2021最新版数据结构与算法面试题手册

2021最新版数据结构与算法面试题手册

题目来源网络，涉版权问题联系删除

1 | Java 工程师

1.1 | 哈希

- **请说一说，Java中的HashMap的工作原理是什么?**

参考回答：

HashMap类有一个叫做Entry的内部类。这个Entry类包含了key-value作为实例变量。 每当往hashmap里面存放key-value对的时候，都会为它们实例化一个Entry对象，这个Entry对象就会存储在前面提到的Entry数组table中。Entry具体存在table的那个位置是 根据key的hashcode()方法计算出来的hash值（来决定）。

- **介绍一下，什么是Hashmap?**

参考回答：

HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。

HashMap 继承于AbstractMap，实现了Map、Cloneable、java.io.Serializable接口。

HashMap 的实现不是同步的，这意味着它不是线程安全的。它的key、value都可以为null。此外，HashMap中的映射不是有序的。

HashMap 的实例有两个参数影响其性能：“初始容量”和“加载因子”。容量 是哈希表中桶的数量，初始容量 只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。

通常，默认加载因子是 0.75, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数 HashMap 类的操作中，包

括 get 和 put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作。

hashmap共有4个构造函数：

// 默认构造函数。HashMap()

// 指定“容量大小”的构造函数

HashMap(int capacity)

// 指定“容量大小”和“加载因子”的构造函数

HashMap(int capacity, float loadFactor)

// 包含“子Map”的构造函数

HashMap(Map<? extends K, ? extends V> map)

- **讲一讲，如何构造一致性哈希算法。**

参考回答：

先构造一个长度为232的整数环（这个环被称为一致性Hash环），根据节点名称的Hash值（其分布为[0, 232-1]）将服务器节点放置在这个Hash环上，然后根据数据的Key值计算得到其Hash值（其分布也为[0, 232-1]），接着在Hash环上顺时针查找距离这个Key值的Hash值最近的服务器节点，完成Key到服务器的映射查找。

这种算法解决了普通余数Hash算法伸缩性差的问题，可以保证在上线、下线服务器的情况下尽量有多的请求命中原来路由到的服务器。

- **请谈一谈，hashCode() 和equals() 方法的重要性体现在什么地方?**

参考回答：

Java中的HashMap使用hashCode()和equals()方法来确定键值对的索引，当根据键获取值的时候也会用到这两个方法。如果没有正确的实现这两个方法，两个不同的键可能会有相同的hash值，因此，可能会被集合认为是相等的。而且，这两个方法也用来发现重复元素。所以这两个方法的实现对HashMap的精确性和正确性是至关重要的。

- **请问，Object作为HashMap的key的话，对Object有什么要求吗?**

参考回答：

要求Object中hashcode不能变。

- **请问 hashset 存的数是有序的吗?**

参考回答：

Hashset是无序的。

1.2 | 二叉树

原文链接：https://www.jianshu.com/p/0190985635eb

- **求二叉树的最大深度**

参考回答：

```
class TreeNode{
    int val;
    //左孩子
    TreeNode left;
    //右孩子
```

```
    TreeNode right;
}
```

- **求二叉树的最小深度**

参考回答：

```
int getMinDepth(TreeNode root){
    if(root == null){
        return 0;
    }
    return getMin(root);
}
int getMin(TreeNode root){
    if(root == null){
        return Integer.MAX_VALUE;
    }
    if(root.left == null&&root.right == null){
        return 1;
    }
    return Math.min(getMin(root.left),getMin(root.right)) + 1;
}
```

- **求二叉树中节点的个数**

参考回答：

```
    int numOfTreeNode(TreeNode root){
        if(root == null){
            return 0;

        }
        int left = numOfTreeNode(root.left);
        int right = numOfTreeNode(root.right);
        return left + right + 1;
    }
```

- **求二叉树中叶子节点的个数**

参考回答：

```
    int numsOfNoChildNode(TreeNode root){
        if(root == null){
            return 0;
        }
        if(root.left==null&&root.right==null){
            return 1;
        }
        return
numsOfNodeTreeNode(root.left)+numsOfNodeTreeNode(root.right);

