作业一：数据结构与算法基础

判断数组中所有的数字是否只出现一次。给定一个数组array，判断数组 array 中是否所有的数字都只出现过一次。例如，arr = {1, 2, 3}，输出 YES。又如，arr = {1, 2, 1}，输出 NO。约束时间复杂度为 O(n)。

分析：

我们先用常规方法解决：

遍历数组是O(n)，每个数遍历一次是O(n)，所以总的时间复杂度是$O(n^2)$

约束时间复杂度为 O(n)，不符合要求。

给定一个数组array，没有说有序，所以也不能使用二分法解决

根据题目来看，你可以理解这是一个数据去重的问题。

每轮迭代需要去判断当前元素在先前已经扫描过的区间内是否出现过，这就是一个查找的动作

在优化数值特性的查找时，我们应该想到哈希表。因为它能在 O(1) 的时间内完成查找动作。这样，整体的时间复杂度就可以被降低为 O(n) 了。与此同时，空间复杂度也提高到了 O(n)。

代码如下：

```
package com.lgedu.homework;

import java.util.HashMap;
import java.util.Map;

/**
 * 查找唯一数
 */
public class UniqNum {

    public static boolean isUnique(int[] array){
        /*
            借助HashMap的key唯一性，也可以自己实现hash函数
         */
        Map<Integer, Integer> map = new HashMap<>();
        for(int i=0;i<array.length;i++){
            if(map.containsKey(array[i])){
                return false;
            }
            //将array设置到key中，value任意
            map.put(array[i],1);
        }
        return true;
    }
    public static void main(String[] args) {
        int[] array={1,2,1};
        if(isUnique(array)){
            System.out.println("YES");
        }
        else{
            System.out.println("NO");
```

```
            }
        }
    }
```

作业二：数据结构与算法高级

很久很久以前，有一位国王拥有5座金矿，每座金矿的黄金储量不同，需要参与挖掘的工人人数也不同。例如有的金矿储量是500kg黄金，需要5个工人来挖掘；有的金矿储量是200kg黄金，需要3个工人来挖掘……

如果参与挖矿的工人的总数是10。每座金矿要么全挖，要么不挖，不能派出一半人挖取一半的金矿。要求用程序求出，要想得到尽可能多的黄金，应该选择挖取哪几座金矿？

分析：

读题会知道这是一个算法思维问题，限定范围为贪心算法（部分背包）和动态规划（0-1背包）。

每座金矿要么全挖，要么不挖：确定是0-1背包，采用动态规划法，抽象如下：

有n个金矿和一个最多参与挖坑人数W，每座金矿需要人数是w[i]，金矿储量是v[i]

在保证总重量不超过W的前提下，选择挖哪些金矿，黄金储量最大，可以求出最大值？

假设：W=10，有5个金矿，需要人数和储量如下：

w[1]=2,v[1]=60

w[2]=2,v[2]=30

w[3]=6,v[3]=50

w[4]=5,v[4]=40

w[5]=4,v[5]=60

dp数组的计算结果如下表：

i\j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1 (w[1]=2,v[1]=60)	0	0	60	60	60	60	60	60	60	60	60
2（w[2]=2,v[2]=30）	0	0	60	60	90	90	90	90	90	90	90
3（w[3]=6,v[3]=50）	0	0	60	60	90	90	90	90	110	110	140
4（w[4]=5,v[4]=40）	0	0	60	60	90	90	90	100	110	130	140
5（w[5]=4,v[5]=60）	0	0	60	60	90	90	120	120	150	150	150

