字典的用法：

```
一.排序
# 1. 按 key 升序排序
sorted_items = sorted(my_dict.items())

# 2. 按 key 降序排序
sorted_items = sorted(my_dict.items(), reverse=True)

# 3. 按 value 升序排序
sorted_items = sorted(my_dict.items(), key=lambda x: x[1])

# 4. 按 value 降序排序
sorted_items = sorted(my_dict.items(), key=lambda x: x[1], reverse=True)

# 5. 按 value 的第一个元素排序（value 是元组 / 列表）
sorted_items = sorted(my_dict.items(), key=lambda x: x[1][0])  # 升序
sorted_items = sorted(my_dict.items(), key=lambda x: x[1][0], reverse=True)  # 降序

# 6. 多条件排序（先按 value[0] 降序，再按 value[1] 升序）
sorted_items = sorted(my_dict.items(), key=lambda x: (-x[1][0], x[1][1]))

# 7. 取排序后的前 k 项
top_k = sorted(my_dict.items(), key=lambda x: x[1], reverse=True)[:k]

# 8. 排序后转换为 dict
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1]))

# 9. 使用 operator.itemgetter 简化按 value 排序
from operator import itemgetter
sorted_items = sorted(my_dict.items(), key=itemgetter(1))  # 等价于 lambda x: x[1]

# 10. 按 value 中的字段排序（value 是字典）
my_dict = {
    'A': {'score': 90, 'age': 18},
    'B': {'score': 85, 'age': 20}
}
sorted_items = sorted(my_dict.items(), key=lambda x: x[1]['score'], reverse=True)

# 11. 按嵌套 list/tuple 中的多个值排序
my_dict = {
    'x': (3, 5),
    'y': (2, 9),
    'z': (3, 1)
}
sorted_items = sorted(my_dict.items(), key=lambda x: (x[1][0], -x[1][1]))

二.其他
# 字典操作模板（分类详注，适用于统计、查找、排序、最值）

# 【1】统计频率：用 dict 或 collections.defaultdict
s = input().strip()  # 输入字符串
count = {}  # 或用 from collections import defaultdict; count = defaultdict(int)
for ch in s:
    count[ch] = count.get(ch, 0) + 1
```

```
# 【2】判断是否包含某键（如字符'a'）
if 'a' in count:
    print("'a' exists")

# 【3】遍历键（key）或键值对（key-value）
for key in count:
    print(f"{key} => {count[key]}")

for key, value in count.items():
    print(f"{key} appears {value} times")

# 【4】查找第一个只出现一次的字符
for ch in s:
    if count[ch] == 1:
        print("First unique character:", ch)
        break
else:
    print("no")

# 【5】按频率排序（从高到低）
sorted_items = sorted(count.items(), key=lambda x: -x[1])
for k, v in sorted_items:
    print(f"Sorted: {k} => {v}")

# 【6】找出出现次数最多的字符及其次数
max_key = max(count, key=lambda k: count[k])  # 出现次数最多的字符
print("Max freq:", max_key, "=>", count[max_key])

# 【7】找出出现次数最少的字符及其次数
min_key = min(count, key=lambda k: count[k])  # 出现次数最少的字符
print("Min freq:", min_key, "=>", count[min_key])

# 【8】找出最大值（频率）本身，而不是字符
max_freq = max(count.values())
min_freq = min(count.values())
print("Max freq value:", max_freq)
print("Min freq value:", min_freq)

# 【9】找出所有频率为 max_freq 的字符
most_common_chars = [k for k, v in count.items() if v == max_freq]
print("Most common chars:", most_common_chars)


#【10】找出键最小的 key
min_key = min(d.keys())
print("最小键:", min_key)

# 对应的值
print("对应值:", d[min_key])

【11】访问：
# 常规方式访问键值
val = d['b']       # 如果不存在会报错

# 安全访问（推荐）：使用 get() 方法
val = d.get('b')             # 存在返回值，不存在返回 None
val = d.get('z', 0)          # 不存在时返回默认值 0
```

```
【12】遍历
# 遍历键
for k in d:
    print(k)

# 遍历值
for v in d.values():
    print(v)

# 遍历键值对
for k, v in d.items():
    print(f"{k} -> {v}")
```

