

python刷题算法汇总

一：Python程序员面试算法宝典

1. 二叉树

```
1. # @Filename: trie树实现DNS缓存查找.py
2. class TrieNode:
3.     def __init__(self):
4.         CHAR_COUNT = 11
5.         self.isLeaf = False
6.         self.url = None
7.         self.child = [None] * CHAR_COUNT
8.         for i in range(len(self.child)):
9.             self.child[i] = None
10.        i += 1
11.    —
12.    def getIndexFromChar(c):
13.        # ord()返回ascii码
14.        return 10 if c == "." else (ord(c) - ord('0'))
15.    —
16.    def getCharFromIndex(i):
17.        return '.' if i == 10 else ('0' + str(i))
18.    —
19.    class DNSCache:
20.        def __init__(self):
21.            self.CHAR_COUNT = 11
22.            self.root = TrieNode()
23.        —
24.        def insert(self, ip, url):
25.            lens = len(ip)
26.            pCrawl = self.root
27.            level = 0
28.            while level < lens:
29.                index = getIndexFromChar(ip[level])
30.                if pCrawl.child[index] == None:
31.                    pCrawl.child[index] = TrieNode()
32.                pCrawl = pCrawl.child[index]
33.                level += 1
34.                pCrawl.isLeaf = True
35.                pCrawl.url = url
36.        —
37.        def searchDNSCache(self, ip):
38.            pCrawl = self.root
```

```
39.     lens = len(ip)
40.     level = 0
41.     while level < lens:
42.         index = getIndexFromChar(ip[level])
43.         if pCrawl.child[index] == None:
44.             return None
45.         pCrawl = pCrawl.child[index]
46.         print(pCrawl.url)
47.         level += 1
48.         if pCrawl != None and pCrawl.isLeaf:
49.             return pCrawl.url
50.     return None
51. —
52. if __name__ == "__main__":
53.     ipAdds = ["10.57.11.127", "121.57.61.129", "66.125.100.103", "10.57.11.122"]
54.     # ipAdds = ["10.57.11.127", "10.57.11.122"]
55.     # url = ["www.samsung.com", "www.huawei.com"]
56.     url = ["www.samsung.com", "www.samsung.net", "www.google.in", "www.huawei.com"]
57.     n = len(ipAdds)
58.     cache = DNSCache()
59.     for i in range(n):
60.         cache.insert(ipAdds[i], url[i])
61.         print(i)
62.     ip = "10.57.11.127"
63.     res_url = cache.searchDNSCache(ip)
64.     print(res_url)
```

```
1. # @Filename: 二叉树最大子树和.py
2. # 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
3. # 只能找到最大值, 怎么才算找到最大子树呢?
4. —
5. class BiTree:
6.     def __init__(self):
7.         self.data = None
8.         self.lchild = None
9.         self.rchild = None
10. —
11. class Test:
12.     def __init__(self):
13.         self.maxSum = -2 ** 31
14. —
15.     def constructTree(self):
16.         root = BiTree()
17.         node1 = BiTree()
18.         node2 = BiTree()
19.         node3 = BiTree()
20.         node4 = BiTree()
21.         root.data = 6
22.         node1.data = 3
23.         node2.data = -7
24.         node3.data = -1
25.         node4.data = 9
26.         root.lchild = node1
27.         root.rchild = node2
28.         node1.lchild = node3
29.         node1.rchild = node4
30.         return root
31. —
32.     def findMaxSubTree(self, root, maxRoot):
33.         if not root:
34.             return 0
35.         sum_l = self.findMaxSubTree(root.lchild, maxRoot)
36.         sum_r = self.findMaxSubTree(root.rchild, maxRoot)
37.         sums = sum_l + sum_r + root.data
38.         if sums > self.maxSum:
39.             self.maxSum = sums
40.         maxRoot.data = root.data
41.         return sums
42. —
```

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

```
1. # @Filename: 二叉树的层次遍历.py
2. # 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
3. from collections import deque
4. class BiTree:
5.     def __init__(self):
6.         self.data = None
7.         self.lchild = None
8.         self.rchild = None
9.     -
10. def arrtotree(arr):
11.     if not arr:
12.         return
13.     bt = BiTree()
14.     middle = len(arr) // 2
15.     bt.data = arr[middle]
16.     bt.lchild = arrtotree(arr[:middle])
17.     bt.rchild = arrtotree(arr[middle + 1:])
18.     return bt
19.     -
20. def printTreeLayer(root):
21.     if not root:
22.         return
23.     queue = deque()
24.     queue.append(root)
25.     while queue:
26.         root = queue.popleft()
27.         print(root.data, end=" ")
28.         if root.lchild:
29.             queue.append(root.lchild)
30.         if root.rchild:
31.             queue.append(root.rchild)
32.     -
33. if __name__ == "__main__":
34.     arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
35.     root = arrtotree(arr)
36.     printTreeLayer(root)
```

```
1. # @Filename: 判断一个数组是否是二元查找树后序遍历.py
2. """
3. 二元查找树后序遍历特性
4. 1. 最后访问节点一定为根节点
5. 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>=
6. """
7. def IsAfterOrder(arr, start, end):
8.     if not arr:
9.         return
10.    # 找到比end的值大的最先的那个值的下标
11.    root = arr[end]
12.    i = start
13.    while i < end:
14.        if arr[i] > root:
15.            break
16.        i += 1
17.    j = i
18.    # 判定条件
19.    while j < end:
20.        if arr[j] < root:
21.            return False
22.        j += 1
23.    left_IsAfterOrder = True
24.    right_IsAfterOrder = True
25.    if i > start:
26.        left_IsAfterOrder = IsAfterOrder(arr, start, i - 1)
27.    if j < end:
28.        right_IsAfterOrder = IsAfterOrder(arr, i, end)
29.    return left_IsAfterOrder and right_IsAfterOrder
30.
31. if __name__ == "__main__":
32.     arr = [1, 3, 2, 5, 7, 6, 4]
33.     result = IsAfterOrder(arr, 0, len(arr) - 1)
34.     print(result)
```

```
1. # @Filename: 判断两颗二叉树是否相等.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 两颗二叉树相等"
6. 1. 当前点相等
7. 2. 左子树相等、右子树相等
8. 递归下去
9. """
10. class BiTree:
11.     def __init__(self):
12.         self.data = None
13.         self.lchild = None
14.         self.rchild = None
15. —
16.     def isEqual(root1, root2):
17.         if not root1 and not root2:
18.             return True
19.         if root1 and not root2:
20.             return False
21.         if not root1 and root2:
22.             return False
23.         # 上面判断结构相等
24.         # 下面判断数据相等
25.         if root1.data == root2.data:
26.             return isEqual(root1.lchild, root2.lchild) and isEqual(root1.rchild,
27. root2.rchild)
28.         else:
29.             return False
30. —
31.     def constructTree():
32.         root = BiTree()
33.         node1 = BiTree()
34.         node2 = BiTree()
35.         node3 = BiTree()
36.         node4 = BiTree()
37. —
38.         root.data = 3
39.         node1.data = 6
40.         node2.data = 6
41.         node3.data = 6
42.         node4.data = 6
```

```
42.     root.lchild = node1
43.     root.rchild = node2
44.     node1.rchild = node3
45.     node2.rchild = node4
46.     return root
47. —
48.     if __name__ == "__main__":
49.         root1 = constructTree()
50.         root1.data = 9999
51.         root2 = constructTree()
52.         print(isEqual(root1, root2))
```



```
1. # @Filename: 在二叉排序数中找出第一个大于中间值的节点.py
2. # 1. 找到最小值和最大值, 分别为lchild.lchild...和rchild.rchild...直到为None
3. # 2. 利用中序遍历, 由于是有顺序的, 直接比较即可
4. --
5. class BiTree:
6.     def __init__(self):
7.         self.data = None
8.         self.lchild = None
9.         self.rchild = None
10. --
11. def arrtotree(arr):
12.     if not arr:
13.         return
14.     bt = BiTree()
15.     middle = len(arr) // 2
16.     bt.data = arr[middle]
17.     bt.lchild = arrtotree(arr[:middle])
18.     bt.rchild = arrtotree(arr[middle + 1:])
19.     return bt
20. --
21. def getMinNode(root):
22.     if not root:
23.         return
24.     while root.lchild:
25.         root = root.lchild
26.     return root
27. --
28. def getMaxNode(root):
29.     if not root:
30.         return
31.     while root.rchild:
32.         root = root.rchild
33.     return root
34. --
35. def getNode(root):
36.     min = getMinNode(root).data
37.     max = getMaxNode(root).data
38.     mid = (min + max) / 2
39.     --
40.     result = None
41.     # 直接通过比较大小找到相应节点
42.     while root != None:
43.         if root.data <= mid:
```

```
44.         root = root.rchild
45.     else:
46.         result = root
47.         root = root.lchild
48.     return result
49. —
50. if __name__ == "__main__":
51.     arr = [1, 2, 3, 4, 5, 6, 7]
52.     root = arrtotree(arr)
53.     print(getNode(root).data)
```

```
1. # @Filename: 在二叉树中找出与输入整数相等的所有路径.py
2. class BiTNode:
3.     def __init__(self):
4.         self.data = None
5.         self.lchild = None
6.         self.rchild = None
7. —
8. def constructTree():
9.     root = BiTNode()
10.    node1 = BiTNode()
11.    node2 = BiTNode()
12.    node3 = BiTNode()
13.    node4 = BiTNode()
14.    root.data = 6
15.    node1.data = 3
16.    node2.data = 2
17.    node3.data = -1
18.    node4.data = 9
19.    root.lchild = node1
20.    root.rchild = node2
21.    node1.lchild = node3
22.    node1.rchild = node4
23.    return root
24. —
25. def FindRoad(root, num, sums, v):
26.     sums += root.data
27.     v.append(root.data)
28. —
29.     # 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
30.     # if not root.lchild and not root.rchild and num == sums:
31.     if num == sums:
32.         for i in v:
33.             print(i, end=" ")
34.         print()
35.         if root.lchild:
36.             FindRoad(root.lchild, num, sums, v)
37.         if root.rchild:
38.             FindRoad(root.rchild, num, sums, v)
39.     # 清除当前节点数据
40.     sums -= root.data
41.     v.pop()
42. —
```

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

```

1. # @Filename: 在二叉树中找出路径最大的和.py
2. # 这里的路径包含任意节点作为起点和终点
3. """
4. 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)
5. """
6. class BiTNode:
7.     def __init__(self, val):
8.         self.val = val
9.         self.lchild = None
10.        self.rchild = None
11.    -
12.    class InfRef:
13.        def __init__(self):
14.            self.val = None
15.    -
16.    def Max(a, b, c, d):
17.        maxs = a if a > b else b
18.        maxs = maxs if maxs > c else c
19.        maxs = maxs if maxs > d else d
20.    -
21.    return maxs
22.    -
23.    def findMaxPathRecursive(root, maxs):
24.        if not root:
25.            return 0
26.        sum_left = findMaxPathRecursive(root.lchild, maxs)
27.        sum_right = findMaxPathRecursive(root.rchild, maxs)
28.        -
29.        # 这里需要加上本身节点, 才能在上一层变成连续的路径
30.        allMax = root.val + sum_left + sum_right
31.        rightMax = root.val + sum_right
32.        leftMax = root.val + sum_left
33.        # 同时统计本身是否比左、右、左加右都还大
34.        tmpMax = Max(allMax, rightMax, leftMax, root.val)
35.        -
36.        # 更新最大值
37.        if tmpMax > maxs.val:
38.            maxs.val = tmpMax
39.        subMax = sum_left if sum_left > sum_right else sum_right
40.        return root.val + subMax
41.    -
42.    def findMaxPath(root):

```

```
43.     maxs = InfRef()
44.     maxs.val = -2 ** 31
45.     findMaxPathRecursive(root, maxs)
46.     return maxs.val
47. —
48.     if __name__ == "__main__":
49.         root = BiTNode(10)
50.         left = BiTNode(-1)
51.         right = BiTNode(-2)
52.         root.lchild = left
53.         root.rchild = right
54.         left1 = BiTNode(5)
55.         right1 = BiTNode(4)
56.         left.lchild = left1
57.         left.rchild = right1
58.         print(findMaxPath(root))
```

```
1. # @Filename: 复制二叉树.py
2. class BiTNode:
3.     def __init__(self):
4.         self.data = None
5.         self.lchild = None
6.         self.rchild = None
7. —
8. def createDupTree(root):
9.     if not root:
10.         return None
11.     dupTree = BiTNode()
12.     dupTree.data = root.data
13.     dupTree.lchild = createDupTree(root.lchild)
14.     dupTree.rchild = createDupTree(root.rchild)
15.     return dupTree
16. —
17. def printTreeMidOrder(root):
18.     if not root:
19.         return
20.     printTreeMidOrder(root.lchild)
21.     print(root.data, end=" ")
22.     printTreeMidOrder(root.rchild)
23. —
24. def constructTree():
25.     root = BiTNode()
26.     node1 = BiTNode()
27.     node2 = BiTNode()
28.     node3 = BiTNode()
29.     node4 = BiTNode()
30.     root.data = 6
31.     node1.data = 3
32.     node2.data = -7
33.     node3.data = -1
34.     node4.data = 9
35.     root.lchild = node1
36.     root.rchild = node2
37.     node1.lchild = node3
38.     node1.rchild = node4
39.     return root
40. if __name__ == "__main__":
41.     root1 = constructTree()
42.     root2 = createDupTree(root1)
```

```
43. printTreeMidOrder(root1)
44. print("\ndup it")
45. printTreeMidOrder(root2)
```



```
1. # @Filename: 对二叉树进行镜像翻转.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. --
5. from collections import deque
6. --
7. --
8. class BiTree:
9.     def __init__(self):
10.         self.data = None
11.         self.lchild = None
12.         self.rchild = None
13. --
14. def reverseTree(root):
15.     if not root:
16.         return
17.     reverseTree(root.lchild)
18.     reverseTree(root.rchild)
19.     tmp = root.lchild
20.     root.lchild = root.rchild
21.     root.rchild = tmp
22. --
23. def arrtotree(arr):
24.     if not arr:
25.         return
26.     bt = BiTree()
27.     middle = len(arr) // 2
28.     bt.data = arr[middle]
29.     bt.lchild = arrtotree(arr[:middle])
30.     bt.rchild = arrtotree(arr[middle + 1:])
31.     return bt
32. --
33. def printTreeLayer(root):
34.     if not root:
35.         return
36.     queue = deque()
37.     queue.append(root)
38.     while queue:
39.         p = queue.popleft()
40.         print(p.data, end=" ")
41.         if p.lchild:
42.             queue.append(p.lchild)
43.         if p.rchild:
```

```
44.     queue.append(p.rchild)
45.     print()
46.     -
47.     if __name__ == "__main__":
48.         arr = [1, 2, 3, 4, 5, 6, 7]
49.         root = arrtotree(arr)
50.         printTreeLayer(root)
51.         reverseTree(root)
52.         printTreeLayer(root)
```

```
1. # @Filename: 将有序数组放到二叉树中.py
2. # 可以采用中序遍历的顺序将数据放入进去
3. # 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
4. class BiTree:
5.     def __init__(self):
6.         self.data = None
7.         self.lchild = None
8.         self.rchild = None
9.     -
10. def arr2tree(arr):
11.     # 将采用递归调用
12.     # 1. 确定递归返回条件
13.     if not arr:
14.         return None
15.     # 2. 找到中点, 并将该点放入二叉树
16.     middle = len(arr) // 2
17.     bt = BiTree()
18.     bt.data = arr[middle]
19.     # 递归建立左子树节点
20.     bt.lchild = arr2tree(arr[:middle])
21.     # 递归建立右子树
22.     bt.rchild = arr2tree(arr[middle + 1:])
23.     return bt
24.     -
25. def printTree_middle(root):
26.     if not root:
27.         return
28.     if root.lchild:
29.         printTree_middle(root.lchild)
30.     print(root.data, end=" ")
31.     if root.rchild:
32.         printTree_middle(root.rchild)
33.     -
34. if __name__ == "__main__":
35.     sorted_arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
36.     print("数组")
37.     for num in sorted_arr:
38.         print(num, end=" ")
39.     print("\n")
40.     root = arr2tree(sorted_arr)
41.     # 采用中序遍历整个二叉树
42.     printTree_middle(root)
```

```

1. # @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
2. # 遍历所有路径, 当前点与所需点相等, 返回True, 并push该节点
3. """
4. 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 $O(n)$ 
5. """
6. class BiTree:
7.     def __init__(self):
8.         self.data = None
9.         self.lchild = None
10.        self.rchild = None
11.    -
12.    class Stack:
13.        def __init__(self):
14.            self.items = []
15.        -
16.        def is_empty(self):
17.            return True if not self.items else False
18.        -
19.        def size(self):
20.            return len(self.items)
21.        -
22.        def peek(self):
23.            if not self.is_empty():
24.                return self.items[-1]
25.            else:
26.                return None
27.        -
28.        def pop(self):
29.            if not self.is_empty():
30.                return self.items.pop()
31.            else:
32.                print("栈已空")
33.                return None
34.        -
35.        def push(self, item):
36.            self.items.append(item)
37.        -
38.    def getPathFromRoot(root, node, s):
39.        # 路径遍历, 传入栈s, 将找到的路径递归加入栈中
40.        if not root:
41.            return False
42.        if root == node:

