




# 题目一：最小字典序

## 2734. 执行子串操作后的字典序最小字符串

已解答 

中等

 相关标签

 相关企业

 提示

Ax

给你一个仅由小写英文字母组成的字符串 `s`。在一步操作中，你可以完成以下行为：

- 选择 `s` 的任一非空子字符串，可能是整个字符串，接着将字符串中的每一个字符替换为英文字母表中的前一个字符。例如，'b' 用 'a' 替换，'a' 用 'z' 替换。

返回执行上述操作 **恰好一次** 后可以获得的 **字典序最小** 的字符串。

**子字符串** 是字符串中的一个连续字符序列。

现有长度相同的两个字符串 `x` 和字符串 `y`，在满足 `x[i] != y[i]` 的第一个位置 `i` 上，如果 `x[i]` 在字母表中先于 `y[i]` 出现，则认为字符串 `x` 比字符串 `y` **字典序更小**。

示例 1：

输入：s = "cbabc"

输出："baabc"

解释：我们选择从下标 0 开始、到下标 1 结束的子字符串执行操作。  
可以证明最终得到的字符串是字典序最小的。

示例 2：

输入：s = "acbbc"

输出："abaab"

解释：我们选择从下标 1 开始、到下标 4 结束的子字符串执行操作。  
可以证明最终得到的字符串是字典序最小的。

示例 3：

## 解题思路一：

首先考虑一般情况，遇到a之前的数据全部变成s[i]-1即可，接着考虑特殊情况，如果全是a,那么一定有s[len-1]=a,这时把s[len-1]变成z即可，如果是形如aabbaa之类的形式，那么需要跳过所有连续的a然后进行修改。也就是说，流程是先找到第一个非a的数据，遍历每个数据直到找到或者i=len-1，如果i-1走s[len-1]='z'，否则就以碰到下一个a或者结尾为条件，对于每个字符依次-1即可

```
class Solution {
public:
    string smallestString(string s) {
        if (s.empty()) return "";
        int len = s.size();
        int i = 0;

        // 找到第一个不是 'a' 的字符的位置
        while (i < len && s[i] == 'a') {
            ++i;
        }
    }
};
```

```

    }

    // 如果全是 'a', 把最后一个 'a' 变成 'z'
    if (i == len) {
        s[len - 1] = 'z';
    }
    else {
        // 从第一个不是 'a' 的字符开始, 逐个减1直到遇到 'a' 或到末尾
        while (i < len && s[i] != 'a') {
            s[i] = s[i] - 1;
            ++i;
        }
    }

    return s;
}
};

```

## 解题思路二：

思路与解题思路一高度相似，但是把对是否是全a的判断放到了最后，设置标志位进行判断

```

#include <iostream>
#include <string>
using namespace std;

class Solution {
public:
    string smallestString(string s) {
        int n = s.size();
        bool changed = false;

        for (int i = 0; i < n; ++i) {
            if (s[i] != 'a') {
                for (int j = i; j < n && s[j] != 'a'; ++j) {
                    s[j] = s[j] - 1;
                }
                changed = true;
                break;
            }
        }

        // 判断是不是全a
        if (!changed) {
            s[n-1] = 'z';
        }

        return s;
    }
};


int main() {
    string s = "aabaa";
    Solution sol;
    s = sol.smallestString(s);
    cout << s << endl; // 输出: "aaaza"
    return 0;
}

```


```
}
```


## 题目二：两数之和

### 2. 两数相加

已解答 

中等

 相关标签

 相关企业

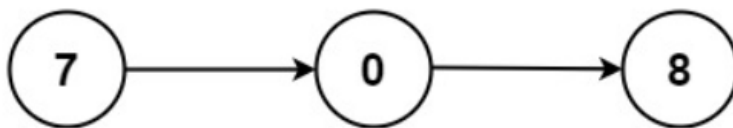
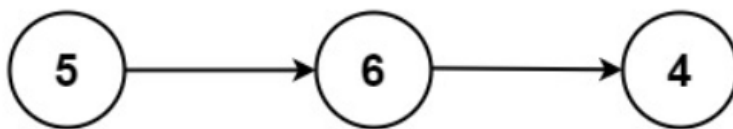
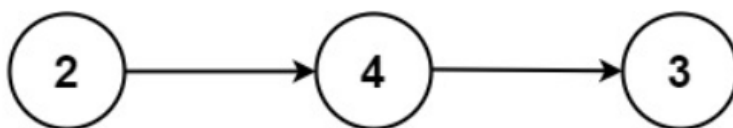
Ax

