程设错题 21-24 2024-11-1 2024-11-4

1. Leetcode 453 最小操作次数使数组元素相等

不要所有方法都使用最原始的编程方法解决！

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。

示例 1：

```
输入：nums = [1,2,3]
输出：3
解释：
只需要3次操作（注意每次操作会增加两个元素的值）：
[1,2,3]  =>  [2,3,3]  =>  [3,4,3]  =>  [4,4,4]
```

解法1 最基本解法（计算+枚举）

时间复杂度过高（最坏时间复杂度：O(n^2)）

```
class Solution {
public:
    int minMoves(vector<int>& nums) {
        if(nums.size()==1){
            return 0;
        }
        int max=nums[0];
        long long sum=0;
        for(auto num:nums){
            sum+=num;
            if(num>max){
                max=num;
            }
        }
        long long diff=nums.size()*max-sum;
        while(diff%(nums.size()-1)){
            diff+=(nums.size());
        }
        bool flag=0;
        long long standard=(long(diff)+sum)/nums.size();
        while(!flag){
            for(auto num:nums){
                if(standard-num>(diff/(nums.size()-1))){
                    flag=0;
                    standard+=nums.size()-1;
                    diff+=nums.size()*(nums.size()-1);
                    break;
                }
                flag=1;
            }
        }
        return diff/(nums.size()-1);
```

```
    }
};
```

解法2 巧妙的转化

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。

这个方法没有次数的限制，所以不用通过枚举找到可行解！

```
class Solution {
public:
    int minMoves(vector<int>& nums) {
        int minNum = *min_element(nums.begin(),nums.end());
        //找到数组中的最小值
        int res = 0;
        for (int num : nums) {
            res += num - minNum;
        }
        //通过累加得到一共需要操作的次数
        return res;
    }
};
```

2. Leetcode 448 找到数组中消失的数字

给你一个含 `n` 个整数的数组 `nums` ，其中 `nums[i]` 在区间 `[1, n]` 内。请你找出所有在 `[1, n]` 范围内但没有出现在 `nums` 中的数字，并以数组的形式返回结果。

解法1 使用动态数组（new&delete）

```
class Solution {
public:
    vector<int> findDisappearedNumbers(vector<int>& nums) {
        vector <int> successful;
        int n=nums.size();
        bool *list=new bool[n];
        for(int i=0;i<n;i++){
            list[i]=false;
        }
        for(auto num:nums){
            list[num-1]=true;
        }
        for(int i=0;i<n;i++){
            if(!list[i]){
                successful.push_back(i+1);
            }
```

```
        }
        delete[] list;
        return successful;

    }
};
```

解法2 原地哈希表

不开辟新的内存空间，在原来的数组上进行操作

我们可以用一个哈希表记录数组 nums 中的数字，由于数字范围均在 [1,n] 中，记录数字后我们再利用哈希表检查 [1,n] 中的每一个数是否出现，从而找到缺失的数字。

由于数字范围均在 [1,n] 中，我们也可以用一个长度为 n 的数组来代替哈希表。这一做法的空间复杂度是 O(n) 的。我们的目标是优化空间复杂度到 O(1)。

注意到 nums 的长度恰好也为 n，能否让 nums 充当哈希表呢？

由于 nums 的数字范围均在 [1,n] 中，我们可以**利用这一范围之外的数字**，来表达「是否存在」的含义。

具体来说，遍历 nums，每遇到一个数 x，就让 nums[x−1] 增加 n。由于 nums 中所有数均在 [1,n] 中，增加以后，这些数必然大于 n。最后我们遍历 nums，若 nums[i] 未大于 n，就说明没有遇到过数 i+1。这样我们就找到了缺失的数字。