i:选择i个金矿 j:最大人数

代码如下：

```
package com.lgedu.homework;
```

```java
/**
 * 0-1背包，计算最大价值  DP方程
 */
public class Gold {
    /**
     * 计算最大价值
     * @param values 物品的价值数组
     * @param weights 物品的重量数组
     * @param max 背包最大承重
     * @return
     */
    public static  int maxValue(int[] values,int[] weights,int max){
        if(values==null|| values.length==0) return 0;
        if(weights==null|| weights.length==0) return 0;
        if(values.length!=weights.length|| max<=0) return 0;

        //dp数组  dp[i-1]  i从1开始
        int[][] dp=new int[values.length+1][max+1];

        for(int i=1;i<=values.length;i++){
            for(int j=1;j<=max;j++){
                //选择的物品超过最大承重
                if(weights[i-1]>j){
                    //不能选该物品 等于上轮的最大价值
                    dp[i][j]=dp[i-1][j];

                }
                //选择的物品不超过最大承重
                else{
                    //上轮的最大价值
                    int proValue=dp[i-1][j];
                    //选择该商品后的最大价值
                    int curValue=values[i-1]+dp[i-1][j-weights[i-1]];
                    //两者取最大值
                    dp[i][j]=Math.max(proValue,curValue);

                }


            }
        }
        int mv=dp[values.length][max];
        //逆推找出装入背包的所有商品的编号（选的矿的编号）
        int j=max;
        String numStr="";
        for(int i=values.length;i>0;i--){
            //若果dp[i][j]>dp[i-1][j],这说明第i件物品是放入背包的（第i个矿挖）
            if(dp[i][j]>dp[i-1][j]){
                numStr = i+" "+numStr;
                j=j-weights[i-1];
            }
            if(j==0)
                break;
        }
        System.out.println("选择的金矿有："+numStr);

        return mv;
```

```
    }

    public static void main(String[] args) {
        int[] values={60,30,50,40,60};
        int[] weights={2,2,6,5,4};
        int max=10;
        System.out.println("挖出的最大储量是："+maxValue(values,weights,max));
    }
}
```

补充知识

1、HashMap（JDK1.8）

在JDK1.8中HashMap使用数组+链表+红黑树的数据结构

- **数组**

  数组具有遍历快，增删慢的特点。数组在堆中是一块连续的存储空间，遍历时数组的首地址是知道的（首地址=首地址+元素字节数 * 下标），所以遍历快（数组遍历的时间复杂度为O(1)）；增删慢是因为，当在中间插入或删除元素时，会造成该元素后面所有元素地址的改变，所以增删慢（增删的时间复杂度为O(n)）。

- **链表**

  链表具有增删快，遍历慢的特点。链表中各元素的内存空间是不连续的，一个节点至少包含节点数据与后继节点的引用，所以在插入删除时，只需修改该位置的前驱节点与后继节点即可，链表在插入删除时的时间复杂度为O(1)。但是在遍历时，get(n)元素时，需要从第一个开始，依次拿到后面元素的地址，进行遍历，直到遍历到第n个元素（时间复杂度为O(n)），所以效率极低。

- **红黑树**

  红黑树首先是一种树形结构，同时又是一个二叉树，为了保证树的左右孩子树相对平衡（深度相同），红黑树使用了节点标色的方式，将节点标记为红色或者黑色，在计算树的深度时只统计黑色节点的数量，不统计红色节点数量。而保持左右子树深度相同的原因是减少树的最大深度，从而提高查询的效率，尽量维持左右子树的深度一致，避免某个子树深度过深的情况出现。

Hash函数计算

将key的hashcode值再与该值的高16位进行异或运算得到最终的hash值

使用hash值与数组长度减一的值进行异或得到分散的数组下标

Hash碰撞

hash是指，两个元素通过hash函数后计算出的值是一样的，是同一个存储地址。当后面的元素要插入到这个地址时，发现已经被占用了，这时候就产生了hash冲突

hash冲突的解决方法

hashmap采用的就是链地址法，jdk1.7中，当冲突时，在冲突的地址上生成一个链表，将冲突的元素的key，通过equals进行比较，相同即覆盖，不同则添加到链表上，此时如果链表过长，效率就会大大降低，查找和添加操作的时间复杂度都为O(n)；但是在jdk1.8中如果链表长度大于8，链表就会转化为红黑树,时间复杂度也降为了O(logn)

HashMap的重要字段

```
//默认初始容量为16，0000 0001 左移4位 0001 0000为16，主干数组的初始容量为16，而且这个数组
//必须是2的倍数
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

//最大容量为int的最大值除2
static final int MAXIMUM_CAPACITY = 1 << 30;// aka 1,073,741,824

//默认加载因子为0.75
static final float DEFAULT_LOAD_FACTOR = 0.75f;

//阈值，如果主干数组上的链表的长度大于8，链表转化为红黑树
 static final int TREEIFY_THRESHOLD = 8;

//hash表扩容后，如果发现某一个红黑树的长度小于6，则会重新退化为链表
 static final int UNTREEIFY_THRESHOLD = 6;

//当hashmap容量大于64时，链表才能转成红黑树
 static final int MIN_TREEIFY_CAPACITY = 64;

//临界值=主干数组容量*负载因子
int threshold;
```

