python刷题算法汇总

一：Python程序员面试算法宝典

1. 二叉树

```
# @Filename: trie树实现DNS缓存查找.py
class TrieNode:
    def __init__(self):
        CHAR_COUNT = 11
        self.isLeaf = False
        self.url = None
        self.child = [None] * CHAR_COUNT
        for i in range(len(self.child)):
            self.child[i] = None
            i += 1

def getIndexFromChar(c):
    # ord()返回ascii码
    return 10 if c == "." else (ord(c) - ord('0'))

def getCharFromIndex(i):
    return '.' if i == 10 else ('0' + str(i))

class DNSCache:
    def __init__(self):
        self.CHAR_COUNT = 11
        self.root = TrieNode()

    def insert(self, ip, url):
        lens = len(ip)
        pCrawl = self.root
        level = 0
        while level < lens:
            index = getIndexFromChar(ip[level])
            if pCrawl.child[index] == None:
                pCrawl.child[index] = TrieNode()
            pCrawl = pCrawl.child[index]
            level += 1
        pCrawl.isLeaf = True
        pCrawl.url = url

    def searchDNSCache(self, ip):
        pCrawl = self.root
```

```
        lens = len(ip)
        level = 0
        while level < lens:
            index = getIndexFromChar(ip[level])
            if pCrawl.child[index] == None:
                return None
            pCrawl = pCrawl.child[index]
            print(pCrawl.url)
            level += 1
        if pCrawl != None and pCrawl.isLeaf:
            return pCrawl.url
        return None

if __name__ == "__main__":
    ipAdds = ["10.57.11.127", "121.57.61.129", "66.125.100.103", "10.57.11.122"]
    # ipAdds = ["10.57.11.127", "10.57.11.122"]
    # url = ["www.samsung.com", "www.huawei.com"]
    url = ["www.samsung.com", "www.samsung.net", "www.google.in", "www.huawei.com"]
    n = len(ipAdds)
    cache = DNSCache()
    for i in range(n):
        cache.insert(ipAdds[i], url[i])
        print(i)
    ip = "10.57.11.127"
    res_url = cache.searchDNSCache(ip)
    print(res_url)
```

```
# @Filename: 二叉树最大子树和.py
# 想到将所有子树进行求和，然后进行最大值对比，由于后续遍历会访问所有的子树，所以采用后续遍历，同时记录每颗子树的和
# 只能找到最大值，怎么才算找到最大子树呢?

class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

class Test:
    def __init__(self):
        self.maxSum = -2 ** 31

    def constructTree(self):
        root = BiTree()
        node1 = BiTree()
        node2 = BiTree()
        node3 = BiTree()
        node4 = BiTree()
        root.data = 6
        node1.data = 3
        node2.data = -7
        node3.data = -1
        node4.data = 9
        root.lchild = node1
        root.rchild = node2
        node1.lchild = node3
        node1.rchild = node4
        return root

    def findMaxSubTree(self, root, maxRoot):
        if not root:
            return 0
        sum_l = self.findMaxSubTree(root.lchild, maxRoot)
        sum_r = self.findMaxSubTree(root.rchild, maxRoot)
        sums = sum_l + sum_r + root.data
        if sums > self.maxSum:
            self.maxSum = sums
            maxRoot.data = root.data
        return sums

```

```
if __name__ == "__main__":
    test = Test()
    root = test.constructTree()
    # 创建一个节点用于记录最大子树
    maxRoot = BiTree()
    test.findMaxSubTree(root, maxRoot)
    # test.findMaxSubTree2(root, 0, maxRoot)
    print("最大子树和为：", test.maxSum)
    print("最大子树节点为：", maxRoot.data)
```

```python
# @Filename: 二叉树的层次遍历.py
# 二叉树的层次遍历，将每次访问到的节点入队列，访问出栈时，将他的孩子节点入队列
from collections import deque
class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def arrtotree(arr):
    if not arr:
        return
    bt = BiTree()
    middle = len(arr) // 2
    bt.data = arr[middle]
    bt.lchild = arrtotree(arr[:middle])
    bt.rchild = arrtotree(arr[middle + 1:])
    return bt

def printTreeLayer(root):
    if not root:
        return
    queue = deque()
    queue.append(root)
    while queue:
        root = queue.popleft()
        print(root.data, end=" ")
        if root.lchild:
            queue.append(root.lchild)
        if root.rchild:
            queue.append(root.rchild)

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    root = arrtotree(arr)
    printTreeLayer(root)
```

```
# @Filename: 判断一个数组是否是二元查找树后序遍历.py
"""
二元查找树后序遍历特性
1. 最后访问节点一定为根节点
2. 比根节点大的且最先访问的节点作为分割点，前面的点都比根节点<=,该点和后续节都比根节点>=
"""
def IsAfterOrder(arr, start, end):
    if not arr:
        return
    # 找到比end的值大的最先的那个值的下标
    root = arr[end]
    i = start
    while i < end:
        if arr[i] > root:
            break
        i += 1
    j = i
    # 判定条件
    while j < end:
        if arr[j] < root:
            return False
        j += 1
    left_IsAfterOrder = True
    right_IsAfterOrder = True
    if i > start:
        left_IsAfterOrder = IsAfterOrder(arr, start, i - 1)
    if j < end:
        right_IsAfterOrder = IsAfterOrder(arr, i, end)
    return left_IsAfterOrder and right_IsAfterOrder

if __name__ == "__main__":
    arr = [1, 3, 2, 5, 7, 6, 4]
    result = IsAfterOrder(arr, 0, len(arr) - 1)
    print(result)
```

```
# @Filename: 判断两颗二叉树是否相等.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
两颗二叉树相等"
1. 当前点相等
2. 左子树相等、右子树相等
递归下去
"""
class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def isEqual(root1, root2):
    if not root1 and not root2:
        return True
    if root1 and not root2:
        return False
    if not root1 and root2:
        return False
    # 上面判断结构相等
    # 下面判断数据相等
    if root1.data == root2.data:
        return isEqual(root1.lchild, root2.lchild) and isEqual(root1.rchild, root2.rchild)
    else:
        return False

def constructTree():
    root = BiTree()
    node1 = BiTree()
    node2 = BiTree()
    node3 = BiTree()
    node4 = BiTree()

    root.data = 3
    node1.data = 6
    node2.data = 6
    node3.data = 6
    node4.data = 6
```

```
    root.lchild = node1
    root.rchild = node2
    node1.rchild = node3
    node2.rchild = node4
    return root

if __name__ == "__main__":
    root1 = constructTree()
    root1.data = 9999
    root2 = constructTree()
    print(isEqual(root1, root2))
```

```
# @Filename: 在二叉排序数中找出第一个大于中间值的节点.py
# 1. 找到最小值和最大值，分别为lchild.lchild.。。。和rchild.rchild.。。。直到为None
# 2. 利用中序遍历，由于是有顺序的，直接比较即可

class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def arrtotree(arr):
    if not arr:
        return
    bt = BiTree()
    middle = len(arr) // 2
    bt.data = arr[middle]
    bt.lchild = arrtotree(arr[:middle])
    bt.rchild = arrtotree(arr[middle + 1:])
    return bt

def getMinNode(root):
    if not root:
        return
    while root.lchild:
        root = root.lchild
    return root

def getMaxNode(root):
    if not root:
        return
    while root.rchild:
        root = root.rchild
    return root

def getNode(root):
    min = getMinNode(root).data
    max = getMaxNode(root).data
    mid = (min + max) / 2

    result = None
    # 直接通过比较大小找到相应节点
    while root != None:
        if root.data <= mid:
```

```
            root = root.rchild
        else:
            result = root
            root = root.lchild
    return result

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7]
    root = arrtotree(arr)
    print(getNode(root).data)
```

```python
# @Filename: 在二叉树中找出与输入整数相等的所有路径.py
class BiTNode:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def constructTree():
    root = BiTNode()
    node1 = BiTNode()
    node2 = BiTNode()
    node3 = BiTNode()
    node4 = BiTNode()
    root.data = 6
    node1.data = 3
    node2.data = 2
    node3.data = -1
    node4.data = 9
    root.lchild = node1
    root.rchild = node2
    node1.lchild = node3
    node1.rchild = node4
    return root

def FindRoad(root, num, sums, v):
    sums += root.data
    v.append(root.data)

    # 如果必须是根节点到叶子节点的路径，则添加左右子树为None的判断，否则，不用添加
    # if not root.lchild and not root.rchild and num == sums:
    if num == sums:
        for i in v:
            print(i, end=" ")
        print()
    if root.lchild:
        FindRoad(root.lchild, num, sums, v)
    if root.rchild:
        FindRoad(root.rchild, num, sums, v)
    # 清除当前节点数据
    sums -= root.data
    v.pop()

```

```
if __name__ == "__main__":
    root = constructTree()
    FindRoad(root, 8, 0, [])
```

```
# @Filename: 在二叉树中找出路径最大的和.py
# 这里的路径包含任意节点作为起点和终点
"""
分析：采用后续遍历，一次计算左子树最大节点和（连续的路径），右子树最大节点和（连续的路径），以及最大节点和（局部连续）
"""
class BiTNode:
    def __init__(self, val):
        self.val = val
        self.lchild = None
        self.rchild = None

class InfRef:
    def __init__(self):
        self.val = None

def Max(a, b, c, d):
    maxs = a if a > b else b
    maxs = maxs if maxs > c else c
    maxs = maxs if maxs > d else d

    return maxs

def findMaxPathRecursive(root, maxs):
    if not root:
        return 0
    sum_left = findMaxPathRecursive(root.lchild, maxs)
    sum_right = findMaxPathRecursive(root.rchild, maxs)

    # 这里需要加上本身节点，才能在上一层变成连续的路径
    allMax = root.val + sum_left + sum_right
    rightMax = root.val + sum_right
    leftMax = root.val + sum_left
    # 同时统计本身是否比左、右、左加右都还大
    tmpMax = Max(allMax, rightMax, leftMax, root.val)

    # 更新最大值
    if tmpMax > maxs.val:
        maxs.val = tmpMax
    subMax = sum_left if sum_left > sum_right else sum_right
    return root.val + subMax

def findMaxPath(root):
```

```
    maxs = InfRef()
    maxs.val = -2 ** 31
    findMaxPathRecursive(root, maxs)
    return maxs.val

if __name__ == "__main__":
    root = BiTNode(10)
    left = BiTNode(-1)
    right = BiTNode(-2)
    root.lchild = left
    root.rchild = right
    left1 = BiTNode(5)
    right1 = BiTNode(4)
    left.lchild = left1
    left.rchild = right1
    print(findMaxPath(root))
```

```
# @Filename: 复制二叉树.py
class BiTNode:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def createDupTree(root):
    if not root:
        return None
    dupTree = BiTNode()
    dupTree.data = root.data
    dupTree.lchild = createDupTree(root.lchild)
    dupTree.rchild = createDupTree(root.rchild)
    return dupTree

def printTreeMidOrder(root):
    if not root:
        return
    printTreeMidOrder(root.lchild)
    print(root.data, end=" ")
    printTreeMidOrder(root.rchild)

def constructTree():
    root = BiTNode()
    node1 = BiTNode()
    node2 = BiTNode()
    node3 = BiTNode()
    node4 = BiTNode()
    root.data = 6
    node1.data = 3
    node2.data = -7
    node3.data = -1
    node4.data = 9
    root.lchild = node1
    root.rchild = node2
    node1.lchild = node3
    node1.rchild = node4
    return root
if __name__ == "__main__":
    root1 = constructTree()
    root2 = createDupTree(root1)
```

```
    printTreeMidOrder(root1)
    print("\ndup it")
    printTreeMidOrder(root2)
```

```
# @Filename: 对二叉树进行镜像翻转.py
# @Software: PyCharm
# @e_mail:   sancica@163.com

from collections import deque


class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def reverseTree(root):
    if not root:
        return
    reverseTree(root.lchild)
    reverseTree(root.rchild)
    tmp = root.lchild
    root.lchild = root.rchild
    root.rchild = tmp

def arrtotree(arr):
    if not arr:
        return
    bt = BiTree()
    middle = len(arr) // 2
    bt.data = arr[middle]
    bt.lchild = arrtotree(arr[:middle])
    bt.rchild = arrtotree(arr[middle + 1:])
    return bt

def printTreeLayer(root):
    if not root:
        return
    queue = deque()
    queue.append(root)
    while queue:
        p = queue.popleft()
        print(p.data, end=" ")
        if p.lchild:
            queue.append(p.lchild)
        if p.rchild:
```

```
            queue.append(p.rchild)
    print()

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7]
    root = arrtotree(arr)
    printTreeLayer(root)
    reverseTree(root)
    printTreeLayer(root)
```

```python
# @Filename: 将有序数组放到二叉树中.py
# 可以采用中序遍历的顺序将数据放入进去
# 实现思路，将中间节点作为根节点，然后将左子树的中间节点作为左子树的根节点，依次循环
class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def arr2tree(arr):
    # 将采用递归调用
    # 1. 确定递归返回条件
    if not arr:
        return None
    # 2. 找到中点,并将该点放入二叉树
    middle = len(arr) // 2
    bt = BiTree()
    bt.data = arr[middle]
    # 递归建立左子树节点
    bt.lchild = arr2tree(arr[:middle])
    # 递归建立右子树
    bt.rchild = arr2tree(arr[middle + 1:])
    return bt

def printTree_middle(root):
    if not root:
        return
    if root.lchild:
        printTree_middle(root.lchild)
    print(root.data, end=" ")
    if root.rchild:
        printTree_middle(root.rchild)

if __name__ == "__main__":
    sorted_arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    print("数组")
    for num in sorted_arr:
        print(num, end=" ")
    print("\n")
    root = arr2tree(sorted_arr)
    # 采用中序遍历整个二叉树
    printTree_middle(root)
```

```
# @Filename: 找出排序二叉树上任意两点的最近共同父节点1（路径对比法）.py
# 遍历所有路径，当前点与所需点相等，返回True，并push该节点
"""
分析：时间复杂度，最坏情况访问整个二叉树，为O(n)，空间复杂度，由于导入栈s存储数据，最坏情况是存储整个二叉树，也为O（n）
"""
class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return True if not self.items else False

    def size(self):
        return len(self.items)

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return None

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("栈已空")