注意，**当我们遍历到某个位置时，其中的数可能已经被增加过，因此需要对 n 取模**来还原出它本来的值。

取模&加倍数（进制）可以让一个数通过不同的运算储存不同的信息，例如32以10为进制可以储存3,2两个信息

```
class Solution {
public:
    vector<int> findDisappearedNumbers(vector<int>& nums) {
        int n = nums.size();
        for (auto& num : nums) {
            int x = (num - 1) % n;
            //-1 problem，注意数组下标从0开始
            //通过%取余运算得到原来的值
            nums[x] += n;
        }
        vector<int> ret;
        for (int i = 0; i < n; i++) {
            if (nums[i] <= n) {
                //证明这个值未出现过
                ret.push_back(i + 1);
            }
        }
        return ret;
    }
};
```

3. Leetcode 455 贪心算法分配饼干

补充：什么时候使用贪心算法：保证局部最优解=全局最优解！

1. 最优子结构性质

定义：一个问题具有最优子结构性质，意味着问题的最优解可以通过其子问题的最优解来构建。这是动态规划和贪婪算法的共同特征。

作用：如果一个问题具有最优子结构性质，贪婪算法可以通过解决每个子问题的最优解来逐步构建全局最优解。例如，在最小生成树问题中，局部最优的边选择能够合并成全局最优的生成树。

2. 无后效性

定义：无后效性是指当前选择不会影响未来选择的可行性或最优性。这意味着每一步做出的选择不会限制后续步骤中其他选择的可能性。

作用：无后效性确保了贪婪算法在每一步做出局部最优选择时，不会对后续步骤产生负面影响。例如，在活动选择问题中，选择结束时间最早的活动不会影响后续活动的选择空间。

3. 贪婪选择性质的应用场景

一些典型的应用场景包括：

- **活动选择问题**：选择一组互不重叠的活动，贪婪选择性质确保选择最早结束的活动是最优的。
- **最小生成树问题**：通过选择权重最小且不形成环的边来构建最小生成树。
- **单源最短路径问题（无负权边）**：在Dijkstra算法中，每次选择当前未访问顶点中距离最小的顶点。

4. 不适用贪婪算法的问题

并不是所有问题都适合用贪婪算法来解决。对于不具备贪婪选择性质的问题，贪婪算法可能无法找到全局最优解。例如：

- **旅行商问题（TSP）**：贪婪算法可能无法找到最短路径，因为局部最优选择可能导致全局次优解。
- **背包问题**：在0-1背包问题中，贪婪算法可能无法找到最优解，因为选择价值密度最高的物品可能导致无法装入其他更有价值的组合。

5. 识别适用性

在应用贪婪算法之前，识别问题是否具备贪婪选择性质和最优子结构是关键。通常需要通过理论分析或构造反例来验证贪婪策略的有效性。

总结来说，贪婪算法适用于那些具有最优子结构和无后效性的问题。在这些问题中，贪婪选择性质确保每一步的局部最优选择最终能组合成全局最优解。对于不具备这些性质的问题，可能需要其他算法（如动态规划或回溯）来找到全局最优解。

题目：

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 `i`，都有一个胃口值 `g[i]`，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 `j`，都有一个尺寸 `s[j]` 。如果 `s[j] >= g[i]`，我们可以将这个饼干 `j` 分配给孩子 `i` ，这个孩子会得到满足。你的目标是满足尽可能多的孩子，并输出这个最大数值。