给你两个 **非空** 的链表，表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的，并且每个节点只能存储 **一位** 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1:



 10.7K

 9.9K







### 解题思路一：

我们创建一个新的链表，对于这个链表设置一个虚拟头节点0，由于题中链表的每个节点只存储一位元素，所以对于进位数据需要单独提取出来做处理。由于链表长度不一定相等，所以需要遍历完两个链表且保证进位不再产生，设置carry=0来表示进位，对于每位数字都有当前数初始化为sum=carry，然后再根据每位数相加的结构%10和/10确定进位和余位

```
#include <iostream>
#include <string>
using namespace std;

struct ListNode {
```

```

int val;
ListNode* next;
ListNode() : val(0), next(nullptr) {}
ListNode(int x) : val(x), next(nullptr) {}
ListNode(int x, ListNode* next) : val(x), next(next) {}
};

class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        // 使用一个dummy节点来简化处理和返回结果的逻辑
        ListNode dummy(0);
        ListNode* curr = &dummy;
        int carry = 0; // 进位初始化为0

        // 遍历两个链表，直到两个链表都为空且没有进位
        while (l1 != nullptr || l2 != nullptr || carry != 0) {
            int sum = carry; // 当前位的和初始化为进位值
            if (l1 != nullptr) { // 如果l1不为空，累加l1的当前值，并移动到下一个节点
                sum += l1->val;
                l1 = l1->next;
            }
            if (l2 != nullptr) { // 如果l2不为空，累加l2的当前值，并移动到下一个节点
                sum += l2->val;
                l2 = l2->next;
            }

            carry = sum / 10; // 计算新的进位
            curr->next = new ListNode(sum % 10); // 将当前位的结果存入新节点
            curr = curr->next; // 移动到新节点
        }

        return dummy.next; // 返回结果链表的头节点 (dummy.next)
    }
};

int main() {
    // 创建链表 l1: 342 在链表中逆序存储为 2 -> 4 -> 3
    ListNode* l1 = new ListNode(2);
    l1->next = new ListNode(4);
    l1->next->next = new ListNode(3);

    // 创建链表 l2: 465 在链表中逆序存储为 5 -> 6 -> 4
    ListNode* l2 = new ListNode(5);
    l2->next = new ListNode(6);
    l2->next->next = new ListNode(4);

    Solution sol;
    ListNode* result = sol.addTwoNumbers(l1, l2);

    // 输出结果链表
    while (result != nullptr) {
        cout << result->val << " ";
        result = result->next;
    }
    cout << endl;

    return 0;
}

```

```
}
```

## 解题思路二：

使用递归简化代码，对于递归的终点一定是两个链表都是空链表，如果这时候还有进位，也就是两个链表同样长，链表中最后的元素产生了进位，那么创建一个新的节点，如果没有那么就是空节点，对于其他节点，我们选择以l1作为链表的返回值，也就是说，我们得时刻保证l1非空，假设l1为空，由于我们设置了递归的终点是两个链表都为空，如果能走到判断l1为空的语句，那么l2一定不是空，这个时候我们交换l2和l1的节点来保证l1非空，接着继续计算进位，由于l1一定非空（空的话已经交换），我们需要判断l2是否为空。紧接着，保留进位，递归处理两个链表的下一个元素

```
class Solution {
public:
    // l1 和 l2 为当前遍历的节点，carry 为进位
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2, int carry = 0) {
        if (l1 == nullptr && l2 == nullptr) // 递归边界: l1 和 l2 都是空节点
            return carry ? new ListNode(carry) : nullptr; // 如果进位了，就额外创建一个节点
        if (l1 == nullptr) // 如果 l1 是空的，那么此时 l2 一定不是空节点
            swap(l1, l2); // 交换 l1 与 l2，保证 l1 非空，从而简化代码
        carry += l1->val + (l2 ? l2->val : 0); // 节点值和进位加在一起
        l1->val = carry % 10; // 每个节点保存一个数位
        l1->next = addTwoNumbers(l1->next, (l2 ? l2->next : nullptr), carry / 10); // 进位
        return l1;
    }
};
```

