CPlusPlus

1 简介

- 开发工具：VsCode、mingw-w64
- 软件环境：Window11
- 编译工具：gcc version 8.1.0 (x86_64-posix-seh-rev0)，C/C++编译

2 目录结构

```
CPlusPlus
  ├──Array
  ├──List
  └──String
```

3 Array

3.1 最小操作次数使数组元素相等

给你一个长度为 `n` 的整数数组，每次操作将会使 `n - 1` 个元素增加 `1` 。返回让数组所有元素相等的最小操作次数。

> 因为只需要找出让数组所有元素相等的最小操作次数，所以我们不需要考虑数组中各个元素的绝对大小，即不需要真正算出数组中所有元素相等时的元素值，只需要考虑数组中元素相对大小的变化即可。
>
> 因此，每次操作既可以理解为使 n−1 个元素增加 1，也可以理解使 1个元素减少 1。显然，后者更利于我们的计算。
>
> 于是，要计算让数组中所有元素相等的操作数，我们只需要计算将数组中所有元素都减少到数组中元素最小值所需的操作数，即计算
>
> $$\sum_{i=0}^{n-1}(nums[i]-min(nums))$$
>
> 其中$min(nums)$为数组 nums 中元素的最小值

题解见minMoves.cpp

复杂度分析：

- 时间复杂度：O(n)，其中 n 为数组中的元素数量。我们需要一次遍历求出最小值，一次遍历计算操作次数。
- 空间复杂度：O(1)

3.2 三数之和

给你一个整数数组 `nums` ，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k` ，同时还满足 `nums[i] + nums[j] + nums[k] == 0` 。请

你返回所有和为 `0` 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

题目中要求找到所有「不重复」且和为 0 的三元组，这个「不重复」的要求使得我们无法简单地使用三重循环枚举所有的三元组。

这是因为在最坏的情况下，数组中的元素全部为 0，任意一个三元组的和都为 0。如果我们直接使用三重循环枚举三元组，会得到 O(N3) 个满足题目要求的三元组（其中 N 是数组的长度）时间复杂度至少为 O(N3)。在这之后，我们还需要使用哈希表进行去重操作，得到不包含重复三元组的最终答案，又消耗了大量的空间。这个做法的时间复杂度和空间复杂度都很高。

「不重复」的本质是什么？我们保持三重循环的大框架不变，只需要保证：

- 第二重循环枚举到的元素不小于当前第一重循环枚举到的元素；
- 第三重循环枚举到的元素不小于当前第二重循环枚举到的元素。

也就是说，我们枚举的三元组 (a,b,c) 满足 a≤b≤c，保证了只有 (a,b,c) 这个顺序会被枚举到，而 (b,a,c)、(c,b,a)等等这些不会，这样就减少了重复。要实现这一点，我们可以将数组中的元素从小到大进行排序，随后使用普通的三重循环就可以满足上面的要求。

同时，对于每一重循环而言，相邻两次枚举的元素不能相同，否则也会造成重复。

这种方法的时间复杂度仍然为 O(N^3)，毕竟我们还是没有跳出三重循环的大框架。然而它是很容易继续优化的，可以发现，如果我们固定了前两重循环枚举到的元素 a 和 b，那么只有唯一的 c 满足 a+b+c=0。当第二重循环往后枚举一个元素 b′时，由于 b′>b，那么满足 a+b′+c′=0 的 c′一定有 c′<c，即 c′在数组中一定出现在 c 的左侧。也就是说，我们可以从小到大枚举 b，同时从大到小枚举 c，即第二重循环和第三重循环实际上是并列的关系。

有了这样的发现，我们就可以保持第二重循环不变，而将**第三重循环变成一个从数组最右端开始向左移动的指针**，这个方法就是我们常说的「双指针」，当我们需要枚举数组中的两个元素时，如果我们发现随着第一个元素的递增，第二个元素是递减的，那么就可以使用双指针的方法，将枚举的时间复杂度从 O(N^2)减少至 O(N)。为什么是 O(N) 呢？这是因为在枚举的过程每一步中，「左指针」会向右移动一个位置（也就是题目中的 b），而「右指针」会向左移动若干个位置，这个与数组的元素有关，但我们知道它一共会移动的位置数为 O(N)，均摊下来，每次也向左移动一个位置，因此时间复杂度为 O(N)。