            return None

    def push(self, item):
        self.items.append(item)

def getPathFromRoot(root, node, s):
    # 路径遍历，传入栈s，将找到的路径递归加入栈中
    if not root:
        return False
    if root == node:
```

```
            s.push(root)
            return True

    if getPathFromRoot(root.lchild, node, s) or getPathFromRoot(root.rchild, node, s):
            s.push(root)
            return True
    return False

def arraytotree(arr):
    if not arr:
            return
    bt = BiTree()
    middle = len(arr) // 2
    bt.data = arr[middle]
    bt.lchild = arraytotree(arr[:middle])
    bt.rchild = arraytotree(arr[middle + 1:])
    return bt

def FindParentNode(root, node1, node2):
    stack1 = Stack()
    stack2 = Stack()
    getPathFromRoot(root, node1, stack1)
    getPathFromRoot(root, node2, stack2)
    common_parent = None
    while stack1.peek() == stack2.peek():
            common_parent = stack1.peek()
            stack2.pop()
            stack1.pop()
    return common_parent

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    root = arraytotree(arr)
    node1 = root.lchild.lchild.lchild
    node2 = root.lchild.rchild
    res = FindParentNode(root, node1, node2)
    if res:
            print(str(node1.data)+"与"+str(node2.data)+"的最近公共父节点为："+str(res.data))
    else:
            print("没有公共交接点")
```

```
# @Filename: 找出排序二叉树上任意两点的最近共同父节点2（节点编号法）.py
"""
将根节点设为编码1，左子树添加0，右子树添加1，对整个树进行编码，所以时间复杂度为O（n）
对两个子节点的编码分别max(n1, n2) = max(n1, n2)/2,直至两个节点相等，最终确定父节点编码
"""
import math
class BiTree:
    def __init__(self):
        self.data = None
        self.lchild = None
        self.rchild = None

def arraytotree(arr):
    if not arr:
        return
    bt = BiTree()
    middle = len(arr) // 2
    bt.data = arr[middle]
    bt.lchild = arraytotree(arr[:middle])
    bt.rchild = arraytotree(arr[middle + 1:])
    return bt

class IntRef:
    def __init__(self):
        self.num = None

def getNo(root, node, number):
    # 找到node点的number编码，左子树*2，右子树*2+1
    if not root:
        return False
    if root == node:
        return True
    tmp = number.num
    number.num = 2 * tmp
    if getNo(root.lchild, node, number):
        return True
    else:
        number.num = 2 * tmp + 1
        return getNo(root.rchild, node, number)

def getNodeFromNum(root, number):
    if not root or number < 0:
```

```
        return None
    if number == 1:
        return root
    # 计算number对应二进制的长度，根据完全二叉树的性质ceil(log2(n))+1
    lens = int(math.log(number) / math.log(2))
    number -= 1 << lens
    while lens > 0:
        # 这里终于搞明白了，获取当前这一位二进制的值
        print(1 << (lens - 1) & number >= 1)
        if (1 << (lens - 1) & number) >= 1:
            root = root.rchild
        else:
            root = root.lchild
        lens -= 1
    return root

def FindParentNode(root, node1, node2):
    ref1 = IntRef()
    ref1.num = 1
    ref2 = IntRef()
    ref2.num = 1
    getNo(root, node1, ref1)
    getNo(root, node2, ref2)
    num1 = ref1.num
    num2 = ref2.num

    while num1 != num2:
        if num1 > num2:
            num1 //= 2
        else:
            num2 //= 2
    return getNodeFromNum(root, num1)

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11,12,13,14,15,16,17,18,19,20]
    # arr = [1,2,3,4, 5,6,7,8,9,10]
    root = arraytotree(arr)
    node1 = root.lchild.rchild.lchild
    node2 = root.lchild.rchild.rchild
    res = FindParentNode(root, node1, node2)
    print(res.data)
    if res:
```

```
85.         print(str(node1.data) + "与" + str(node2.data) + "的最近公共父节点为：" + str(res.data))
86.     else:
87.         print("没有公共交接点")
```

```
# @Filename: 把二叉树转换为双向链表.py
"""
搜索二叉树：左子树<根节点<右子树
通过中序遍历能够找出有序的顺序
本方法通过中序遍历，在遍历过程中修改指针的指向实现双向链表，即当前点的left = 上一节点，上一节点的right = 当前点
"""
class BiTree:
    def __init__(self):
        self.data = None
        self.lc = None
        self.rc = None

class Test:
    def __init__(self):
        self.pHead = None
        self.pEnd = None

    def arraytotree(self, arr):
        if not arr:
            return None
        bt = BiTree()
        middle = len(arr) // 2
        bt.data = arr[middle]
        bt.lc = self.arraytotree(arr[:middle])
        bt.rc = self.arraytotree(arr[middle + 1:])
        return bt

    def treetolist(self, root):
        if not root:
            return
        self.treetolist(root.lc)
        # 当前节点左子树指向前一节点
        root.lc = self.pEnd
        if not self.pEnd:
            # 保留头结点
            self.pHead = root
        else:
            # 前一节点的右节点指向当前节点
            self.pEnd.rc = root
        # 前移
        self.pEnd = root
```

```
        self.treetolist(root.rc)

if __name__ == "__main__":
    arr = [1,2,3,4,5,6,7,8,9,10]
    test = Test()
    root = test.arraytotree(arr)
    test.treetolist(root)
    cur = test.pHead
    print("正向遍历")
    while cur:
        print(cur.data, end=" ")
        cur = cur.rc
    cur = test.pEnd
    print("\n")
    print("逆向遍历")
    while cur:
        print(cur.data, end=" ")
        cur = cur.lc
```

2. 基本数字运算

```
# @Filename: 判断1024的阶乘末尾有多少个0.py
"""
分析:
1. 结果中0的来源
    a. 5与偶数的乘积，由于偶数的数量肯定比5的倍数的数量多，所以只考虑5的倍数的数量
    b. 乘子中本来就有0的数，如100等，但由于100本身就是5和25的倍数，在a中就进行统计了，所以就不用考虑b这一项了
"""
def zeroCount(n):
    count = 5
    res = 0
    while True:
        i = 1
        while i <= n:
            if i % count == 0:
                res += 1
            i += 1
        count *= 5

        if count > n:
            break
    print(res)

def zeroCount_nice(n):
    count = 0
    while n > 0:
        n = n // 5
        count += n
    print(count)
zeroCount_nice(1)
```

```
# @Filename: 判断一个数是否为2的n次方.py
"""
分析:
与2沾边的数，最好采用位移操作
1. 构造2的倍数，判断是否相等
2. 判断该数的二进制的第一位是否为1，其他为0
"""
def construct_num(n):
    i = 1
    while i <= n:
        if i == n:
            return True
        i <<= 1
    return False

def judge_ord(n):
    # 负数在内存中是补码的形式存在
    if n < 1:
        return False
    if not (n & n - 1):
        return True
    else:
        return False
print(judge_ord(0))
```

```
# @Filename: 判断一个自然数是否是某个数的平方.py
"""
前提：不使用开方运算
分析：
1. 直接计算，i递增，计算i**2与n的大小关系
2. 利用二分查找法找到对应的i，减少运算的次数
3. 减法运算法：通过对平方数的构造规律得：(n+1)^2 = n^2+2n+1 = (n-1)^2+(2(n-1)+1)+2n+1 = ... = 1+(2*1+1)+(2*2+1)+...+(2*n+1)
"""
def isPower_bn(n):
    i = 0
    j = n
    mid = (i+j)//2
    while i <=j:
        if mid**2 == n:
            return True
        elif mid**2 > n:
            j = mid-1
        else:
            i = mid+1
        mid = (i+j)//2
    return False

def isPower_sub(n):
    """
    减法运算法
    """
    res = n
    flag = 1
    while res >= 0:
        if res == 0:
            return True
        res -= flag
        flag += 2
    return False

print(isPower_sub(25))
```

```
# @Filename: 如何不使用^实现异或运算.py
"""
使用或 |实现
"""
def xor(x, y):
    # (x | y)为1时, (0,1) (1,0) (1,1) : 需要排除 (1,1) ,则在后面& (~x|~y)
    return (x | y) & (~x | ~y)

print(xor(2, 2))
```

```
# @Filename: 如何不使用加减乘除运算实现减法.py
"""
分析:
减法就是加一个负数，同加法一样处理
"""
def add(a, b):
    # 将b转为b的负数形式，然后采用加法的运算
    # b = ~b + 1
    while True:
        res = a ^ b
        carry = (res & b) << 1
        if carry == 0:
            return res
        a = res
        b = carry
    return res

def sub(a, b):
    return add(a, b)

print(sub(14, 4))
```

```
# @Filename: 如何不使用加减乘除运算实现加法.py
"""
# 利用二进制的异或和 与运算实现加法
"""
# 注：python没有自动溢出的功能，int是无线长的，随着位数增长，自动改变类型为long，所以这里需要用到截断
def sum(num1, num2):
    while True:
        res = (num1 ^ num2) & 0xffffffff
        carry = ((num1 & num2) << 1) & 0xffffffff
        if carry == 0:
            # 前面部分，表示里面如果没有负数，后面部分表示有负数
            return res if res <= 0x7fffffff else ~(res ^ 0xffffffff)
        num1 = res
        num2 = carry
print(sum(-11, 2))
```

```
# @Filename: 如何不使用除法操作符实现两个正整数的除法.py
"""
分析:
1. 减法，被除数循环减去除数，直到值小于除数，次数即为商，余数为当前值(不使用%实现%)
2. 位移法，通过位移加快逼近速度
"""
def sub_divid(n, m):
    count = 0
    while n >= m:
        n -= m
        count += 1
    return count

def shift_divid(n, m):
    res = 0
    index = 0
    while n >= m:
        flag = m * 2 ** index
        count = 2 ** index
        if flag > n:
            n -= flag // 2
            count //= 2
            res += count
            index = 0
        elif flag < n:
            index += 1
        else:
            return res + count
    return res
print(shift_divid(200, 20))
```

```
# @Filename: 如何找最小的不重复数.py
"""
分析:
不重复数：数的每一位相邻之间不同，如1201  1212
题目：给定n，找到最小的那个不重复数
1. 首先n += 1
2. 从右往左找到第一对重复的数i-1和i，对i对应的数字加一（需考虑进位）
    a. 如果此时i和i+1对应的数字相等，则i++
    b. 否则continue
"""
def find_min_num(n):
    n = n+1
    num = list(str(n))
    i = len(num) - 1
    while i > 0:
        if num[i] == num[i - 1]:
            n += 10 ** (len(num) - i - 1)
            num = list(str(n))
            flag = 1
            j = i+1
            while j < len(num):
                flag = 1 ^ flag
                num[j] = str(flag)
                j += 1
            print(num)
            n = int(''.join(num))
            if i < len(num)-1 and num[i] == num[i + 1]:
                i += 1
                continue
        i -= 1
    return n

print(find_min_num(8989))
```

```
# @Filename: 如何按要求比较两个数的大小.py
"""
题目：比较两个数大小，不能使用大于、小于、if语句
"""
def maxs(a, b):
    return ((a + b) + abs(a - b)) / 2
print(maxs(5, 6))
```

```
# @Filename: 如何根据已知随机数生成函数计算新的随机数.py
"""
题目：假设已知随机生成函数rand7()能生成整数1~7之间的均匀分布，如何构造rand10()函数，使其产生的随机数是整数1~10的均匀分布
分析：
 (rand7()-1) *7+rand()可以造出1-49的均匀分布，通过截取1-40的数据然后求余得到1-10的均匀分布
1. 这里的已知随机函数可以为任意自然数，都可以推导出来（n>=4）
"""
import random
def rand7():
    return int(random.uniform(1, 7))


def rand10():
    while True:
        num = (rand7()-1)*7+rand7()
        if num <=40:
            break
    return num % 10+1

from collections import Counter

res = []
for i in range(10000000):
    res.append(rand10())

print([i/10000000 for i in dict(Counter(res)).values()])
```

```
# @Filename: 如何求有序数列的1500个数的值.py
"""
题目：数列是有序数列，里面的数能够被2或者3或者5整除
分析:
1. 遍历法
2. 规律法
"""
def search(n):
    a = [0, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28]
    print(len(a))
    ret = (n // 22) * 30 + a[n % 22]
    print(ret)
search(1500)
```

```
# @Filename: 如何计算一个数的n次方.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
分析:
1. 连乘或连除
2. 递归或者动态规划法
"""
def power(d, n):
    # 判断是奇数还是偶数次，将采用不同的公式
    if n == 0: return 1
    if n == 1: return d
    tmp = power(d, abs(n) // 2)
    if n > 0:
        if n % 2 == 1:
            return tmp * tmp * d
        else:
            return tmp * tmp
    else:
        if n % 2 == 1:
            return 1 / (tmp * tmp * d)
        else:
            return 1 / (tmp * tmp)

def dynamic_power(d, n):
    if n == 0:
        return 1
    if n == 1:
        return d
    if n == -1:
        return 1/d
    arr = [0] * abs(n)
    arr[0] = 1
    arr[1] = d
    i = 2
    while i < abs(n):
        arr[i] = arr[i // 2] ** 2
        i = i*2
    if i == abs(n):
        if n > 0:
            return arr[n]
```

```
        else:
            return 1 / arr[-n]
    else:
        if n > 0:
            return arr[n//2]**2 * d
        else:
            return 1 / (arr[-n//2]**2 * d)

print(dynamic_power(2, -1))
```

```
# @Filename: 将十进制数分别以二进制和十六进制形式输出.py
"""
分析:
十进制转二进制可以考虑采用"除2倒取余法"
"""
def intToBinary(n):
    hexNum = 8 * 8
    bit = []
    for i in range(hexNum):
        b = n >> i
        # 同时计算商和余数
        c, d = divmod(b, 2)
        bit.append(str(d))
    return ''.join(bit[::-1])

def intToHex(s):
    hexs = ""
    while s != 0:
        remainder = s % 16
        if remainder < 10:
            hexs = str(remainder + ord('0')) + hexs
        else:
            hexs = str(remainder - 10 + ord('A')) + hexs
        s = s >> 4
    return chr(int(hexs))

print(intToBinary(11))
print(intToHex(10))
```

```
# @Filename: 求二进制数中1的个数.py
"""
分析:
1. 依次向右位移，判断最后一位是否为1（采用&1判断）
2. n依次与n-1相与&，每次都会减少一个1，直到结果为0
"""
def judge_count(n):
    count = 0
    while n > 0:
        if n & 1 == 1:
            count += 1
        n = n >> 1
    return count

print(judge_count(8))

def judge_count_and(n):
    count = 0
    while True:
        if n != 0:
            n = n & (n - 1)
            count += 1
        else:
            break
    return count

print(judge_count_and(8))
```

3. 字符串

```
# @Filename: 判断一个字符串是否为另一个字符串旋转得到.py
"""
分析:
s2由s1旋转得到，那么s2一定是s1s1的子串
"""
def rotateSame(s1, s2):
    if len(s1) != len(s2):
        return False
    s1 = s1 + s1
    # 这里判断是否为子串用in，但自己实现的话需要用到kmp算法
    if s2 in s1:
        return True
    else:
        return False
if __name__ == "__main__":
    print(rotateSame("waterbollte", "erbolltewat"))
```

```
# @Filename: 判断一个字符串是否包含重复字符.py
"""
题目：字符串中是否包含重复字符串
分析：
1. 蛮力法，双重循环判断是否有相等字符
2. hash表法
3. set长度比较法
本代码实现hash法
"""
def isDup(strs):
    str_dic = {}
    for i in strs:
        if i not in str_dic:
            str_dic[i] = 1
        else:
            return False
    return True

def isDup1(strs):
    return len(strs) == len(set(strs))

print(isDup("fdsasd"))
```

```
# @Filename: 判断两个字符串是否为换为字符串.py
"""
换位字符串：两个字符串由同样的字符构成，但位置不同
分析：
1. 字典法
"""
from collections import Counter