```

```
    }
```

- **求二叉树中第k层节点的个数**

参考回答：

```
        int numsOfkLevelTreeNode(TreeNode root,int k){
            if(root == null||k<1){
                return 0;
            }
            if(k==1){
                return 1;
            }
            int numsLeft = numsOfkLevelTreeNode(root.left,k-1);
            int numsRight = numsOfkLevelTreeNode(root.right,k-1);
            return numsLeft + numsRight;
        }
```

- **判断二叉树是否是平衡二叉树**

参考回答：

```
    boolean isBalanced(TreeNode node){
        return maxDeath2(node)!=-1;
    }
    int maxDeath2(TreeNode node){
        if(node == null){
            return 0;
        }
        int left = maxDeath2(node.left);
        int right = maxDeath2(node.right);
        if(left==-1||right==-1||Math.abs(left-right)>1){
            return -1;
        }
        return Math.max(left, right) + 1;
    }
```

- **判断二叉树是否是完全二叉树**

参考回答：

```
    boolean isCompleteTreeNode(TreeNode root){
        if(root == null){
            return false;
        }
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);
        boolean result = true;
        boolean hasNoChild = false;
        while(!queue.isEmpty()){
            TreeNode current = queue.remove();
```

```
            if(hasNoChild){
                if(current.left!=null||current.right!=null){
                    result = false;
                    break;
                }
            }else{
                if(current.left!=null&&current.right!=null){
                    queue.add(current.left);
                    queue.add(current.right);
                }else if(current.left!=null&&current.right==null){
                    queue.add(current.left);
                    hasNoChild = true;

                }else if(current.left==null&&current.right!=null){
                    result = false;
                    break;
                }else{
                    hasNoChild = true;
                }
            }

        }
        return result;
    }
```

- **两个二叉树是否完全相同**

参考回答：

```
    boolean isSameTreeNode(TreeNode t1,TreeNode t2){
        if(t1==null&&t2==null){
            return true;
        }
        else if(t1==null||t2==null){
            return false;
        }
        if(t1.val != t2.val){
            return false;
        }
        boolean left = isSameTreeNode(t1.left,t2.left);
        boolean right = isSameTreeNode(t1.right,t2.right);
        return left&&right;

    }
```

- **两个二叉树是否互为镜像**

参考回答：

```
    boolean isMirror(TreeNode t1,TreeNode t2){
```

```
        if(t1==null&&t2==null){
            return true;
        }
        if(t1==null||t2==null){
            return false;
        }
        if(t1.val != t2.val){
            return false;
        }
        return isMirror(t1.left,t2.right)&&isMirror(t1.right,t2.left);

    }
```

- **翻转二叉树or镜像二叉树**

参考回答：

```
    TreeNode mirrorTreeNode(TreeNode root){
        if(root == null){
            return null;
        }
        TreeNode left = mirrorTreeNode(root.left);
        TreeNode right = mirrorTreeNode(root.right);
        root.left = right;
        root.right = left;
        return root;
    }
```

- **求两个二叉树的最低公共祖先节点**

参考回答：

```
    TreeNode getLastCommonParent(TreeNode root,TreeNode t1,TreeNode t2){
        if(findNode(root.left,t1)){
            if(findNode(root.right,t2)){
                return root;
            }else{
                return getLastCommonParent(root.left,t1,t2);
            }
        }else{
            if(findNode(root.left,t2)){
                return root;
            }else{
                return getLastCommonParent(root.right,t1,t2)
            }
        }
    }
    // 查找节点node是否在当前 二叉树中
    boolean findNode(TreeNode root,TreeNode node){
        if(root == null || node == null){
```

```
            return false;
        }
        if(root == node){
            return true;
        }
        boolean found = findNode(root.left,node);
        if(!found){
            found = findNode(root.right,node);
        }
        return found;
    }
```

- **二叉树的前序遍历**

参考回答:

迭代解法

```
    ArrayList<Integer> preOrder(TreeNode root){
        Stack<TreeNode> stack = new Stack<TreeNode>();
        ArrayList<Integer> list = new ArrayList<Integer>();
        if(root == null){
            return list;
        }
        stack.push(root);
        while(!stack.empty()){
            TreeNode node = stack.pop();
            list.add(node.val);
            if(node.right!=null){
                stack.push(node.right);
            }
            if(node.left != null){
                stack.push(node.left);
            }

        }
        return list;
    }
```