```

```
43.     s.push(root)
44.     return True
45. -
46.     if getPathFromRoot(root.lchild, node, s) or getPathFromRoot(root.rchild, node,
    s):
47.         s.push(root)
48.         return True
49.     return False
50. -
51. def arraytotree(arr):
52.     if not arr:
53.         return
54.     bt = BiTree()
55.     middle = len(arr) // 2
56.     bt.data = arr[middle]
57.     bt.lchild = arraytotree(arr[:middle])
58.     bt.rchild = arraytotree(arr[middle + 1:])
59.     return bt
60. -
61. def FindParentNode(root, node1, node2):
62.     stack1 = Stack()
63.     stack2 = Stack()
64.     getPathFromRoot(root, node1, stack1)
65.     getPathFromRoot(root, node2, stack2)
66.     common_parent = None
67.     while stack1.peek() == stack2.peek():
68.         common_parent = stack1.peek()
69.         stack2.pop()
70.         stack1.pop()
71.     return common_parent
72. -
73. if __name__ == "__main__":
74.     arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
75.     root = arraytotree(arr)
76.     node1 = root.lchild.lchild.lchild
77.     node2 = root.lchild.rchild
78.     res = FindParentNode(root, node1, node2)
79.     if res:
80.         print(str(node1.data)+"与"+str(node2.data)+"的最近公共父节点为: "+str(res.data))
81.     else:
82.         print("没有公共交接点")
```

```

1. # @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
2. """
3. 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)
4. 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
5. """
6. import math
7. class BiTree:
8.     def __init__(self):
9.         self.data = None
10.        self.lchild = None
11.        self.rchild = None
12.    —
13.    def arraytotree(arr):
14.        if not arr:
15.            return
16.        bt = BiTree()
17.        middle = len(arr) // 2
18.        bt.data = arr[middle]
19.        bt.lchild = arraytotree(arr[:middle])
20.        bt.rchild = arraytotree(arr[middle + 1:])
21.        return bt
22.    —
23.    class IntRef:
24.        def __init__(self):
25.            self.num = None
26.    —
27.    def getNo(root, node, number):
28.        # 找到node点的number编码, 左子树*2, 右子树*2+1
29.        if not root:
30.            return False
31.        if root == node:
32.            return True
33.        tmp = number.num
34.        number.num = 2 * tmp
35.        if getNo(root.lchild, node, number):
36.            return True
37.        else:
38.            number.num = 2 * tmp + 1
39.            return getNo(root.rchild, node, number)
40.    —
41.    def getNodeFromNum(root, number):
42.        if not root or number < 0:

```

```
43.         return None
44.     if number == 1:
45.         return root
46.     # 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
47.     lens = int(math.log(number) / math.log(2))
48.     number -= 1 << lens
49.     while lens > 0:
50.         # 这里终于搞明白了, 获取当前这一位二进制的值
51.         print(1 << (lens - 1) & number >= 1)
52.         if (1 << (lens - 1) & number) >= 1:
53.             root = root.rchild
54.         else:
55.             root = root.lchild
56.         lens -= 1
57.     return root
58. —
59. def FindParentNode(root, node1, node2):
60.     ref1 = IntRef()
61.     ref1.num = 1
62.     ref2 = IntRef()
63.     ref2.num = 1
64.     getNo(root, node1, ref1)
65.     getNo(root, node2, ref2)
66.     num1 = ref1.num
67.     num2 = ref2.num
68.     —
69.     while num1 != num2:
70.         if num1 > num2:
71.             num1 //= 2
72.         else:
73.             num2 //= 2
74.     return getNodeFromNum(root, num1)
75. —
76. if __name__ == "__main__":
77.     arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
78.     # arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
79.     root = arraytotree(arr)
80.     node1 = root.lchild.rchild.lchild
81.     node2 = root.lchild.rchild.rchild
82.     res = FindParentNode(root, node1, node2)
83.     print(res.data)
84.     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")
```



```
1. # @Filename: 把二叉树转换为双向链表.py
2. """
3. 搜索二叉树: 左子树<根节点<右子树
4. 通过中序遍历能够找出有序的顺序
5. 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
6. """
7. class BiTree:
8.     def __init__(self):
9.         self.data = None
10.        self.lc = None
11.        self.rc = None
12.    —
13.    class Test:
14.        def __init__(self):
15.            self.pHead = None
16.            self.pEnd = None
17.        —
18.        def arraytotree(self, arr):
19.            if not arr:
20.                return None
21.            bt = BiTree()
22.            middle = len(arr) // 2
23.            bt.data = arr[middle]
24.            bt.lc = self.arraytotree(arr[:middle])
25.            bt.rc = self.arraytotree(arr[middle + 1:])
26.            return bt
27.        —
28.        def treetolist(self, root):
29.            if not root:
30.                return
31.            self.treetolist(root.lc)
32.            # 当前节点左子树指向前一节点
33.            root.lc = self.pEnd
34.            if not self.pEnd:
35.                # 保留头结点
36.                self.pHead = root
37.            else:
38.                # 前一节点的右节点指向当前节点
39.                self.pEnd.rc = root
40.            # 前移
41.            self.pEnd = root
```

```
42.         self.treetolist(root.rc)
43.     --
44.     if __name__ == "__main__":
45.         arr = [1,2,3,4,5,6,7,8,9,10]
46.         test = Test()
47.         root = test.arraytotree(arr)
48.         test.treetolist(root)
49.         cur = test.pHead
50.         print("正向遍历")
51.         while cur:
52.             print(cur.data, end=" ")
53.             cur = cur.rc
54.         cur = test.pEnd
55.         print("\n")
56.         print("逆向遍历")
57.         while cur:
58.             print(cur.data, end=" ")
59.             cur = cur.lc
```

2. 基本数字运算

```
1. # @Filename: 判断1024的阶乘末尾有多少个0.py
2. """
3. 分析:
4. 1. 结果中0的来源
5. 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
6. 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
7. """
8. def zeroCount(n):
9.     count = 5
10.     res = 0
11.     while True:
12.         i = 1
13.         while i <= n:
14.             if i % count == 0:
15.                 res += 1
16.                 i += 1
17.                 count *= 5
18.         —
19.         if count > n:
20.             break
21.         print(res)
22.     —
23. def zeroCount_nice(n):
24.     count = 0
25.     while n > 0:
26.         n = n // 5
27.         count += n
28.     print(count)
29. zeroCount_nice(1)
```

```
1. # @Filename: 判断一个数是否为2的n次方.py
2. """
3. 分析:
4. 与2沾边的数, 最好采用位移操作
5. 1. 构造2的倍数, 判断是否相等
6. 2. 判断该数的二进制的第一位是否为1, 其他为0
7. """
8. def construct_num(n):
9.     i = 1
10.    while i <= n:
11.        if i == n:
12.            return True
13.        i <<= 1
14.    return False
15. -
16. def judge_ord(n):
17.     # 负数在内存中是补码的形式存在
18.     if n < 1:
19.         return False
20.     if not (n & n - 1):
21.         return True
22.     else:
23.         return False
24. print(judge_ord(0))
```

```

1. # @Filename: 判断一个自然数是否是某个数的平方.py
2. """
3. 前提: 不使用开方运算
4. 分析:
5. 1. 直接计算, i递增, 计算i**2与n的大小关系
6. 2. 利用二分查找法找到对应的i, 减少运算的次数
7. 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)
8. """
9. def isPower_bn(n):
10.     i = 0
11.     j = n
12.     mid = (i+j)//2
13.     while i <=j:
14.         if mid**2 == n:
15.             return True
16.         elif mid**2 > n:
17.             j = mid-1
18.         else:
19.             i = mid+1
20.         mid = (i+j)//2
21.     return False
22. —
23. def isPower_sub(n):
24.     """
25.     减法运算法
26.     """
27.     res = n
28.     flag = 1
29.     while res >= 0:
30.         if res == 0:
31.             return True
32.         res -= flag
33.         flag += 2
34.     return False
35. —
36. print(isPower_sub(25))

```

```
1. # @Filename: 如何不使用^实现异或运算.py
2. """
3. 使用或 | 实现
4. """
5. def xor(x, y):
6.     # (x | y)为1时, (0,1) (1,0) (1,1): 需要排除 (1,1), 则在后面& (~x|~y)
7.     return (x | y) & (~x | ~y)
8. -
9. print(xor(2, 2))
```

```
1. # @Filename: 如何不使用加减乘除运算实现减法.py
2. """
3. 分析:
4. 减法就是加一个负数, 同加法一样处理
5. """
6. def add(a, b):
7.     # 将b转为b的负数形式, 然后采用加法的运算
8.     # b = ~b + 1
9.     while True:
10.         res = a ^ b
11.         carry = (res & b) << 1
12.         if carry == 0:
13.             return res
14.         a = res
15.         b = carry
16.     return res
17. —
18. def sub(a, b):
19.     return add(a, b)
20. —
21. print(sub(14, 4))
```

```
1. # @Filename: 如何不使用加减乘除运算实现加法.py
2. """
3. # 利用二进制的异或和 与运算实现加法
4. """
5. # 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
6. def sum(num1, num2):
7.     while True:
8.         res = (num1 ^ num2) & 0xffffffff
9.         carry = ((num1 & num2) << 1) & 0xffffffff
10.        if carry == 0:
11.            # 前面部分, 表示里面如果没有负数, 后面部分表示有负数
12.            return res if res <= 0x7fffffff else ~(res ^ 0xffffffff)
13.        num1 = res
14.        num2 = carry
15.    print(sum(-11, 2))
```



```
1. # @Filename: 如何不使用除法操作符实现两个正整数的除法.py
2. """
3. 分析:
4. 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%)
5. 2. 位移法, 通过位移加快逼近速度
6. """
7. def sub_divid(n, m):
8.     count = 0
9.     while n >= m:
10.         n -= m
11.         count += 1
12.     return count
13. —
14. def shift_divid(n, m):
15.     res = 0
16.     index = 0
17.     while n >= m:
18.         flag = m * 2 ** index
19.         count = 2 ** index
20.         if flag > n:
21.             n -= flag // 2
22.             count //= 2
23.             res += count
24.             index = 0
25.         elif flag < n:
26.             index += 1
27.         else:
28.             return res + count
29.     return res
30. print(shift_divid(200, 20))
```

```
1. # @Filename: 如何找最小的不重复数.py
2. """
3. 分析:
4. 不重复数: 数的每一位相邻之间不同, 如1201 1212
5. 题目: 给定n, 找到最小的那个不重复数
6. 1. 首先n += 1
7. 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)
8.     a. 如果此时i和i+1对应的数字相等, 则i++
9.     b. 否则continue
10. """
11. def find_min_num(n):
12.     n = n+1
13.     num = list(str(n))
14.     i = len(num) - 1
15.     while i > 0:
16.         if num[i] == num[i - 1]:
17.             n += 10 ** (len(num) - i - 1)
18.             num = list(str(n))
19.             flag = 1
20.             j = i+1
21.             while j < len(num):
22.                 flag = 1 ^ flag
23.                 num[j] = str(flag)
24.                 j += 1
25.             print(num)
26.             n = int(''.join(num))
27.             if i < len(num)-1 and num[i] == num[i + 1]:
28.                 i += 1
29.             continue
30.         i -= 1
31.     return n
32. —
33. print(find_min_num(8989))
```

1. `# @Filename: 如何按要求比较两个数的大小.py`
2. `"""`
3. 题目：比较两个数大小，不能使用大于、小于、`if`语句
4. `"""`
5. `def maxs(a, b):`
6. `return ((a + b) + abs(a - b)) / 2`
7. `print(maxs(5, 6))`

```
1. # @Filename: 如何根据已知随机数生成函数计算新的随机数.py
2. """
3. 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
4. 分析:
5. (rand7()-1)*7+rand7()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
6. 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)
7. """
8. import random
9. def rand7():
10.     return int(random.uniform(1, 7))
11. -
12. -
13. def rand10():
14.     while True:
15.         num = (rand7()-1)*7+rand7()
16.         if num <=40:
17.             break
18.         return num % 10+1
19. -
20. from collections import Counter
21. -
22. res = []
23. for i in range(10000000):
24.     res.append(rand10())
25. -
26. print([i/10000000 for i in dict(Counter(res)).values()])
```

```
1. # @Filename: 如何求有序数列的1500个数的值.py
2. """
3. 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
4. 分析:
5. 1. 遍历法
6. 2. 规律法
7. """
8. def search(n):
9.     a = [0, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27,
10.         28]
11.     print(len(a))
12.     ret = (n // 22) * 30 + a[n % 22]
13.     print(ret)
14.     search(1500)
```

```
1. # @Filename: 如何计算一个数的n次方.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 分析:
6. 1. 连乘或连除
7. 2. 递归或者动态规划法
8. """
9. def power(d, n):
10.     # 判断是奇数还是偶数次, 将采用不同的公式
11.     if n == 0: return 1
12.     if n == 1: return d
13.     tmp = power(d, abs(n) // 2)
14.     if n > 0:
15.         if n % 2 == 1:
16.             return tmp * tmp * d
17.         else:
18.             return tmp * tmp
19.     else:
20.         if n % 2 == 1:
21.             return 1 / (tmp * tmp * d)
22.         else:
23.             return 1 / (tmp * tmp)
24. —
25. def dynamic_power(d, n):
26.     if n == 0:
27.         return 1
28.     if n == 1:
29.         return d
30.     if n == -1:
31.         return 1/d
32.     arr = [0] * abs(n)
33.     arr[0] = 1
34.     arr[1] = d
35.     i = 2
36.     while i < abs(n):
37.         arr[i] = arr[i // 2] ** 2
38.         i = i*2
39.     if i == abs(n):
40.         if n > 0:
41.             return arr[n]
```

```
42.     else:
43.         return 1 / arr[-n]
44.     else:
45.         if n > 0:
46.             return arr[n//2]**2 * d
47.         else:
48.             return 1 / (arr[-n//2]**2 * d)
49. -
50. print(dynamic_power(2, -1))
```

```
1. # @Filename: 将十进制数分别以二进制和十六进制形式输出.py
2. """
3. 分析:
4. 十进制转二进制可以考虑采用“除2倒取余法”
5. """
6. def intToBinary(n):
7.     hexNum = 8 * 8
8.     bit = []
9.     for i in range(hexNum):
10.         b = n >> i
11.         # 同时计算商和余数
12.         c, d = divmod(b, 2)
13.         bit.append(str(d))
14.     return ''.join(bit[::-1])
15. —
16. def intToHex(s):
17.     hexs = ""
18.     while s != 0:
19.         remainder = s % 16
20.         if remainder < 10:
21.             hexs = str(remainder + ord('0')) + hexs
22.         else:
23.             hexs = str(remainder - 10 + ord('A')) + hexs
24.         s = s >> 4
25.     return chr(int(hexs))
26. —
27. print(intToBinary(11))
28. print(intToHex(10))
```



```
1. # @Filename: 求二进制数中1的个数.py
2. """
3. 分析:
4. 1. 依次向右位移, 判断最后一位是否为1 (采用&1判断)
5. 2. n依次与n-1相与&, 每次都会减少一个1, 直到结果为0
6. """
7. def judge_count(n):
8.     count = 0
9.     while n > 0:
10.         if n & 1 == 1:
11.             count += 1
12.             n = n >> 1
13.     return count
14. -
15. print(judge_count(8))
16. -
17. def judge_count_and(n):
18.     count = 0
19.     while True:
20.         if n != 0:
21.             n = n & (n - 1)
22.             count += 1
23.         else:
24.             break
25.     return count
26. -
27. print(judge_count_and(8))
```

