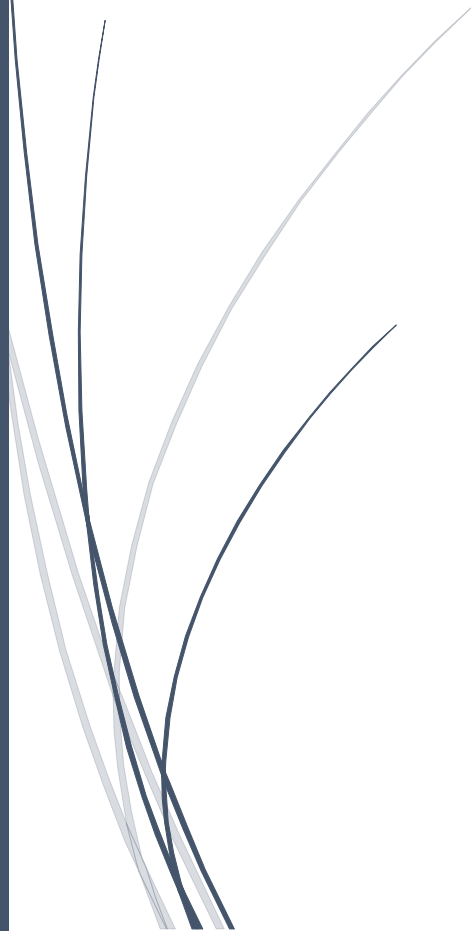
2020-5-2

数据结构与算法

LeetCode 刷题宝典 V1.0



yanpengma@163.com

华北电力大学（保定）马燕鹏

目录

序言

前段时间，在知识星球立了一个 Flag，今天 Flag 的进度为 100%，很是开心。为了大家学习的方便，所以整理了这份 150 多页的小册子。可以作为学习数据结构与算法或备考计算机类研究生的参考资料，希望对大家有所帮助。

所有内容来自以下的图文，并有所扩充：

- 技术图文：排序技术在求解算法题中的应用
- 技术图文：集合技术在求解算法题中的应用
- 技术图文：双指针技术在求解算法题中的应用
- 技术图文：字典技术在求解算法题中的应用
- 技术图文：位运算技术在求解算法题中的应用
- 技术图文：浅析 C# 语言中的扩展方法
- 技术图文：浅析 C# Dictionary 实现原理
- 技术图文：Python 位运算防坑指南
- 技术图文：有符号整型的数据范围为什么负数比正数多一个？

我是 终身学习者“**老马**”，一个长期践行“结伴式学习”理念的 **中年大叔。**

我崇尚分享，渴望成长，于 2010 年创立了“**LSGO 软件技术团队**”，并加入了国内著名的开源组织“**Datawhale**”，也是“**Dre@mtech**”、“**智能机器人研究中心**”和“**大数据与哲学社会科学实验室**”的一员。

愿我们一起学习，一起进步，相互陪伴，共同成长。

后台回复「搜搜搜」，随机获取电子资源！

欢迎关注，请扫描二维码：

1 双指针技术在求解算法题中的应用

1.1 C# 和 Python 中的链表结构

Python `list` 的源码地址：

https://github.com/python/cpython/blob/master/Include/listobject.h

https://github.com/python/cpython/blob/master/Objects/listobject.c

C# `List<T>` 的源码地址：

https://referencesource.microsoft.com/#mscorlib/system/collections/generic/list.cs,cf7f4095e4de7646

通过阅读源码，我们发现 Python 的 `list` 与 C# 的 `List<T>` 一致都是通过动态数组的方式来实现的。

Python 的内置结构中没有链表这种结构，而 C# 的内置结构中封装了双向链表 `LinkedList<T>`，内部结点为 `LinkedListNode<T>`，源码地址如下：

https://referencesource.microsoft.com/#System/compmod/system/collections/generic/linkedlist.cs,df5a6c7b6b60da4f

LinkedListNode<T>

- `public LinkedListNode<T> Next { get; }` -> 获取下一个节点

- `public LinkedListNode<T> Previous { get; }` -> 获取上一个节点
- `public T Value { get; set; }` -> 获取或设置包含在节点中的值。

LinkedList<T>

- `public LinkedListNode<T> AddFirst(T value);` -> 添加包含指定的值的开头的新节点
- `public LinkedListNode<T> AddLast(T value);` -> 添加包含指定的值的末尾的新节点
- `public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);` -> 添加包含在指定的现有节点前的指定的值的新节点
- `public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);` -> 添加包含指定的值中指定的现有节点后的新节点
- `public void AddFirst(LinkedListNode<T> node);` -> 将指定的新节点添加的开头
- `public void AddLast(LinkedListNode<T> node);` -> 将指定的新节点添加的末尾
- `public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode);` -> 在指定的现有节点之前添加指定的新节点
- `public void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode);` -> 在指定的现有节点之后添加指定的新节点
- `public bool Remove(T value);` -> 移除从指定的值的第一个匹配项
- `public void Remove(LinkedListNode<T> node);` -> 移除指定的节点
- `public void RemoveFirst();` -> 删除的开始处的节点
- `public void RemoveLast();` -> 删除节点的末尾

- `public LinkedListNode<T> Find(T value);` -> 查找包含指定的值的第一个节点。
- `public LinkedListNode<T> FindLast(T value);` -> 查找包含指定的值的最后一个节点。
- `public void Clear();` -> 删除所有节点
- `public int Count { get; }` -> 获取中实际包含的节点数
- `public LinkedListNode<T> First { get; }` -> 获取的第一个节点
- `public LinkedListNode<T> Last { get; }` -> 获取的最后一个节点

```
public static void LinkedListSample()
{
    LinkedList<int> lst = new LinkedList<int>();
    lst.AddFirst(3);
    lst.AddLast(1);
    lst.AddLast(4);
    foreach (int item in lst)
    {
        Console.Write(item+" ");
    }
    Console.WriteLine();

    LinkedListNode<int> cur = lst.Find(3);
    lst.AddBefore(cur, 2);
    foreach (int item in lst)
    {
        Console.Write(item + " ");
    }
    Console.WriteLine();

    lst.Remove(3);
    foreach (int item in lst)
    {
        Console.Write(item + " ");
    }
    Console.WriteLine();
    lst.Clear();
}
```

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

1.2 反转链表

- 题号：206
- 难度：简单
- https://leetcode-cn.com/problems/reverse-linked-list/

反转一个单链表。

示例:

```
输入：1->2->3->4->5->NULL
输出：5->4->3->2->1->NULL
```

进阶:

你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

思路：利用双指针的方式

p1 作为前面的指针探路，p2 作为后面的指针跟进，顺着链表跑一圈，搞定问题。

C# 语言

- 状态：通过
- 27 / 27 个通过测试用例
- 执行用时: 116 ms, 在所有 C# 提交中击败了 97.50% 的用户
- 内存消耗: 23.3 MB, 在所有 C# 提交中击败了 5.26% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
```

```
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode ReverseList(ListNode head)
    {
        if (head == null || head.next == null)
            return head;

        ListNode p1 = head;
        ListNode p2 = null;
        while (p1 != null)
        {
            ListNode temp = p1.next;
            p1.next = p2;
            p2 = p1;
            p1 = temp;
        }
        return p2;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：36 ms，在所有 Python3 提交中击败了 92.27% 的用户
- 内存消耗：14.6 MB，在所有 Python3 提交中击败了 17.65% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if head is None or head.next is None:
```

```
            return head
        p1 = head
        p2 = None
        while p1 is not None:
            temp = p1.next
            p1.next = p2
            p2 = p1
            p1 = temp
        return p2
```

1.3 删除链表的倒数第 N 个节点

- 题号：19
- 难度：中等
- https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

```
给定一个链表：1->2->3->4->5，和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

说明：

给定的 n 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

思路：利用双指针的方式

使用两个指针，前面的指针 p1 先走 n 步，接着让后面的指针 p2 与 p1 同步走，p1 走到终点，p2 即走到要移除的结点位置。

C# 语言

- 执行结果：通过
- 执行用时：104 ms，在所有 C# 提交中击败了 86.93% 的用户

- 内存消耗：24.6 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */
public class Solution
{
    public ListNode RemoveNthFromEnd(ListNode head, int n)
    {
        ListNode p1 = head;
        ListNode p2 = head;

        while (n > 0)
        {
            p1 = p1.next;
            n--;
        }

        if (p1 == null) //移除头结点
        {
            return head.next;
        }

        while (p1.next != null)
        {
            p1 = p1.next;
            p2 = p2.next;
        }

        p2.next = p2.next.next;
        return head;
    }
}
```

Python 语言

- 执行结果：通过

- 执行用时：48 ms, 在所有 Python3 提交中击败了 23.58% 的用户
- 内存消耗：13.5 MB, 在所有 Python3 提交中击败了 7.83% 的用户

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        p2 = head
        p1 = head
        while (n > 0):
            p1 = p1.next
            n -= 1

        if (p1 is None):  # 移除头结点
            return head.next

        while (p1.next):
            p2 = p2.next
            p1 = p1.next

        p2.next = p2.next.next
        return head
```

1.4 删除排序链表中的重复元素

- 题号：83
- 难度：简单
- https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

```
输入：1->1->2
输出：1->2
```

示例 2:

```
输入：1->1->2->3->3
输出：1->2->3
```

思路：利用双指针的方式

`p1` 作为前面的指针探路，`p2` 作为后面的指针跟进，如果遇到重复元素，`p2.next` 跳过去，`p1` 跑完整个链表所有重复元素都被摘下来。

C# 语言

- 执行结果：通过
- 执行用时：160 ms, 在所有 C# 提交中击败了 5.23% 的用户
- 内存消耗：25.9 MB, 在所有 C# 提交中击败了 5.72% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
```

```
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode DeleteDuplicates(ListNode head)
    {
        if (head == null)
            return head;

        ListNode p1 = head.next;
        ListNode p2 = head;
        while (p1 != null)
        {
            if (p1.val == p2.val)
                p2.next = p1.next;
            else
                p2 = p2.next;
            p1 = p1.next;
        }
        return head;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：52 ms, 在所有 Python3 提交中击败了 33.88% 的用户
- 内存消耗：13.5 MB, 在所有 Python3 提交中击败了 12.75% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        if head is None:
```

```
            return head

        p1 = head.next
        p2 = head
        while p1 is not None:
            if p1.val == p2.val:
                p2.next = p1.next
            else:
                p2 = p2.next
            p1 = p1.next
        return head
```

1.5 环形链表

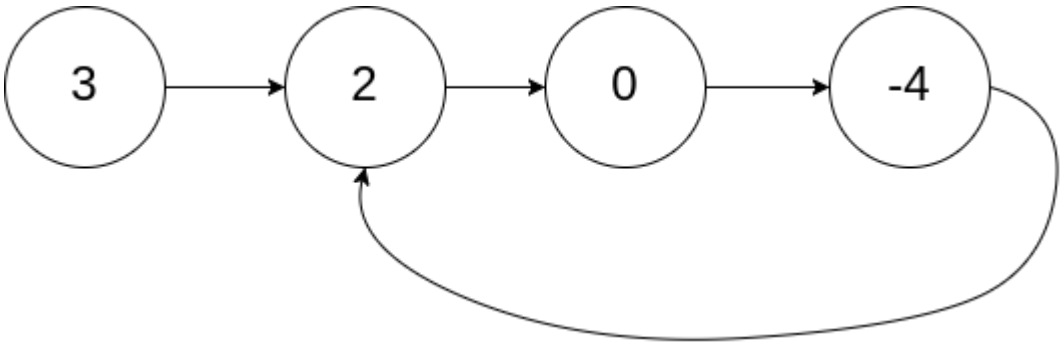
- 题号：141
- 难度：简单
- https://leetcode-cn.com/problems/linked-list-cycle/

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

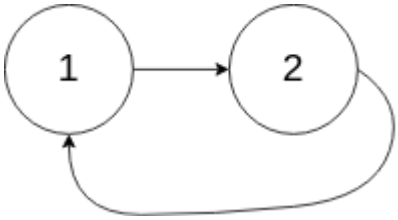
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3:

```
输入：head = [1], pos = -1
输出：false
解释：链表中没有环。
```

进阶:

你能用 O(1)（即，常量）内存解决此问题吗?

思路：利用双指针的方式

通常情况下，判断是否包含了重复的元素，我们使用 Hash 的方式来做。对于单链表的这种场景，我们也可以使用双指针的方式。

第一个指针 p1 每次移动两个节点，第二个指针 p2 每次移动一个节点，如果该链表存在环的话，第一个指针一定会再次碰到第二个指针，反之，则不存在环。

比如：head = [1,2,3,4,5]，奇数

```
p1：1   3   5   2   4   1
p2：1   2   3   4   5   1
```

比如：head = [1,2,3,4]，偶数

```
p1：1   3   1   3   1
p2：1   2   3   4   1
```

C# 语言

- 状态：通过
- 执行用时: 112 ms, 在所有 C# 提交中击败了 98.43% 的用户
- 内存消耗: 24.9 MB, 在所有 C# 提交中击败了 5.13% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public bool HasCycle(ListNode head) {
        ListNode p1 = head;
        ListNode p2 = head;

        while (p1 != null && p1.next != null)
        {
            p1 = p1.next.next;
            p2 = p2.next;
            if (p1 == p2)
                return true;
        }
        return false;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：56 ms, 在所有 Python3 提交中击败了 60.97% 的用户
- 内存消耗：16.6 MB, 在所有 Python3 提交中击败了 11.81% 的用户

```
# Definition for singly-linked list.
```

```python
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        p1 = head
        p2 = head
        while p1 is not None and p1.next is not None:
            p1 = p1.next.next
            p2 = p2.next
            if p1 == p2:
                return True
        return False
```

1.6 排序链表

- 题号：148
- 难度：中等
- https://leetcode-cn.com/problems/sort-list/

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

```
输入：4->2->1->3
输出：1->2->3->4
```

示例 2:

```
输入：-1->5->3->4->0
输出：-1->0->3->4->5
```

思路：模仿并归排序的思路，典型的回溯算法。

如果待排的元素存储在数组中，我们可以用并归排序。而这些元素存储在链表中，我们无法直接利用并归排序，只能借鉴并归排序的思想对算法进行修改。

并归排序的思想是将待排序列进行分组，直到包含一个元素为止，然后回溯合并两个有序序列，最后得到排序序列。

对于链表我们可以递归地将当前链表分为两段，然后 merge，分两段的方法是使用双指针法，`p1` 指针每次走两步，`p2` 指针每次走一步，直到 `p1` 走到末尾，这时 `p2` 所在位置就是中间位置，这样就分成了两段。

C# 语言

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 124 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 29 MB, 在所有 C# 提交中击败了 25.00% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode SortList(ListNode head)
    {
        if (head == null)
            return null;
        return MergeSort(head);
    }

    private ListNode MergeSort(ListNode node)
    {
        if (node.next == null)
        {
            return node;
        }
        ListNode p1 = node;
        ListNode p2 = node;
        ListNode cut = null;
        while (p1 != null && p1.next != null)
        {
            cut = p2;
            p2 = p2.next;
            p1 = p1.next.next;
        }
        cut.next = null;
        ListNode l1 = MergeSort(node);
```

```
        ListNode l2 = MergeSort(p2);
        return MergeTwoLists(l1, l2);
    }

    private ListNode MergeTwoLists(ListNode l1, ListNode l2)
    {
        ListNode pHead = new ListNode(-1);
        ListNode temp = pHead;

        while (l1 != null && l2 != null)
        {
            if (l1.val < l2.val)
            {
                temp.next = l1;
                l1 = l1.next;
            }
            else
            {
                temp.next = l2;
                l2 = l2.next;
            }
            temp = temp.next;
        }

        if (l1 != null)
            temp.next = l1;

        if (l2 != null)
            temp.next = l2;

        return pHead.next;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：216 ms, 在所有 Python3 提交中击败了 75.99% 的用户
- 内存消耗：20.7 MB, 在所有 Python3 提交中击败了 28.57% 的用户

```
# Definition for singly-linked list.
```

```python
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        if head is None:
            return head
        return self.mergeSort(head)

    def mergeSort(self, node: ListNode) -> ListNode:
        if node.next is None:
            return node
        p1 = node
        p2 = node
        cute = None
        while p1 is not None and p1.next is not None:
            cute = p2
            p2 = p2.next
            p1 = p1.next.next
        cute.next = None
        l1 = self.mergeSort(node)
        l2 = self.mergeSort(p2)
        return self.mergeTwoLists(l1, l2)

    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        pHead = ListNode(-1)
        temp = pHead
        while l1 is not None and l2 is not None:
            if l1.val < l2.val:
                temp.next = l1
                l1 = l1.next
            else:
                temp.next = l2
                l2 = l2.next
            temp = temp.next

        if l1 is not None:
            temp.next = l1
        if l2 is not None:
            temp.next = l2

        return pHead.next
```

2 集合技术在求解算法题中的应用

2.1 C# 和 Python 中的集合结构

集合技术在解题中主要用于处理有数据重复出现的问题。

HashSet<T>

C# 语言中 `HashSet<T>` 是包含不重复项的无序列表，称为“集合(set)”。由于 `set` 是一个保留字，所以用 `HashSet` 来表示。

源码：

https://referencesource.microsoft.com/#System.Core/System/Collections/Generic/HashSet.cs,2d265edc718b158b

HashSet 的成员方法

- `public HashSet();` -> 构造函数
- `public HashSet(IEnumerable<T> collection);` -> 构造函数
- `public int Count { get; }` -> 获取集合中包含的元素数。
- `public bool Add(T item);` -> 将指定的元素添加到集合中。
- `public bool Remove(T item);` -> 从集合中移除指定元素。
- `public void Clear();` -> 从集合中移除所有元素。
- `public bool Contains(T item);` -> 确定集合中是否包含指定的元素。
- `public void UnionWith(IEnumerable<T> other);` -> 并集

- `public void IntersectWith(IEnumerable<T> other);` -> 交集
- `public void ExceptWith(IEnumerable<T> other);` -> 差集
- `public bool IsSubsetOf(IEnumerable<T> other);` -> 确定当前集合是否为指定集合的子集。
- `public bool IsProperSubsetOf(IEnumerable<T> other);` -> 确定当前集合是否为指定集合的真子集。
- `public bool IsSupersetOf(IEnumerable<T> other);` -> 确定当前集合是否为指定集合的超集。
- `public bool IsProperSupersetOf(IEnumerable<T> other);` -> 确定当前集合是否为指定集合的真超集。
- `public bool Overlaps(IEnumerable<T> other);` -> 确定是否当前集合和指定的集合共享通用元素。
- `public bool SetEquals(IEnumerable<T> other);` -> 确定是否当前集合和指定集合包含相同的元素。

set

Python 中 `set` 与 `dict` 类似，也是一组 `key` 的集合，但不存储 `value`。由于 `key` 不能重复，所以，在 `set` 中，没有重复的 `key`。

注意，`key` 为不可变类型，即可哈希的值。