def compare(str1, str2):
    counter1 = Counter(str1)
    counter2 = Counter(str2)
    for key in counter1.keys():
        if key in counter2:
            if counter1[key] != counter2[key]:
                return False
        else:
            return False
    return True

if __name__ == "__main__":
    str1 = "aaaabcc"
    str2 = "abcbaaa"
    print(compare(str1, str2))
```

```
# @Filename: 判断两个字符串的包含关系.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
题目：判断字符串的包含关系：即B中出现的字符在A中都有出现，则A包含B
分析：
1. 双层循环
2. 空间换时间，利用hash表
"""
# 本代码实现第二种方法
def is_contain(str1, str2):
    cha_dic = {}
    for char in str2:
        if char not in cha_dic:
            cha_dic[char] = 1

    for char in str1:
        if char in cha_dic:
            cha_dic[char] = 0

    return not sum(cha_dic.values())
print(is_contain("abcd", "agd"))
```

```
# @Filename: 判断字符串是否为整数.py
"""
分析:
字符串形式的整数有三种表达方法  "52", "-52", "+52"
"""
def judge_int(str):
    if str[0] == "+" or str[0] == "-":
        index = 1
    else:
        index = 0
    result = 0
    while index < len(str):
        if str[index].isdigit():
            result = result*10 + int(str[index])
        else:
            print("不是数字")
            return
        index += 1
    return -result if str[0] == "-" else result

print(judge_int("+4234"))
```

```
# @Filename: 句子中的单词反转.py
"""
分析:
1. 先对整个句子进行反转，最后再对每个单词进行反转
"""
def reverseStr(stl, begin, end):
    # 使用异或的方式进行反转

    while begin < end:
        stl[begin] = ord(stl[begin]) ^ ord(stl[end])
        stl[end] = stl[begin] ^ ord(stl[end])
        stl[begin] = stl[begin] ^ stl[end]
        stl[begin] = chr(stl[begin])
        stl[end] = chr(stl[end])
        begin += 1
        end -= 1

def swapWord(str):
    stl = list(str)
    reverseStr(stl, 0, len(stl) - 1)
    begin = 0
    end = 0
    while end < len(stl):
        if stl[end] == " ":
            reverseStr(stl, begin, end - 1)
            end = end + 1
            begin = end
        else:
            end += 1
    return "".join(stl)

if __name__ == "__main__":
    print(swapWord("I love you"))
```

```
# @Filename: 如何在二维数组中寻找最短路径（动态规划）.py
"""
分析：动态规划
1. 定义一个二维数组存储中间路径最小值
    a. 初始化f[i,0] = arr[1][0] + ... + arr[i][0]    f[0][j] = arr[0][1] + ... + arr[0][j]
    b. 动态推导公式f[i][j] = arr[i][j]+min(f[i-1][j], f[i][j-1])
"""
def getMinPath(arr):
    len1 = len(arr)
    len2 = len(arr[0])
    # 初始化二维数组
    f = [[0] * len2 for _ in range(len1)]
    i = 0
    temp_sum = 0
    while i < len1:
        temp_sum += arr[i][0]
        f[i][0] = temp_sum
        i += 1
    i = 0
    temp_sum = 0
    while i < len2:
        temp_sum += arr[0][i]
        f[0][i] = temp_sum
        i += 1
    for i in range(1, len1):
        for j in range(1, len2):
            f[i][j] = arr[i][j] + min(f[i - 1][j], f[i][j - 1])
    print("二维数组最短路径长度为：", f[i][j])
    return f[i][j]

getMinPath([[1, 4, 3], [8, 7, 5], [2, 1, 5]])
```

```
# @Filename: 如何在二维数组中寻找最短路径（递归）.py
"""
分析:
采用递归的方法，从最后值
"""
def getMinPath(arr, i, j):
    if i == 0 and j == 0:
        return arr[i][j]
    elif i > 0 and j > 0:
        return arr[i][j] + min(getMinPath(arr, i - 1, j), getMinPath(arr, i, j - 1))
    elif i > 0 and j == 0:
        return arr[i][j] + getMinPath(arr, i - 1, j)
    elif i == 0 and j > 0:
        return arr[i][j] + getMinPath(arr, i, j - 1)

if __name__ == "__main__":
    arr = [[1, 4, 3], [8, 7, 5], [2, 1, 5]]
    print(getMinPath(arr, len(arr) - 1, len(arr[0]) - 1))
```

```
# @Filename: 如何找到到达目标词的最短链长度.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
# 实现BFS算法，广度优先遍历
"""

from collections import deque


class QItem:
    def __init__(self, word, lens):
        self.word = word
        self.lens = lens


def isAdjacent(a, b):
    """
    判断两个字符串是否为邻接字符串
    """
    index = 0
    for i in range(len(a)):
        if a[i] != b[i]:
            index += 1
        if index > 1:
            return False
    return index == 1


def shortestChainLen(start, target, D):
    Q = deque()
    item = QItem(start, 1)
    Q.append(item)
    while Q:
        curr = Q[0]
        Q.pop()
        for it in D:
            temp = it
            if isAdjacent(curr.word, temp):
                item.word = temp
                item.lens = curr.lens + 1
                Q.append(item)
                D.remove(temp)
```

```
                if temp == target:
                    return item.lens
    return 0

if __name__ == "__main__":
    D = []
    D.append("pooN")
    D.append("pbcc")
    D.append("zamc")
    D.append("poIc")
    D.append("pbca")
    D.append("pbIc")
    D.append("poIN")
    start = "TooN"
    target = "pbca"
    print(shortestChainLen(start, target, D))
```

```
# @Filename: 如何找到由其他单词组成的最长单词.py
"""
分析:
1. 首先按照单词长度进行排序
2. 由最长单词开始进行递归判断，从左到右判断子串是否在单词表中，存在继续判断右边
"""
class LongestWord:
    def find(self, strArray, strs):
        if strs in strArray:
            return True
        else:
            return False

    def isContain(self, strArray, word, length):
        lens = len(word)
        if lens == 0:
            return True
        i = 1
        while i <= lens:
            if i == length:
                return False
            strs = word[0:i]
            if self.find(strArray, strs):
                if self.isContain(strArray, word[i:], length):
                    return True
            i += 1
        return False

    def getLongestStr(self, strArray):
        strArray = sorted(strArray, key=len, reverse=True)
        i = 0
        while i < len(strArray):
            if self.isContain(strArray, strArray[i], len(strArray[i])):
                return strArray[i]
            i += 1
        return None

if __name__ == "__main__":
    strArray = ['a', 'b', 'ab']
    lw = LongestWord()
    print(lw.getLongestStr(strArray))
```

```
# @Filename: 如何求字符串的编辑距离(动态规划).py
"""
分析:
编辑距离:将一个字符串s1通过增、删、改(替换)的方式转为字符串s2所需要的步骤
动态规划求解:
1. 建立一个2为数组D,表示将s1的i子串变为s2的j子串所需要的编辑距离
    a. i = 0时, D[i][j] = j
    b. j = 0 时,  D[i][j] = i
2. 增:如果计算出了D(i, j-1)的值,那么D(i, j) 等于D(i,j-1)+1
3. 删:如果计算出了D(i-1, j)的值,那么D (i, j) 等于D (i-1, j) +1
4. 改:如果计算出了D (i-1, j-1) 的值,如果s1[i]==s2[j],则D (i,j)  = D(i-1, j-1),否则,
D (i,j)  = D(i-1, j-1)+1:将s2[j]改为s1[i]

"""
def minEditDistance(s1, s2):
    len1 = len(s1)
    len2 = len(s2)
    editDistance = [[None] * (len2+1) for _ in range(len1+1)]
    # 初始化边缘值
    for i in range(len1+1):
        editDistance[i][0] = i
    for j in range(len2+1):
        editDistance[0][j] = j
    for i in range(1, len1+1):
        for j in range(1, len2+1):
            if s1[i-1] == s2[j-1]:
                editDistance[i][j] = min(editDistance[i-1][j]+1, editDistance[i][j-1]+1, editDistance[i-1][j-1])
            else:
                editDistance[i][j] = min(editDistance[i-1][j]+1, editDistance[i][j-1]+1, editDistance[i-1][j-1]+1)
    return editDistance[-1][-1]

print(minEditDistance("", "ros"))
```

```
# @Filename: 如何消除字符串的内嵌括号.py
# @Software: PyCharm
# @e_mail:   sancica@163.com

"""
题目：字符串中的括号问题
分析：
1. 用栈或者一个数来确保括号的配对
"""
def removeNestedPare(strs):
    arr = list(strs)[1:-1]
    register = 0
    for i in range(len(arr)):
        if arr[i] == "(":
            register += 1
            arr[i] = ''
        elif arr[i] == ")":

            arr[i] = ''
            register -= 1
            if register < 0:
                print("非法表达式，括号不匹配")
                return None
    print(arr)
    return '(' + ''.join(arr) + ')'

print(removeNestedPare("(1,(2,3),(4,(5,6),7))"))
```

```
# @Filename: 如果按照给定的字符序列对字符数组排序.py
# 根据给定的字符串顺序对字符串进行排序
"""
分析:
对字符串顺序建立hash表，value值为顺序值，比较大小的时候利用value值进行比较，越小优先级越高
"""
def compare(str1, str2, char_to_int):
    i = 0
    while i < len(str1) and i < len(str2):
        if char_to_int[str1[i]] < char_to_int[str2[i]]:
            return True
        elif char_to_int[str1[i]] > char_to_int[str2[i]]:
            return False
        i += 1

def insertSort(s, char_to_int):
    # 插入排序
    lens = len(s)
    i = 1
    while i < lens:
        temp = s[i]
        j = i - 1
        while j >= 0:
            if compare(temp, s[j], char_to_int):
                # 每一项后移
                s[j + 1] = s[j]
            else:
                break
            j -= 1
        # 将对应位置赋值
        s[j + 1] = temp
        i += 1

if __name__ == "__main__":
    s = ["bed", "dog", "dear", "eye"]
    sequence = "dgecfboa"
    char_to_int = {}
    for i in range(len(sequence)):
        char_to_int[sequence[i]] = i

    insertSort(s, char_to_int)
    print(s)
```

```
# @Filename: 字符串反转（异或）.py

def reverse(strs):
    i = 0
    j = len(strs) - 1
    # 这里必须转为list，因为字符串属于常量，无法修改
    strs = list(strs)
    while i < j:
        # 通过异或的方式进行交换
        print(type(ord(strs[i])))
        strs[i] = chr(ord(strs[i]) ^ ord(strs[j]))
        strs[j] = chr(ord(strs[i]) ^ ord(strs[j]))
        strs[i] = chr(ord(strs[i]) ^ ord(strs[j]))
        i += 1
        j -= 1
    return ''.join(strs)

if __name__ == "__main__":
    str = "abcdefg"
    print(reverse(str))
```

```
# @Filename: 实现字符串的匹配（KMP）.py
"""
题目：字符串匹配算法 KMP：
分析：
1. 首先建立nexts数组，该数组的值用于表明当子串第j位与主串第i位不匹配时，此时应该将j移动到的位置k
    a. 当j==0时，next[j] = -1
    b. 当j == 1时，next[j] = 0
    c. 当p[j] == p[k]:next[j+1] = next[j] + 1
    d. 当p[j] != p[k]: k = next[k]

https://www.cnblogs.com/yjiyjige/p/3263858.html
"""
def getNext(p, nexts):
    k = 0
    j = -1
    nexts[0] = -1
    while k < len(p):
        if j == -1 or p[j] == p[k]:
            k += 1
            j += 1
            nexts[k] = j
        else:
            j = nexts[j]

def match(s, p, nexts):
    if not s or not p:
        return -1
    slen = len(s)
    plen = len(p)

    if slen < plen:
        return -1
    i = 0
    j = 0
    while i < slen and j < plen:
        print("i="+str(i)+","+str(j))
        if j == -1 or s[i] == p[j]:
            i += 1
            j += 1
        else:
            j = nexts[j]
    if j >=plen:
```

```
            return i-plen
        return -1

if __name__ == "__main__":
    s = "abababaabcbab"
    p = "abaabc"
    lens = len(p)
    nexts = [0]*(lens+1)
    s = list(s)
    p = list(p)
    getNext(p, nexts)
    print("匹配结果: "+str(match(s, p, nexts)))
```

```
# @Filename: 对大小写字母组成的字符数组进行排序.py
"""
题目：这里的排序指，将其中的所有小写字母排在大写字母的前面，不需要考虑小写字母之间或者大写字母之间的排序关系
分析：
1. 首尾指针一次遍历法
"""
def sort(str):
    arr = list(str)
    i = 0
    j = len(arr) - 1
    while i < j:
        if 'A' <= arr[i] <= 'Z' and 'a' <= arr[j] <= 'z':
            arr[i] = ord(arr[i]) ^ ord(arr[j])
            arr[j] = arr[i] ^ ord(arr[j])
            arr[i] = arr[i] ^ arr[j]
            arr[i] = chr(arr[i])
            arr[j] = chr(arr[j])
            i += 1
            j -= 1
            continue
        if 'z' >= arr[i] >= 'a':
            i += 1
        if 'A' <= arr[j] <= 'Z':
            j -= 1
    return "".join(arr)

print(sort("sFSDFfsDdfsDFDdfDFDF"))
```

```
# @Filename: 求一个串中出现的第一个最长重复子串（后缀数组法）.py
"""
分析:
后缀数组法：将字符串中找重复子串问题转化为从后缀数组中通过对比相邻的两个子串的公共串的长度
后缀数组：字符串r从第i个字符开始的后缀表示为suuffix（i）
"""

class CommonSubString():
    # 找出最长的公共子串长度
    def maxPrefix(self, s1, s2):
        i = 0
        while i < len(s1) and i < len(s2):
            if s1[i] == s2[i]:
                i += 1
            else:
                return i
        return i

    def getMaxCommonStr(self, s):
        suffixes = []
        for i in range(len(s)):
            suffixes.append(s[i:])
        suffixes.sort()
        maxCommonStrLen = 0
        maxCommonStr = None
        for i in range(len(suffixes)-1):
            curMaxCommonStrLen = self.maxPrefix(suffixes[i], suffixes[i+1])
            if maxCommonStrLen < curMaxCommonStrLen:
                maxCommonStrLen = curMaxCommonStrLen
                maxCommonStr = suffixes[i][:maxCommonStrLen]
        return maxCommonStr, maxCommonStrLen
if __name__ == "__main__":
    c = CommonSubString()
    str, len = c.getMaxCommonStr("banana")
    print(str, len)
```

```
# @Filename: 求一个字符串的所有排列（递归法）.py
"""
题目：输入字符串，输出字符串的所有排列
分析：
1. 递归法
    a:固定第一个字符，对后面的数据进行全排列
    b:全排列结束后，第一个字符与后面字符进行替换
    c: 换回原来的字符串
"""

def swap(str, i, j):
    tmp = str[i]
    str[i] = str[j]
    str[j] = tmp

def Permulation(str, start):
    if str==None or start < 0:
        return
    if start == len(str)-1:
        print("".join(str))
    else:
        i = start
        while i <len(str):
            # 下面代码即可剔除重复的排列元素
            # if i != start and str[i] == str[start]:
            #     i += 1
            #     continue
            swap(str, start, i)
            Permulation(str, start+1)
            swap(str, start, i)
            i += 1

def Permulation_transe(s):
    str = list(s)
    Permulation(str, 0)

if __name__ == "__main__":
    Permulation_transe("abbc")
```

```
# @Filename: 求一个字符串的所有排列（非递归法）.py
"""
题目：输入字符串，输出字符串的所有排列
分析：
1. 非递归法
    1. 对字符串中的字符进行排序（升序）
    2. 从右往左找到第一个递增的子字符串（两个字符）
    3. 将较小的那个字符（index=pmin）与后面比它大的最小的字符进行交换
    4. 将pmin后的字符进行逆序
    5. 循环，知道找不到递增的子字符串
"""
def swap(str, i, j):
    tmp = str[i]
    str[i] = str[j]
    str[j] = tmp