注意到我们的伪代码中还有第一重循环，时间复杂度为 O(N)，因此枚举的总时间复杂度为O(N^2)。由于排序的时间复杂度为 O(NlogN)，在渐进意义下小于前者，因此算法的总时间复杂度为 O(N2)。

额外优化点：

1. 设 s = nums[first] + nums[first+1] + nums[first+2]，如果 s > 0，由于数组已经排序，后面无论怎么选，选出的三个数的和不会比 s 还小，所以只要 s > 0 就可以直接 break 外层循环了。
2. 如果 nums[first] + nums[n-2] + nums[n-1] < 0，由于数组已经排序，nums[first] 加上后面任意两个数都是小于 0 的，所以下面的双指针就不需要跑了。但是后面可能有更大的 nums[first]，所以还需要继续枚举，continue 外层循环。

题解见threesum.cpp

复杂度分析

- 时间复杂度：O(n^2)，其中 n 为 nums 的长度。排序 O(nlogn)。外层循环枚举第一个数，，做法是 O(n) 双指针。所以总的时间复杂度为 O(n^2)。
- 空间复杂度：O(1)，仅用到若干变量（忽略排序的栈开销）。

3.3 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 **围成一圈** ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警** 。

给定一个代表每个房屋存放金额的非负整数数组，计算你 **在不触动警报装置的情况下** ，今晚能够偷窃到的最高金额。

> 首先考虑最简单的情况。如果只有一间房屋，则偷窃该房屋，可以偷窃到最高总金额。如果只有两间房屋，则由于两间房屋相邻，不能同时偷窃，只能偷窃其中的一间房屋，因此选择其中金额较高的房屋进行偷窃，可以偷窃到最高总金额。
>
> 注意到当房屋数量不超过两间时，最多只能偷窃一间房屋，因此不需要考虑首尾相连的问题。如果房屋数量大于两间，就必须考虑首尾相连的问题，第一间房屋和最后一间房屋不能同时偷窃。
>
> 如何才能保证第一间房屋和最后一间房屋不同时偷窃呢？如果偷窃了第一间房屋，则不能偷窃最后一间房屋，因此偷窃房屋的范围是第一间房屋到最后第二间房屋；如果偷窃了最后一间房屋，则不能偷窃第一间房屋，因此偷窃房屋的范围是第二间房屋到最后一间房屋。
>
> 假设数组 nums 的长度为 n。如果不偷窃最后一间房屋，则偷窃房屋的下标范围是 [0, n-2]；如果不偷窃第一间房屋，则偷窃房屋的下标范围是 [1,n−1]。对于两段下标范围分别计算可以偷窃到的最高总金额，其中的最大值即为在 n 间房屋中可以偷窃到的最高总金额。
>
> 假设偷窃房屋的下标范围是 [start,end]，用 dp[i] 表示在下标范围 [start,i] 内可以偷窃到的最高总金额，那么就有如下的状态转移方程：
>
> $$dp[i] = max(dp[i-2] + nums[i], dp[i-1])$$
>
> 边界条件为：
>
> $$\begin{cases} dp[start] = nums[start] \\ dp[start] = max(nums[start], num[start+1]) \end{cases}$$
>
> 计算得到 dp[end] 即为下标范围 [start,end] 内可以偷窃到的最高总金额。
>
> 分别取 (start,end)=(0,n−2) 和 (start,end)=(1,n−1) 进行计算，取两个 dp[end] 中的最大值，即可得到最终结果。

题解见rob.cpp

复杂度分析：

- 时间复杂度：O(n)，其中 n是数组的长度。需要对数组遍历两次，计算可以偷窃到的最高总金额。
- 空间复杂度：O(1)。

4 List

```
/**
 * Definition for singly-linked list.
 * 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) {}
 * };
 */
```