```
num = {}
print(type(num))  # <class 'dict'>
num = {1, 2, 3, 4}
print(type(num))  # <class 'set'>
```

集合的创建

- 先创建对象再加入元素。
- 在创建空集合的时候只能使用 `s = set()`，因为 `s = {}`创建的是空字典。

```
basket = set()
basket.add('apple')
basket.add('banana')
print(basket)  # {'banana', 'apple'}
```

- 直接把一堆元素用花括号括起来`{元素 1，元素 2，...，元素 n}`。
- 重复元素在 `set` 中会被自动被过滤。

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)  # {'banana', 'apple', 'pear', 'orange'}
```

- 使用 `set(value)`工厂函数，把列表或元组转换成集合。

```
a = set('abracadabra')
print(a)
# {'r', 'b', 'd', 'c', 'a'}

b = set(("Google", "Lsgogroup", "Taobao", "Taobao"))
print(b)
# {'Taobao', 'Lsgogroup', 'Google'}

c = set(["Google", "Lsgogroup", "Taobao", "Google"])
print(c)
# {'Taobao', 'Lsgogroup', 'Google'}
```

- 去掉列表中重复的元素

```
lst = [0, 1, 2, 3, 4, 5, 5, 3, 1]

temp = []
for item in lst:
    if item not in temp:
        temp.append(item)

print(temp)  # [0, 1, 2, 3, 4, 5]
```

```
a = set(lst)
print(list(a))  # [0, 1, 2, 3, 4, 5]
```

从结果发现集合的两个特点：无序（unordered）和唯一（unique）。

由于 set 存储的是无序集合，所以我们不可以为集合创建索引或执行切片(slice)操作，也没有键(keys)可用来获取集合中元素的值，但是可以判断一个元素是否在集合中。

访问集合中的值

- 可以使用 len()内建函数得到集合的大小。

```
thisset = set(['Google', 'Baidu', 'Taobao'])
print(len(thisset))  # 3
```

- 可以使用 for 把集合中的数据一个个读取出来。

```
thisset = set(['Google', 'Baidu', 'Taobao'])
for item in thisset:
    print(item)

# Baidu
# Google
# Taobao
```

- 可以通过 in 或 not in 判断一个元素是否在集合中已经存在

```
thisset = set(['Google', 'Baidu', 'Taobao'])
print('Taobao' in thisset)  # True
print('Facebook' not in thisset)  # True
```

集合的内置方法

- set.add(elmnt) -> 给集合添加元素，如果添加的元素在集合中已存在，则不执行任何操作。

- `set.update(set)` -> 修改当前集合，可以添加新的元素或集合到当前集合中，如果添加的元素在集合中已存在，则该元素只会出现一次，重复的会忽略。
- `set.remove(item)` -> 移除集合中的指定元素。如果元素不存在，则会发生错误。
- `set.discard(value)` -> 移除指定的集合元素。`remove()` 方法在移除一个不存在的元素时会发生错误，而 `discard()` 方法不会。
- `set.pop()` -> 随机移除一个元素。
- `set.intersection(set1, set2 ...)` -> 返回两个集合的交集。
- `set1 & set2` 返回两个集合的交集。
- `set.intersection_update(set1, set2 ...)` -> 交集，在原始的集合上移除不重叠的元素。
- `set.union(set1, set2...)` -> 返回两个集合的并集。
- `set1 | set2` -> 返回两个集合的并集。
- `set.difference(set)` -> 返回集合的差集。
- `set1 - set2` -> 返回集合的差集。
- `set.difference_update(set)` -> 集合的差集，直接在原来的集合中移除元素，没有返回值。
- `set.symmetric_difference(set)` -> 返回集合的异或。
- `set1 ^ set2` -> 返回集合的异或。
- `set.symmetric_difference_update(set)` -> 移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。
- `set.issubset(set)` -> 判断集合是不是被其他集合包含，如果是则返回 True，否则返回 False。

- `set1 <= set2` -> 判断集合是不是被其他集合包含，如果是则返回 True，否则返回 False。
- `set.issuperset(set)` -> 判断集合是不是包含其他集合，如果是则返回 True，否则返回 False。
- `set1 >= set2` -> 判断集合是不是包含其他集合，如果是则返回 True，否则返回 False。
- `set.isdisjoint(set)` -> 判断两个集合是不是不相交，如果是返回 True，否则返回 False。

frozenset

Python 提供了不能改变元素的集合的实现版本，即不能增加或删除元素，类型名叫 `frozenset`。需要注意的是 `frozenset` 仍然可以进行集合操作，只是不能用带有 `update` 的方法。

- `frozenset([iterable])` -> 返回一个冻结的集合，冻结后集合不能再添加或删除任何元素。

2.2 两个数组的交集

- 题号：349
- 难度：简单
- https://leetcode-cn.com/problems/intersection-of-two-arrays/

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。

思路：直接利用集合这种结构

C# 语言

- 执行结果：通过
- 执行用时：276 ms, 在所有 C# 提交中击败了 96.33% 的用户
- 内存消耗：31.5 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int[] Intersection(int[] nums1, int[] nums2)
    {
        HashSet<int> h1 = new HashSet<int>(nums1);
        HashSet<int> h2 = new HashSet<int>(nums2);
        return h1.Intersect(h2).ToArray();
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：60 ms, 在所有 Python3 提交中击败了 64.11% 的用户
- 内存消耗：13.8 MB, 在所有 Python3 提交中击败了 20.00% 的用户

```
class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        h1 = set(nums1)
        h2 = set(nums2)
        return list(h1.intersection(h2))
```

2.3 存在重复元素

- 题号：217
- 难度：简单
- https://leetcode-cn.com/problems/contains-duplicate/

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
```

思路：通过集合的方法

C# 语言

- 状态：通过
- 18 / 18 个通过测试用例

- 执行用时: 156 ms, 在所有 C# 提交中击败了 93.33% 的用户
- 内存消耗: 30.3 MB, 在所有 C# 提交中击败了 5.31% 的用户

```
public class Solution
{
    public bool ContainsDuplicate(int[] nums)
    {
        if (nums.Length < 2)
            return false;

        HashSet<int> h = new HashSet<int>();
        foreach (int num in nums)
        {
            if (h.Contains(num))
                return true;
            h.Add(num);
        }
        return false;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：48 ms, 在所有 Python3 提交中击败了 78.11% 的用户
- 内存消耗：18.9 MB, 在所有 Python3 提交中击败了 24.00% 的用户

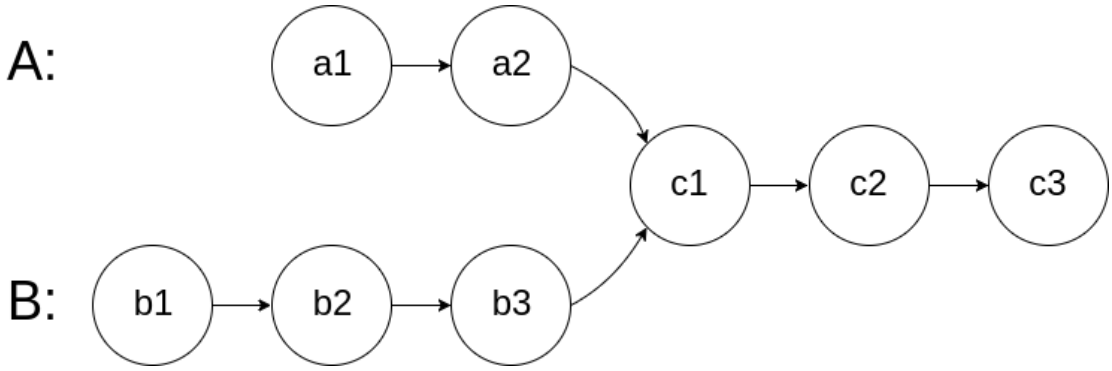
```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        if len(nums) < 2:
            return False

        h = set()
        for num in nums:
            if num in h:
                return True
            h.add(num)
        return False
```

2.4 相交链表

- 题号：160
- 难度：简单
- https://leetcode-cn.com/problems/intersection-of-two-linked-lists/

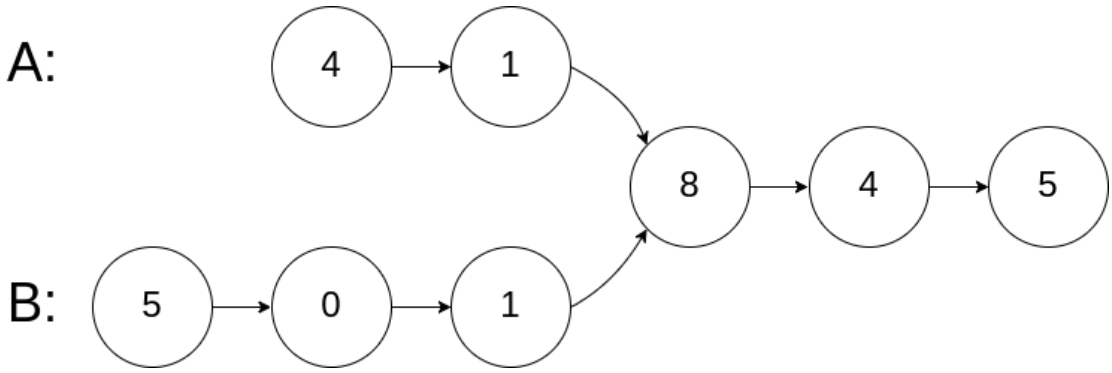
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

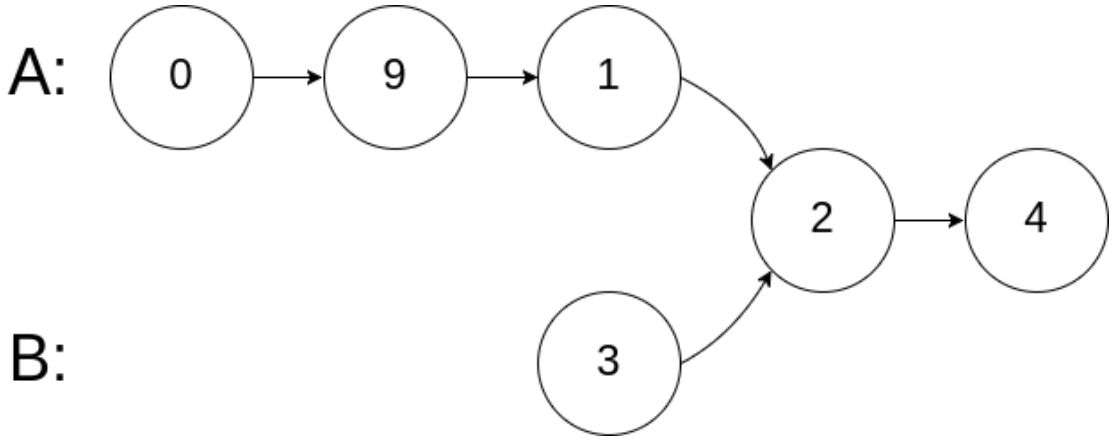
示例 1：




```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
```

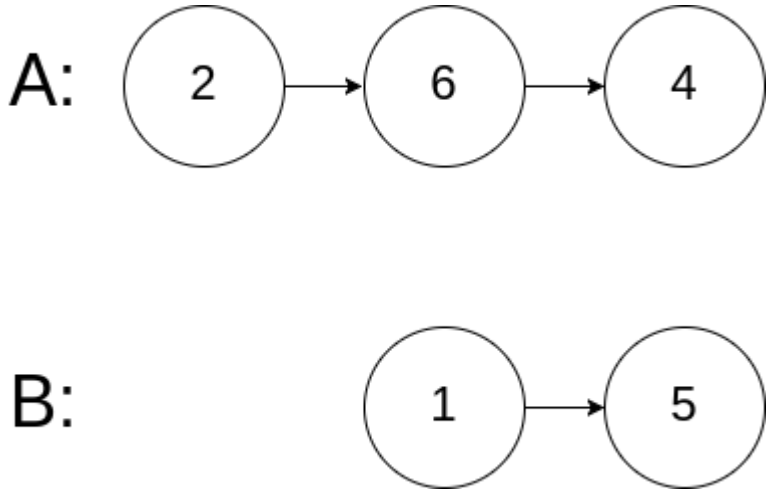
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。
在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：



输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：相交节点的值为 2 （注意，如果两个列表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。
在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：



输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。
由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释：这两个链表不相交，因此返回 null。

注意：

- 如果两个链表没有交点，返回 null.
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足 O(n) 时间复杂度，且仅用 O(1) 内存。

思路：通过集合的方法

C# 语言

- 状态：通过
- 45 / 45 个通过测试用例
- 执行用时: 172 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 37.6 MB, 在所有 C# 提交中击败了 5.88% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) { val = x; }
 * }
 */

public class Solution
{
    public ListNode GetIntersectionNode(ListNode headA, ListNode headB)
    {
        HashSet<ListNode> hash = new HashSet<ListNode>();
        ListNode temp = headA;
        while (temp != null)
        {
```

```
            hash.Add(temp);
            temp = temp.next;
        }
        temp = headB;
        while (temp != null)
        {
            if (hash.Contains(temp))
                return temp;
            temp = temp.next;
        }
        return null;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：200 ms, 在所有 Python3 提交中击败了 40.19% 的用户
- 内存消耗：29.4 MB, 在所有 Python3 提交中击败了 5.00% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> None:
        h = set()
        temp = headA
        while temp is not None:
            h.add(temp)
            temp = temp.next
        temp = headB
        while temp is not None:
            if temp in h:
                return temp
            temp = temp.next
        return None
```

2.5 环形链表

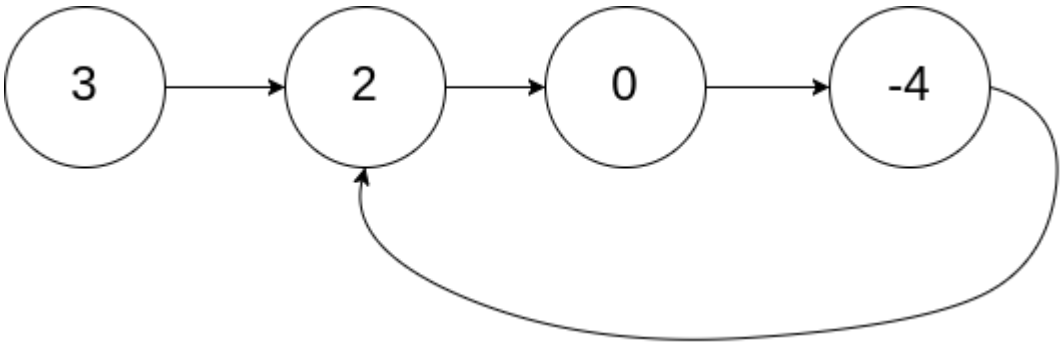
- 题号：141
- 难度：简单
- https://leetcode-cn.com/problems/linked-list-cycle/

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

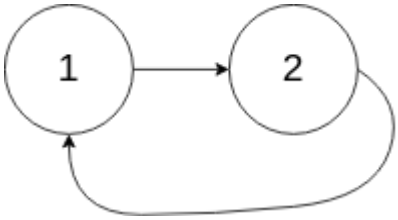
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3:

```
输入：head = [1], pos = -1
输出：false
解释：链表中没有环。
```

进阶:

你能用 O(1)（即，常量）内存解决此问题吗?

思路：通过集合的方法

通过检查一个结点此前是否被访问过来判断链表是否为环形链表。

C# 语言

- 状态：通过
- 执行用时：112 ms, 在所有 C# 提交中击败了 84.04% 的用户
- 内存消耗：26.5 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
```

```
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public bool HasCycle(ListNode head)
    {
        HashSet<ListNode> h = new HashSet<ListNode>();
        ListNode temp = head;
        while (temp != null)
        {
            if (h.Contains(temp))
                return true;

            h.Add(temp);
            temp = temp.next;
        }
        return false;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：60 ms, 在所有 Python3 提交中击败了 64.49% 的用户
- 内存消耗：17.3 MB, 在所有 Python3 提交中击败了 9.52% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        h = set()
        temp = head
        while temp is not None:
            if temp in h:
                return True
```

```
            h.add(temp)
            temp = temp.next
        return False
```

2.6 环形链表 II

- 题号：142
- 难度：中等
- https://leetcode-cn.com/problems/linked-list-cycle-ii/

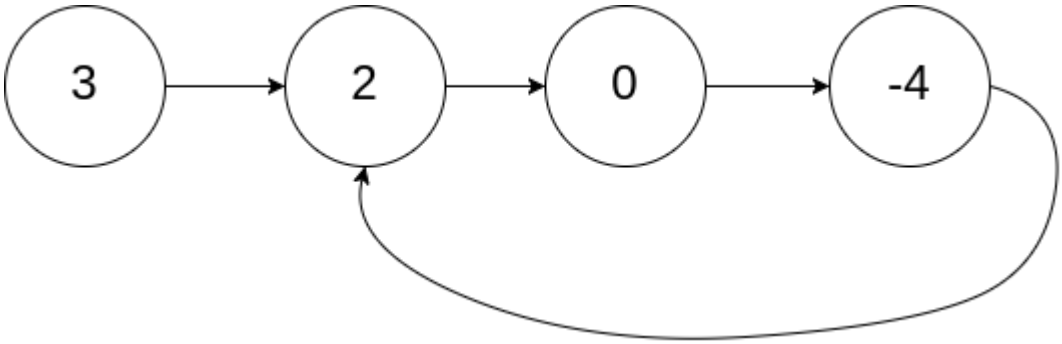
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

说明：不允许修改给定的链表。

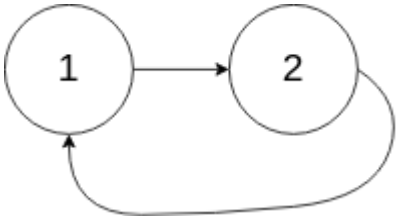
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：tail connects to node index 0
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3:

```
输入：head = [1], pos = -1
输出：no cycle
解释：链表中没有环。
```

进阶：

你是否可以不用额外空间解决此题？

思路：通过集合的方法

C# 语言

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 140 ms, 在所有 C# 提交中击败了 82.93% 的用户
- 内存消耗: 26 MB, 在所有 C# 提交中击败了 5.00% 的用户

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public int val;
 *     public ListNode next;
 *     public ListNode(int x) {
```

```
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution
{
    public ListNode DetectCycle(ListNode head)
    {
        HashSet<ListNode> h = new HashSet<ListNode>();
        ListNode temp = head;
        while (temp != null)
        {
            if (h.Contains(temp))
                return temp;

            h.Add(temp);
            temp = temp.next;
        }
        return null;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：72 ms, 在所有 Python3 提交中击败了 36.52% 的用户
- 内存消耗：17.2 MB, 在所有 Python3 提交中击败了 7.69% 的用户