## 题目三：扁平化多级链表

### 430. 扁平化多级双向链表

已解答 

中等

 相关标签

 相关企业

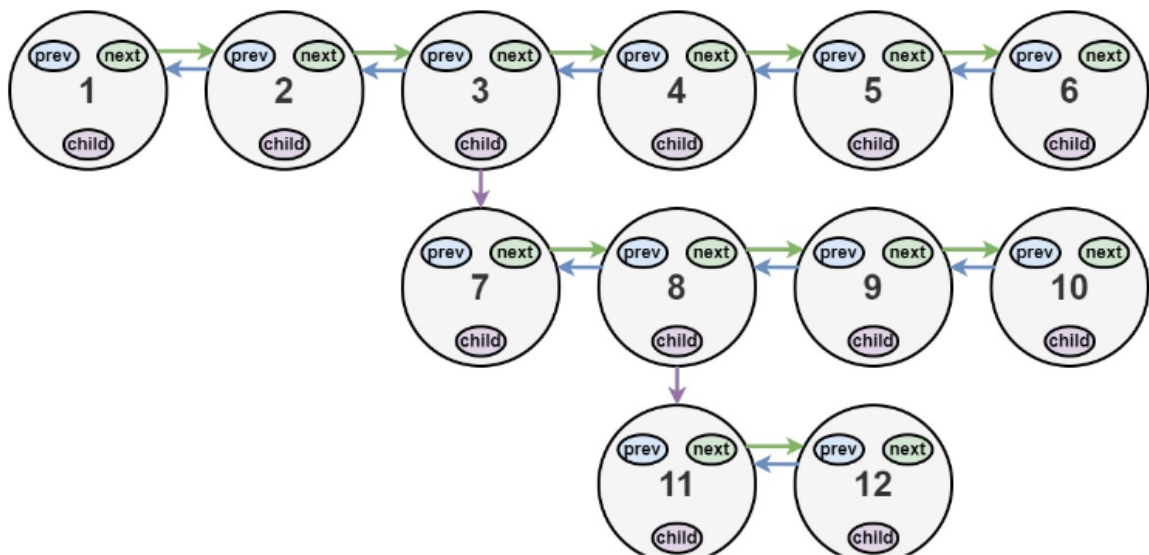
A文

你会得到一个双链表，其中包含的节点有一个下一个指针、一个前一个指针和一个额外的 **子指针**。这个子指针可能指向一个单独的双向链表，也包含这些特殊的节点。这些子列表可以有一个或多个自己的子列表，以此类推，以生成如下面的示例所示的 **多层数据结构**。

给定链表的头节点 `head`，将链表 **扁平化**，以便所有节点都出现在单层双链表中。让 `curr` 是一个带有子列表的节点。子列表中的节点应该出现在**扁平化列表**中的 `curr` 之后和 `curr.next` 之前。

返回 扁平列表的 `head`。列表中的节点必须将其 **所有** 子指针设置为 `null`。

### 示例 1:



👍 430    💬 773    ⭐    ➦    ?

