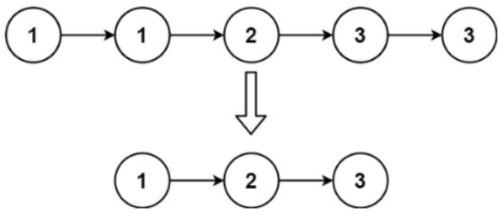
问题 1

> 给定一个已排序的链表的头 `head` ， *删除所有重复的元素，使每个元素只出现一次* 。返回 *已排序的链表* 。链表的类如下所示:

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

输入是一个数组，你首先需要将数组转化为链表，然后删除链表中的重复元素，再遍历链表元素，以一个数组的形式返回。请设计一个算法解决上述任务，分析算法设计思路，计算时间复杂度, 并基于python编程实现。

e.g. 输入：head=[1, 1, 2, 3, 3] 输出：[1, 2, 3]



采用双指针法，一个指向当前节点，一个指向下一个节点，若两节点的值相同，则删除下一个节点。

算法思路:

```
def remove_duplication(head):
    current = head
    while current and current.right://当前节点与其下一节点均存在
        if current.val == current. right. val//值相同
            current.right =current.right.right//删除重复元素
        else
            current = current.right//右移
    return head
```

时间复杂度：由于只遍历了一次链表，因此时间复杂度为 O(n)

问题 2

> 下面是一个经典的算法问题:
>
> - 给定包含n个整数的一个整数数组 `nums` 和一个整数目标值 `target` ，请你在该数组中找出 **和为目标值** *`target`* 的那 **两个** 整数，并返回它们的**数组下标**。假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。
>
> 由于要多次查找数组中元素的位置，为了提高查询效率可以使用哈希表来存储数组中的数据，在哈希表中查询一个元素的复杂度为O(1)。 已知python中的字典是使用哈希表实现的，即使用 `dict[key]` 查询对应的value时间复杂度为O(1), python提供了查询字典是否包含某个key的功能: `key in dict` ，时间复杂度也是O(1)

请根据上面信息，设计一个时间复杂度为O(n) 的算法，解决上述算法问题

通过目标值与数组中每一元素的差值，在哈希表中进行查找。

问题 3:

> 栈是一种常用的数据结构，编译器中通常使用栈来实现表达式求值。
>
> 以表达式 $3+5\times8-6$ 为例。编译器使用两个栈来完成运算，即一个栈保持操作数，另一个栈保存运算符。
>
> 1. 从左向右遍历表达式，遇到数字就压入操作数栈；
> 2. 遇到运算符，就与运算符栈的栈顶元素进行比较。如果比运算符栈顶元素的优先级高，就将当前运算符压入栈；如果比运算符栈顶元素的优先级低或者相同，从运算符栈中取栈顶运算符，从操作数栈的栈顶取 2 个操作数，然后进行计算，再把计算完的结果压入操作数栈，继续比较。
>
> 下图是 $3+5\times8-6$ 这个表达式的计算过程:

遇到数字时，直接压入栈中，遇到操作符时，若操作符栈为空或当前操作符优先级高于栈顶元素，则入栈，否则，出栈，与数字栈栈顶的两个元素进行计算后再入栈，直至操作符优先级高于栈顶元素，最后再将操作符入栈。

算法伪代码如下:

```
def evaluate(expression):
    values = [ ] #数字栈
    ops = [ ] #操作符栈
    i=0
    while i< len(expression):未至表达式最后一个元素
        if expression[i].isdigit():如果是数字，入栈
            val = 0
            while i<len(expression) and expression[i].isdigit():
                val = (val*10)+ expression[i]恢复为数字
                i+=1
            values.append(val)
            i-=1
        else:是操作符
            while(ops and precedence(ops[-1]>=precedence expression[i])):
                val2=values.pop( )
                val1=values.pop( )
                op=ops.pop()
                values.append(apply.op(val1,val2,op))计算并入栈
            ops.append(expression[i])
        i+=1
    while ops:清空栈
        val2=values.pop()
        val1=values.pop()
        op=ops.pop()
        values.append(apply_op(val1,val2,op))
    return values[-1]
```

算法时间复杂度：由于其中每一个元素只会入栈，出栈一次，所以时间复杂度为O(n)。

问题 4:

> 星球碰撞问题：现有n个星球，在同一条直线上运行，如数组A所示，元素的绝对值表示星球的质量，负数表示星球自右向左运动，正数表示星球自左向右运动，当两个星球相撞的时候，质量小的会消失，大的保持不变，**质量相同的两个星球碰撞后自右向左运动的星球消失，自左向右的星球保持不变**，假设所有星球的速度大小相同。
>
> $A = [23, -8, 9, -3, -7, 9, -23, 22]$