def Reverse(str, begin, end):
    i = begin
    j = end
    while i < j:
        swap(str, i, j)
        i += 1
        j -= 1

def getNextPermutation(str):
    end = len(str)-1
    cur = end
    suc =0
    tmp = 0
    while cur != 0:
        suc = cur
        cur = cur - 1
        # 找到子字符串
        if str[cur] < str[suc]:
            tmp = end
            # 找到比cur大的最小字符串的字符
            while str[tmp] < str[cur]:
                tmp -= 1
            # 交换
            swap(str, cur, tmp)
            # 逆序
```

```
            Reverse(str, suc, end)
            return True
    return False

def Permulation(s):
    if s == None or len(s)< 1:
        return
    str = list(s)
    str.sort()
    print(str)
    while getNextPermutation(str):
        print(str)

if __name__ == "__main__":
    Permulation("abc")
```

```
# @Filename: 求两个字符串的最大公共子串（动态规划法）.py
"""
题目：求最长公共子串
分析：
1. 采用动态规划，记录s1[i]结尾的子串与s2[j]结尾的子串的最长公共子串
"""

def getMaxSubStr(str1, str2):
    len1 = len(str1)
    len2 = len(str2)
    sb = ""
    maxI = 0
    maxs = 0
    M = [[None]*(len1+1) for _ in range(len2+1)]
    i = 0
    # ！！！
    while i < len1+1:
        M[0][i] = 0
        i += 1
    j = 0
    while j < len2+1:
        M[j][0] = 0
        j += 1
    i = 1
    while i < len1+1:
        j = 1
        while j < len2+1:
            if str1[i-1] == str2[j-1]:
                M[j][i] = M[j-1][i-1] + 1
                if M[j][i] > maxs:
                    maxI = j
                    maxs = M[j][i]
            else:
                M[j][i] = 0
            j += 1
        i += 1
    j = maxI - maxs
    while j < maxI:
        sb += str2[j]
        j += 1
    return sb
```

```
if __name__ == "__main__":
    str1 = "abccad"
    str2 = "dgcadde"
    print(getMaxSubStr(str1, str2))
```

```
# @Filename: 求两个字符串的最大公共子串（滑动比较法）.py
"""
# 题目：求两个字符串的最大公共子串
分析：
1. 滑动比较法
    a. 保持s1不变，依次滑动s2，
    b. 记录此时的最大公共子串maxLen，最大公共子串在s1的起始位置
    c. 直到j > len(s1)-1
"""

def getMaxSubStr(s1, s2):
    len1 = len(s1)
    len2 = len(s2)
    maxLen = 0
    maxLenEnd1 = 0
    sb = ""
    i = 0
    while i < len1 + len2:
        s1begin = s2begin = 0
        tmpMaxLen = 0
        if i < len1:
            s1begin = len1 - i
        else:
            s2begin = i - len1
        j = 0
        while s1begin + j < len1 and s2begin + j < len2:
            if s1[s1begin + j] == s2[s2begin + j]:
                tmpMaxLen += 1
            else:
                if tmpMaxLen > maxLen:
                    maxLen = tmpMaxLen
                    maxLenEnd1 = s1begin + j
                else:
                    tmpMaxLen = 0
            j += 1
        if tmpMaxLen > maxLen:
            maxLen = tmpMaxLen
            maxLenEnd1 = s1begin + j
        i += 1
    i = maxLenEnd1 - maxLen
    while i < maxLenEnd1:
```

```
            sb += s1[i]
            i += 1
        return sb

if __name__ == "__main__":
    str1 = "abccade"
    str2 = "dgcadde"
    print(getMaxSubStr(str1, str2))
```

```
# @Filename: 求字符串里的最长回文子串（manacher）.py
"""
Manacher算法
分析：时间复杂度o(N),空间复杂度O（N）
1. 使用*对字符串进行填充，使得只出现一种奇数回文串的情况
1. 设置P[i]数组记录以i为中心的最长回文串的半径，未填充字符串最大回文串长度为P[i]-1
2. 四种情况
    a. 如果i没有落到P[id]回文串中，则直接对i作为中心，求回文串
    b. i落在P[id]中，找到i对应回文串的另一边：2*id-i， 如果P[2*id-i]的左边回文数覆盖范围超出P[id]覆盖范围，则P[i] = id+P[id]-i
    c. i落在P[id]中，找到i对应回文串的另一边：2*id-i， 如果P[2*id-i]的左边回文数覆盖范围刚好为P[id]覆盖范围，则P[i]的长度从P[2*id-i]开始算起，再查看是否能继续增加
    d. i落在P[id]中，找到i对应回文串的另一边：2*id-i， 如果P[2*id-i]的左边回文数覆盖范围刚好为P[id]覆盖范围,则P[i]=P[2*id-i]
3. 综合以上情况 P[i] = min(P[2*id-i], P[id]-i)
"""
class Test:
    def __int__(self):
        self.center = None
        self.palindromeLen = None

    def mins(self, a, b):
        return a if a < b else b

    def getCenter(self):
        return self.center

    def getLen(self):
        return self.palindromeLen

    def Manacher(self, strs):
        lens = len(strs)
        newLen = 2 * lens + 1
        s = [None] * newLen
        p = [None] * newLen
        id = 0
        i = 0
        for i in range(newLen):
            if i % 2 == 0:
                s[i] = '*'
            else:
                s[i] = strs[i // 2]
            p[i] = 0
```

```
        print(s)
        print(p)
        self.center = -1
        self.palindromeLen = -1
        i = 1
        while i < newLen:
            if id + p[id] > i:
                p[i] = self.mins(id + p[id] - i, p[2 * id - i])
            else:
                p[i] = 1
            while i + p[i] < newLen and i - p[i] > 0 and s[i - p[i]] == s[i + p[i]]:
                p[i] += 1
            if i + p[i] > id + p[id]:
                # 此时后面的数将进入i点的回文领域
                id = i
            if p[i] - 1 > self.palindromeLen:
                self.center = (i + 1) // 2 - 1
                self.palindromeLen = p[i] - 1
            i += 1
        print(1)

if __name__ == "__main__":
    strs = "abcbax"
    t = Test()
    t.Manacher(strs)
    # 这里还要进行处理
    if t.palindromeLen % 2 == 0:
        print("fsdf",strs[t.getCenter() - (t.getLen() // 2 + 1):t.getCenter() + (t.getLen() // 2 - 1)])
    else:
        print("ddd",strs[t.getCenter() - (t.getLen() // 2):t.getCenter() + (t.getLen() // 2)])
```

```
# @Filename: 求字符串里的最长回文子串（中心扩展法）.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
中心扩展法:
分析：时间复杂度（O（n^2））,空间复杂度（O（1））
1. 当中心为一个点
2. 当中心为两个点
"""


class Test:
    def __init__(self):
        self.startIndex = None
        self.lens = 0

    def getStartIndex(self):
        return self.startIndex

    def getLens(self):
        return self.lens

    def expandBothSide(self, strs, c1, c2):
        n = len(strs)
        while c1 >= 0 and c2 < n and strs[c1] == strs[c2]:
            c1 -= 1
            c2 += 1
        tmpStartIndex = c1 + 1
        tmpLen = c2 - c1 - 1
        if tmpLen > self.lens:
            self.lens = tmpLen
            self.startIndex = tmpStartIndex

    def getLongestPalindrome(self, strs):
        if not strs:
            return
        n = len(strs)
        for i in range(n - 1):
            # 奇数个回文数，中间只有一个点
            self.expandBothSide(strs, i, i)
            # 偶数个回文数，中间有两个点
            self.expandBothSide(strs, i, i + 1)


```

```
45. if __name__ == "__main__":
46.     strs = "cbbd"
47.     t = Test()
48.     t.getLongestPalindrome(strs)
49.     print(strs[t.getStartIndex():t.getStartIndex() + t.getLens()])
```

```
# @Filename: 求字符串里的最长回文子串（动态规划）.py
"""
# 最长回文子串
分析:
动态规划:时间复杂度O（n^2），空间复杂度O（n^2）
1.
P[i,i] = 1
如果Si != Si+1->P[i,i+1] = 0 else P[i,i+1] = 1
如果Si = Sj -> P(i, j) = P(i+1,j-1)
"""


class Test:
    def __init__(self):
        self.startIndex = None
        self.lens = None

    def getStartIndex(self):
        return self.startIndex

    def getLen(self):
        return self.lens

    def getLongestPalindrome(self, strs):
        if not strs:
            return
        n = len(strs)
        if n < 1:
            return
        self.startIndex = 0
        self.lens = 1
        historyRecord = [[0] * n for _ in range(n)]

        # 更新长度为1的回文字符串信息
        for i in range(n):
            historyRecord[i][i] = 1

        # 更新长度为2的回文字符串信息
        for i in range(n - 1):
            if strs[i] == strs[i + 1]:
                historyRecord[i][i + 1] = 1
                self.startIndex = i
                self.lens = 2

```

```
        # 从长度为3的字符串开始找起
        pLen = 3
        while pLen <= n:
            i = 0
            while i < n - pLen + 1:
                j = i + pLen - 1
                if strs[i] == strs[j] and historyRecord[i + 1][j - 1] == 1:
                    historyRecord[i][j] = 1
                    self.startIndex = i
                    self.lens = pLen
                i += 1
            pLen += 1


if __name__ == "__main__":
    strs = "babad"
    t = Test()
    t.getLongestPalindrome(strs)
    if t.getStartIndex() != -1 and t.getLen() != -1:
        print("最长的回文子串为: ", strs[t.getStartIndex():t.getStartIndex() + t.getLen()])
    else:
        print("查找失败")
```

```
# @Filename: 求解字符串中字典序最大的子序列（逆序遍历法）.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
分析:
字典序:首先找到字符串a中最大的字符，然后在该字符后面的子字符串中找到最大的字符，最终找到最后一个字符
"""


def getLargestSub(src):
    res = [src[-1]]
    i = len(src) - 2
    while i > -1:
        if src[i] >= res[0]:
            res.insert(0, src[i])
        i -= 1
    return "".join(res)


if __name__ == "__main__":
    print(getLargestSub("acbdxmng"))
```

```
# @Filename: 统计字符串中连续的重复字符个数.py
"""
分析:
1. 普通遍历法，增加两个变量maxLen和curMaxLen即可
2. 采用递归法
"""
def getMaxDupChar(s, startIndex, curMaxLen, maxLen):
    '''
    if startIndex == len(s)-1:
        return max(maxLen, curMaxLen)
    if s[startIndex] != s[startIndex + 1]:
        if maxLen < curMaxLen:
            maxLen = curMaxLen
        startIndex = startIndex + 1
        curMaxLen = 1
        return getMaxDupChar(s, startIndex, curMaxLen, maxLen)
    elif s[startIndex] == s[startIndex + 1]:
        return getMaxDupChar(s, startIndex+1, curMaxLen+1, maxLen)
    '''
    if startIndex + curMaxLen >= len(s):
        return max(maxLen, curMaxLen)
    if s[startIndex] != s[startIndex+curMaxLen]:
        if curMaxLen > maxLen:
            maxLen = curMaxLen
        startIndex += curMaxLen
        curMaxLen = 1
        return getMaxDupChar(s, startIndex, curMaxLen, maxLen)
    else:
        curMaxLen += 1
        return getMaxDupChar(s, startIndex, curMaxLen, maxLen)

if __name__ == "__main__":
    s = "aabbbbddddddddfdffffff"
    print(getMaxDupChar(s, 0, 1, 0))
```

4. 排列组合与概率

```
# @Filename: 如何判断还有几盏灯泡亮着.py
"""
判断还有几盏灯泡亮着
1. 100个灯泡排成一排，第一次将拉动所有灯泡，第二次将拉动2的倍数的灯泡，第三次将拉动3的倍数的灯泡，一次递推，拉动造成 相反的结果
2. 拉满100次后，剩下亮着的灯泡的编号

引申：如果没有拉满100次，如果拉了50次呢
那么50次以内的灯泡，按照以上判断，50次之后的灯泡需要判断是否存在组合因子统计（还要统计这样的个数）（一个在50内，一个在50外，这样的话，这个也算拉了奇数次，同时还要）
"""
def factorisOdd(a):
    i = 1
    count = 0
    while i <= a:
        if a % i == 0:
            count += 1
        i +=1
    if count % 2 == 0:
        return 0
    return 1

def totalCount(n):
    count = 0
    for i in range(1, n+1):
        if factorisOdd(i):
            count += 1
            print(i)

totalCount(100)
```

```
# @Filename: 如何拿到最多金币.py
"""
题目：10个房间有金币，每个房间只能进入一次，只能在一个房间中拿金币
策略：前四个房间只看不拿，后面的房间只要看到比前4个房间都多的金币就拿，否则就拿最后一个房间的金币
分析：
1. 首先需要生成所有10个房间的金币出现的情况
2. 模拟策略拿金币，模拟n次，最后统计n次，共拿到最多金币的次数m，最后结果为m/n
3. 考虑n的值为1000
"""
import random
def getMaxNum(count):
    # 先随机生成10个门的金币
    door = [random.randint(1, count) for _ in range(count)]
    max_4 = max(door[:4])
    i = 4
    while i < count:
        if door[i] >= max_4:
            return True
        i += 1
    return False

if __name__ == "__main__":
    n = 10000
    m = 0
    for i in range(n):
        m += int(getMaxNum(10))
    print(m/n)
```

```
# @Filename: 如何求数字的组合.py
"""
题目：给定一串数字（1,2,2,3,4,5），求出所有数字的排列组合
条件：4不能出现在第三位，3和5不能相连
"""
class Test:
    def __int__(self, arr):
        self.numbers = arr
        self.visited = [None] * len(self.numbers)
        self.graph = [[None] * len(self.numbers) for _ in range(len(self.numbers))]
        self.n = 6
        self.combination = ''
        self.s = set()

    def depthFistSearch(self, start):
        self.visited[start] = True
        self.combination += str(self.numbers[start])
        # 遍历结束
        if len(self.combination) == self.n:
            # 判断4是否出现在第三位
            if self.combination.index("4") != 2:
                self.s.add(self.combination)
                pass
        j = 0
        while j < self.n:
            if self.graph[start][j] == 1 and self.visited[j] == False:
                self.depthFistSearch(j)
            j += 1
        # 下面两行特别关键，用于恢复递归前一次的现状
        self.combination = self.combination[:-1]
        self.visited[start] = False

    def getAllCombinations(self):
        # 构造图
        i = 0
        while i < self.n:
            j = 0
            while j < self.n:
                if i == j:
                    self.graph[i][j] = 0
                else:
```

```
                    self.graph[i][j] = 1
                j += 1
            i += 1
        # 确保3和5之间不可达
        self.graph[3][5] = 0
        self.graph[5][3] = 0

        i = 0
        while i < self.n:
            self.depthFistSearch(i)
            i += 1

    def printAllCombinations(self):
        print(self.s)

if __name__ == "__main__":
    arr = [1, 2, 2, 3, 4, 5]
    t = Test(arr)
    t.getAllCombinations()
    t.printAllCombinations()
```

```
# @Filename: 如何求正整数n所有可能的整数组合.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
题目：给定一个整数n，找出所有能够和为n的整数组合，组合按照递增的形式展示
分析：典型的递归问题
1. 递归参数：sums,  result
2. 递归结束条件：当sums为0的时候，打印result的值，并返回
3. 递归方式：外层一个while，控制第一个整数的大小，将第一个整数加入result，同时减小sums的值，递归调用
"""


def getAllcombination(sums, result):
    if sums == 0:
        print(''.join(result))
        return
    # 每次都从1开始，会出现3,1这种逆序的模式，所以需要改变
    # i = 1
    i = 1 if not result else int(result[-1])
    while i <= sums:
        result.append(str(i))
        getAllcombination(sums - i, result)
        result.pop()
        i += 1


getAllcombination(4, [])
```

```
