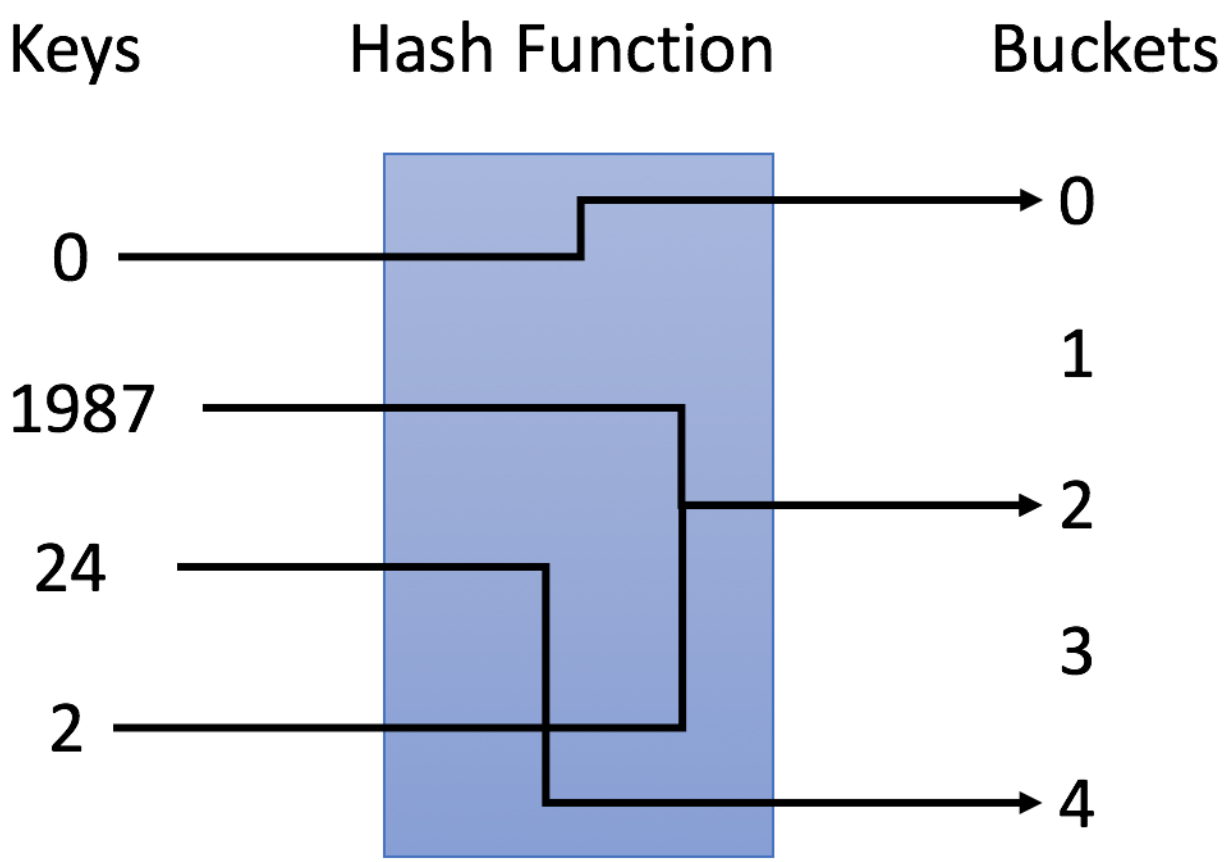
# 哈希表的原理

哈希表的关键思想是使用哈希函数`将键映射到存储桶`。更确切地说，

1. 当我们插入一个新的键时，哈希函数将决定该键应该分配到哪个桶中，并将该键存储在相应的桶中；
2. 当我们想要搜索一个键时，哈希表将使用相同的哈希函数来查找对应的桶，并只在特定的桶中进行搜索。

## 示例




在示例中，我们使用`y = x % 5`作为哈希函数。让我们使用这个例子来完成插入和搜索策略：

1. 插入：我们通过哈希函数解析键，将它们映射到相应的桶中。
    - 例如，1987 分配给桶 2，而 24 分配给桶 4。
2. 搜索：我们通过相同的哈希函数解析键，并仅在特定存储桶中搜索。
    - 如果我们搜索 1987，我们将使用相同的哈希函数将1987 映射到 2。因此我们在桶 2 中搜索，我们在那个桶中成功找到了 1987。
    - 例如，如果我们搜索 23，将映射 23 到 3，并在桶 3 中搜索。我们发现 23 不在桶 3 中，这意味着 23 不在哈希表中。

# 设计哈希表的关键

在设计哈希表时，你应该注意两个基本因素。

1. 哈希函数

哈希函数是哈希表中最重要的组件，该哈希表用于将键映射到特定的桶。在上一篇文章中的示例中，我们使用 `y = x % 5` 作为散列函数，其中 `x` 是键值，`y` 是分配的桶的索引。

散列函数将取决于 `键值的范围` 和 `桶的数量`。

下面是一些哈希函数的示例：

| Key Type | Key Range | Number of Buckets | Hash Function Example |
| --- | --- | --- | --- |
| integer | 0 to 100,000 | 1000 | y = x % 1000 |
| char | 'a' to 'z' | 26 | y = x – 'a' |
| array of integers | size < 10, each number ∈ [0,1] | 1024 | $y = x_0 * 2^0 + x_1 * 2^1 + ... + x_9 * 2^9$ |
| array of integers | size < 10, each number ∈ [0,3] | 1024 | $y = x_0 * 4^0 + x_1 * 4^1 + ... + x_4 * 4^4$ |

* In the table above, we use $x$ to represent the key and $y$ to represent our hash result.

哈希函数的设计是一个开放的问题。其思想是尽可能将键分配到桶中，理想情况下，完美的哈希函数将是键和桶之间的一对一映射。然而，在大多数情况下，哈希函数并不完美，它需要在桶的数量和桶的容量之间进行权衡。

2. 冲突解决

理想情况下，如果我们的哈希函数是完美的一对一映射，我们将不需要处理冲突。不幸的是，在大多数情况下，冲突几乎是不可避免的。例如，在我们之前的哈希函数（$y = x \ \% \ 5$）中，1987 和 2 都分配给了桶 2，这是一个 `冲突`。

冲突解决算法应该解决以下几个问题：

1. 如何组织在同一个桶中的值?
2. 如果为同一个桶分配了太多的值，该怎么办?
3. 如何在特定的桶中搜索目标值?

根据我们的哈希函数，这些问题与 `桶的容量` 和可能映射到 `同一个桶` 的 `键的数目` 有关。

让我们假设存储最大键数的桶有 `N` 个键。

通常，如果 $N$ 是常数且很小，我们可以简单地使用一个数组将键存储在同一个桶中。如果 $N$ 是可变的或很大，我们可能需要使用 `高度平衡的二叉树` 来代替.。

## 705.设计哈希集合（简单）

1. 题目描述

不使用任何内建的哈希表库设计一个哈希集合

具体地说，你的设计应该包含以下的功能

- `add(value)`：向哈希集合中插入一个值。
- `contains(value)` ：返回哈希集合中是否存在这个值。
- `remove(value)`：将给定值从哈希集合中删除。如果哈希集合中没有这个值，什么也不做。

**示例:**

```
MyHashSet hashSet = new MyHashSet();
hashSet.add(1);
hashSet.add(2);
hashSet.contains(1);    // 返回 true
hashSet.contains(3);    // 返回 false (未找到)
hashSet.add(2);
hashSet.contains(2);    // 返回 true
hashSet.remove(2);
hashSet.contains(2);    // 返回  false (已经被删除)
```