递归解法

```
    ArrayList<Integer> preOrderReverse(TreeNode root){
        ArrayList<Integer> result = new ArrayList<Integer>();
        preOrder2(root,result);
        return result;

    }
    void preOrder2(TreeNode root,ArrayList<Integer> result){
        if(root == null){
            return;
        }
        result.add(root.val);
```

```
        preOrder2(root.left,result);
        preOrder2(root.right,result);
    }
```

- **二叉树的中序遍历**

参考回答：

```
    ArrayList<Integer> inOrder(TreeNode root){
        ArrayList<Integer> list = new ArrayList<<Integer>();
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode current = root;
        while(current != null|| !stack.empty()){
            while(current != null){
                stack.add(current);
                current = current.left;
            }
            current = stack.peek();
            stack.pop();
            list.add(current.val);
            current = current.right;

        }
        return list;

    }
```

- **二叉树的后序遍历**

参考回答：

```
    ArrayList<Integer> postOrder(TreeNode root){
        ArrayList<Integer> list = new ArrayList<Integer>();
        if(root == null){
            return list;
        }
        list.addAll(postOrder(root.left));
        list.addAll(postOrder(root.right));
        list.add(root.val);
        return list;
    }
```

- **前序遍历和后序遍历构造二叉树**

参考回答：

```
    TreeNode buildTreeNode(int[] preorder,int[] inorder){
        if(preorder.length!=inorder.length){
            return null;
        }
```

```
        return myBuildTree(inorder,0,inorder.length-1,preorder,0,preorder.length-1);
    }
    TreeNode myBuildTree(int[] inorder,int instart,int inend,int[] preorder,int prestart,int preend){
        if(instart>inend){
            return null;
        }
        TreeNode root = new TreeNode(preorder[prestart]);
        int position = findPosition(inorder,instart,inend,preorder[start]);
        root.left = myBuildTree(inorder,instart,position-1,preorder,prestart+1,prestart+position-instart);
        root.right = myBuildTree(inorder,position+1,inend,preorder,position-inend+preend+1,preend);
        return root;
    }
    int findPosition(int[] arr,int start,int end,int key){
        int i;
        for(i = start;i<=end;i++){
            if(arr[i] == key){
                return i;
            }
        }
        return -1;
    }
```

- **在二叉树中插入节点**

参考回答：

```
    TreeNode insertNode(TreeNode root,TreeNode node){
        if(root == node){
            return node;
        }
        TreeNode tmp = new TreeNode();
        tmp = root;
        TreeNode last = null;
        while(tmp!=null){
            last = tmp;
            if(tmp.val>node.val){
                tmp = tmp.left;
            }else{
                tmp = tmp.right;
            }
        }
        if(last!=null){
            if(last.val>node.val){
                last.left = node;
```

```
            }else{
                last.right = node;
            }
        }
        return root;
    }
```

- **输入一个二叉树和一个整数，打印出二叉树中节点值的和等于输入整数所有的路径**

参考回答：

```
    void findPath(TreeNode r,int i){
        if(root == null){
            return;
        }
        Stack<Integer> stack = new Stack<Integer>();
        int currentSum = 0;
        findPath(r, i, stack, currentSum);

    }
    void findPath(TreeNode r,int i,Stack<Integer> stack,int currentSum){
        currentSum+=r.val;
        stack.push(r.val);
        if(r.left==null&&r.right==null){
            if(currentSum==i){
                for(int path:stack){
                    System.out.println(path);
                }

            }
        }
        if(r.left!=null){
            findPath(r.left, i, stack, currentSum);
        }
        if(r.right!=null){
            findPath(r.right, i, stack, currentSum);
        }
        stack.pop();
    }
```

- **二叉树的搜索区间**

> 给定两个值 k1 和 k2（k1 < k2）和一个二叉查找树的根节点。找到树中所有值在 k1 到 k2 范围内的节点。即打印所有x (k1 <= x <= k2) 其中 x 是二叉查找树的中的节点值。返回所有升序的节点值。
>
> 作者：IOExceptioner
> 链接：https://www.jianshu.com/p/0190985635eb
> 来源：简书
> 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