枚举和二分法：

1.河中跳房子：

```
L, n, m = map(int, input().split())
rock = [0]
for i in range(n):
    rock.append(int(input()))
rock.append(L)

def check(x):
    num = 0
    now = 0
    for i in range(1, n + 2):
        if rock[i] - now < x:
            num += 1
        else:
            now = rock[i]
    if num > m:
        return True
    else:
        return False

lo, hi = 0, L + 1
ans = -1
while lo < hi:
    mid = (lo + hi) // 2
    if check(mid):
        hi = mid
    else:
        ans = mid
        lo = mid + 1
print(ans)
```

递归和分治

1.放苹果（把M个同样的苹果放在N个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？（用K表示）5，1，1和1，5，1 是同一种分法。）：

```
def ways(m, n):
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for j in range(n + 1):
        dp[0][j] = 1  # 0个苹果，只有一种分法（全空）
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if i < j:
                dp[i][j] = dp[i][i]  # 盘子多于苹果，多的盘子空着即可
            else:
                dp[i][j] = dp[i - j][j] + dp[i][j - 1]
                # 1. 每个盘子至少一个：dp[i - j][j]
                # 2. 至少一个盘子空：dp[i][j - 1]
    return dp[m][n]

t = int(input())
for _ in range(t):
    m, n = map(int, input().split())
    print(ways(m, n))
```

2.七的倍数有多少种

```
def count_divisible_by_seven(lst):
    count = 0

    def dfs(index, current_sum):
        nonlocal count
        if index == len(lst):
            if current_sum % 7 == 0:
                count += 1
            return
        # 包含当前元素
        dfs(index + 1, current_sum + lst[index])
        # 不包含当前元素
        dfs(index + 1, current_sum)

    dfs(0, 0)
    return count

# 主程序
n = int(input())
for _ in range(n):
    s = list(map(int, input().split()))
    a, nums = s[0], s[1:]
    print(count_divisible_by_seven(nums))
```

3.整数划分

```
n = int(input().strip())  # 输入一个正整数 n，要求划分成若干正整数之和

# 创建一个二维列表 board[m][k]，表示将整数 m 划分成最大加数不超过 k 的方案数
```

```
board = [[0] * (n + 1) for _ in range(n + 1)]

# 初始条件：1只能划分为1，只有一种方法
board[1][1] = 1

# 初始化：任意整数 m 划分为最大加数为 1 的情况，只有一种 —— 全部是 1
for i in range(2, n + 1):
    board[i][1] = 1  # 例如：4 = 1+1+1+1

# 动态规划填表
for m in range(2, n + 1):       # 当前整数 m，从 2 到 n
    for k in range(2, n + 1):   # 当前允许的最大加数 k，从 2 到 n
        if m - k >= 0:
            # 两种情况的加和：
            # 1. 至少使用一个 k（即 board[m-k][k]）
            # 2. 不使用 k（即最大加数为 k-1，board[m][k-1]）
            board[m][k] = board[m - k][k] + board[m][k - 1]
        else:
            # k 太大，不能用，只能考虑最大加数为 k-1
            board[m][k] = board[m][k - 1]

# 结果：将 n 划分成任意加数（1~n）的方案总数
ans = board[n][n]
print(ans)
```

栈

1.中序转后序表达式

```
def precedence(op):
    if op in ('*', '/'):
        return 2
    if op in ('+', '-'):
        return 1
    return 0
def infix_to_postfix(expr):
    output = []
    stack = []
    i = 0
    while i < len(expr):
        ch = expr[i]
        if ch.isdigit() or ch == '.':
            num = []
            while i < len(expr) and (expr[i].isdigit() or expr[i] == '.'):
                num.append(expr[i])
                i += 1
            output.append(''.join(num))
            continue
        elif ch == '(':
            stack.append(ch)
        elif ch == ')':
            while stack and stack[-1] != '(':
```

```
                output.append(stack.pop())
            stack.pop()
        elif ch in '+-*/':
            while (stack and stack[-1] != '(' and
                   precedence(stack[-1]) >= precedence(ch)):
                output.append(stack.pop())
            stack.append(ch)
        i += 1

    while stack:
        output.append(stack.pop())

    return ' '.join(output)

n=int(input())
for i in range(n):
    s=input()
    result=infix_to_postfix(s)
    print(result)
```

