)
        return count
    return 0

n = int(input())
nums = list(map(int, input().split()))[::-1]
print(merge_sort(nums, 0, n-1))
```

其实用树状数组可以过（上面那一道会MLE）：

```
def low_bit(x):
    return x & -x
n = int(input())
nums = list(map(int,input().split()))
my_dict = dict(zip(sorted(nums),list(range(1,n+1))))
tree = [0 for _ in range(n+1)]
ans = 0
for i in range(n):
    num = my_dict[nums[i]]
    x = num - 1
    while x:
        ans += tree[x]
        x -= low_bit(x)
    while num <= n:
        tree[num] += 1
        num += low_bit(num)
print(ans)
```

五、插入排序（Insertion Sort）

插入排序是一种基本的排序算法，其思想在于通过先前的已排好序的数组得到目标插入元素的插入位置，从而达到不断排序的目的。

时间复杂度：O(n^2)；空间复杂度：O(1)。

插入排序是一种稳定的原地排序算法。

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

六、希尔排序（Shell Sort）

希尔排序可以看作是插入排序的变种，也就相当于可以交换远项。

时间复杂度：（最差）O(n^2)；空间复杂度：O(1)。

希尔排序的时间复杂度取决于算法的对象，是不稳定的。

```
def shell_sort(arr, n):
    gap = n // 2
    while gap > 0:
        j = gap
        while j < n:
            i = j - gap
            while i >= 0:
                if arr[i+gap] > arr[i]:
                    break
                else:
                    arr[i+gap], arr[i] = arr[i], arr[i+gap]
                i -= gap
            j += 1
        gap //= 2
```

七、堆排序（Heap Sort）

堆排序是一种基于完全二叉树（堆）的排序算法。它通过将待排序的元素构建成一个堆，然后利用堆的性质来实现排序。在堆中，每个结点的值都必须大于等于其子结点的值。

时间复杂度：O(nlogn)；空间复杂度：O(1)。

堆排序适合处理大型数据集，采取原地排序，但不稳定，因为可能会交换相同元素。

```
def heapify(arr, n, i):
    largest = i
    l = 2*i + 1
    r = 2*i + 2
    if l < n and arr[l] > arr[largest]:
```

```
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

八、计数排序（Counting Sort）

计数排序是一种非比较性的整数排序算法。它的基本思想是对于给定的输入序列中的每一个元素x，确定该序列中值小于x的元素的个数。利用这个信息，可以直接确定x在输出序列中的位置，从而实现排序。

时间复杂度：O(n+k)，其中n为序列长度，k为序列数据范围。空间复杂度也为O(n+k)。

计数排序是一种稳定的排序算法，在数据集数值范围较小的情况下表现优越，但倘若数据离散程度较大则效率将会大大降低。

```
def counting_sort(arr):
    max_val = max(arr)
    min_val = min(arr)
    counts = [0 for _ in range(max_val-min_val+1)]
    for num in arr:
        counts[num-min_val] += 1
    for i in range(1, max_val-min_val+1):
        counts[i] += counts[i-1]
    result = [0 for _ in range(len(arr))]
    for num in reversed(arr):
        result[counts[num-min_val]-1] = num
        counts[num-min_val] -= 1
    return result
```

九、桶排序（Bucket Sort）

桶排序将数组分割成几个部分（桶），然后对每个桶进行排序，最后将所有桶中的元素合并起来得到最终的有序数组。它的基本思想是将待排序数组分割成若干个较小的数组（桶），每个桶再分别排序，最后按照顺序依次取出各个桶中的元素即可。

时间复杂度：理想情况下为O(n+k)，极端情况下为O(n^2)。空间复杂度同样。

桶排序适合相对均匀稠密的数字序列，在处理小数序列时相对更具有优势（小数更适合用于建立桶的索引与数字之间的映射）。

P.S：桶排序中每个桶使用的排序方法时其它的排序方法，所以桶更适合理解为一种思想，同样的，代码不在此展示。

十、基数排序（Radix Sort）

基数排序是一种非比较性的整数排序算法，它根据键值的每位数字来进行排序。基数排序的核心思想是将待排序的元素按照位数切割成不同的数字，然后按照这些位数分别进行排序。这种排序算法属于分配式排序，它的性能取决于桶的使用方法。

时间复杂度：O(d*(n+k))，空间复杂度：O(n+k)。

基数排序不依赖于比较操作，适用于整数等固定长度的数据类型，并且在某些情况下具有稳定性。但它需要额外的空间，而且对于位数较大的数据可能不太实用。

```
def radix_sort(arr):
    max_num = max(arr)
    max_length = len(str(max_num))
    buckets = [[] for _ in range(10)]
    for digit in range(max_length):
        for num in arr:
            bucket_idx = (num // 10**digit) % 10
            buckets[bucket_idx].append(num)
        arr = [num for bucket in buckets for num in bucket]
        buckets = [[] for _ in range(10)]
    return arr
```

Part 3：栈和队列

由于在线性表部分已经基本给出了栈和队列的实现形式，因此在这部分不做赘述。个人认为栈和队列更多地是一种思想，蕴含着元素选取顺序的逻辑。

一、栈

1、栈

https://www.luogu.com.cn/problem/P1044

以第一个元素举例：

在出栈序列的第一个时，应该先进并且马上出；

在第二个时，应该第二个元素先进然后马上出；

在第三个时，第二、三个元素只要保证先出即可；

……（依次直至在最后一个）

记n个元素对应的种数为f(n)，则有公式：

```
f(n)=f(0)*f(n-1)+f(1)*f(n-2)+……+f(n-1)*f(0)
```

这其实就是卡特兰数(Catalan number)，其公式为：

$$f(n)=\frac{C_{2n}^{n}}{n+1}$$

接下来就可以直接敲代码AC了。

```
import math
n = int(input())
print(math.comb(2*n, n)//(n+1))
```

2、日志分析

https://www.luogu.com.cn/problem/P1165

这道题本来想着用堆+懒删除做，但是确实不如使用辅助栈维护最大值。（参考快速堆猪，一样的道理）

```
stack = []
for _ in range(int(input())):
    operation = list(map(int, input().split()))
    if operation[0] == 0:
        if stack:
            stack.append(max(stack[-1], operation[1]))
        else:
            stack.append(operation[1])
    elif operation[0] == 1:
        if stack:
            stack.pop()
    else:
        if stack:
            print(stack[-1])
        else:
            print(0)
```

3、棋盘制作

https://www.luogu.com.cn/problem/P1169

这道题有很多方法，悬线法、dp等等，但是个人认为使用单调栈做一做是对于栈的理解帮助不小的（因为我最开始就没理解明白，想了好久hhh）。

```
n, m = map(int, input().split())
a, b = 1, 1
h = [1 for _ in range(m)]
stack, up = [], []

def cut(height, end):
    global a, b
    while stack and height <= h[stack[-1]]:
        k = stack.pop()
        l = stack[-1] if stack else start
        a = max(a, h[k]*(end-l))
        b = max(b, min(h[k], end-l)**2)

for i in range(n):
    row = list(map(int, input().split()))
    left = 0
    for j, c in enumerate(row):
        if i:
            h[j] = 1 if c == up[j] else h[j]+1
        if not j:
            start = -1
        elif c == left:
            cut(0, j-1)
            start = j-1
        else:
            cut(h[j], j-1)
        stack.append(j)
        left = c
    cut(0, m-1)
    up = row
print(f'{b}\n{a}')
```

4、表达式的转换

https://www.luogu.com.cn/problem/P1175

这道题对于栈的应用算是比较经典且常见的（后面的树部分经常用到），同时需要注意幂指数是从右向左算。以及由于数据给的不好，需要对输入作 strip 处理。

```
def trans(infix):
    stack = []
    output = []
    judge = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    for char in infix:
```

```
        if char in judge:
            while stack and stack[-1] in judge and judge[char] <=
judge[stack[-1]]:
                if char == '^':
                    break
                output.append(stack.pop())
            stack.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()
        else:
            output.append(char)
    while stack:
        output.append(stack.pop())
    return output

string = list(input().strip())
postfix = trans(string)
print(*postfix)
while len(postfix) > 1:
    stack = []
    while True:
        char = postfix.pop(0)
        if char in '+-*/':
            a = stack.pop()
            b = stack.pop()
            stack.append(str(int(eval(b+char+a))))
            break
        elif char == '^':
            a = stack.pop()
            b = stack.pop()
            stack.append(str(int(b)**int(a)))
            break
        else:
            stack.append(char)
    postfix = stack+postfix
    print(*postfix)
```

5、最大数

https://www.luogu.com.cn/problem/P1198

这道题考察单调栈，同时用bisect库实现二分查找（很好的工具），如果只是对单调栈进行直接的修改，会超时（大概得80分？），这时也可以使用并查集，但是感觉这么做相对最好想。

```
import bisect
m, d = map(int, input().split())
f, id = [], []
t, l = 0, 0
for _ in range(m):
```

```
    check, x = input().split()
    x = int(x)
    if check == 'A':
        x = (x+t) % d
        while f and f[-1] <= x:
            f.pop()
            id.pop()
        f.append(x)
        id.append(l)
        l += 1
    else:
        t = f[bisect.bisect_left(id, l-x)]
        print(t)
```

6、赛车

https://www.luogu.com.cn/problem/P3256

这道题同样考察单调栈，其实主要就是利用栈的特性去构造一种贪心策略，从而得到能够获奖的赛车（得到结果其实是按照它们获得头名的顺序），那么最后的答案就显而易见了。

```
class Node:
    def __init__(self, x, v, id):
        self.x = x
        self.v = v
        self.id = id

def time(a, b):
    if a.v == b.v:
        return float('inf')
    return (a.x-b.x)/(b.v-a.v)

n = int(input())
X = list(map(int, input().split()))
V = list(map(int, input().split()))
races = [Node(0, 0, i) for i in range(1, n+1)]
for i in range(n):
    races[i].x = X[i]
for i in range(n):
    races[i].v = V[i]
races.sort(key=lambda x: (x.v, x.x))
stack = []
stack.append(races[0])
for k in range(1, n):
    p = races[k]
    while stack:
        if len(stack) == 1:
            if p.x > stack[-1].x:
                stack.pop()
            else:
                break
        else:
            if p.x > stack[-1].x:
```

```
                stack.pop()
            elif time(stack[-1], p) < time(stack[-2], stack[-1]):
                stack.pop()
            else:
                break
    stack.append(p)
nums = [p.id for p in stack]
nums.sort()
print(len(nums))
print(*nums)
```

