

数据结构习题

一. 数组

1. 给你一个整数数组 `nums` 。如果任一值在数组中出现至少两次，返回 `true`；如果数组中每个元素互不相同，返回 `false`。

1) 解法一

使用冒泡进行暴力破解，这种方法的时间复杂度为 $O(n^2)$ 。

```
public bool ContainsDuplicate(int[] nums)
{
    for (int i = 0; i < nums.Length-1; i++)
    {
        for (int j = i + 1; j < nums.Length; j++)
        {
            if (nums[i] == nums[j])//当有数值相同时就返回 true
            {
                return true;
            }
        }
    }
    return false;
}
```

2) 解法二

将数组排序后再进行比较，时间复杂度 $O(n)$ 。

```
public bool ContainsDuplicate(int[] nums)
{
    Array.Sort(nums);//对数组排序
    for (int i = 0; i < nums.Length - 1; i++)
    {
        if (nums[i] == nums[i + 1])//和后一个进行比较
        {
            return true;
        }
    }
    return false;
}
```

3) 解法三

哈希表是根据关键字而直接进行访问的数据结构，它建立了关键字和存储地址之间的一种直接映射关系，利用哈希表的特点我们可以去判断数组中是否有重复元素。

```
public bool ContainsDuplicate(int[] nums)
{
    HashSet<int> intSet = new HashSet<int>(nums);//设置数组中每一个数值的哈希值
    if (intSet.Count != nums.Length)//哈希表中的元素数，是否等于数组的长度
        return true;//因为一个数值只对应一个哈希值，因此有重复元素时，哈希表中
```

数据会小于数组长度

```
else
    return false;
```

```
}
```

2. 数组里有 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}，请随机打乱顺序，生成一个新的数组。

主要是运用[洗牌算法](#)，将数组打乱。

1) Fisher-Yates shuffle

```
private void FisherList<T>(ref T[] dataArray)
{
    List<T> dataList = new List<T>(dataArray);
    List<T> cache = new List<T>();//构建
    while (dataList.Count > 0)
    {
        int randomIndex = UnityEngine.Random.Range(0, dataList.Count);//随机获得一个数组下标
        cache.Add(dataArray[randomIndex]);//将数组添加到 List 中
        dataList.RemoveAt(randomIndex);//将随机下标对应的元素移除出 List
    }
    dataArray = cache.ToArray();//将 List 转换为数组
}
```

2) Knuth-Durstenfeld Shuffle

```
private void KnuthList<T>(ref T[] dataArray)
{
    for (int i = 0; i < dataArray.Length - 1; i++)
    {
        int randomIndex = UnityEngine.Random.Range(i, dataArray.Length);
        T temp = dataArray[randomIndex];
        dataArray[randomIndex] = dataArray[i];
        dataArray[i] = temp;
    }
}
```

3. 给你一个整数数组 nums，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。（子数组是数组中的一个连续部分）。

1) 解法一——使用贪心算法

当我们从头到尾遍历这个数组的时候，对于数组里的一个整数，它有几种选择呢？它只有两种选择，**一个**是和之前的数值合并，**另一个**是以自己作为起始点，新开一个子数组。

如果之前**子数组的总体和大于 0**的话，我们认为其对后续结果是有贡献的。这种情况下我们将值加入之前的子数组。

如果之前**子数组的总体和为 0 或者小于 0**的话，我们认为其对后续结果是没有贡献，甚至是有害的（小于 0 时）。这种情况下我们选择以这个数字开始，另起一个子数组。

时间复杂度 $O(n)$

空间复杂度 $O(1)$

```
private int MaxSubArray(int[] nums)
{
    int sum = 0; //用于计算子数组中的和
    var result = nums[0]; //将数组第一个元素作为默认返回结果
    for (int i = 0; i < nums.Length; i++)
    {
        if (sum + nums[i] > nums[i]) //意味着在这个数字之前总和 > 0 我们可以使用它，否则我们只使用当前数字
        {
            sum += nums[i];
        }
        else
        {
            sum = nums[i]; //否则我们只使用当前数值，重新开始统计。
        }
        /*可以将前面的部分换成
        * sum = Math.Max(nums[i], sum + nums[i]);
        */

        result = sum > result ? sum : result; //保存当前数字或总和值
    }

    return result;
}
```

2) 动态规划

若前面元素和大于 0，则将其和当前元素相加。若前面元素和小于 0，就让元素和等于当前元素，重新进行统计。

```
public int MaxSubArray(int[] nums)
{
    int sum = nums[0]; //当前连续子数组的和
    int max = nums[0]; //当前和最大的连续子数组
    for (int i = 1; i < nums.Length; i++)
    {
        if (nums[i] > sum + nums[i]) //当前连续子数组的和小于当前数，表示之前的连续子数组和小于 0
        {
            sum = nums[i];
        }
        else //当前连续子数组的和大于当前数
        {
            sum = sum + nums[i];
        }
    }
}
```

```

        if (sum > max)
        {
            max = sum;
        }
    }
    return max;
}

```

4. 给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出和为目标值 `target` 的那两个整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。

1) 解法一——暴力法

暴力遍历数组中所有可能，记录下符合的下标的值。其时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ 。

```

public int[] TwoSum(int[] nums, int target)
{
    int[] n = new int[2];
    for (int i = 0; i < nums.Length; i++)
    {
        for (int j = i + 1; j < nums.Length; j++)
        {
            if (target == nums[i] + nums[j])
            {
                //所得值和目标值相等，就返回下标
                n[0] = i;
                n[1] = j;
            }
        }
    }
    return n;
}