2.波兰表达式（前序表达式求值）

——前序后序表达式都是数字入栈，前序逆序扫描（右到左），后序顺序扫描（左到右）

```
sign = {'+', '-', '*', '/'}

def cal(a, b, char):
    a, b = float(a), float(b)
    if char == '+':
        return a + b
    elif char == '-':
        return a - b
    elif char == '*':
        return a * b
    else:
        return a / b

def polish(s):
    stack = []
    for token in reversed(s):  # 逆序遍历波兰表达式
        if token in sign:
            a = stack.pop()
            b = stack.pop()
            stack.append(cal(a, b, token))
        else:
            stack.append(token)  # 数字直接入栈
    print(f'{stack[0]:.6f}')  # 保持 6 位小数输出

# 读取输入并拆分
s = input().split()
polish(s)
```

3.双端队列（有deque的用法）

- 输入

  第一行输入一个整数t，代表测试数据的组数。 每组数据的第一行输入一个整数n，表示操作的次数。 接着输入n行，每行对应一个操作，首先输入一个整数type。 当type=1，进队操作，接着输入一个整数x，表示进入队列的元素。 当type=2，出队操作，接着输入一个整数c，c=0代表从队头出队，c=1代表从队尾出队。 n <= 1000

- 输出

  对于每组测试数据，输出执行完所有的操作后队列中剩余的元素,元素之间用空格隔开，按队头到队尾的顺序输出，占一行。如果队列中已经没有任何的元素，输出NULL。

```
from collections import deque
t=int(input())
for _ in range(t):
    n=int(input())
    queue=deque()
    for _ in range(n):
        a,b=map(int,input().split())
        if a==1:
            queue.append(b)
        elif a==2:
            if queue:
                if b==1:
                    queue.pop()
                elif b==0:
                    queue.popleft()
            else:
                pass
    if queue:
        print(' '.join(map(str,queue)))  #注意不要直接str(queue)会返回不期望的结果
    else:
        print('NULL')
```

4.发型糟糕的一天（P0550）（书p79）

```
n = int(input())  # 奶牛数量
l = []  # 存奶牛高度列表
for i in range(n):
    l.append(int(input()))

up = [(1e10, n)]  # 单调递增栈（右边看过来的“障碍物”），(高度, 索引)，初始加一个极高障碍，确保永远有终止条件
ans = 0  # 最终能看到的奶牛总数

# 从右向左遍历每头奶牛（从最后一头往前看）
for i in range(n-1, -1, -1):
    # 遍历栈，从栈顶向下找（栈顶是最右边的）
    for k in range(len(up)-1, -1, -1):
        if l[i] > up[k][0]:  # 当前奶牛比栈中那个高：说明可以“看到”它，看完后它就不再是阻碍，pop掉
            up.pop()
        else:
            # 当前奶牛看不到更多了，遇到第一个“不矮于自己”的障碍
            # 所以能看到的个数就是：up[k][1] - i - 1（实际距离），但这里先记总距离
            up.append((l[i], i))  # 把当前奶牛压进栈：成为后面奶牛的“右边障碍”
            ans += (up[k][1] - i)  # 记录可见奶牛数（注意还没减1）
            break
```

```
# 因为每头奶牛默认包括了自己（距离为0），所以最后减掉n
print(ans - n)
```