参考回答：

```
    ArrayList<Integer> result;
    ArrayList<Integer> searchRange(TreeNode root,int k1,int k2){
        result = new ArrayList<Integer>();
        searchHelper(root,k1,k2);
        return result;
    }
    void searchHelper(TreeNode root,int k1,int k2){
        if(root == null){
            return;
        }
        if(root.val>k1){
            searchHelper(root.left,k1,k2);
        }
        if(root.val>=k1&&root.val<=k2){
            result.add(root.val);
        }
        if(root.val<k2){
            searchHelper(root.right,k1,k2);
        }
    }
```

- **二叉树的层次遍历**

参考回答：

```
    ArrayList<ArrayList<Integer>> levelOrder(TreeNode root){
        ArrayList<ArrayList<Integer>> result = new
ArrayList<ArrayList<Integer>>();
        if(root == null){
            return result;
        }
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while(!queue.isEmpty()){
            int size = queue.size();
            ArrayList<<Integer> level = new ArrayList<Integer>():
            for(int i = 0;i < size ;i++){
                TreeNode node = queue.poll();
                level.add(node.val);
                if(node.left != null){
                    queue.offer(node.left);
                }
                if(node.right != null){
                    queue.offer(node.right);
                }
            }
            result.add(Level);
        }
```

```
        return result;
    }
```

- **二叉树内两个节点的最长距离**

> 二叉树中两个节点的最长距离可能有三种情况：
> 1.左子树的最大深度+右子树的最大深度为二叉树的最长距离
> 2.左子树中的最长距离即为二叉树的最长距离
> 3.右子树种的最长距离即为二叉树的最长距离
> 因此，递归求解即可
>
> 作者：IOExceptioner
> 链接：https://www.jianshu.com/p/0190985635eb
> 来源：简书
> 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

参考回答：

```
private static class Result{
    int maxDistance;
    int maxDepth;
    public Result() {
    }

    public Result(int maxDistance, int maxDepth) {
        this.maxDistance = maxDistance;
        this.maxDepth = maxDepth;
    }
}
    int getMaxDistance(TreeNode root){
      return getMaxDistanceResult(root).maxDistance;
    }
    Result getMaxDistanceResult(TreeNode root){
        if(root == null){
            Result empty = new Result(0,-1);
            return empty;
        }
        Result lmd = getMaxDistanceResult(root.left);
        Result rmd = getMaxDistanceResult(root.right);
        Result result = new Result();
        result.maxDepth = Math.max(lmd.maxDepth,rmd.maxDepth) + 1;
        result.maxDistance = Math.max(lmd.maxDepth +
rmd.maxDepth,Math.max(lmd.maxDistance,rmd.maxDistance));
        return result;
    }
```

- **不同的二叉树**

> 给出 n，问由 1...n 为节点组成的不同的二叉查找树有多少种?

参考回答：

```
    int numTrees(int n ){
        int[] counts = new int[n+2];
        counts[0] = 1;
        counts[1] = 1;
        for(int i = 2;i<=n;i++){
            for(int j = 0;j<i;j++){
                counts[i] += counts[j] * counts[i-j-1];
            }
        }
        return counts[n];
    }
```

- **判断二叉树是否是合法的二叉查找树(BST)**

> 一棵BST定义为：
> 节点的左子树中的值要严格小于该节点的值。
> 节点的右子树中的值要严格大于该节点的值。
> 左右子树也必须是二叉查找树。
> 一个节点的树也是二叉查找树。

参考回答：

```
    public int lastVal = Integer.MAX_VALUE;
    public boolean firstNode = true;
    public boolean isValidBST(TreeNode root) {
        // write your code here
        if(root==null){
            return true;
        }
        if(!isValidBST(root.left)){
            return false;
        }
        if(!firstNode&&lastVal >= root.val){
            return false;
        }
        firstNode = false;
        lastVal = root.val;
        if (!isValidBST(root.right)) {
            return false;
        }
        return true;
    }
```