**注意：**

- 所有的值都在 `[1, 1000000]` 的范围内。
- 操作的总数目在 `[1, 10000]` 范围内。
- 不要使用内建的哈希集合库。

2. 简单实现

使用数组下标作为Key

```
class MyHashSet {
public:
    bool has[1000001] = {false};
    MyHashSet() {}
    void add(int key) {
        has[key] = true;
    }
    void remove(int key) {
        has[key] = false;
    }
    bool contains(int key) {
        return has[key];
    }
};
```

## 706.设计哈希映射（简单）

1. 题目描述

不使用任何内建的哈希表库设计一个哈希映射

具体地说，你的设计应该包含以下的功能

- `put(key, value)`：向哈希映射中插入(键,值)的数值对。如果键对应的值已经存在，更新这个值。
- `get(key)`：返回给定的键所对应的值，如果映射中不包含这个键，返回-1。
- `remove(key)`：如果映射中存在这个键，删除这个数值对。

**示例：**

```
MyHashMap hashMap = new MyHashMap();
hashMap.put(1, 1);
hashMap.put(2, 2);
hashMap.get(1);            // 返回 1
hashMap.get(3);            // 返回 -1 (未找到)
hashMap.put(2, 1);         // 更新已有的值
hashMap.get(2);            // 返回 1
hashMap.remove(2);         // 删除键为2的数据
hashMap.get(2);            // 返回 -1 (未找到)
```

**注意：**

- 所有的值都在 `[1, 1000000]` 的范围内。
- 操作的总数目在 `[1, 10000]` 范围内。
- 不要使用内建的哈希库。

2. 简单实现

```
class MyHashMap {
public:
    vector<int> hash;
    MyHashMap() {
        hash = vector<int>(1000001, -1);
    }
    void put(int key, int value) {
        hash[key] = value;
    }
    int get(int key) {
        return hash[key];
    }
    void remove(int key) {
        hash[key] = -1;
    }
};
```

# 复杂度分析

如果总共有 `M` 个键，那么在使用哈希表时，可以很容易地达到 `O(M)` 的空间复杂度。

但是，你可能已经注意到哈希表的时间复杂度与设计有很强的关系。

我们中的大多数人可能已经在每个桶中使用 `数组` 来将值存储在同一个桶中，理想情况下，桶的大小足够小时，可以看作是一个 `常数`。插入和搜索的时间复杂度都是 `O(1)`。

但在最坏的情况下，桶大小的最大值将为 `N`。插入时时间复杂度为 `O(1)`，搜索时为 `O(N)`。

# 内置哈希表的原理

内置哈希表的典型设计是：

1. 键值可以是任何 `可哈希化的` 类型。并且属于可哈希类型的值将具有 `哈希码` 。此哈希码将用于映射函数以获取存储区索引。
2. 每个桶包含一个 `数组` ，用于在初始时将所有值存储在同一个桶中。
3. 如果在同一个桶中有太多的值，这些值将被保留在一个 `高度平衡的二叉树搜索树` 中。

插入和搜索的平均时间复杂度仍为 O(1)。最坏情况下插入和搜索的时间复杂度是 `O(logN)` ，使用高度平衡的 BST。这是在插入和搜索之间的一种权衡。

# 哈希集合的应用

# C++ STL unorderred_set 用法

```
#include <unordered_set>                // 0. include the library
int main() {
    // 1. initialize a hash set
    unordered_set<int> hashset;
    // 2. insert a new key
    hashset.insert(3);
    hashset.insert(2);
    hashset.insert(1);
    // 3. delete a key
    hashset.erase(2);
    // 4. check if the key is in the hash set
    if (hashset.count(2) <= 0) {
        cout << "Key 2 is not in the hash set." << endl;
    }
    // 5. get the size of the hash set
    cout << "The size of hash set is: " << hashset.size() << endl;
    // 6. iterate the hash set
    for (auto it = hashset.begin(); it != hashset.end(); ++it) {
        cout << (*it) << " ";
    }
    cout << "are in the hash set." << endl;
    // 7. clear the hash set
    hashset.clear();
    // 8. check if the hash set is empty
    if (hashset.empty()) {
        cout << "hash set is empty now!" << endl;
    }
}
```

# Tips：set 和 unordered_set有很大的区别！！！

c++ std中set与unordered_set区别和map与unordered_map区别类似：

- set基于红黑树实现，红黑树具有自动排序的功能，因此set内部所有的数据，在任何时候，都是有序的。**因此要求set内的数据类型要重载operator<**

- unordered_set基于哈希表，数据插入和查找的时间复杂度很低，几乎是常数时间，而代价是消耗比较多的内存，无自动排序功能。底层实现上，使用一个下标范围比较大的数组来存储元素，形成很多的桶，利用hash函数对key进行映射到不同区域进行保存。**因此要求unordered_set内数据类型要有hash函数**

详见https://blog.csdn.net/haluoluo211/article/details/82468061

## 217. 存在重复元素（简单）

1. 题目描述

   给定一个整数数组，判断是否存在重复元素。

   如果任何值在数组中出现至少两次，函数返回 true。如果数组中每个元素都不相同，则返回 false。

   **示例 1:**

   ```
   输入: [1,2,3,1]
   输出: true
   ```

   **示例 2:**

   ```
   输入: [1,2,3,4]
   输出: false
   ```

   **示例 3:**

   ```
   输入: [1,1,1,3,3,4,3,2,4,2]
   输出: true
   ```

2. 简单实现

```
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        unordered_set<int> s;
        for(int i = 0; i < nums.size(); i++){
            if(s.count(nums[i]) > 0)
                return true;
            s.insert(nums[i]);
        }
        return false;
    }
};
```

## 136.只出现一次的数字（简单）

1. 题目描述

   给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

**说明**：你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

**示例 1:**

```
输入: [2,2,1]
输出: 1
```

**示例 2:**

```
输入: [4,1,2,1,2]
输出: 4
```

2. 简单实现

使用哈希表可以很简单地实现，但**最优算法是使用异或运算**

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int ans = 0;
        for(int i = 0; i < nums.size(); i++)
            ans = ans ^ nums[i];
        return ans;
    }
};
```

## 349.两个数组的交集（简单）

1. 题目描述

给定两个数组，编写一个函数来计算它们的交集。

**示例 1:**

```
输入: nums1 = [1,2,2,1], nums2 = [2,2]
输出: [2]
```

**示例 2:**

```
输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出: [9,4]
```

**说明:**

- 输出结果中的每个元素一定是唯一的。
- 我们可以不考虑输出结果的顺序。

2. 简单实现

**需要注意的是数组内的元素可以是重复的**，所以需要两个hash表，其中一个hash表记录nums1的值，并对nums2的各值比对是否有重复，另一个hash表记录ans中是否有重复；

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        vector<int> ans;
        int len1 = nums1.size();
        if(len1 <= 0) return ans;
        int len2 = nums2.size();
        if(len2 <= 0) return ans;
        unordered_set<int> s1;
        unordered_set<int> s2;
        for(int i = 0; i < len1; i++)
            s1.insert(nums1[i]);
        for(int i = 0; i < len2; i++){
            if(s1.count(nums2[i]) > 0){
                if(s2.count(nums2[i]) <= 0){
                    ans.push_back(nums2[i]);
                    s2.insert(nums2[i]);
                }
            }
        }
        return ans;
    }
};
```

Tips：第二个hash表也可以省略，只要依次查找s1中的元素值是否出现在s2中即可直接push进ans，因为s1中元素不重复

# # 202.快乐数（简单）

1. 题目描述

   编写一个算法来判断一个数是不是“快乐数”。

   一个“快乐数”定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是无限循环但始终变不到 1。如果可以变为 1，那么这个数就是快乐数。