二叉树

1.文本缩进二叉树：

- 输入

  第一行是树的数目 n 接下来是n棵树，每棵树以'0'结尾。'0'不是树的一部分 每棵树不超过100个节点 （层次用空格表示的，如果其他注意改一改）

- 输出

  对每棵树，分三行先后输出其前序、后序、中序遍历结果 两棵树之间以空行分隔

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

def build_tree(lines):
    root = None
    level_parents = {}  # 保存各层级当前的父节点
    for line in lines:
        if not line.strip():
            continue
        stripped_line = line.lstrip('\t')
        t = len(line) - len(stripped_line)
        val = stripped_line.strip()
        if t == 0:
            root = Node(val)
            level_parents = {0: root}
        else:
            parent_level = t - 1
            if parent_level not in level_parents:
                continue  # 输入错误，但题目保证输入合法
            parent = level_parents[parent_level]
            if not parent.left_processed:
                if val == '*':
                    parent.left = None
                    parent.left_processed = True
                else:
                    node = Node(val)
                    parent.left = node
                    parent.left_processed = True
                    level_parents[t] = node
            else:
                if val == '*':
                    continue  # 右子节点不能为*，题目保证输入合法
                node = Node(val)
                parent.right = node
                parent.right_processed = True
```

```
                level_parents[t] = node
    return root

def preorder(root):
    if not root:
        return ''
    res = root.val
    res += preorder(root.left)
    res += preorder(root.right)
    return res

def inorder(root):
    if not root:
        return ''
    res = inorder(root.left)
    res += root.val
    res += inorder(root.right)
    return res

def postorder(root):
    if not root:
        return ''
    res = postorder(root.left)
    res += postorder(root.right)
    res += root.val
    return res

# 读取输入并处理
import sys
lines = [line.rstrip('\n') for line in sys.stdin]

root = build_tree(lines)

print(preorder(root))
print(inorder(root))
print(postorder(root))
```

打印文本缩进二叉树：

- 输入

  2行。 第1行是中序遍历序列 第2行是后序遍历序列 二叉树不超过100个结点，且结点的字母不会重复。

- 输出

  该二叉树的文本缩进表示形式

```
import sys
sys.setrecursionlimit(10**7)

inorder = input().strip()
postorder = input().strip()

# 建树
def build(in_l, in_r, post_l, post_r):
    if in_l > in_r:
        return None
    root_val = postorder[post_r]
```

```
    root = {'val': root_val, 'left': None, 'right': None}
    root_idx = inorder_index[root_val]
    left_size = root_idx - in_l
    root['left'] = build(in_l, root_idx - 1, post_l, post_l + left_size - 1)
    root['right'] = build(root_idx + 1, in_r, post_l + left_size, post_r - 1)
    return root

def print_tree(node, level):
    if not node:
        return
    # 打印当前节点
    print('\t' * level + node['val'])
    left = node['left']
    right = node['right']
    if left is None and right is not None:
        # 空左子树用*表示
        print('\t' * (level + 1) + '*')
        print_tree(right, level + 1)
    else:
        if left:
            print_tree(left, level + 1)
        if right:
            print_tree(right, level + 1)

# 预处理字母在中序中的索引，便于快速切分
inorder_index = {c: i for i, c in enumerate(inorder)}

root = build(0, len(inorder) - 1, 0, len(postorder) - 1)
print_tree(root, 0)
```

2.二叉树的操作：

给定一棵二叉树，在二叉树上执行两个操作：

1. 节点交换：把二叉树的两个节点交换。
2. 前驱询问：询问二叉树的一个节点对应的子树最左边的节点。

- 输入

  第一行输出一个整数t(t <= 100)，代表测试数据的组数。 对于每组测试数据，第一行输入两个整数n m，n代表二叉树节点的个数，m代表操作的次数。 随后输入n行，每行包含3个整数X Y Z，对应二叉树一个节点的信息。X表示节点的标识，Y表示其左孩子的标识，Z表示其右孩子的标识。 再输入m行，每行对应一次操作。每次操作首先输入一个整数type。 当type=1，节点交换操作，后面跟着输入两个整数x y，表示将标识为x的节点与标识为y的节点交换。输入保证对应的节点不是祖先关系。 当type=2，前驱询问操作，后面跟着输入一个整数x，表示询问标识为x的节点对应子树最左的孩子。 1<=n<=100，节点的标识从0到n-1，根节点始终是0. m<=100

- 输出

  对于每次询问操作，输出相应的结果

