(4/500)从有限范围的数组中找出重复元素

给定一个数组，它有n个元素，数组中包含1到n-1这些值。在这个数组中，除了一个元素是重复的，其他元素都只出现了一次。请找出重复的元素。

例如，
输入: { 1, 2, 3, 4, 4 }
输出: The duplicate element is 4

输入:{ 1, 2, 3, 4, 2 }
输出: The duplicate element is 2

1. é€šè¿‡Hashæ•¥è®¡ç®—

这种算法的思想是使用Hash来解决问题。在这个算法中，我们使用了一个辅助数组，数组中的元素是boolean类型，数组中元素的值用以表示某个值是否出现过。如果出现过，那么就返回true。
C++语言实现：

```
#include <iostream>
using namespace std;

// Function to find a duplicate element in a limited range array
int findDuplicate(int arr[], int n)
{
   // create an visited array of size n+1
   // we can also use map instead of visited array
   bool visited[n];
   fill(visited, visited + n, 0); // sets every value in the array to 0

   // for each element of the array mark it as visited and
   // return the element if it is seen before
   for (int i = 0; i < n; i++)
   {
      // if element is seen before
      if (visited[arr[i]])
         return arr[i];

      // mark element as visited
```

```
        visited[arr[i]] = true;
    }

    // no duplicate found
    return -1;
}

// main function
int main()
{
    // input array contains n numbers between [1 to n - 1]
    // with one duplicate
    int arr[] = { 1, 2, 3, 4, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Duplicate element is " << findDuplicate(arr, n);

    return 0;
}
```

Java语言实现：

```
class findDuplicate
{
    // Function to find a duplicate element in a limited range array
    public static int findDuplicate(int arr[])
    {
        // n is length of the array
        int n = arr.length;

        // create an visited array of size n+1
        // we can also use map instead of visited array
        boolean visited[] = new boolean[n + 1];

        // for each element of the array mark it as visited and
        // return the element if it is seen before
        for (int i = 0; i < n; i++)
        {
            // if element is seen before
            if (visited[arr[i]])
```

```
                return arr[i];

            // mark element as visited
            visited[arr[i]] = true;
        }

        // no duplicate found
        return -1;
    }

    // main function
    public static void main (String[] args)
    {
        // input array contains n numbers between [1 to n - 1]
        // with one duplicate
        int arr[] = { 1, 2, 3, 4, 4 };

        System.out.println("Duplicate element is " + findDuplicate(arr));
    }
}
```

输出: Duplicate element is 4

以上两种方案的时间复杂度都是O(n)，空间复杂度都是o(n)。

2

我们还能使用常量空间（空间复杂度为O(n)）解决问题。既然数组中除了一个元素是重复之外，其他元素都是唯一的，并且元素的值在1到n-1之间，我们可以把数组的下标作为key，把元素取负数来判断元素是否重复。例如，队员每个元素arr[i]，我们得到它的绝对值abs(arr[i])，并修改下标为abs(arr[i]) -
1的这个元素的正负号。最后，我们再遍历一次数组，如果第 i
个元素的值是正数，那么重复的元素就是 i + 1。

上面的算法我们遍历了2次数组。其实，我们可以缩为1次。对于每一个元素arr[i]，我们得到绝对值abs(arr[i])，然后反转第abs(arr[i])-1个元素的符号，如果准备反转符号的元素已经是负数了，那么它一定是那个重复的数字。

C++语言实现：

```
#include <iostream>
using namespace std;

// Function to find a duplicate element in a limited range array
int findDuplicate(int arr[], int n)
{
    int duplicate = -1;

    // do for each element in the array
    for (int i = 0; i < n; i++)
    {
        // get absolute value of current element
        int absVal = (arr[i] < 0) ? -arr[i] : arr[i];

        // make element at index abs(arr[i])-1 negative if it is positive
        if (arr[absVal - 1] >= 0)
            arr[absVal - 1] = -arr[absVal - 1];
        else
        {
            // if element is already negative, it is repeated
            duplicate = absVal;
            break;
        }
    }

    // restore original array before returning
    for (int i = 0; i < n; i++)
    {
        // make negative elements positive
        if (arr[i] < 0)
            arr[i] = -arr[i];
    }

    // return duplicate element
    return duplicate;
}

// main function
int main()
```

```
{
    // input array contains n numbers between [1 to n - 1]
    // with one duplicate
    int arr[] = { 1, 2, 3, 4, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Duplicate element is " << findDuplicate(arr, n);

    return 0;
}
```