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        h = set()
        temp = head
        while temp is not None:
            if temp in h:
```

```
            return temp

        h.add(temp)
        temp = temp.next
    return None
```

2.7 快乐数

- 题号：202
- 难度：简单
- https://leetcode-cn.com/problems/happy-number/

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：
1^2 + 9^2 = 82
8^2 + 2^2 = 68
6^2 + 8^2 = 100
1^2 + 0^2 + 0^2 = 1

输入：7
输出：true

输入：20
输出：false
解释：
20 => 4 + 0
4 => 16
16 => 1 + 36
37 => 9 + 49
58 => 25 + 64
89 => 64 + 81
```

```
145 => 1 + 16 + 25
42 => 16 + 4
20 可以看到，20 再次重复出现了，所以永远不可能等于 1
```

思路：通过集合的方法

C# 语言

- 执行结果：通过
- 执行用时：48 ms，在所有 C# 提交中击败了 80.74% 的用户
- 内存消耗：17 MB，在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public bool IsHappy(int n)
    {
        HashSet<int> h = new HashSet<int>();
        int m = 0;
        while (true)
        {
            while (n != 0)
            {
                m += (int)Math.Pow(n % 10,2);
                n /= 10;
            }
            if (m == 1)
            {
                return true;
            }
            if (h.Contains(m))
            {
                return false;
            }
            h.Add(m);
            n = m;
            m = 0;
        }
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：40 ms，在所有 Python3 提交中击败了 79.79% 的用户
- 内存消耗：13.8 MB，在所有 Python3 提交中击败了 9.09% 的用户

```python
class Solution:
    def isHappy(self, n: int) -> bool:
        h = set()
        m = 0
        while True:
            while n != 0:
                m += (n % 10) ** 2
                n //= 10
            if m == 1:
                return True
            if m in h:
                return False
            h.add(m)
            n = m
            m = 0
```

2.8 只出现一次的数字

- 题号：136
- 难度：简单
- https://leetcode-cn.com/problems/single-number/

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
```

思路：通过集合的方法

C# 语言

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 136 ms, 在所有 C# 提交中击败了 98.86% 的用户

- 内存消耗: 26.4 MB, 在所有 C# 提交中击败了 5.34% 的用户

```
public class Solution
{
    public int SingleNumber(int[] nums)
    {
        HashSet<int> h = new HashSet<int>();
        for (int i = 0; i < nums.Length; i++)
        {
            if (h.Contains(nums[i]))
            {
                h.Remove(nums[i]);
            }
            else
            {
                h.Add(nums[i]);
            }
        }
        return h.ElementAt(0);
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：60 ms, 在所有 Python3 提交中击败了 55.88% 的用户
- 内存消耗：15.6 MB, 在所有 Python3 提交中击败了 5.26% 的用户

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        h = set()
        for num in nums:
            if num in h:
                h.remove(num)
            else:
                h.add(num)
        return list(h)[0]
```

2.9 不邻接植花

- 题号：1042
- 难度：简单
- https://leetcode-cn.com/problems/flower-planting-with-no-adjacent/

有 `N` 个花园，按从 `1` 到 `N` 标记。在每个花园中，你打算种下四种花之一。

`paths[i] = [x, y]` 描述了花园 `x` 到花园 `y` 的双向路径。

另外，没有花园有 3 条以上的路径可以进入或者离开。

你需要为每个花园选择一种花，使得通过边相连的任何两个花园中的花的种类互不相同。

以数组形式返回选择的方案作为答案 `answer`，其中 `answer[i]` 为在第 `(i+1)` 个花园中种植的花的种类。花的种类用 1, 2, 3, 4 表示。保证存在答案。

示例 1：

```
输入：N = 3, paths = [[1,2],[2,3],[3,1]]
输出：[1,2,3]
```

示例 2：

```
输入：N = 4, paths = [[1,2],[3,4]]
输出：[1,2,1,2]
```

示例 3：

```
输入：N = 4, paths = [[1,2],[2,3],[3,4],[4,1],[1,3],[2,4]]
输出：[1,2,3,4]
```

思路：利用 字典 + 集合 构造图的邻接表。

C# 语言

- 执行结果：通过
- 执行用时：440 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗：48.9 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int[] GardenNoAdj(int N, int[][] paths)
    {
        Dictionary<int, HashSet<int>> graph = new Dictionary<int, HashSet<int>>();
        for (int i = 0; i < N; i++)
        {
            graph.Add(i, new HashSet<int>());
        }
        foreach (int[] path in paths)
        {
            int i = path[0] - 1;
            int j = path[1] - 1;
            graph[i].Add(j);
            graph[j].Add(i);
        }
        int[] result = new int[N];
        for (int i = 0; i < N; i++)
        {
            bool[] visited = new bool[5];
            foreach (int adj in graph[i])
            {
                visited[result[adj]] = true;
            }
            for (int j = 1; j <= 4; j++)
            {
                if (visited[j] == false)
                {
                    result[i] = j;
                    break;
                }
            }
        }
        return result;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：536 ms, 在所有 Python3 提交中击败了 62.29% 的用户
- 内存消耗：20.6 MB, 在所有 Python3 提交中击败了 33.33% 的用户

```python
class Solution:
    def gardenNoAdj(self, N: int, paths: List[List[int]]) -> List[int]:
        graph = {i: set() for i in range(0, N)}
        for path in paths:
            i = path[0] - 1
            j = path[1] - 1
            graph[i].add(j)
            graph[j].add(i)
        result = [0] * N
        for i in range(N):
            visited = [False] * 5
            for adj in graph[i]:
                visited[result[adj]] = True
            for j in range(1, 5):
                if visited[j] is False:
                    result[i] = j
                    break
        return result
```

3 字典技术在求解算法题中的应用

3.1 C# 和 Python 中的字典结构

C# 中字典的常用方法

对于 C# 中的 `Dictionary` 类 相信大家都不陌生，这是一个 `Collection`(集合) 类型，可以通过 Key/Value (键值对) 的形式来存放数据；该类最大的优点就是它查找元素的时间复杂度接近 O(1)，实际项目中常被用来做一些数据的本地缓存，提升整体效率。

常用方法如下：

- `public Dictionary();` -> 构造函数
- `public Dictionary(int capacity);` -> 构造函数
- `public void Add(TKey key, TValue value);` -> 将指定的键和值添加到字典中。
- `public bool Remove(TKey key);` -> 将带有指定键的值移除。
- `public void Clear();` -> 将所有键和值从字典中移除。
- `public bool ContainsKey(TKey key);` -> 确定是否包含指定键。
- `public bool ContainsValue(TValue value);` -> 确定否包含特定值。
- `public TValue this[TKey key] { get; set; }` -> 获取或设置与指定的键关联的值。
- `public KeyCollection Keys { get; }` -> 获得键的集合。
- `public ValueCollection Values { get; }` -> 获得值的集合。

举例如下：

```
public static void DicSample()
{
    Dictionary<string, string> dic = new Dictionary<string, string>();
    try
    {
        if (dic.ContainsKey("Item1") == false)
        {
            dic.Add("Item1", "ZheJiang");
        }
        if (dic.ContainsKey("Item2") == false)
        {
            dic.Add("Item2", "ShangHai");
        }
        else
        {
            dic["Item2"] = "ShangHai";
        }
        if (dic.ContainsKey("Item3") == false)
        {
            dic.Add("Item3", "BeiJing");
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Error: {0}", e.Message);
    }

    if (dic.ContainsKey("Item1"))
    {
        Console.WriteLine("Output: " + dic["Item1"]);
    }

    foreach (string key in dic.Keys)
    {
        Console.WriteLine("Output Key: {0}", key);
    }

    foreach (string value in dic.Values)
    {
        Console.WriteLine("Output Value: {0}", value);
    }

    foreach (KeyValuePair<string, string> item in dic)
    {
```

```
        Console.WriteLine("Output Key : {0}, Value : {1} ", item.Key, item.Value);
    }
}

// Output: ZheJiang
// Output Key: Item1
// Output Key: Item2
// Output Key: Item3
// Output Value: ZheJiang
// Output Value: ShangHai
// Output Value: BeiJing
// Output Key: Item1, Value: ZheJiang
// Output Key: Item2, Value: ShangHai
// Output Key: Item3, Value: BeiJing
```

注意：增加键值对之前需要判断是否存在该键，如果已经存在该键而不判断，将抛出异常。

有关更多 字典 的知识参见图文：

- 浅析 C# Dictionary 实现原理

Python 中字典的常用方法

Python 中的 字典 是无序的 键:值（key:value）对集合，在同一个字典之内键必须是互不相同的。

- dict 内部存放的顺序和 key 放入的顺序是没有关系的。
- dict 查找和插入的速度极快，不会随着 key 的增加而增加，但是需要占用大量的内存。

字典 定义语法为 {元素 1，元素 2，...，元素 n}

- 其中每一个元素是一个「键值对」-- 键:值 (key:value)

- 关键点是「大括号 {}」,「逗号 ,」和「冒号 :」
- 大括号 -- 把所有元素绑在一起
- 逗号 -- 将每个键值对分开
- 冒号 -- 将键和值分开

常用方法如下:

- `dict()` -> 构造函数。
- `dict(mapping)` -> 构造函数。
- `dict(**kwargs)` -> 构造函数。
- `dict.keys()` -> 返回一个可迭代对象，可以使用 `list()` 来转换为列表，列表为字典中的所有键。
- `dict.values()` -> 返回一个迭代器，可以使用 `list()` 来转换为列表，列表为字典中的所有值。
- `dict.items()` -> 以列表返回可遍历的 (键, 值) 元组数组。
- `dict.get(key, default=None)` -> 返回指定键的值，如果值不在字典中返回默认值。
- `dict.setdefault(key, default=None)` -> 和 `get()`方法 类似, 如果键不存在于字典中，将会添加键并将值设为默认值。
- `key in dict` -> `in` 操作符用于判断键是否存在于字典中，如果键在字典 dict 里返回 `true`，否则返回 `false`。
- `key not in dict` -> `not in` 操作符刚好相反，如果键在字典 dict 里返回 `false`，否则返回 `true`。

- `dict.pop(key[,default])` -> 删除字典给定键 `key` 所对应的值，返回值为被删除的值。`key` 值必须给出。若 `key` 不存在，则返回 `default` 值。
- `del dict[key]` -> 删除字典给定键 `key` 所对应的值。

举例如下：

```
def DicSample(self):
    dic = dict()
    try:
        if "Item1" not in dic:
            dic["Item1"] = "ZheJiang"
        if "Item2" not in dic:
            dic.setdefault("Item2", "ShangHai")
        else:
            dic["Item2"] = "ShangHai"
        dic["Item3"] = "BeiJing"
    except KeyError as error:
        print("Error:{0}".format(str(error)))

    if "Item1" in dic:
        print("Output: {0}".format(dic["Item1"]))

    for key in dic.keys():
        print("Output Key: {0}".format(key))

    for value in dic.values():
        print("Output Value: {0}".format(value))

    for key, value in dic.items():
        print("Output Key: {0}, Value: {1}".format(key, value))

# Output: ZheJiang
# Output Key: Item1
# Output Key: Item2
# Output Key: Item3
# Output Value: ZheJiang
# Output Value: ShangHai
# Output Value: BeiJing
# Output Key: Item1, Value: ZheJiang
# Output Key: Item2, Value: ShangHai
# Output Key: Item3, Value: BeiJing
```

3.2 两数之和

- 题号：1
- 难度：简单
- https://leetcode-cn.com/problems/two-sum/

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 1:

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

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

示例 2:

```
给定 nums = [230, 863, 916, 585, 981, 404, 316, 785, 88, 12, 70, 435, 384, 778, 887,
755, 740, 337, 86, 92, 325, 422, 815, 650, 920, 125, 277, 336, 221, 847, 168, 23,
677, 61, 400, 136, 874, 363, 394, 199, 863, 997, 794, 587, 124, 321, 212, 957, 764,
173, 314, 422, 927, 783, 930, 282, 306, 506, 44, 926, 691, 568, 68, 730, 933, 737,
531, 180, 414, 751, 28, 546, 60, 371, 493, 370, 527, 387, 43, 541, 13, 457, 328,
227, 652, 365, 430, 803, 59, 858, 538, 427, 583, 368, 375, 173, 809, 896, 370,
789], target = 542

因为 nums[28] + nums[45] = 221 + 321 = 542，所以返回 [28, 45]
```

思路：利用字典的方式

把字典当作一个存储容器，`key` 存储已经出现的数字，`value` 存储数组的下标。

C# 语言

- 执行结果：通过
- 执行用时：280 ms，在所有 C# 提交中击败了 96.53% 的用户
- 内存消耗：31.1 MB，在所有 C# 提交中击败了 6.89% 的用户

```
public class Solution
{
    public int[] TwoSum(int[] nums, int target)
    {
        int[] result = new int[2];
        Dictionary<int, int> dic = new Dictionary<int, int>();
        for (int i = 0; i < nums.Length; i++)
        {
            int find = target - nums[i];
            if (dic.ContainsKey(find))
            {
                result[0] = dic[find];
                result[1] = i;
                break;
            }
            if (dic.ContainsKey(nums[i]) == false)
                dic.Add(nums[i], i);
        }
        return result;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：52 ms，在所有 Python3 提交中击败了 86.77% 的用户
- 内存消耗：15.1 MB，在所有 Python3 提交中击败了 7.35% 的用户

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        result = list()
        dic = dict()
        for index, val in enumerate(nums):
            find = target - val
```

```
            if find in dic is not None:
                result = [dic[find], index]
                break
            else:
                dic[val] = index

        return result
```

3.3 只出现一次的数字 II

- 题号：137
- 难度：中等
- https://leetcode-cn.com/problems/single-number-ii/

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了三次。

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：[2,2,3,2]
输出：3
```

示例 2:

```
输入：[0,1,0,1,0,1,99]
输出：99
```

思路：利用字典的方式

把字典当作一个存储容器，`key` 存储数组中的数字，`value` 存储该数字出现的频数。

C# 语言

- 执行结果：通过
- 执行用时：112 ms，在所有 C# 提交中击败了 91.53% 的用户

- 内存消耗：25.4 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int SingleNumber(int[] nums)
    {
        Dictionary<int, int> dict = new Dictionary<int, int>();
        for (int i = 0; i < nums.Length; i++)
        {
            if (dict.ContainsKey(nums[i]))
            {
                dict[nums[i]]++;
            }
            else
            {
                dict.Add(nums[i], 1);
            }
        }
        return dict.Single(a => a.Value == 1).Key;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 89.20% 的用户
- 内存消耗：15.1 MB, 在所有 Python3 提交中击败了 25.00% 的用户

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        dic = dict()
        for num in nums:
            if num in dic:
                dic[num] += 1
            else:
                dic[num] = 1

        for k, v in dic.items():
            if v == 1:
                return k
```

```
        return -1
```

3.4 罗马数字转整数

- 题号：13
- 难度：简单
- https://leetcode-cn.com/problems/roman-to-integer/

罗马数字包含以下七种字符: I， V， X， L，C，D 和 M。

```
字符          数值
I             1
V             5
X             10
L             50
C             100
D             500
M             1000
```

例如， 罗马数字 2 写做 II，即为两个并列的 1。12 写做 XII，即为 X + II。 27 写做 XXVII，即为 XX + V + II。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

```
I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。
```

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

示例 1:

```
输入:"III"
输出: 3
```

示例 2:

```
输入: "IV"
输出: 4
```

示例 3:

```
输入: "IX"
输出: 9
```

示例 4:

```
输入: "LVIII"
输出: 58
解释: L = 50, V= 5, III = 3.
```

示例 5:

```
输入: "MCMXCIV"
输出: 1994
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

思路：利用字典的方式

把字典当作一个存储容器，key 存储罗马字符的所有组合，value 存储该组合代表的值。

每次取一个字符，判断这个字符之后是否还有字符。如果有，则判断这两个字符是否在字典中，如果存在则取值。否则，按照一个字符去取值即可。

C# 语言

- 执行结果：通过
- 执行用时：120 ms, 在所有 C# 提交中击败了 42.16% 的用户
- 内存消耗：25.8 MB, 在所有 C# 提交中击败了 5.27% 的用户

```
public class Solution
{
    public int RomanToInt(string s)
    {
        Dictionary<string, int> dic = new Dictionary<string, int>();
        dic.Add("I", 1);
```

```
        dic.Add("II", 2);
        dic.Add("IV", 4);
        dic.Add("IX", 9);
        dic.Add("X", 10);
        dic.Add("XL", 40);
        dic.Add("XC", 90);
        dic.Add("C", 100);
        dic.Add("CD", 400);
        dic.Add("CM", 900);
        dic.Add("V", 5);
        dic.Add("L", 50);
        dic.Add("D", 500);
        dic.Add("M", 1000);

        int result = 0;
        int count = s.Length;
        int i = 0;
        while (i < count)
        {
            char c = s[i];
            if (i + 1 < count && dic.ContainsKey(s.Substring(i, 2)))
            {
                result += dic[s.Substring(i, 2)];
                i += 2;
            }
            else
            {
                result += dic[c.ToString()];
                i += 1;
            }
        }
        return result;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：72 ms, 在所有 Python3 提交中击败了 24.93% 的用户
- 内存消耗：13.5 MB, 在所有 Python3 提交中击败了 5.05% 的用户

```python
class Solution:
    def romanToInt(self, s: str) -> int:
        dic = {"I": 1, "II": 2, "IV": 4, "IX": 9, "X": 10, "XL": 40, "XC": 90,
               "C": 100, "CD": 400, "CM": 900, "V": 5,
               "L": 50, "D": 500, "M": 1000}
        result = 0
        count = len(s)
        i = 0
        while i < count:
            c = s[i]
            if i + 1 < count and s[i:i + 2] in dic:
                result += dic[s[i:i + 2]]
                i += 2
            else:
                result += dic[c]
                i += 1
        return result
```

3.5 LRU 缓存机制

- 题号：146
- 难度：中等
- https://leetcode-cn.com/problems/lru-cache/

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 `get(key)` - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 `put(key, value)` - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 O(1) 时间复杂度内完成这两种操作?

示例:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

思路：利用 字典 + 列表 的方式

计算机的缓存容量有限，如果缓存满了就要删除一些内容，给新内容腾位置。但问题是，删除哪些内容呢？我们肯定希望删掉哪些没什么用的缓存，而把有用的数据继续留在缓存里，方便之后继续使用。那么，什么样的数据，我们判定为「有用的」的数据呢？

LRU 缓存淘汰算法就是一种常用策略。LRU 的全称是 Least Recently Used，也就是说我们认为最近使用过的数据应该是是「有用的」，很久都没用过的数据应该是无用的，内存满了就优先删那些很久没用过的数据。

把字典当作一个存储容器，由于字典是无序的，即 `dict` 内部存放的顺序和 `key` 放入的顺序是没有关系的，所以需要一个 `list` 来辅助排序。

C# 语言

- 状态：通过
- 18 / 18 个通过测试用例
- 执行用时: 392 ms, 在所有 C# 提交中击败了 76.56% 的用户
- 内存消耗: 47.9 MB, 在所有 C# 提交中击败了 20.00% 的用户