```
import sys

class Node:
    def __init__(self, val): #
```

```
        self.val = val
        self.left = None
        self.right = None
        self.parent = None

def solve():
    t = int(sys.stdin.readline())
    for test in range(t):
        n, m = map(int, sys.stdin.readline().split())
        nodes_map = [Node(i) for i in range(n)]

        for line in range(n):
            x, y, z = map(int, sys.stdin.readline().split())

            if y != -1:
                nodes_map[x].left = nodes_map[y]
                nodes_map[y].parent = nodes_map[x]
            if z != -1:
                nodes_map[x].right = nodes_map[z]
                nodes_map[z].parent = nodes_map[x]

        for op in range(m):
            op_parts = list(map(int, sys.stdin.readline().split()))
            op_type = op_parts[0]

            if op_type == 1:  #交换操作
                x, y = op_parts[1], op_parts[2]

                node_x = nodes_map[x]
                node_y = nodes_map[y]

                parent_x = node_x.parent
                parent_y = node_y.parent

                if parent_x.val==parent_y.val:
                    parent_xy = parent_x
                    if parent_xy.left==node_x:
                        parent_xy.left=node_y
                        parent_xy.right=node_x
                    else:
                        parent_xy.right=node_y
                        parent_xy.left=node_x

                else:#两节点不是一个父亲
                    if parent_x.left==node_x:#x是他父亲的左节点
                        parent_x.left = node_y
                        if parent_y.left==node_y:#y是他父亲的左节点
                            parent_y.left=node_x
                        else:#y是他父亲的右节点
                            parent_y.right=node_x

                    else:#x是他父亲的右节点
                        parent_x.right=node_y
                        if parent_y.left==node_y:#y是他父亲的左节点
                            parent_y.left=node_x
                        else:#y是他父亲的右节点
                            parent_y.right=node_x
```

```
                    node_x.parent=parent_y
                    node_y.parent=parent_x

            elif op_type == 2:  #查询
                x = op_parts[1]

                current_node = nodes_map[x]

                while current_node.left:
                    current_node = current_node.left

                sys.stdout.write(str(current_node.val) + "\n")


solve()
```

字符串匹配（KMP算法，书p129）：

1.字符串最大跨距

```
s,s1,s2 = map(str, input().split(','))
if s1 in s and s2 in s:
    tail=0
    head=len(s)
    while s1 not in s[:tail]:
        tail+=1
    while s2 not in s[head-1:]:
        head-=1
    if tail<=head:
        print(head-tail-1)
    else:
        print(-1)
else:
    print(-1)
```

2.找出全部子串位置

```
n=int(input())
input1=[]
result=[]
for i in range(n):
    s1,s2=map(str,input().split())
    current=[]
    if len(s1)>=len(s2):
        j=0
        while j in range(len(s1)-len(s2)+1):
            if s2==s1[j:j+len(s2)]:
                current.append(j)
                j+=len(s2)
            else:
                j+=1
        if current!=[]:
            result.append(current)
        else:
```

```
            result.append(['no'])
    else:
        result.append(['no'])
for i in result:
    for j in range(len(i)):
        print(i[j],end=" " if j<len(i)-1 else "\n")
```

动态规划(其余见计概cheatsheet):

1.最大子矩阵（动态规划 + 一维最大子段和（Kadane算法））