   **示例:**

```
输入: 19
输出: true
解释:
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1
```

2. 简单实现

   问题的关键在于如何跳出循环，通过举例观察发现，*非快乐数在计算的过程中**一定会**出现重复值，之后就会陷入无限循环中，因此当计算过程中出现重复值就停止

```
class Solution {
```

```
public:
    bool isHappy(int n) {
        unordered_set<int> s;
        while(n != 1){
            if(s.count(n) > 0) return false; //循环了
            s.insert(n);
            int temp = 0;
            while(n){
                temp += pow(n%10, 2);
                n /= 10;
            }
            n = temp;
        }
        return true;
    }
};
```

3. 最优解法：节省空间

使用“**快慢指针”思想**找出循环：“快指针”每次走两步，“慢指针”每次走一步，当二者相等时，即为一个循环周期。此时，判断是不是因为1引起的循环，是的话就是快乐数，否则不是快乐数。 **注意：**此题不建议用集合记录每次的计算结果来判断是否进入循环，因为这个**集合可能大到无法存储**；另外，也不建议使用递归，同理，如果递归层次较深，会直接导致调用栈崩溃。**不要因为这个题目给出的整数是int型而投机取巧！**

```
class Solution {
public:
    int bitSquareSum(int n) {
        int sum = 0;
        while(n > 0){
            int bit = n % 10;
            sum += bit * bit;
            n = n / 10;
        }
        return sum;
    }
    bool isHappy(int n) {
        int slow = n, fast = n;
        do{
            slow = bitSquareSum(slow);
            fast = bitSquareSum(fast);
            fast = bitSquareSum(fast);
        }while(slow != fast);

        return slow == 1;
    }
};
```

# 哈希映射的应用

# C++ STL unordered_map用法

```cpp
#include <unordered_map>                // 0. include the library

int main() {
    // 1. initialize a hash map
    unordered_map<int, int> hashmap;
    // 2. insert a new (key, value) pair
    hashmap.insert(make_pair(0, 0));
    hashmap.insert(make_pair(2, 3));
    // 3. insert a new (key, value) pair or update the value of existed key
    hashmap[1] = 1;
    hashmap[1] = 2;
    // 4. get the value of a specific key
    cout << "The value of key 1 is: " << hashmap[1] << endl;
    // 5. delete a key
    hashmap.erase(2);
    // 6. check if a key is in the hash map
    if (hashmap.count(2) <= 0) {
        cout << "Key 2 is not in the hash map." << endl;
    }
    // 7. get the size of the hash map
    cout << "the size of hash map is: " << hashmap.size() << endl;
    // 8. iterate the hash map
    for (auto it = hashmap.begin(); it != hashmap.end(); ++it) {
        cout << "(" << it->first << "," << it->second << ") ";
    }
    cout << "are in the hash map." << endl;
    // 9. clear the hash map
    hashmap.clear();
    // 10. check if the hash map is empty
    if (hashmap.empty()) {
        cout << "hash map is empty now!" << endl;
    }
}
```

## 1.两数之和（简单）

1. 题目描述

   给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

   你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

   **示例:**

   ```
   给定 nums = [2, 7, 11, 15], target = 9

   因为 nums[0] + nums[1] = 2 + 7 = 9
   所以返回 [0, 1]
   ```

2. 简单实现

   用哈希映射记录<值，下标>

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> m;
        for(int i = 0 ; i < nums.size(); i++){
            if(m.count(target - nums[i]) > 0)
                return {m[target-nums[i]], i};
            m[nums[i]] = i;
        }
        return vector<int>(2,-1);
    }
};
```

## 205.同构字符串（简单）

1. 题目描述

   给定两个字符串 **s** 和 **t**，判断它们是否是同构的。

   如果 **s** 中的字符可以被替换得到 **t** ，那么这两个字符串是同构的。

   所有出现的字符都必须用另一个字符替换，同时保留字符的顺序。两个字符不能映射到同一个字符上，但字符可以映射自己本身。

   **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
   ```

   **说明:** 你可以假设 ***s*** 和 **t** 具有相同的长度。

2. 简单实现

   用两个哈希映射检查一一映射关系

   ```
   class Solution {
   public:
       bool isIsomorphic(string s, string t) {
           unordered_map<char, char> f1;//s->t
           unordered_map<char, char> f2;//t->s
           for(int i = 0; i < s.size(); i++){
               if(f1.count(s[i]) > 0){
   ```

```
                if (f1[s[i]] != t[i])
                    return false;
            }
            if(f2.count(t[i]) > 0){
                if (f2[t[i]] != s[i])
                    return false;
            }
            f1[s[i]] = t[i];
            f2[t[i]] = s[i];
        }
        return true;
    }
};
```

3. 最优解法

实际上只要比较字符最近一次出现的位置是否相同即可，实际上也是比较第一次出现的位置，因为位置一旦不同会返回false

```
class Solution {
public:
    bool isIsomorphic(string s, string t) {
             int len = s.size();
            vector<int> arrs(256, 0), arrt(256, 0);//ascii码最大255
            for (int i = 0; i < len; i++) {
                if (arrs[s[i]] != arrt[t[i]])
                    return false;
                arrs[s[i]] = i+1;
                arrt[t[i]] = i+1;
            }
            return true;
    }
};
```

## 599.两个列表的最小索引总和（简单）

1. 题目描述

假设Andy和Doris想在晚餐时选择一家餐厅，并且他们都有一个表示最喜爱餐厅的列表，每个餐厅的名字用字符串表示。

你需要帮助他们用**最少的索引和**找出他们**共同喜爱的餐厅**。 如果答案不止一个，则输出所有答案并且不考虑顺序。 你可以假设总是存在一个答案。

**示例 1:**

```
输入:
["Shogun", "Tapioca Express", "Burger King", "KFC"]
["Piatti", "The Grill at Torrey Pines", "Hungry Hunter Steakhouse", "Shogun"]
输出: ["Shogun"]
解释: 他们唯一共同喜爱的餐厅是“Shogun”。
```

**示例 2:**

```
输入:
["Shogun", "Tapioca Express", "Burger King", "KFC"]
["KFC", "Shogun", "Burger King"]
输出: ["Shogun"]
解释: 他们共同喜爱且具有最小索引和的餐厅是“Shogun”，它有最小的索引和1(0+1)。
```

**提示:**

1. 两个列表的长度范围都在 [1, 1000]内。
2. 两个列表中的字符串的长度将在[1，30]的范围内。
3. 下标从0开始，到列表的长度减1。
4. 两个列表都没有重复的元素。

2. 简单实现

用哈希映射保存其中一个数组各餐厅的索引值，遍历另一个数组找索引值和的最小值

```
class Solution {
public:
    vector<string> findRestaurant(vector<string>& list1, vector<string>& list2) {
        vector<string> ans;
        unordered_map<string, int> m;
        for(int i = 0; i < list1.size(); i++)
            m[list1[i]] = i;
        int min = 5000;
        for(int i = 0; i < list2.size(); i++){
            if(m.count(list2[i]) > 0){
                int temp = i + m[list2[i]];
                if(temp < min){
                    ans.clear();
                    ans.push_back(list2[i]);
                    min = temp;
                }
                else if(temp == min)
                    ans.push_back(list2[i]);
            }
        }
        return ans;
    }
};
```