# @Filename: 如何用一个随机函数得到另一个随机函数.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
题目：有一个随机函数能1/2的概率生成0,1，怎样 通过该随机函数也生成0,1概率分别为1/4, 3/4
分析：用两个已知随机函数得到的结果进行或即可
"""

import random


def get_random():
    a1 = int(round(random.random()))
    a2 = int(round(random.random()))
    result = a1 | a2
    return result


n = 0
for i in range(100000):
    n += get_random()
print(n / 100000)
```

```
# @Filename: 如何等概率地从大小为n的数组中选取m个整数.py
"""
分析:
每一次选取之后，需要排除该数据，然后再在剩下的数据中选择才能保证概率相等
1. 第一次1/n
2. [ (n-1) /n]*[1/(n-1)] = 1/n

具体方案:
将随机选到的值与最前面的值进行交换
"""
import random
def getRandomM(a, n, m):
    i = 0
    res = []
    while i < m:
        j = random.randint(i, n - 1)
        res.append(a[j])
        a[j], a[i] = a[i], a[j]
        i += 1
    return res
print(getRandomM([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 10, 6))
```

```
# @Filename: 如何组合1, 2, 5这三个数使其和为100.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
分析:
1. 蛮力法
    a. 最终公式为x+2y+5z=100, x最大为100,  y最大为50,  z最大为20
2. 数字规律法
    a. x+5z = 100-2y,,所以x+5z为偶数
    b. 依次列举z的值（0-20），求得对应给的x与y
"""
def combinationCount(n):
    count = 0
    m = 0
    while m <= n:
        count += (n - m + 2) // 2
        x = 0 if m % 2 == 0 else 1
        while x <= n - m:
            print("{}+2x{}+5x{}={}".format(x, (n - x - m) // 2, m // 5, n))
            x += 2
        m += 5
    return count
print(combinationCount(100))
```

5. 排序

```
# @Filename: 冒泡排序.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
分析:
冒泡排序：每次都进行全部遍历一遍，比较相邻数据大小，若满足条件则交换相邻数据
1. 是一种稳定的排序方法
2. T：最好的情况下时间复杂度为O（n），平均和最坏的情况下时间复杂度为O（n^2），空间复杂度为O（1）：交换空间
"""


def bubble_sort(arr):
    for i in range(len(arr) - 1):
        for j in range(1, len(arr) - i):
            if arr[j - 1] > arr[j]:
                arr[j - 1], arr[j] = arr[j], arr[j - 1]


arr = [3, 4, 2, 8, 9, 5, 1]
bubble_sort(arr)
print(arr)
```

```
# @Filename: 基数排序.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
分析:
1. 根据个位数把数字分配到0-9变好的桶中
2. 将这些树串联起来
3. 接着再对十位数、百位数分配
"""

import math


def radix_sort(arr, radix=10):
    # 计算最大数据的最多位数
    k = int(math.ceil(math.log(max(arr), radix)))
    bucket = [[] for _ in range(radix)]
    for i in range(1, k + 1):
        for j in arr:
            bucket[j // (radix ** (i - 1)) % (radix)].append(j)
        del arr[:]
        for z in bucket:
            arr += z
            del z[:]
    return arr
```

```
# @Filename: 堆排序.py
"""
分析:
不稳定的排序方法
T: 最坏情况也是O (nlogn)
"""

def adjust_heap(arr, i, size):
    while True:
        maxs = i
        lchild = 2 * i + 1
        rchild = 2 * i + 2
        if i < size // 2:
            if lchild < size and arr[lchild] > arr[maxs]:
                maxs = lchild
            if rchild < size and arr[rchild] > arr[maxs]:
                maxs = rchild
            if maxs != i:
                arr[maxs], arr[i] = arr[i], arr[maxs]
                i = maxs
            else:
                break
        else:
            break

def heap_sort(arr):
    first = len(arr) // 2 - 1
    for i in range(first, -1, -1):
        print(arr[i])
        adjust_heap(arr, i, len(arr) - 1)
    # 交换堆顶和堆尾
    for head_end in range(len(arr)-1, 0, -1):
        arr[head_end], arr[0] = arr[0], arr[head_end]
        # 由于前面已经调整为大顶堆，所以这里只需对交换头尾之后的对再进行一次调整即可
        adjust_heap(arr, 0, head_end-1)

if __name__ == "__main__":
    arr = [3, 4, 2, 8, 9, 5, 1]
    print(arr)
    heap_sort(arr)
    print(arr)
```

```python
# @Filename: 堆排序.py
""
分析:
不稳定的排序方法
T: 最坏情况也是O (nlogn)
"""
def adjust_heap(arr, i, size):
    while True:
        maxs = i
        lchild = 2 * i + 1
        rchild = 2 * i + 2
        if i < size // 2:
            if lchild < size and arr[lchild] > arr[maxs]:
                maxs = lchild
            if rchild < size and arr[rchild] > arr[maxs]:
                maxs = rchild
            if maxs != i:
                arr[maxs], arr[i] = arr[i], arr[maxs]
                i = maxs
            else:
                break
        else:
            break

def heap_sort(arr):
    first = len(arr) // 2 - 1
    for i in range(first, -1, -1):
        print(arr[i])
        adjust_heap(arr, i, len(arr) - 1)
    # 交换堆顶和堆尾
    for head_end in range(len(arr)-1, 0, -1):
        arr[head_end], arr[0] = arr[0], arr[head_end]
        # 由于前面已经调整为大顶堆，所以这里只需对交换头尾之后的对再进行一次调整即可
        adjust_heap(arr, 0, head_end-1)

if __name__ == "__main__":
    arr = [3, 4, 2, 8, 9, 5, 1]
    print(arr)
    heap_sort(arr)
    print(arr)
```

```
# @Filename: 归并排序.py
"""
分析:
归并排序:
1. 首先将两个相邻的长度为1的子序列进行归并，得到n/2个长度为2或1的有序子序列
2. 将其两两归并
3. 重复1和2直到得到一个有序序列为止

是一种稳定的算法， T: O (nlogn)  S:O(n)
"""
def merge(left, right):
    i, j = 0, 0
    result = []
    while i < len(left) and j < len(right):
        # print(left[i], right[j], right)
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # 添加某边还剩余的部分
    result += left[i:]
    result += right[j:]
    print(result)
    return result

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    num = len(arr) // 2
    left = merge_sort(arr[:num])
    right = merge_sort(arr[num:])
    return merge(left, right)

arr = [3, 4, 2, 8, 9, 5, 1]
print(merge_sort(arr))
```

```
# @Filename: 快速排序.py
"""
# 分析:
1. 设置两个变量left, right, 最开始初始化left=0, right= n-1
2. 设置关键值key = arr[left]
3. 从right开始往左遍历，如果arr[right] < key,那么将right和left的数据进行交换
4. 从left开始往右遍历，如果arr[left] > key, 那么将left和right的数据进行交换
5. 以上步骤知道left == right

6. 1-5后，会得到两组数据，对每组数据递归调用1-5

*: 当初始的序列整体或者部分有序时，快排性能降低，退化为冒泡排序
T: 最坏情况下O（n^2）， 最好情况O（nlogn），平均O（nlogn），S（logn）

*: 不稳定的排序算法
"""
def quick_sort(arr, left, right):
    if left < right:
        i, j = left, right
        key = arr[i]

        while i < j:
            # 循环从右边遍历
            while i < j and arr[j] >= key:
                j -= 1
            arr[i] = arr[j]
            # 循环从左边遍历
            while i < j and arr[i] <= key:
                i += 1
            arr[j] = arr[i]
        # 将key赋予中间值
        arr[i] = key

        quick_sort(arr, left, i - 1)
        quick_sort(arr, i + 1, right)

arr = [3, 4, 2, 8, 9, 5, 1]
quick_sort(arr, 0, len(arr)-1)
print(arr)
```

```
# @Filename: 快速排序.py
"""
# 分析:
1. 设置两个变量left, right, 最开始初始化left=0, right= n-1
2. 设置关键值key = arr[left]
3. 从right开始往左遍历，如果arr[right] < key,那么将right和left的数据进行交换
4. 从left开始往右遍历，如果arr[left] > key, 那么将left和right的数据进行交换
5. 以上步骤知道left == right

6. 1-5后，会得到两组数据，对每组数据递归调用1-5

*: 当初始的序列整体或者部分有序时，快排性能降低，退化为冒泡排序
T: 最坏情况下O（n^2）， 最好情况O（nlogn），平均O（nlogn），S（logn）

*: 不稳定的排序算法
"""
def quick_sort(arr, left, right):
    if left < right:
        i, j = left, right
        key = arr[i]

        while i < j:
            # 循环从右边遍历
            while i < j and arr[j] >= key:
                j -= 1
            arr[i] = arr[j]
            # 循环从左边遍历
            while i < j and arr[i] <= key:
                i += 1
            arr[j] = arr[i]
        # 将key赋予中间值
        arr[i] = key

        quick_sort(arr, left, i - 1)
        quick_sort(arr, i + 1, right)

arr = [3, 4, 2, 8, 9, 5, 1]
quick_sort(arr, 0, len(arr)-1)
print(arr)
```

```
# @Filename: 选择排序.py
"""
分析:
选择排序，每次选择最大或最小的一个数与前面的数进行交换
1. 是一种不稳定的排序方法， 如: 5,8,5,2,9， 第一次选择2，将与第一个5交换，这里就破坏了稳定性了
2. T: 无论好坏都是O (n^2) , S: O (1) :min
"""
def select_sort(arr):
    i = 0
    while i < len(arr) - 1:
        min = i
        j = i + 1
        while j < len(arr):
            if arr[j] < arr[min]:
                min = j
            j += 1
        # 这里是选择后面最小的那一个数据，与最前面的数据进行交换
        arr[i], arr[min] = arr[min], arr[i]
        i += 1
        print("第{}次排序: {}".format(i, arr))

arr = [3, 4, 2, 8, 9, 5, 1]
select_sort(arr)
print(arr)
```

6. 数组

```
# @Filename: 从三个有序数组中找出他们的公共元素.py
"""
数组为递增数组
分析：分别设定i,j,k进行三个数组的遍历
1. 如果a[i]< b[j],则i++, 直到a[i]>=a[j]
2. 如果a[j] < b[i],则j++, 直到b[j]>=a[i]
3. 如果两者相等，和c[k]比较,如果相等，记录并i++，如果不相等，k++直到>=a[i],
"""
def findCommon(ar1, ar2, ar3):
    i = j = k = 0
    common = []
    while i < len(ar1) and j < len(ar2) and k < len(ar3):
        if ar1[i] < ar2[j]:
            i += 1
        elif ar1[i] > ar2[j]:
            j += 1
        else:
            k += 1
            if ar1[i] == ar2[j] == ar3[k]:
                common.append(ar1[i])
                i += 1
    return common
```

```python
# @Filename: 判断请求能否在给定的存储条件下完成.py
def swap(arr, i, j):
    tmp = arr[i]
    arr[i] = arr[j]
    arr[j] = tmp

def bubbleSort(R, O):
    # 冒泡法，由大到小排序
    lens = len(R)
    i = 0
    while i < lens - 1:
        j = lens - 1
        while j > i:
            if R[j] - O[j] > R[j - 1] - O[j - 1]:
                swap(R, j, j - 1)
                swap(O, j, j - 1)
            j -= 1
        i += 1

def schedule(R, O, M):
    # 先按照R-O的大小对R和O就行排序
    bubbleSort(R, O)
    # 判断按照当前顺序是否能填充满
    total = 0
    i = 0
    while i < len(R):
        total += R[i]
        if total > M:
            return False
        else:
            total -= R[i]
            total += O[i]
        i += 1
    return True

if __name__ == "__main__":
    R = [10, 15, 23, 20, 6, 9, 7, 16]
    O = [2, 7, 8, 4, 5, 8, 6, 8]
    N = 8
    M = 50
    scheduleResult = schedule(R, O, M)
    print(scheduleResult)
```

```
    print(dict(zip(R, O)))
```

```
# @Filename: 在不排序的情况下求数组中的中位数.py
# 中位数：奇数个数据的数组，中位数k = arr((e-s)/2)； 偶数个数据的数组，中位数k = arr((e-s)/2) + arr((e-s)/2 +1)
"""
分析：使用快排的思想：左边的数据比右边的数据小，我们只需关注一边，知道找到第k大的值即可
"""
def partition(arr, low, high):
    key = arr[low]
    while low < high:
        while low < high and arr[high] > key:
            high -= 1
        arr[low] = arr[high]
        while low < high and arr[low] < key:
            low += 1
        arr[high] = arr[low]
    arr[low] = key
    return low

def getMid(arr):
    le = len(arr)
    k = (le - 1) // 2

    low = 0
    high = le - 1
    while True:
        # 将数组分为两半
        pos = partition(arr, low, high)
        if pos + 1 == k:
            break
        elif pos + 1 < k:
            low = pos + 1
        else:
            high = pos - 1
    return arr[k] if le % 2 == 1 else (arr[k] + arr[k + 1]) / 2

if __name__ == "__main__":
    arr = [7, 5, 3, 1, 11, 9]
    print(getMid(arr))
```

```
# @Filename: 在有规律的二维数组中进行高效的数据查找.py
"""
描述：一个二维数组，每一行都是递增顺序，每一列也是递增顺序，给定一个这样的arr和值k，找到k是否存在且所在位置
分析:
1. 从右上角开始遍历，如果当前值大于k，则这一列没必要继续访问，j--，如果小于k，这一行没必要访问,i++,否则相等，找到位置
"""
def findWithBinary(arr, data):
    if not arr:
        return False
    row = 0
    col = len(arr[0])-1
    while col >= 0 and row < len(arr):
        if arr[row][col] == data:
            return True
        elif arr[row][col] > data:
            col -= 1
        elif arr[row][col] < data:
            row += 1
    return False

if __name__ == "__main__":
    arr = [[0, 1, 2, 3, 4], [10, 11, 12, 13, 14]]
    print(findWithBinary(arr, 10))
```

```
# @Filename: 如何对任务进行调度.py
"""
题目：有n个相同的任务，有m个不同的服务器，每个服务器完成一个任务的时间t[i]，求一个任务分配，求出完成n个任务，最短的时间
分析:
1. 贪心法
"""
def calculate_process_time(t, n):
    if t == None:
        return None
    m = len(t)
    proTime = [0] * m
    i = 0
    while i < n:
        minTime = proTime[0] + t[0]
        minIndex = 0
        j = 1
        while j < m:
            if minTime > proTime[j] + t[j]:
                minTime = proTime[j] + t[j]
                minIndex = j
            j += 1
        proTime[minIndex] += t[minIndex]
        i += 1
    return proTime

if __name__ == "__main__":
    t = [7, 10]
    n = 6
    proTime = calculate_process_time(t, n)
    print(proTime)
```

```
# @Filename: 如何对数组进行循环位移.py
# 要求：时间复杂度O（N），空间复杂度为O（1）：允许使用两个附加变量
# 和之间旋转数组的实现一样的
"""
1. 循环移动法
2. 空间换时间法：先将k到-1之间的数据放入tmp数组中，然后将0到k之间的数据放入tmp中，这里注意k有可能大于n，此时和移动n-k次结果一样
3. 翻转法：分为A 和B ，先翻转A， 在翻转B， 最后翻转A+B
"""
def reverse(arr, start, end):
    # python 切片方便字符或者数组的操作，如果不用切片的话，就要自己建立循环和辅助变量，依次交换前后顺序达到逆序的功能
    arr[start:end] = arr[start:end][::-1]

def rightShift(arr, k):
    if not arr:
        return
    lens = len(arr)
    reverse(arr, 0, lens-k)
    reverse(arr, lens-k, lens)
    reverse(arr, 0, lens)

if __name__ == "__main__":
    k = 4
    arr = [1, 2, 3, 4, 5, 6, 7, 8]
    rightShift(arr, k)
    print(arr)
```

```
# @Filename: 如何对磁盘分区.py
"""
题目：M个磁盘，磁盘大小D[i]， m个分区，分区大小P[j]， *顺序分配*
分析:
依次比较即可
"""
def is_allocate(d, p):
    i = 0
    j = 0
    while i < len(d):
        if d[i] >= p[j]:
            remain = d[i] - p[j]
            j += 1
            if j == len(p):
                return True
        else:
            i += 1
            continue
        while remain >= p[j]:
            j += 1
            if j == len(p):
                return True
            remain = remain - p[j]
        i += 1
    return False

if __name__ == "__main__":
    d = [120, 120, 120]
    p = [60, 80, 80, 20, 80]
    print(is_allocate(d, p))
```

```
# @Filename: 如何寻找最多的覆盖点.py
# 描述：长为L的木棍，一个坐标轴数据的数组，求L能包含最多坐标轴多少个点
"""
分析：
1. 一次加入点，直到长度大于L，回退j--，记录此时长度，同时i++ j++
"""
def maxCover(arr, L):
    maxCount = 1
    lens = len(arr)
    i = 0
    j = 1
    while j < lens:
        dis_sum = arr[j] - arr[i]
        if dis_sum > L:
            j -= 1
            if j - i + 1 > maxCount:
                maxCount = j - i + 1
            i += 1
            j += 1