```
import sys
n = int(input())
nums = []
while len(nums) < n * n:
    line = sys.stdin.readline()
    if not line:
        break
    parts = line.strip().split()
    if parts:
        nums.extend(map(int, parts))

matrix = [nums[i*n:(i+1)*n] for i in range(n)]

def max_submatrix_sum(matrix):
    rows, cols = len(matrix), len(matrix[0])
    max_sum = -127 * n * n
    for top in range(rows):
        temp = [0] * cols
        for bottom in range(top, rows):
            for col in range(cols):
                temp[col] += matrix[bottom][col]
            current_sum = max_ending_here = temp[0]
            for x in temp[1:]:
                max_ending_here = max(x, max_ending_here + x)
                current_sum = max(current_sum, max_ending_here)
            max_sum = max(max_sum, current_sum)
    return max_sum

print(max_submatrix_sum(matrix))
```

2.数字三角形（异形）——不存在的用0填充

```
n = int(input())  # 输入数字三角形的层数（行数）

mat = []  # 存储原始三角形的数值
result = [[0]*n for i in range(n)]  # 初始化一个结果矩阵，用于保存每个位置的最大路径和

# 输入三角形的每一行
for i in range(n):
    row = list(map(int, input().split()))  # 读取当前行
```

```
    mat.append(row)  # 加入到三角形中

# 初始化最后一行（最底层）的最大路径和等于原始值
result[-1] = mat[-1]

# 从倒数第二行开始向上计算每个点的最大路径和
for j in range(n-2, -1, -1):  # 从第 n-2 行到第 0 行
    for l in range(j+1):  # 每行的元素个数等于行号 + 1
        # 当前点的最大路径和 = 当前点值 + 它下面两个相邻点中的最大路径和
        result[j][l] = max(result[j+1][l], result[j+1][l+1]) + mat[j][l]

# 输出从顶部到底部的最大路径和（起点是 [0][0]）
print(result[0][0])
```

3.移动办公（双dp类似股民老张）

```
T, M = map(int, input().split())
P = []
N = []
for _ in range(T):
    p, n = map(int, input().split())
    P.append(p)
    N.append(n)

# dp_beijing[i] 表示第 i 个月在北京办公的最大收益
# dp_nanjing[i] 表示第 i 个月在南京办公的最大收益
dp_beijing = [0] * T
dp_nanjing = [0] * T

# 初始化第一个月，可以在北京或南京开始，无交通费
dp_beijing[0] = P[0]
dp_nanjing[0] = N[0]

for i in range(1, T):
    # 如果第 i 个月选择北京办公
    # 可能上个月也是北京（不扣费），也可能上个月南京（扣交通费）
    dp_beijing[i] = max(dp_beijing[i-1] + P[i], dp_nanjing[i-1] + P[i] - M)

    # 如果第 i 个月选择南京办公
    dp_nanjing[i] = max(dp_nanjing[i-1] + N[i], dp_beijing[i-1] + N[i] - M)

# 最大总收益是最后一个月在北京或南京办公的最大值
print(max(dp_beijing[T-1], dp_nanjing[T-1]))
```

4.开餐馆（有条件限制，一定一动法）

```
T = int(input())
for _ in range(T):
    n, k = map(int, input().split())
    m = list(map(int, input().split()))
    p = list(map(int, input().split()))
    dp = p.copy()
    for i in range(n):
        max_prev = 0
```

```
        for j in range(i):
            if m[i] - m[j] > k:
                if dp[j] > max_prev:
                    max_prev = dp[j]
        dp[i] += max_prev
    print(max(dp))
```

5.zipper（字符交错）

```
def can_form_interleaving(A, B, C):
    # 获取字符串长度
    len_a, len_b = len(A), len(B)

    # 如果C的长度不等于A和B长度之和，直接返回False
    if len(C) != len_a + len_b:
        return False

    # 创建二维dp数组，dp[i][j]表示A的前i个字符和B的前j个字符是否能交错组成C的前i+j个字符
    dp = [[False] * (len_b + 1) for _ in range(len_a + 1)]

    # 空字符串和空字符串可以组成空字符串，初始化为True
    dp[0][0] = True

    # 初始化第一列，只有A串能组成C的前缀部分
    for i in range(1, len_a + 1):
        # 前i-1个A字符已匹配，且第i个A字符等于C的对应字符，dp[i][0]为True
        dp[i][0] = dp[i-1][0] and A[i-1] == C[i-1]