# 场景 II - 按键聚合

另一个常见的场景是 `按键聚合所有信息` 。我们也可以使用哈希映射来实现这一目标。

## 示例

这是一个例子：

> 给定一个字符串，找到它中的第一个非重复字符并返回它的索引。如果它不存在，则返回 -1。

解决此问题的一种简单方法是首先`计算每个字符的出现次数`。然后通过结果找出第一个与众不同的角色。

因此，我们可以维护一个哈希映射，其键是字符，而值是相应字符的计数器。每次迭代一个字符时，我们只需将相应的值加 1。

## 更重要的是

解决此类问题的关键是在`遇到现有键时确定策略`。

在上面的示例中，我们的策略是计算事件的数量。有时，我们可能会将所有值加起来。有时，我们可能会用最新的值替换原始值。策略取决于问题，实践将帮助您做出正确的决定。

## 387.字符串中的第一个唯一字符（简单）

1. 题目描述

   给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

   **案例:**

   ```
   s = "leetcode"
   返回 0.

   s = "loveleetcode",
   返回 2.
   ```

   **注意事项：**您可以假定该字符串只包含小写字母

2. 简单实现

   ```
   class Solution {
   public:
       int firstUniqChar(string s) {
           int count['z'+1] = {0};
           for(int i=0; i<s.length(); i++)
               count[s[i]]++;
           for(int i=0; i<s.length(); i++)
               if(count[s[i]]==1)
                   return i;
           return -1;
       }
   };
   ```

## 350两个数组的交集 II（简单）

1. 题目描述

给定两个数组，编写一个函数来计算它们的交集。

**示例 1:**

```
输入: nums1 = [1,2,2,1], nums2 = [2,2]
输出: [2,2]
```

**示例 2:**

```
输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出: [4,9]
```

**说明:**

- 输出结果中每个元素出现的次数，应与元素在两个数组中出现的次数一致。
- 我们可以不考虑输出结果的顺序。

**进阶:**

- 如果给定的数组已经排好序呢？你将如何优化你的算法？
- 如果 *nums1* 的大小比 *nums2* 小很多，哪种方法更优？
- 如果 *nums2* 的元素存储在磁盘上，磁盘内存是有限的，并且你不能一次加载所有的元素到内存中，你该怎么办？

2. 简单实现

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int, int> m;
        vector<int> ans;
        if(nums1.size() == 0 || nums2.size() == 0) return ans;
        //计数
        for(int i = 0; i < nums1.size(); i++){
            if(m.count(nums1[i]) <= 0)
                m[nums1[i]] = 1;
            else
                m[nums1[i]] += 1;
        }
        //查重
        for(int i = 0; i < nums2.size(); i++){
            if(m.count(nums2[i]) > 0 && m[nums2[i]] != 0){
                ans.push_back(nums2[i]);
                m[nums2[i]] -= 1;
            }
        }
        return ans;
    }
};
```

3. 最优解法

排序，然后就按下标比就好啦

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        vector<int> answer;
        int len1 = nums1.size();
        int len2 = nums2.size();
        sort(nums1.begin(), nums1.end());
        sort(nums2.begin(), nums2.end());
        int i=0, j=0;
        while(i<len1 && j<len2) {
            if(nums1[i] == nums2[j]) {
                answer.push_back(nums1[i]);
                i++;
                j++;
            }
            else {
                if(nums1[i] < nums2[j])
                    i++;
                else
                    j++;
            }
        }
        return answer;
    }
};
```

## 219.存在重复元素 II（简单）

1. 题目描述

   给定一个整数数组和一个整数 *k*，判断数组中是否存在两个不同的索引 *i* 和 *j*，使得 **nums [i] = nums [j]**，并且 *i* 和 *j* 的差的绝对值最大为 *k*。

   **示例 1:**

   ```
   输入: nums = [1,2,3,1], k = 3
   输出: true
   ```

   **示例 2:**

   ```
   输入: nums = [1,0,1,1], k = 1
   输出: true
   ```

   **示例 3:**

   ```
   输入: nums = [1,2,3,1,2,3], k = 2
   输出: false
   ```

2. 简单实现

   暴力解法为：暴力遍历，时间复杂度O(kn)

哈希映射法： unordered_map<int, int> m;记录每个值的最大坐标，每次有重复的就减，小于等于k就返回true，否则更新

```
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        if(k == 0) return false;
        unordered_map<int, int> m;
        for(int i = 0; i < nums.size(); i++){
            if(m.count(nums[i]) <= 0)
                m[nums[i]] = i;
            else{
                if(i - m[nums[i]] <= k)
                    return true;
                else
                     m[nums[i]] = i;
            }
        }
        return false;
    }
};
```

3. 其他解法

以k为滑窗大小，用set记录滑窗内是否有重复值

```
class Solution {
public:
    set<int> temp;

    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        for(int cnt=0;cnt<nums.size();cnt++) {
                if(temp.count(nums[cnt]))
                    return true;
                temp.insert(nums[cnt]);
            if(temp.size()>k)
                temp.erase(nums[cnt-k]);
        return false;
    }
};
```

# 设计键

在以前的问题中，键的选择相对简单。不幸的是，有时你必须考虑在使用哈希表时`设计合适的键`。

**示例**

我们来看一个例子：

> 给定一组字符串，将字母异位词组合在一起。

众所周知，哈希映射可以很好地按键分组信息。但是我们不能直接使用原始字符串作为键。我们必须设计一个合适的键来呈现字母异位词的类型。例如，有字符串 “eat” 和 “ate” 应该在同一组中。但是 “eat” 和 “act” 不应该组合在一起。

**解决方案**

---

实际上，`设计关键`是在原始信息和哈希映射使用的实际键之间`建立映射关系`。设计键时，需要保证：

> 1. 属于同一组的所有值都将映射到同一组中。
> 2. 需要分成不同组的值不会映射到同一组。

此过程类似于设计哈希函数，但这是一个本质区别。哈希函数满足第一个规则但`可能不满足第二个规则`。但是你的映射函数应该满足它们。

在上面的示例中，我们的映射策略可以是：对字符串进行排序并使用排序后的字符串作为键。也就是说，“eat” 和 “ate” 都将映射到 “aet”。

有时，设计映射策略可能是非常`棘手的`。我们将在本章为您提供一些练习，并在此之后给出总结。

## 49.字母异位词分组（中等）

---

1. 题目描述

   给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

   **示例:**

   ```
   输入: ["eat", "tea", "tan", "ate", "nat", "bat"],
   输出:
   [
     ["ate","eat","tea"],
     ["nat","tan"],
     ["bat"]
   ]
   ```

   **说明:**

   - 所有输入均为小写字母。
   - 不考虑答案输出的顺序。

2. 简单实现

   按照前面讲述的方法实现

   ```
   class Solution {
   public:
       vector<vector<string>> groupAnagrams(vector<string>& strs) {
           unordered_map<string, vector<string>> m;
           for(int i = 0; i < strs.size(); i++){
               string temp = strs[i];
               sort(temp.begin(), temp.end());
               if(m.count(temp) <= 0)
                   m[temp] = {strs[i]};
               else
   ```

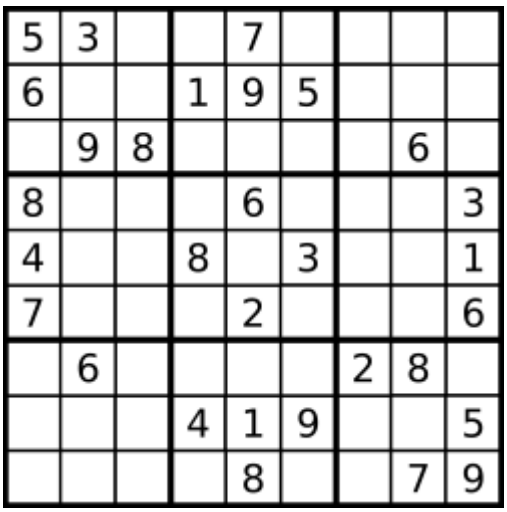
```
                m[temp].push_back(strs[i]);
            }
            vector<vector<string>> ans;
            for(unordered_map<string, vector<string>>::iterator it = m.begin(); it !=
 m.end(); it++)
                ans.push_back(it->second);
            return ans;
        }
 };
```