二、队列

1、合唱队形

https://www.luogu.com.cn/problem/P1091

先分别求出以对应同学为结尾的长度（向左&向右），再扫一遍即可。（利用了一些单调队列的思想）

```
n = int(input())
nums = list(map(int, input().split()))
a, b = [1 for _ in range(n)], [1 for _ in range(n)]
for i in range(n):
    for j in range(i):
        if nums[i] > nums[j]:
            a[i] = max(a[i], a[j]+1)
for i in range(n-1, -1, -1):
    for j in range(n-1, i, -1):
        if nums[i] > nums[j]:
            b[i] = max(b[i], b[j]+1)
ans = 0
for k in range(n):
    ans = max(ans, a[k]+b[k]-1)
print(n-ans)
```

2、机器人搬重物

https://www.luogu.com.cn/problem/P1126

典型的bfs，使用双端队列可以降时间复杂度，同时这里是不建议使用vis的，因为变量太多，用memo去判断会更方便一些，当然这里对于题目的理解也需要注意一下，机器人是不能在边缘行走的（可以认为边缘是墙壁）。

P.S：Openjudge上面也有这道题——Robot：http://cs101.openjudge.cn/dsapre/01376/

```
from collections import deque
def bfs(start, end):
    if start[0] == end[0] and start[1] == end[1]:
```

```
        return 0
    queue = deque()
    queue.append(start)
    dir = [(0, -1), (-1, 0), (0, 1), (1, 0)]
    while queue:
        x, y, d, t = queue.popleft()
        if memo[x][y][(d+3) % 4] > t+1:
            memo[x][y][(d+3) % 4] = t+1
            queue.append([x, y, (d+3) % 4, t+1])
        if memo[x][y][(d+1) % 4] > t+1:
            memo[x][y][(d+1) % 4] = t+1
            queue.append([x, y, (d+1) % 4, t+1])
        for k in range(1, 4):
            nx = x + dir[d][0]*k
            ny = y + dir[d][1]*k
            if 1 <= nx < n and 1 <= ny < m:
                if maze[nx][ny]:
                    break
                if nx == end[0] and ny == end[1]:
                    return t+1
                if memo[nx][ny][d] > t+1:
                    memo[nx][ny][d] = t+1
                    queue.append([nx, ny, d, t+1])
    return -1

n, m = map(int, input().split())
matrix = [list(map(int, input().split())) for _ in range(n)]
maze = [[0 for _ in range(m+1)] for _ in range(n+1)]
for i in range(n):
    for j in range(m):
        if matrix[i][j]:
            maze[i][j], maze[i][j+1], maze[i+1][j], maze[i+1][j+1] = 1, 1, 1, 1
memo = [[[float('inf') for _ in range(4)] for _ in range(m+1)] for _ in
range(n+1)]
z = input().split()
judge = {'W': 0, 'N': 1, 'E': 2, 'S': 3}
d = judge[z[4]]
start = [int(z[0]), int(z[1]), d, 0]
end = [int(z[2]), int(z[3])]
memo[start[0]][start[1]][d] = 0
print(bfs(start, end))
```

3、逛画展

https://www.luogu.com.cn/problem/P1638

这道题和滑动窗口很像，总之就是开一个字典（或者开一个数组也可）记录出现的是哪个画家的作品，扫一遍，保留最短的答案就可以。

```
from collections import defaultdict
n, m = map(int, input().split())
nums = list(map(int, input().split()))
my_dict = defaultdict(int)
```

```
my_dict[nums[0]] = 1
k = 1
ans = float('inf')
left, right, a, b = 0, 0, 0, 0
while left <= right < n:
    if k == m:
        if ans > right - left + 1:
            ans = right - left + 1
            a, b = left, right
        my_dict[nums[left]] -= 1
        if my_dict[nums[left]] == 0:
            k -= 1
        left += 1
    else:
        if right == n-1:
            break
        right += 1
        my_dict[nums[right]] += 1
        if my_dict[nums[right]] == 1:
            k += 1
print(a+1, b+1)
```

4、Saving Tong Monk

http://cs101.openjudge.cn/practice/04130/

个人感觉做了这道题之后一般的bfs都不会出问题了😂，思路是用优先队列（PriorityQueue或者heapq）储存结点，然后获得时间最短的，再拿记忆化搜索就可以（开个三维数组），同时记得使用二进制记录蛇的存活状态。

```
from queue import PriorityQueue

class Node:
    def __init__(self, x, y, t, k, s):
        self.x = x
        self.y = y
        self.t = t
        self.k = k
        self.s = s
    def __lt__(self, other):
        return self.t < other.t

def bfs(maze, n, m):
    x0, y0, count = 0, 0, 0
    lst = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if maze[i][j] == 'K':
                x0, y0 = i, j
            if maze[i][j] == 'S':
                lst[i][j] = count
                count += 1
```

```
    dir = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    memo = [[[float('inf') for _ in range(m+1)] for _ in range(n)] for _ in
range(n)]
    queue = PriorityQueue()
    queue.put(Node(x0, y0, 0, 0, 0))
    memo[x0][y0][0] = 0
    while not queue.empty():
        node = queue.get()
        if maze[node.x][node.y] == 'T' and node.k == m:
            return node.t
        for a, b in dir:
            nx, ny = node.x+a, node.y+b
            if 0 <= nx < n and 0 <= ny < n:
                if maze[nx][ny] == '#':
                    continue
                elif maze[nx][ny] == 'S':
                    if (node.s >> lst[nx][ny]) & 1:
                        if node.t+1 < memo[nx][ny][node.k]:
                            memo[nx][ny][node.k] = node.t+1
                            queue.put(Node(nx, ny, node.t+1, node.k, node.s))
                    else:
                        if node.t+2 < memo[nx][ny][node.k]:
                            memo[nx][ny][node.k] = node.t+2
                            queue.put(Node(nx, ny, node.t+2, node.k, node.s | (1
<< lst[nx][ny])))
                elif maze[nx][ny].isdigit():
                    if int(maze[nx][ny]) == node.k+1:
                        if node.t+1 < memo[nx][ny][node.k+1]:
                            memo[nx][ny][node.k+1] = node.t+1
                            queue.put(Node(nx, ny, node.t+1, node.k+1, node.s))
                    else:
                        if node.t+1 < memo[nx][ny][node.k]:
                            memo[nx][ny][node.k] = node.t+1
                            queue.put(Node(nx, ny, node.t+1, node.k, node.s))
                else:
                    if node.t+1 < memo[nx][ny][node.k]:
                        memo[nx][ny][node.k] = node.t+1
                        queue.put(Node(nx, ny, node.t+1, node.k, node.s))
    return 'impossible'

while True:
    n, m = map(int, input().split())
    if n == m == 0:
        break
    maze = [list(input()) for _ in range(n)]
    print(bfs(maze, n, m))
```

5、火锅盛宴

https://www.luogu.com.cn/problem/P4032

这道题不知道怎么才能减小时间，先把代码放这里吧（TLE版本）。

```
import heapq
from collections import defaultdict

def low_bit(x):
    return x & -x

def query(x, y):
    ans = 0
    while y > x:
        ans += tree[y]
        y -= low_bit(y)
    while x > y:
        ans -= tree[x]
        x -= low_bit(x)
    return ans

def add(x, k):
    while x <= n:
        tree[x] += k
        x += low_bit(x)

for _ in range(int(input())):
    n = int(input())
    s = list(map(int, input().split()))
    tree = [0 for _ in range(n+1)]
    eat = defaultdict(int)
    food = [[] for _ in range(n)]
    for _ in range(int(input())):
        operation = list(map(int, input().split()))
        if operation[1] == 0:
            heapq.heappush(food[operation[2]-1], operation[0]+s[operation[2]-1])
        else:
            t = operation[0]
            for k in range(n):
                while food[k] and food[k][0] <= t:
                    heapq.heappop(food[k])
                    eat[k+1] += 1
                    add(k+1, 1)
            if operation[1] == 1:
                for k in range(1, n+1):
                    if eat[k]:
                        add(k, -1)
                        eat[k] -= 1
                        print(k)
                        break
                else:
                    print('Yazid is angry.')
            elif operation[1] == 2:
                idx = operation[2]
                if eat[idx]:
```

```
                    eat[idx] -= 1
                    add(idx, -1)
                    print('Succeeded!')
                else:
                    if food[idx-1]:
                        print(food[idx-1][0]-t)
                    else:
                        print('YJQQQAQ is angry.')
            else:
                l, r = operation[2:]
                print(query(l-1, r))
```

Part 4：树

植树节开始讲/整理树这一部分，蛮好……

一、定义

1、节点和边

树由节点及连接节点的边构成。树有以下属性：

- ❑ 有一个根节点；
- ❑ 除根节点外，其他每个节点都与其唯一的父节点相连；
- ❑ 从根节点到其他每个节点都有且仅有一条路径；
- ❑ 如果每个节点最多有两个子节点，我们就称这样的树为二叉树。