    # 初始化第一行，只有B串能组成C的前缀部分
    for j in range(1, len_b + 1):
        # 前j-1个B字符已匹配，且第j个B字符等于C的对应字符，dp[0][j]为True
        dp[0][j] = dp[0][j-1] and B[j-1] == C[j-1]

    # 填充dp表格
    for i in range(1, len_a + 1):
        for j in range(1, len_b + 1):
            # 当前位置C的字符可以来自A的第i个字符且前面状态dp[i-1][j]为True
            # 或者来自B的第j个字符且前面状态dp[i][j-1]为True
            dp[i][j] = (dp[i-1][j] and A[i-1] == C[i+j-1]) or (dp[i][j-1] and B[j-1] == C[i+j-1])

    # 返回A和B能否交错组成C的结果
    return dp[len_a][len_b]

def main():
    n = int(input().strip())  # 输入测试用例数
    for case_num in range(1, n + 1):
        A, B, C = input().strip().split()  # 输入3个字符串
        result = "yes" if can_form_interleaving(A, B, C) else "no"
        print(f"Data set {case_num}: {result}")

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

6.硬币（子集和问题，想法是逆向排除，如果不可行则必须）

方法一：直接单数据较大会超时

```
n, x = map(int, input().split())
coins = list(map(int, input().split()))

# 先用01背包判断能否凑出X
# dp[i]表示是否能凑出金额i
dp = [False] * (x + 1)
dp[0] = True

for c in coins:
    for amount in range(x, c - 1, -1):
        if dp[amount - c]:
            dp[amount] = True

# 如果连X都凑不出来，说明无解
if not dp[x]:
    print(0)
    print()
    exit()

must_use = []
# 判断每个硬币是否必须用
for i in range(n):
    c = coins[i]
    # 构造不使用coins[i]的dp数组
    dp2 = [False] * (x + 1)
    dp2[0] = True
    for j in range(n):
        if i == j:  # 跳过当前硬币
            continue
        coin = coins[j]
        for amount in range(x, coin - 1, -1):
            if dp2[amount - coin]:
                dp2[amount] = True
    # 如果不使用当前硬币不能凑出X，则该硬币必须用
    if not dp2[x]:
        must_use.append(c)

print(len(must_use))
print(*must_use)
```

方法二：（空间换时间，找到所有能凑出金额都必须用的硬币）

```
n,x=map(int,input().split())
coins=list(map(int,input().split()))
achi={0:set()}
todolist=[]
def my_add(mon,coinset):
    if mon>x:
        return
    if mon in achi.keys():
        achi[mon]=achi[mon]&coinset
    else:
        achi[mon]=coinset
for i in coins:
    for keys,values in achi.items():
        todolist.append([keys+i,values|{i}])
    for j in todolist:
        my_add(j[0],j[1])
```

```
    todolist=[]
ans=list(achi[x])
ans.sort()
print(len(ans))
if ans:
    for i in ans:
        print(i,end=' ')
```

7.经典的切割回文

```
def f(s):
    n = len(s)
    # l[i][j] 表示子串 s[i:j+1] 是否是回文串，初始化为False
    l = [[False] * n for i in range(n)]

    # 单个字符都是回文串
    for i in range(n):
        l[i][i] = True

    # 判断长度为2的子串是否是回文串
    for i in range(n - 1):
        l[i][i + 1] = (s[i] == s[i + 1])

    # 判断长度>=3的子串是否是回文串，动态规划
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            # s[i:j+1] 是回文串当且仅当 s[i+1:j] 是回文串 且 s[i] == s[j]
            l[i][j] = l[i + 1][j - 1] and (s[i] == s[j])

    # dp[i] 表示子串 s[0:i+1] 最少分割次数（使每部分都是回文串）
    dp = [0] * n

    for i in range(n):
        # 如果 s[0:i+1] 本身是回文串，分割次数为0
        if l[0][i]:
            dp[i] = 0
        else:
            # 否则初始化为最大分割次数 i（每个字符单独成一部分）
            dp[i] = i
            # 尝试分割点 j，找到最小分割次数
            for j in range(i):
                # 如果 s[j+1:i+1] 是回文串，则更新 dp[i]
                if l[j + 1][i]:
                    dp[i] = min(dp[i], dp[j] + 1)

    # 返回整个字符串的最少分割次数
    return dp[n - 1]


n = int(input())
for _ in range(n):
    s = input()
    print(f(s))
```

