**def** solve():

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

solve()

**Python2.7**

**"""输入"""**# 矩阵维度

n, m = map(int, raw\_input().split())

# 数组  
list\_in = []  
list\_in = list(map(int, raw\_input().split()))

*# 循环输入*while True:

try:

n=int(raw\_input())

r=[]

for i in range(n):

r.append(int(raw\_input()))

for i in sorted(set(r)):

print(i)

except:

break

try:

while True:

n=int(raw\_input())

r=[]

for i in range(n):

r.append(int(raw\_input()))

for i in sorted(set(r)):

print(i)

except:

pass

**前序遍历**

*# 法一：递归***def** preorderTraversal1(self, root):  
 *"""* **:type** *root: TreeNode* **:rtype***: List[int]  
 """* res = []  
 **if not** root:  
 **return** res  
 self.helper(root, res)  
 **return** res  
  
**def** helper(self, root, res):  
 res.append(root.val)  
 **if** root.left == **None and** root.right == **None**:  
 **return  
 if** root.left != **None**:  
 self.helper(root.left, res)  
 **if** root.right != **None**:  
 self.helper(root.right, res)  
*# 法二：迭代  
# 模拟系统栈，这种思路可以将任何递归算法改为为非递归***def** preorderTraversal2(self, root):  
 *"""* **:type** *root: TreeNode* **:rtype***: List[int]  
 """* res = []  
 **if not** root:  
 **return** res  
  
 stack = []  
 stack.append(root)  
 **while** stack != []:  
 tmp = stack[-1]  
 res.append(tmp.val) *# 取根节点的值* stack.pop()  
 **if** tmp.right:  
 stack.append(tmp.right) *# 先压右节点* **if** tmp.left:  
 stack.append(tmp.left) *# 再压左节点* **return** res

**HashMap**

map= dict() / map= {}

**for** i **in** nums:  
 map[i] = map.get(i, 0) + 1 *# 返回指定键的值，如果值不在字典中返回default值*

**return** [x[0] **for** x **in** sorted(map.items(), key = **lambda** item : item[1], reverse = **True**)][:k]

**动态规划、添加记忆**

**class** Solution(object):  
 **def** rob(self, root):storage\_0 = dict() *# 不添加记忆，则会超时* storage\_1 = dict()  
  
 **def** helper(root, take):  
 **if not** root:  
 **return** 0  
 **if** take:  
 **if** root.left **not in** storage\_0:  
 storage\_0[root.left] = helper(root.left, 0)  
 **if** root.right **not in** storage\_0:  
 storage\_0[root.right] = helper(root.right, 0)  
 **return** root.val + storage\_0[root.left] + storage\_0[root.right]  
 **else**:  
 **if** root.left **not in** storage\_0:  
 storage\_0[root.left] = helper(root.left, 0)  
 **if** root.right **not in** storage\_0:  
 storage\_0[root.right] = helper(root.right, 0)  
 **if** root.left **not in** storage\_1:  
 storage\_1[root.left] = helper(root.left, 1)  
 **if** root.right **not in** storage\_1:  
 storage\_1[root.right] = helper(root.right, 1)  
 **return** max( storage\_0[root.left], storage\_1[root.left] ) +

max( storage\_0[root.right], storage\_1[root.right] )

**return** max(helper(root, 0), helper(root, 1))