2、递归

一棵树要么为空，要么由一个根节点和零棵或多棵子树构成，子树本身也是一棵树。每棵子树的根节点通过一条边连到父树的根节点。

二、数据结构

1、二叉堆

二叉堆通过树的特性，由列表实现，每次加入元素时，进行“上浮”操作，而删除元素时则进行“下沉”操作。

以下代码是http://cs101.openjudge.cn/practice/04078/的AC代码，自行定义BinHeap类实现了堆结构。

```
class BinHeap:
    def __init__(self):
        self.list = [0]
        self.size = 0
```

```
    def up(self, i):
        while i // 2 > 0:
            if self.list[i] < self.list[i // 2]:
                tmp = self.list[i // 2]
                self.list[i // 2] = self.list[i]
                self.list[i] = tmp
            i //= 2

    def heappush(self, k):
        self.list.append(k)
        self.size += 1
        self.up(self.size)

    def min(self, i):
        if i*2+1 > self.size:
            return i*2
        else:
            if self.list[i*2] < self.list[i*2+1]:
                return i*2
            else:
                return i*2+1

    def down(self, j):
        while (j*2) <= self.size:
            t = self.min(j)
            if self.list[j] > self.list[t]:
                tmp = self.list[j]
                self.list[j] = self.list[t]
                self.list[t] = tmp
            j = t

    def heappop(self):
        ans = self.list[1]
        self.list[1] = self.list[self.size]
        self.size -= 1
        self.list.pop()
        self.down(1)
        return ans

Q = BinHeap()
for _ in range(int(input())):
    operation = list(map(int, input().split()))
    if operation[0] == 1:
        Q.heappush(operation[1])
    else:
        print(Q.heappop())
```

2、AVL树

二叉搜索树中，如果每一个子节点都小于等于根节点，那么会导致查找效率大大降低，这也就是我们使用平衡二叉搜索树的原因。AVL树的平衡因子（左子树和右子树高度之差）绝对值小于1，因此可以大大加快查找效率。

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1

class AVL:
    def __init__(self):
        self.root = None

    def insert(self, v):
        if self.root is None:
            self.root = Node(v)
        else:
            self.root = self._insert(v, self.root)

    def _insert(self, v, node):
        if node is None:
            return Node(v)
        elif v < node.val:
            node.left = self._insert(v, node.left)
        else:
            node.right = self._insert(v, node.right)
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        balance = self._get_balance(node)
        if balance > 1:
            if v < node.left.val:
                return self.rotate_right(node)
            else:
                node.left = self.rotate_left(node.left)
                return self.rotate_right(node)
        elif balance < -1:
            if v > node.right.val:
                return self.rotate_left(node)
            else:
                node.right = self.rotate_right(node.right)
                return self.rotate_left(node)
        return node

    def _get_height(self, node):
        if node is None:
            return 0
        return node.height

    def _get_balance(self, node):
        if node is None:
            return 0
```

```
        return self._get_height(node.left) - self._get_height(node.right)

    def rotate_left(self, node):
        nd = node.right
        tmp = nd.left
        nd.left = node
        node.right = tmp
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        nd.height = 1 + max(self._get_height(nd.left),
self._get_height(nd.right))
        return nd

    def rotate_right(self, node):
        nd = node.left
        tmp = nd.right
        nd.right = node
        node.left = tmp
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        nd.height = 1 + max(self._get_height(nd.left),
self._get_height(nd.right))
        return nd

    def _prefix(self, node):
        if node is None:
            return []
        return [node.val]+self._prefix(node.left)+self._prefix(node.right)

    def prefix(self):
        return ' '.join(map(str, self._prefix(self.root)))
```

三、题目

1、求二叉树的高度和叶子数目

http://dsbpython.openjudge.cn/dspythonbook/P0610/

先写一个TreeNode的类，然后用列表进行状态的更新，再找到哪个是根节点，最后就可以进行高度计算和叶子数数了。

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

def build(n, nodes):
    tree = [TreeNode(i) for i in range(n)]
    check = set()
    for k in range(n):
```

```
        l, r = nodes[k]
        if l != -1:
            tree[k].left = tree[l]
            check.add(l)
        if r != -1:
            tree[k].right = tree[r]
            check.add(r)
    for p in range(n):
        if p not in check:
            return tree[p]

def height(root):
    if root is None:
        return 0
    return max(height(root.left), height(root.right))+1

def count(root):
    if root is None:
        return 0
    if root.left is None and root.right is None:
        return 1
    return count(root.left)+count(root.right)

n = int(input())
nodes = [list(map(int, input().split())) for _ in range(n)]
root = build(n, nodes)
print(height(root)-1, count(root))
```

2、Disk Tree

http://cs101.openjudge.cn/dsapre/01760/

虽然没有写类，但是其实已经很好地锻炼了树的思想，怎么建&怎么输出，这都是很有考究的。这里使用的setdefault方法，很有意思，或许之后还会用到。

```
tree = {}
for _ in range(int(input())):
    components = input().split('\\')
    node = tree
    for tmp in components:
        node = node.setdefault(tmp, {})
def Print(tree, depth):
    for key in sorted(tree.keys()):
        print(' '*depth+key)
        Print(tree[key], depth+1)
Print(tree, 0)
```

3、重建二叉树

http://cs101.openjudge.cn/dsapre/02255/

最经典的题目，通过两种遍历来得到另一种遍历。

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

def build(prefix, infix):
    if len(prefix) == 0:
        return None
    v = prefix[0]
    root = TreeNode(v)
    idx = infix.index(v)
    root.left = build(prefix[1:idx+1], infix[:idx])
    root.right = build(prefix[idx+1:], infix[idx+1:])
    return root

def postfix(root):
    if root is None:
        return ''
    return postfix(root.left)+postfix(root.right)+root.val

while True:
    try:
        prefix, infix = input().split()
        print(postfix(build(prefix, infix)))
    except EOFError:
        break
```

4、四分树

http://cs101.openjudge.cn/practice/01610/

这次树的子节点大于二，因此我们用列表来存储。

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.children = []

def build(n, matrix):
    check = sum(sum(row) for row in matrix)
    if check == 0:
        return TreeNode('00')
    elif check == n**2:
        return TreeNode('01')
    else:
        a = [matrix[i][:n//2] for i in range(n//2)]
```

```
        b = [matrix[i][n//2:] for i in range(n//2)]
        c = [matrix[i][:n//2] for i in range(n//2, n)]
        d = [matrix[i][n//2:] for i in range(n//2, n)]
        root = TreeNode('1')
        for p in [a, b, c, d]:
            root.children.append(build(n//2, p))
        return root

def get(root):
    result = ''
    queue = [root]
    while queue:
        node = queue.pop(0)
        result += node.val
        queue += node.children
    return result

for _ in range(int(input())):
    n = int(input())
    matrix = [list(map(int, input().split())) for _ in range(n)]
    ans = get(build(n, matrix))
    p = int(ans, 2)
    print(hex(p)[2:].upper())
```

5、表达式·表达式树·表达式求值

http://cs101.openjudge.cn/practice/05430/

有一说一，这道题能顺着做下来，感觉树这方面就没啥问题了？首先使用栈进行中序转后序，然后建树，之后需要注意树的打印，最后再进行一个带入求值。

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

def trans(infix):
    stack = []
    result = []
    operator = {'+': 1, '-': 1, '*': 2, '/': 2}
    for token in infix:
        if token.isalpha():
            result.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                result.append(stack.pop())
            stack.pop()
        else:
            while stack and stack[-1] in operator and operator[token] <=
operator[stack[-1]]:
```

```
                result.append(stack.pop())
            stack.append(token)
    while stack:
        result.append(stack.pop())
    return result

def build(postfix):
    stack = []
    for char in postfix:
        if char in '+-*/':
            node = TreeNode(char)
            node.right = stack.pop()
            node.left = stack.pop()
        else:
            node = TreeNode(char)
        stack.append(node)
    return stack[0]

def depth(root):
    if root is None:
        return 0
    return max(depth(root.left), depth(root.right))+1

def Print(root, d):
    if d == 0:
        return root.val
    graph = [' '*(2**d-1)+root.val+' '*(2**d-1)]
    graph.append(' '*(2**d-2)+('/' if root.left is not None else ' ')
                 +' '+('\\' if root.right is not None else ' ')+' '*(2**d-2))
    d -= 1
    l = Print(root.left, d) if root.left is not None else [' '*(2**(d+1)-1)]*
(2*d+1)
    r = Print(root.right, d) if root.right is not None else [' '*(2**(d+1)-1)]*
(2*d+1)
    for i in range(2*d+1):
        graph.append(l[i]+' '+r[i])
    return graph

def cal(root):
    if root.val.isalpha():
        return my_dict[root.val]
    else:
        lv = cal(root.left)
        rv = cal(root.right)
        return int(eval(str(lv)+root.val+str(rv)))

infix = list(input())
postfix = trans(infix)
root = build(postfix)
my_dict = {}
for _ in range(int(input())):
    a, b = input().split()
    my_dict[a] = b
print(''.join(postfix))
d = depth(root)
result = Print(root, d-1)
```

```
for part in result:
    print(part)
print(cal(root))
```

6、树的转换

http://cs101.openjudge.cn/practice/04081/

这个题不难，但是很好地诠释了什么叫做“左儿子右兄弟”，当时我理解了半天还没明白的东西hhh。

```
s = list(input())
a, b, h, nh = 0, 0, 0, 0
now = [0]
for char in s:
    if char == 'd':
        a, b = a+1, b+1
        now.append(a)
    else:
        a, b = now.pop(), b-1
    nh = max(nh, a)
    h = max(h, b)
print(f'{h} => {nh}')
```

7、森林的带度数层次序列存储

http://dsbpython.openjudge.cn/dspythonbook/P0770/

通过层次遍历的结果进行建树。

```
from collections import deque

class TreeNode:
    def __init__(self, val, degree):
        self.val = val
        self.degree = degree
        self.children = []

def build(nodes):
    queue = deque()
    a, b = nodes.pop(0)
    root = TreeNode(a, b)
    f = root
    queue.append(f)
    while queue:
        node = queue.popleft()
        for _ in range(node.degree):
            a, b = nodes.pop(0)
            tmp = TreeNode(a, b)
            node.children.append(tmp)
            queue.append(tmp)
```