8.滑雪（计概上，dfs结合dp）

图的遍历与搜索（基本模板见计概）

1.踩方格

```
n = int(input())  # 输入允许的步数

# 初始状态：
# a: 当前在起点（0,0）的走法数（步数为0）
# b: 当前在北方向的走法数（即已经走了一步到北边）
# c: 当前在东或西方向的走法数
a, b, c = 1, 1, 1

# 动态规划：每次更新一步后的 a, b, c 状态
for i in range(n):
    # 设下一步状态为 a', b', c'
    # a' = 从原点走北、东、西三个方向过来
    # b' = 从原点或东西走北方向（走到北方向的路径）
    # c' = 从原点或北方向走东或西（走到东西方向的路径）
    a, b, c = a + b + c, a + b, a + c

# 输出最终在 n 步时总共有多少种路径（以 a 为主，因为它记录所有方案数）
print(a)
```

2.红与黑（本质为可行路径总数）

```
# 深度优先搜索函数
def dfs(x, y, visited, map, count):
    visited[x][y] = True  # 标记当前位置为已访问
    count += 1  # 计数 +1
    row = len(map)       # 行数
    col = len(map[0])    # 列数
    directions = [(0,1),(1,0),(-1,0),(0,-1)]  # 右、下、上、左 四个方向
    for dx, dy in directions:
        nx = x + dx
        ny = y + dy
        # 判断新位置是否在边界内、未访问、可通行（为 0）
        if 0 <= nx < row and 0 <= ny < col and not visited[nx][ny] and map[nx][ny] == 0:
            count = dfs(nx, ny, visited, map, count)  # 递归搜索
    return count  # 返回从该点出发的可达格子数

while True:
    n, m = map(int, input().split())  # 读入列数 n，行数 m
    if n == 0 and m == 0:
        break  # 输入为 0 0 时退出

    maps = [[0 for _ in range(n)] for _ in range(m)]  # 初始化地图，0 表示可走
    for i in range(m):  # 遍历每一行
        a = input()  # 读入字符串
        a = list(str(a))  # 转为字符列表
        for j in range(n):  # 遍历每一列
            if a[j] == '#':
                maps[i][j] = 1  # 墙壁，标记为 1
```

```
            elif a[j] == '.':
                maps[i][j] = 0  # 可通行，标记为 0
            else:
                maps[i][j] = 0  # 起点 '@' 也视为可通行
                current_x = i   # 记录起点坐标
                current_y = j

    visited = [[0 for _ in range(n)] for _ in range(m)]  # 访问标记数组
    print(dfs(current_x, current_y, visited, maps, 0))  # 输出可达格子数
```

3.判断有向图是否有回路：（用到了构件图邻接表，较新）

```
def has_cycle(n, edges):
    # 初始化邻接表
    adj_list = {i: [] for i in range(n)}
    for u, v in edges:
        adj_list[u].append(v)

    # 初始化访问状态数组
    visited = [False] * n
    rec_stack = [False] * n

    # 定义DFS函数
    def dfs(v):
        if rec_stack[v]:
            return True
        if visited[v]:
            return False

        visited[v] = True
        rec_stack[v] = True

        for neighbor in adj_list[v]:
            if dfs(neighbor):
                return True

        rec_stack[v] = False
        return False

    # 对每个顶点进行DFS
    for i in range(n):
        if not visited[i]:
            if dfs(i):
                return "yes"

    return "no"

def main():
    map_list = None
    n, m = map(int, input().split())
    map_list = [n, []]
    for _ in range(m):
        begin, end = map(int, input().split())
        map_list[1].append([begin, end])
```

```
    print(has_cycle(map_list[0], map_list[1]))
    new_map = True



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