请设计一个算法模拟星球的运行情况，输出最终的星球存续情况（输出一个数组），分析算法设计思路，计算时间复杂度，并基于python编程实现。

使用栈来存储存活的星球，星球向右移动则入栈，向左移动则处理碰撞至无碰撞发生

```
def collision(planets):
    stuck = [ ]
    for planet in planets:
        if stack[-1]<-planet:质量小不出栈
            stack.pop()
            continue
        else stack[-1] == -planet:质量相同出栈
            stack.pop()
        break
```

else:未发生碰撞，入栈
stack.append(planet)
return stack
时间复杂度：每个行星只会入栈出栈一次，所以时间复杂度为 O(n)

问题 5

> 给定一个无序数组nums=[9,-3,-10,0,9,7,33]，请建立一个二叉搜索树存储数组中的所有元素，之后删除二叉树中的元素"0"，再使用中序遍历输出二叉搜索树中的所有元素。

使用python编程完成上述任务，并计算时间复杂度

插入操作时间复杂度为 O(logn),删除操作时间复杂度为 O(logn),中序遍历时间复杂度为 O(n),由于需要对 n 个元素均进行插入操作,故时间复杂度为 O(nlogn)

问题 6

> 给定一个包含大写字母和小写字母的字符串 s ，返回 字符串包含的 **最长的回文子串的长度** 。请注意 区分大小写 。比如 "Aa" 不能当做一个回文字符串。

请设计一个算法解决上述问题，只需要输出最长回文子串的长度，分析算法设计思路，计算时间复杂度，并基于python编程实现

e.g. 输入： s="adccaccd"， 输出：7。 最长回文子串为："dccaccd", 长度为7

依次遍历每个元素，从该元素向两端寻找
def longest_palindrome(s):
start = 0
end = 0
for i in range(len(s)):
len1=expand_around_center(s,i,i)向左移动寻找回文子串
len2= expand_around_center(s,i,i+1)向右移动寻找回文子串
max_len=max(len1,len2)
if max_len>end_start:
start = i-(max-len-1)
end = i+max_len
return end-start+1
时间复杂度为：$O(n^2)$

问题 7

> 沿一条长河流分散着n座房子。你可以把这条河想象成一条轴，房子是由它们在这条轴上的坐标按顺序排列的。你的公司想在河边的特定地点设置手机基站，这样每户人家都在距离基站4公里的范围内。输入可以看作为一个升序数组，数组元素的取值为大于等于0的正整数，你需要输出最小基站的数目，基站的位置。

1. 给出一个时间复杂度为$O(n)$的算法，使所使用的基站数量最小化，分析算法设计思路，使用python编程实现
2. 证明1.中算法产生了最优解决方案。

e.g.

输入： [1, 5, 12, 33, 34,35] 输出：基站数目为3， 基站位置为[1，12，33]

1.使用贪心算法遍历房子位置，将基站放在覆盖尽可能多的房子的最右端，重复该过程直至覆盖所有房子。
def place_base_stations(houses):
if not houses:
return 0, []
n = len(houses)
base_stations = []
i = 0

```
    while i < n:
        loc = houses[i] + 4
        base_stations.append(loc)#选择当前能覆盖的最远的房子位置作为基站的位置
        while i < n and houses[i] <= loc:
            i += 1                          # 跳过所有在当前基站覆盖范围内的房子
    return len(base_stations), base_stations
```

2.假设使用贪心算法放置了 k 个基站，若存在一个更优解可以使用少于 k 个基站覆盖，则在贪心算法选择过程中，可以选择覆盖更多房子的基站位置，与贪心算法的选择原则相违背，因此，贪心选择一定是局部最优。

问题 8

> 给定由n个正整数组成的一个集合$S=\{a_1,a_2,\cdots,a_n\}$和一个正整数W，设计一个算法确定是否存在S的一个子集$K\subseteq S$, 使K中所有数之和为W, 如果存在返回"True"，否则返回"False"

请设计一个时间复杂度为$O(nW)$动态规划算法，解决上述问题，分析算法的设计思路，并且基于python编程实现（不需要输出子集）。

e.g.

输入：S = {1,4,7,3,5}， W = 11，输出：True。 因为K可以是{4,7}。

dp[i][j]表示前 i 个元素中是否存在一个子集，其和为 j

状态转移方程:

$$\mathrm{dp}[i][j]=\begin{cases}\mathrm{dp}[i-1][j], j<S[i-1]\\ \mathrm{dp}[i-1][j-S[i-1]], j\geq S[i-1]\end{cases}$$

```
def subset_sum(S, W):
    n = len(S)
    dp = [[False] * (W + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        dp[i][0] = True
    for i in range(1, n + 1):
        for j in range(1, W + 1):
            if j < S[i-1]:#不选择第 i 个元素
                dp[i][j] = dp[i-1][j]
            else: #选择第 i 个元素
                dp[i][j] = dp[i-1][j] or dp[i-1][j-S[i-1]]
    return dp[n][W]
```