java语言实现：

```
class findDuplicate
{
    // Function to find a duplicate element in a limited range array
    public static int findDuplicate(int arr[])
    {
        // n is length of the array
        int n = arr.length;

        int duplicate = -1;

        // do for each element in the array
        for (int i = 0; i < n; i++)
        {
            // get absolute value of current element
            int absVal = (arr[i] < 0) ? -arr[i] : arr[i];

            // make element at index abs(arr[i]) - 1 negative
            // if it is positive
            if (arr[absVal - 1] >= 0)
                arr[absVal - 1] = -arr[absVal - 1];
            else
            {
                // if element is already negative, it is repeated
                duplicate = absVal;
                break;
            }
        }
```

```
        // restore original array before returning
        for (int i = 0; i < n; i++)
        {
            // make negative elements positive
            if (arr[i] < 0)
                arr[i] = -arr[i];
        }

        // return duplicate element
        return duplicate;
    }

    // main function
    public static void main (String[] args)
    {
        // input array contains n numbers between [1 to n - 1]
        // with one duplicate
        int arr[] = { 1, 2, 3, 4, 4 };

        System.out.println("Duplicate element is " + findDuplicate(arr));
    }
}
```

输出: Duplicate element is 2

以上两种方案的时间复杂度都是O(n)，空间复杂度都是o(1)。

3. ä½¿ç”¨å¼‚æ––

我们还可以使用异或（XOR）来解决问题。思路就是把所有的元素和1到n-1的数都异或一次，由于a^a = 0, 0^0 = 0，a^0 =
a。在这个异或中，所有的元素都出现了2次，而重复的元素出现了3次，所以最后的值就是重复的元素。

C++语言实现：

```
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to find a duplicate element in a limited range array
int findDuplicate(int A[], int n)
{
    int XOR = 0;

    // take xor of all array elements
    for (int i = 0; i < n; i++)
        XOR ^= A[i];

    // take xor of numbers from 1 to n-1
    for (int i = 1; i <= n-1; i++)
        XOR ^= i;

    // same elements will cancel out each other as a ^ a = 0,
    // 0 ^ 0 = 0 and a ^ 0 = a

    // xor will contain the missing number
    return XOR;
}

// main function
int main()
{
    // input array contains n numbers between [1 to n - 1]
    // with one duplicate
    int arr[] = { 1, 2, 3, 4, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Duplicate element is " << findDuplicate(arr, n);

    return 0;
}
```

java语言实现：

```
class findDuplicate
{
    // Function to find a duplicate element in a limited range array
    public static int findDuplicate(int A[])
```

```
    {
        // n is length of the array
        int n = A.length;

        int XOR = 0;

        // take xor of all array elements
        for (int i = 0; i < n; i++)
            XOR ^= A[i];

        // take xor of numbers from 1 to n-1
        for (int i = 1; i <= n - 1; i++)
            XOR ^= i;

        // same elements will cancel out each other as a ^ a = 0,
        // 0 ^ 0 = 0 and a ^ 0 = a

        // xor will contain the missing number
        return XOR;
    }

    // main function
    public static void main (String[] args)
    {
        // input array contains n numbers between [1 to n - 1]
        // with one duplicate
        int arr[] = { 1, 2, 3, 4, 4 };

        System.out.println("Duplicate element is " + findDuplicate(arr));
    }
}
```

输出: Duplicate element is 2

这种方法时间复杂度是O(n)，空间复杂度是O(1)。

4.æ±‚Œ

最后，我们可以通过求和的方式解决问题：先计算出所有元素的和，然后再减去1到n-1的和，最后的值就是重复的数。简单吧。:)

C++语言：

```
#include <numeric>
#include <iostream>
#include <array>

template <typename It>
int find_duplicate(It start, It finish)
{
   auto size = std::distance(start, finish);

   int actual_sum = std::accumulate(start, finish, 0);
   int expected_sum = size * (size - 1) / 2;

   return actual_sum - expected_sum;
}

int main()
{
   std::array<int, 5> arr1 = {{ 1, 2, 3, 4, 4 }};
   std::array<int, 5> arr2 = {{ 1, 2, 3, 4, 2 }};

   std::cout << "The duplicate element is " <<
            find_duplicate(arr1.begin(), arr1.end()) << '\n';

   std::cout << "The duplicate element is " <<
            find_duplicate(arr2.begin(), arr2.end()) << '\n';
}
```

输出:

```
The duplicate element is 4
The duplicate element is 2
```

这种方法时间复杂度是O(n)，空间复杂度是O(1)。