## 36.有效的数独（中等）

1. 题目描述

   判断一个 9x9 的数独是否有效。只需要**根据以下规则**，验证已经填入的数字是否有效即可。

   1. 数字 `1-9` 在每一行只能出现一次。
   2. 数字 `1-9` 在每一列只能出现一次。
   3. 数字 `1-9` 在每一个以粗实线分隔的 `3x3` 宫内只能出现一次。



   上图是一个部分填充的有效的数独。

   数独部分空格内已填入了数字，空白格用 `'.'` 表示。

   **示例 1:**

```
输入:
[
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".",".","2",".",".","6"],
  [".","6",".",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".",".","8",".",".","7","9"]
]
输出: true
```

**示例 2:**

```
输入:
[
  ["8","3",".",".","7",".",".",".",".","."],
  ["6",".",".","1","9","5",".",".",".","."],
  [".","9","8",".",".",".",".",".","6","."],
  ["8",".",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".",".","1"],
  ["7",".",".",".",".","2",".",".",".",".","6"],
  [".","6",".",".",".",".",".",".","2","8","."],
  [".",".",".",".","4","1","9",".",".",".","5"],
  [".",".",".",".",".",".","8",".",".",".","7","9"]
]
输出: false
解释: 除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。
      但由于位于左上角的 3x3 宫内有两个 8 存在，因此这个数独是无效的。
```

**说明:**

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 给定数独序列只包含数字 `1-9` 和字符 `'.'` 。
- 给定数独永远是 `9x9` 形式的。

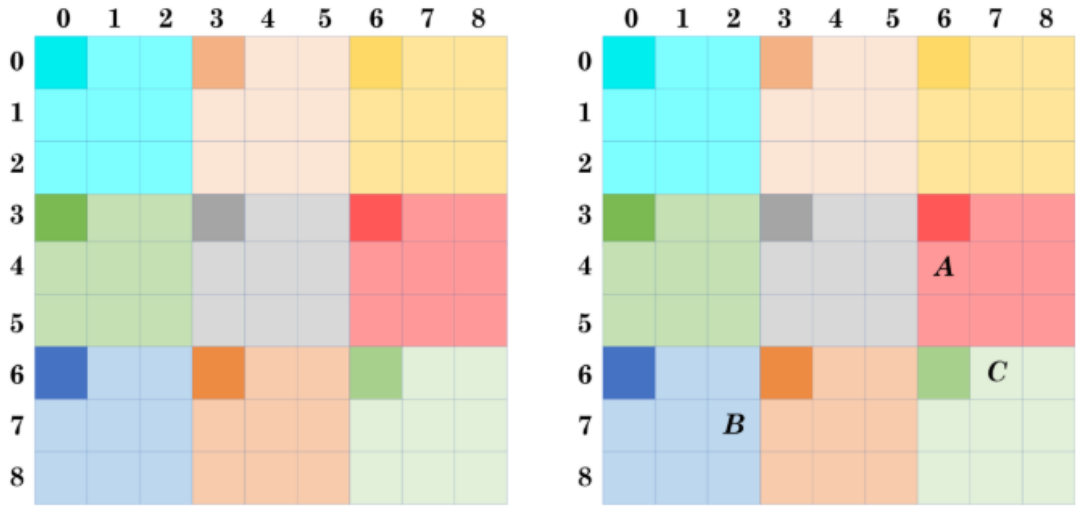
2. 简单实现

用哈希映射记录每个值所在的所有坐标位置，每出现相同值都判断是否和已有的有同行、同列或者同块

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        unordered_map<char,vector<pair<int,int>>> m;
        for(int i = 0; i < 9; i++){
            for(int j = 0; j < 9; j++){
                if(board[i][j] != '.'){
                    if(m.count(board[i][j]) <= 0)
                        m[board[i][j]] = {make_pair(i,j)};
                    else{
                        vector<pair<int,int>> locs = m[board[i][j]];
                        for(int k = 0; k < locs.size(); k++){
                            if(locs[k].first == i) return false;//同行
                            if(locs[k].second == j) return false;//同列
                            if((locs[k].first/3==i/3)&&(locs[k].second/3==j/3))
                                return false;//同块
                            m[board[i][j]].push_back(make_pair(i,j));
                        }
                    }
                }
            }
        }
        return true;
    }
};
```

3. 位运算法：

- 如何用坐标表示小宫内的格子？
  - 由于一共有$9$个小宫，每个小宫有$9$格，每次遍历时都会依次检查小宫里的每一格。相当于**固定一个小宫$i$，遍历小宫内的格子$j$。**
  - 如图所示，九个小宫用不同的颜色表示，每个小宫的左上角为小宫的第$0$个格子(深色标记)。
  - 我们设标号顺序为从左往右，从上往下。那么：
    - 格子$A$可以记为：第$5$宫，第$3$格($i=5, j=3$)，坐标为$(4,6)$
    - 格子$B$可以记为：第$6$宫，第$5$格($i=6, j=5$)，坐标为$(7,2)$
    - 格子$C$可以记为：第$8$宫，第$1$格($i=8, j=1$)，坐标为$(6,7)$



        - 综上所述，第$i$宫，第$j$格的坐标为：

$$(3*(i/3)+j/3, 3*(i\%3)+j\%3)$$

- 由嵌套循环的知识(**外层固定，内层遍历**)：
    - $borad[i][j]$表示了对固定的一行$i$，遍历列$j$(即每次检查一整行)
    - $board[j][i]$表示了对固定的一列$i$，遍历行$j$(即每次检查一整列)
    - $board[3*(i/3)+j/3][3*(i\%3)+j\%3]$表示了对固定的宫$i$，遍历格$j$(即每次检查一个宫)

## 3. 位运算

- 基本知识：
    - 与运算($a\&b$)：$a,b$均为$1$时，返回$1$，否则返回$0$
    - 异或运算($a\wedge b$)：$a,b$不同时为$0$或$1$时，返回$1$，否则返回$0$

- 本题可以使用一个$9$位二进制数判断数字是否被访问。第$k$位数为$1$代表已加入，为$0$代表未加入
- 更新方式(记九位数为$val$，传入的数字为$n$)：
    - 判断是否加入：将九位数右移位$n$位，与$1$进行与运算
        - 结果为$0$：未加入，将传入的数字加入九位数
        - 结果为$1$：已加入，返回 `false`
    - 将传入的数字加入九位数：将$1$左移位$n$位，与$val$异或即可

- 例子：对于数字$1010010000$，其第$4,7,9$位为$1$，表示当前$4,7,9$已经访问过了
    - 新来数字为$3$：
        - 将$1010010000$右移$3$位得到$1010010$，与$1$进行与运算，结果为$0$，未访问过。
        - 将$1$左移位$3$位得到$1000$，异或后得到$1010011000$
    - 新来数字为$4$：
        - 将$1010010000$右移$4$位得到$101001$，与$1$进行与运算，结果为$1$，访问过。
        - 返回 `false`