1.3 | 链表

- **谈一谈，bucket如果用链表存储，它的缺点是什么？**

参考回答：

①查找速度慢，因为查找时，需要循环链表访问

②如果进行频繁插入和删除操作，会导致速度很慢。

- **有一个链表，奇数位升序偶数位降序，如何将链表变成升序?**

参考回答:

```
public class
OddIncreaseEvenDecrease {
    /**
     * 按照奇偶位拆分成两个链表
     * @param head
     * @return
     */
    public static Node[] getLists(Node head){
        Node head1 = null;
        Node head2 = null;

        Node cur1 = null;
        Node cur2 = null;
        int count = 1;//用来计数
        while(head != null){
            if(count % 2 == 1){
                if(cur1 != null){
                    cur1.next = head;
                    cur1 = cur1.next;
                }else{
                    cur1 = head;
                    head1 = cur1;
                }
            }else{
                if(cur2 != null){
                    cur2.next = head;
                    cur2 = cur2.next;
                }else{
                    cur2 = head;
                    head2 = cur2;
                }
            }
            head = head.next;
            count++;
        }
        //跳出循环，要让最后两个末尾元素的下一个都指向null
        cur1.next = null;
        cur2.next = null;

        Node[] nodes = new Node[]{head1,
head2};
        return nodes;
    }
```

```
44
45      /**
46       * 反转链表
47       * @param head
48       * @return
49       */
50      public static Node reverseList(Node head){
51          Node pre = null;
52          Node next = null;
53          while(head != null){
54              next = head.next;
55              head.next = pre;
56              pre = head;
57              head = next;
58          }
59          return pre;
60      }
61
62      /**
63       * 合并两个有序链表
64       * @param head1
65       * @param head2
66       * @return
67       */
68      public static Node CombineList(Node head1,
69  Node head2){
70          if(head1 == null || head2 == null){
71              return head1 != null ? head1 :
72  head2;
73          }
74          Node head = head1.value < head2.value ?
75  head1 : head2;
76          Node cur1 = head == head1 ? head1 :
77  head2;
78          Node cur2 = head == head1 ? head2 :
79  head1;
80          Node pre = null;
81          Node next = null;
82          while(cur1 != null && cur2 !=
83  null){
84              if(cur1.value <= cur2.value){//这里一定要有=，否则一旦cur1的value
    和cur2的value相等的话，下面的pre.next会出现空指针异常
85                  pre = cur1;
86                  cur1 = cur1.next;
87              }else{
88                  next = cur2.next;
89                  pre.next = cur2;
90                  cur2.next = cur1;
91                  pre = cur2;
```

```
                cur2 = next;
            }
        }
        pre.next = cur1 == null ? cur2 : cur1;

        return head;
    }

}
```

- **如何反转单链表**

参考回答：

```
ListNode
reverseList(ListNode* head) {
        if(head == nullptr || head->next ==
nullptr)
            return head;
        ListNode* p;
        ListNode* q;
        ListNode* r;
        p = head;
        q = head->next;
        head->next = nullptr;//旧的头指针是新的尾指针 指向NULL
        while(q){
            r = q->next;//用来保存下一步要处理的指针
            q->next = p;//p q 交替处理 进行反转单链表
            p = q;
            q = r;
        }
        head = p;//最后的q必定指向NULL，p就成了新链表的头指针
        return head;
}
```

- **现在有一个单向链表，谈一谈，如何判断链表中是否出现了环**

参考回答：

单链表有环，是指单链表中某个节点的next指针域指向的是链表中在它之前的某一个节点，这样在链表的尾部形成一个环形结构。

// 链表的节点结构如下 typedef struct node { int data; struct node *next; } NODE;

最常用方法：定义两个指针，同时从链表的头节点出发，一个指针一次走一步，另一个指针一次走两步。如果走得快的指针追上了走得慢的指针，那么链表就是环形链表；如果走得快的指针走到了链表的末尾（next指向 NULL）都没有追上第一个指针，那么链表就不是环形链表。

通过使用STL库中的map表进行映射。首先定义 map<NODE *, int> m; 将一个 NODE * 指针映射成数组的下标，并赋值为一个 int 类型的数值。然后从链表的头指针开始往后遍历，每次遇到一个指针p，就判断 m[p] 是否为0。如果为0，则将m[p]赋值为1，表示该节点第一次访问；而如果m[p]的值为1，则说明这个节点已经被访问过一次了，于是就形成了环。