        j += 1
    return maxCount

if __name__ == "__main__":
    a = [1, 3, 7, 8, 10, 11, 12, 13, 15, 16, 17, 19, 25]
    print(maxCover(a, 8))
```

```
# @Filename: 如何求两个有序集合的交集.py
"""
题目：两个有序集合，集合中的元素为一段范围 {[4,8], [9,13]}
分析：
1. 不利用有序集合这一特性，进行两个循环遍历
2. 利用有序集合特性
    a. 对于a[i]与b[j]
        1. a[i][0] < b[j][0] and a[i][1] < b[j][0]：不挨着，只有i++有可能挨着
        1. a[i][0] < b[j][0] and b[j][0] <= a[i][1] < b[j][1]：挨着，记录交集，后面i++有可能挨着
        1. b[j][0] < a[i][0] < b[j][1] and b[j][0] < a[i][1] < b[j][0]:挨着，记录交集，交集为a[i]，后面i++有可能挨着
        1. b[j][0] < a[i][0] <= b[j][1] and a[i][1] > b[j][1]:挨着，记录交集，后面j++有可能挨着
        1. b[j][0] < a[i][0]:不挨着，后面j++有可能挨着
        1. a[i][0] < b[j][0] and a[i][1] > b[j][0]:挨着，记录交集，后面j++有可能挨着
"""

def getIntersection(a, b):
    result = []
    i = j = 0
    while j < len(b) and i < len(a):
        if a[i][0] < b[j][0] and a[i][1] < b[j][0]:
            i += 1
        elif a[i][0] < b[j][0] and b[j][0] <= a[i][1] < b[j][1]:
            result.append([b[j][0], a[i][1]])
            i += 1
        elif b[j][0] <= a[i][0] < b[j][1] and b[j][0] < a[i][1] < b[j][1]:
            result.append([a[i][0], a[i][1]])
            i += 1
        elif b[j][0] < a[i][0] <= b[j][1] and a[i][1] > b[j][1]:
            result.append([a[i][0], b[j][1]])
            j += 1
        elif b[j][0] < a[i][0]:
            j += 1
        else:
            result.append([b[j][0], b[j][1]])
            j += 1
    return result

if __name__ == "__main__":
    l1 = [[4, 8], [9, 13]]
```

```
    l2 = [[6, 12]]
    result = getIntersection(l1, l2)
    i = 0
    print(result)
```

```
# @Filename: 如何求集合的所有子集.py
"""
迭代法:
1. set1中首先放入一个元素
2. 在加入一个元素，更新set1（1. 在set1中在原来元素中增加当前元素的组合， 2. 增加自身元素）
3. 循环操作
"""

def getAllSub(str):
    if not str:
        print("参数不合理")
        return None
    arr = []
    # 将第一个节点加入集合中
    arr.append(str[0])
    i = 1
    while i < len(str):
        lens = len(arr)
        j = 0
        # 迭代轮次，添加构造组合
        while j < lens:
            arr.append(arr[j] + str[i])
            j += 1
        arr.append(str[i])
        i += 1
    return arr

if __name__ == "__main__":
    result = getAllSub("abc")
    print(result)
```

```
# @Filename: 如何获得最好的矩阵链相乘方法（动态规划-重点）.py
"""
题目：求解多矩阵相乘，如何运用结合律实现计算数量最小化
分析:
1. 使用递归法
2. 使用动态规划
分析：建立一个矩阵（n*n），记录每一个组合的乘积次数最小值
"""
def MatrixChainOrder(p, n):
    cost = [[None] * n for _ in range(n)]
    i = 1
    while i < n:
        # 本身到本身的计算次数为0
        cost[i][i] = 0
        i += 1
    cLen = 2
    while cLen < n:
        i = 1
        while i < n - cLen + 1:
            j = i + cLen - 1
            cost[i][j] = 2 ** 31
            k = i
            while k < j:
                q = cost[i][k] + cost[k + 1][j] + p[i - 1] * p[k] * p[j]
                if q < cost[i][j]:
                    cost[i][j] = q
                k += 1
            i += 1
        cLen += 1
    return cost[1][n - 1]

if __name__ == "__main__":
    arr = [1, 5, 2, 4, 6]
    n = len(arr)
    print(str(MatrixChainOrder(arr, n)))
```

```
# @Filename: 如何获得最好的矩阵链相乘方法（递归法）.py
"""
题目：求解多矩阵相乘，如何运用结合律实现计算数量最小化
分析：
1. 使用递归法
"""
def MatrixChainOrder(p, i, j):
    # p为数组， i和j为括号
    if i == j:
        return 0
    mins = 2 ** 32
    k = i
    while k < j:
        count = MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k + 1, j) + p[i - 1] * p[k] * p[j]
        if count < mins:
            mins = count
        k += 1
    return mins

if __name__ == "__main__":
    arr = [1, 5, 2, 4, 6]
    n = len(arr)
    print(str(MatrixChainOrder(arr, 1, n-1)))
```

```
# @Filename: 对二维数组进行旋转.py
"""
合理设置起始标记大小，确定起始点后，内部循环col++ & row++进行打印
"""
def rotateArr(arr):
    le = len(arr[0])
    for i in range(len(arr[0])):
        col = le - i
        row = 0
        while col < le:
            print(arr[row][col], end=" ")
            col += 1
            row += 1
        print()
    le1 = len(arr)
    for i in range(le1):
        row = 0 + i
        col = 0
        while row < le1:
            print(arr[row][col], end=" ")
            col += 1
            row += 1
        print()

if __name__ == "__main__":
    arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
    rotateArr(arr)
```

```
# @Filename: 对大量有重复的数字的数组排序（字典法）.py
"""
题目：对有大量重复数字的数组排序
分析：
1. 利用AVL树
2. 利用hash法
"""

def sort(arr):
    num_count = {}
    for i in arr:
        if i in num_count:
            num_count[i] += 1
        else:
            num_count[i] = 1
    keys = sorted(num_count.keys(), key=lambda i: num_count[i])
    i = 0
    k = 0
    while i < len(arr):
        j = 0
        while j < num_count[keys[k]]:
            arr[i + j] = keys[k]
            j += 1
        k += 1
        i += j
    print(arr)
    return arr

if __name__ == "__main__":
    arr = [15, 12, 15, 2, 2, 12, 2, 3, 12, 100, 3, 3]
    sort(arr)
```

```python
# @Filename: 找出数组中出现一次的数（三个数是唯一的，只需输出一个数即可）.py
# 和找出出现奇数次的数（两个奇数）差不多
"""
1. 所有值进行异或得到d，通过找到一个位分为两组，三个奇数数中分开，变为两组，然后求得一个组中的数，
2. 得到这个数后，后面还能继续求解剩下的数据
"""
def isOne(n, i):
    return (n >> i & 1) == 1


def findSingle(arr):
    d = 0
    for i in arr:
        d ^= i
    i = 0
    while d >> i:
        arr1 = 0
        arr2 = 0
        len_1 = 0
        len_2 = 0
        if isOne(d, i):
            # 对原数组分为两组
            for j in arr:
                if j >> i & 1 == 1:
                    arr1 ^= j
                    len_1 += 1
                else:
                    arr2 ^= j
                    len_2 += 1
            if len_2 % 2 == 0 and arr2 != 0:
                return arr1
            elif len_1 % 2 == 0 and arr1 != 0:
                return arr2
            else:
                i += 1
        else:
            i += 1


if __name__ == "__main__":
    arr = [6, 3, 4, 5, 9, 4, 3]
    print(findSingle(arr))
```

```
# @Filename: 找出数组中出现奇数次的数（只有两个奇数）.py
# 前提：只有两个元素出现奇数次
# 1. 词典法：记录出现次数
# 2. 异或法：所有数据全部异或，得到出现奇数次的数的异或结果c，找到c中不为0的某一位数n,将c与数组中n位为1的数异或，得到a或者b， 然后c^a = b
def get2Num(arr):
    # 首先全部异或
    c = 0
    for i in arr:
        c ^= i
    # 找到c中不为0的位置
    position = 0
    tmpResult = c
    while tmpResult & 1 == 0:
        position += 1
        tmpResult = tmpResult >> 1
    a = c
    for i in arr:
        if i >> position & 1 == 1:
            a ^= i
    b = c ^ a
    return a, b

if __name__ == "__main__":
    arr = [3, 5, 6, 6, 5, 7, 2, 2]
    print(get2Num(arr))
```

```
# @Filename: 找出数组中唯一重复元素2（累加求和法）.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 这里的前提是，数组里面的数据是连续的
# 时间复杂度为O（N），  空间复杂度为O（1）


def findDup(arr):
    mi = min(arr)
    ma = max(arr)
    return sum(arr) - sum([i for i in range(mi, ma + 1)])


if __name__ == "__main__":
    arr = [1, 3, 4, 5, 2, 5, 6]
    print(findDup(arr))
```

```
# @Filename: 找出数组中唯一重复元素3（异或法）.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 同样建立在数据是单一连续的基础上
# time:O(n)  space:O(1)

def findDup(arr):
    mi = min(arr)
    ma = max(arr)
    l = ma-mi+1
    res = 0
    for i in range(len(arr)):
        res ^= arr[i]
    for i in range(l):
        res ^= (mi+i)
    return res

if __name__ == "__main__":
    arr = [1, 3, 4, 5, 2, 5, 6]
    print(findDup(arr))
```

```
# @Filename: 找出数组中唯一重复元素4（数据映射法）.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 通过元素值作为下标访问后续节点，同时将当前节点的值设置为相反数
# 采用剑指offer效果更好

def findDup(arr):
    index = 0
    while 1:
        if arr[index] > 0:
            arr[index] *= -1
            index = -arr[index]
        else:
            return -arr[index]


if __name__ == "__main__":
    arr = [1, 3, 4, 5, 2, 2, 6]
    print(findDup(arr))
```

```
# @Filename: 找出数组中唯一重复的元素1（字典法）.py
# @Software: PyCharm
# @e_mail:   sancica@163.com

# 采用dict存储数据，如果当前数据已经在dict中，则该数据为重复数据
# 时间复杂度 ：O（N），  空间复杂度O（N）

def findDup(arr):
    dic = {}
    for i in arr:
        if not dic.get(i, 0):
            dic[i] = 1
        else:
            return i


if __name__ == "__main__":
    arr = [1, 3, 4, 1, 2, 5, 6]
    print(findDup(arr))
```

```python
# @Filename: 找出数组中的最大值和最小值（分治一次）.py
# 采用分治法，将相邻数据分为一组，将每一组的数组小的放在左边，大的放在右边
# 分析：仅仅采用一次分治，但实际上左右可以继续采用分治的思想，利用递归的方法
class MaxMin:
    def __init__(self):
        self.max = None
        self.min = None

    def getMax(self):
        return self.max

    def getMin(self):
        return self.min

    def getMaxAndMin(self, arr):
        if not arr:
            print("参数不合法")
            return
        i = 0
        lens = len(arr)
        while i < lens-1:
            if arr[i] > arr[i + 1]:
                tmp = arr[i]
                arr[i] = arr[i + 1]
                arr[i + 1] = tmp
            i += 2
        i = 1
        self.max = arr[1]
        while i < lens:
            if arr[i] > self.max:
                self.max = arr[i]
            i += 2
        i = 0
        self.min = arr[0]
        while i < lens:
            if arr[i] < self.min:
                self.min = arr[i]
            i += 2
if __name__ == "__main__":
    arr = [7, 3, 19, 40, 4, 7, 1]
    m = MaxMin()
    m.getMaxAndMin(arr)
```

```
    print(m.getMax())
    print(m.getMin())
```

```python
# @Filename: 找出数组中的最大值和最小值（分治递归）.py
class MaxMin:
    def __init__(self):
        self.max = None
        self.min = None

    def getMax(self):
        return self.max

    def getMin(self):
        return self.min

    def getMaxAndMin(self, arr, l, r):
        if not arr:
            print("参数不合法")
            return
        list = []
        m = (l + r) // 2
        if l == r:  # l与r之间只存在一个点
            list.append(arr[l])
            list.append(arr[r])
            return list
        if l + 1 == r:  # 之间存在两个点
            list.append(arr[l] if arr[l] <= arr[r] else arr[r])
            list.append(arr[r] if arr[l] < arr[r] else arr[l])
            return list
        lList = self.getMaxAndMin(arr, l, m)
        rList = self.getMaxAndMin(arr, m + 1, r)
        max = lList[1] if lList[1] > rList[1] else rList[1]
        min = lList[0] if lList[0] < rList[0] else rList[0]
        list.append(min)
        list.append(max)
        return list

if __name__ == "__main__":
    arr = [7, 3, 19, 40, 4, 7, 1]
    m = MaxMin()
    result = m.getMaxAndMin(arr, 0, len(arr) - 1)
    print(result[0])
    print(result[1])
```

```
# @Filename: 找出数组中第k小的数.py
"""
1. 排序法：排好序之后，取下标为k-1的数据即可 O（nlog2n）
2. 部分排序法：对选择排序改造：首先找到最小，然后找第二小，知道找到第k小O(k*n)
3. 类似快速排序方法
"""
def findSmallK(arr, low, high, k):
    i = low
    j = high
    splitElem = arr[i]
    # 将数据分为三部分，左边小于splitElem，中间arr[i]=splitElem，右边大于splitElem
    while i < j:
        while i < j and arr[j] >= splitElem:
            j -= 1
        if i < j:
            arr[i] = arr[j]
            i += 1
        while i < j and arr[i] <= splitElem:
            i += 1
        if i < j:
            arr[j] = arr[i]
            j -= 1
    arr[i] = splitElem
    subArrayIndex = i - low
    if subArrayIndex == k - 1:
        return arr[i]
    elif subArrayIndex > k - 1:
        # 第k小的数依然在左边
        return findSmallK(arr, low, i - 1, k)
    else:
        # 前面已经有i-low+1个小于第k小的数据了，所以剩下k - (i-low+1)
        return findSmallK(arr, i + 1, high, k-(i-low)-1)
if __name__ == "__main__":
    k = 3
    arr = [4, 0, 1, 0, 2, 3]
    print(findSmallK(arr, 0, len(arr) - 1, k))
```

```
# @Filename: 找出数组中绝对值最小的数.py
"""
前提：升序数组
采用二分法查找：
1. arr[mid] = 0,
2. arr[mid]<0 arr[mid+1]>0: min(-arr[mid], arr[mid+1)
3. 如果arr[mid-1]<0 arr[mid]>0: min(-arr[mid-1], arr[mid)
4. arr[mid]>0, 最小值在左边
5. arr[mid]<0, 最小值在右边
"""
def findMin(arr):
    if not arr:
        print("输入参数不合理")
        return 0
    if arr[0] > 0:
        return arr[0]
    if arr[-1] < 0:
        return arr[-1]
    begin = 0
    end = len(arr) - 1
    while True:
        mid = begin + (end - begin) // 2
        if arr[mid] == 0:
            return 0
        elif arr[mid] > 0:
            if arr[mid - 1] > 0:
                end = mid - 1
            elif arr[mid] == 0:
                return 0
            else:
                return min(-arr[mid - 1], arr[mid])
        elif arr[mid] < 0:
            if arr[mid + 1] < 0:
                begin = mid + 1
            elif arr[mid + 1] == 0:
                return 0
            else:
                return min(-arr[mid], arr[mid + 1])
if __name__ == "__main__":
    arr = [-10, -5, -2, 7, 15, 50]
    print(findMin(arr))
```

```
# @Filename: 找出数组连续最大和（动态规划-经典）.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
"""
经典的动态规划题：
当访问到第i个节点时：
1. 最大连续和子数组包含该点 if End[i-1]+arr[i]> arr[i];  End[i] =  End[i-1]+arr[i];  All[i] = max(End[i], All[i-1])
2. 最大连续和子数组不包含该点 if End[i-1]+ arr[i] <= arr[i]; End[i] = arr[i];  All[i] = max(End[i], All[i-1])
"""
def maxSubarr(arr):
    if not arr:
        return 0
    # 负责记录连续的子数组的和的值
    nEnd = arr[0]
    # 负责记录最大的子数组的和的值
    nAll = arr[0]
    for i in range(1, len(arr)):
        if nEnd + arr[i] < arr[i]:
            nEnd = arr[i]
        else:
            nEnd += arr[i]
        nAll = max(nAll, nEnd)
    return nAll
```

```