```java
public boolean isValidSudoku(char[][] board) {
    for(int i = 0; i < 9; i ++){
        // hori, veti, sqre分别表示行、列、小宫
        int hori = 0, veti = 0, sqre = 0;
        for(int j = 0; j < 9; j ++){
            // 由于传入为char，需要转换为int，减48
            int h = board[i][j] - 48;
            int v = board[j][i] - 48;
            int s = board[3 * (i / 3) + j / 3][3 * (i % 3) + j % 3] - 48;
            // "."的ASCII码为46，故小于0代表着当前符号位"."，不用讨论
            if(h > 0){
                hori = sodokuer(h, hori);
            }
            if(v > 0){
                veti = sodokuer(v, veti);
            }
            if(s > 0){
                sqre = sodokuer(s, sqre);
            }
            if(hori == -1 || veti == -1 || sqre == -1){
                return false;
            }
        }
    }
    return true;
}

private int sodokuer(int n, int val){
    return ((val >> n) & 1) == 1 ? -1 : val ^ (1 << n);
}
```

时间复杂度$O(1)$，空间复杂度$O(1)$。

## 652.寻找重复的子树（中等）

1. 题目描述

   给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意**一棵**的根结点即可。

   两棵树重复是指它们具有相同的结构以及相同的结点值。

   **示例 1：**

```
      1
    / \
   2   3
  /   / \
 4   2   4
    /
   4
```

下面是两个重复的子树：

```
   2
  /
 4
```

和

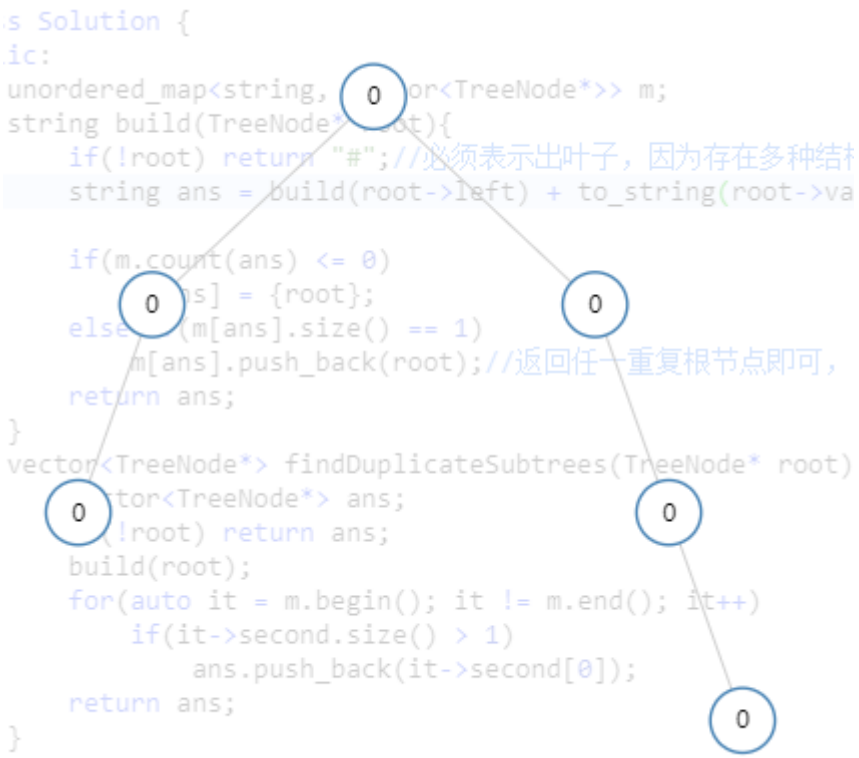
```
  4
```

因此，你需要以列表的形式返回上述重复子树的根结点。

2. 简单实现

用以节点为根的子树的后序遍历结果为键，该节点指针为值

- **必须显示示出叶子，因为存在多种树结构遍历结果相同的情况**
- **必须使用前序或后序遍历，不可使用中序遍历,因为中序遍历即便有#标识空指针也不可唯一恢复二叉树结构，如下图最左两个0和最右两个0的树的中序遍历结果一样**




```
class Solution {
public:
```

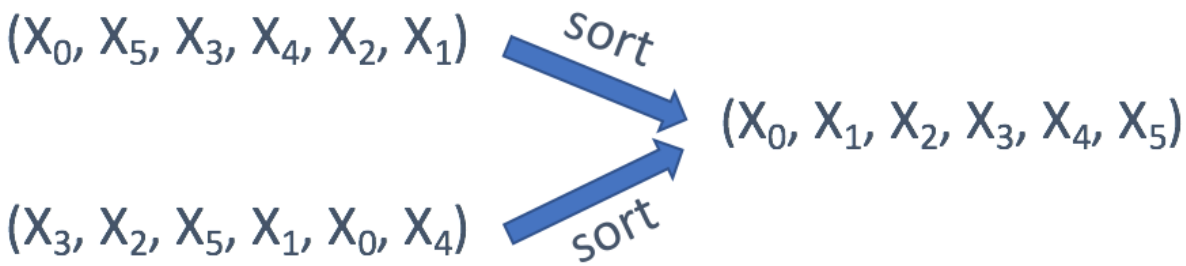
```
    unordered_map<string, vector<TreeNode*>> m;
    string build(TreeNode* root){
        if(!root) return "#";
        string ans = build(root->left) + to_string(root->val) + build(root->right);