- **随机链表的复制**

参考回答：

```
public RandomListNode copyRandomList(RandomListNode head) {

    if (head == null)
        return null;

    RandomListNode p = head;

    // copy every node and insert to list
    while (p != null) {
        RandomListNode copy = new RandomListNode(p.label);
        copy.next = p.next;
        p.next = copy;
        p = copy.next;
    }

    // copy random pointer for each new node
    p = head;
    while (p != null) {
        if (p.random != null)
            p.next.random = p.random.next;
        p = p.next.next;
    }

    // break list to two
    p = head;
    RandomListNode newHead = head.next;
    while (p != null) {
        RandomListNode temp = p.next;
        p.next = temp.next;
        if (temp.next != null)
            temp.next = temp.next.next;
        p = p.next;
    }

```

```
    return newHead;
}
```

1.4 | 数组

- **写一个算法，可以将一个二维数组顺时针旋转90度。**

参考回答：

```
public void
rotate(int[][] matrix) {
        int n = matrix.length;
        for (int i = 0; i < n/2; i++) {
            for (int j = i; j < n-1-i; j++)
{
                int temp = matrix[i][j];
                matrix[i][j] =
matrix[n-1-j][i];
                matrix[n-1-j][i] =
matrix[n-1-i][n-1-j];
                matrix[n-1-i][n-1-j] =
matrix[j][n-1-i];
                matrix[j][n-1-i] = temp;
            }
        }
}
```

- **一个数组，除一个元素外其它都是两两相等，求那个元素?**

参考回答：

```
 public static int find1From2(int[] a){
        int len = a.length, res = 0;
        for(int i = 0; i < len; i++){
            res= res ^ a[i];
        }
        return res;
}
```

- **找出数组中和为S的一对组合，找出一组就行**

参考回答：

```
public int[]
twoSum(int[] nums, int target) {
```

```
        HashMap<Integer, Integer> map =
new HashMap<Integer, Integer>();
        int[] a = new int[2];
        map.put(nums[0], 0);
        for (int i = 1; i < nums.length;
i++) {
            if (map.containsKey(target - nums[i])) {
                a[0] = map.get(target -
nums[i]);
                a[1] = i;
                return a;
            } else {
                map.put(nums[i], i);
            }
        }
        return a;
}
```

- **求一个数组中连续子向量的最大和**

参考回答：

```
public int
maxSubArray(int[] nums) {
        int sum = 0;
        int maxSum = Integer.MIN_VALUE;
        if (nums == null || nums.length == 0) {
            return sum;
        }
        for (int i = 0; i < nums.length;
i++) {
            sum += nums[i];
            maxSum = Math.max(maxSum, sum);
            if (sum < 0) {
                sum = 0;
            }
        }
        return maxSum;
}
```

- **寻找一数组中前K个最大的数**

参考回答：

```
public int
findKthLargest(int[] nums, int k) {
    if (k < 1 || nums == null) {
        return
0;
    }

    return getKth(nums.length - k +1, nums, 0,
nums.length - 1);
}

public int
getKth(int k, int[] nums, int start, int end) {

    int pivot = nums[end];

    int left = start;
    int right = end;

    while (true) {

        while
(nums[left] < pivot && left < right) {
            left++;
        }

        while
(nums[right] >= pivot && right > left) {
            right--;
        }

        if
(left == right) {
            break;
        }

        swap(nums,
left, right);
    }

    swap(nums, left, end);

```

```
    if (k == left + 1) {
        return
pivot;
    } else if (k < left + 1) {
        return
getKth(k, nums, start, left - 1);
    } else {
        return
getKth(k, nums, left + 1, end);
    }
}

public void
swap(int[] nums, int n1, int n2) {
    int tmp = nums[n1];
    nums[n1] = nums[n2];
    nums[n2] = tmp;
}
```