```
public class LRUCache
{
    private readonly List<int> _keys;
    private readonly Dictionary<int, int> _dict;


    public LRUCache(int capacity)
    {
        _keys = new List<int>(capacity);
        _dict = new Dictionary<int, int>(capacity);
    }
```

```
    public int Get(int key)
    {
        if (_dict.ContainsKey(key))
        {
            _keys.Remove(key);
            _keys.Add(key);
            return _dict[key];
        }
        return -1;
    }

    public void Put(int key, int value)
    {
        if (_dict.ContainsKey(key))
        {
            _dict.Remove(key);
            _keys.Remove(key);
        }
        else if (_keys.Count == _keys.Capacity)
        {
            _dict.Remove(_keys[0]);
            _keys.RemoveAt(0);
        }
        _keys.Add(key);
        _dict.Add(key, value);
    }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.Get(key);
 * obj.Put(key,value);
 */
```

Python 语言

- 执行结果：通过
- 执行用时：628 ms，在所有 Python3 提交中击败了 12.15% 的用户
- 内存消耗：22 MB，在所有 Python3 提交中击败了 65.38% 的用户

```python
class LRUCache:

    def __init__(self, capacity: int):
        self._capacity = capacity
        self._dict = dict()
        self._keys = list()

    def get(self, key: int) -> int:
        if key in self._dict:
            self._keys.remove(key)
            self._keys.append(key)
            return self._dict[key]
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self._dict:
            self._dict.pop(key)
            self._keys.remove(key)
        elif len(self._keys) == self._capacity:
            self._dict.pop(self._keys[0])
            self._keys.remove(self._keys[0])
        self._keys.append(key)
        self._dict[key] = value

# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

注意，这两行代码不能颠倒顺序，否则 `dict` 中就不会存在`_keys[0]`了。

```
self._dict.pop(self._keys[0])
self._keys.remove(self._keys[0])
```

3.6 不邻接植花

- 题号：1042
- 难度：简单
- https://leetcode-cn.com/problems/flower-planting-with-no-adjacent/

有 `N` 个花园，按从 `1` 到 `N` 标记。在每个花园中，你打算种下四种花之一。

`paths[i] = [x, y]` 描述了花园 `x` 到花园 `y` 的双向路径。

另外，没有花园有 3 条以上的路径可以进入或者离开。

你需要为每个花园选择一种花，使得通过边相连的任何两个花园中的花的种类互不相同。

以数组形式返回选择的方案作为答案 `answer`，其中 `answer[i]` 为在第 `(i+1)` 个花园中种植的花的种类。花的种类用 1, 2, 3, 4 表示。保证存在答案。

示例 1：

```
输入：N = 3, paths = [[1,2],[2,3],[3,1]]
输出：[1,2,3]
```

示例 2：

```
输入：N = 4, paths = [[1,2],[3,4]]
输出：[1,2,1,2]
```

示例 3：

```
输入：N = 4, paths = [[1,2],[2,3],[3,4],[4,1],[1,3],[2,4]]
输出：[1,2,3,4]
```

思路：利用 字典 + 集合 构造图的邻接表。

C# 语言

- 执行结果：通过
- 执行用时：440 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗：48.9 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int[] GardenNoAdj(int N, int[][] paths)
    {
        Dictionary<int, HashSet<int>> graph = new Dictionary<int, HashSet<int>>();
        for (int i = 0; i < N; i++)
        {
            graph.Add(i, new HashSet<int>());
        }
        foreach (int[] path in paths)
        {
            int i = path[0] - 1;
            int j = path[1] - 1;
            graph[i].Add(j);
            graph[j].Add(i);
        }
        int[] result = new int[N];
        for (int i = 0; i < N; i++)
        {
            bool[] visited = new bool[5];
            foreach (int adj in graph[i])
            {
                visited[result[adj]] = true;
            }
            for (int j = 1; j <= 4; j++)
            {
                if (visited[j] == false)
                {
                    result[i] = j;
                    break;
                }
            }
        }
        return result;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：536 ms, 在所有 Python3 提交中击败了 62.29% 的用户
- 内存消耗：20.6 MB, 在所有 Python3 提交中击败了 33.33% 的用户

```python
class Solution:
    def gardenNoAdj(self, N: int, paths: List[List[int]]) -> List[int]:
        graph = {i: set() for i in range(0, N)}
        for path in paths:
            i = path[0] - 1
            j = path[1] - 1
            graph[i].add(j)
            graph[j].add(i)
        result = [0] * N
        for i in range(N):
            visited = [False] * 5
            for adj in graph[i]:
                visited[result[adj]] = True
            for j in range(1, 5):
                if visited[j] is False:
                    result[i] = j
                    break
        return result
```

4 排序技术在求解算法题中的应用

4.1 C# 和 Python 中的排序操作

C# 中的排序

对集合类的排序，我们通常使用位于 System.Core 程序集，System.Linq 命名空间下，Enumerable 静态类中的扩展方法。

```
public static class Enumerable
{
    public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
}
```

该 OrderBy 是对 IEnumerable<T>类型的扩展，而 IEnumerable<T>是整个 LINQ 的基础。

C# 大部分数据结构都实现了 IEnumerable<T>，比如：List<T>，IDictionary<K,T>，LinkedList<T>，Stack<T>，Queue<T>等，都可以使用 LINQ 中的扩展方法。

有关更多扩展方法的知识参见图文：

- 浅析 C# 语言中的扩展方法

Python 中的排序

对列表的排序通常有两种方法，第一种使用 list 本身的 sort 方法，第二种使用 Python 的内置方法 sorted。

list.sort(key=None, reverse=False) 对原列表进行排序。

- key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。
- reverse -- 排序规则，reverse = True 降序， reverse = False 升序（默认）。
- 该方法没有返回值，但是会对列表的对象进行排序。

Sample01：

```
list1 = [123, 456, 789, 213]
list1.sort()
print(list1)  # [123, 213, 456, 789]

list1.sort(reverse=True)
print(list1)  # [789, 456, 213, 123]
```

Sample02：

```
# 获取列表的第二个元素
def takeSecond(elem):
    return elem[1]


r = [(2, 2), (3, 4), (4, 1), (1, 3)]
r.sort(key=takeSecond)
print(r)

# [(4, 1), (2, 2), (1, 3), (3, 4)]
```

sorted(iterable, key=None, reverse=False) 对所有可迭代的对象进行排序操作。

- iterable -- 可迭代对象。
- key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。
- reverse -- 排序规则，reverse = True 降序 ， reverse = False 升序（默认）。
- 返回重新排序的列表。

Sample01:

```
numbers = [-8, 99, 3, 7, 83]
print(sorted(numbers))  # [-8, 3, 7, 83, 99]
print(sorted(numbers, reverse=True))  # [99, 83, 7, 3, -8]
```

Sample02:

```
array = [{"age": 20, "name": "a"}, {"age": 25, "name": "b"}, {"age": 10, "name":
"c"}]
array = sorted(array, key=lambda x: x["age"])
print(array)

# [{'age': 10, 'name': 'c'}, {'age': 20, 'name': 'a'}, {'age': 25, 'name': 'b'}]
```

4.2 求众数

- 题号：169
- 难度：简单
- https://leetcode-cn.com/problems/majority-element/

给定一个大小为 n 的数组，找到其中的众数。众数是指在数组中出现次数大于⌊ n/2 ⌋的元素。

你可以假设数组是非空的，并且给定的数组总是存在众数。

示例 1:

```
输入: [3,2,3]
输出: 3
```

示例 2:

```
输入: [2,2,1,1,1,2,2]
输出: 2
```

思路：利用排序的方法

C# 语言

- 状态：通过
- 44 / 44 个通过测试用例
- 执行用时：192 ms

```
public class Solution
{
    public int MajorityElement(int[] nums)
    {
```

```
        nums = nums.OrderBy(a => a).ToArray();
        return nums[nums.Length / 2];
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：48 ms，在所有 Python3 提交中击败了 82.08% 的用户
- 内存消耗：15.2 MB，在所有 Python3 提交中击败了 6.90% 的用户

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        nums.sort()
        return nums[len(nums) // 2]
```

4.3 数组中的第 K 个最大元素

- 题号：215
- 难度：中等
- https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
```

示例 2:

```
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

思路：利用排序的方法

C# 语言

- 状态：通过
- 32 / 32 个通过测试用例
- 执行用时: 152 ms, 在所有 C# 提交中击败了 76.47% 的用户
- 内存消耗: 24.6 MB, 在所有 C# 提交中击败了 5.55% 的用户

```
public class Solution
{
    public int FindKthLargest(int[] nums, int k)
    {
        nums = nums.OrderBy(a => a).ToArray();
        return nums[nums.Length - k];
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 92.64% 的用户
- 内存消耗：14.4 MB, 在所有 Python3 提交中击败了 15.79% 的用户

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        nums.sort()
        return nums[len(nums) - k]
```

4.4 两个数组的交集 II

- 题号：350
- 难度：简单
- https://leetcode-cn.com/problems/intersection-of-two-arrays-ii/

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？

思路：利用 排序 + 双索引 的方法

C# 语言

- 执行结果：通过
- 执行用时：320 ms, 在所有 C# 提交中击败了 23.03% 的用户
- 内存消耗：31.2 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int[] Intersect(int[] nums1, int[] nums2)
    {
        nums1 = nums1.OrderBy(a => a).ToArray();
        nums2 = nums2.OrderBy(a => a).ToArray();
        List<int> result = new List<int>();
        int i = 0, j = 0;
        while (i < nums1.Length && j < nums2.Length)
        {
            if (nums1[i] < nums2[j])
            {
                i++;
            }
            else if (nums1[i] > nums2[j])
            {
                j++;
            }
            else
            {
                result.Add(nums1[i]);
                i++;
                j++;
            }
        }
        return result.ToArray();
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：64 ms, 在所有 Python3 提交中击败了 62.00% 的用户
- 内存消耗：13.7 MB, 在所有 Python3 提交中击败了 12.50% 的用户

```python
class Solution:
    def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
        nums1.sort()
        nums2.sort()
        result = []
        i, j = 0, 0
        while i < len(nums1) and j < len(nums2):
            if nums1[i] < nums2[j]:
                i += 1
            elif nums1[i] > nums2[j]:
                j += 1
            else:
                result.append(nums1[i])
                i += 1
                j += 1
        return result
```

4.5 最接近的三数之和

- 题号：16
- 难度：中等
- https://leetcode-cn.com/problems/3sum-closest/

给定一个包括 n 个整数的数组 nums 和一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

示例：

```
例如，给定数组 nums = [-1, 2, 1, -4]，和 target = 1.
与 target 最接近的三个数的和为 2. (-1 + 2 + 1 = 2).
```

思路：利用 排序 + 三索引 的方法

C# 实现

- 状态：通过
- 125 / 125 个通过测试用例
- 执行用时: 132 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 24 MB, 在所有 C# 提交中击败了 5.55% 的用户

```
public class Solution
{
    public int ThreeSumClosest(int[] nums, int target)
    {
        nums = nums.OrderBy(a => a).ToArray();
        int result = nums[0] + nums[1] + nums[2];
        for (int i = 0; i < nums.Length - 2; i++)
        {
            int start = i + 1, end = nums.Length - 1;
```

```
            while (start < end)
            {
                int sum = nums[start] + nums[end] + nums[i];
                if (Math.Abs(target - sum) < Math.Abs(target - result))
                    result = sum;
                if (sum > target)
                    end--;
                else if (sum < target)
                    start++;
                else
                    return result;
            }
        }
        return result;
    }
}
```

Pyhton 实现

- 执行结果：通过
- 执行用时：124 ms, 在所有 Python3 提交中击败了 72.19% 的用户
- 内存消耗：13.2 MB, 在所有 Python3 提交中击败了 22.06% 的用户

```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums = sorted(nums)
        result = nums[0] + nums[1] + nums[2]
        for i in range(0, len(nums) - 2):
            start = i + 1
            end = len(nums) - 1
            while start < end:
                sum = nums[start] + nums[end] + nums[i]
                if abs(target - sum) < abs(target - result):
                    result = sum
                if sum > target:
                    end -= 1
                elif sum < target:
                    start += 1
                else:
                    return result
        return result
```

4.6 三数之和

- 题号：15
- 难度：中等
- https://leetcode-cn.com/problems/3sum/

给定一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c，使得 a + b + c = 0？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

```
给定数组 nums = [-1, 0, 1, 2, -1, -4],

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

思路：利用 排序 + 三索引 的方法

为了避免三次循环，提升执行效率。首先，对 nums 进行排序。然后，固定 3 个索引 i,l(left),r(right)，i 进行最外层循环，l 指向 nums[i]之后数组的最小值，r 指向 nums[i]之后数组的最大值。模仿快速排序的思路，如果 nums[i] > 0 就不需要继续计算了，否则计算 nums[i] + nums[l] + nums[r]是否等于零并进行相应的处理。如果大于零，向 l 方向移动 r 指针，如果小于零，向 r 方向移动 l 索引，如果等于零，则加入到存储最后结果的 result 链表中。当然，题目中要求这个三元组不可重复，所以在进行的过程中加入去重就好。

C# 实现

- 执行结果：通过
- 执行用时：348 ms, 在所有 C# 提交中击败了 99.54% 的用户
- 内存消耗：35.8 MB, 在所有 C# 提交中击败了 6.63% 的用户

```
public class Solution
{
    public IList<IList<int>> ThreeSum(int[] nums)
    {
        IList<IList<int>> result = new List<IList<int>>();
        nums = nums.OrderBy(a => a).ToArray();
        int len = nums.Length;

        for (int i = 0; i < len - 2; i++)
        {
            if (nums[i] > 0)
                break; // 如果最小的数字大于0，后面的操作已经没有意义

            if (i > 0 && nums[i - 1] == nums[i])
                continue; // 跳过三元组中第一个元素的重复数据

            int l = i + 1;
            int r = len - 1;

            while (l < r)
            {
                int sum = nums[i] + nums[l] + nums[r];
                if (sum < 0)
                {
                    l++;
                }
                else if (sum > 0)
                {
                    r--;
                }
                else
                {
                    result.Add(new List<int>() {nums[i], nums[l], nums[r]});
                    // 跳过三元组中第二个元素的重复数据
```

```
                while (l < r && nums[l] == nums[l + 1])
                {
                    l++;
                }
                // 跳过三元组中第三个元素的重复数据
                while (l < r && nums[r - 1] == nums[r])
                {
                    r--;
                }
                l++;
                r--;
            }
        }
    }
    return result;
  }
}
```

Python 实现

- 执行结果：通过
- 执行用时：660 ms，在所有 Python3 提交中击败了 95.64% 的用户
- 内存消耗：16.1 MB，在所有 Python3 提交中击败了 75.29% 的用户

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums = sorted(nums)
        result = []

        for i in range(0, len(nums) - 2):
            # 如果最小的数字大于0，后面的操作已经没有意义
            if nums[i] > 0:
                break
            # 跳过三元组中第一个元素的重复数据
            if i > 0 and nums[i-1] == nums[i]:
                continue

            # 限制nums[i]是三元组中最小的元素
            l = i + 1
            r = len(nums) - 1
            while l < r:
```

```
            sum = nums[i] + nums[l] + nums[r]
            if sum < 0:
                l += 1
            elif sum > 0:
                r -= 1
            else:
                result.append([nums[i], nums[l], nums[r]])
                # 跳过三元组中第二个元素的重复数据
                while l < r and nums[l] == nums[l+1]:
                    l += 1
                # 跳过三元组中第三个元素的重复数据
                while l < r and nums[r] == nums[r-1]:
                    r -= 1
                l += 1
                r -= 1
        return result
```

5 位运算技术在求解算法题中的应用

5.1 C# 和 Python 中的位运算操作

1. 原码、反码和补码

二进制有三种不同的表示形式：原码、反码和补码，计算机内部使用补码来表示。

原码：就是其二进制表示（注意，有一位符号位）。

```
00 00 00 11 -> 3
10 00 00 11 -> -3
```

反码：正数的反码就是原码，负数的反码是符号位不变，其余位取反（对应正数按位取反）。

```
00 00 00 11 -> 3
11 11 11 00 -> -3
```

补码：正数的补码就是原码，负数的补码是反码+1。

```
00 00 00 11 -> 3
11 11 11 01 -> -3
```

符号位：最高位为符号位，0 表示正数，1 表示负数。在位运算中符号位也参与运算。

2. 按位非操作 ~

```
~ 1 = 0
~ 0 = 1
```

~ 把 num 的补码中的 0 和 1 全部取反（0 变为 1，1 变为 0）有符号整数的符号位在 ~ 运算中同样会取反。

```
00 00 01 01 -> 5
~
---
11 11 10 10 -> -6

11 11 10 11 -> -5
~
---
00 00 01 00 -> 4
```

3. 按位与操作 &

```
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
```

只有两个对应位都为 1 时才为 1

```
00 00 01 01 -> 5
&
00 00 01 10 -> 6
---
00 00 01 00 -> 4
```

4. 按位或操作 |

```
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
```

只要两个对应位中有一个 1 时就为 1

```
00 00 01 01 -> 5
|
00 00 01 10 -> 6
---
00 00 01 11 -> 7
```

5. 按位异或操作 ^

```
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
```

```
0 ^ 0 = 0
```

只有两个对应位不同时才为 1

```
00 00 01 01 -> 5
^
00 00 01 10 -> 6
---
00 00 00 11 -> 3
```

异或操作的性质：满足交换律和结合律

```
A: 00 00 11 00
B: 00 00 01 11

A^B: 00 00 10 11
B^A: 00 00 10 11

A^A: 00 00 00 00
A^0: 00 00 11 00

A^B^A: = A^A^B = B = 00 00 01 11
```

6. 按位左移操作 <<

`num << i` 将 `num` 的二进制表示向左移动 `i` 位所得的值。

```
00 00 10 11 -> 11
11 << 3
---
01 01 10 00 -> 88
```

7. 按位右移操作 >>

`num >> i` 将 `num` 的二进制表示向右移动 `i` 位所得的值。

```
00 00 10 11 -> 11
11 >> 2
---
00 00 00 10 -> 2
```

8. 利用位运算实现快速计算

通过 `<<`, `>>` 快速计算 2 的倍数问题。

```
n << 1 -> 计算 n*2
n >> 1 -> 计算 n/2，负奇数的运算不可用
n << m -> 计算 n*(2^m)，即乘以 2 的 m 次方
n >> m -> 计算 n/(2^m)，即除以 2 的 m 次方
1 << n -> 2^n
```