3. 字符串

```
1. # @Filename: 判断一个字符串是否为另一个字符串旋转得到.py
2. """
3. 分析:
4. s2由s1旋转得到, 那么s2一定是s1s1的子串
5. """
6. def rotateSame(s1, s2):
7.     if len(s1) != len(s2):
8.         return False
9.     s1 = s1 + s1
10.    # 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
11.    if s2 in s1:
12.        return True
13.    else:
14.        return False
15. if __name__ == "__main__":
16.     print(rotateSame("waterbollte", "erbolltewat"))
```

```
1. # @Filename: 判断一个字符串是否包含重复字符.py
2. """
3. 题目：字符串中是否包含重复字符串
4. 分析：
5. 1. 蛮力法，双重循环判断是否有相等字符
6. 2. hash表法
7. 3. set长度比较法
8. 本代码实现hash法
9. """
10. def isDup(strs):
11.     str_dic = {}
12.     for i in strs:
13.         if i not in str_dic:
14.             str_dic[i] = 1
15.         else:
16.             return False
17.     return True
18. -
19. def isDup1(strs):
20.     return len(strs) == len(set(strs))
21. -
22. print(isDup("fdsasd"))
```

```
1. # @Filename: 判断两个字符串是否为换为字符串.py
2. """
3. 换位字符串：两个字符串由同样的字符构成，但位置不同
4. 分析：
5. 1. 字典法
6. """
7. from collections import Counter
8. def compare(str1, str2):
9.     counter1 = Counter(str1)
10.    counter2 = Counter(str2)
11.    for key in counter1.keys():
12.        if key in counter2:
13.            if counter1[key] != counter2[key]:
14.                return False
15.        else:
16.            return False
17.    return True
18. -
19. if __name__ == "__main__":
20.     str1 = "aaaabcc"
21.     str2 = "abcbaaa"
22.     print(compare(str1, str2))
```

```
1. # @Filename: 判断两个字符串的包含关系.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 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
6. 分析：
7. 1. 双层循环
8. 2. 空间换时间，利用hash表
9. """
10. # 本代码实现第二种方法
11. def is_contain(str1, str2):
12.     cha_dic = {}
13.     for char in str2:
14.         if char not in cha_dic:
15.             cha_dic[char] = 1
16.     —
17.     for char in str1:
18.         if char in cha_dic:
19.             cha_dic[char] = 0
20.     —
21.     return not sum(cha_dic.values())
22. print(is_contain("abcd", "agd"))
```

```
1. # @Filename: 判断字符串是否为整数.py
2. """
3. 分析:
4. 字符串形式的整数有三种表达方法 "52", "-52", "+52"
5. """
6. def judge_int(str):
7.     if str[0] == "+" or str[0] == "-":
8.         index = 1
9.     else:
10.         index = 0
11.     result = 0
12.     while index < len(str):
13.         if str[index].isdigit():
14.             result = result*10 + int(str[index])
15.         else:
16.             print("不是数字")
17.             return
18.         index += 1
19.     return -result if str[0] == "-" else result
20. -
21. print(judge_int("+4234"))
```

```
1. # @Filename: 句子中的单词反转.py
2. """
3. 分析:
4. 1. 先对整个句子进行反转, 最后再对每个单词进行反转
5. """
6. def reverseStr(stl, begin, end):
7.     # 使用异或的方式进行反转
8.     —
9.     while begin < end:
10.         stl[begin] = ord(stl[begin]) ^ ord(stl[end])
11.         stl[end] = stl[begin] ^ ord(stl[end])
12.         stl[begin] = stl[begin] ^ stl[end]
13.         stl[begin] = chr(stl[begin])
14.         stl[end] = chr(stl[end])
15.         begin += 1
16.         end -= 1
17.     —
18. def swapWord(str):
19.     stl = list(str)
20.     reverseStr(stl, 0, len(stl) - 1)
21.     begin = 0
22.     end = 0
23.     while end < len(stl):
24.         if stl[end] == " ":
25.             reverseStr(stl, begin, end - 1)
26.             end = end + 1
27.             begin = end
28.         else:
29.             end += 1
30.     return "".join(stl)
31. —
32. if __name__ == "__main__":
33.     print(swapWord("I love you"))
```

```

1. # @Filename: 如何在二维数组中寻找最短路径 (动态规划) .py
2. """
3. 分析: 动态规划
4. 1. 定义一个二维数组存储中间路径最小值
5.     a. 初始化  $f[i,0] = arr[1][0] + \dots + arr[i][0]$       $f[0][j] = arr[0][1] + \dots + arr[0][j]$ 
6.     b. 动态推导公式  $f[i][j] = arr[i][j] + \min(f[i-1][j], f[i][j-1])$ 
7. """
8. def getMinPath(arr):
9.     len1 = len(arr)
10.    len2 = len(arr[0])
11.    # 初始化二维数组
12.    f = [[0] * len2 for _ in range(len1)]
13.    i = 0
14.    temp_sum = 0
15.    while i < len1:
16.        temp_sum += arr[i][0]
17.        f[i][0] = temp_sum
18.        i += 1
19.    i = 0
20.    temp_sum = 0
21.    while i < len2:
22.        temp_sum += arr[0][i]
23.        f[0][i] = temp_sum
24.        i += 1
25.    for i in range(1, len1):
26.        for j in range(1, len2):
27.             $f[i][j] = arr[i][j] + \min(f[i-1][j], f[i][j-1])$ 
28.    print("二维数组最短路径长度为: ", f[i][j])
29.    return f[i][j]
30. -
31. getMinPath([[1, 4, 3], [8, 7, 5], [2, 1, 5]])

```



```
1. # @Filename: 如何在二维数组中寻找最短路径 (递归) .py
2. """
3. 分析:
4. 采用递归的方法, 从最后值
5. """
6. def getMinPath(arr, i, j):
7.     if i == 0 and j == 0:
8.         return arr[i][j]
9.     elif i > 0 and j > 0:
10.         return arr[i][j] + min(getMinPath(arr, i - 1, j), getMinPath(arr, i, j - 1))
11.     elif i > 0 and j == 0:
12.         return arr[i][j] + getMinPath(arr, i - 1, j)
13.     elif i == 0 and j > 0:
14.         return arr[i][j] + getMinPath(arr, i, j - 1)
15. —
16. if __name__ == "__main__":
17.     arr = [[1, 4, 3], [8, 7, 5], [2, 1, 5]]
18.     print(getMinPath(arr, len(arr) - 1, len(arr[0]) - 1))
```

```
1. # @Filename: 如何找到到达目标词的最短链长度.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. # 实现BFS算法, 广度优先遍历
6. """
7. —
8. from collections import deque
9. —
10. —
11. class QItem:
12.     def __init__(self, word, lens):
13.         self.word = word
14.         self.lens = lens
15. —
16. —
17. def isAdjacent(a, b):
18.     """
19.     判断两个字符串是否为邻接字符串
20.     """
21.     index = 0
22.     for i in range(len(a)):
23.         if a[i] != b[i]:
24.             index += 1
25.         if index > 1:
26.             return False
27.     return index == 1
28. —
29. —
30. def shortestChainLen(start, target, D):
31.     Q = deque()
32.     item = QItem(start, 1)
33.     Q.append(item)
34.     while Q:
35.         curr = Q[0]
36.         Q.pop()
37.         for it in D:
38.             temp = it
39.             if isAdjacent(curr.word, temp):
40.                 item.word = temp
41.                 item.lens = curr.lens + 1
42.                 Q.append(item)
43.                 D.remove(temp)
```

```
44.         if temp == target:
45.             return item.lens
46.     return 0
47. —
48. if __name__ == "__main__":
49.     D = []
50.     D.append("pooN")
51.     D.append("pbcc")
52.     D.append("zmc")
53.     D.append("poIc")
54.     D.append("pbca")
55.     D.append("pbIc")
56.     D.append("poIN")
57.     start = "TooN"
58.     target = "pbca"
59.     print(shortestChainLen(start, target, D))
```

```
1. # @Filename: 如何找到由其他单词组成的最长单词.py
2. """
3. 分析:
4. 1. 首先按照单词长度进行排序
5. 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
6. """
7. class LongestWord:
8.     def find(self, strArray, strs):
9.         if strs in strArray:
10.             return True
11.         else:
12.             return False
13.     —
14.     def isContain(self, strArray, word, length):
15.         lens = len(word)
16.         if lens == 0:
17.             return True
18.         i = 1
19.         while i <= lens:
20.             if i == length:
21.                 return False
22.             strs = word[0:i]
23.             if self.find(strArray, strs):
24.                 if self.isContain(strArray, word[i:], length):
25.                     return True
26.             i += 1
27.             return False
28.     —
29.     def getLongestStr(self, strArray):
30.         strArray = sorted(strArray, key=len, reverse=True)
31.         i = 0
32.         while i < len(strArray):
33.             if self.isContain(strArray, strArray[i], len(strArray[i])):
34.                 return strArray[i]
35.             i += 1
36.             return None
37.     —
38. if __name__ == "__main__":
39.     strArray = ['a', 'b', 'ab']
40.     lw = LongestWord()
41.     print(lw.getLongestStr(strArray))
```

```

1. # @Filename: 如何求字符串的编辑距离 (动态规划).py
2. """
3. 分析:
4. 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
5. 动态规划求解:
6. 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
7.     a. i = 0时, D[i][j] = j
8.     b. j = 0 时, D[i][j] = i
9. 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
10. 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
11. 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]
12. -
13. """
14. def minEditDistance(s1, s2):
15.     len1 = len(s1)
16.     len2 = len(s2)
17.     editDistance = [[None] * (len2+1) for _ in range(len1+1)]
18.     # 初始化边缘值
19.     for i in range(len1+1):
20.         editDistance[i][0] = i
21.         for j in range(len2+1):
22.             editDistance[0][j] = j
23.             for i in range(1, len1+1):
24.                 for j in range(1, len2+1):
25.                     if s1[i-1] == s2[j-1]:
26.                         editDistance[i][j] = min(editDistance[i-1][j]+1, editDistance[i][j-1]+1, editDistance[i-1][j-1])
27.                     else:
28.                         editDistance[i][j] = min(editDistance[i-1][j]+1, editDistance[i][j-1]+1, editDistance[i-1][j-1]+1)
29.             return editDistance[-1][-1]
30. -
31. print(minEditDistance("", "ros"))

```

```
1. # @Filename: 如何消除字符串的内嵌括号.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. --
5. """
6. 题目: 字符串中的括号问题
7. 分析:
8. 1. 用栈或者一个数来确保括号的配对
9. """
10. def removeNestedPare(strs):
11.     arr = list(strs)[1:-1]
12.     register = 0
13.     for i in range(len(arr)):
14.         if arr[i] == "(":
15.             register += 1
16.             arr[i] = ''
17.         elif arr[i] == ")":
18.             --
19.             arr[i] = ''
20.             register -= 1
21.             if register < 0:
22.                 print("非法表达式, 括号不匹配")
23.                 return None
24.     print(arr)
25.     return '(' + ''.join(arr) + ')'
26. --
27. print(removeNestedPare("(1, (2, 3), (4, (5, 6), 7))"))
```

```
1. # @Filename: 如果按照给定的字符序列对字符数组排序.py
2. # 根据给定的字符串顺序对字符串进行排序
3. """
4. 分析:
5. 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
6. """
7. def compare(str1, str2, char_to_int):
8.     i = 0
9.     while i < len(str1) and i < len(str2):
10.         if char_to_int[str1[i]] < char_to_int[str2[i]]:
11.             return True
12.         elif char_to_int[str1[i]] > char_to_int[str2[i]]:
13.             return False
14.         i += 1
15. —
16. def insertSort(s, char_to_int):
17.     # 插入排序
18.     lens = len(s)
19.     i = 1
20.     while i < lens:
21.         temp = s[i]
22.         j = i - 1
23.         while j >= 0:
24.             if compare(temp, s[j], char_to_int):
25.                 # 每一项后移
26.                 s[j + 1] = s[j]
27.             else:
28.                 break
29.             j -= 1
30.         # 将对应位置赋值
31.         s[j + 1] = temp
32.         i += 1
33. —
34. if __name__ == "__main__":
35.     s = ["bed", "dog", "dear", "eye"]
36.     sequence = "dgecfboa"
37.     char_to_int = {}
38.     for i in range(len(sequence)):
39.         char_to_int[sequence[i]] = i
40.     —
41.     insertSort(s, char_to_int)
42.     print(s)
```

```
1. # @Filename: 字符串反转 (异或) .py
2. —
3. def reverse(strs):
4.     i = 0
5.     j = len(strs) - 1
6.     # 这里必须转为list, 因为字符串属于常量, 无法修改
7.     strs = list(strs)
8.     while i < j:
9.         # 通过异或的方式进行交换
10.        print(type(ord(strs[i])))
11.        strs[i] = chr(ord(strs[i]) ^ ord(strs[j]))
12.        strs[j] = chr(ord(strs[i]) ^ ord(strs[j]))
13.        strs[i] = chr(ord(strs[i]) ^ ord(strs[j]))
14.        i += 1
15.        j -= 1
16.    return ''.join(strs)
17. —
18. if __name__ == "__main__":
19.     str = "abcdefg"
20.     print(reverse(str))
```



```

1. # @Filename: 实现字符串的匹配 (KMP) .py
2. """
3. 题目: 字符串匹配算法 KMP:
4. 分析:
5. 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
6.     a. 当j==0时, next[j] = -1
7.     b. 当j == 1时, next[j] = 0
8.     c. 当p[j] == p[k]: next[j+1] = next[k] + 1
9.     d. 当p[j] != p[k]: k = next[k]
10. -
11. https://www.cnblogs.com/yjiyjige/p/3263858.html
12. """
13. def getNext(p, nexts):
14.     k = 0
15.     j = -1
16.     nexts[0] = -1
17.     while k < len(p):
18.         if j == -1 or p[j] == p[k]:
19.             k += 1
20.             j += 1
21.             nexts[k] = j
22.         else:
23.             j = nexts[j]
24. -
25. def match(s, p, nexts):
26.     if not s or not p:
27.         return -1
28.     slen = len(s)
29.     plen = len(p)
30.     -
31.     if slen < plen:
32.         return -1
33.     i = 0
34.     j = 0
35.     while i < slen and j < plen:
36.         print("i="+str(i)+", "+str(j))
37.         if j == -1 or s[i] == p[j]:
38.             i += 1
39.             j += 1
40.         else:
41.             j = nexts[j]
42.         if j >= plen:

```

```
43.     return i-plen
44.     return -1
45.
46. if __name__ == "__main__":
47.     s = "abababaabcbab"
48.     p = "abaabc"
49.     lens = len(p)
50.     nexts = [0]*(lens+1)
51.     s = list(s)
52.     p = list(p)
53.     getNext(p, nexts)
54.     print("匹配结果: "+str(match(s, p, nexts)))
```

1. # @Filename: 对大小写字母组成的字符数组进行排序.py

2. """

3. 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

4. 分析：

5. 1. 首尾指针一次遍历法

6. """

7. def sort(str):

8. arr = list(str)

9. i = 0

10. j = len(arr) - 1

11. while i < j:

12. if 'A' <= arr[i] <= 'Z' and 'a' <= arr[j] <= 'z':

13. arr[i] = ord(arr[i]) ^ ord(arr[j])

14. arr[j] = arr[i] ^ ord(arr[j])

15. arr[i] = arr[i] ^ arr[j]

16. arr[i] = chr(arr[i])

17. arr[j] = chr(arr[j])

18. i += 1

19. j -= 1

20. continue

21. if 'z' >= arr[i] >= 'a':

22. i += 1

23. if 'A' <= arr[j] <= 'Z':

24. j -= 1

25. return "".join(arr)

26. —

27. print(sort("sFSDFfsDdfsDFDdfDFDF"))

```

1. # @Filename: 求一个串中出现的第一个最长重复子串 (后缀数组法) .py
2. """
3. 分析:
4. 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
5. 后缀数组: 字符串s从第i个字符开始的后缀表示为suffix (i)
6. """
7. -
8. class CommonSubString():
9.     # 找出最长的公共子串长度
10.     def maxPrefix(self, s1, s2):
11.         i = 0
12.         while i < len(s1) and i < len(s2):
13.             if s1[i] == s2[i]:
14.                 i += 1
15.             else:
16.                 return i
17.         return i
18. -
19.     def getMaxCommonStr(self, s):
20.         suffixes = []
21.         for i in range(len(s)):
22.             suffixes.append(s[i:])
23.         suffixes.sort()
24.         maxCommonStrLen = 0
25.         maxCommonStr = None
26.         for i in range(len(suffixes)-1):
27.             curMaxCommonStrLen = self.maxPrefix(suffixes[i], suffixes[i+1])
28.             if maxCommonStrLen < curMaxCommonStrLen:
29.                 maxCommonStrLen = curMaxCommonStrLen
30.             maxCommonStr = suffixes[i][:maxCommonStrLen]
31.         return maxCommonStr, maxCommonStrLen
32. if __name__ == "__main__":
33.     c = CommonSubString()
34.     str, len = c.getMaxCommonStr("banana")
35.     print(str, len)