        if(m.count(ans) <= 0)
            m[ans] = {root};
        else if(m[ans].size() == 1)
            m[ans].push_back(root);//返回任一重复根节点即可，因此只存两个以区分是否重复就可以了
        return ans;
    }
    vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        vector<TreeNode*> ans;
        if(!root) return ans;
        build(root);
        for(auto it = m.begin(); it != m.end(); it++)
            if(it->second.size() > 1)
                ans.push_back(it->second[0]);
        return ans;
    }
};
```

# TIP：血的教训，中序遍历要慎用，不可用来唯一表示树结构！！！！！

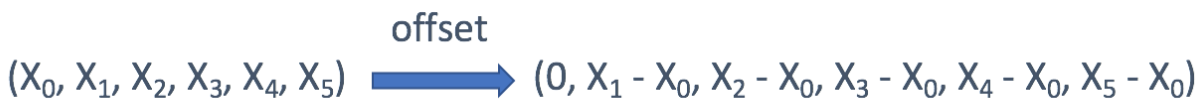
## 设计键 - 总结

这里有一些为你准备的关于如何设计键的建议。

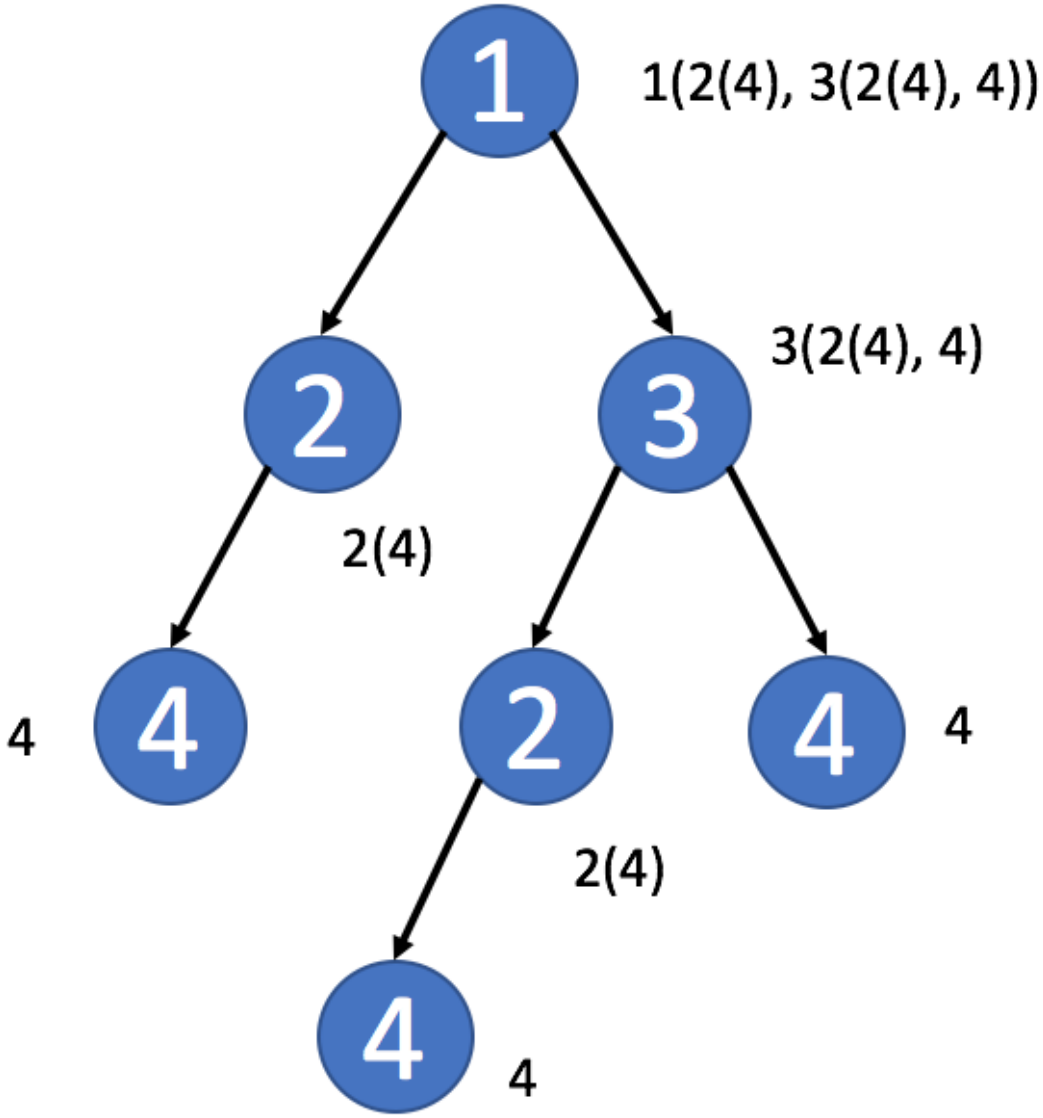
1. 当字符串 / 数组中每个元素的顺序不重要时，可以使用 `排序后的字符串 / 数组` 作为键。




2. 如果只关心每个值的偏移量，通常是第一个值的偏移量，则可以使用 `偏移量` 作为键。

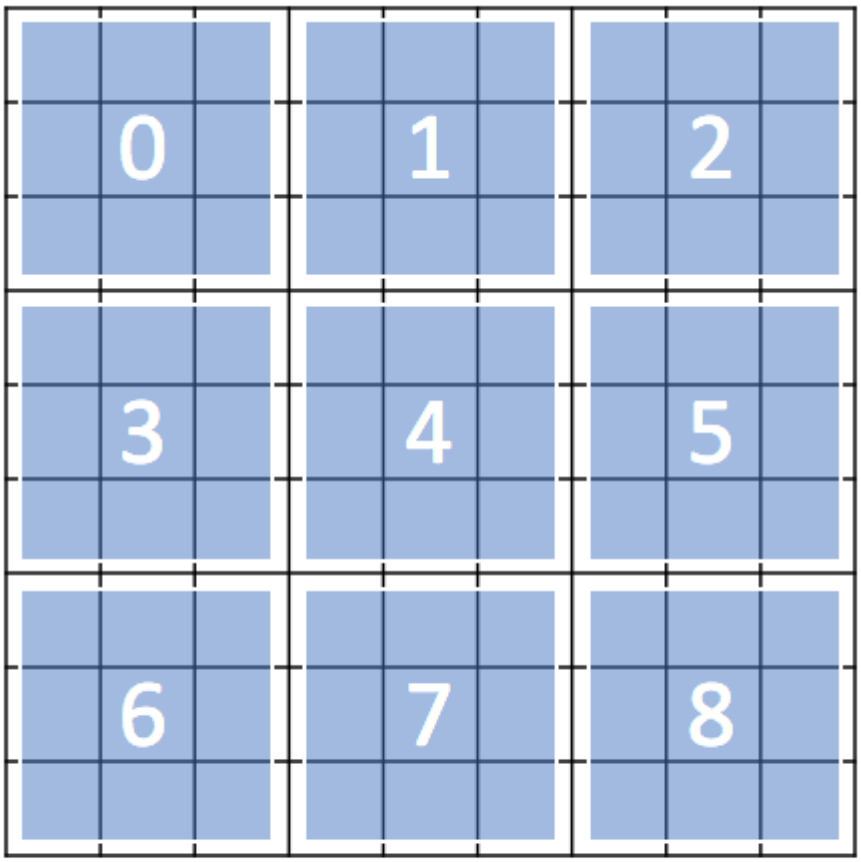



3. 在树中，你有时可能会希望直接使用 `TreeNode` 作为键。 但在大多数情况下，采用 `子树的序列化表述` 可能是一个更好的主意。



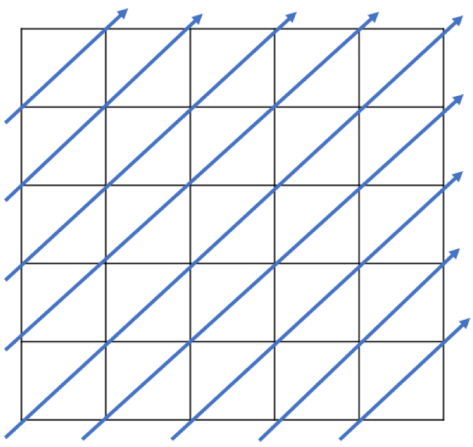

4. 在矩阵中，你可能希望使用 `行索引` 或 `列索引` 作为键。

5. 在数独中，可以将行索引和列索引组合来标识此元素属于哪个 `块` 。



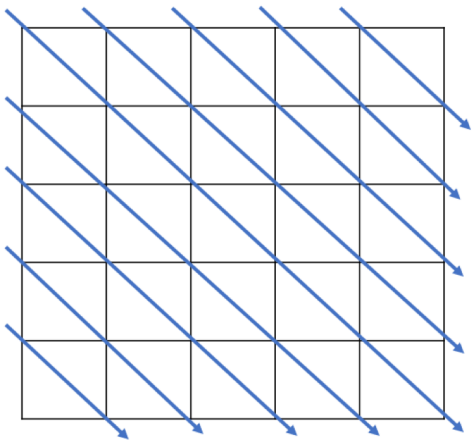

$$(i, j) \rightarrow (i / 3) * 3 + j / 3$$

6. 有时，在矩阵中，您可能希望将值聚合在 `同一对角线` 中。



Anti-Diagonal Order

$(i, j) \rightarrow i + j$



Diagonal Order

$(i, j) \rightarrow i - j$

# 小结

## 771.宝石与石头（简单）

1. 题目描述

   给定字符串 `J` 代表石头中宝石的类型，和字符串 `S` 代表你拥有的石头。 `S` 中每个字符代表了一种你拥有的石头的类型，你想知道你拥有的石头中有多少是宝石。

   `J` 中的字母不重复，`J` 和 `S` 中的所有字符都是字母。字母区分大小写，因此 `"a"` 和 `"A"` 是不同类型的石头。

   **示例 1:**

   ```
   输入: J = "aA", S = "aAAbbbb"
   输出: 3
   ```

   **示例 2:**

   ```
   输入: J = "z", S = "ZZ"
   输出: 0
   ```

   **注意:**

   - `S` 和 `J` 最多含有50个字母。
   - `J` 中的字符不重复。

2. 简单实现

   ```
   class Solution {
   public:
       int numJewelsInStones(string J, string S) {
           bool isjewel[256] = {false};
           for(int i = 0; i < J.size(); i++)
               isjewel[J[i]] = true;
           int ans = 0;
           for(int i = 0; i < S.size(); i++)
               if(isjewel[S[i]])
                   ans++;
           return ans;
       }
   };
   ```

## 3.无重复字符的最长子串（中等）

1. 题目描述

   给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

   **示例 1:**