通过 ^ 快速交换两个整数。

```
a ^= b
b ^= a
a ^= b
```

通过 `a & (-a)` 快速获取 a 的最后为 1 位置的整数。

```
00 00 01 01 -> 5
&
11 11 10 11 -> -5
---
00 00 00 01 -> 1

00 00 11 10 -> 14
&
11 11 00 10 -> -14
---
00 00 00 10 -> 2
```

9. 利用位运算实现整数集合

一个数的二进制表示可以看作是一个集合（0 表示不在集合中，1 表示在集合中）。

比如集合 `{1, 3, 4, 8}`，可以表示成 `01 00 01 10 10` 而对应的位运算也就可以看作是对集合进行的操作。

元素与集合的操作：

```
a | (1<<i)  -> 把 i 插入到集合中
a & ~(1<<i) -> 把 i 从集合中删除
a & (1<<i)  -> 判断 i 是否属于该集合（零不属于，非零属于）
```

集合之间的操作：

```
a 补   -> ~a
a 交 b -> a & b
a 并 b -> a | b
```

```
a 差 b -> a & (~b)
```

5.2 只出现一次的数字

- 题号：136
- 难度：简单
- https://leetcode-cn.com/problems/single-number/

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
```

思路： 利用"异或"操作的性质。

```
A: 00 00 11 00
B: 00 00 01 11

A^A: 00 00 00 00
A^0: 00 00 11 00

A^B^A: = A^A^B = B = 00 00 01 11
```

C# 实现

- 状态：通过
- 16 / 16 个通过测试用例
- 执行用时: 144 ms, 在所有 C# 提交中击败了 91.76% 的用户
- 内存消耗: 25.4 MB, 在所有 C# 提交中击败了 11.39% 的用户

```
public class Solution
{
    public int SingleNumber(int[] nums)
    {
        int result = 0;

        for (int i = 0; i < nums.Length; i++)
        {
            result ^= nums[i];
        }
        return result;
    }
}
```

Python 实现

- 执行结果：通过
- 执行用时：44 ms, 在所有 Python3 提交中击败了 84.17% 的用户
- 内存消耗：15.3 MB, 在所有 Python3 提交中击败了 5.26% 的用户

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        result = 0
        for item in nums:
            result ^= item
        return result
```

5.3 2 的幂

- 题号：231
- 难度：简单
- https://leetcode-cn.com/problems/power-of-two/

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

示例 1:

```
输入：1
输出：true
解释：2^0 = 1
```

示例 2:

```
输入：16
输出：true
解释：2^4 = 16
```

示例 3:

```
输入：218
输出：false
```

思路： 利用"异或"操作的性质。

```
A: 00 00 11 00

A^A: 00 00 00 00
```

C# 语言

- 状态：通过
- 1108 / 1108 个通过测试用例

- 执行用时: 36 ms, 在所有 C# 提交中击败了 100.00% 的用户
- 内存消耗: 14.7 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public bool IsPowerOfTwo(int n)
    {
        if (n < 0)
            return false;
        for (int i = 0; i < 32; i++)
        {
            int mask = 1 << i;
            if ((n ^ mask) == 0)
                return true;
        }
        return false;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：44 ms, 在所有 Python3 提交中击败了 51.91% 的用户
- 内存消耗：13.6 MB, 在所有 Python3 提交中击败了 6.25% 的用户

```
class Solution:
    def isPowerOfTwo(self, n: int) -> bool:
        for i in range(32):
            mask = 1 << i
            if n ^ mask == 0:
                return True
        return False
```

5.4 只出现一次的数字 III

- 题号：260
- 难度：中等
- https://leetcode-cn.com/problems/single-number-iii/

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。

找出只出现一次的那两个元素。

示例 ：

```
输入: [1,2,1,3,2,5]
输出: [3,5]
```

注意：

1. 结果输出的顺序并不重要，对于上面的例子， [5, 3] 也是正确答案。
2. 你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现?

思路： 利用"异或"操作的性质。

通过异或操作 ^ 去除掉恰好重复出现两次的元素，这时得到两个只出现一次整数的异或结果 `different`。

```
A: 00 00 11 00
B: 00 00 01 11

A^B: 00 00 10 11
B^A: 00 00 10 11

A^A: 00 00 00 00
A^0: 00 00 11 00
```

```
A^B^A: = A^A^B = B = 00 00 01 11
```

获取 `different` 二进制中最后一位 1，通过该位，可以把这两个数分离出来，这两个数在该位是不同的。也即通过该位把 `nums` 分成了两组，该位是 1 的一组，该位是 0 的一组，然后求两组中只出现一次的整数。

通过 `a & (-a)` 快速获取 `a` 的最后为 1 位置的整数。

```
00 00 01 01 -> 5
&
11 11 10 11 -> -5
---
00 00 00 01 -> 1

00 00 11 10 -> 14
&
11 11 00 10 -> -14
---
00 00 00 10 -> 2
```

C# 语言

- 执行结果：通过
- 执行用时：280 ms, 在所有 C# 提交中击败了 83.33% 的用户
- 内存消耗：31.2 MB, 在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int[] SingleNumber(int[] nums)
    {
        if (nums.Length < 2)
            return new int[2];
        int different = 0;
        for (int i = 0; i < nums.Length; i++)
        {
            different ^= nums[i];
        }
        different &= -1 * different;
        int num1 = 0, num2 = 0;
```

```
        for (int i = 0; i < nums.Length; i++)
        {
            if ((different & nums[i]) == 0)
            {
                num1 ^= nums[i];
            }
            else
            {
                num2 ^= nums[i];
            }
        }
        return new int[] { num1, num2 };
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：44 ms, 在所有 Python3 提交中击败了 83.70% 的用户
- 内存消耗：14.8 MB, 在所有 Python3 提交中击败了 33.33% 的用户

```
class Solution:
    def singleNumber(self, nums: List[int]) -> List[int]:
        if len(nums) < 2:
            return []
        different = 0
        for num in nums:
            different ^= num
        different &= -1 * different
        num1, num2 = 0, 0
        for num in nums:
            if (num & different) == 0:
                num1 ^= num
            else:
                num2 ^= num
        return [num1, num2]
```

5.5 子集

- 题号：78
- 难度：中等
- https://leetcode-cn.com/problems/subsets/

给定一组 不含重复元素 的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

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

思路： 利用整数集合的思路。

以{1,2,3}为例，三个数，共 2^3 个子集。

```
000 -> []
100 -> [1]
101 -> [1,3]
110 -> [1,2]
111 -> [1,2,3]
...
```

C# 语言

- 状态：通过
- 10 / 10 个通过测试用例
- 执行用时: 348 ms, 在所有 C# 提交中击败了 97.80% 的用户
- 内存消耗: 29.5 MB, 在所有 C# 提交中击败了 6.67% 的用户

```
public class Solution
{
    public IList<IList<int>> Subsets(int[] nums)
    {
        IList<IList<int>> result = new List<IList<int>>();
        int count = nums.Length;

        for (int i = 0; i < 1 << count; i++)
        {
            IList<int> item = new List<int>();
            for (int j = 0; j < count; j++)
            {
                int mask = 1 << j;
                if ((mask & i) != 0)
                    item.Add(nums[j]);
            }
            result.Add(item);
        }
        return result;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：40 ms, 在所有 Python3 提交中击败了 63.08% 的用户
- 内存消耗：13.8 MB, 在所有 Python3 提交中击败了 5.72% 的用户

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        count = len(nums)
        result = []
```

```
        for i in range(1 << count):
            item = []
            for j in range(count):
                mask = 1 << j
                if (mask & i) != 0:
                    item.append(nums[j])
            result.append(item)
        return result
```

5.6 Pow(x, n)

- 题号：50
- 难度：中等
- https://leetcode-cn.com/problems/powx-n/

实现 pow(x, n)，即计算 x 的 n 次幂函数。

示例 1:

```
输入：2.00000, 10
输出：1024.00000
```

示例 2:

```
输入：2.10000, 3
输出：9.26100
```

示例 3:

```
输入：2.00000, -2
输出：0.25000
解释：2-2 = 1/22 = 1/4 = 0.25
```

示例 4:

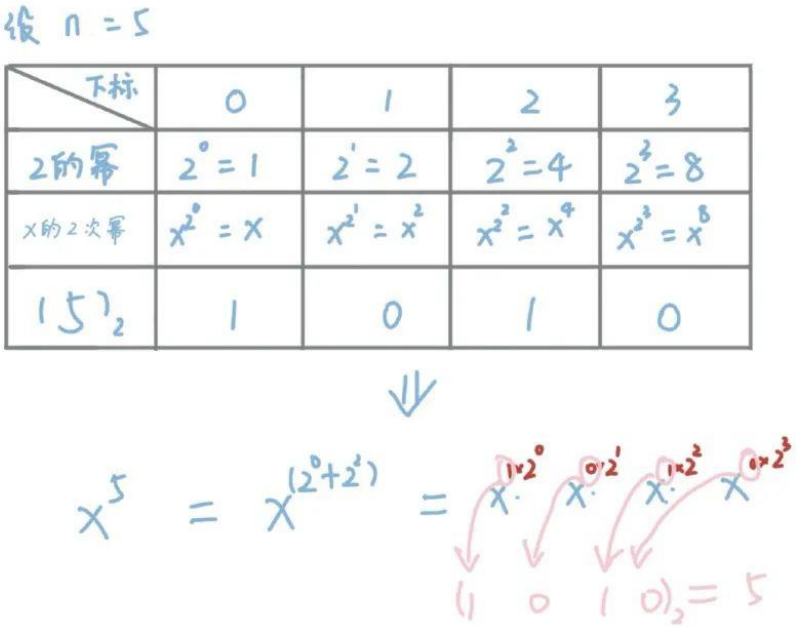
```
输入：1.00000, -2147483648
输出：1.00000
```

说明:

- -100.0 < x < 100.0
- n 是 32 位有符号整数，其数值范围是 [−2^31, 2^31 − 1] 。

思路：利用快速幂法。

假设我们要求 a^b，那么 b 可以拆成二进制表示，例如当 b = 5 时，5 的二进制是 0101，5 = 2^3×0 + 2^2×1 + 2^1×0 + 2^0×1，因此，我们将 a^5 转化为算 a^(2^3×0 + 2^2×1 + 2^1×0 + 2^0×1)，即 a^(2^0) × a^(2^2)。



我们先算出所有 2 的幂，然后在算出所有 x 的 2 的幂次方。再把 n 拆成二进制，把二进制当中对应位置是 1 的值乘起来，就得到了结果。这套方法称为 **快速幂法。**

C# 实现

- 执行结果：通过
- 执行用时：56 ms, 在所有 C# 提交中击败了 51.87% 的用户
- 内存消耗：15.1 MB, 在所有 C# 提交中击败了 50.00% 的用户

```
public class Solution
{
    public double MyPow(double x, int n)
    {
        int neg = n < 0 ? -1 : 1;
```

```
        long g = Math.Abs((long)n);

        double[] d = new double[32];
        d[0] = x;
        for (int i = 1; i < 32; i++)
        {
            d[i] = d[i - 1] * d[i - 1];
        }

        double result = 1.0d;
        for (int i = 0; i < 32; i++)
        {
            int mask = 1 << i;
            if ((mask & g) != 0)
            {
                result *= d[i];
            }
        }
        return neg != -1 ? result : 1.0 / result;
    }
}
```

注意：long g = Math.Abs((long)n);需要把 n 转换成 long，因为 Math.Abs(int.MinValue) 会产生溢出错误。

Python 实现

- 执行结果：通过
- 执行用时：36 ms，在所有 Python3 提交中击败了 75.02% 的用户
- 内存消耗：13.8 MB，在所有 Python3 提交中击败了 8.33% 的用户

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        neg = -1 if n < 0 else 1
        n = abs(n)
        d = list()
        d.append(x)
        for i in range(1, 32):
            d.append(d[-1] * d[-1])
        result = 1.0
```

```
        for i in range(32):
            mask = 1 << i
            if (mask & n) != 0:
                result *= d[i]
        return result if neg != -1 else 1.0 / result
```

5.7 只出现一次的数字 II

- 题号：137
- 难度：中等
- https://leetcode-cn.com/problems/single-number-ii/

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了三次。

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: [2,2,3,2]
输出: 3
```

示例 2:

```
输入: [0,1,0,1,0,1,99]
输出: 99
```

思路：

初始 `result = 0`，将每个数想象成 32 位的二进制，对于每一位的二进制的 1 累加起来必然是 `3N` 或者 `3N + 1`(出现 3 次和 1 次)；`3N` 代表目标值在这一位没贡献，`3N + 1` 代表目标值在这一位有贡献（=1），然后将所有有贡献的位记录到 `result` 中。这样做的好处是如果题目改成 `k` 个一样，只需要把代码改成 `count % k` 即可，很通用并列去找每一位。

C# 语言

- 执行结果：通过
- 执行用时：112 ms，在所有 C# 提交中击败了 91.53% 的用户
- 内存消耗：25.2 MB，在所有 C# 提交中击败了 100.00% 的用户

```
public class Solution
{
    public int SingleNumber(int[] nums)
    {
        int result = 0;
        for (int i = 0; i < 32; i++)
        {
            int mask = 1 << i;
            int count = 0;
            for (int j = 0; j < nums.Length; j++)
            {
                if ((nums[j] & mask) != 0)
                {
                    count++;
                }
            }
            if (count % 3 != 0)
            {
                result |= mask;
            }
        }
        return result;
    }
}
```

Python 语言

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        result = 0
        for i in range(32):
            mask = 1 << i
            count = 0
            for num in nums:
                if num & mask != 0:
                    count += 1
            if count % 3 != 0:
```

```
                result |= mask
        return result
```

以上 Python 代码与 C# 代码逻辑完全一致，但提交时报错，错误信息如下：

```
输入：[-2,-2,1,1,-3,1,-3,-3,-4,-2]
输出：4294967292
预期结果：-4
```

我们发现：

```
-4 补码为 1111 1111 1111 1111 1111 1111 1111 1100
```

如果不考虑符号位

```
1111 1111 1111 1111 1111 1111 1111 1100 -> 4294967292
```

是不是很坑，C++，C#，Java 等语言的整型是限制长度的，如：byte 8 位，int 32 位，long 64 位，但 Python 的整型是不限制长度的（即不存在高位溢出），所以，当输出是负数的时候，会导致认为是正数！因为它把 32 位有符号整型认为成了无符号整型，真是坑。

我们对以上的代码进行修改，加入判断条件 `if result > 2 ** 31-1:` 超过 32 位整型的范围就表示负数了 `result -= 2 ** 32`，即可得到对应的负数。

- 执行结果：通过
- 执行用时：96 ms, 在所有 Python3 提交中击败了 19.00% 的用户
- 内存消耗：14.8 MB, 在所有 Python3 提交中击败了 25.00% 的用户

```python
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        result = 0
        for i in range(32):
            mask = 1 << i
            count = 0
            for num in nums:
                if num & mask != 0:
```

```
                count += 1
            if count % 3 != 0:
                result |= mask
            if result > 2 ** 31-1:
                result -= 2 ** 32
        return result
```

5.8 格雷编码

- 题号：89
- 难度：中等
- https://leetcode-cn.com/problems/gray-code/

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。

给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。格雷编码序列必须以 0 开头。

示例 1:

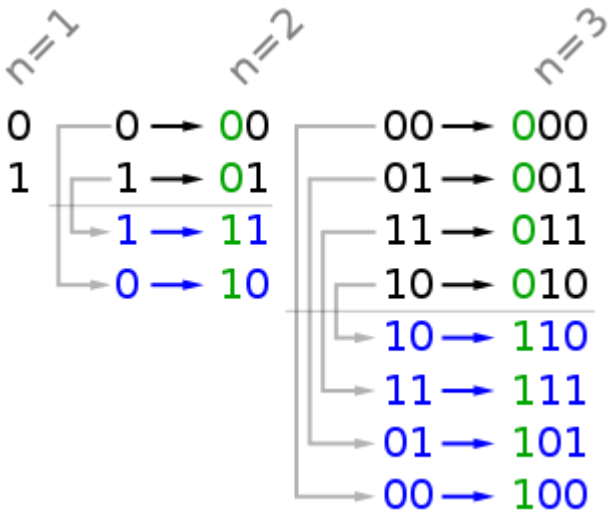
```
输入: 2
输出: [0,1,3,2]
解释:
00 - 0
01 - 1
11 - 3
10 - 2

对于给定的 n，其格雷编码序列并不唯一。
例如，[0,2,3,1] 也是一个有效的格雷编码序列。

00 - 0
10 - 2
11 - 3
01 - 1
```

示例 2:

```
输入: 0
输出: [0]
解释: 我们定义格雷编码序列必须以 0 开头。
    给定编码总位数为 n 的格雷编码序列，其长度为 2^n。
    当 n = 0 时，长度为 2^0 = 1。
```

因此，当 n = 0 时，其格雷编码序列为 [0]。

思路：



由 n 位推导 n+1 位结果时，n+1 位结果包含 n 位结果，同时包含 n 位结果中在高位再增加一个位 1 所形成的令一半结果，但是这一半结果需要与前一半结果镜像排列。

C# 语言

- 状态：通过
- 12 / 12 个通过测试用例
- 执行用时: 296 ms, 在所有 C# 提交中击败了 95.83% 的用户
- 内存消耗: 24.8 MB, 在所有 C# 提交中击败了 16.67% 的用户

```
public class Solution
{
    public IList<int> GrayCode(int n)
    {
        IList<int> lst = new List<int>();
        lst.Add(0);
        for (int i = 1; i <= n; i++)
        {
            for (int j = lst.Count - 1; j >= 0; j--)
            {
```

```
                int item = lst[j] + (1 << i - 1);
                lst.Add(item);
            }
        }
        return lst;
    }
}
```

Python 语言

- 执行结果：通过
- 执行用时：44 ms，在所有 Python3 提交中击败了 45.92% 的用户
- 内存消耗：13.8 MB，在所有 Python3 提交中击败了 20.00% 的用户

```
class Solution:
    def grayCode(self, n: int) -> List[int]:
        lst = [0]
        for i in range(1, n + 1):
            count = len(lst)
            for j in range(count - 1, -1, -1):
                lst.append(lst[j] + (1 << i - 1))
        return lst
```

注意：运算符的优先级

- 一元运算符优于二元运算符。如正负号。
- 先算术运算，后移位运算，最后位运算。例如 1 << 3 + 2 & 7 等价于 (1 << (3 + 2)) & 7
- 逻辑运算最后结合

6 扩展阅读

6.1 浅析 C# 语言中的扩展方法

1．扩展方法概述

扩展方法能够向现有类型“添加”方法，而无需创建新的派生类型、重新编译或以其它方式修改原始类型。

最常见的扩展方法是 LINQ 标准查询运算符，它将查询功能添加到现有的 `System.Collection.IEnumerable` 和 `Syste.Collection.Generic.IEnumerable<T>` 类型。

若要使用标准查询运算符，要先使用 `using System.Linq` 指令将它们置于范围中。然后，任何实现了 `IEnumerable<T>` 的类型都具有 `GroupBy<TSource,Tkey>`、`OrderBy<TSource,TKey>`、`Average` 等实例方法。在 `IEnumerable<T>` 类型的实例（如 `List<T>` 或 `Array`）后键入“.”时，可以在 IntelliSense 语句完成中看到这些附加方法。