```

```
1. # @Filename: 求一个字符串的所有排列（递归法）.py
2. """
3. 题目：输入字符串，输出字符串的所有排列
4. 分析：
5. 1. 递归法
6. a: 固定第一个字符，对后面的数据进行全排列
7. b: 全排列结束后，第一个字符与后面字符进行替换
8. c: 换回原来的字符串
9. """
10. -
11. def swap(str, i, j):
12.     tmp = str[i]
13.     str[i] = str[j]
14.     str[j] = tmp
15. -
16. def Permutation(str, start):
17.     if str==None or start < 0:
18.         return
19.     if start == len(str)-1:
20.         print("".join(str))
21.     else:
22.         i = start
23.         while i < len(str):
24.             # 下面代码即可剔除重复的排列元素
25.             # if i != start and str[i] == str[start]:
26.             #     i += 1
27.             #     continue
28.             swap(str, start, i)
29.             Permutation(str, start+1)
30.             swap(str, start, i)
31.             i += 1
32. -
33. def Permutation_transe(s):
34.     str = list(s)
35.     Permutation(str, 0)
36. -
37. if __name__ == "__main__":
38.     Permutation_transe("abbc")
```

```

1. # @Filename: 求一个字符串的所有排列（非递归法）.py
2. """
3. 题目：输入字符串，输出字符串的所有排列
4. 分析：
5. 1. 非递归法
6.     1. 对字符串中的字符进行排序（升序）
7.     2. 从右往左找到第一个递增的子字符串（两个字符）
8.     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
9.     4. 将pmin后的字符进行逆序
10.    5. 循环，知道找不到递增的子字符串
11. """
12. def swap(str, i, j):
13.     tmp = str[i]
14.     str[i] = str[j]
15.     str[j] = tmp
16. —
17. def Reverse(str, begin, end):
18.     i = begin
19.     j = end
20.     while i < j:
21.         swap(str, i, j)
22.         i += 1
23.         j -= 1
24. —
25. def getNextPermutation(str):
26.     end = len(str)-1
27.     cur = end
28.     suc = 0
29.     tmp = 0
30.     while cur != 0:
31.         suc = cur
32.         cur = cur - 1
33.         # 找到子字符串
34.         if str[cur] < str[suc]:
35.             tmp = end
36.             # 找到比cur大的最小字符串的字符
37.             while str[tmp] < str[cur]:
38.                 tmp -= 1
39.             # 交换
40.             swap(str, cur, tmp)
41.             # 逆序

```

```
42.         Reverse(str, suc, end)
43.     return True
44. return False
45. —
46. def Permutation(s):
47.     if s == None or len(s) < 1:
48.         return
49.     str = list(s)
50.     str.sort()
51.     print(str)
52.     while getNextPermutation(str):
53.         print(str)
54. —
55. if __name__ == "__main__":
56.     Permutation("abc")
```

```

1. # @Filename: 求两个字符串的最大公共子串 (动态规划法) .py
2. """
3. 题目: 求最长公共子串
4. 分析:
5. 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
6. """
7. —
8. def getMaxSubStr(str1, str2):
9.     len1 = len(str1)
10.    len2 = len(str2)
11.    sb = ""
12.    maxI = 0
13.    maxs = 0
14.    M = [[None]*(len1+1) for _ in range(len2+1)]
15.    i = 0
16.    # !!!
17.    while i < len1+1:
18.        M[0][i] = 0
19.        i += 1
20.        j = 0
21.        while j < len2+1:
22.            M[j][0] = 0
23.            j += 1
24.            i = 1
25.            while i < len1+1:
26.                j = 1
27.                while j < len2+1:
28.                    if str1[i-1] == str2[j-1]:
29.                        M[j][i] = M[j-1][i-1] + 1
30.                        if M[j][i] > maxs:
31.                            maxI = j
32.                            maxs = M[j][i]
33.                        else:
34.                            M[j][i] = 0
35.                            j += 1
36.                            i += 1
37.                            j = maxI - maxs
38.                            while j < maxI:
39.                                sb += str2[j]
40.                                j += 1
41.                            return sb

```



```
42. —  
43. if __name__ == "__main__":  
44.     str1 = "abccad"  
45.     str2 = "dgcadde"  
46.     print(getMaxSubStr(str1, str2))
```

```
1. # @Filename: 求两个字符串的最大公共子串（滑动比较法）.py
2. """
3. # 题目：求两个字符串的最大公共子串
4. 分析：
5. 1. 滑动比较法
6.     a. 保持s1不变，依次滑动s2，
7.     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
8.     c. 直到j > len(s1)-1
9. """
10. -
11. def getMaxSubStr(s1, s2):
12.     len1 = len(s1)
13.     len2 = len(s2)
14.     maxLen = 0
15.     maxLenEnd1 = 0
16.     sb = ""
17.     i = 0
18.     while i < len1 + len2:
19.         s1begin = s2begin = 0
20.         tmpMaxLen = 0
21.         if i < len1:
22.             s1begin = len1 - i
23.         else:
24.             s2begin = i - len1
25.         j = 0
26.         while s1begin + j < len1 and s2begin + j < len2:
27.             if s1[s1begin + j] == s2[s2begin + j]:
28.                 tmpMaxLen += 1
29.             else:
30.                 if tmpMaxLen > maxLen:
31.                     maxLen = tmpMaxLen
32.                     maxLenEnd1 = s1begin + j
33.             else:
34.                 tmpMaxLen = 0
35.                 j += 1
36.             if tmpMaxLen > maxLen:
37.                 maxLen = tmpMaxLen
38.                 maxLenEnd1 = s1begin + j
39.             i += 1
40.             i = maxLenEnd1 - maxLen
41.         while i < maxLenEnd1:
```

```
42.     sb += s1[i]
43.     i += 1
44.     return sb
45. —
46. if __name__ == "__main__":
47.     str1 = "abccade"
48.     str2 = "dgcadde"
49.     print(getMaxSubStr(str1, str2))
```

```

1. # @Filename: 求字符串里的最长回文子串 (manacher) .py
2. """
3. Manacher算法
4. 分析: 时间复杂度 $O(N)$ , 空间复杂度 $O(N)$ 
5. 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
6. 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
7. 2. 四种情况
8. 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
9. 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
10. 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
11. 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[ $2*id-i$ ]
12. 3. 综合以上情况 P[i] = min(P[ $2*id-i$ ], P[id]-i)
13. """
14. class Test:
15.     def __init__(self):
16.         self.center = None
17.         self.palindromeLen = None
18.     --
19.     def mins(self, a, b):
20.         return a if a < b else b
21.     --
22.     def getCenter(self):
23.         return self.center
24.     --
25.     def getLen(self):
26.         return self.palindromeLen
27.     --
28.     def Manacher(self, strs):
29.         lens = len(strs)
30.         newLen = 2 * lens + 1
31.         s = [None] * newLen
32.         p = [None] * newLen
33.         id = 0
34.         i = 0
35.         for i in range(newLen):
36.             if i % 2 == 0:
37.                 s[i] = '*'
38.             else:
39.                 s[i] = strs[i // 2]
40.                 p[i] = 0

```

```
41.     print(s)
42.     print(p)
43.     self.center = -1
44.     self.palindromeLen = -1
45.     i = 1
46.     while i < newLen:
47.         if id + p[id] > i:
48.             p[i] = self.mins(id + p[id] - i, p[2 * id - i])
49.         else:
50.             p[i] = 1
51.             while i + p[i] < newLen and i - p[i] > 0 and s[i - p[i]] == s[i + p[i]]:
52.                 p[i] += 1
53.                 if i + p[i] > id + p[id]:
54.                     # 此时后面的数将进入i点的回文领域
55.                     id = i
56.                     if p[i] - 1 > self.palindromeLen:
57.                         self.center = (i + 1) // 2 - 1
58.                         self.palindromeLen = p[i] - 1
59.                 i += 1
60.     print(1)
61. —
62.     if __name__ == "__main__":
63.         strs = "abcbax"
64.         t = Test()
65.         t.Manacher(strs)
66.         # 这里还要进行处理
67.         if t.palindromeLen % 2 == 0:
68.             print("fsdf", strs[t.getCenter() - (t.getLen() // 2 + 1):t.getCenter() +
(t.getLen() // 2 - 1)])
69.         else:
70.             print("ddd", strs[t.getCenter() - (t.getLen() // 2):t.getCenter() +
(t.getLen() // 2)])
```

```
1. # @Filename: 求字符串里的最长回文子串 (中心扩展法) .py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 中心扩展法:
6. 分析: 时间复杂度 ( $O(n^2)$ ), 空间复杂度 ( $O(1)$ )
7. 1. 当中心为一个点
8. 2. 当中心为两个点
9. """
10. —
11. —
12. class Test:
13.     def __init__(self):
14.         self.startIndex = None
15.         self.lens = 0
16.     —
17.     def getStartIndex(self):
18.         return self.startIndex
19.     —
20.     def getLens(self):
21.         return self.lens
22.     —
23.     def expandBothSide(self, strs, c1, c2):
24.         n = len(strs)
25.         while c1 >= 0 and c2 < n and strs[c1] == strs[c2]:
26.             c1 -= 1
27.             c2 += 1
28.             tmpStartIndex = c1 + 1
29.             tmpLen = c2 - c1 - 1
30.             if tmpLen > self.lens:
31.                 self.lens = tmpLen
32.                 self.startIndex = tmpStartIndex
33.     —
34.     def getLongestPalindrome(self, strs):
35.         if not strs:
36.             return
37.         n = len(strs)
38.         for i in range(n - 1):
39.             # 奇数个回文数, 中间只有一个点
40.             self.expandBothSide(strs, i, i)
41.             # 偶数个回文数, 中间有两个点
42.             self.expandBothSide(strs, i, i + 1)
43.     —
44.     —
```

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

```

1. # @Filename: 求字符串里的最长回文子串 (动态规划) .py
2. """
3. # 最长回文子串
4. 分析:
5. 动态规划:时间复杂度 $O(n^2)$ , 空间复杂度 $O(n^2)$ 
6. 1.
7. P[i,i] = 1
8. 如果Si != Si+1->P[i,i+1] = 0 else P[i,i+1] = 1
9. 如果Si = Sj -> P(i, j) = P(i+1,j-1)
10. """
11. —
12. —
13. class Test:
14.     def __init__(self):
15.         self.startIndex = None
16.         self.lens = None
17. —
18.     def getStartIndex(self):
19.         return self.startIndex
20. —
21.     def getLen(self):
22.         return self.lens
23. —
24.     def getLongestPalindrome(self, strs):
25.         if not strs:
26.             return
27.         n = len(strs)
28.         if n < 1:
29.             return
30.         self.startIndex = 0
31.         self.lens = 1
32.         historyRecord = [[0] * n for _ in range(n)]
33.         —
34.         # 更新长度为1的回文字符串信息
35.         for i in range(n):
36.             historyRecord[i][i] = 1
37.         —
38.         # 更新长度为2的回文字符串信息
39.         for i in range(n - 1):
40.             if strs[i] == strs[i + 1]:
41.                 historyRecord[i][i + 1] = 1
42.                 self.startIndex = i
43.                 self.lens = 2
44.         —

```



```
45.     # 从长度为3的字符串开始找起
46.     pLen = 3
47.     while pLen <= n:
48.         i = 0
49.         while i < n - pLen + 1:
50.             j = i + pLen - 1
51.             if strs[i] == strs[j] and historyRecord[i + 1][j - 1] == 1:
52.                 historyRecord[i][j] = 1
53.                 self.startIndex = i
54.                 self.lens = pLen
55.                 i += 1
56.                 pLen += 1
57.         —
58.         —
59.     if __name__ == "__main__":
60.         strs = "babad"
61.         t = Test()
62.         t.getLongestPalindrome(strs)
63.         if t.getStartIndex() != -1 and t.getLen() != -1:
64.             print("最长的回文子串为: ", strs[t.getStartIndex():t.getStartIndex() +
t.getLen()])
65.         else:
66.             print("查找失败")
```

```
1. # @Filename: 求解字符串中字典序最大的子序列（逆序遍历法）.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 分析:
6. 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
7. """
8. —
9. —
10. def getLargestSub(src):
11.     res = [src[-1]]
12.     i = len(src) - 2
13.     while i > -1:
14.         if src[i] >= res[0]:
15.             res.insert(0, src[i])
16.             i -= 1
17.     return "".join(res)
18. —
19. —
20. if __name__ == "__main__":
21.     print(getLargestSub("acbdxmng"))
```

```
1. # @Filename: 统计字符串中连续的重复字符个数.py
2. """
3. 分析:
4. 1. 普通遍历法, 增加两个变量maxLen和curMaxLen即可
5. 2. 采用递归法
6. """
7. def getMaxDupChar(s, startIndex, curMaxLen, maxLen):
8.     '''
9.     if startIndex == len(s)-1:
10.         return max(maxLen, curMaxLen)
11.     if s[startIndex] != s[startIndex + 1]:
12.         if maxLen < curMaxLen:
13.             maxLen = curMaxLen
14.             startIndex = startIndex + 1
15.             curMaxLen = 1
16.         return getMaxDupChar(s, startIndex, curMaxLen, maxLen)
17.     elif s[startIndex] == s[startIndex + 1]:
18.         return getMaxDupChar(s, startIndex+1, curMaxLen+1, maxLen)
19.     '''
20.     if startIndex + curMaxLen >= len(s):
21.         return max(maxLen, curMaxLen)
22.     if s[startIndex] != s[startIndex+curMaxLen]:
23.         if curMaxLen > maxLen:
24.             maxLen = curMaxLen
25.             startIndex += curMaxLen
26.             curMaxLen = 1
27.         return getMaxDupChar(s, startIndex, curMaxLen, maxLen)
28.     else:
29.         curMaxLen += 1
30.         return getMaxDupChar(s, startIndex, curMaxLen, maxLen)
31. —
32. if __name__ == "__main__":
33.     s = "aabbbbdddddfffdfffff"
34.     print(getMaxDupChar(s, 0, 1, 0))
```

4. 排列组合与概率

```
1. # @Filename: 如何判断还有几盏灯泡亮着.py
2. """
3. 判断还有几盏灯泡亮着
4. 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
5. 2. 拉满100次后，剩下亮着的灯泡的编号
6. —
7. 引申：如果没有拉满100次，如果拉了50次呢
8. 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）
9. """
10. def factorisOdd(a):
11.     i = 1
12.     count = 0
13.     while i <= a:
14.         if a % i == 0:
15.             count += 1
16.             i += 1
17.         if count % 2 == 0:
18.             return 0
19.         return 1
20. —
21. def totalCount(n):
22.     count = 0
23.     for i in range(1, n+1):
24.         if factorisOdd(i):
25.             count += 1
26.         print(i)
27. —
28. totalCount(100)
```

```
1. # @Filename: 如何拿到最多金币.py
2. """
3. 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
4. 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
5. 分析:
6. 1. 首先需要生成所有10个房间的金币出现的情况
7. 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
8. 3. 考虑n的值为1000
9. """
10. import random
11. def getMaxNum(count):
12.     # 先随机生成10个门的金币
13.     door = [random.randint(1, count) for _ in range(count)]
14.     max_4 = max(door[:4])
15.     i = 4
16.     while i < count:
17.         if door[i] >= max_4:
18.             return True
19.         i += 1
20.     return False
21. —
22. if __name__ == "__main__":
23.     n = 10000
24.     m = 0
25.     for i in range(n):
26.         m += int(getMaxNum(10))
27.     print(m/n)
```

```
1. # @Filename: 如何求数字的组合.py
2. """
3. 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
4. 条件: 4不能出现在第三位, 3和5不能相连
5. """
6. class Test:
7.     def __init__(self, arr):
8.         self.numbers = arr
9.         self.visited = [None] * len(self.numbers)
10.        self.graph = [[None] * len(self.numbers) for _ in range(len(self.numbers))]
11.        self.n = 6
12.        self.combination = ''
13.        self.s = set()
14.    -
15.    def depthFistSearch(self, start):
16.        self.visited[start] = True
17.        self.combination += str(self.numbers[start])
18.        # 遍历结束
19.        if len(self.combination) == self.n:
20.            # 判断4是否出现在第三位
21.            if self.combination.index("4") != 2:
22.                self.s.add(self.combination)
23.            pass
24.            j = 0
25.            while j < self.n:
26.                if self.graph[start][j] == 1 and self.visited[j] == False:
27.                    self.depthFistSearch(j)
28.                j += 1
29.            # 下面两行特别关键, 用于恢复递归前一次的状态
30.            self.combination = self.combination[:-1]
31.            self.visited[start] = False
32.    -
33.    def getAllCombinations(self):
34.        # 构造图
35.        i = 0
36.        while i < self.n:
37.            j = 0
38.            while j < self.n:
39.                if i == j:
40.                    self.graph[i][j] = 0
41.                else:
```

```
42.         self.graph[i][j] = 1
43.         j += 1
44.         i += 1
45.         # 确保3和5之间不可达
46.         self.graph[3][5] = 0
47.         self.graph[5][3] = 0
48.     -
49.         i = 0
50.         while i < self.n:
51.             self.depthFistSearch(i)
52.             i += 1
53.     -
54.     def printAllCombinations(self):
55.         print(self.s)
56.     -
57. if __name__ == "__main__":
58.     arr = [1, 2, 2, 3, 4, 5]
59.     t = Test(arr)
60.     t.getAllCombinations()
61.     t.printAllCombinations()
```

```
1. # @Filename: 如何求正整数n所有可能的整数组合.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 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
6. 分析: 典型的递归问题
7. 1. 递归参数: sums, result
8. 2. 递归结束条件: 当sums为0的时候, 打印result的值, 并返回
9. 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
10. """
11. —
12. —
13. def getAllcombination(sums, result):
14.     if sums == 0:
15.         print(''.join(result))
16.         return
17.     # 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
18.     # i = 1
19.     i = 1 if not result else int(result[-1])
20.     while i <= sums:
21.         result.append(str(i))
22.         getAllcombination(sums - i, result)
23.         result.pop()
24.         i += 1
25. —
26. —
27. getAllcombination(4, [])
```



```
1. # @Filename: 如何用一个随机函数得到另一个随机函数.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 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
6. 分析: 用两个已知随机函数得到的结果进行或即可
7. """
8. —
9. import random
10. —
11. —
12. def get_random():
13.     a1 = int(round(random.random()))
14.     a2 = int(round(random.random()))
15.     result = a1 | a2
16.     return result
17. —
18. —
19. n = 0
20. for i in range(100000):
21.     n += get_random()
22. print(n / 100000)
```

```
1. # @Filename: 如何等概率地从大小为n的数组中选取m个整数.py
2. """
3. 分析:
4. 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
5. 1. 第一次  $1/n$ 
6. 2.  $[(n-1)/n] * [1/(n-1)] = 1/n$ 
7. —
8. 具体方案:
9. 将随机选到的值与最前面的值进行交换
10. """
11. import random
12. def getRandomM(a, n, m):
13.     i = 0
14.     res = []
15.     while i < m:
16.         j = random.randint(i, n - 1)
17.         res.append(a[j])
18.         a[j], a[i] = a[i], a[j]
19.         i += 1
20.     return res
21. print(getRandomM([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 10, 6))
```

```

1. # @Filename: 如何组合1, 2, 5这三个数使其和为100.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 分析:
6. 1. 蛮力法
7. 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
8. 2. 数字规律法
9. a.  $x+5z = 100-2y$ , 所以 $x+5z$ 为偶数
10. b. 依次列举 $z$ 的值 (0-20), 求得对应给的 $x$ 与 $y$ 
11. """
12. def combinationCount(n):
13.     count = 0
14.     m = 0
15.     while m <= n:
16.         count += (n - m + 2) // 2
17.         x = 0 if m % 2 == 0 else 1
18.         while x <= n - m:
19.             print("{}+2x{}+5x{}={}".format(x, (n - x - m) // 2, m // 5, n))
20.             x += 2
21.             m += 5
22.         return count
23. print(combinationCount(100))