4.1 奇偶链表

给定单链表的头节点 `head` ，将所有索引为奇数的节点和索引为偶数的节点分别组合在一起，然后返回重新排序的列表。

第一个节点的索引被认为是**奇数** ， **第二个节点**的索引为**偶数** ，以此类推。

请注意，偶数组和奇数组内部的相对顺序应该与输入时保持一致。

> 如果链表为空，则直接返回链表。
> 对于原始链表，每个节点都是奇数节点或偶数节点。头节点是奇数节点，头节点的后一个节点是偶数节点，相邻节点的奇偶性不同。因此可以将奇数节点和偶数节点分离成奇数链表和偶数链表，然后将偶数链表连接在奇数链表之后，合并后的链表即为结果链表。

题解见oddEvenList.cpp

复杂度分析:

- 时间复杂度：O(n)，其中 n是链表的节点数。需要遍历链表中的每个节点，并更新指针
- 空间复杂度：O(1)。只需要维护有限的指针。

4.2 反转链表

给你单链表的头节点 `head` ，请你反转链表，并返回反转后的链表。

> 假设链表为:$n_1->\ldots->n_{k-1}->n_k->n_{k+1}->\ldots->n_m->\varnothing$
>
> 若从节点n_{k+1}到n_m已经被反转，而我们正处于n_k
>
> $n_1->\ldots->n_{k-1}->n_k->n_{k+1}<-\ldots<-n_m$
>
> 我们希望n_{k+1}的下一个节点指向n_k
>
> 所以，$n_k.next.next=n_k$
>
> 需要注意的是n_1下一个节点必须指向 $\varnothing$。如果忽略了这一点，链表中可能会产生环。

题解见reverseList.cpp

复杂度分析:

- 时间复杂度：O(n)，其中 n是链表的长度。需要对链表的每个节点进行反转操作。
- 空间复杂度：O(n)，其中 n是链表的长度。空间复杂度主要取决于递归调用的栈空间，最多为 n 层。

5 String

5.1 同构字符串

给定两个字符串 `s` 和 `t` ，判断它们是否是同构的。

如果 `s` 中的字符可以按某种映射关系替换得到 `t` ，那么这两个字符串是同构的。

每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。不同字符不能映射到同一个字符上，相同字符只能映射到同一个字符上，字符可以映射到自己本身。

示例 1：

```
输入：s = "egg", t = "add"
输出：true
```

示例 2：

```
输入：s = "foo", t = "bar"
输出：false
```

示例 3：

```
输入：s = "paper", t = "title"
输出：true
```

> 我们维护两张哈希表，第一张哈希表 s2t 以 s 中字符为键，映射至 t 的字符为值，第二张哈希表 t2s 以 t 中字符为键，映射至 s 的字符为值。从左至右遍历两个字符串的字符，不断更新两张哈希表，如果出现冲突（即当前下标 index 对应的字符 s[index] 已经存在映射且不为 t[index] 或当前下标 index 对应的字符 t[index] 已经存在映射且不为 s[index]）时说明两个字符串无法构成同构，返回 false
>
> 如果遍历结束没有出现冲突，则表明两个字符串是同构的，返回 true 即可。

题解见isIsomorphic.cpp

复杂度分析：

- 时间复杂度：O(n)，其中 n 为字符串的长度。我们只需同时遍历一遍字符串 s 和 t即可。
- 空间复杂度：O(|Σ|)，其中 Σ 是字符串的字符集。哈希表存储字符的空间取决于字符串的字符集大小，最坏情况下每个字符均不相同，需要 O(|Σ|) 的空间。

5.2 替换空格

请实现一个函数，把字符串 `s` 中的每个空格替换成"%20"。

> 遍历字符串，使用三目运算符判断空格并替换重组

题解见replaceSpace.cpp

复杂度分析：

- 时间复杂度：O(n)，其中 n 为字符串的长度。我们只需遍历一遍字符串即可。
- 空间复杂度：O(n)，额外创建字符串，长度为s.size()+2*n。

5.3 左旋字符串

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

> 获取str[n:] 和 str[:n]子串，目标串target = str[:n] + str[n:];