```
    return root

def postfix(root):
    ans = []
    for node in root.children:
        ans += postfix(node)
    ans.append(root.val)
    return ans

result = []
for _ in range(int(input())):
    lst = list(input().split())
    n = len(lst)//2
    nodes = [[lst[2*i], int(lst[2*i+1])] for i in range(n)]
    result += postfix(build(nodes))
print(' '.join(result))
```

8、二叉搜索树的层次遍历

http://dsbpython.openjudge.cn/dspythonbook/P1320/

二叉搜索树，就是左子节点小于根节点，根节点小于右子节点。知道这个剩下的就好做了。

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

def build(root, k):
    if root is None:
        return TreeNode(k)
    if root.val < k:
        root.right = build(root.right, k)
    if root.val > k:
        root.left = build(root.left, k)
    return root

def operate(root):
    result = []
    queue = [root]
    while queue:
        node = queue.pop(0)
        result.append(node.val)
        if node.left is not None:
            queue.append(node.left)
        if node.right is not None:
            queue.append(node.right)
    return ' '.join(map(str, result))

nodes = list(map(int, input().split()))
root = None
for node in nodes:
```

```
    root = build(root, node)
print(operate(root))
```

9、括号嵌套二叉树

http://dsbpython.openjudge.cn/dspythonbook/P0680/

使用栈对括号和逗号进行处理，之后进行遍历即可。

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.children = []

def build(nodes):
    if not nodes:
        return None
    if '(' not in nodes:
        return TreeNode(nodes[0])
    root = TreeNode(nodes[0])
    stack = []
    tmp = []
    tree = []
    for char in nodes[2: -1]:
        if char == '(':
            stack.append(char)
        elif char == ')':
            stack.pop()
        elif char == ',' and not stack:
            tree.append(tmp)
            tmp = []
            continue
        tmp.append(char)
    tree.append(tmp)
    for t in tree:
        root.children.append(build(t))
    return root

def prefix(root):
    if root is None:
        return ''
    ans = root.val
    for child in root.children:
        ans += prefix(child)
    return ans

def postfix(root):
    if root is None:
        return ''
    ans = ''
    for child in root.children:
        ans += postfix(child)
    ans += root.val
```

```
    return ans

nodes = list(input())
root = build(nodes)
print(prefix(root))
print(postfix(root))
```

10、哈夫曼编码树

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

非常经典的一种算法！需要注意，我们之前的建树过程都是自上向下建立（先建立根节点，之后不断添加子树），这里我们自下向上，逐渐向上，“拼凑”出一棵完整的树。

```
import heapq

class TreeNode:
    def __init__(self, weight, val=None):
        self.weight = weight
        self.val = val
        self.left = None
        self.right = None
    def __lt__(self, other):
        if self.weight == other.weight:
            return self.val < other.val
        return self.weight < other.weight

def build(nodes):
    Q = []
    for p, char in nodes:
        heapq.heappush(Q, TreeNode(p, char))
    while len(Q) > 1:
        l = heapq.heappop(Q)
        r = heapq.heappop(Q)
        merge = TreeNode(l.weight + r.weight)
        merge.left = l
        merge.right = r
        heapq.heappush(Q, merge)
    return Q[0]

def encode(root):
    codes = {}
    def operate(node, code):
        if node.val is not None:
            codes[node.val] = code
        else:
            operate(node.left, code+'0')
            operate(node.right, code+'1')
    operate(root, '')
    return codes

def encoding(codes, s):
    ans = ''
```

```
    for p in s:
        ans += codes[p]
    return ans

def decoding(root, s):
    ans = ''
    node = root
    for p in s:
        if p == '0':
            node = node.left
        else:
            node = node.right
        if node.val is not None:
            ans += node.val
            node = root
    return ans

nodes = []
for _ in range(int(input())):
    a, b = input().split()
    nodes.append([int(b), a])
root = build(nodes)
codes = encode(root)
while True:
    try:
        s = list(input())
        if s[0].isdigit():
            print(decoding(root, s))
        else:
            print(encoding(codes, s))
    except EOFError:
        break
```

11、01 Tree

https://www.luogu.com.cn/problem/CF1919D

为了不让大家看英文，所以选的洛谷上面的这个（好吧是我不想看……）。感觉这道题也是需要从下向上对树进行处理的思维，挨个对叶子节点进行处理（因为输入数据是按照顺序来的），逐渐向上，相当于逐渐收到最终的根节点，如果根节点符合要求，那就输出YES。当然了，处理的途中加入一些特判也是需要的。

```
for _ in range(int(input())):
    n = int(input())
    stack = [[-1, 0]]
    check = 0
    top = 1
    judge = True
    for x in map(int, input().split()):
        x += 1
        check += (x == 1)
        if check > 1:
```

```
            judge = False
            break
        if x <= stack[-1][1]:
            judge = False
            break
        elif x == stack[-1][1]+1:
            if stack[-1][0] == stack[-1][1]:
                stack.pop()
            else:
                stack[-1][1] -= 1
        elif top <= x-2:
            stack.append([top, x-2])
        top = x
    if stack != [[-1, -1]] or check != 1 or not judge:
        print('NO')
    else:
        print('YES')
```

12、Preorder

https://www.luogu.com.cn/problem/CF1671E

这道题我没有使用dp（对于dp我是一直用的不熟练，树形dp会在后面出现的），直接递归，取子树的先序遍历，进行比较，一旦有不一样的就乘以2。需要注意，在返回先序遍历时应当比较左子树和右子树的字典序，这相当于进行了判重。

```
n = int(input())
s = ['']+list(input())
ans, mod = 1, 998244353
def recursion(x):
    global ans
    if x >= 1 << (n-1):
        return s[x]
    l = recursion(x*2)
    r = recursion(x*2+1)
    if l != r:
        ans = (ans*2) % mod
    if l > r:
        l, r = r, l
    return s[x]+l+r
recursion(1)
print(ans)
```

13、文件结构“图”

http://cs101.openjudge.cn/practice/02775/

这道题和Disk Tree比较像，都属于目录树的内容（当时找到Disk Tree的时候就记得有一道也很有意思，但是没找到。。。），对于树的思想，是很有帮助的。

```
op = '|     '

def dfs(n, k):
    file = []
    c = n
    while True:
        tmp = input()
        if tmp[0] == '#':
            return
        if c == 1:
            print(f'DATA SET {k}:')
            print('ROOT')
            c += 1
        if tmp[0] == '*':
            file.sort()
            for f in file:
                print(op*(n-1)+f)
            print()
            dfs(1, k+1)
            return
        elif tmp[0] == 'd':
            print(op*n+tmp)
            dfs(n+1, k)
        elif tmp[0] == ']':
            file.sort()
            for f in file:
                print(op*(n-1)+f)
            return
        else:
            file.append(tmp)

dfs(1, 1)
```

14、魔族密码

https://www.luogu.com.cn/problem/P1481

这道题就是可以用dp或字典树来解决，这里都展示出来，都是很有用的。

dp：

```
n = int(input().strip())
s = [input().strip() for _ in range(n)]
dp = [1 for _ in range(n)]
for i in range(1, n):
    for j in range(i-1, -1, -1):
        if s[i].find(s[j]) == 0:
            dp[i] = max(dp[j]+1, dp[i])
print(max(dp))
```

字典树：

```
tree = [[0 for _ in range(26)] for _ in range(2*10**5)]
word = [0 for _ in range(10**6)]
n = int(input().strip())
tot, ans = 0, 0

def insert(s):
    global tot, ans
    u, res = 0, 0
    for c in s:
        k = ord(c)-ord('a')
        if tree[u][k] == 0:
            tot += 1
            tree[u][k] = tot
        u = tree[u][k]
        res += word[u]
    word[u] += 1
    ans = max(ans, res+1)

for _ in range(n):
    insert(input().strip())
print(ans)
```

15、没有上司的舞会

https://www.luogu.com.cn/problem/P1352

这道题是非常经典的一道树形dp，感觉其实有了dp的思想后，主要需要的就是应用在各种各样的数据结构上面。

```
import sys
sys.setrecursionlimit(1 << 30)

n = int(input())
r = [0]+[int(input()) for _ in range(n)]
dp = [[0, r[i]] for i in range(n+1)]
graph = [[] for _ in range(n+1)]
check = set()
for _ in range(n-1):
    u, v = map(int, input().split())
    graph[v].append(u)
    check.add(u)
boss = [i for i in range(1, n+1) if i not in check][0]

def dfs(x):
    for y in graph[x]:
        dfs(y)
        dp[x][0] += max(dp[y][0], dp[y][1])
        dp[x][1] += dp[y][0]

dfs(boss)
print(max(dp[boss]))
```

Part 5：图

最后一个部分！

一、定义

图论(Graph theory)是数学的一个分支，图是图论的主要研究对象。**图(Graph)**是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

图(Graph)是一个二元组G=(V(G),E(G))。其中V(G)是非空集，称为点集(vertex set)，对于V中的每个元素，我们称其为**顶点**(vertex)或**节点**(node)，简称**点**；E(G)为V(G)各节点之间边的集合，称为**边集**(edge set)。

图有多种，包括**无向图(undirected graph)**，**有向图(directed graph)**，**混合图(mixed graph)**等。

若G为无向图，则E中的每个元素为一个无序二元组(u,v)，称作**无向边(undirected edge)**，简称为**边(edge)**，其中$u, v \in V$。设e=(u,v)，则u和v称为e的**端点(endpoint)**。

若G为有向图，则E中的每个元素为一个有序二元组(u,v)，有时也写作$u \to v$，称作**有向边(directed\space edge)**或**弧(arc)**，在不引起混淆的情况下也可以称作**边(edge)**。设$e = u \to v$，则此时u称为e的**起点(tail)**，v称为e的**终点(head)**，起点和终点也称为e的**端点(endpoint)**，并称u是v的直接前驱，v是u的直接后继。

若G为混合图，则E中既有**有向边**，又有**无向边**。

若G的每条边e_k=(u_k,v_k)都被赋予一个数作为该边的**权**，则称G为**赋权图**。如果这些权都是正实数，就称G为**正权图**。

与一个顶点v关联的边的条数称作该顶点的**度(degree)**，记作d(v)。特别地，对于边(v,v)，则每条这样的边要对d(v)产生2的贡献。

握手定理（图论基本定理）：对于任何无向图G=(V,E)，有$\sum_{v \in V} d(v) = 2\backslash \mathrm{abs} E$。

对一张图，所有节点的度数的最小值称为G的**最小度(minimum degree)**，记作$\delta(G)$；最大值称为**最大度(maximum degree)**，记作$\Delta(G)$。即：$\delta(G) = min_{v \in G}\ d(v)$，$\Delta(G) = max_{v \in G}\ d(v)$。

在有向图G=(V,E)中，以一个顶点v为起点的边的条数称为该节点的**出度(out-degree)**，记作$d^+(v)$。以一个顶点v为终点的边的条数称为该节点的**入度(in-degree)**，记作$d^-(v)$。显然$d^+(v) + d^-(v) = d(v)$。

图G的点数$\backslash \mathrm{abs} V(G)$也被称作图G的**阶(order)**。

形象地说，图是由若干点以及连接点与点的边构成的。

二、图的表示方法（图的存储）

1、直接存边

方法：

使用一个数组来存边，数组中的每个元素都包含一条边的起点与终点（带边权的图还包含边权）。（或者使用多个数组分别存起点，终点和边权）

参考代码：