```

5. 排序

```
1. # @Filename: 冒泡排序.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 分析:
6. 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
7. 1. 是一种稳定的排序方法
8. 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
9. """
10. —
11. —
12. def bubble_sort(arr):
13.     for i in range(len(arr) - 1):
14.         for j in range(1, len(arr) - i):
15.             if arr[j - 1] > arr[j]:
16.                 arr[j - 1], arr[j] = arr[j], arr[j - 1]
17. —
18. —
19. arr = [3, 4, 2, 8, 9, 5, 1]
20. bubble_sort(arr)
21. print(arr)
```

```
1. # @Filename: 基数排序.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 分析:
6. 1. 根据个位数把数字分配到0-9变好的桶中
7. 2. 将这些树串联起来
8. 3. 接着再对十位数、百位数分配
9. """
10. —
11. import math
12. —
13. —
14. def radix_sort(arr, radix=10):
15.     # 计算最大数据的最多位数
16.     k = int(math.ceil(math.log(max(arr), radix)))
17.     bucket = [[] for _ in range(radix)]
18.     for i in range(1, k + 1):
19.         for j in arr:
20.             bucket[j // (radix ** (i - 1)) % (radix)].append(j)
21.         del arr[:]
22.         for z in bucket:
23.             arr += z
24.         del z[:]
25.     return arr
```

```
1. # @Filename: 堆排序.py
2. """
3. 分析:
4. 不稳定的排序方法
5. T: 最坏情况也是O (nlogn)
6. """
7. —
8. def adjust_heap(arr, i, size):
9.     while True:
10.         maxs = i
11.         lchild = 2 * i + 1
12.         rchild = 2 * i + 2
13.         if i < size // 2:
14.             if lchild < size and arr[lchild] > arr[maxs]:
15.                 maxs = lchild
16.             if rchild < size and arr[rchild] > arr[maxs]:
17.                 maxs = rchild
18.             if maxs != i:
19.                 arr[maxs], arr[i] = arr[i], arr[maxs]
20.                 i = maxs
21.             else:
22.                 break
23.         else:
24.             break
25. —
26. def heap_sort(arr):
27.     first = len(arr) // 2 - 1
28.     for i in range(first, -1, -1):
29.         print(arr[i])
30.         adjust_heap(arr, i, len(arr) - 1)
31.     # 交换堆顶和堆尾
32.     for head_end in range(len(arr)-1, 0, -1):
33.         arr[head_end], arr[0] = arr[0], arr[head_end]
34.         # 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
35.         adjust_heap(arr, 0, head_end-1)
36. —
37. if __name__ == "__main__":
38.     arr = [3, 4, 2, 8, 9, 5, 1]
39.     print(arr)
40.     heap_sort(arr)
41.     print(arr)
```

```
1. # @Filename: 堆排序.py
2. """
3. 分析:
4. 不稳定的排序方法
5. T: 最坏情况也是O (nlogn)
6. """
7. def adjust_heap(arr, i, size):
8.     while True:
9.         maxs = i
10.        lchild = 2 * i + 1
11.        rchild = 2 * i + 2
12.        if i < size // 2:
13.            if lchild < size and arr[lchild] > arr[maxs]:
14.                maxs = lchild
15.            if rchild < size and arr[rchild] > arr[maxs]:
16.                maxs = rchild
17.            if maxs != i:
18.                arr[maxs], arr[i] = arr[i], arr[maxs]
19.                i = maxs
20.            else:
21.                break
22.        else:
23.            break
24. —
25. def heap_sort(arr):
26.     first = len(arr) // 2 - 1
27.     for i in range(first, -1, -1):
28.         print(arr[i])
29.         adjust_heap(arr, i, len(arr) - 1)
30.     # 交换堆顶和堆尾
31.     for head_end in range(len(arr)-1, 0, -1):
32.         arr[head_end], arr[0] = arr[0], arr[head_end]
33.         # 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
34.         adjust_heap(arr, 0, head_end-1)
35. —
36. if __name__ == "__main__":
37.     arr = [3, 4, 2, 8, 9, 5, 1]
38.     print(arr)
39.     heap_sort(arr)
40.     print(arr)
```

```
1. # @Filename: 归并排序.py
2. """
3. 分析:
4. 归并排序:
5. 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
6. 2. 将其两两归并
7. 3. 重复1和2直到得到一个有序序列为止
8. —
9. 是一种稳定的算法, T: O (nlogn)   S:O(n)
10. """
11. def merge(left, right):
12.     i, j = 0, 0
13.     result = []
14.     while i < len(left) and j < len(right):
15.         # print(left[i], right[j], right)
16.         if left[i] < right[j]:
17.             result.append(left[i])
18.             i += 1
19.         else:
20.             result.append(right[j])
21.             j += 1
22.     # 添加某边还剩余的部分
23.     result += left[i:]
24.     result += right[j:]
25.     print(result)
26.     return result
27. —
28. def merge_sort(arr):
29.     if len(arr) <= 1:
30.         return arr
31.     num = len(arr) // 2
32.     left = merge_sort(arr[:num])
33.     right = merge_sort(arr[num:])
34.     return merge(left, right)
35. —
36. arr = [3, 4, 2, 8, 9, 5, 1]
37. print(merge_sort(arr))
```



```

1. # @Filename: 快速排序.py
2. """
3. # 分析:
4. 1. 设置两个变量left, right, 最开始初始化left=0, right= n-1
5. 2. 设置关键值key = arr[left]
6. 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
7. 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
8. 5. 以上步骤知道left == right
9. —
10. 6. 1-5后, 会得到两组数据, 对每组数据递归调用1-5
11. —
12. *: 当初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
13. T: 最坏情况下O (n^2) , 最好情况O (nlogn), 平均O (nlogn) , S (logn)
14. —
15. *: 不稳定的排序算法
16. """
17. def quick_sort(arr, left, right):
18.     if left < right:
19.         i, j = left, right
20.         key = arr[i]
21.         —
22.         while i < j:
23.             # 循环从右边遍历
24.             while i < j and arr[j] >= key:
25.                 j -= 1
26.             arr[i] = arr[j]
27.             # 循环从左边遍历
28.             while i < j and arr[i] <= key:
29.                 i += 1
30.             arr[j] = arr[i]
31.             # 将key赋予中间值
32.             arr[i] = key
33.         —
34.         quick_sort(arr, left, i - 1)
35.         quick_sort(arr, i + 1, right)
36.         —
37. arr = [3, 4, 2, 8, 9, 5, 1]
38. quick_sort(arr, 0, len(arr)-1)
39. print(arr)

```

```

1. # @Filename: 快速排序.py
2. """
3. # 分析:
4. 1. 设置两个变量left, right, 最开始初始化left=0, right= n-1
5. 2. 设置关键值key = arr[left]
6. 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
7. 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
8. 5. 以上步骤知道left == right
9. —
10. 6. 1-5后, 会得到两组数据, 对每组数据递归调用1-5
11. —
12. *: 当初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
13. T: 最坏情况下O (n^2), 最好情况O (nlogn), 平均O (nlogn), S (logn)
14. —
15. *: 不稳定的排序算法
16. """
17. def quick_sort(arr, left, right):
18.     if left < right:
19.         i, j = left, right
20.         key = arr[i]
21.         —
22.         while i < j:
23.             # 循环从右边遍历
24.             while i < j and arr[j] >= key:
25.                 j -= 1
26.             arr[i] = arr[j]
27.             # 循环从左边遍历
28.             while i < j and arr[i] <= key:
29.                 i += 1
30.             arr[j] = arr[i]
31.             # 将key赋予中间值
32.             arr[i] = key
33.         —
34.         quick_sort(arr, left, i - 1)
35.         quick_sort(arr, i + 1, right)
36.         —
37. arr = [3, 4, 2, 8, 9, 5, 1]
38. quick_sort(arr, 0, len(arr)-1)
39. print(arr)

```

```
1. # @Filename: 选择排序.py
2. """
3. 分析:
4. 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
5. 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
6. 2. T: 无论好坏都是  $O(n^2)$ , S:  $O(1)$ : min
7. """
8. def select_sort(arr):
9.     i = 0
10.    while i < len(arr) - 1:
11.        min = i
12.        j = i + 1
13.        while j < len(arr):
14.            if arr[j] < arr[min]:
15.                min = j
16.            j += 1
17.        # 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
18.        arr[i], arr[min] = arr[min], arr[i]
19.        i += 1
20.        print("第{}次排序: {}".format(i, arr))
21.    —
22.    arr = [3, 4, 2, 8, 9, 5, 1]
23.    select_sort(arr)
24.    print(arr)
```

6. 数组

```
1. # @Filename: 从三个有序数组中找出他们的公共元素.py
2. """
3. 数组为递增数组
4. 分析: 分别设定i, j, k进行三个数组的遍历
5. 1. 如果a[i] < b[j], 则i++, 直到a[i] >= a[j]
6. 2. 如果a[j] < b[i], 则j++, 直到b[j] >= a[i]
7. 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],
8. """
9. def findCommon(ar1, ar2, ar3):
10.     i = j = k = 0
11.     common = []
12.     while i < len(ar1) and j < len(ar2) and k < len(ar3):
13.         if ar1[i] < ar2[j]:
14.             i += 1
15.         elif ar1[i] > ar2[j]:
16.             j += 1
17.         else:
18.             k += 1
19.             if ar1[i] == ar2[j] == ar3[k]:
20.                 common.append(ar1[i])
21.                 i += 1
22.     return common
```

```
1. # @Filename: 判断请求能否在给定的存储条件下完成.py
2. def swap(arr, i, j):
3.     tmp = arr[i]
4.     arr[i] = arr[j]
5.     arr[j] = tmp
6. —
7. def bubbleSort(R, O):
8.     # 冒泡法, 由大到小排序
9.     lens = len(R)
10.    i = 0
11.    while i < lens - 1:
12.        j = lens - 1
13.        while j > i:
14.            if R[j] - O[j] > R[j - 1] - O[j - 1]:
15.                swap(R, j, j - 1)
16.                swap(O, j, j - 1)
17.            j -= 1
18.        i += 1
19. —
20. def schedule(R, O, M):
21.     # 先按照R-O的大小对R和O就行排序
22.     bubbleSort(R, O)
23.     # 判断按照当前顺序是否能填满
24.     total = 0
25.     i = 0
26.     while i < len(R):
27.         total += R[i]
28.         if total > M:
29.             return False
30.         else:
31.             total -= R[i]
32.             total += O[i]
33.             i += 1
34.     return True
35. —
36. if __name__ == "__main__":
37.     R = [10, 15, 23, 20, 6, 9, 7, 16]
38.     O = [2, 7, 8, 4, 5, 8, 6, 8]
39.     N = 8
40.     M = 50
41.     scheduleResult = schedule(R, O, M)
42.     print(scheduleResult)
```

43.

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

```

1. # @Filename: 在不排序的情况下求数组中的中位数.py
2. # 中位数: 奇数个数据的数组, 中位数 $k = \text{arr}((e-s)/2)$ ; 偶数个数据的数组, 中位数 $k = \text{arr}((e-s)/2) + \text{arr}((e-s)/2 + 1)$ 
3. """
4. 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
5. """
6. def partition(arr, low, high):
7.     key = arr[low]
8.     while low < high:
9.         while low < high and arr[high] > key:
10.             high -= 1
11.         arr[low] = arr[high]
12.         while low < high and arr[low] < key:
13.             low += 1
14.         arr[high] = arr[low]
15.     arr[low] = key
16.     return low
17. —
18. def getMid(arr):
19.     le = len(arr)
20.     k = (le - 1) // 2
21.     —
22.     low = 0
23.     high = le - 1
24.     while True:
25.         # 将数组分为两半
26.         pos = partition(arr, low, high)
27.         if pos + 1 == k:
28.             break
29.         elif pos + 1 < k:
30.             low = pos + 1
31.         else:
32.             high = pos - 1
33.     return arr[k] if le % 2 == 1 else (arr[k] + arr[k + 1]) / 2
34. —
35. if __name__ == "__main__":
36.     arr = [7, 5, 3, 1, 11, 9]
37.     print(getMid(arr))

```

1. # @Filename: 在有规律的二维数组中进行高效的数据查找.py

2. """

3. 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

4. 分析:

5. 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

6. """

7. def findWithBinary(arr, data):

8. if not arr:

9. return False

10. row = 0

11. col = len(arr[0])-1

12. while col >= 0 and row < len(arr):

13. if arr[row][col] == data:

14. return True

15. elif arr[row][col] > data:

16. col -= 1

17. elif arr[row][col] < data:

18. row += 1

19. return False

20. —

21. if __name__ == "__main__":

22. arr = [[0, 1, 2, 3, 4], [10, 11, 12, 13, 14]]

23. print(findWithBinary(arr, 10))


```
1. # @Filename: 如何对任务进行调度.py
2. """
3. 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
4. 分析:
5. 1. 贪心法
6. """
7. def calculate_process_time(t, n):
8.     if t == None:
9.         return None
10.    m = len(t)
11.    proTime = [0] * m
12.    i = 0
13.    while i < n:
14.        minTime = proTime[0] + t[0]
15.        minIndex = 0
16.        j = 1
17.        while j < m:
18.            if minTime > proTime[j] + t[j]:
19.                minTime = proTime[j] + t[j]
20.                minIndex = j
21.            j += 1
22.        proTime[minIndex] += t[minIndex]
23.        i += 1
24.    return proTime
25. —
26. if __name__ == "__main__":
27.     t = [7, 10]
28.     n = 6
29.     proTime = calculate_process_time(t, n)
30.     print(proTime)
```

```

1. # @Filename: 如何对数组进行循环位移.py
2. # 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
3. # 和之前旋转数组的实现一样的
4. """
5. 1. 循环移动法
6. 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
7. 3. 翻转法: 分为A 和B , 先翻转A, 在翻转B, 最后翻转A+B
8. """
9. def reverse(arr, start, end):
10.     # 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
11.     arr[start:end] = arr[start:end][::-1]
12. —
13. def rightShift(arr, k):
14.     if not arr:
15.         return
16.     lens = len(arr)
17.     reverse(arr, 0, lens-k)
18.     reverse(arr, lens-k, lens)
19.     reverse(arr, 0, lens)
20. —
21. if __name__ == "__main__":
22.     k = 4
23.     arr = [1, 2, 3, 4, 5, 6, 7, 8]
24.     rightShift(arr, k)
25.     print(arr)