# @Filename: 找出旋转数组中的最小值.py
# 把一个有序数组最开始的若干个元素搬到数组的末尾，成为数组的旋转
# 两个指针 low  high, mid = (high+low)//2
# 由于旋转数组的特性，采用左右递归的方式遍历，直到arr[mid]与arr[mid-1] arr[mid+1]之间满足有一定的大小关系
# 分析：二分查找T：O（log2n），但在满足特殊情况是，需要全部遍历一遍O（N）
def getMin(arr, low, high):
    if high < low:
        # ???,这里貌似是没有必要判断的
        return arr[0]
    if high == low:
        return arr[low]
    # mid = (low+high)/2
    # 防止low+high溢出
    mid = low + ((high - low) >> 1)

    if arr[mid] > arr[mid + 1]:
        return arr[mid + 1]
    # 这里还是要对应一种特殊情况，即mid为0的时候，但恰好这里能返回正确值
    elif arr[mid - 1] > arr[mid]:
        return arr[mid]
    elif arr[high] > arr[mid]:
        # 最小值在左半边
        return getMin(arr[:mid], low, mid - 1)
    elif arr[low] < arr[mid]:
        return getMin(arr[mid + 1:], mid + 1, high)
    else:
        # 对于一些特殊情况，如[2,2,2,1,2,2]和[2,1,2,2,2,2,2]，无法确定是在左半边还是右半边
        return min(getMin(arr[:mid], low, mid - 1), getMin(arr[mid + 1:], mid + 1, high))
if __name__ == "__main__":
    # array1 = [5, 6, 1, 2, 3, 4]
    # mins = getMin(array1, 0, len(array1) - 1)
    # print(mins)
    # array2 = [1, 1, 0, 1]
    # print(getMin(array2, 0, len(array2) - 1))
    print(getMin([1, 2], 0, 1))
```

```
# @Filename: 按照要求构造新的数组.py
# 给定数组a[N], 获得b[N],b中的每一个元素都是a中除了当前下标元素的乘积
"""
分析:
1. 首先遍历一遍a,将连乘数据错开放入b中，如b[0] = 1, b[1] = a[0], b[2]=a[0]*a[1]。。。
2. 再次逆序遍历一遍b，将连乘数据错开与对应b数据相乘，如b[-1] = b[-1]*1  b[-2] = b[-2]*a[-1], b[-3] = b[-1]*b[-2]

"""
def calculate(a, b):
    b[0] = 1
    multi = 1
    for i in range(1, len(a)):
        multi *= a[i - 1]
        b[i] = multi
    multi = 1
    for i in range(len(a) - 2, -1, -1):
        print(i)
        multi *= a[i+1]
        b[i] *= multi

if __name__ == "__main__":
    a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    b = [None] * len(a)
    calculate(a, b)
    print(b)
```

```
# @Filename: 旋转数组的实现方式.py
# 分析：遍历了两次数组T:O(N), S:O(1)
def swap(arr, low, high):
    while low < high:
        tmp = arr[low]
        arr[low] = arr[high]
        arr[high] = tmp
        low += 1
        high -= 1

def rotate(arr, div):
    # 采用inspace的修改方法，不需要返回数组
    if not arr or div < 0 or div >= len(arr):
        return
    if div == 0 or div == len(arr):
        return
    swap(arr, 0, div)
    swap(arr, div + 1, len(arr) - 1)
    swap(arr, 0, len(arr) - 1)

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5]
    rotate(arr, 2)
    print(arr)
```

```
# @Filename: 求数组中两个元素的最小距离.py
# 采用动态规划，利用两个变量记录最近访问到的num1和num2
def minDistance(arr, num1, num2):
    lastPos1 = None
    lastPos2 = None
    min_dis = len(arr)
    for i in range(len(arr)):
        if arr[i] == num1:
            lastPos1 = i
            if lastPos2:
                # min_dis = min_dis if abs(lastPos2 - lastPos1) > min_dis else abs(lastPos2 - lastPos1)
                min_dis = min(min_dis, lastPos1 - lastPos2)
        if arr[i] == num2:
            lastPos2 = i
            if lastPos1:
                # min_dis = min_dis if abs(lastPos2 - lastPos1) > min_dis else abs(lastPos2 - lastPos1)
                min_dis = min(min_dis, lastPos2 - lastPos1)
    return min_dis

if __name__ == "__main__":
    arr = [4, 5, 6, 4, 7, 4, 6, 4, 7, 8, 5, 6, 4, 3, 10, 8]
    num1 = 4
    num2 = 8
    print(minDistance(arr, num1, num2))
```

```
# @Filename: 求解最小三元组距离.py
"""
1. 蛮力法：三个循环依次记录
2. 最小距离法
"""
def mins(a, b, c):
    mins = min(a, b, c)
    return mins

def maxes(a, b, c):
    return max(a, b, c)

def minDistance(a, b, c):
    aLen = len(a)
    blen = len(b)
    clen = len(c)
    curDist = 0
    minsd = 0
    minDist = 2 ** 32
    i = 0
    j = 0
    k = 0
    while True:
        # 计算得到最短距离
        curDist = maxes(a[i], b[j], c[k]) - mins(a[i], b[j], c[k])
        if curDist < minDist:
            minDist = curDist
        # 获取当前最小值的那一个数组，并在该数组中进行移动操作，只有在该数组中进行操作才能降低距离
        minsd = mins(a[i], b[j], c[k])
        if minsd == a[i]:
            i += 1
            if i >= aLen:
                break
        elif minsd == b[j]:
            j += 1
            if j >= aLen:
                break
        else:
            k += 1
            if k >= clen:
                break
```

```
    return minDist

if __name__ == "__main__":
    a = [3, 4, 5, 7, 15]
    b = [10, 12, 14, 16, 17]
    c = [20, 21, 23, 24, 37, 30]
    print(minDistance(a,b,c))
```

```
# @Filename: 求解迷宫问题.py
"""
题目：从一个矩阵的左上角走到右下角，其中1表示路径，0表示不通
分析：采用递归法
"""
class Maze:
    def __init__(self):
        self.N = 4

    def printSolution(self, sol):
        i = 0
        while i < self.N:
            j = 0
            while j < self.N:
                print(sol[i][j], end=" ")
                j += 1
            print()
            i += 1

    def isSafe(self, maze, x, y):
        return x >= 0 and x < self.N and y >= 0 and y < self.N and maze[x][y] == 1

    def getPath(self, maze, x, y, sol):
        if x == self.N - 1 and y == self.N - 1:
            # 到达中点
            sol[x][y] = 1
            return True
        if self.isSafe(maze, x, y):
            sol[x][y] = 1
            if self.getPath(maze, x, y + 1, sol):
                return True
            if self.getPath(maze, x + 1, y, sol):
                return True
            sol[x][y] = 0
            return False
        return False
if __name__ == "__main__":
    rat = Maze()
    maze = [[1, 0, 0, 0], [1, 1, 0, 1], [0, 1, 0, 0], [1, 1, 1, 1]]
    sol = [[0, 0, 0, 0] for _ in range(4)]
    res = rat.getPath(maze, 0, 0, sol)
    if res:
```

```
43.             rat.printSolution(sol)
```

```python
# @Filename: 给定一个矩阵，顺时针向里打印数据.py
# matrix类型为二维列表，需要返回列表
def printMatrix(matrix):
    # write code here
    if not matrix:
        return matrix
    res = []
    startX = 0
    while len(matrix) > 2 * startX and len(matrix[0]) > 2 * startX:
        res.extend(print_matrix_incircle(matrix, startX))
        startX += 1
    return res

def print_matrix_incircle(matrix, start):
    res = []
    endX = len(matrix[0]) - 1 - start
    endY = len(matrix) - 1 - start

    # 从左往右打印,没有约束条件
    for i in range(start, endX + 1):
        print(matrix[start][i])
        res.append(matrix[start][i])

    # 从上往下打印，至少有两行
    if endY > start:
        for i in range(start + 1, endY + 1):
            print(matrix[i][endX])
            res.append(matrix[i][endX])

    # 从右往左打印，至少有两行两列
    if start < endX and endY > start:
        for i in range(endX - 1, start - 1, -1):
            print(matrix[endY][i])
            res.append(matrix[endY][i])
    # 从下往上打印，至少有三行两列
    if start < endY - 1 and start < endX:
        for i in range(endY - 1, start, -1):
            print(matrix[i][start])
            res.append(matrix[i][start])
    return res

print(printMatrix([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]))
```

7. 栈

```
# @Filename: 从数组中找出满足a+b=c+d的数.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 使用字典存储 key:两个数的值 value为数据对


def findPairs(arr):
    dic = {}
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            if arr[i] + arr[j] in dic:
                print(dic[arr[i] + arr[j]][0], " + ", dic[arr[i] + arr[j]][1], " = ", arr[i],  " + ", arr[j])
            else:
                dic[arr[i] + arr[j]] = [arr[i], arr[j]]


if __name__ == "__main__":
    arr = [3, 4, 7, 10, 20, 9, 8]
```

```
# @Filename: 实现LRU缓存.py
# LRU : least recently used,最近最少使用
# 1. 使用双向链表实现队列的操作，将最近使用的数据放在链表头，当链表满了之后，删除最后节点
# 2. 使用哈希表用于缓存已经存在的页面
# 每当访问一个页面，首先在哈希表中查找存不存在，存在则移动队列中的数据到表头，不存在则添加数据到队列表头（需要考虑队列是否已满）

# deque本身是为了高效实现插入和删除的双向列表，使用与栈和队列， 增加了appendleft()和popleft()
from collections import deque
class LRU:
    def __init__(self, cacheSize):
        self.cacheSize = cacheSize
        self.queue = deque()
        self.hashSet = set()

    def isQueueFull(self):
        return len(self.queue) == self.cacheSize

    # 如果队列满了，删除末尾数据
    def enqueue(self, pageNum):
        if self.isQueueFull():
            self.hashSet.remove(self.queue[-1])
            self.queue.pop()
        self.queue.appendleft(pageNum)
        self.hashSet.add(pageNum)

    def accessPage(self, pageNum):
        if pageNum not in self.hashSet:
            self.enqueue(pageNum)
        elif pageNum != self.queue[0]:
            self.queue.remove(pageNum)
            self.queue.appendleft(pageNum)

    def printQueue(self):
        while len(self.queue) > 0:
            print(self.queue.popleft(), sep=" ")
if __name__ == "__main__":
    lru = LRU(3)
    lru.accessPage(1)
    lru.accessPage(2)
    lru.accessPage(4)
    lru.accessPage(5)
    lru.accessPage(6)
```

```
    lru.accessPage(7)
    lru.printQueue()
```

```python
# @Filename: 栈排序.py
class Stack:
    # 模拟栈
    def __init__(self):
        self.items = []

    def empty(self):
        return True if not self.items else False

    def size(self):
        return len(self.items)

    # 返回栈顶元素
    def peek(self):
        if not self.empty():
            return self.items[-1]
        else:
            return None
    # 弹栈
    def pop(self):
        if not self.empty():
            return self.items.pop()
        else:
            print("栈已空")
            return None

    # 压栈
    def push(self, item):
        self.items.append(item)

def moveBottomToTop(s):
    # 将栈底元素移动到栈顶，其他栈元素顺序不变
    if s.empty():
        return
    top1 = s.pop()
    if not s.empty():
        moveBottomToTop(s)
        top2 = s.peek()
        if top2 < top1:
            s.pop()
            s.push(top1)
            s.push(top2)
            return
```

```
    s.push(top1)

# 翻转栈
def sort_stack(s):
    if s.empty():
        return
    moveBottomToTop(s)
    # 记录逆序的顺序，后面递归要还原回去
    top = s.peek()
    print(top)
    s.pop()
    # 将子栈的栈底移动到子栈的栈顶
    sort_stack(s)
    s.push(top)
if __name__ == "__main__":
    s = Stack()
    for i in range(1, 6):
        s.push(i)
    print("before reverse")
    print(s.items)
    sort_stack(s)
    print("after reverse")
```

```
# @Filename: 根据入栈顺序判断可能的出栈顺序.py
# 依次入栈，如果当前入栈元素与出栈元素第一个相等，则出栈，出栈对比元素删除，继续入栈，循环
class Stack:
    def __init__(self):
        self.items = []

    # 判断栈是否为空
    def empty(self):
        return True if not self.items else False

    # 返回栈的大小
    def size(self):
        return len(self.items)

    # 返回栈顶元素
    def peek(self):
        return self.items[-1]

    # 出栈
    def pop(self):
        if self.empty():
            print("栈已空")
            return None
        else:
            return self.items.pop()

    # 入栈
    def push(self, item):
        self.items.append(item)

def isPopSerial(push, pop):
    if not push or not pop:
        return False
    if len(push) != len(pop):
        return False
    pushIndex = 0
    popIndex = 0
    stack = Stack()
    while pushIndex < len(push):
        stack.push(push[pushIndex])
        pushIndex += 1
        while not stack.empty() and stack.peek() == pop[popIndex]:
            stack.pop()
```

```
            popIndex += 1
    # 这里只需判断入栈是否为空就行了
    return stack.empty()

if __name__ == "__main__":
    push = "123324"
    pop = "5213254"
    if isPopSerial(push, pop):
        print(pop +"是出栈顺序")
    else:
        print(pop +"不是出栈顺序")
```

```
# @Filename: 用两个栈模拟队列操作.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 入队列时将栈A正常入栈，当要出队列时判断：B是否为空：1.为空，则依次弹出A元素，放入栈B，2：不为空，直接输出栈B的内容
class Stack:
    def __init__(self):
        self.items = []

    def empty(self):
        return True if not self.items else False

    def size(self):
        return len(self.items)

    def peek(self):
        return self.items[-1]
    def pop(self):
        return self.items.pop()
    def push(self, item):
        self.items.append(item)

class MyQueue:
    def __init__(self):
        self.A = Stack()
        self.B = Stack()

    def push(self, item):
        self.A.push(item)

    def pop(self):
        if not self.B.empty():
            return self.B.pop()
        else:
            for i in range(self.A.size()):
                self.B.push(self.A.pop())

            return self.B.pop()

if __name__ == "__main__":
    queue = MyQueue()
    queue.push(1)
    queue.push(2)
    print("输出队列元素", queue.pop())
```

```
    print("输出队列元素", queue.pop())
```

```
# @Filename: 翻转栈的所有元素.py
# 由于栈有两种实现方式，数组和链表
# 栈的翻转对于链表来说，相当于链表的逆序
# 本程序考虑数组的实现模式
"""
首先将栈底数据放在栈顶，然后对剩下栈内容做相同操作进行递归
"""
class Stack:
    # 模拟栈
    def __init__(self):
        self.items = []

    def empty(self):
        return True if not self.items else False

    def size(self):
        return len(self.items)

    # 返回栈顶元素
    def peek(self):
        if not self.empty():
            return self.items[-1]
        else:
            return None
    # 弹栈
    def pop(self):
        if not self.empty():
            return self.items.pop()
        else:
            print("栈已空")
            return None

    # 压栈
    def push(self, item):
        self.items.append(item)

def moveBottomToTop(s):
    # 将栈底元素移动到栈顶，其他栈元素顺序不变
    if s.empty():
        return
    top1 = s.pop()
    if not s.empty():
        moveBottomToTop(s)
```

```
            top2 = s.peek()
            s.pop()
            s.push(top1)
            s.push(top2)
        else:
            s.push(top1)