解法1:排序后遍历（繁琐做法）

时间复杂度：$O(m\log m+n\log n+m\cdot n)$

mlogm和nlogn是排序的时间复杂度，mn是最后遍历的时间复杂度

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(),g.end());
        sort(s.begin(),s.end());
        int count=0;
        int select=0;
        for(int i=0;i<g.size()&&select<s.size();i++){
            for(int j=select;j<s.size();j++){
                if(s[j]>=g[i]){
                    count++;
                    j++;
                    select=j;
                    break;
                }
            }
            //其实这里通过select不断提升j的枚举起点有点类似于双指针策略，但是代码的冗余之处在于如果出现遍历完还没有找到的情况，应该直接跳出外循环（我们已经排序过数组了）
        }
        return count;
    }
};
```

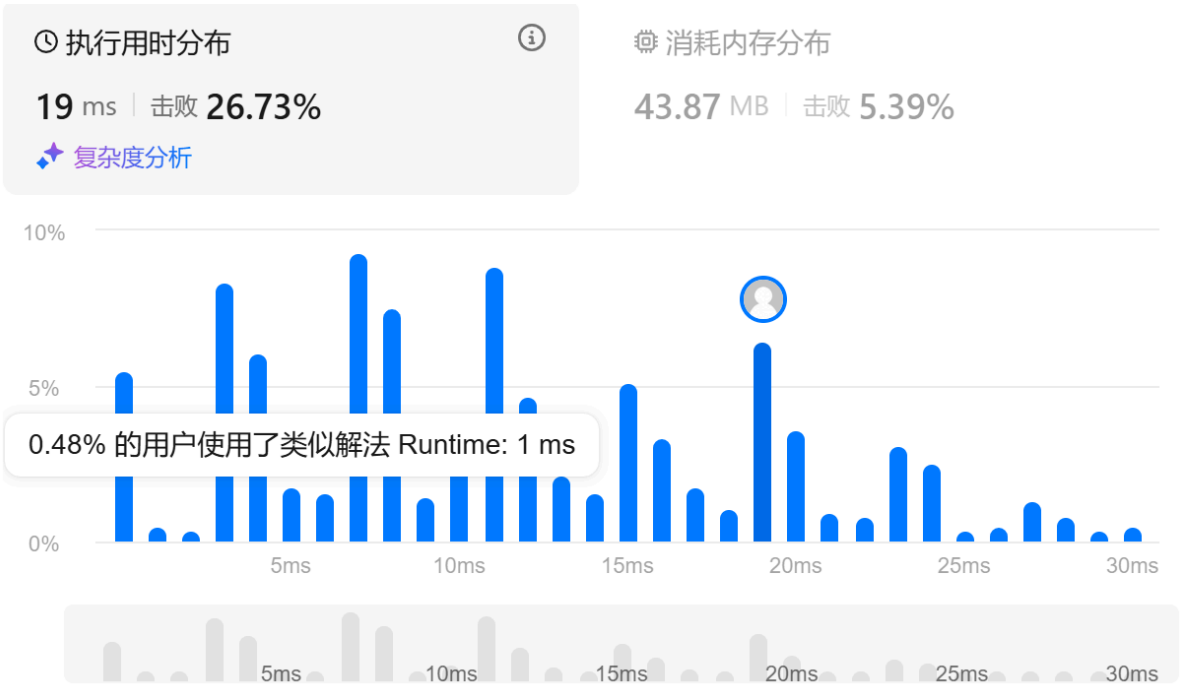
解法1的小修改:

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(),g.end());
        sort(s.begin(),s.end());
        int count=0;
        int select=0;
        bool flag=false;
        for(int i=0;i<g.size()&&select<s.size();i++){
            for(int j=select;j<s.size();j++){
                if(s[j]>=g[i]){
                    flag=true;
                    count++;
                    j++;
                    select=j;
                    break;
                }
            }
            if(!flag){
                return count;
            }
        }
        return count;
```

```
    }
};
```

(这样就不会超时了，但是代码还有优化的空间)

- 这里其实不需要写两层for循环控制两个指针!