```
class Program
{
    static void Main(string[] args)
    {
        int[] nums = new int[] { 10, 45, 15, 39, 21, 26 };
        List<int> result = nums.SelectMany(
            g =>
            {
                List<int> lst = new List<int>();
                if (g > 20) lst.Add(g);
                return lst;
            }
            ).ToList();
        Show(result);
        result = nums.OrderBy(g => g).ToList();
        Show(result);
    }
```

```
    static void Show(List<int> lst)
    {
        foreach (int i in lst)
        {
            Console.Write(i + "\t");
        }
        Console.WriteLine();
    }
}

// 45      39      21      26
// 10      15      21      26      39      45
```

2．扩展方法的实现与调用

扩展方法必须在非嵌套的、非泛型静态类型内部，且该方法必须是静态（static）方法，通过实例方法语法进行调用。

第一个参数 指定该方法所操作的类型，并且该参数以 `this` 修饰符为前缀。仅当使用 `using` 指令将命名空间显式导入到源代码中之后，扩展方法才位于范围中。

下面示例演示如何为 `System.String` 类定义一个扩展方法。

程序代码如下：

```
namespace CustomExtensions
{
    //定义一个静态类以包含扩展方法
    public static class MyExtensions
    {
        public static int WordCount(this string str)
        {
            int result = str.Split(new char[] {' ', '.', '?'},
                StringSplitOptions.RemoveEmptyEntries).Length;
            return result;
        }
    }
}

//在调用代码中，添加一条 using 指令以指定包含扩展方法类的命名空间。
using CustomExtensions;
```

```
namespace Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            //按照与调用类型上的实例方法一样的方式调用扩展方法。
            int i = s.WordCount();
            Console.WriteLine("World count of s is {0}", i);
        }
    }
}
// World count of s is 9
```

3. 为枚举创建新方法

可以使用扩展方法对枚举类型进行扩展。

程序代码如下：

```
public enum Grades
{
    A = 4,
    B = 3,
    C = 2,
    D = 1,
    F = 0
}

public static class Extensions
{
    public static Grades MinPassing = Grades.D;

    public static bool Passing(this Grades grade)
    {
        return grade >= MinPassing;
    }
}

class Program
{
    static void Main(string[] args)
```

```
    {
        Grades g1 = Grades.D;
        Grades g2 = Grades.F;
        Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is
not");
        Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is
not");

        Extensions.MinPassing = Grades.C;
        Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is
not");
        Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is
not");
    }
}

// First is a passing grade.
// Second is not a passing grade.
// First is not a passing grade.
// Second is not a passing grade.
```

4．在编译时绑定扩展方法的规则

可以使用扩展方法来扩展类或接口，但不能重写（override）扩展方法。与接口或类方法具有相同名称和签名的扩展方法永远不会被调用。

编译时，扩展方法的优先级总是比类型本身中定义的实例方法低。当编译器遇到方法调用时，它首先在该类型的实例方法中寻找匹配的方法。如果未找到任何匹配方法，编译器将搜索为该类型定义的任何扩展方法，并且绑定到它找到的第一个扩展方法。

程序代码如下：

```
namespace DefineIMyInterface
{
    public interface IMyInterface
    {
        void MethodB();
    }
}
```

```
namespace Extensions
{
    using DefineIMyInterface;
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, int
i)");
        }
        public static void MethodA(this IMyInterface myInterface,string s)
        {
            Console.WriteLine("Extension.MethodA(this IMyInterface myInterface,
string s)");
        }
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

namespace Sample
{
    using DefineIMyInterface;
    using Extensions;

    class A : IMyInterface
    {
        public void MethodB()
        {
            Console.WriteLine("A.MethodB()");
        }
    }

    class B : IMyInterface
    {
        public void MethodB()
        {
            Console.WriteLine("B.MethodB()");
        }
        public void MethodA(int i)
        {
```

```
            Console.WriteLine("B.MethodA(int i)");
        }
    }

    class C : IMyInterface
    {
        public void MethodB()
        {
            Console.WriteLine("C.MethodB()");
        }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            A a = new A();
            B b = new B();
            C c = new C();

            a.MethodA(1);
            a.MethodA("hello");
            a.MethodB();
            Console.WriteLine("----------");

            b.MethodA(1);
            b.MethodB();
            b.MethodA("hello");
            Console.WriteLine("----------");

            c.MethodA(1);
            c.MethodA("hello");
            c.MethodB();
        }
    }
}

// Extension.MethodA(this IMyInterface myInterface, int i)
// Extension.MethodA(this IMyInterface myInterface, string s)
// A.MethodB()
```

```
// ----------
// B.MethodA(int i)
// B.MethodB()
// Extension.MethodA(this IMyInterface myInterface, string s)
// ----------
// C.MethodA(object obj)
// C.MethodA(object obj)
// C.MethodB()
```

6.2 浅析 C# Dictionary 实现原理

对于 C# 中的 Dictionary 类 相信大家都不陌生，这是一个 Collection(集合) 类型，可以通过 Key/Value (键值对) 的形式来存放数据；该类最大的优点就是它查找元素的时间复杂度接近 O(1)，实际项目中常被用来做一些数据的本地缓存，提升整体效率。

那么是什么样的设计能使得 Dictionary 类 能实现 O(1) 的时间复杂度呢？那就是本篇文章想和大家讨论的东西；这些都是个人的一些理解和观点，如有表述不清楚、错误之处，请大家批评指正，共同进步。

理论知识

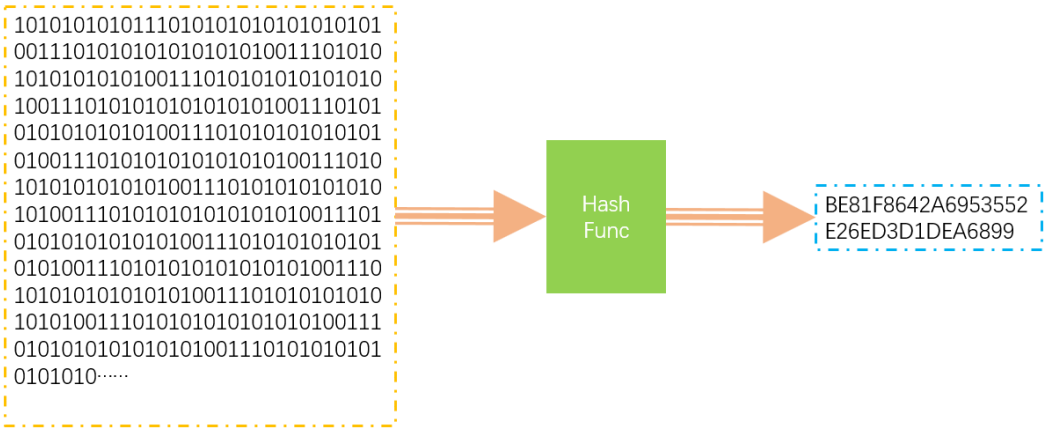
对于 Dictionary 的实现原理，其中有两个关键的算法，一个是 Hash 算法，一个是用于应对 Hash 碰撞冲突解决算法。

1、Hash 算法

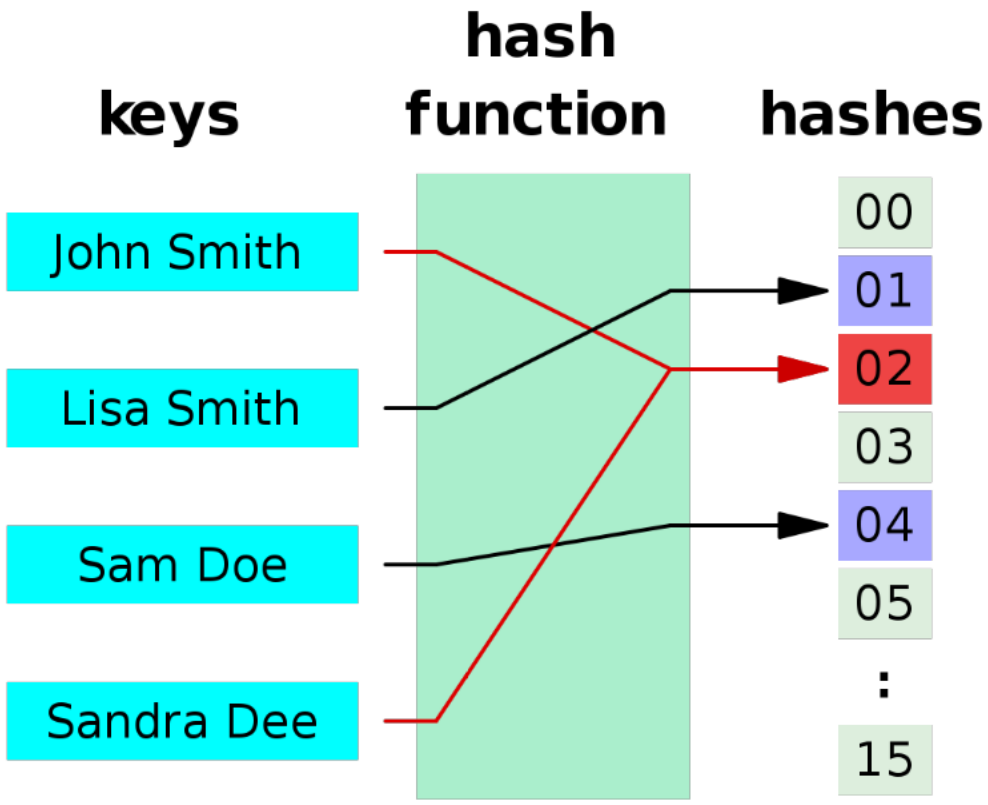
Hash 算法是一种数字摘要算法，它能将不定长度的二进制数据集给映射到一个较短的二进制长度数据集，常见的 MD5 算法就是一种 Hash 算法，通过 MD5 算法可对任何数据生成数字摘要。而实现了 Hash 算法的函数叫做 Hash 函数。Hash 函数有以下几点特征。

1. 相同的数据进行 Hash 运算，得到的结果一定相同。HashFunc(key1) == HashFunc(key1)。
2. 不同的数据进行 Hash 运算，其结果也可能会相同，(Hash 会产生碰撞)。key1 != key2 => HashFunc(key1) == HashFunc(key2)。
3. Hash 运算是不可逆的，不能由 hashCode 获取原始的数据。key1 => hashCode 但是 hashCode =\=> key1。

下图就是 Hash 函数的一个简单说明，任意长度的数据通过 HashFunc 映射到一个较短的数据集中。



关于 Hash 碰撞下图很清晰的就解释了，可从图中得知 Sandra Dee 和 John Smith 通过 hash 运算后都落到了 02 的位置，产生了碰撞和冲突。



2、Hash 桶算法

说到 Hash 算法大家就会想到 Hash 表，一个 Key 通过 Hash 函数运算后可快速的得到 hashCode，通过 hashCode 的映射可直接 Get 到 Value，但是 hashCode 一般取值都是非常大的，经常是 2^32 以上，不可能对每个 hashCode 都指定一个映射。

因为这样的一个问题，所以人们就将生成的 HashCode 以分段的形式来映射，把每一段称之为一个 Bucket（桶），一般常见的 Hash 桶就是直接对结果取余。

假设将生成的 hashCode 可能取值有 2^32 个，然后将其切分成一段一段，使用 8 个桶来映射，那么就可以通过 bucketIndex = HashFunc(key1) % 8 这样一个算法来确定这个 hashCode 映射到具体的哪个桶中。

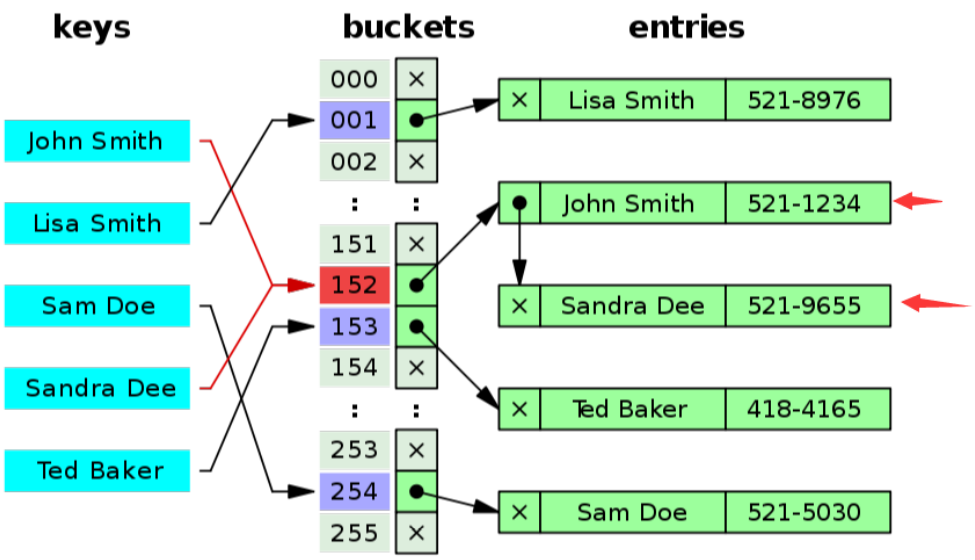
大家可以看出来，通过 hash 桶这种形式来进行映射，所以会加剧 hash 的冲突。

3、解决冲突算法

对于一个 hash 算法，不可避免的会产生冲突，那么产生冲突以后如何处理，是一个很关键的地方，目前常见的冲突解决算法为拉链法(`Dictionary` 实现采用)。

拉链法：这种方法的思路是将产生冲突的元素建立一个单链表，并将头指针地址存储至 hash 表对应桶的位置。这样定位到 hash 表桶的位置后可通过遍历单链表的形式来查找元素。

对于拉链法有一张图来描述，通过在冲突位置建立单链表，来解决冲突。



hashCode 的作用：查找的快捷性。

比如我们有一个能存放 1000 个数的内存，在其中要存放 1000 个不一样的数字，用最笨的方法，就是存一个数字，就遍历一遍，看有没有相同的数，当存了 900 个数字，开始存 901 个数字的时候，就需要跟 900 个数字进行对比，这样就很麻烦，很是消耗时间。

可以用 hashcode 来记录对象的位置。hash 表中有 1、2、3、4、5、6、7、8 个位置，存第一个数，hashcode 为 1，该数就放在 hash 表中 1 的位置，存到 100 个数字，hash 表中 8 个位置会有很多数字了，1 中可能有 20 个数字，存 101 个数字时，先查 hashcode 值对应的位置，假设为 1，那么就有 20 个数字和它的 hashcode 相同，它只需要跟这 20 个数字相比较(equals)，如果没有一个相同，那么就放在 1 这个位置，这样比较的次数就少了很多，实际上 hash 表中有很多位置，这里只是举例只有 8

个，所以比较的次数会让你觉得也挺多的，实际上，如果 hash 表很大，那么比较的次数就很少很少了。

Dictionary 实现

`Dictionary` 实现我们主要对照源码来解析，目前对照源码的版本是 .Net Framwork 4.7。

源码地址：

https://referencesource.microsoft.com/#mscorlib/system/collections/generic/dictionary.cs,d3599058f8d79be0

1. Entry 结构体

首先我们引入 `Entry` 这样一个结构体，它的定义如下代码所示。这是 `Dictionary` 中存放数据的最小单位，调用 `Add(Key,Value)` 方法添加的元素都会被封装在这样的一个结构体中。

```
private struct Entry {
    public int hashCode;    // 除符号位以外的 31 位 hashCode 值，如果该 Entry 没有被使用，那么为-1
    public int next;        // 下一个元素的下标索引，如果没有下一个就为-1
    public TKey key;        // 存放元素的键
    public TValue value;    // 存放元素的值
}
```

2. 其它关键私有变量

除了 Entry 结构体外，还有几个关键的私有变量，其定义和解释如下代码所示。

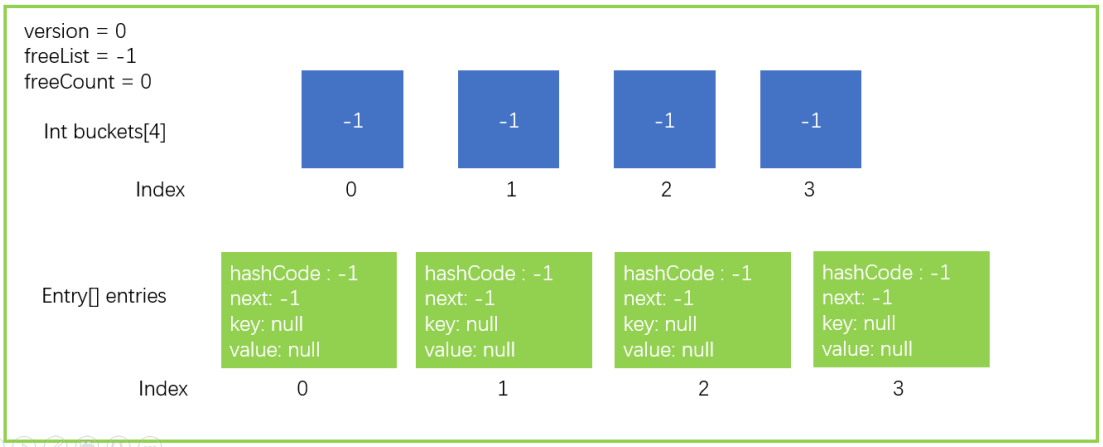
```
private int[] buckets;               // Hash 桶
private Entry[] entries;    // Entry 数组，存放元素
private int count;                   // 当前 entries 的 index 位置
private int version;                 // 当前版本，防止迭代过程中集合被更改
private int freeList;                // 被删除 Entry 在 entries 中的下标 index，这个位置是空闲的
private int freeCount;               // 有多少个被删除的 Entry，有多少个空闲的位置
private IEqualityComparer<TKey> comparer;      // 比较器
private KeyCollection keys;          // 存放 Key 的集合
```