问题 9

> 给定一个n个物品的集合。物体的重量为$w_1,w_2,\cdots,w_n$，物品的价值分别是$v_1、v_2、\cdots v_n$。给你**两个**重量为 c 的背包。如果你带了一个东西，它可以放在一个背包里，也可以放在另一个背包里，但不能同时放在两个背包里。所有权重和价值都是正整数。

1. 设计一个时间复杂度为 $O(nc^2)$ 的动态规划算法，确定可以放入两个背包的物体的最大价值。分析算法设计思路，并基于python编程实现
2. * 修改1中的算法，输出每个背包的内容（物品对应下标）。

e.g.:

输入 V=[1,3,2,5,8,7], W=[1,3,2,5,8,7], c=7, 输出：最大价值=14，背包装的物品为：[6] [4，3] （同一个背包中物品装入顺序对结果无影响）

1. 使用三维动态规划数组 d[i][j][k]表示前 i 个物品放入两个背包中，第一个背包当前容量为 j,第二个背包当前容量为 k 的最大价值。

状态转移方程:

$$dp[i][j][k]=\begin{cases} dp[i-1][j][k], \text{不选择第}i\text{个物品} \\ \max\left(dp[i][j][k], dp[i-1][j-W[i]][k]+V[i]\right), j \geq W[i] \\ \max\left(dp[i][j][k], dp[i-1][j][k-W[i]]+V[i]\right), k \geq W[i] \end{cases}$$

```
def knapsack_two_bags(n, W, weights, values):
    dp = [[[0]*(W+1) for _ in range(W+1)] for _ in range(n+1)]
    trace = [[[None]*(W+1) for _ in range(W+1)] for _ in range(n+1)]
    for i in range(1, n+1):
        for j in range(W+1):
            for k in range(W+1):
                dp[i][j][k] = dp[i-1][j][k]
                trace[i][j][k] = (j, k, 0)
                if j >= weights[i-1]:
                    val = dp[i-1][j - weights[i-1]][k] + values[i-1]
                    if val > dp[i][j][k]:
                        dp[i][j][k] = val
                        trace[i][j][k] = (j - weights[i-1], k, 1)
                if k >= weights[i-1]:
                    val = dp[i-1][j][k - weights[i-1]] + values[i-1]
                    if val > dp[i][j][k]:
                        dp[i][j][k] = val
                        trace[i][j][k] = (j, k - weights[i-1], 2)
    j, k = W, W
    b1, b2 = [], []
    for i in range(n, 0, -1):
        prev_j, prev_k, decision = trace[i][j][k]
        if decision == 1:
            b1.append(i-1)
        elif decision == 2:
            b2.append(i-1)
        j, k = prev_j, prev_k
    return dp[n][W][W], b1[::-1], b2[::-1]
```

问题 10

> 给定两个字符串 $x[1..n]$ 和 $y[1..m]$，我们想通过以下操作将 x 变换为 y：
>
> **插入**：在 x 中插入一个字符(在任何位置)；**删除**：从 x 中删除一个字符(在任何位置)； **替换**：用另一个字符替换 x 中的一个字符。
>
> 例如: $x = abcd, y = bcfe$,
>
> - 将 x 转换为 y 的一种可能方法是：1. 删除 x 开头的 a, x变成 bcd； 2. 将 x 中的字符 d 替换为字符 f。x 变成 bcf； 3. 在 x 的末尾插入字符 e。x 变成 $bcfe$。
> - 另一种可能的方法：1. 删除 x 开头的 a, x 变成 bcd； 2. 在 x 中字符 d 之前插入字符 f。x 变成 $bcfd$。3. 将 x 中的字符 d 替换为字符 e。x 变成 $bcfe$。

设计一个时间复杂度为 $O(mn)$ 的算法，返回将 x 转换为 y 所需的最少操作次数。分析算法设计思路，并基于python编程实现。

dp[i][j]表示将 x 的前 i 个字符转换为 y 的前 j 个字符所需的最少操作次数
状态转移方程:

$$dp[i][j]=\begin{cases} dp[i-1][j-1], if\ x[i-1]==y[j-1] 最后一个字符不操作 \\ \min\left(dp[i][j-1]+1, dp[i-1][j]+1, dp[i-1][j-1]+1\right) \end{cases}$$

```
def min_edit_distance(x, y):
    n = len(x)
    m = len(y)
    # 初始化 dp 数组
    dp = [[0] * (m + 1) for _ in range(n + 1)]
    # 初始化边界条件
    for i in range(n + 1):
        dp[i][0] = i
    for j in range(m + 1):
        dp[0][j] =
    # 填充 dp 数组
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if x[i-1] == y[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(dp[i][j-1] + 1,      # 插入
                               dp[i-1][j] + 1,      # 删除
                               dp[i-1][j-1] + 1)    # 替换
    return dp[n][m]
```

注：本作业在完成时使用了 chatgpt