> 给你输入一个升序排列的数组nums（可能包含重复数字），请你判断nums是否能够被分割成若干个长度至少为 3 的子序列，每个子序列都由连续的整数组成。

分析： 将nums分配到若干个数组中，其框架逻辑如下：

```
for (int v : nums) {
    if(...){
        // 将v划分到某个子序列中
    }else{
        // 无法分配
        return false;
    }

    return true;
}
```

分析可知： 对一个字符v有以下可能：

1. 当前元素v自成一派，构成一个以自己为开始的长度不小于3的字符串；
2. 当前元素v接到已存在的子序列后面。

为实现上述方法需要以下辅助：

1. freq哈希表判断能否作为开头；

> 比如freq[3] == 2说明元素3在nums中出现了 2 次。
> 那么如果我发现freq[3], freq[4], freq[5]都是大于 0 的，那就说明元素3可以作为开头组成一个长度为 3 的子序列。

2. need哈希表帮助一个元素判断自己是否能够可以被接到其他元素后面。

> 比如说现在已经组成了两个子序列[1,2,3,4]和[2,3,4]，那么need[5]的值就应该是 2，说明对元素5的需求为 2。

解法很巧妙

```
class Solution {
public:
  bool isPossible(std::vector<int>& nums) {
    std::unordered_map<int, int> freq;
    std::unordered_map<int, int> need;

    // 统计元素出现频率
```

```
    for (auto item : nums) {
      freq[item]++;
    }

    for (auto v : nums) {
      if (0 == freq[v]) {
        // 已被用到其他对列
        continue;
      }

      // 首先判断v是否可以接到其他对列之后
      if (need.count(v) && need[v] > 0) {
        // v 可以接到之前某个序列之后
        freq[v]--;
        // 对v的需求减1
        need[v]--;
        // v+1的需求加1
        need[v + 1]++;
      } else if (freq[v] > 0 && freq[v + 1] > 0 && freq[v + 2] > 0) {
        // 将v作为一个新的队列开始
        freq[v]--;
        freq[v + 1]--;
        freq[v + 2]--;

        // 对v+3的需求加1
        need[v + 3]++;
      } else {
        // 两种情况都不符合，返回false
        return false;
      }
    }

    return true;
  }
};
```

如果改为k个子序列连续，则在判断v是否为新序列开头时，将判断改为连续判断k个。

进阶

如果需要返回所有的连续子序列，如何操作:

```
class Solution {
public:
  bool isPossible(std::vector<int>& nums) {
    std::unordered_map<int, int> freq;  // 频率统计
    std::unordered_map<int, std::vector<std::vector<int>>> need;  // 需求列
表

    for (auto item : nums) {
      freq[item]++;
```

```
    }

    for (auto item : nums) {
      if (0 == freq[item]) {
        continue;
      }
      // 首先判断是否可以连接到某个队列之后
      if (need.count(item) && need[item].size() > 0) {
        // 随机选择一个队列加入
        freq[item]--;
        std::vector<int> tmp = need[item].back();
        need[item].pop_back();
        tmp.push_back(item);
        need[item + 1].push_back(tmp);
      } else if (freq[item] > 0 && freq[item + 1] > 0 && freq[item + 2] >
0) {
        freq[item]--;
        freq[item + 1]--;
        freq[item + 2]--;
        std::vector<int> tmp{item, item + 1, item + 2};
        need[item + 3].push_back(tmp);
      } else {
        return false;  // 不可构成子序列
      }
    }

    std::vector<std::vector<int>> res;
    for (auto it : need) {
      for (auto i : it.second) {
        res.push_back(i);
      }
    }

    return true;
  }
};
```