```
private ValueCollection values;                    // 存放 Value 的集合
```

上面代码中，需要注意的是 `buckets`、`entries` 这两个数组，这是实现 `Dictionary` 的关键。

3. Dictionary - Add 操作

首先我们用图的形式来描述一个 `Dictionary` 的数据结构，其中只画出了关键的地方。桶大小为 4 以及 `Entry` 大小也为 4 的一个数据结构。



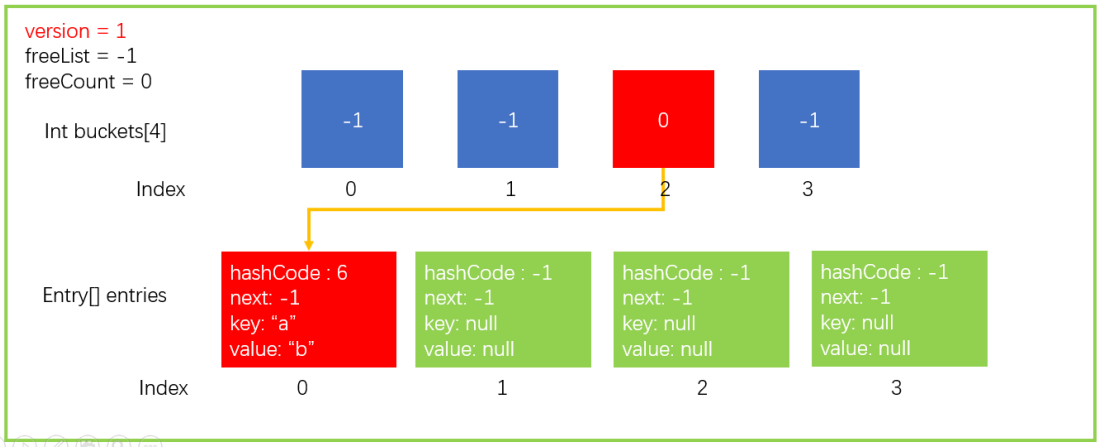
然后我们假设需要执行一个 `Add` 操作，`dictionary.Add("a","b")`，其中 `key = "a",value = "b"`。

1. 根据 key 的值，计算出它的 hashCode。我们假设 "a" 的 hash 值为 6（GetHashCode("a") = 6）。
2. 通过对 hashCode 取余运算，计算出该 hashCode 落在哪一个 buckets 桶中。现在桶的长度（buckets.Length）为 4，那么就是 6 % 4 最后落在 index 为 2 的桶中，也就是 buckets[2]。
3. 避开一种其它情况不谈，接下来它会将 hashCode、key、value 等信息存入 entries[count] 中，因为 count 位置是空闲的；继续 count++ 指向下一个空闲位置。上图中第一个位置，index = 0 就是空闲的，所以就存放在 entries[0] 的位置。

4. 将 Entry 的下标 entryIndex 赋值给 buckets 中对应下标的 bucket。步骤 3 中是存放在 entries[0] 的位置，所以 buckets[2]=0。
5. 最后 version++，集合发生了变化，所以版本需要 +1。只有增加、替换和删除元素才会更新版本。

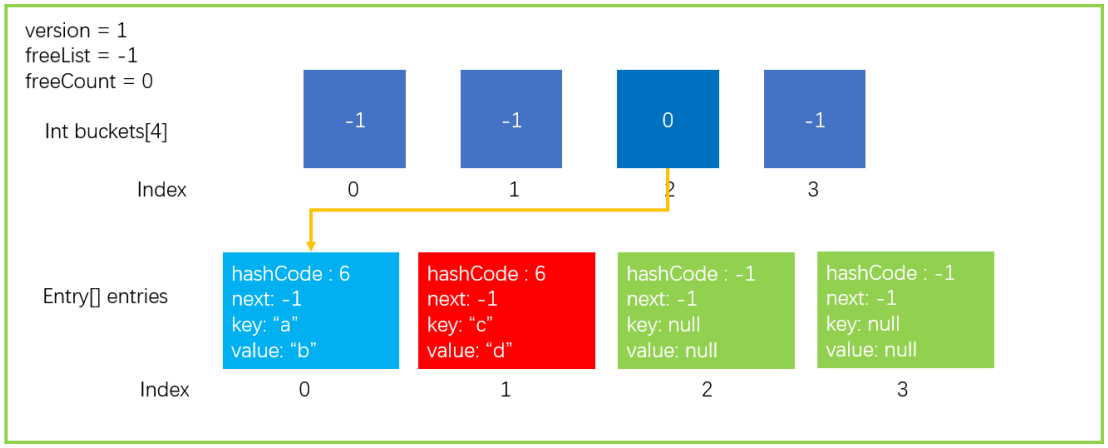
上文中的步骤 1~5 只是方便大家理解，实际上有一些偏差，后文再谈 Add 操作小节中会补充。

完成上面 Add 操作后，数据结构更新成了下图这样的形式。



这样是理想情况下的操作，一个 bucket 中只有一个 hashCode 没有碰撞的产生，但是实际上是会经常产生碰撞；那么 Dictionary 类中又是如何解决碰撞的呢。

我们继续执行一个 Add 操作，dictionary.Add("c","d")，假设 GetHashCode("c")=6，最后 6 % 4 = 2。最后桶的 index 也是 2，按照之前的步骤 1~3 是没有问题的，执行完后数据结构如下图所示。

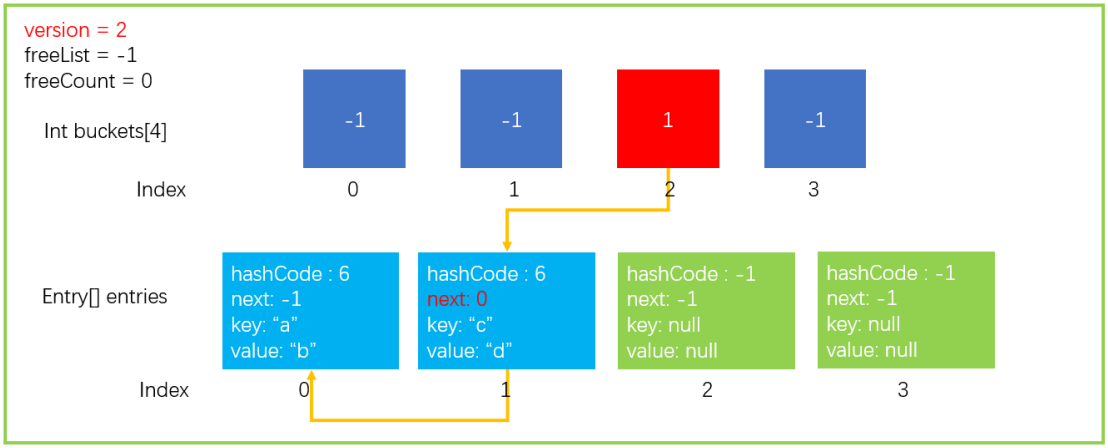


如果继续执行 步骤 4 那么 `buckets[2] = 1`，然后原来的 `buckets[2]=>entries[0]` 的关系就会丢失，这是我们不愿意看到的。现在 `Entry` 中的 `next` 就发挥大作用了。

如果对应的 `buckets[index]` 有其它元素已经存在，那么会执行以下两条语句，让新的 `entry.next` 指向之前的元素，让 `buckets[index]` 指向现在的新的元素，就构成了一个单链表。

```
entries[index].next = buckets[targetBucket];
...
buckets[targetBucket] = index;
```

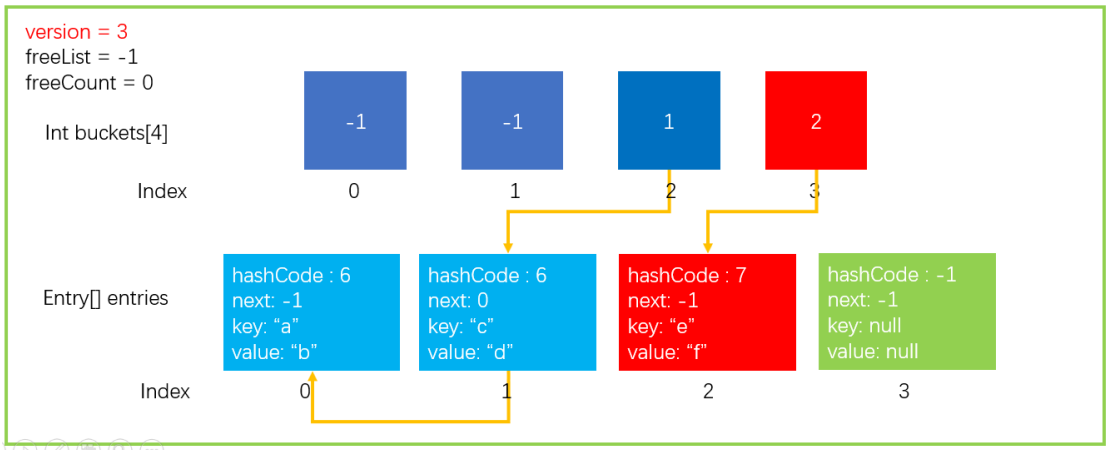
经过上面的步骤以后，数据结构就更新成了下图这个样子。



4. Dictionary - Find 操作

为了方便演示如何查找，我们继续 `Add` 一个元素 `dictionary.Add("e","f")`，

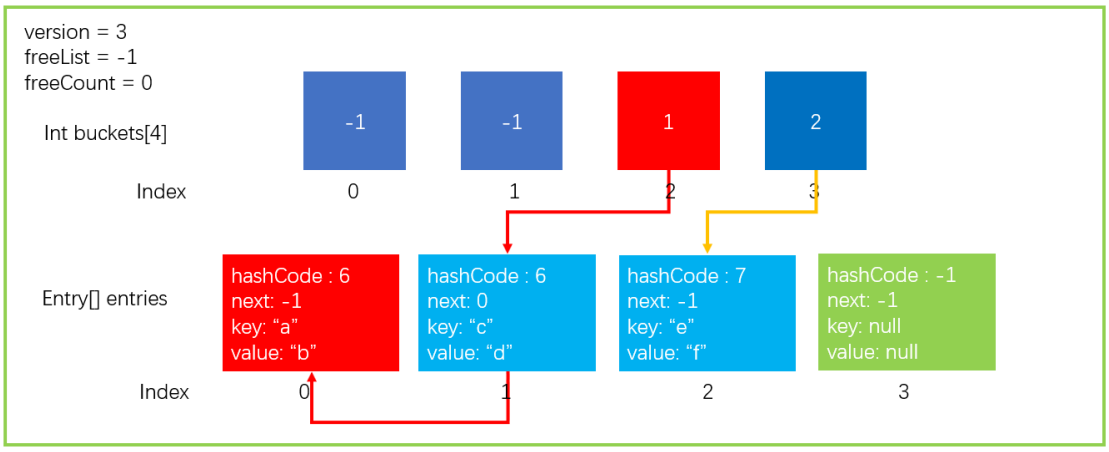
`GetHashCode("e") = 7;, 7% buckets.Length=3,` 数据结构如下所示。



假设我们现在执行这样一条语句 `dictionary.GetValueOrDefault("a")`，会执行以下步骤.

1. 获取 key 的 hashCode，计算出所在的桶位置。我们之前提到，"a" 的 hashCode=6，所以最后计算出来 targetBucket=2。
2. 通过 buckets[2]=1 找到 entries[1]，比较 key 的值是否相等，相等就返回 entryIndex，不相等就继续 entries[next] 查找，直到找到与 key 相等的元素或者 next == -1 的时候。这里我们找到了 key == "a" 的元素，返回 entryIndex =0。
3. 如果 entryIndex >= 0 那么返回对应的 entries[entryIndex] 元素，否则返回 default(TValue)。这里我们直接返回 entries[0].value。

整个查找的过程如下图所示：



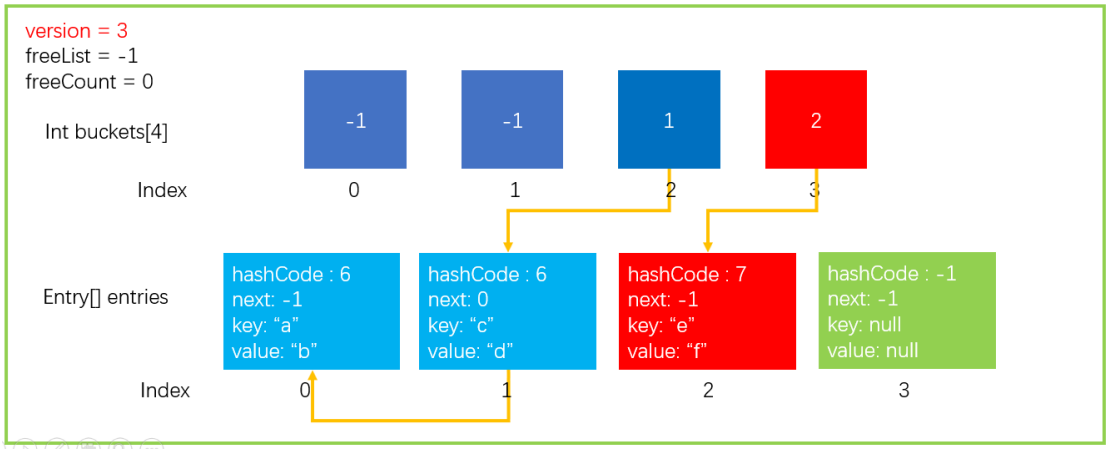
将查找的代码摘录下来，如下所示：

```
// 寻找 Entry 元素的位置
private int FindEntry(TKey key)
{
    if( key == null)
    {
        ThrowHelper.ThrowArgumentNullException(ExceptionArgument.key);
    }

    if (buckets != null)
    {
        int hashCode = comparer.GetHashCode(key) & 0x7FFFFFFF; // 获取 HashCode，忽略符号位
        // int i = buckets[hashCode % buckets.Length] 找到对应桶，然后获取 entry 在 entries 中位置
        // i >= 0; i = entries[i].next 遍历单链表
        for (int i = buckets[hashCode % buckets.Length]; i >= 0; i = entries[i].next)
        {
            // 找到就返回了
            if (entries[i].hashCode == hashCode && comparer.Equals(entries[i].key, key)) return i;
        }
    }
    return -1;
}
...
internal TValue GetValueOrDefault(TKey key)
{
    int i = FindEntry(key);
    // 大于等于 0 代表找到了元素位置，直接返回 value
```

```
    // 否则返回该类型的默认值
    if (i >= 0) {
        return entries[i].value;
    }
    return default(TValue);
}
```

5. Dictionary - Remove 操作

前面已经向大家介绍了增加、查找，接下来向大家介绍 Dictionary 如何执行删除操作。我们沿用之前的 Dictionary 数据结构。



删除前面步骤和查找类似，也是需要找到元素的位置，然后再进行删除的操作。

我们现在执行这样一条语句 dictionary.Remove("a")，hashFunc 运算结果和上文中一致。

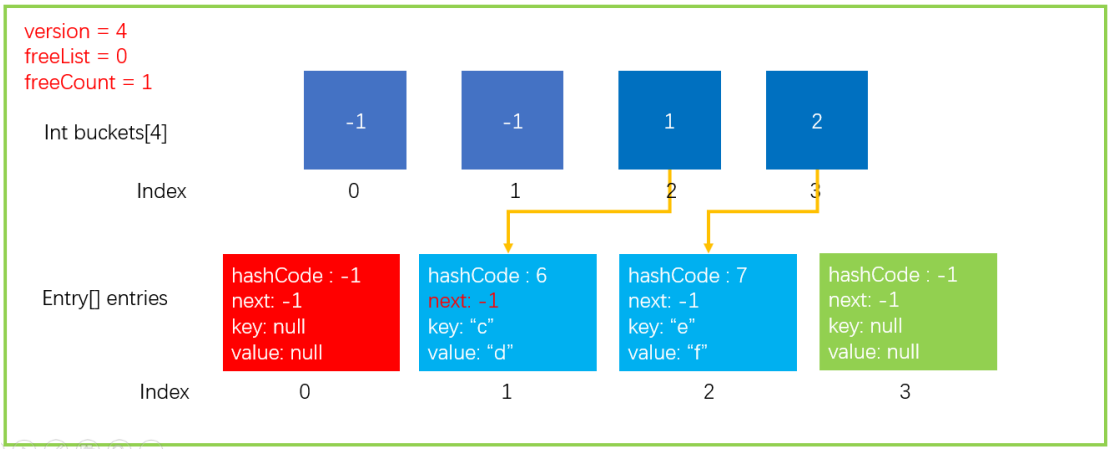
步骤大部分与查找类似，我们直接看摘录的代码，如下所示。

```
public bool Remove(TKey key)
{
    if(key == null)
    {
        ThrowHelper.ThrowArgumentNullException(ExceptionArgument.key);
    }

    if (buckets != null)
    {
        // 1. 通过 key 获取 hashCode
        int hashCode = comparer.GetHashCode(key) & 0x7FFFFFFF;
        // 2. 取余获取 bucket 位置
        int bucket = hashCode % buckets.Length;
```