```

```
1. # @Filename: 如何对磁盘分区.py
2. """
3. 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*
4. 分析:
5. 依次比较即可
6. """
7. def is_allocate(d, p):
8.     i = 0
9.     j = 0
10.    while i < len(d):
11.        if d[i] >= p[j]:
12.            remain = d[i] - p[j]
13.            j += 1
14.            if j == len(p):
15.                return True
16.        else:
17.            i += 1
18.            continue
19.        while remain >= p[j]:
20.            j += 1
21.            if j == len(p):
22.                return True
23.        remain = remain - p[j]
24.        i += 1
25.    return False
26. -
27. if __name__ == "__main__":
28.     d = [120, 120, 120]
29.     p = [60, 80, 80, 20, 80]
30.     print(is_allocate(d, p))
```

```
1. # @Filename: 如何寻找最多的覆盖点.py
2. # 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
3. """
4. 分析:
5. 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++
6. """
7. def maxCover(arr, L):
8.     maxCount = 1
9.     lens = len(arr)
10.    i = 0
11.    j = 1
12.    while j < lens:
13.        dis_sum = arr[j] - arr[i]
14.        if dis_sum > L:
15.            j -= 1
16.            if j - i + 1 > maxCount:
17.                maxCount = j - i + 1
18.            i += 1
19.            j += 1
20.            j += 1
21.    return maxCount
22. -
23. if __name__ == "__main__":
24.     a = [1, 3, 7, 8, 10, 11, 12, 13, 15, 16, 17, 19, 25]
25.     print(maxCover(a, 8))
```

```

1. # @Filename: 如何求两个有序集合的交集.py
2. """
3. 题目: 两个有序集合, 集合中的元素为一段范围 {[4,8], [9,13]}
4. 分析:
5. 1. 不利用有序集合这一特性, 进行两个循环遍历
6. 2. 利用有序集合特性
7. a. 对于a[i]与b[j]
8. 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
9. 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
10. 1. b[j][0] < a[i][0] < b[j][1] and b[j][0] < a[i][1] < b[j][1]: 挨着, 记录交集, 交集为a[i], 后面i++有可能挨着
11. 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
12. 1. b[j][0] < a[i][0]: 不挨着, 后面j++有可能挨着
13. 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
14. """
15. —
16. def getIntersection(a, b):
17.     result = []
18.     i = j = 0
19.     while j < len(b) and i < len(a):
20.         if a[i][0] < b[j][0] and a[i][1] < b[j][0]:
21.             i += 1
22.         elif a[i][0] < b[j][0] and b[j][0] <= a[i][1] < b[j][1]:
23.             result.append([b[j][0], a[i][1]])
24.             i += 1
25.         elif b[j][0] <= a[i][0] < b[j][1] and b[j][0] < a[i][1] < b[j][1]:
26.             result.append([a[i][0], a[i][1]])
27.             i += 1
28.         elif b[j][0] < a[i][0] <= b[j][1] and a[i][1] > b[j][1]:
29.             result.append([a[i][0], b[j][1]])
30.             j += 1
31.         elif b[j][0] < a[i][0]:
32.             j += 1
33.         else:
34.             result.append([b[j][0], b[j][1]])
35.             j += 1
36.     return result
37. —
38. if __name__ == "__main__":
39.     l1 = [[4, 8], [9, 13]]

```

```
40.     l2 = [[6, 12]]
41.     result = getIntersection(l1, l2)
42.     i = 0
43.     print(result)
```

```
1. # @Filename: 如何求集合的所有子集.py
2. """
3. 迭代法:
4. 1. set1中首先放入一个元素
5. 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)
6. 3. 循环操作
7. """
8. -
9. def getAllSub(str):
10.     if not str:
11.         print("参数不合理")
12.         return None
13.     arr = []
14.     # 将第一个节点加入集合中
15.     arr.append(str[0])
16.     i = 1
17.     while i < len(str):
18.         lens = len(arr)
19.         j = 0
20.         # 迭代轮次, 添加构造组合
21.         while j < lens:
22.             arr.append(arr[j] + str[i])
23.             j += 1
24.         arr.append(str[i])
25.         i += 1
26.     return arr
27. -
28. if __name__ == "__main__":
29.     result = getAllSub("abc")
30.     print(result)
```

```

1. # @Filename: 如何获得最好的矩阵链相乘方法 (动态规划-重点) .py
2. """
3. 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
4. 分析:
5. 1. 使用递归法
6. 2. 使用动态规划
7. 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
8. """
9. def MatrixChainOrder(p, n):
10.     cost = [[None] * n for _ in range(n)]
11.     i = 1
12.     while i < n:
13.         # 本身到本身的计算次数为0
14.         cost[i][i] = 0
15.         i += 1
16.         cLen = 2
17.         while cLen < n:
18.             i = 1
19.             while i < n - cLen + 1:
20.                 j = i + cLen - 1
21.                 cost[i][j] = 2 ** 31
22.                 k = i
23.                 while k < j:
24.                     q = cost[i][k] + cost[k + 1][j] + p[i - 1] * p[k] * p[j]
25.                     if q < cost[i][j]:
26.                         cost[i][j] = q
27.                     k += 1
28.                 i += 1
29.                 cLen += 1
30.             return cost[1][n - 1]
31. —
32. if __name__ == "__main__":
33.     arr = [1, 5, 2, 4, 6]
34.     n = len(arr)
35.     print(str(MatrixChainOrder(arr, n)))

```



```
1. # @Filename: 如何获得最好的矩阵链相乘方法（递归法）.py
2. """
3. 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
4. 分析：
5. 1. 使用递归法
6. """
7. def MatrixChainOrder(p, i, j):
8.     # p为数组， i和j为括号
9.     if i == j:
10.         return 0
11.     mins = 2 ** 32
12.     k = i
13.     while k < j:
14.         count = MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k + 1, j) + p[i - 1]
            * p[k] * p[j]
15.         if count < mins:
16.             mins = count
17.         k += 1
18.     return mins
19. —
20. if __name__ == "__main__":
21.     arr = [1, 5, 2, 4, 6]
22.     n = len(arr)
23.     print(str(MatrixChainOrder(arr, 1, n-1)))
```

```
1. # @Filename: 对二维数组进行旋转.py
2. """
3. 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
4. """
5. def rotateArr(arr):
6.     le = len(arr[0])
7.     for i in range(len(arr[0])):
8.         col = le - i
9.         row = 0
10.        while col < le:
11.            print(arr[row][col], end=" ")
12.            col += 1
13.            row += 1
14.        print()
15.        le1 = len(arr)
16.        for i in range(le1):
17.            row = 0 + i
18.            col = 0
19.            while row < le1:
20.                print(arr[row][col], end=" ")
21.                col += 1
22.                row += 1
23.            print()
24. —
25. if __name__ == "__main__":
26.     arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
27.     rotateArr(arr)
```

```
1. # @Filename: 对大量有重复的数字的数组排序 (字典法) .py
2. """
3. 题目: 对有大量重复数字的数组排序
4. 分析:
5. 1. 利用AVL树
6. 2. 利用hash法
7. """
8. —
9. def sort(arr):
10.     num_count = {}
11.     for i in arr:
12.         if i in num_count:
13.             num_count[i] += 1
14.         else:
15.             num_count[i] = 1
16.     keys = sorted(num_count.keys(), key=lambda i: num_count[i])
17.     i = 0
18.     k = 0
19.     while i < len(arr):
20.         j = 0
21.         while j < num_count[keys[k]]:
22.             arr[i + j] = keys[k]
23.             j += 1
24.             k += 1
25.             i += j
26.     print(arr)
27.     return arr
28. —
29. if __name__ == "__main__":
30.     arr = [15, 12, 15, 2, 2, 12, 2, 3, 12, 100, 3, 3]
31.     sort(arr)
```

```
1. # @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
2. # 和找出出现奇数次的数（两个奇数）差不多
3. """
4. 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，
5. 2. 得到这个数后，后面还能继续求解剩下的数据
6. """
7. def isOne(n, i):
8.     return (n >> i & 1) == 1
9. -
10. def findSingle(arr):
11.     d = 0
12.     for i in arr:
13.         d ^= i
14.     i = 0
15.     while d >> i:
16.         arr1 = 0
17.         arr2 = 0
18.         len_1 = 0
19.         len_2 = 0
20.         if isOne(d, i):
21.             # 对原数组分为两组
22.             for j in arr:
23.                 if j >> i & 1 == 1:
24.                     arr1 ^= j
25.                     len_1 += 1
26.                 else:
27.                     arr2 ^= j
28.                     len_2 += 1
29.             if len_2 % 2 == 0 and arr2 != 0:
30.                 return arr1
31.             elif len_1 % 2 == 0 and arr1 != 0:
32.                 return arr2
33.             else:
34.                 i += 1
35.         else:
36.             i += 1
37. -
38. if __name__ == "__main__":
39.     arr = [6, 3, 4, 5, 9, 4, 3]
40.     print(findSingle(arr))
```

```
1. # @Filename: 找出数组中出现奇数次的数 (只有两个奇数) .py
2. # 前提: 只有两个元素出现奇数次
3. # 1. 词典法: 记录出现次数
4. # 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
5. def get2Num(arr):
6.     # 首先全部异或
7.     c = 0
8.     for i in arr:
9.         c ^= i
10.    # 找到c中不为0的位置
11.    position = 0
12.    tmpResult = c
13.    while tmpResult & 1 == 0:
14.        position += 1
15.        tmpResult = tmpResult >> 1
16.    a = c
17.    for i in arr:
18.        if i >> position & 1 == 1:
19.            a ^= i
20.    b = c ^ a
21.    return a, b
22. —
23. if __name__ == "__main__":
24.     arr = [3, 5, 6, 6, 5, 7, 2, 2]
25.     print(get2Num(arr))
```

```
1. # @Filename: 找出数组中唯一重复元素2 (累加求和法) .py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 这里的前提是, 数组里面的数据是连续的
5. # 时间复杂度为 $O(N)$ , 空间复杂度为 $O(1)$ 
6. —
7. def findDup(arr):
8.     mi = min(arr)
9.     ma = max(arr)
10.     return sum(arr) - sum([i for i in range(mi, ma + 1)])
11. —
12. —
13. if __name__ == "__main__":
14.     arr = [1, 3, 4, 5, 2, 5, 6]
15.     print(findDup(arr))
```

```
1. # @Filename: 找出数组中唯一重复元素3 (异或法) .py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 同样建立在数据是单一连续的基础上
5. # time:O(n) space:O(1)
6. —
7. def findDup(arr):
8.     mi = min(arr)
9.     ma = max(arr)
10.    l = ma-mi+1
11.    res = 0
12.    for i in range(len(arr)):
13.        res ^= arr[i]
14.        for i in range(l):
15.            res ^= (mi+i)
16.    return res
17. —
18. if __name__ == "__main__":
19.    arr = [1, 3, 4, 5, 2, 5, 6]
20.    print(findDup(arr))
```

```
1. # @Filename: 找出数组中唯一重复元素4 (数据映射法) .py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 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
5. # 采用剑指offer效果更好
6. --
7. def findDup(arr):
8.     index = 0
9.     while 1:
10.         if arr[index] > 0:
11.             arr[index] *= -1
12.             index = -arr[index]
13.         else:
14.             return -arr[index]
15. --
16. --
17. if __name__ == "__main__":
18.     arr = [1, 3, 4, 5, 2, 2, 6]
19.     print(findDup(arr))
```



```
1. # @Filename: 找出数组中唯一重复的元素1 (字典法) .py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. --
5. # 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
6. # 时间复杂度 :  $O(N)$  , 空间复杂度 $O(N)$ 
7. --
8. def findDup(arr):
9.     dic = {}
10.    for i in arr:
11.        if not dic.get(i, 0):
12.            dic[i] = 1
13.        else:
14.            return i
15.    --
16.    --
17.    if __name__ == "__main__":
18.        arr = [1, 3, 4, 1, 2, 5, 6]
19.        print(findDup(arr))
```

```
1. # @Filename: 找出数组中的最大值和最小值 (分治一次) .py
2. # 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
3. # 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
4. class MaxMin:
5.     def __init__(self):
6.         self.max = None
7.         self.min = None
8.     --
9.     def getMax(self):
10.         return self.max
11.     --
12.     def getMin(self):
13.         return self.min
14.     --
15.     def getMaxAndMin(self, arr):
16.         if not arr:
17.             print("参数不合法")
18.             return
19.             i = 0
20.             lens = len(arr)
21.             while i < lens-1:
22.                 if arr[i] > arr[i + 1]:
23.                     tmp = arr[i]
24.                     arr[i] = arr[i + 1]
25.                     arr[i + 1] = tmp
26.                     i += 2
27.             i = 1
28.             self.max = arr[1]
29.             while i < lens:
30.                 if arr[i] > self.max:
31.                     self.max = arr[i]
32.                     i += 2
33.             i = 0
34.             self.min = arr[0]
35.             while i < lens:
36.                 if arr[i] < self.min:
37.                     self.min = arr[i]
38.                     i += 2
39. if __name__ == "__main__":
40.     arr = [7, 3, 19, 40, 4, 7, 1]
41.     m = MaxMin()
42.     m.getMaxAndMin(arr)
```

43. `print(m.getMax())`

44. `print(m.getMin())`

```
1. # @Filename: 找出数组中的最大值和最小值 (分治递归) .py
2. class MaxMin:
3.     def __init__(self):
4.         self.max = None
5.         self.min = None
6.     -
7.     def getMax(self):
8.         return self.max
9.     -
10.    def getMin(self):
11.        return self.min
12.    -
13.    def getMaxAndMin(self, arr, l, r):
14.        if not arr:
15.            print("参数不合法")
16.            return
17.            list = []
18.            m = (l + r) // 2
19.            if l == r: # l与r之间只存在一个点
20.                list.append(arr[l])
21.                list.append(arr[r])
22.                return list
23.            if l + 1 == r: # 之间存在两个点
24.                list.append(arr[l] if arr[l] <= arr[r] else arr[r])
25.                list.append(arr[r] if arr[l] < arr[r] else arr[l])
26.                return list
27.            lList = self.getMaxAndMin(arr, l, m)
28.            rList = self.getMaxAndMin(arr, m + 1, r)
29.            max = lList[1] if lList[1] > rList[1] else rList[1]
30.            min = lList[0] if lList[0] < rList[0] else rList[0]
31.            list.append(min)
32.            list.append(max)
33.            return list
34.    -
35.    if __name__ == "__main__":
36.        arr = [7, 3, 19, 40, 4, 7, 1]
37.        m = MaxMin()
38.        result = m.getMaxAndMin(arr, 0, len(arr) - 1)
39.        print(result[0])
40.        print(result[1])
```

```

1. # @Filename: 找出数组中第k小的数.py
2. """
3. 1. 排序法: 排好序之后, 取下标为k-1的数据即可  $O(n\log n)$ 
4. 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  $O(k*n)$ 
5. 3. 类似快速排序方法
6. """
7. def findSmallK(arr, low, high, k):
8.     i = low
9.     j = high
10.    splitElem = arr[i]
11.    # 将数据分为三部分, 左边小于splitElem, 中间arr[i]=splitElem, 右边大于splitElem
12.    while i < j:
13.        while i < j and arr[j] >= splitElem:
14.            j -= 1
15.        if i < j:
16.            arr[i] = arr[j]
17.            i += 1
18.        while i < j and arr[i] <= splitElem:
19.            i += 1
20.        if i < j:
21.            arr[j] = arr[i]
22.            j -= 1
23.        arr[i] = splitElem
24.        subArrayIndex = i - low
25.        if subArrayIndex == k - 1:
26.            return arr[i]
27.        elif subArrayIndex > k - 1:
28.            # 第k小的数依然在左边
29.            return findSmallK(arr, low, i - 1, k)
30.        else:
31.            # 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)
32.            return findSmallK(arr, i + 1, high, k - (i - low + 1))
33. if __name__ == "__main__":
34.     k = 3
35.     arr = [4, 0, 1, 0, 2, 3]
36.     print(findSmallK(arr, 0, len(arr) - 1, k))