```
class Edge:
    def __init__(self, u=0, v=0):
        self.u = u
        self.v = v

n, m = map(int, input().split())
graph = [Edge() for _ in range(m)]
vis = [False for _ in range(n)]
for i in range(m):
    graph[i].u, graph[i].v = map(int, input().split())

def find_edge(a, b):
    for k in range(m):
        if graph[k].u == a and graph[k].v == b:
            return True
    return False

def dfs(p):
    if vis[p]:
        return
    vis[p] = True
    for j in range(m):
        if graph[j].u == p:
            dfs(graph[j].v)
```

复杂度：

查询是否存在某条边：O(m)。

遍历一个点的所有出边：O(m)。

遍历整张图：O(nm)。

空间复杂度：O(m)。

应用：

由于直接存边的遍历效率低下，一般不用于遍历图。

在$Kruskal$算法中，由于需要将边按边权排序，需要直接存边。

在有的题目中，需要多次建图（如建一遍原图，建一遍反图），此时既可以使用多个其它数据结构来同时存储多张图，也可以将边直接存下来，需要重新建图时利用直接存下的边来建图。

2、邻接矩阵

方法:

使用一个二维数组graph来存边，其中graph[u] [v]为1表示存在u到v的边，为0表示不存在。如果是带边权的图，可以在graph[u] [v]中存储u到v的边的边权。

参考代码:

```
n, m = map(int, input().split())
graph = [[False for _ in range(n+1)] for _ in range(n+1)]
vis = [False for _ in range(n+1)]
for _ in range(m):
    u, v = map(int, input().split())
    graph[u][v] = True

def find_edge(a, b):
    return graph[a][b]

def dfs(p):
    if vis[p]:
        return
    vis[p] = True
    for q in range(n+1):
        if graph[p][q]:
            dfs(q)
```

复杂度:

查询是否存在某条边：O(1)。

遍历一个点的所有出边：O(n)。

遍历整张图：O(n^2)。

空间复杂度：O(n^2)。

应用:

邻接矩阵只适用于没有重边（或重边可以忽略）的情况。

其最显著的优点是可以O(1)查询一条边是否存在。

由于邻接矩阵在稀疏图上效率很低（尤其是在点数较多的图上，空间无法承受），所以一般只会在稠密图上使用邻接矩阵。

3、邻接表

方法:

为了实现稀疏连接的图，更高效的方式是使用邻接表。在邻接表实现中，我们为图对象的所有顶点保存一个主列表，同时为每一个顶点对象都维护一个列表，其中记录了与它相连的顶点。

参考代码:

```
n, m = map(int, input().split())
```

```
graph = {i: [] for i in range(1, n+1)}
for _ in range(m):
    u, v = map(int, input().split())
    graph[u].append(v)
vis = [False for _ in range(n+1)]

def find_edge(a, b):
    for tmp in graph[a]:
        if tmp == b:
            return True
    return False

def dfs(p):
    if vis[p]:
        return
    vis[p] = True
    for q in graph[p]:
        dfs(q)
```

复杂度:

查询是否存在u到v的边：$O(d^+(v))$。（若排序，二分查找可以降低复杂度）

遍历点u的所有出边：$O(d^+(v))$。

遍历整张图：O(n+m)。

空间复杂度：O(m)。

应用:

存各种图都很适合，除非有特殊需求（如需要快速查询一条边是否存在，且点数较少，可以使用邻接矩阵）。

尤其适用于需要对一个点的所有出边进行排序的场合。

4、链式前向星

如果说邻接表是不好写但效率好，邻接矩阵是好写但效率低的话，前向星就是一个相对中庸的数据结构。前向星固然好写，但效率并不高。而在优化为链式前向星后，效率也得到了较大的提升。虽然说，世界上对链式前向星的使用并不是很广泛，但在不愿意写复杂的邻接表的情况下，链式前向星也是一个很优秀的数据结构。

方法:

本质就是用链表实现的邻接表，但邻接表存的是点，而链式前向星存的是边。

参考代码:

```
class Edge:
    def __init__(self, to, w, nxt):
        self.to = to
        self.w = w
        self.next = nxt

n, m = map(int, input().split())
```

```
edge = [Edge(0, 0, 0) for _ in range(m)]
head = [-1 for _ in range(n+1)]
cnt = 0
for _ in range(m):
    u, v, w = map(int, input().split())
    edge[cnt].to = v
    edge[cnt].w = w
    edge[cnt].next = head[u]
    head[u] = cnt
    cnt += 1
vis = [False for _ in range(n+1)]

def find_edge(a, b):
    i = head[a]
    while i != -1:
        if edge[i].to == b:
            return True
        i = edge[i].next
    return False

def dfs(p):
    if vis[p]:
        return
    vis[p] = True
    i = head[u]
    while i != -1:
        dfs(edge[i].to)
        i = edge[i].next
```

复杂度:

查询是否存在u到v的边：$O(d^+(u))$。

遍历点u的所有出边：$O(d^+(u))$。

遍历整张图：O(n+m)。

空间复杂度：O(m)。

应用:

存各种图都很适合，但不能快速查询一条边是否存在，也不能方便地对一个点的出边进行排序。

优点是边是带编号的，有时会非常有用，而且如果cnt的初始值为奇数，存双向边时`i ^ 1`即是`i`的反边（常用于网络流）。

三、图的遍历

I. 广度优先搜索

1、词梯

http://cs101.openjudge.cn/practice/28046/

这道题显然是需要使用bfs的，但是如果仅仅只是使用邻接表储存这个图然后遍历的话，会出现MLE的情况。因此，我们使用通配符匹配，既然字符长度只有四个，那么我们考虑使用"xx_x"、“x_xx”这样的形式储存，同时进行遍历，能够减小内存并保证时间不会增加太多。

```
from collections import deque
def check(a, b):
    for k in range(len(a)):
        if a[k] == '_':
            continue
        if a[k] != b[k]:
            return False
    return True

n = int(input())
graph = {}
degree = {}
vis = {}
for _ in range(n):
    word = input()
    vis[word] = False
    for p in range(4):
        tmp = word[:p]+'_'+word[p+1:]
        if word[:p]+'_'+word[p+1:] not in graph:
            graph[tmp] = [word]
            degree[tmp] = 1
        else:
            graph[tmp].append(word)
            degree[tmp] += 1

def bfs():
    start, ending = input().split()
    queue = deque()
    queue.append([start, [start]])
    vis[start] = True
    for p in range(4):
        tmp = start[:p] + '_' + start[p + 1:]
        degree[tmp] -= 1
    while queue:
        wd, now = queue.popleft()
        if wd == ending:
            print(*now)
            return
        for p in range(4):
            tmp = wd[:p] + '_' + wd[p + 1:]
            if degree[tmp] > 0:
                for wor in graph[tmp]:
                    if not vis[wor]:
                        vis[wor] = True
```

```
                        degree[tmp] -= 1
                        queue.append([wor, now + [wor]])
    print('NO')
bfs()
```

2、子串变换

https://www.luogu.com.cn/problem/P1032

挺简单的一道搜索题，需要注意的是直接使用find函数的话，只能找到第一个字符串，后面若有重复的话就找不到。这种情况很好解决，同时记得开vis集合用来判重。

```
from collections import deque

A, B = input().split()
graph = {}
while True:
    try:
        a, b = input().split()
        if a in graph:
            graph[a].append(b)
        else:
            graph[a] = [b]
    except EOFError:
        break

queue = deque()
queue.append([A, 0])
vis = set()
vis.add(A)
while queue:
    now, t = queue.popleft()
    if t > 10:
        continue
    if now == B:
        print(t)
        break
    for key in graph:
        tmp = []
        st = 0
        while True:
            idx = now[st:].find(key)
            if idx == -1:
                break
            tmp.append(idx+st)
            st = idx+st+len(key)
        for idx in tmp:
            ni = idx+len(key)
            for v in graph[key]:
                new = now[:idx]+v+now[ni:]
                if new not in vis:
                    vis.add(new)
                    queue.append([now[:idx]+v+now[ni:], t+1])
```

```
else:
    print('NO ANSWER!')
```

3、树的计数

https://www.luogu.com.cn/problem/P1232

先定序，这个喜欢用哪种确定都可以；然后判断哪个点必须分，哪个点不可能分，哪个点可分可不分（这种无论分不分都对后续状态无影响），分别加1、0、0.5即可；最后输出即可。

```
n = int(input())
sd, sb = [0 for _ in range(n)], [0 for _ in range(n)]
dfn = list(map(int, input().split()))
bfn = list(map(int, input().split()))
for i in range(n):
    sd[dfn[i]-1] = i
    sb[bfn[i]-1] = i
for j in range(n):
    dfn[j] = sb[dfn[j]-1]
    bfn[j] = sd[bfn[j]-1]
d = [0 for _ in range(n)]
d[0], d[1] = 1, -1
ans = 2
for i in range(n-1):
    if bfn[i] > bfn[i+1]:
        d[i] += 1
        d[i+1] -= 1
        ans += 1
for i in range(n-1):
    if dfn[i]+1 < dfn[i+1]:
        d[dfn[i]] += 1
        d[dfn[i+1]] -= 1
w = 0
for i in range(n-1):
    w += d[i]
    if not w:
        ans += 0.5
print(f'{ans:.3f}')
```

4、Artificial Idiot

http://cs101.openjudge.cn/practice/28052/

这道题就是使用heap+记忆化搜索即可，主要感觉题目情景以及描述比较新颖。同时数据比较逆天.....