```
        // last 用于确定是否当前bucket 的单链表中最后一个元素
        int last = -1;
        // 3. 遍历 bucket 对应的单链表
        for (int i = buckets[bucket]; i >= 0; last = i, i = entries[i].next)
        {
            if (entries[i].hashCode == hashCode && comparer.Equals(entries[i].key,
key))
            {
                // 4. 找到元素后，如果 last<0，代表当前是 bucket 中最后一个元素，那么直接让
bucket 内下标赋值为 entries[i].next 即可
                if (last < 0)
                {
                    buckets[bucket] = entries[i].next;
                }
                else
                {
                    // 4.1 last 不小于0，代表当前元素处于bucket 单链表中间位置，需要将该元
素的头结点和尾节点相连起来,防止链表中断
                    entries[last].next = entries[i].next;
                }
                // 5. 将 Entry 结构体内数据初始化
                entries[i].hashCode = -1;
                // 5.1 建立 freeList 单链表
                entries[i].next = freeList;
                entries[i].key = default(TKey);
                entries[i].value = default(TValue);
                // *6. 关键的代码，freeList 等于当前的 entry 位置，下一次 Add 元素会优先 Add
到该位置
                freeList = i;
                freeCount++;
                // 7. 版本号+1
                version++;
                return true;
            }
        }
    }
    return false;
}
```

执行完上面代码后，数据结构就更新成了下图所示。需要注意 version、freeList、

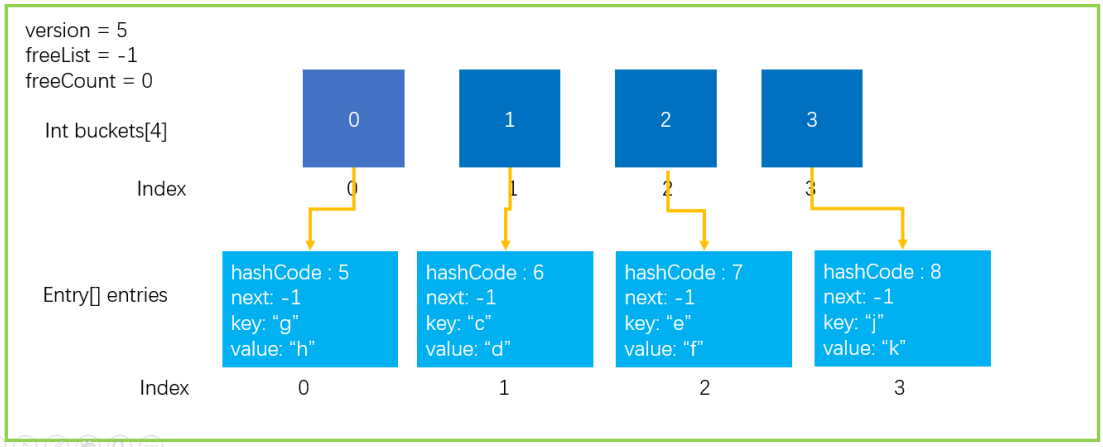
freeCount 的值都被更新了。



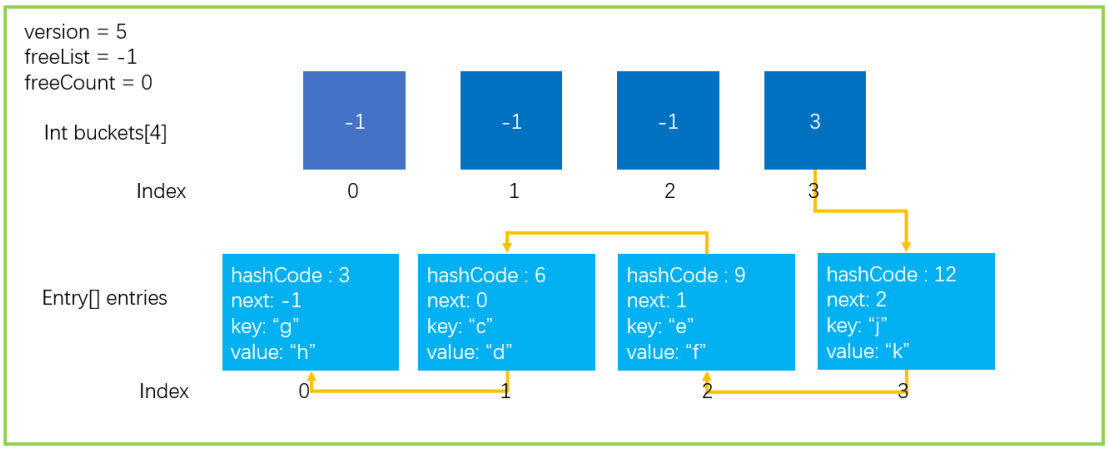
6. Dictionary - Resize 操作(扩容)

6.1 扩容操作的触发条件

首先我们需要知道在什么情况下，会发生扩容操作；第一种情况自然就是数组已经满了，没有办法继续存放新的元素。如下图所示的情况。



从上文中大家都知道，Hash 运算会不可避免的产生冲突，`Dictionary` 中使用拉链法来解决冲突的问题，但是大家看下图中的这种情况。



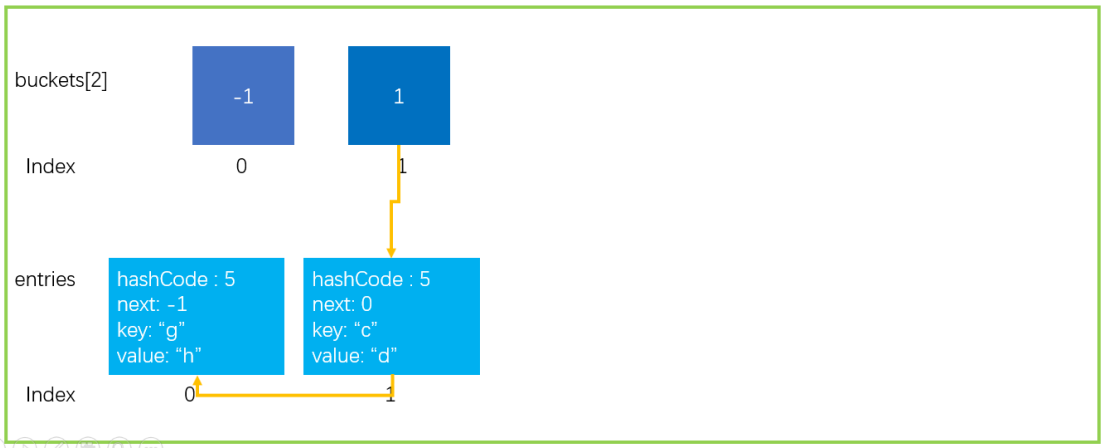
所有的元素都刚好落在 `buckets[3]`上面，结果就是导致了时间复杂度 O(n)，查找性能会下降；所以第二种，`Dictionary` 中发生的碰撞次数太多，会严重影响性能，也会触发扩容操作。

目前 .Net Framwork 4.7 中设置的碰撞次数阈值为 100。

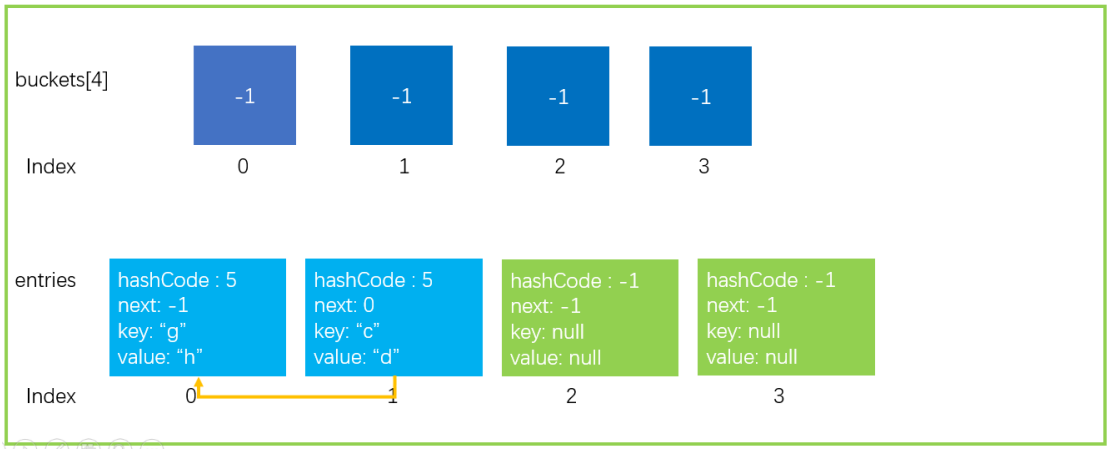
```
public const int HashCollisionThreshold = 100;
```

6.2 扩容操作如何进行

为了给大家演示的清楚，模拟了以下这种数据结构，大小为 2 的 `Dictionary`，假设碰撞的阈值为 2；现在触发 Hash 碰撞扩容。

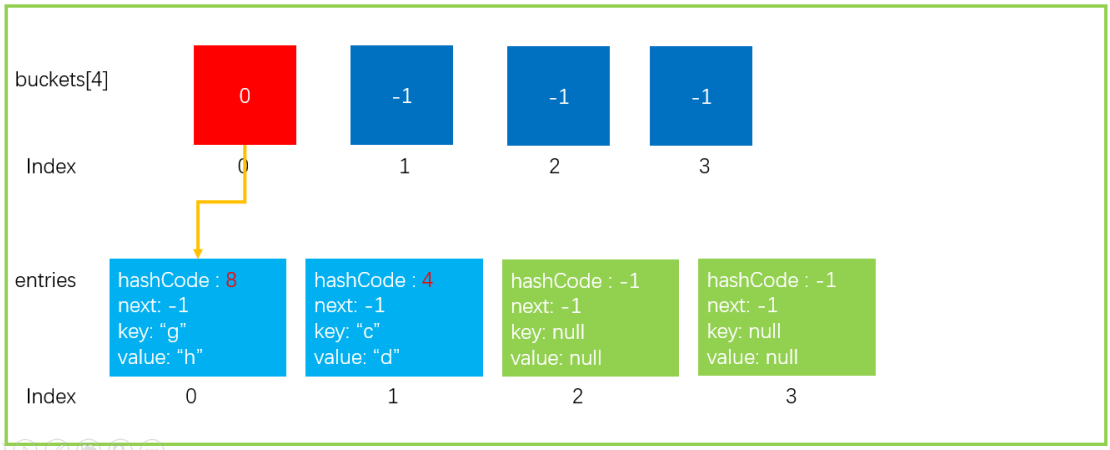


1. 申请两倍于现在大小的 buckets、entries
2. 将现有的元素拷贝到新的 entries



3. 如果是 Hash 碰撞扩容，使用新 HashCode 函数重新计算 Hash 值

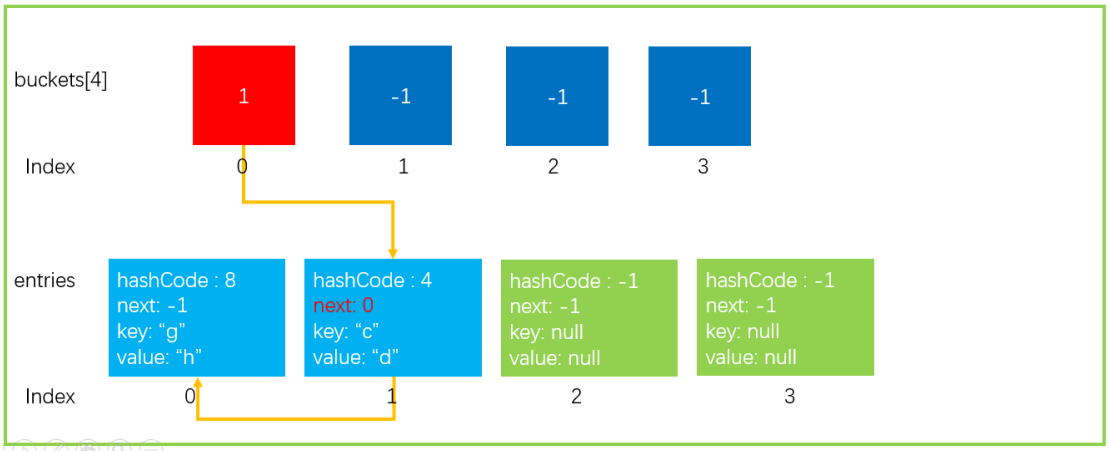
新的 Hash 函数并一定能解决碰撞的问题，有可能会更糟，像下图中一样的还是会落在同一个 bucket 上。



4. 对 entries 每个元素 bucket = newEntries[i].hashCode % newSize 确定新 buckets 位置

5. 重建 hash 链，newEntries[i].next = buckets[bucket]; buckets[bucket] = i;

因为 buckets 也扩充为两倍大小了，所以需要重新确定 hashCode 在哪个 bucket 中；最后重新建立 hash 单链表。



这就完成了扩容的操作，如果是达到 Hash 碰撞阈值触发的扩容可能扩容后结果会更差。

在 JDK 中，HashMap 如果碰撞的次数太多了，那么会将单链表转换为 **红黑树** 提升查找性能。目前 .Net Framwork 中还没有这样的优化， .Net Core 中已经有了类似的优化。

每次扩容操作都需要遍历所有元素，会影响性能。所以创建 Dictionary 实例时最好设置一个预估的初始大小。

```
private void Resize(int newSize, bool forceNewHashCodes)
{
    Contract.Assert(newSize >= entries.Length);
    // 1. 申请新的 Buckets 和 entries
    int[] newBuckets = new int[newSize];
    for (int i = 0; i < newBuckets.Length; i++) newBuckets[i] = -1;
    Entry[] newEntries = new Entry[newSize];
    // 2. 将 entries 内元素拷贝到新的 entries 总
    Array.Copy(entries, 0, newEntries, 0, count);
    // 3. 如果是 Hash 碰撞扩容，使用新 HashCode 函数重新计算 Hash 值
    if(forceNewHashCodes)
    {
        for (int i = 0; i < count; i++)
        {
            if(newEntries[i].hashCode != -1)
            {
                newEntries[i].hashCode = (comparer.GetHashCode(newEntries[i].key) &
0x7FFFFFFF);
            }
        }
    }
    // 4. 确定新的 bucket 位置
```

```
    // 5. 重建 Hahs 单链表
    for (int i = 0; i < count; i++)
    {
        if (newEntries[i].hashCode >= 0)
        {
            int bucket = newEntries[i].hashCode % newSize;
            newEntries[i].next = newBuckets[bucket];
            newBuckets[bucket] = i;
        }
    }
    buckets = newBuckets;
    entries = newEntries;
}
```

7. Dictionary - 再谈 Add 操作

在我们之前的 Add 操作步骤中，提到了这样一段话，这里提到会有一种其它的情况，那就是有元素被删除的情况。

> 3. 避开一种其它情况不谈，接下来它会将 hashCode、key、value 等信息存入 entries[count] 中，因为 count 位置是空闲的；继续 count++ 指向下一个空闲位置。上图中第一个位置，index = 0 就是空闲的，所以就存放在 entries[0] 的位置。

因为 count 是通过自增的方式来指向 entries[] 下一个空闲的 entry，如果有元素被删除了，那么在 count 之前的位置就会出现一个空闲的 entry；如果不处理，会有很多空间被浪费。

这就是为什么 Remove 操作会记录 freeList、freeCount，就是为了将删除的空间利用起来。实际上 Add 操作会优先使用 freeList 的空闲 entry 位置，摘录代码如下。

```
private void Insert(TKey key, TValue value, bool add)
{

    if( key == null )
    {
```

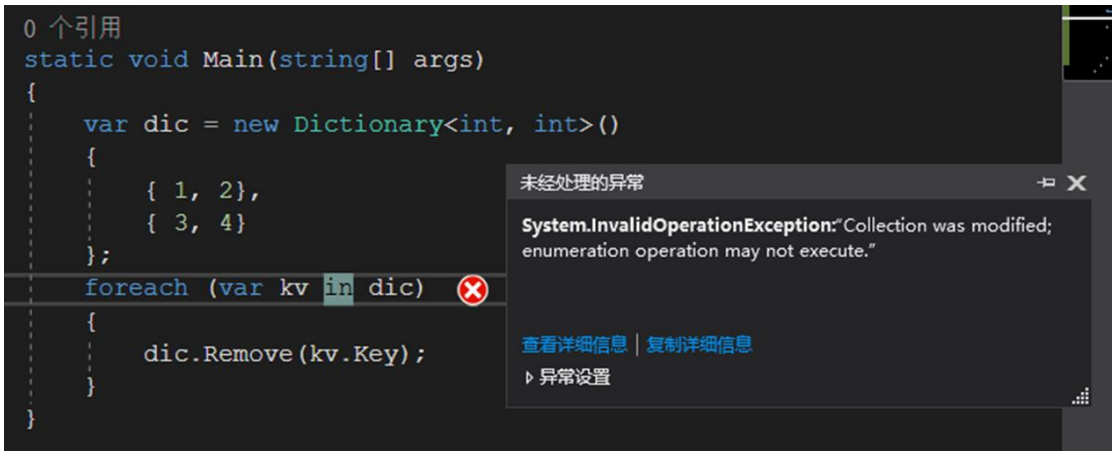
```
        ThrowHelper.ThrowArgumentNullException(ExceptionArgument.key);
    }

    if (buckets == null) Initialize(0);
    // 通过key获取hashCode
    int hashCode = comparer.GetHashCode(key) & 0x7FFFFFFF;
    // 计算出目标bucket下标
    int targetBucket = hashCode % buckets.Length;
          // 碰撞次数
    int collisionCount = 0;
    for (int i = buckets[targetBucket]; i >= 0; i = entries[i].next)
    {
        if (entries[i].hashCode == hashCode && comparer.Equals(entries[i].key, key))
        {
            // 如果是增加操作，遍历到了相同的元素，那么抛出异常
            if (add)
            {

          ThrowHelper.ThrowArgumentException(ExceptionResource.Argument_AddingDuplic
ate);
            }
            // 如果不是增加操作，那可能是索引赋值操作 dictionary["foo"] = "foo"
            // 那么赋值后版本++，退出
            entries[i].value = value;
            version++;
            return;
        }
        // 每遍历一个元素，都是一次碰撞
        collisionCount++;
    }
    int index;
    // 如果有被删除的元素，那么将元素放到被删除元素的空闲位置
    if (freeCount > 0)
    {
        index = freeList;
        freeList = entries[index].next;
        freeCount--;
    }
    else
    {
        // 如果当前entries已满，那么触发扩容
        if (count == entries.Length)
        {
            Resize();
```

```
            targetBucket = hashCode % buckets.Length;
        }
        index = count;
        count++;
    }

    // 给 entry 赋值
    entries[index].hashCode = hashCode;
    entries[index].next = buckets[targetBucket];
    entries[index].key = key;
    entries[index].value = value;
    buckets[targetBucket] = index;
    // 版本号++
    version++;

    // 如果碰撞次数大于设置的最大碰撞次数，那么触发 Hash 碰撞扩容
    if(collisionCount > HashHelpers.HashCollisionThreshold &&
HashHelpers.IsWellKnownEqualityComparer(comparer))
    {
        comparer = (IEqualityComparer<TKey>)
HashHelpers.GetRandomizedEqualityComparer(comparer);
        Resize(entries.Length, true);
    }
}
```

8. Collection 版本控制

在上文中一直提到了 `version` 这个变量，在每一次新增、修改和删除操作时，都会使 `version++`；那么这个 `version` 存在的意义是什么呢？

首先我们来看一段代码，这段代码中首先实例化了一个 `Dictionary` 实例，然后通过 `foreach` 遍历该实例，在 `foreach` 代码块中使用 `dic.Remove(kv.Key)`删除元素。



结果就是抛出了 `System.InvalidOperationException:"Collection was modified..."`这样的异常，迭代过程中不允许集合出现变化。如果在 Java 中遍历直接删除元素，会出现诡异的问题，所以 .Net 中就使用了 `version` 来实现版本控制。

那么如何在迭代过程中实现版本控制的呢？我们看一看源码就很清楚的知道。

```
[Serializable]
public struct Enumerator : IEnumerator<TValue>, System.Collections.IEnumerator
{
    private Dictionary<TKey, TValue> dictionary;
    private int index;
    private int version;
    private TValue currentValue;

    internal Enumerator(Dictionary<TKey, TValue> dictionary) {
        this.dictionary = dictionary;
        version = dictionary.version;
        index = 0;
        currentValue = default(TValue);
    }

    public void Dispose() {
    }

    public bool MoveNext() {
        if (version != dictionary.version) {
            ThrowHelper.ThrowInvalidOperationException(ExceptionResource.InvalidOperation_EnumFailedVersion);
        }
```

在迭代器初始化时，就会记录 `dictionary.version` 版本号，之后每一次迭代过程都会检查版本号是否一致，如果不一致将抛出异常。

这样就避免了在迭代过程中修改了集合，造成很多诡异的问题。