HashMap的Put方法流程

1、put(key, value)中直接调用了内部的putVal方法，并且先对key进行了hash操作；

2、putVal方法中，先检查HashMap数据结构中的索引数组表是否位空，如果是的话则进行一次resize操作；

3、以HashMap索引数组表的长度减一与key的hash值进行与运算，得出在数组中的索引，如果索引指定的位置值为空，则新建一个k-v的新节点；

4、如果不满足的3的条件，则说明索引指定的数组位置的已经存在内容，这个时候称之**碰撞出现**；

5、在上面判断流程走完之后，计算HashMap全局的modCount值，以便**对外部并发的迭代操作提供修改的Fail-fast判断提供依据**，于此同时增加map中的记录数，并判断记录数是否触及容量扩充的阈值，触及则进行一轮resize操作；

6、在步骤4中出现碰撞情况时，从步骤7开始展开新一轮逻辑判断和处理；

7、判断key索引到的节点（暂且称作被碰撞节点）的hash、key是否和当前待插入节点（新节点）的一致，如果是一致的话，则先保存记录下该节点；如果新旧节点的内容不一致时，则再看被碰撞节点是否是树（TreeNode）类型，如果是树类型的话，则按照树的操作去追加新节点内容；如果被碰撞节点不是树类型，则说明当前发生的碰撞在链表中（此时链表尚未转为红黑树），此时进入一轮循环处理逻辑中；

8、循环中，先判断被碰撞节点的后继节点是否为空，为空则将新节点作为后继节点，作为后继节点之后并判断当前链表长度是否超过最大允许链表长度8，如果大于的话，需要进行一轮是否转树的操作；如果在一开始后继节点不为空，则先判断后继节点是否与新节点相同，相同的话就记录并跳出循环；如果两个条件判断都满足则继续循环，直至进入某一个条件判断然后跳出循环；

9、步骤8中转树的操作treeifyBin，如果map的索引表为空或者当前索引表长度还小于**64（最大转红黑树的索引数组表长度）**，那么进行resize操作就行了；否则，如果被碰撞节点不为空，那么就顺着被碰撞节点这条树往后新增该新节点；

10、最后，回到那个被记住的被碰撞节点，如果它不为空，默认情况下，新节点的值将会替换被碰撞节点的值，同时返回被碰撞节点的值（V）。

```java
/**
 * Associates the specified value with the specified key in this map.
 * If the map previously contained a mapping for the key, the old
 * value is replaced.
 *
 * @param key key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 * @return the previous value associated with <tt>key</tt>, or
 *         <tt>null</tt> if there was no mapping for <tt>key</tt>.
 *         (A <tt>null</tt> return can also indicate that the map
 *         previously associated <tt>null</tt> with <tt>key</tt>.)
 */
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}


final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // table未初始化或者长度为0，进行扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // (n - 1) & hash 确定元素存放在哪个桶中，桶为空，新生成结点放入桶中(此时，这个结点是放在数组中)
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // 桶中已经存在元素
    else {
        Node<K,V> e; K k;
        // 比较桶中第一个元素(数组中的结点)的hash值相等，key相等
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
                // 将第一个元素赋值给e，用e来记录
                e = p;
        // hash值不相等，即key不相等；为红黑树结点
        else if (p instanceof TreeNode)
            // 放入树中
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 为链表结点
        else {
            // 在链表最末插入结点
            for (int binCount = 0; ; ++binCount) {
                // 到达链表的尾部
                if ((e = p.next) == null) {
                    // 在尾部插入新结点
                    p.next = newNode(hash, key, value, null);
                    // 结点数量达到阈值，转化为红黑树
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    // 跳出循环
                    break;
                }
                // 判断链表中结点的key值与插入的元素的key值是否相等
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    // 相等，跳出循环
                    break;
```

```
                // 用于遍历桶中的链表，与前面的e = p.next组合，可以遍历链表
                p = e;
            }
        }
        // 表示在桶中找到key值、hash值与插入元素相等的结点
        if (e != null) {
            // 记录e的value
            V oldValue = e.value;
            // onlyIfAbsent为false或者旧值为null
            if (!onlyIfAbsent || oldValue == null)
                //用新值替换旧值
                e.value = value;
            // 访问后回调
            afterNodeAccess(e);
            // 返回旧值
            return oldValue;
        }
    }
    // 结构性修改
    ++modCount;
    // 实际大小大于阈值则扩容
    if (++size > threshold)
        resize();
    // 插入后回调
    afterNodeInsertion(evict);
    return null;
}
```