```
import heapq

dir = [(-1, -1), (-1, 0), (0, -1), (0, 1), (1, 0), (1, 1)]
n = int(input())
tmp = [list(map(int, input().split())) for _ in range(n)]
```

```
memo = [[float('inf') for _ in range(n)] for _ in range(n)]

a, b = sum(tmp[i].count(1) for i in range(n)), sum(tmp[j].count(2) for j in
range(n))
matrix = [[0 for _ in range(n)] for _ in range(n)]
if a == b:
    for i in range(n):
        for j in range(n):
            if tmp[i][j] == 1:
                matrix[j][i] = 1
            elif tmp[i][j] == 2:
                matrix[j][i] = -1
elif a > b:
    for i in range(n):
        for j in range(n):
            if tmp[i][j] == 1:
                matrix[i][j] = -1
            elif tmp[i][j] == 2:
                matrix[i][j] = 1
else:
    print(-1)
    exit()

queue = []
for i in range(n):
    if matrix[i][0] == 1:
        heapq.heappush(queue, (0, i, 0))
        memo[i][0] = 0
    elif matrix[i][0] == 0:
        heapq.heappush(queue, (1, i, 0))
        memo[i][0] = 1
while queue:
    t, x, y = heapq.heappop(queue)
    if y == n-1:
        print(t)
        exit()
    for a, b in dir:
        nx, ny = x+a, y+b
        if 0 <= nx < n and 0 <= ny < n:
            if matrix[nx][ny] == -1:
                continue
            elif matrix[nx][ny] == 1:
                if t < memo[nx][ny]:
                    memo[nx][ny] = t
                    heapq.heappush(queue, (t, nx, ny))
            else:
                if t+1 < memo[nx][ny]:
                    memo[nx][ny] = t+1
                    heapq.heappush(queue, (t+1, nx, ny))
print(-1)
```

5、最后的迷宫

https://www.luogu.com.cn/problem/P2199

这道题耗了好多时间……本来以为是个挺普通的bfs，结果反复TLE、MLE，只能说这道题的数据确实会出现比较极端的情况，所以也可以比较好地要求对于存储结构的利用，以及对数据的预先处理。看题解，好像可以压成一维处理，应该是行的，降低内存，但是脑子笨就直接写二维了（也就导致了如果稍微哪里写的过头就会超时或者超内存……）。

```
from collections import deque

dir = [(-1, 0), (0, -1), (0, 1), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]
n, m = map(int, input().split())
maze = [list(input()) for _ in range(n)]
vis = [[0]*m for _ in range(n)]

def bfs(x0, y0):
    queue = deque()
    queue.append((x0, y0))
    if vis[x0][y0] == -1:
        return 0
    vis[x0][y0] = 1
    while queue:
        x, y = queue.popleft()
        for k in range(4):
            nx, ny = x+dir[k][0], y+dir[k][1]
            if 0 <= nx < n and 0 <= ny < m and maze[nx][ny] == 'O':
                if vis[nx][ny] == -1:
                    return vis[x][y]
                if vis[nx][ny] == 0:
                    queue.append((nx, ny))
                    vis[nx][ny] = vis[x][y]+1
    return 'Poor Harry'

def change(p):
    p.clear()
    for _ in range(n):
        p.append([0]*m)

def mark(p, q):
    for a, b in dir:
        r, c = p, q
        while 0 <= r < n and 0 <= c < m and maze[r][c] == 'O':
            vis[r][c] = -1
            r += a
            c += b

while True:
    d, j, w, v = map(int, input().split())
    if d == j == w == v == 0:
        break
    change(vis)
    mark(d-1, j-1)
    print(bfs(w-1, v-1))
```

6、孤舟蓑笠翁

https://www.luogu.com.cn/problem/P4730

II. 深度优先搜索

1、正方形破坏者

http://cs101.openjudge.cn/practice/01084/

使用估价函数剪枝的一道题目，建议去看这篇博客：https://blog.csdn.net/2301_79402523/article/details/137194237，也是我写的。

```
import copy
import sys
sys.setrecursionlimit(1 << 30)
found = False

def check1(x, tmp):
    for y in graph[x]:
        if tmp[y]:
            return False
    return True

def check2(x):
    for y in graph[x]:
        if judge[y]:
            return False
    return True

def estimate():
    cnt = 0
    tmp = copy.deepcopy(judge)
    for x in range(1, total+1):
        if check1(x, tmp):
            cnt += 1
            for u in graph[x]:
                tmp[u] = True
    return cnt

def dfs(t):
    global found
    if t + estimate() > limit:
        return
    for x in range(1, total+1):
        if check2(x):
            for y in graph[x]:
                judge[y] = True
                dfs(t+1)
                judge[y] = False
                if found:
                    return
            return
```

```
    found = True

for _ in range(int(input())):
    n = int(input())
    lst = list(map(int, input().split()))
    d, m, nums, total = 2*n+1, lst[0], lst[1:], 0
    graph = {}
    for i in range(n):
        for j in range(n):
            for k in range(1, n+1):
                if i+k <= n and j+k <= n:
                    total += 1
                    graph[total] = []
                    for p in range(1, k+1):
                        graph[total] += [d*i+j+p, d*(i+p)+j-n, d*(i+p)+j-n+k, d*
(i+k)+j+p]
    judge = [False for _ in range(2*n*(n+1)+1)]
    for num in nums:
        judge[num] = True
    limit = estimate()
    found = False
    while True:
        dfs(0)
        if found:
            print(limit)
            break
        limit += 1
```

2、骑士周游

http://cs101.openjudge.cn/practice/28050/

这道题我们姑且不考虑一些奇技淫巧，正常用dfs做的话，是会超时的，所以我们需要使用Warnsdorff算法，每一次选择的时候都选择可行子节点最少的子节点，从而达到最快得到答案的目的。

```
import sys
sys.setrecursionlimit(1 << 30)

dir = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)]
n = int(input())
s, e = map(int, input().split())
edge = {}
vis = {}
degree = {}
for i in range(n):
    for j in range(n):
        vis[(i, j)] = False
        edge[(i, j)] = []
        for a, b in dir:
            ni, nj = i+a, j+b
            if 0 <= ni < n and 0 <= nj < n:
                edge[(i, j)].append((ni, nj))
        degree[(i, j)] = len(edge[(i, j)])
```

```
def dfs(x, y):
    if all(vis[key] for key in vis):
        print('success')
        exit()
    if degree[(x, y)] <= 0:
        return
    for p, q in dir:
        nx, ny = x+p, y+q
        if 0 <= nx < n and 0 <= ny < n and not vis[(nx, ny)]:
            degree[(nx, ny)] -= 1
    edge[(x, y)].sort(key=lambda m: degree[m])
    for r, c in edge[(x, y)]:
        if not vis[(r, c)]:
            vis[(r, c)] = True
            dfs(r, c)
            vis[(r, c)] = False
    for p, q in dir:
        nx, ny = x+p, y+q
        if 0 <= nx < n and 0 <= ny < n and not vis[(nx, ny)]:
            degree[(nx, ny)] += 1

vis[(s, e)] = True
dfs(s, e)
print('fail')
```

3、The Rotation Game

http://cs101.openjudge.cn/practice/02286/

又一道估价函数剪枝的，不得不说这种做法真的可以大大减少时间复杂度。试图总结一下，dfs中的剪枝可以分成以下的类别：

- **最优性剪枝**（发现已经次优于当前最优，那么就可以放弃后续搜索）
- **可行性剪枝**（判断是否违反了问题的约束条件）
- **启发式剪枝**（提前预测哪些分支很可能不会产生最优解，从而加速搜索）
- **对称性剪枝**（利用对称性剪枝）
- **记忆化剪枝**（记录先前搜索过的状态进行判重）

```
line = {'A': [0, 2, 6, 11, 15, 20, 22],
        'B': [1, 3, 8, 12, 17, 21, 23],
        'C': [10, 9, 8, 7, 6, 5, 4],
        'D': [19, 18, 17, 16, 15, 14, 13],
        'E': [23, 21, 17, 12, 8, 3, 1],
        'F': [22, 20, 15, 11, 6, 2, 0],
        'G': [13, 14, 15, 16, 17, 18, 19],
        'H': [4, 5, 6, 7, 8, 9, 10]
        }
center = [6, 7, 8, 11, 12, 15, 16, 17]

def check():
```

```
    for i in range(8):
        if mp[center[i]] != mp[center[0]]:
            return False
    return True

def move(r):
    tmp = [mp[line[r][i]] for i in range(7)]
    for j in range(7):
        mp[line[r][j-1]] = tmp[j]

def move_back(c):
    tmp = [mp[line[c][i]] for i in range(7)]
    for j in range(-1, 6):
        mp[line[c][j+1]] = tmp[j]

def diff(t):
    cnt = 0
    for i in range(8):
        if mp[center[i]] != t:
            cnt += 1
    return cnt

def h():
    return min(diff(1), diff(2), diff(3))

def dfs(dep, max_d):
    if check():
        print(''.join(ans))
        return True
    if dep+h() > max_d:
        return False
    for letter in 'ABCDEFGH':
        ans.append(letter)
        move(letter)
        if dfs(dep+1, max_d):
            return True
        ans.pop()
        move_back(letter)
    return False

while True:
    mp = list(map(int, input().split()))
    if mp == [0]:
        break
    ans = []
    if check():
        print('No moves needed')
    else:
        limit = 1
        while True:
            if dfs(0, limit):
                break
            limit += 1
    print(mp[6])
```

4、魔法指纹

https://www.luogu.com.cn/problem/P1822

这是一道挺好的dfs题目，很好地考察了与队列的结合。思路是很简单的，从7开始不断扩展并进行判断即可。