```

```
1. # @Filename: 找出数组中绝对值最小的数.py
2. """
3. 前提: 升序数组
4. 采用二分法查找:
5. 1. arr[mid] = 0,
6. 2. arr[mid]<0 arr[mid+1]>0: min(-arr[mid], arr[mid+1])
7. 3. 如果arr[mid-1]<0 arr[mid]>0: min(-arr[mid-1], arr[mid])
8. 4. arr[mid]>0, 最小值在左边
9. 5. arr[mid]<0, 最小值在右边
10. """
11. def findMin(arr):
12.     if not arr:
13.         print("输入参数不合理")
14.         return 0
15.     if arr[0] > 0:
16.         return arr[0]
17.     if arr[-1] < 0:
18.         return arr[-1]
19.     begin = 0
20.     end = len(arr) - 1
21.     while True:
22.         mid = begin + (end - begin) // 2
23.         if arr[mid] == 0:
24.             return 0
25.         elif arr[mid] > 0:
26.             if arr[mid - 1] > 0:
27.                 end = mid - 1
28.             elif arr[mid] == 0:
29.                 return 0
30.         else:
31.             return min(-arr[mid - 1], arr[mid])
32.         elif arr[mid] < 0:
33.             if arr[mid + 1] < 0:
34.                 begin = mid + 1
35.             elif arr[mid + 1] == 0:
36.                 return 0
37.         else:
38.             return min(-arr[mid], arr[mid + 1])
39. if __name__ == "__main__":
40.     arr = [-10, -5, -2, 7, 15, 50]
41.     print(findMin(arr))
```

```
1. # @Filename: 找出数组连续最大和（动态规划-经典）.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. """
5. 经典的动态规划题:
6. 当访问到第i个节点时:
7. 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])
8. 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])
9. """
10. def maxSubarr(arr):
11.     if not arr:
12.         return 0
13.     # 负责记录连续的子数组的和的值
14.     nEnd = arr[0]
15.     # 负责记录最大的子数组的和的值
16.     nAll = arr[0]
17.     for i in range(1, len(arr)):
18.         if nEnd + arr[i] < arr[i]:
19.             nEnd = arr[i]
20.         else:
21.             nEnd += arr[i]
22.         nAll = max(nAll, nEnd)
23.     return nAll
```

```

1. # @Filename: 找出旋转数组中的最小值.py
2. # 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
3. # 两个指针 low high, mid = (high+low)//2
4. # 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
5. # 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)
6. def getMin(arr, low, high):
7.     if high < low:
8.         # ???,这里貌似是没有必要判断的
9.         return arr[0]
10.    if high == low:
11.        return arr[low]
12.    # mid = (low+high)/2
13.    # 防止low+high溢出
14.    mid = low + ((high - low) >> 1)
15.    —
16.    if arr[mid] > arr[mid + 1]:
17.        return arr[mid + 1]
18.    # 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
19.    elif arr[mid - 1] > arr[mid]:
20.        return arr[mid]
21.    elif arr[high] > arr[mid]:
22.        # 最小值在左边
23.        return getMin(arr[:mid], low, mid - 1)
24.    elif arr[low] < arr[mid]:
25.        return getMin(arr[mid + 1:], mid + 1, high)
26.    else:
27.        # 对于一些特殊情况，如[2,2,2,1,2,2]和[2,1,2,2,2,2]，无法确定是在左边还是右边
28.        return min(getMin(arr[:mid], low, mid - 1), getMin(arr[mid + 1:], mid + 1, high))
29. if __name__ == "__main__":
30.     # array1 = [5, 6, 1, 2, 3, 4]
31.     # mins = getMin(array1, 0, len(array1) - 1)
32.     # print(mins)
33.     # array2 = [1, 1, 0, 1]
34.     # print(getMin(array2, 0, len(array2) - 1))
35.     print(getMin([1, 2], 0, 1))

```



```

1. # @Filename: 按照要求构造新的数组.py
2. # 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
3. """
4. 分析:
5. 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]...
6. 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]
7. --
8. """
9. def calculate(a, b):
10.     b[0] = 1
11.     multi = 1
12.     for i in range(1, len(a)):
13.         multi *= a[i - 1]
14.         b[i] = multi
15.     multi = 1
16.     for i in range(len(a) - 2, -1, -1):
17.         print(i)
18.         multi *= a[i+1]
19.         b[i] *= multi
20. --
21. if __name__ == "__main__":
22.     a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
23.     b = [None] * len(a)
24.     calculate(a, b)
25.     print(b)

```

```
1. # @Filename: 旋转数组的实现方式.py
2. # 分析: 遍历了两次数组T:O(N), S:O(1)
3. def swap(arr, low, high):
4.     while low < high:
5.         tmp = arr[low]
6.         arr[low] = arr[high]
7.         arr[high] = tmp
8.         low += 1
9.         high -= 1
10. —
11. def rotate(arr, div):
12.     # 采用inspace的修改方法, 不需要返回数组
13.     if not arr or div < 0 or div >= len(arr):
14.         return
15.     if div == 0 or div == len(arr):
16.         return
17.     swap(arr, 0, div)
18.     swap(arr, div + 1, len(arr) - 1)
19.     swap(arr, 0, len(arr) - 1)
20. —
21. if __name__ == "__main__":
22.     arr = [1, 2, 3, 4, 5]
23.     rotate(arr, 2)
24.     print(arr)
```

```
1. # @Filename: 求数组中两个元素的最小距离.py
2. # 采用动态规划, 利用两个变量记录最近访问到的num1和num2
3. def minDistance(arr, num1, num2):
4.     lastPos1 = None
5.     lastPos2 = None
6.     min_dis = len(arr)
7.     for i in range(len(arr)):
8.         if arr[i] == num1:
9.             lastPos1 = i
10.            if lastPos2:
11.                # min_dis = min_dis if abs(lastPos2 - lastPos1) > min_dis else
                abs(lastPos2 - lastPos1)
12.                min_dis = min(min_dis, lastPos1 - lastPos2)
13.            if arr[i] == num2:
14.                lastPos2 = i
15.                if lastPos1:
16.                    # min_dis = min_dis if abs(lastPos2 - lastPos1) > min_dis else
                    abs(lastPos2 - lastPos1)
17.                    min_dis = min(min_dis, lastPos2 - lastPos1)
18.            return min_dis
19.
20. if __name__ == "__main__":
21.     arr = [4, 5, 6, 4, 7, 4, 6, 4, 7, 8, 5, 6, 4, 3, 10, 8]
22.     num1 = 4
23.     num2 = 8
24.     print(minDistance(arr, num1, num2))
```

```
1. # @Filename: 求解最小三元组距离.py
2. """
3. 1. 蛮力法: 三个循环依次记录
4. 2. 最小距离法
5. """
6. def mins(a, b, c):
7.     mins = min(a, b, c)
8.     return mins
9. -
10. def maxes(a, b, c):
11.     return max(a, b, c)
12. -
13. def minDistance(a, b, c):
14.     aLen = len(a)
15.     bLen = len(b)
16.     cLen = len(c)
17.     curDist = 0
18.     minsd = 0
19.     minDist = 2 ** 32
20.     i = 0
21.     j = 0
22.     k = 0
23.     while True:
24.         # 计算得到最短距离
25.         curDist = maxes(a[i], b[j], c[k]) - mins(a[i], b[j], c[k])
26.         if curDist < minDist:
27.             minDist = curDist
28.         # 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
29.         minsd = mins(a[i], b[j], c[k])
30.         if minsd == a[i]:
31.             i += 1
32.             if i >= aLen:
33.                 break
34.             elif minsd == b[j]:
35.                 j += 1
36.                 if j >= bLen:
37.                     break
38.             else:
39.                 k += 1
40.                 if k >= cLen:
41.                     break
```

```
42.     return minDist
43.
44. if __name__ == "__main__":
45.     a = [3, 4, 5, 7, 15]
46.     b = [10, 12, 14, 16, 17]
47.     c = [20, 21, 23, 24, 37, 30]
48.     print(minDistance(a,b,c))
```

```

1. # @Filename: 求解迷宫问题.py
2. """
3. 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
4. 分析: 采用递归法
5. """
6. class Maze:
7.     def __init__(self):
8.         self.N = 4
9.     -
10.    def printSolution(self, sol):
11.        i = 0
12.        while i < self.N:
13.            j = 0
14.            while j < self.N:
15.                print(sol[i][j], end=" ")
16.                j += 1
17.            print()
18.            i += 1
19.    -
20.    def isSafe(self, maze, x, y):
21.        return x >= 0 and x < self.N and y >= 0 and y < self.N and maze[x][y] == 1
22.    -
23.    def getPath(self, maze, x, y, sol):
24.        if x == self.N - 1 and y == self.N - 1:
25.            # 到达中点
26.            sol[x][y] = 1
27.            return True
28.        if self.isSafe(maze, x, y):
29.            sol[x][y] = 1
30.            if self.getPath(maze, x, y + 1, sol):
31.                return True
32.            if self.getPath(maze, x + 1, y, sol):
33.                return True
34.            sol[x][y] = 0
35.            return False
36.        return False
37.    if __name__ == "__main__":
38.        rat = Maze()
39.        maze = [[1, 0, 0, 0], [1, 1, 0, 1], [0, 1, 0, 0], [1, 1, 1, 1]]
40.        sol = [[0, 0, 0, 0] for _ in range(4)]
41.        res = rat.getPath(maze, 0, 0, sol)
42.        if res:

```

43.

`rat.printSolution(sol)`

```
1. # @Filename: 给定一个矩阵, 顺时针向里打印数据.py
2. # matrix类型为二维列表, 需要返回列表
3. def printMatrix(matrix):
4.     # write code here
5.     if not matrix:
6.         return matrix
7.     res = []
8.     startX = 0
9.     while len(matrix) > 2 * startX and len(matrix[0]) > 2 * startX:
10.         res.extend(print_matrix_incircle(matrix, startX))
11.         startX += 1
12.     return res
13. —
14. def print_matrix_incircle(matrix, start):
15.     res = []
16.     endX = len(matrix[0]) - 1 - start
17.     endY = len(matrix) - 1 - start
18.     —
19.     # 从左往右打印, 没有约束条件
20.     for i in range(start, endX + 1):
21.         print(matrix[start][i])
22.         res.append(matrix[start][i])
23.     —
24.     # 从上往下打印, 至少有两行
25.     if endY > start:
26.         for i in range(start + 1, endY + 1):
27.             print(matrix[i][endX])
28.             res.append(matrix[i][endX])
29.     —
30.     # 从右往左打印, 至少有两行两列
31.     if start < endX and endY > start:
32.         for i in range(endX - 1, start - 1, -1):
33.             print(matrix[endY][i])
34.             res.append(matrix[endY][i])
35.     # 从下往上打印, 至少有三行两列
36.     if start < endY - 1 and start < endX:
37.         for i in range(endY - 1, start, -1):
38.             print(matrix[i][start])
39.             res.append(matrix[i][start])
40.     return res
41. —
42. print(printMatrix([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]))
```


7. 栈

```
1. # @Filename: 从数组中找出满足a+b=c+d的数.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 使用字典存储 key:两个数的值 value为数据对
5. —
6. —
7. def findPairs(arr):
8.     dic = {}
9.     for i in range(len(arr)):
10.         for j in range(i, len(arr)):
11.             if arr[i] + arr[j] in dic:
12.                 print(dic[arr[i] + arr[j]][0], " + ", dic[arr[i] + arr[j]][1], " = ",
arr[i], " + ", arr[j])
13.             else:
14.                 dic[arr[i] + arr[j]] = [arr[i], arr[j]]
15. —
16. —
17. if __name__ == "__main__":
18.     arr = [3, 4, 7, 10, 20, 9, 8]
```

```

1. # @Filename: 实现LRU缓存.py
2. # LRU : least recently used,最近最少使用
3. # 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
4. # 2. 使用哈希表用于缓存已经存在的页面
5. # 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)
6. —
7. # 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()
8. from collections import deque
9. class LRU:
10.     def __init__(self, cacheSize):
11.         self.cacheSize = cacheSize
12.         self.queue = deque()
13.         self.hashSet = set()
14. —
15.     def isQueueFull(self):
16.         return len(self.queue) == self.cacheSize
17. —
18.     # 如果队列满了, 删除末尾数据
19.     def enqueue(self, pageNum):
20.         if self.isQueueFull():
21.             self.hashSet.remove(self.queue[-1])
22.             self.queue.pop()
23.             self.queue.appendleft(pageNum)
24.             self.hashSet.add(pageNum)
25. —
26.     def accessPage(self, pageNum):
27.         if pageNum not in self.hashSet:
28.             self.enqueue(pageNum)
29.         elif pageNum != self.queue[0]:
30.             self.queue.remove(pageNum)
31.             self.queue.appendleft(pageNum)
32. —
33.     def printQueue(self):
34.         while len(self.queue) > 0:
35.             print(self.queue.popleft(), sep=" ")
36. if __name__ == "__main__":
37.     lru = LRU(3)
38.     lru.accessPage(1)
39.     lru.accessPage(2)
40.     lru.accessPage(4)
41.     lru.accessPage(5)
42.     lru.accessPage(6)

```

43. `lru.accessPage(7)`

44. `lru.printQueue()`

```
1. # @Filename: 栈排序.py
2. class Stack:
3.     # 模拟栈
4.     def __init__(self):
5.         self.items = []
6.     —
7.     def empty(self):
8.         return True if not self.items else False
9.     —
10.    def size(self):
11.        return len(self.items)
12.    —
13.    # 返回栈顶元素
14.    def peek(self):
15.        if not self.empty():
16.            return self.items[-1]
17.        else:
18.            return None
19.    # 弹栈
20.    def pop(self):
21.        if not self.empty():
22.            return self.items.pop()
23.        else:
24.            print("栈已空")
25.            return None
26.    —
27.    # 压栈
28.    def push(self, item):
29.        self.items.append(item)
30.    —
31.    def moveBottomToTop(s):
32.        # 将栈底元素移动到栈顶, 其他栈元素顺序不变
33.        if s.empty():
34.            return
35.        top1 = s.pop()
36.        if not s.empty():
37.            moveBottomToTop(s)
38.            top2 = s.peek()
39.            if top2 < top1:
40.                s.pop()
41.                s.push(top1)
42.                s.push(top2)
43.            return
```

```
44.     s.push(top1)
45.     —
46.     # 翻转栈
47.     def sort_stack(s):
48.         if s.empty():
49.             return
50.         moveBottomToTop(s)
51.         # 记录逆序的顺序, 后面递归要还原回去
52.         top = s.peek()
53.         print(top)
54.         s.pop()
55.         # 将子栈的栈底移动到子栈的栈顶
56.         sort_stack(s)
57.         s.push(top)
58.     if __name__ == "__main__":
59.         s = Stack()
60.         for i in range(1, 6):
61.             s.push(i)
62.             print("before reverse")
63.             print(s.items)
64.             sort_stack(s)
65.             print("after reverse")
```

```
1. # @Filename: 根据入栈顺序判断可能的出栈顺序.py
2. # 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
3. class Stack:
4.     def __init__(self):
5.         self.items = []
6.     —
7.     # 判断栈是否为空
8.     def empty(self):
9.         return True if not self.items else False
10.    —
11.    # 返回栈的大小
12.    def size(self):
13.        return len(self.items)
14.    —
15.    # 返回栈顶元素
16.    def peek(self):
17.        return self.items[-1]
18.    —
19.    # 出栈
20.    def pop(self):
21.        if self.empty():
22.            print("栈已空")
23.            return None
24.        else:
25.            return self.items.pop()
26.    —
27.    # 入栈
28.    def push(self, item):
29.        self.items.append(item)
30.    —
31.    def isPopSerial(push, pop):
32.        if not push or not pop:
33.            return False
34.        if len(push) != len(pop):
35.            return False
36.        pushIndex = 0
37.        popIndex = 0
38.        stack = Stack()
39.        while pushIndex < len(push):
40.            stack.push(push[pushIndex])
41.            pushIndex += 1
42.            while not stack.empty() and stack.peek() == pop[popIndex]:
43.                stack.pop()
```

```
44.     popIndex += 1
45.     # 这里只需判断入栈是否为空就行了
46.     return stack.empty()
47. —
48. if __name__ == "__main__":
49.     push = "123324"
50.     pop = "5213254"
51.     if isPopSerial(push, pop):
52.         print(pop + "是出栈顺序")
53.     else:
54.         print(pop + "不是出栈顺序")
```

```
1. # @Filename: 用两个栈模拟队列操作.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 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
5. class Stack:
6.     def __init__(self):
7.         self.items = []
8.     —
9.     def empty(self):
10.         return True if not self.items else False
11.     —
12.     def size(self):
13.         return len(self.items)
14.     —
15.     def peek(self):
16.         return self.items[-1]
17.     def pop(self):
18.         return self.items.pop()
19.     def push(self, item):
20.         self.items.append(item)
21.     —
22. class MyQueue:
23.     def __init__(self):
24.         self.A = Stack()
25.         self.B = Stack()
26.     —
27.     def push(self, item):
28.         self.A.push(item)
29.     —
30.     def pop(self):
31.         if not self.B.empty():
32.             return self.B.pop()
33.         else:
34.             for i in range(self.A.size()):
35.                 self.B.push(self.A.pop())
36.         —
37.         return self.B.pop()
38.     —
39. if __name__ == "__main__":
40.     queue = MyQueue()
41.     queue.push(1)
42.     queue.push(2)
43.     print("输出队列元素", queue.pop())
```