```

2) 解法二——哈希

哈希表可以用空间来换取时间，将查找时间从 $O(n)$ 降低为 $O(1)$ 。哈希表正是为此目的而构建的，它支持以近似恒定的时间进行快速查找。这里用“近似”来描述，是因为一旦出现冲突，查找用时可能会退化到 $O(n)$ 。但只要仔细的挑选哈希函数，在哈希表中进行查找的用时应当被摊销为 $O(1)$ 。

散列表（Hash table，也叫哈希表），是根据关键码值（Key value）而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

给定表 M ，存在函数 $f(\text{key})$ ，对任意给定的关键字值 key ，代入函数后若能得到包含该关键字的记录在表中的地址，则称表 M 为哈希（Hash）表，函数 $f(\text{key})$ 为哈希（Hash）函数。

```

public int[] TwoSum(int[] nums, int target)
{
    Dictionary<int, int> arrayDictionary = new Dictionary<int, int>();
    for (int i = 0; i < nums.Length; i++)
    {
        if (arrayDictionary.ContainsKey(target - nums[i]))//检查数据字典中是否有
该关键字
        {
            return new int[] { arrayDictionary[target - nums[i]], i };
        }
        else
        {
            arrayDictionary[nums[i]] = i;
            //net4.0 以后就可以直接赋值了，而且当出现相同的键值时会直接替
换，反而如果在这里使用 add 方法当如果有重复值时，如果不判断的时候就会报错。
        }
    }
    return new int[] { 0, 0 };
}

```

5. 给你两个按**非递减顺序排列**的整数数组 `nums1` 和 `nums2`，另有两个整数 `m` 和 `n`，分别表示 `nums1` 和 `nums2` 中的元素数目。请你合并 `nums2` 到 `nums1` 中，使合并后的数组同样按 **非递减顺序排列**。

注意：最终，合并后数组不应由函数返回，而是存储在数组 `nums1` 中。为了应对这种情况，`nums1` 的初始长度为 `m + n`，其中前 `m` 个元素表示应合并的元素，后 `n` 个元素为 0，应忽略。`nums2` 的长度为 `n`。

二. 计算机组成原理

1. 写出代码判断一个整数是不是 2 的 N 次方。

1) 解法一——循环除 2

我们知道，2 的幂是一定可以被 2 整除的。我们只需要让其和每一个 2 的幂值比较，直到相等为止，也就是采用穷举法。

```

private void NumberIsTwoPower(int number)
{
    int i = 0;
    while (i <= number)
    {
        i *= 2;
        if (i == number)
        {
            Debug.Log("是 2 的幂次");
            return;
        }
    }
}

```

```

    }
}
Debug.Log("不是 2 的幂次");
return;
}

```

2) 解法二——按位与运算判断

其实一个数 n ，如果是 2 的幂次方数，则 n 的二进制的补码中一定只有一个 1，那 $n \& (n-1)$ 就会让 n 的 2 进制补码中的 1 消失，所以 $n \& (n-1) == 0$ 。

例 1: n 是 2 的幂次方

$n = 8 \Rightarrow 0000\ 1000$

$n-1 = 7 \Rightarrow 0000\ 0111$

例 2: n 不是 2 的幂次方

$n = 7 \Rightarrow 0000\ 0111$

$n-1 = 6 \Rightarrow 0000\ 0110$

根据[按位与](#)的运算法则，相同为 1，不同为 0。则，上面两个例子的结果一定为 0。

```

private void NumberIsTwoPower(int number)
{
    if ((number > 0) && (number & (number - 1)) == 0)
    {
        Debug.Log("是 2 的幂次");
    }
    else
    {
        Debug.Log("不是 2 的幂次");
    }
}

```

2. 写出下面的程序的打印结果，并解释其目的。

```

int Foo()
{
    int n = 742;
    int count = 0;
    while (n != 0)
    {
        count++;
        n &= (n - 1);
    }
    return count;
}

```

令: $n=11\ 0101\ 1000$ ，则

$n-1=11\ 0101\ 0111$ 。

有[按位与](#)运算得 $n \& (n-1)=11\ 0101\ 0000$ 。

程序输出的结果为 6，返回 n 的二进制表达式中，数值位 ‘1’ 的个数。

3. 写出下面的程序的打印结果，并解释其目的。

```
int Bar()
{
    return ((195 & 0xF0) >> 4) | ((195 & 0xF) << 4);
}
输出的结果为 60。
```

1) 原理

三. 递归

- 斐波那契数（通常用 $F(n)$ 表示）形成的序列称为斐波那契数列。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是： $F(0) = 0$ ， $F(1) = 1$ ， $F(n) = F(n-1) + F(n-2)$ ，其中 $n > 1$ 。现给定 n 请计算 $F(n)$ 。

```
private int Fibonacci(int n)
{
    if (n <= 0)
    {
        return 0;
    }
    if (n <= 2)
    {
        return 1;
    }
    return checked(Fibonacci(n - 2) + Fibonacci(n - 1));
}
```

- 已知 28657, 46368 是斐波那契数列中的两个相邻数，输出这两个数之前的斐波那契数列。

```
private void FindForwardFibonacci(int max_value, int min_value, ref List<int> fibonacciList)
{
    int tempMinValue; // 创建一个变量用于存储计算前的最小值
    while (max_value != 1)
    {
        tempMinValue = min_value;
        min_value = max_value - tempMinValue;
        max_value = tempMinValue;
        fibonacciList.Add(min_value);
    }
}
```