```
from collections import deque
a = int(input())
b = int(input())
queue = deque([7])
ans = 1 if a <= 7 <= b else 0

def dfs(x, y, t):
    global ans
    if y > b:
        return
    if x == 0:
        last = y // (t // 10)
        if not last:
            return
        dfs(x, y+last*t, t*10)
        if a <= y <= b:
            ans += 1
        if t < b:
            queue.append(y)
        return
    last, nxt = y // (t // 10), x % 10
    x //= 10
    if last-nxt >= 0:
        dfs(x, y+(last-nxt)*t, t*10)
    if nxt and last+nxt < 10:
        dfs(x, y+(last+nxt)*t, t*10)

while queue:
    x = queue.popleft()
    for i in range(10):
        dfs(x, i, 10)
print(ans)
```

5、切蛋糕

https://www.luogu.com.cn/problem/P1528

没想到……是我拷贝的时候写多了，导致时间复杂度太大。思路是清晰的，贪心+dfs+二分，主要是剪枝不算好想，不过也还好，倒是真没想到自己会在拷贝这么一个小坑里面栽个大跟头。

```
import sys
sys.setrecursionlimit(1 << 30)

def dfs(x, st, cake):
```

```
    global w
    if x == 0:
        return True
    if total-w < prev[mid]:
        return False
    for j in range(st, n+1):
        if cake[j] >= mouth[x]:
            cake[j] -= mouth[x]
            if cake[j] < mouth[1]:
                w += cake[j]
            if mouth[x] == mouth[x-1]:
                if dfs(x-1, j, cake):
                    return True
            else:
                if dfs(x-1, 1, cake):
                    return True
            if cake[j] < mouth[1]:
                w -= cake[j]
            cake[j] += mouth[x]
    return False

n = int(input())
cake = sorted([0]+[int(input()) for _ in range(n)])
total = sum(cake)
m = int(input())
mouth = sorted([0]+[int(input()) for _ in range(m)])
prev = [0 for _ in range(m+1)]
for i in range(1, m+1):
    prev[i] = prev[i-1]+mouth[i]
left, right = 1, m
while mouth[right] > cake[-1] and right > 0:
    right -= 1
ans = 0
while left <= right:
    w = 0
    mid = (left+right)//2
    if dfs(mid, 1, cake[:]):
        ans = mid
        left = mid+1
    else:
        right = mid-1
print(ans)
```

6、兔兔与蛋蛋游戏

https://www.luogu.com.cn/problem/P1971

四、拓扑排序

1、Sorting It All Out

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

其实就是正常的拓扑排序，只是每次添加新的边时都需要进行判断。

```
def topo_sort(x):
    if vis[x] == -1:
        return -1
    if pos[x] != -1:
        return pos[x]
    vis[x] = -1
    p = n
    for i in range(len(graph[x])):
        p = min(p, topo_sort(graph[x][i]))
        if p == -1:
            return -1
    topo[p-1] = x
    pos[x], vis[x] = p-1, 0
    return p-1

while True:
    n, m = map(int, input().split())
    if n == m == 0:
        break
    graph = {i: [] for i in range(n)}
    lst = [input() for _ in range(m)]
    topo = [0 for _ in range(n)]
    ans = n
    for i in range(m):
        graph[ord(lst[i][0])-ord('A')].append(ord(lst[i][2])-ord('A'))
        vis = [0 for _ in range(n)]
        pos = [-1 for _ in range(n)]
        for j in range(n):
            ans = min(ans, topo_sort(j))
        if ans == -1:
            print(f'Inconsistency found after {i+1} relations.')
            break
        elif ans == 0:
            print(f'Sorted sequence determined after {i+1} '
                  f'relations: {"".join([chr(topo[k]+ord("A")) for k in
range(n)])}.')
            break
    if ans > 0:
        print('Sorted sequence cannot be determined.')
```

2、神经网络

https://www.luogu.com.cn/problem/P1038

使用拓扑排序，最后进行节点的判断并输出即可。

```
from collections import deque
n, p = map(int, input().split())
nerve = {i: 0 for i in range(1, n+1)}
queue = deque()
for k in range(1, n+1):
    c, u = map(int, input().split())
    nerve[k] = c
    if c == 0:
        nerve[k] -= u
    else:
        queue.append(k)
in_degree = {i: 0 for i in range(1, n+1)}
out_degree = {i: 0 for i in range(1, n+1)}
W = {i: {} for i in range(1, n+1)}
for _ in range(p):
    i, j, w = map(int, input().split())
    W[i][j] = w
    in_degree[j] += 1
    out_degree[i] += 1
while queue:
    node = queue.popleft()
    for j in W[node]:
        nerve[j] += W[node][j]*nerve[node]
        in_degree[j] -= 1
        if in_degree[j] == 0 and nerve[j] > 0:
            queue.append(j)
flag = True
for q in range(1, n+1):
    if out_degree[q] == 0:
        if nerve[q] > 0:
            flag = False
            print(q, nerve[q])
if flag:
    print('NULL')
```

3、Elaxia的路线

https://www.luogu.com.cn/problem/P2149

这道题很不错，首先需要跑几次dijkstra（两对点，也就是四个点，分别出发跑一次）来获取dis数组，随后做拓扑排序，得到最短路的DAG，最后使用队列做判断，找到最长链即可。总体来说，考察的内容很丰富，对于时间和内存的限制没有那么死但也没有很宽松，可以很好地考察对于图论相关算法知识以及对于数据结构的理解应用。

```
from collections import deque
import heapq
```

```python
import sys
input = sys.stdin.readline
def dijkstra(x, k):
    dis[k][x] = 0
    queue = []
    heapq.heappush(queue, (0, x))
    while queue:
        t, p = heapq.heappop(queue)
        if t != dis[k][p]:
            continue
        for q in graph[p]:
            if dis[k][p]+graph[p][q] < dis[k][q]:
                dis[k][q] = dis[k][p]+graph[p][q]
                heapq.heappush(queue, (dis[k][p]+graph[p][q], q))

n, m = map(int, input().split())
x1, y1, x2, y2 = map(int, input().split())
graph = {i: {} for i in range(1, n+1)}
for _ in range(m):
    u, v, w = map(int, input().split())
    graph[u][v] = graph[v][u] = w
dis = [[float('inf') for _ in range(n+1)] for _ in range(4)]
dijkstra(x1, 0)
dijkstra(y1, 1)
dijkstra(x2, 2)
dijkstra(y2, 3)

in_degree = [0 for _ in range(n+1)]
for u in range(1, n+1):
    for v in list(graph[u]):
        if dis[0][u]+graph[u][v]+dis[1][v] == dis[0][y1]:
            in_degree[v] += 1
        else:
            del graph[u][v]

ans = 0
f, g = [0 for _ in range(n+1)], [0 for _ in range(n+1)]
queue = deque()
queue.append(x1)
while queue:
    u = queue.popleft()
    ans = max(ans, f[u], g[u])
    for v in graph[u]:
        in_degree[v] -= 1
        if dis[2][u]+graph[u][v]+dis[3][v] == dis[2][y2]:
            f[v] = max(f[v], f[u]+graph[u][v])
        if dis[3][u]+graph[u][v]+dis[2][v] == dis[2][y2]:
            g[v] = max(g[v], g[u]+graph[u][v])
        if in_degree[v] == 0:
            queue.append(v)
print(ans)
```

4、道路

https://www.luogu.com.cn/problem/P2505

这道题……应该是用python没有办法通过它的全部测试点了（C++跑都需要六七百毫秒），这里跑n次dijkstra，每次跑完之后取拓扑序然后反着来统计两个数组（一个记录起点到某一点的最短路数，一个记录终点到某一点的最短路数），二者相乘再累加即可。

```
import heapq, sys
input = sys.stdin.readline

n, m = map(int, input().split())
G = [[] for _ in range(n+1)]
for i in range(m):
    x, y, z = map(int, input().split())
    G[x].append([y, z, i])
ans, cnt = [0]*m, [0]*(n+1)
mod = 10**9+7

def dijkstra(x0):
    tmp, dis = [0]*(n+1), [float('inf')]*(n+1)
    queue = []
    tmp[x0] = 1
    dis[x0] = 0
    heapq.heappush(queue, (0, x0))
    dag = []
    while queue:
        d, u = heapq.heappop(queue)
        if d != dis[u]:
            continue
        dag.append(u)
        for p in G[u]:
            v, w, idx = p
            if dis[u]+w < dis[v]:
                dis[v] = dis[u]+w
                tmp[v] = tmp[u]
                heapq.heappush(queue, (dis[v], v))
            elif dis[u]+w == dis[v]:
                tmp[v] += tmp[u]
    dag.reverse()
    for u in dag:
        cnt[u] = 1
        for f in G[u]:
            v, w, idx = f
            if dis[u]+w == dis[v]:
                cnt[u] = (cnt[u]+cnt[v]) % mod
                ans[idx] = (ans[idx]+tmp[u]*cnt[v]%mod) % mod

for i in range(1, n+1):
    dijkstra(i)

for j in range(m):
    print(ans[j])
```

5、摄像头

https://www.luogu.com.cn/problem/P2712

一道很水的拓扑排序，算是拿去练练手感吧。数据范围要是写错会RE，同时需要注意摄像头不是连续的。

```
from collections import deque
graph = {i: [] for i in range(1, 1001)}
n = int(input())
now = []
in_degree = {i: 0 for i in range(1, 1001)}
for _ in range(n):
    tmp = list(map(int, input().split()))
    now.append(tmp[0])
    for p in tmp[2:]:
        graph[tmp[0]].append(p)
        in_degree[p] += 1
queue = deque()
for w in now:
    if in_degree[w] == 0:
        queue.append(w)
ans = n
while queue:
    u = queue.popleft()
    ans -= 1
    for v in graph[u]:
        in_degree[v] -= 1
        if in_degree[v] == 0 and v in now:
            queue.append(v)
if ans == 0:
    print('YES')
else:
    print(ans)
```

五、生成树

1、买礼物

https://www.luogu.com.cn/problem/P1194

这里使用了Kruskal算法，简单说就是我们找边，只要找到n-1条最小边，同时每次向生成树添加边的时候判断会不会出现连接之前已经连起来的节点（判断的时候使用并查集）。