44. `print("输出队列元素", queue.pop())`

```
1. # @Filename: 翻转栈的所有元素.py
2. # 由于栈有两种实现方式, 数组和链表
3. # 栈的翻转对于链表来说, 相当于链表的逆序
4. # 本程序考虑数组的实现模式
5. """
6. 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
7. """
8. class Stack:
9.     # 模拟栈
10.     def __init__(self):
11.         self.items = []
12.     —
13.     def empty(self):
14.         return True if not self.items else False
15.     —
16.     def size(self):
17.         return len(self.items)
18.     —
19.     # 返回栈顶元素
20.     def peek(self):
21.         if not self.empty():
22.             return self.items[-1]
23.         else:
24.             return None
25.     # 弹栈
26.     def pop(self):
27.         if not self.empty():
28.             return self.items.pop()
29.         else:
30.             print("栈已空")
31.             return None
32.     —
33.     # 压栈
34.     def push(self, item):
35.         self.items.append(item)
36.     —
37.     def moveBottomToTop(s):
38.         # 将栈底元素移动到栈顶, 其他栈元素顺序不变
39.         if s.empty():
40.             return
41.         top1 = s.pop()
42.         if not s.empty():
43.             moveBottomToTop(s)
```

```
44.         top2 = s.peak()
45.     s.pop()
46.     s.push(top1)
47.     s.push(top2)
48. else:
49.     s.push(top1)
50. —
51. # 翻转栈
52. def reverse_stack(s):
53.     if s.empty():
54.         return
55.     moveBottomToTop(s)
56.     # 记录逆序的顺序, 后面递归要还原回去
57.     top = s.peak()
58.     print(top)
59.     s.pop()
60.     # 将子栈的栈底移动到子栈的栈顶
61.     reverse_stack(s)
62.     s.push(top)
63. —
64. if __name__ == "__main__":
65.     s = Stack()
66.     for i in range(1, 6):
67.         s.push(i)
68.     print("before reverse")
69.     print(s.items)
70.     reverse_stack(s)
71.     print("after reverse")
72.     print(s.items)
```

```
1. # @Filename: 通过给定的车票找出旅程.py
2. # 不考虑环的情况
3. # 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
4. def printResult(inputs):
5.     reverse_inputs = dict(zip(inputs.values(), inputs.keys()))
6.     reverse_values = reverse_inputs
7.     start = None
8.     for k, v in inputs.items():
9.         if k not in reverse_values:
10.             start = k
11.             break
12.     print(start)
13.     # 通过开始点, 构建旅途
14.     while start in inputs:
15.         print(start+"->"+inputs[start])
16.         start = inputs[start]
17. if __name__ == "__main__":
18.     inputs = dict()
19.     inputs['西安'] = '成都'
20.     inputs['北京'] = '上海'
21.     inputs['大连'] = '西安'
22.     inputs['上海'] = '大连'
23.     printResult(inputs)
```

8. 链表

```
1. # @Filename: 两个链表数之和 (链表相加法) .py
2. # 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
3. class LNode:
4.     def __init__(self):
5.         self.data = None
6.         self.next = None
7.
8. def add(head1, head2):
9.     if not head1 or not head1.next:
10.         return head2
11.     if not head2 or not head2.next:
12.         return head1
13.     c = 0 # 表示进位
14.     result_head = LNode()
15.     cur1 = head1.next
16.     cur2 = head2.next
17.     cur3 = result_head
18.     # 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
19.     while cur1 and cur2:
20.         tmp = LNode()
21.         result = cur1.data + cur2.data + c
22.         if result > 9:
23.             c = 1
24.             tmp.data = result - 10
25.         else:
26.             tmp.data = result
27.             c = 0
28.         cur3.next = tmp
29.         cur3 = tmp
30.         cur1 = cur1.next
31.         cur2 = cur2.next
32.     # 如果cur1数据较多, 将cur1的数据继续添加在result_head中
33.     if cur1:
34.         while cur1:
35.             tmp = LNode()
36.             sum = cur1.data + c
37.             tmp.data = sum % 10
38.             c = sum // 10
39.             cur3.next = tmp
40.             cur3 = tmp
```

```
41.         cur1 = cur1.next
42.     if cur2:
43.         while cur2:
44.             tmp = LNode()
45.             sum = cur2.data + c
46.             tmp.data = sum % 10
47.             c = sum // 10
48.             cur3.next = tmp
49.             cur3 = tmp
50.             cur2 = cur2.next
51.     -
52.     # 如果任然有进位的话
53.     if c:
54.         tmp = LNode()
55.         tmp.data = 1
56.         cur3.next = tmp
57.     return result_head
```

```
1. # @Filename: 判断链表是否有环.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 采用快慢指针, 如果有环, 那么总有一次, 两个指针相等
5. # 如果有环, 则怎样确定环的入口位置:
6. import random
7. """
8. 1. 先找到相遇点
9. 2. 分别从头结点和相遇点一次遍历, 相遇的位置即入口点
10. """
11. class LNode:
12.     def __init__(self):
13.         self.data = None
14.         self.next = None
15. —
16. def has_circle(head):
17.     slow = head
18.     fast = head
19.     while fast and fast.next:
20.         slow = slow.next
21.         fast = fast.next.next
22.         if slow == fast:
23.             return slow
24.     return False
25. —
26. if __name__ == "__main__":
27.     # 创建一个循环链表
28.     circle_entrance = random.randint(5, 10)
29.     print(circle_entrance)
30.     entrance_node = None
31.     i = 1
32.     head = LNode()
33.     cur = head
34.     while i < 20:
35.         tmp = LNode()
36.         tmp.data = i
37.         cur.next = tmp
38.         cur = tmp
39.         if i == circle_entrance:
40.             entrance_node = cur
41.         i += 1
42. —
```

```
43.     # 连接形成环
44.     cur.next = entrance_node
45.     —
46.     # 判断是否有环
47.     if has_circle(head.next):
48.         meet_node = has_circle(head.next)
49.         cur = head.next
50.         while cur != meet_node:
51.             cur = cur.next
52.             meet_node = meet_node.next
53.         print(meet_node.data)
54.     else:
55.         print("没有环")
56.     # 如果有环, 找出入口点
```



```
1. # @Filename: 合并两个有序链表.py
2. # 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
3. -
4. class LNode:
5.     def __init__(self):
6.         self.data = None
7.         self.next = None
8.     def merge_two_list(h1, h2):
9.         if not h1 and not h1.next:
10.            return h2
11.         if not h2 and not h2.next:
12.            return h1
13.         cur1 = h1.next
14.         cur2 = h2.next
15.         next = h2.next.next
16.         pre = h1
17.         while cur1 and cur2:
18.             if cur1.data >= cur2.data:
19.                 cur2.next = pre.next
20.                 pre.next = cur2
21.                 pre = cur2
22.                 cur2 = next
23.                 if cur2:
24.                     next = next.next
25.             else:
26.                 pre = cur1
27.                 cur1 = cur1.next
28.         # 判断哪一个链表先完
29.         if not cur1:
30.             pre.next = cur2
31.             return
32.         else:
33.             return
```

```
1. # @Filename: 向右旋转k个位置.py
2. class LNode:
3.     def __init__(self):
4.         self.data = None
5.         self.next = None
6. —
7. def ReverseK(head, k):
8.     cur = head
9.     next_k = head
10.    i = 0
11.    while i < k and next_k:
12.        next_k = next_k.next
13.        i += 1
14.    if i < k:
15.        # 链表还没有k长
16.        return None
17.    while next_k.next:
18.        cur = cur.next
19.        next_k = next_k.next
20.    second_head = cur.next
21.    cur.next = None
22.    next_k.next = head.next
23.    head.next = second_head
24.    return head
```

```
1. # @Filename: 如何展开链接列表.py
2. class Node:
3.     def __init__(self):
4.         self.data = None
5.         self.right = None
6.         self.down = None
7. —
8. class MergeList:
9.     def __init__(self):
10.         self.head = None
11. —
12.     def insert(self, head_ref, data):
13.         tmp = Node()
14.         tmp.data = data
15.         tmp.down = head_ref
16.         head_ref = tmp
17.         return head_ref
18. —
19.     def merge(self, a, b):
20.         if not a:
21.             return b
22.         if not b:
23.             return a
24.         if a.data < b.data:
25.             result = a
26.             result.down = self.merge(a.down, b)
27.         else:
28.             result = b
29.             result.down = self.merge(a, b.down)
30.         return result
31. —
32.     def flatten(self, root):
33.         if not root or not root.right:
34.             return root
35.         # 递归处理root.right链表
36.         root.right = self.flatten(root.right)
37. —
38.         root = self.merge(root, root.right)
```

```
1. # @Filename: 无序链表移除重复项 (hashset) 3.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. —
5. # 空间换时间, 遍历时将数据载入list, 每次遍历查看是否出现
6. —
7. class LNode:
8.     def __init__(self):
9.         self.data = None
10.        self.next = None
11. —
12. —
13. def remove_duplication(head):
14.     if not head or not head.next or not head.next.next:
15.         return
16.     node = []
17.     cur = head.next
18.     pre = head
19.     while cur:
20.         if cur.data not in node:
21.             node.append(cur.data)
22.             pre = cur
23.             cur = cur.next
24.         else:
25.             pre.next = cur.next
26.             cur = pre.next
```

```
1. # @Filename: 无序链表移除重复项 (双重循环) 3.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. --
5. # 对于无顺序的链表, 删除其中重复的项目
6. """
7. 分析: 利用双重循环, 时间复杂度为 $O(N*N)$ , 空间复杂度为 $O(1)$ , 常数个指针变量保存前继节点等
8. """
9. --
10. class LNode:
11.     def __init__(self):
12.         self.data = None
13.         self.next = None
14. --
15. def remove_duplication(head):
16.     if not head or not head.next:
17.         return
18.     Outer_cur = head.next
19.     while Outer_cur:
20.         Inter_pre = Outer_cur
21.         Inter_cur = Outer_cur.next
22.         while Inter_cur:
23.             if Inter_cur.data == Outer_cur.data:
24.                 # 删除当前节点
25.                 Inter_pre.next = Inter_cur.next
26.                 Inter_cur = Inter_cur.next
27.             else:
28.                 Inter_pre = Inter_cur
29.                 Inter_cur = Inter_cur.next
30.         print(Outer_cur.data, Outer_cur.next)
31.         Outer_cur = Outer_cur.next
```

```
1. # @Filename: 无序链表移除重复项 (递归) 2.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. —
5. """
6. 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
7. """
8. class LNode:
9.     def __init__(self):
10.         self.data = None
11.         self.next = None
12. —
13. def removeDupRecursive(head):
14.     if not head.next:
15.         return head
16.     cur = head
17.     head.next = removeDupRecursive(head.next)
18.     pointer = head.next
19. —
20.     while pointer:
21.         if head.data == pointer.data:
22.             cur.next = pointer.next
23.             pointer = cur.next
24.         else:
25.             pointer = pointer.next
26.             cur = cur.next
27.     return head
28. —
29. def remove_duplication(head):
30.     if not head:
31.         return
32.     head.next = removeDupRecursive(head.next)
```

```
1. # @Filename: 查找链表倒数第k个元素.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 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
5. —
6. class LNode:
7.     def __init__(self):
8.         self.data = None
9.         self.next = None
10. —
11. def FindLastK(head, k):
12.     cur = head
13.     next_k = head
14.     i = 0
15.     while i < k and next_k:
16.         next_k = next_k.next
17.         i += 1
18.     if i < k:
19.         # 链表还没有k长
20.         return None
21.     while next_k:
22.         cur = cur.next
23.         next_k = next_k.next
24.     return cur.data
```

```
1. # @Filename: 链表以K个节点为一组翻转.py
2. # 这道题不复杂, 就是比较繁琐
3. class LNode:
4.     def __init__(self):
5.         self.data = None
6.         self.next = None
7. —
8.     def reverse(begin, end):
9.         pre = None
10.        cur = begin
11.        next = cur.next
12.        while cur != end:
13.            cur.next = pre
14.            pre = cur
15.            cur = next
16.            next = next.next
17.        cur.next = pre
18. —
19.     def reverse_per_k(head, k):
20.         if not head and not head.next:
21.             return
22.         j = k
23.         pre = head
24.         begin = head.next
25.         end = begin
26.         while end:
27.             # 找到end指针
28.             while j > 1 and end:
29.                 end = end.next
30.                 j -= 1
31.             if end:
32.                 lnext = end.next
33.                 reverse(begin, end)
34.                 pre.next = end
35.                 pre = begin
36.                 pre.next = lnext
37.                 begin = lnext
38.                 end = begin
39.                 j = k
40.         else:
41.             return
```



```
1. # @Filename: 链表相邻元素翻转.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 1. 通过交换相邻链表的值, 但对某些场景不适用
5. # 2. 遍历就地逆序法, 需要添加一些指针
6. —
7. —
8. class LNode:
9.     def __init__(self):
10.         self.data = None
11.         self.next = None
12. —
13. —
14. def Reverse_eachother(head):
15.     if not head and not head.next:
16.         return
17.     pre = head
18.     cur = pre.next
19.     while cur and cur.next:
20.         next = cur.next.next
21.         pre.next = cur.next
22.         cur.next.next = cur
23.         cur.next = next
24.         pre = cur
25.         cur = next
```

```
1. # @Filename: 链表逆序输出 (不改变原结构: 递归) .py
2. # 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,
3. # 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
4. —
5. class LNode:
6.     def __init__(self):
7.         self.data = None
8.         self.next = None
9. —
10. —
11. def RecursiveReverse(head):
12.     if not head or not head.next:
13.         print(head.data)
14.     else:
15.         # 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
16.         RecursiveReverse(head.next)
17.         print(head.data)
18. —
19. def Reverse_print(head):
20.     if not head:
21.         return
22.     firstNode = head.next
23.     newhead = RecursiveReverse(firstNode)
```

```
1. # @Filename: 链表逆序 (递归) 2.py
2. # 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,
3. # 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
4. —
5. class LNode:
6.     def __init__(self):
7.         self.data = None
8.         self.next = None
9. —
10. def RecursiveReverse(head):
11.     if not head or not head.next:
12.         return head
13.     else:
14.         # 这里的新_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
15.         new_head = RecursiveReverse(head.next)
16.         head.next.next = head
17.         head.next = None
18.         return new_head
19. —
20. def Reverse(head):
21.     if not head:
22.         return
23.     firstNode = head.next
24.     newhead = RecursiveReverse(firstNode)
25.     head.next = newhead
26.     return head
```

```
1. # @Filename: 链表逆序 (遍历+指针节点) 1.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 创建链表结点类, 使用引用方式实现链表的指针功能
5.
6. """
7. 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 $O(1)$ 
8. """
9. class LNode:
10.     def __init__(self):
11.         self.data = None
12.         self.next = None
13.
14. def Reverse(head):
15.     if not head or not head.next:
16.         return
17.     pre = None
18.     cur = head.next
19.     while cur.next:
20.         next = cur.next
21.         cur.next = pre
22.         pre = cur
23.         cur = next
24.     cur.next = pre
25.     head.next = cur
26.     return head
```

```
1. # @Filename: 链表逆序 (遍历+插入) 3.py
2. # @Software: PyCharm
3. # @e_mail: sancica@163.com
4. # 采用插入法将, 遍历数据时, 将数据插入到头结点之后
5. —
6. —
7. class LNode:
8.     def __init__(self):
9.         self.data = None
10.        self.next = None
11. —
12. —
13. def Reverse(head):
14.     if not head or not head.next:
15.         return
16.     cur = head.next.next
17.     # 斩断第一个数据指针
18.     head.next.next = None
19.     while cur:
20.         next = cur.next
21.         cur.next = head.next
22.         head.next = cur
23.         cur = next
```

```
1. # @Filename: 链表重新排序.py
2. # 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
3. # 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  $O(1)$ 
4. class LNode:
5.     def __init__(self):
6.         self.data = None
7.         self.next = None
8. —
9. def FindMiddleNode(head):
10.     # 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
11.     fast = head
12.     slow = head
13.     while fast and fast.next:
14.         fast = fast.next.next
15.         slow = slow.next
16. —
17.     # 将链表进行拆分
18.     second_head = slow.next
19.     slow.next = None
20.     return second_head
21. —
22. def Reverse(head):
23.     pre = None
24.     cur = head
25.     next = head.next
26.     while next:
27.         cur.next = pre
28.         pre = cur
29.         cur = next
30.         next = next.next
31.     cur.next = pre
32.     return cur
33. —
34. def Reorder(head):
35.     if not head or not head.next or not head.next.next or not head.next.next.next:
36.         return
37.     mid = FindMiddleNode(head.next)
38.     # 对后半部分链表进行逆序
39.     cur2 = Reverse(mid)
40.     cur1 = head.next
41.     # 合并两个链表
```

```
42.     while cur2 and cur1:
43.         next2 = cur2.next
44.         cur2.next = cur1.next
45.         cur1.next = cur2
46.         cur1 = cur2.next
47.         cur2 = next2
```