### 解题思路一：

对于链表的每个节点，如果它的孩子是空，那么不做处理，如果孩子非空，这个时候得非两种情况，一种是当前节点还有后继节点，那么我们需要把这个后继节点给保留，如果没有后继节点，只需连接到这个孩子节点继续遍历即可，而观察样例我们不难发现，每个孩子节点不为空的节点，它的后继节点与当前节点出现的顺序成反比，也就是先后来处理，那么我们可以定义一个辅助栈来存储链表的后续节点，然后把后继节点改为孩子节点，孩子节点的前驱节点该为当前节点，当孩子节点连接的链表遍历结束后，如果栈中还有节点，那么这个节点成为新的后继节点，节点出栈

```
#include<iostream>
#include<string>
#include<vector>
#include<stack>
using namespace std;
```

```
// 定义链表节点结构
class Node {
public:
```

```

int val;
Node* prev;
Node* next;
Node* child;

Node(int _val) : val(_val), prev(nullptr), next(nullptr), child(nullptr) {}
};

class Solution {
public:
    Node* flatten(Node* head) {
        if (!head) return nullptr;

        Node* curr = head;
        stack<Node*> tempStack;

        while (curr) {
            // 如果当前节点有子链表
            if (curr->child) {
                // 如果当前节点有next节点，则将其压入栈中
                if (curr->next) {
                    tempStack.push(curr->next);
                }
                // 将子链表的头节点接到当前节点的next位置
                curr->next = curr->child;
                // 设置子链表头节点的prev指针
                if (curr->next) {
                    curr->next->prev = curr;
                }
                // 清空当前节点的child指针
                curr->child = nullptr;
            }
            // 如果当前节点没有next节点，但栈不为空
            else if (!curr->next && !tempStack.empty()) {
                // 弹出栈顶节点，接到当前节点的next位置
                curr->next = tempStack.top();
                tempStack.pop();
                // 设置新的next节点的prev指针
                curr->next->prev = curr;
            }
            // 移动到下一个节点
            curr = curr->next;
        }

        return head;
    }
};

void printList(Node* head) {
    Node* curr = head;
    while (curr) {
        cout << curr->val << " ";
        curr = curr->next;
    }
    cout << endl;
}

int main() {

```

```

// 创建一个示例多层双向链表
Node* head = new Node(1);
head->next = new Node(2);
head->next->prev = head;
head->next->next = new Node(3);
head->next->next->prev = head->next;
head->next->child = new Node(4);
head->next->child->next = new Node(5);
head->next->child->next->prev = head->next->child;
head->next->child->child = new Node(6);

Solution sol;
Node* flatHead = sol.flatten(head);

// 打印扁平化后的链表
printList(flatHead);

return 0;
}

```

## 解题思路二：

递归：对于每个节点，我们依然需要判断它有没有孩子节点，有的话有没有后续节点，仔细观察，不能发现，父链表和孩子链表的处理方式是一样的，有孩子节点连接到孩子节点，直到孩子节点遍历完，如果在连接之前，当前节点还有后续节点，那么连接完子节点过后，子节点的最后把当前节点给链接上，也就是说，它满足递归的要求

```

class Solution {
public:
    Node* flatten(Node* head) {
        if (!head) return nullptr;

        flattenDFS(head);
        return head;
    }

private:
    // 返回扁平化后的链表的最后一个节点
    Node* flattenDFS(Node* node) {
        Node* curr = node;
        Node* last = node;

        while (curr) {
            Node* next = curr->next;

            // 如果当前节点有子节点，处理子链表
            if (curr->child) {
                Node* childLast = flattenDFS(curr->child);

                // 将子链表插入当前节点和下一个节点之间
                curr->next = curr->child;
                curr->child->prev = curr;
                curr->child = nullptr;

                // 如果有下一个节点，将子链表的最后一个节点连接到下一个节点
                if (next) {

```



```
        childLast->next = next;
        next->prev = childLast;
    }
    //更新尾节点

    last = childLast;
}
else {
    last = curr;
}


curr = next;
}

return last;
}
};
```


## 题目四：数组中重复出现两次的元素


---

## 442. 数组中重复的数据

已解答 

中等

 相关标签

 相关企业

Ax

给你一个长度为  $n$  的整数数组 `nums`，其中 `nums` 的所有整数都在范围  $[1, n]$  内，且每个整数出现 **一次** 或 **两次**。请你找出所有出现 **两次** 的整数，并以数组形式返回。

你必须设计并实现一个时间复杂度为  $O(n)$  且仅使用常量额外空间的算法解决此问题。

示例 1:

输入: `nums = [4,3,2,7,8,2,3,1]`  
 输出: `[2,3]`

示例 2:

输入: `nums = [1,1,2]`  
 输出: `[1]`

示例 3:

输入: `nums = [1]`  
 输出: `[]`

提示:

- `n == nums.length`
- `1 <= n <= 10^5`

### 解题思路一:

由于数组的元素一定是在 $[1-n]$ , $n$ 是数组长度，那么对于