题解见reverseLeftWords.cpp

复杂度分析：

- 时间复杂度：O(n)，其中 n 为左旋字符串的长度。我们只需遍历一遍左旋字符串即可。
- 空间复杂度：O(n)。

5.4 反转字符串中的单词

给你一个字符数组 `s` ，反转其中 **单词** 的顺序。

单词 的定义为：单词是一个由非空格字符组成的序列。`s` 中的单词将会由单个空格分隔。

必须设计并实现 **原地** 解法来解决此问题，即不分配额外的空间。

示例 1：

```
输入：s = ["t","h","e"," ","s","k","y"," ","i","s"," ","b","l","u","e"]
输出：["b","l","u","e"," ","i","s"," ","s","k","y"," ","t","h","e"]
```

示例 2：

```
输入：s = ["a"]
输出：["a"]
```

> 1.第一次翻转： 以“单词”为单位进行翻转，调整单词之间的位置关系
>
> 2.第二次翻转： 以char为单位进行翻转，对每个单词，调整单词内部的位置关系
>
> 3.这两次调整的顺序不分先后

题解见reverseWords.cpp

复杂度分析：

- 时间复杂度：O(n)，其中 n 为字符串的长度。
- 空间复杂度：O(1)。

5.5 最长回文子串

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

如果字符串的反序与原始字符串相同，则该字符串称为回文字符串。

示例 1：

输入： `s = "babad"`
输出： `"bab"`
解释： `"aba"` 同样是符合题意的答案。

示例 2：

输入： `s = "cbbd"`
输出： `"bb"`

对于一个子串而言，如果它是回文串，并且长度大于 222，那么将它首尾的两个字母去除之后，它仍然是个回文串。例如对于字符串 “ababa”，如果我们已经知道 “bab” 是回文串，那么 “ababa” 一定是回文串，这是因为它的首尾两个字母都是 “a”。

根据这样的思路，我们就可以用动态规划的方法解决本题。我们用 P(i,j)表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 s[i:j]）是否为回文串

$$P(i,j)=\begin{cases}true, & \text{如果子串} S_i \dots S_j \text{是回文串} \\ false, & \text{其他情况}\end{cases}$$

这里的「其它情况」包含两种可能性：

- s[i,j]本身不是一个回文子串；
- i > j, 此时s[i,j]本身不合法。

那么我们就可以写出状态转移方程：

$$\begin{cases}P(i,i) & = & true \\ P(i,i+1) & = & (s_i == s_{i+1}) \\ P(i,j) & = & P(i+1,j-1) \cap (s_i == s_j)\end{cases}$$

其中状态转移链为：$P(i,j) < -P(i+1,j-1) < -P(i+2,j-2) < - \cdots <$ -某一边界情况

可以发现，所有的状态在转移的时候的可能性都是唯一的。也就是说，我们可以从每一种边界情况开始「扩展」，也可以得出所有的状态对应的答案。

边界情况即为子串长度为 1 或 2 的情况。我们枚举每一种边界情况，并从对应的子串开始不断地向两边扩展。如果两边的字母相同，我们就可以继续扩展，例如从 P(i+1,j−1) 扩展到 P(i,j)；如果两边的字母不同，我们就可以停止扩展，因为在这之后的子串都不能是回文串了。

「边界情况」对应的子串实际上就是我们「扩展」出的回文串的「回文中心」。本质即为：我们枚举所有的「回文中心」并尝试「扩展」，直到无法扩展为止，此时的回文串长度即为此「回文中心」下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

题解见selectLongestPalindrome.cpp

复杂度分析

- 时间复杂度：O(n^2)，其中 n 是字符串的长度。长度为 1和 2的回文中心分别有 n 和 n−1个，每个回文中心最多会向外扩展 O(n) 次。
- 空间复杂度：O(1)。