```
输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

**示例 2:**

```
输入: "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

**示例 3:**

```
输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
     请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

2. 简单实现

滑窗法，哈希映射记录字符出现的位置

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> m;
        int ans = 0;
        int l = 0;//当前无重复字串起点
        int r = 0;//当前遍历点
        while(r < s.size()){
            if(m.count(s[r]) <= 0)
                m[s[r]] = r;
            else{//出现重复
                if(r - l > ans)//s[l,r)
                    ans = r - l;
                //l移至m[s[r]]+1
                while(l <= m[s[r]]){
                    m.erase(s[l]);
                    l++;
                }
                m[s[r]] = r;
            }
            r++;
        }
        if(r - l + 1> ans)//s[l,r]
            ans = r - l;
        return ans;
    }
};
```

3. 自我改进

以char为键的都可以改为用数组存储，这样比用哈希映射快

**Tips：int m[256] = {-1}的初始化结果为 -1, 0, 0, 0, ...，而不是-1, -1, ...**

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<int> m = vector<int>(256, -1);
        int ans = 0;
        int l = 0;
        int r = 0;
        while(r < s.size()){
            if(m[s[r]] < 0)
                m[s[r]] = r;
            else{//出现重复
                if(r - l > ans)//s[l,r)
                    ans = r - l;
                //l移至m[s[r]]+1
                while(l <= m[s[r]]){
                    m[s[l]] = -1;
                    l++;
                }
                m[s[r]] = r;
            }
            r++;
        }
        if(r - l + 1> ans)//s[l,r]
            ans = r - l;
        return ans;
    }
};
```

4. 最优解法

实际上l移至m[s[r]]+1的过程中无需改变m[s[l]]，因为r每遍历到一个地方时，只关心s[r]是否出现在s[l,r-1]中，而不关心l之前的字符——即只需记录每个字符的最近出现的位置即可

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<int> m = vector<int>(256, -1);
        int ans = 0;
        int l = -1;
        int r = 0;
        while(r < s.size()){
            l = max(l, m[s[r]]);
            m[s[r]] = r;
            ans = max(ans, r-l);
            r++;
        }
        return ans;
    }
};
```

# 454.四数相加 II（中等）

1. 题目描述

   给定四个包含整数的数组列表 A , B , C , D ,计算有多少个元组 `(i, j, k, l)` ，使得 `A[i] + B[j] + C[k] + D[l] = 0`。

   为了使问题简单化，所有的 A, B, C, D 具有相同的长度 N，且 0 ≤ N ≤ 500 。所有整数的范围在 -228 到 228 - 1 之间，最终结果不会超过 231 - 1 。

   **例如:**

   ```
   输入:
   A = [ 1, 2]
   B = [-2,-1]
   C = [-1, 2]
   D = [ 0, 2]

   输出:
   2

   解释:
   两个元组如下:
   1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
   2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0
   ```

2. 简单实现

   根据数据规模，最多只能用两层循环

   想起以前做两个数组（还是单个数组来着...）找和为目标数的题了，类推过来，记算A B数组内每种元组组合的和，统计每个和出现了几次，然后再遍历C D的元组组合的和，找互补结果

   以此类推，六个数组就是三层循环，n个数组需要n/2层循环

   ```
   class Solution {
   public:
       int fourSumCount(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>&
   D) {
           unordered_map<int, int> m;//<A、B某两元素和值，该值出现次数>
           int len = A.size();
           int ans = 0;
           for(int i = 0; i < len; i++)
               for(int j = 0; j < len; j++){
                   int sum = A[i] + B[j];
                   if(m.count(sum) <= 0)
                       m[sum] = 1;
                   else
                       m[sum]++;
               }
           for(int i = 0; i < len; i++)
               for(int j = 0; j < len; j++){
                   int aim = 0 - C[i] - D[j];
                   if(m.count(aim) > 0)
   ```

```
                    ans += m[aim];
                }
            return ans;
        }
};
```

## 347.前 K 个高频元素（中等）

1. 题目描述

   给定一个非空的整数数组，返回其中出现频率前 **k** 高的元素。

   **示例 1:**

   ```
   输入: nums = [1,1,1,2,2,3], k = 2
   输出: [1,2]
   ```

   **示例 2:**

   ```
   输入: nums = [1], k = 1
   输出: [1]
   ```

   **说明:**

   - 你可以假设给定的 *k* 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
   - 你的算法的时间复杂度**必须**优于 O(*n* log *n*) , *n* 是数组的大小。

2. 简单实现

   先用哈希映射统计出现次数，再用堆找出前k个，**注意要重载堆的排序函数**

```
class Solution {
public:
    struct cmp{
        bool operator()(pair<int, int>& a, pair<int, int>& b){ return a.second >
b.second; }
    };
    vector<int> topKFrequent(vector<int>& nums, int k) {
        vector<int> ans;
        unordered_map<int, int> m;
        for(int i = 0; i < nums.size(); i++)
            if(m.count(nums[i]) <= 0)
                m[nums[i]] = 1;
            else
                m[nums[i]]++;

        priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> m2;
        for(auto it = m.begin(); it != m.end(); it++){
            m2.push(make_pair(it->first, it->second));
            if (m2.size() > k)
                m2.pop();
        }
         while (!m2.empty()){
```

```
            ans.push_back(m2.top().first);
            m2.pop();
        }
        return ans;
    }
};
```

## 380.常数时间插入、删除和获取随机元素（中等）

1. 题目描述

设计一个支持在*平均*时间复杂度 **O(1)** 下，执行以下操作的数据结构。

1. `insert(val)`：当元素 val 不存在时，向集合中插入该项。
2. `remove(val)`：元素 val 存在时，从集合中移除该项。
3. `getRandom`：随机返回现有集合中的一项。每个元素应该有**相同的概率**被返回。

**示例：**

```
// 初始化一个空的集合。
RandomizedSet randomSet = new RandomizedSet();

// 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomSet.insert(1);

// 返回 false ，表示集合中不存在 2 。
randomSet.remove(2);

// 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomSet.insert(2);

// getRandom 应随机返回 1 或 2 。
randomSet.getRandom();

// 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
randomSet.remove(1);

// 2 已在集合中，所以返回 false 。
randomSet.insert(2);

// 由于 2 是集合中唯一的数字，getRandom 总是返回 2 。
randomSet.getRandom();
```

2. 简单实现

为了能O(1)实现getRandom()，使用动态数组nums保存所有值，用哈希映射存储各个值所在的Index

```
class RandomizedSet {
public:
    unordered_map<int, int> m;//val-index
    vector<int> nums;
    /** Initialize your data structure here. */
```

```
    RandomizedSet() {
    }
    /** Inserts a value to the set. Returns true if the set did not already contain
the specified element. */
    bool insert(int val) {
        if (m.count(val)) return false;
        nums.push_back(val);
        m[val] = nums.size() - 1;
        return true;
    }
    /** Removes a value from the set. Returns true if the set contained the
specified element. */
    bool remove(int val) {
        if (!m.count(val)) return false;
        int last = nums.back();
        m[last] = m[val];
        nums[m[last]] = last;
        nums.pop_back();
        m.erase(val);
        return true;
    }
    /** Get a random element from the set. */
    int getRandom() {
        return nums[rand() % nums.size()];
    }
};
```