解法2：优化使用双指针

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());
        int count = 0;
        int i = 0, j = 0;
        //define two pointers
        while (i < g.size() && j < s.size()) {
            if (s[j] >= g[i]) {
                count++;
                i++;
                //if allocate successfully,then the both pointers get forward
            }
            j++;
            //whenever the allocate, the biscuit pointer gets forward
        }
        return count;
    }
};
```

时间复杂度：O(mlogm)+O(nlogn)+O(m+n)=O(mlogm+nlogn)

补充：双指针

双指针是一种常用的算法技巧，尤其在处理数组或链表等线性数据结构时非常有效。它通过使用两个指针来遍历数据结构，能够在某些情况下优化时间复杂度或简化逻辑。以下是双指针技巧的一些常见应用场景和简单介绍：

1. 快慢指针

- **应用场景**：检测链表中的环、找到链表的中间节点。
- **方法**：
  - 使用两个指针，一个快指针每次移动两步，一个慢指针每次移动一步。
  - 如果快指针和慢指针相遇，说明存在环。
  - 找中间节点时，当快指针到达链表末尾时，慢指针正好在中间。

2. 左右指针

- **应用场景**：解决排序数组中的问题，如二分查找、三数之和。
- **方法**：
  - 初始化两个指针，分别指向数组的两端。
  - 根据问题的要求，向中间移动指针。
  - 常用于查找满足某种条件的对或子序列。
  - 也可以实现动态的查找。

3. 滑动窗口

- **应用场景**：解决子数组问题，如最长无重复子串、最小覆盖子串。
- **方法**：
  - 使用两个指针定义一个窗口，初始时窗口为空。
  - 移动右指针扩展窗口，移动左指针缩小窗口，直到满足条件。

- 在每次窗口变化时更新结果。

示例：使用双指针解决有序数组中的两数之和

假设我们有一个有序数组，想找到两个数，使它们的和等于目标值。

```
vector<int> twoSum(vector<int>& numbers, int target) {
    int left = 0, right = numbers.size() - 1;
    while (left < right) {
        int sum = numbers[left] + numbers[right];
        if (sum == target) {
            return {left + 1, right + 1}; // 返回索引从1开始
        } else if (sum < target) {
            left++; // 增加左指针以增加总和
        } else {
            right--; // 减少右指针以减少总和
        }
    }
    return {}; // 如果没有找到，返回空
}
```

总结

双指针技巧通过合理移动两个指针，可以有效地减少遍历次数或简化算法逻辑。选择合适的双指针策略（如快慢指针、左右指针、滑动窗口）可以帮助解决许多复杂的算法问题。

4. ACwing 633 两数平方和判断

给定一个非负整数 `c` ，你要判断是否存在两个整数 `a` 和 `b`，使得 `a^2 + b^2 = c` 。

示例 1：

```
输入：c = 5
输出：true
解释：1 * 1 + 2 * 2 = 5
```

示例 2：

```
输入：c = 3
输出：false
```

提示：

- `0 <= c <= 2^31 - 1`

最暴力的解法：双循环暴力枚举（复杂度：O(n^2)）

优化方法：使用sqrt函数省去一层循环

```
class Solution {
public:
    bool judgeSquareSum(int c) {
        double b;
        for(long long  i=0;(i)*i<=c;i++){
```

```
            b=sqrt(c-i*i);{
                if(b==int(b)){
                    return true;
                }
            }
        }
        return false;
    }
};
```

补充方法：使用双指针枚举

不失一般性，可以假设 $a \leq b$。初始时 $a = 0$，$b = \sqrt{c}$，进行如下操作：

- 如果 $a^2 + b^2 = c$，我们找到了题目要求的一个解，返回 true；
- 如果 $a^2 + b^2 < c$，此时需要将 a 的值加 1，继续查找；
- 如果 $a^2 + b^2 > c$，此时需要将 b 的值减 1，继续查找。

当 $a = b$ 时，结束查找，此时如果仍然没有找到整数 a 和 b 满足 $a^2 + b^2 = c$，则说明不存在题目要求的解，返回 false。

```
class Solution {
public:
    bool judgeSquareSum(int c) {
        long left = 0;
        long right = (int)sqrt(c);
        while (left <= right) {
            long sum = left * left + right * right;
            if (sum == c) {
                return true;
            } else if (sum > c) {
                right--;
            } else {
                left++;
            }
        }
        return false;
    }
};
```