# 翻转栈
def reverse_stack(s):
    if s.empty():
        return
    moveBottomToTop(s)
    # 记录逆序的顺序，后面递归要还原回去
    top = s.peek()
    print(top)
    s.pop()
    # 将子栈的栈底移动到子栈的栈顶
    reverse_stack(s)
    s.push(top)

if __name__ == "__main__":
    s = Stack()
    for i in range(1, 6):
        s.push(i)
    print("before reverse")
    print(s.items)
    reverse_stack(s)
    print("after reverse")
    print(s.items)
```

```
# @Filename: 通过给定的车票找出旅程.py
# 不考虑环的情况
# 构建一个车票旅途的字典，同时建立一个reverse字典，由于只有起点没有前继，所以只有起点在reverse字典中没有key
def printResult(inputs):
    reverse_inputs = dict(zip(inputs.values(), inputs.keys()))
    reverse_values = reverse_inputs
    start = None
    for k, v in inputs.items():
        if k not in reverse_values:
            start = k
            break
    print(start)
    # 通过开始点，构建旅途
    while start in inputs:
        print(start+"->"+inputs[start])
        start = inputs[start]
if __name__ == "__main__":
    inputs = dict()
    inputs['西安'] = '成都'
    inputs['北京'] = '上海'
    inputs['大连'] = '西安'
    inputs['上海'] = '大连'
    printResult(inputs)
```

8. 链表

```
# @Filename: 两个链表数之和（链表相加法）.py
# 时间复杂度：由于只需要一次遍历，为O（N），  空间复杂度：O（N），需要一个长为N的链表存储
class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def add(head1, head2):
    if not head1 or not head1.next:
        return head2
    if not head2 or not head2.next:
        return head1
    c = 0 # 表示进位
    result_head = LNode()
    cur1 = head1.next
    cur2 = head2.next
    cur3 = result_head
    # 使用另一个链表保存结果，  也可以采用其中一条链表保存结果
    while cur1 and cur2:
        tmp = LNode()
        result = cur1.data + cur2.data + c
        if result > 9:
            c = 1
            tmp.data = result - 10
        else:
            tmp.data = result
            c = 0
        cur3.next = tmp
        cur3 = tmp
        cur1 = cur1.next
        cur2 = cur2.next
    # 如果cur1数据较多，将cur1的数据继续添加在result_head中
    if cur1:
        while cur1:
            tmp = LNode()
            sum = cur1.data + c
            tmp.data = sum % 10
            c = sum//10
            cur3.next = tmp
            cur3 = tmp
```

```
            cur1 = cur1.next
    if cur2:
        while cur2:
            tmp = LNode()
            sum = cur2.data + c
            tmp.data = sum % 10
            c = sum // 10
            cur3.next = tmp
            cur3 = tmp
            cur2 = cur2.next

    # 如果任然有进位的话
    if c:
        tmp = LNode()
        tmp.data = 1
        cur3.next = tmp
    return result_head
```

```
# @Filename: 判断链表是否有环.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 采用快慢指针，如果有环，那么总有一次，两个指针相等
# 如果有环，则怎样确定环的入口位置：
import random
"""
1. 先找到相遇点
2. 分别从头结点相遇点一次遍历，相遇的位置即入口点
"""
class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def has_circle(head):
    slow = head
    fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return slow
    return False

if __name__ == "__main__":
    # 创建一个循环链表
    circle_entrance = random.randint(5, 10)
    print(circle_entrance)
    entrance_node = None
    i = 1
    head = LNode()
    cur = head
    while i < 20:
        tmp = LNode()
        tmp.data = i
        cur.next = tmp
        cur = tmp
        if i == circle_entrance:
            entrance_node = cur
        i += 1

```

```
    # 连接形成环
    cur.next = entrance_node

    # 判断是否有环
    if has_circle(head.next):
        meet_node = has_circle(head.next)
        cur = head.next
        while cur != meet_node:
            cur = cur.next
            meet_node = meet_node.next
        print(meet_node.data)
    else:
        print("没有环")
    # 如果有环，找出入口点
```

```
# @Filename: 合并两个有序链表.py
# 分别用两个指针遍历，分别比较当前两个指针的大小，根据要求进行插入操作

class LNode:
    def __init__(self):
        self.data = None
        self.next = None
def merge_two_list(h1, h2):
    if not h1 and not h1.next:
        return h2
    if not h2 and not h2.next:
        return h1
    cur1 = h1.next
    cur2 = h2.next
    next = h2.next.next
    pre = h1
    while cur1 and cur2:
        if cur1.data >= cur2.data:
            cur2.next = pre.next
            pre.next = cur2
            pre = cur2
            cur2 = next
            if cur2:
                next = next.next
        else:
            pre = cur1
            cur1 = cur1.next
    # 判断哪一个链表先完
    if not cur1:
        pre.next = cur2
        return
    else:
        return
```

```
# @Filename: 向右旋转k个位置.py
class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def ReverseK(head, k):
    cur = head
    next_k = head
    i = 0
    while i < k and next_k:
        next_k = next_k.next
        i += 1
    if i < k:
        # 链表还没有k长
        return None
    while next_k.next:
        cur = cur.next
        next_k = next_k.next
    second_head = cur.next
    cur.next = None
    next_k.next = head.next
    head.next = second_head
    return head
```

```python
# @Filename: 如何展开链接列表.py
class Node:
    def __init__(self):
        self.data = None
        self.right = None
        self.down = None

class MergeList:
    def __init__(self):
        self.head = None

    def insert(self, head_ref, data):
        tmp = Node()
        tmp.data = data
        tmp.down = head_ref
        head_ref = tmp
        return head_ref

    def merge(self, a, b):
        if not a:
            return b
        if not b:
            return a
        if a.data < b.data:
            result = a
            result.down = self.merge(a.down, b)
        else:
            result = b
            result.down = self.merge(a, b.down)
        return result

    def flatten(self, root):
        if not root or not root.right:
            return root
        # 递归处理root.right链表
        root.right = self.flatten(root.right)

        root = self.merge(root, root.right)
```

```
# @Filename: 无序链表移除重复项（hashset）3.py
# @Software: PyCharm
# @e_mail:   sancica@163.com

# 空间换时间，遍历时将数据载入list，每次遍历查看是否出现

class LNode:
    def __init__(self):
        self.data = None
        self.next = None


def remove_duplication(head):
    if not head or not head.next or not head.next.next:
        return
    node = []
    cur = head.next
    pre = head
    while cur:
        if cur.data not in node:
            node.append(cur.data)
            pre = cur
            cur = cur.next
        else:
            pre.next = cur.next
            cur = pre.next
```

```
# @Filename: 无序链表移除重复项（双重循环）3.py
# @Software: PyCharm
# @e_mail:   sancica@163.com

# 对于无顺序的链表，删除其中重复的项目
"""
分析：利用双重循环，时间复杂度为O(N*N)， 空间负责度为O(1)，常数个指针变量保存前继节点等
"""

class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def remove_duplication(head):
    if not head or not head.next:
        return
    Outer_cur = head.next
    while Outer_cur:
        Inter_pre = Outer_cur
        Inter_cur = Outer_cur.next
        while Inter_cur:
            if Inter_cur.data == Outer_cur.data:
                # 删除当前节点
                Inter_pre.next = Inter_cur.next
                Inter_cur = Inter_cur.next
            else:
                Inter_pre = Inter_cur
                Inter_cur = Inter_cur.next
        print(Outer_cur.data, Outer_cur.next)
        Outer_cur = Outer_cur.next
```

```
# @Filename: 无序链表移除重复项（递归）2.py
# @Software: PyCharm
# @e_mail:   sancica@163.com

"""
分析：还是双重遍历，时间复杂度为O（N*N），增加了额外的函数调用，效率较低
"""
class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def removeDupRecursive(head):
    if not head.next:
        return head
    cur = head
    head.next = removeDupRecursive(head.next)
    pointer = head.next

    while pointer:
        if head.data == pointer.data:
            cur.next = pointer.next
            pointer = cur.next
        else:
            pointer = pointer.next
            cur = cur.next
    return head

def remove_duplication(head):
    if not head:
        return
    head.next = removeDupRecursive(head.next)
```

```
# @Filename: 查找链表倒数第K个元素.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 采用两个指针，指针之间间隔k个node，当前面一个指针指向none，返回第一个指针的值

class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def FindLastK(head, k):
    cur = head
    next_k = head
    i = 0
    while i < k and next_k:
        next_k = next_k.next
        i += 1
    if i < k:
        # 链表还没有k长
        return None
    while next_k:
        cur = cur.next
        next_k = next_k.next
    return cur.data
```

```
# @Filename: 链表以K个节点为一组翻转.py
# 这道题不复杂，就是比较繁琐
class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def reverse(begin, end):
    pre = None
    cur = begin
    next = cur.next
    while cur != end:
        cur.next = pre
        pre = cur
        cur = next
        next = next.next
    cur.next = pre

def reverse_per_k(head, k):
    if not head and not head.next:
        return
    j = k
    pre = head
    begin = head.next
    end = begin
    while end:
        # 找到end指针
        while j > 1 and end:
            end = end.next
            j -= 1
        if end:
            lnext = end.next
            reverse(begin, end)
            pre.next = end
            pre = begin
            pre.next = lnext
            begin = lnext
            end = begin
            j = k
        else:
            return
```

```
# @Filename: 链表相邻元素翻转.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 1. 通过交换相邻链表的值，但对某些场景不适用
# 2. 遍历就地逆序法，需要添加一些指针
—
—
class LNode:
    def __init__(self):
        self.data = None
        self.next = None
—
—
def Reverse_eachother(head):
    if not head and not head.next:
        return
    pre = head
    cur = pre.next
    while cur and cur.next:
        next = cur.next.next
        pre.next = cur.next
        cur.next.next = cur
        cur.next = next
        pre = cur
        cur = next
```

```
# @Filename: 链表逆序输出（不改变原结构：递归）.py
# 采用递归的方法实现链表的逆序，原链表为1 2 3 4 5 6 7， 那么只要变为 1 （7 6 5 4 3 2）时，只需要将1 放在最后即可实现排序，
# 同时要得到 7 6 5 4 3 2， 只需要变为 2 （3 4 5 6 7）时，将2放在最末尾即可，依次进行递归


class LNode:
    def __init__(self):
        self.data = None
        self.next = None


def RecursiveReverse(head):
    if not head or not head.next:
        print(head.data)
    else:
        # 这里的new_head是最后一个数字的指针，一直向上传递，为了让head与新的第一个节点相连
        RecursiveReverse(head.next)
        print(head.data)

def Reverse_print(head):
    if not head:
        return
    firstNode = head.next
    newhead = RecursiveReverse(firstNode)
```

```
# @Filename: 链表逆序（递归）2.py
# 采用递归的方法实现链表的逆序，原链表为1 2 3 4 5 6 7， 那么只要变为 1 （7 6 5 4 3 2）时，只需要将1 放在最后即可实现排序，
# 同时要得到 7 6 5 4 3 2， 只需要变为 2 （3 4 5 6 7）时，将2放在最末尾即可，依次进行递归

class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def RecursiveReverse(head):
    if not head or not head.next:
        return head
    else:
        # 这里的new_head是最后一个数字的指针，一直向上传递，为了让head与新的第一个节点相连
        new_head = RecursiveReverse(head.next)
        head.next.next = head
        head.next = None
    return new_head

def Reverse(head):
    if not head:
        return
    firstNode = head.next
    newhead = RecursiveReverse(firstNode)
    head.next = newhead
    return head
```

```
# @Filename: 链表逆序（遍历+指针节点）1.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 创建链表结点类，使用引用方式实现链表的指针功能

"""
总结：对链表进行一次遍历，时间复杂度为O（N），N为链表长度，需要常数个向量保存pre cur next，空间复杂度为O（1）
"""
class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def Reverse(head):
    if not head or not head.next:
        return
    pre = None
    cur = head.next
    while cur.next:
        next = cur.next
        cur.next = pre
        pre = cur
        cur = next
    cur.next = pre
    head.next = cur
    return head
```

```
# @Filename: 链表逆序（遍历+插入）3.py
# @Software: PyCharm
# @e_mail:   sancica@163.com
# 采用插入法将，遍历数据时，将数据插入到头结点之后


class LNode:
    def __init__(self):
        self.data = None
        self.next = None


def Reverse(head):
    if not head or not head.next:
        return
    cur = head.next.next
    # 斩断第一个数据指针
    head.next.next = None
    while cur:
        next = cur.next
        cur.next = head.next
        head.next = cur
        cur = next
```

```
# @Filename: 链表重新排序.py
# 将原链表 0 1 2 3 4 5 重新排序为0 5 1 4 2 3， 要求只能修改当前链表，只能修改next域，不能修改数据域
# 分析：先找到中间节点，对后面数据进行逆序，然后在将前半部分与后半部分子链表进行合并， 时间复杂度为O（N），空间复杂度为O（1）
class LNode:
    def __init__(self):
        self.data = None
        self.next = None

def FindMiddleNode(head):
    # 使用两个指针，一个速度为1， 一个速度为2，当2为None，1就为中间节点
    fast = head
    slow = head
    while fast and fast.next:
        fast = fast.next.next
        slow = slow.next

    # 将链表进行拆分
    second_head = slow.next
    slow.next = None
    return second_head

def Reverse(head):
    pre = None
    cur = head
    next = head.next
    while next:
        cur.next = pre
        pre = cur
        cur = next
        next = next.next
    cur.next = pre
    return cur

def Reorder(head):
    if not head or not head.next or not head.next.next or not head.next.next.next:
        return
    mid = FindMiddleNode(head.next)
    # 对后半部分链表进行逆序
    cur2 = Reverse(mid)
    cur1 = head.next
    # 合并两个链表
```

```
    while cur2 and cur1:
        next2 = cur2.next
        cur2.next = cur1.next
        cur1.next = cur2
        cur1 = cur2.next
        cur2 = next2
```