```
def find(x):
    if p[x] != x:
        p[x] = find(p[x])
    return p[x]
```

```
def union(x, y):
    fx = find(x)
    fy = find(y)
    if fx != fy:
        p[fx] = fy
        return True
    return False

a, b = map(int, input().split())
edges = [(0, i, a) for i in range(1, b)]
for i in range(b):
    tmp = list(map(int, input().split()))
    for j in range(i):
        if tmp[j] != 0:
            edges.append((i, j, tmp[j]))
p = [i for i in range(b)]
edges.sort(key=lambda x: x[2])
ans = a
for u, v, w in edges:
    if union(u, v):
        ans += w
print(ans)
```

2、Arctic Network

http://dsbpython.openjudge.cn/dspythonbook/P1240/

同样是Kruskal算法，这里不需要连全部边，p-s条即可。类似的题目，洛谷也有一道：口袋的天空，https://www.luogu.com.cn/problem/P1195。

```
import math

def cal(a, b):
    return math.sqrt(pow(a[0]-b[0], 2)+pow(a[1]-b[1], 2))

def find(x):
    if f[x] != x:
        f[x] = find(f[x])
    return f[x]

def union(x, y):
    fx = find(x)
    fy = find(y)
    if fx != fy:
        f[fx] = fy
        return True
    return False

for _ in range(int(input())):
    s, p = map(int, input().split())
    outposts = [list(map(int, input().split())) for _ in range(p)]
    f = [i for i in range(p)]
    edges = []
```

```
    for j in range(p):
        for i in range(j):
            edges.append((i, j, cal(outposts[i], outposts[j])))
    edges.sort(key=lambda x: x[2])
    ans, cnt = 0, 0
    for u, v, w in edges:
        if union(u, v):
            ans = max(ans, w)
            cnt += 1
        if cnt >= p-s:
            break
    print(f'{ans:.2f}')
```

3、Cheering up the Cow G

https://www.luogu.com.cn/problem/P2916

这道题还是使用Kruskal算法（Prim在OJ上面大家做了很多，而在洛谷上面找到的比较合适的题目用python又都过不了），主要对于权值的处理比较有趣。

```
def find(x):
    if f[x] != x:
        f[x] = find(f[x])
    return f[x]

def union(x, y):
    fx, fy = find(x), find(y)
    if fx != fy:
        f[fx] = fy
        return True
    return False

n, p = map(int, input().split())
cows = [0]+[int(input()) for _ in range(n)]
f = [i for i in range(n+1)]
edges = []
for _ in range(p):
    u, v, w = map(int, input().split())
    edges.append((u, v, cows[u]+cows[v]+2*w))
edges.sort(key=lambda x: x[2])
ans = 0
for u, v, w in edges:
    if union(u, v):
        ans += w
print(ans+min(cows[1:]))
```

4、瞎破坏

http://dsbpython.openjudge.cn/dspythonbook/P1290/

这道题最开始没理解意思……首先我们先想想，最小生成树有几条边？而有环的图又至少有几条边，由此我们就发现，我们只需要去掉一颗最小生成树就可以了。

```
def find(x):
    if f[x] != x:
        f[x] = find(f[x])
    return f[x]

def union(x, y):
    fx, fy = find(x), find(y)
    if fx != fy:
        f[fx] = fy
        return True
    return False

n, m = map(int, input().split())
f = [i for i in range(n)]
edges = []
ans = 0
for _ in range(m):
    x, y, v = map(int, input().split())
    edges.append((x, y, v))
    ans += v
edges.sort(key=lambda x: x[2])
cnt = 0
for u, v, w in edges:
    if union(u, v):
        ans -= w
        cnt += 1
    if cnt == n-1:
        break
print(ans)
```

5、小店购物

https://www.luogu.com.cn/problem/P2792

omg……这道题卡在手头卡了两三天，终于AC了。首先我们看题面就注意到，这并不是最小生成树，准确说不是无向图的最小生成树，而应该是有向图的最小树形图（和最小生成树一个意思，不过由于有向，所以叫做树形图）。这样的问题有两种算法，一个是朱刘算法，另一个是Tarjan的DMST算法，后者对我来说太难了😭。这里介绍一下朱刘算法。

其实朱刘算法的想法很简单，就是找到n-1条最小边组成树，问题在对于环怎么处理。实际上我们把环看成一个单元进行处理（做个比喻就是当成黑盒子），就可以了，这也是朱刘算法的精髓。oiwiki上是进行这样的总结的——

1、对于每个点，选择指向它的边权最小的那条边。

2、如果没有环，算法终止；否则进行缩环并更新其他点到环的距离。

emmm感觉用文字可能说的不太清楚，这里放一道模板题以及代码+注释叭~

P4716 【模板】最小树形图(https://www.luogu.com.cn/problem/P4716)

```
class Edge:
    def __init__(self, u, v, w):
        self.u = u
        self.v = v
        self.w = w

def zhu_liu(n, root):
    ans = 0
    while True:
        pre, vis, in_vector = [-1]*n, [-1]*n, [float('inf')]*n
        # 寻找并记录每个点的最小入边
        in_vector[root] = 0
        for i in range(m):
            if edges[i].u != edges[i].v and edges[i].w < in_vector[edges[i].v]:
                pre[edges[i].v] = edges[i].u
                in_vector[edges[i].v] = edges[i].w
        # 有孤立点，不存在最小树形图
        for i in range(n):
            if i != root and in_vector[i] == float('inf'):
                return -1
        # 找有向环
        loop = 0  # 记录环的个数，用来在之后缩环成点提供编号
        circle = [-1]*n
        for i in range(n):
            ans += in_vector[i]
            v = i
            # 向前遍历环，中止情况如下：
            # 1、出现带有相同标记的点，成环（这个情况是我们之后需要做处理的）
            # 2、节点属于其他环，说明进入了其他环
            # 3、遍历到根节点了
            while vis[v] != i and circle[v] == -1 and v != root:
                vis[v] = i
                v = pre[v]
            # 成环才能进入while循环，把环内的点用circle进行标记（同样是为之后的缩环成点做准
备）
            if v != root and circle[v] == -1:
                while circle[v] != loop:
                    circle[v] = loop
                    v = pre[v]
                loop += 1
        # 如果缩到没有环了，那么就找全了
        if loop == 0:
            break
        # 否则将所有孤立点也当作自环看待
        for i in range(n):
            if circle[i] == -1:
                circle[i] = loop
                loop += 1
        # 统一地进行缩环
        for j in range(m):
            v = edges[j].v
            edges[j].u = circle[edges[j].u]
```

```
            edges[j].v = circle[edges[j].v]
            # 如果边不属于同一个环，则进行更新
            if edges[j].u != edges[j].v:
                edges[j].w -= in_vector[v]
        n = loop
        root = circle[root]
    return ans

n, m, r = map(int, input().split())
edges = []
for i in range(m):
    u, v, w = map(int, input().split())
    edges.append(Edge(u-1, v-1, w))
print(zhu_liu(n, r-1))
```

然后我们来看这道题，首先读题发现，应该要进行编号的重整，不买的商品不加入图中，然后我们不难想到，只要添加一个超级源点，就可以跑朱刘算法了，所以用原价与源点相连，接着贪心地想到，对于需要买n (n>1)个的商品，我们可以先当成1个来买，最后加上(n-1)乘以其最低价格就可以。思路想清楚之后套朱刘算法的模板就能够轻松AC~

```
class Edge:
    def __init__(self, u, v, w):
        self.u = u
        self.v = v
        self.w = w

def zhu_liu(n, m, root, edges):
    ans = 0
    while True:
        pre, vis, in_vector = [-1]*n, [-1]*n, [float('inf')]*n
        in_vector[root] = 0
        for i in range(m):
            if edges[i].u != edges[i].v and edges[i].w < in_vector[edges[i].v]:
                pre[edges[i].v] = edges[i].u
                in_vector[edges[i].v] = edges[i].w
        for i in range(n):
            if i != root and in_vector[i] == float('inf'):
                return -1
        loop = 0
        circle = [-1]*n
        for i in range(n):
            ans += in_vector[i]
            v = i
            while vis[v] != i and circle[v] == -1 and v != root:
                vis[v] = i
                v = pre[v]
            if v != root and circle[v] == -1:
                while circle[v] != loop:
                    circle[v] = loop
                    v = pre[v]
                loop += 1
        if loop == 0:
```

```
            break
        for i in range(n):
            if circle[i] == -1:
                circle[i] = loop
                loop += 1
        for j in range(m):
            v = edges[j].v
            edges[j].u = circle[edges[j].u]
            edges[j].v = circle[edges[j].v]
            if edges[j].u != edges[j].v:
                edges[j].w -= in_vector[v]
        n = loop
        root = circle[root]
    return ans

n = int(input())
P, nums, edges = [], [], []
my_dict = {}
for i in range(1, n+1):
    tmp = input().split()
    c, q = float(tmp[0]), int(tmp[1])
    if q:
        P.append(c)
        nums.append(q)
        my_dict[i] = len(P)
        edges.append(Edge(0, len(P), c))
m = int(input())
for _ in range(m):
    tmp = input().split()
    u, v, w = int(tmp[0]), int(tmp[1]), float(tmp[2])
    if u in my_dict and v in my_dict:
        edges.append(Edge(my_dict[u], my_dict[v], w))
        P[my_dict[v]-1] = min(P[my_dict[v]-1], w)
result = zhu_liu(len(P)+1, len(edges), 0, edges)
for i in range(len(P)):
    if nums[i] > 1:
        result += (nums[i]-1)*P[i]
print(f'{result:.2f}')
```