1.5 | 排序

- **用Java写一个冒泡排序?**

参考回答:

```
import
java.util.Comparator;
 /**
  * 排序器接口(策略模式: 将算法封装到具有共同接口的独立的类中使得它们可以相互替换)
  */
 public interface Sorter {
    /**
     *
  */
 public interface Sorter {
    /**
     * 排序
     * @param list 待排序的数组
     */
    public <T extends Comparable<T>> void sort(T[] list);
    /**
     * 排序
     * @param list 待排序的数组
```

```
     * @param comp 比较两个对象的比较器
     */
    public <T> void sort(T[] list, Comparator<T> comp);
 }
 import java.util.Comparator;
 /**
  * 冒泡排序
  *
  */
 public class BubbleSorter implements Sorter {
     @Override
     public <T extends Comparable<T>> void sort(T[]
list) {
         boolean swapped = true;
         for (int i = 1, len = list.length; i
< len && swapped; ++i) {
             swapped =
false;
             for (int j =
0; j < len - i; ++j) {

if (list[j].compareTo(list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}
             }
         }
     }
     @Override
     public <T> void sort(T[] list, Comparator<T>
comp) {
         boolean swapped = true;
         for (int i = 1, len = list.length; i
< len && swapped; ++i) {
             swapped =
```

```
61 false;
62              for (int j =
63 0; j < len - i; ++j) {
64
65 if (comp.compare(list[j], list[j + 1]) > 0) {
66
67 T temp = list[j];
68
69 list[j] = list[j + 1];
70
71 list[j + 1] = temp;
72
73 swapped = true;
74
75 }
76              }
77          }
```

- **介绍一下，排序都有哪几种方法？请列举出来**

参考回答：

排序的方法有：插入排序（直接插入排序、希尔排序），交换排序（冒泡排序、快速排序），选择排序（直接选择排序、堆排序），
归并排序，分配排序（箱排序、基数排序）
快速排序的伪代码。
/ /使用快速排序方法对a[0 :n- 1]排序
从a[0 :n- 1]中选择一个元素作为m i d d l e，该元素为支点
把余下的元素分割为两段left 和r i g h t，使得l e f t中的元素都小于等于支点，而right 中的元素都大于等于支点
递归地使用快速排序方法对left 进行排序
递归地使用快速排序方法对right 进行排序
所得结果为l e f t + m i d d l e + r i g h t

- **介绍一下，归并排序的原理是什么？**

参考回答：

（1）归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

（2）首先考虑下如何将将二个有序数列合并。这个非常简单，只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

（3）解决了上面的合并有序数列问题，再来看归并排序，其的基本思路就是将数组分成二组A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序。如何让这二组组内数据有序了？

可以将A，B组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。这样通过先递归的分解数列，再合并数列就完成了归并排序。

- **介绍一下，堆排序的原理是什么？**

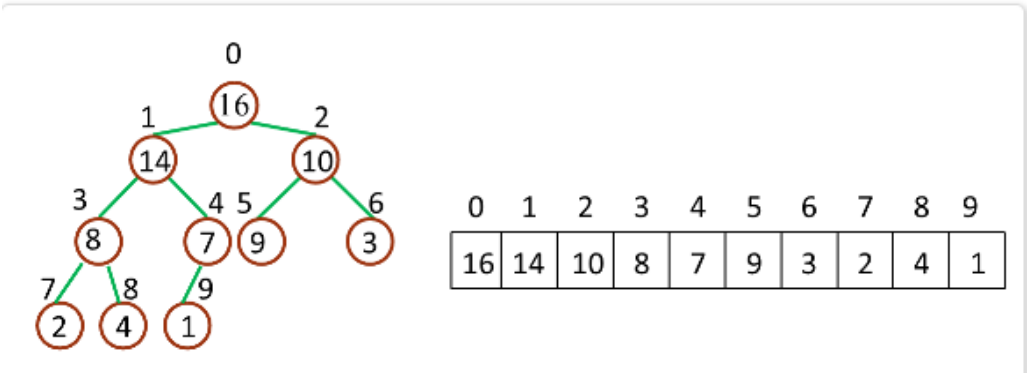
参考回答：

堆排序就是把最大堆堆顶的最大数取出，将剩余的堆继续调整为最大堆，再次将堆顶的最大数取出，这个过程持续到剩余数只有一个时结束。在堆中定义以下几种操作：

（1）最大堆调整（Max-Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点。

（2）创建最大堆（Build-Max-Heap）：将堆所有数据重新排序，使其成为最大堆。

（3）堆排序（Heap-Sort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算



- **谈一谈，如何得到一个数据流中的中位数？**

参考回答：

数据是从一个数据流中读出来的，数据的数目随着时间的变化而增加。如果用一个数据容器来保存从流中读出来的数据，当有新的数据流中读出来时，这些数据就插入到数据容器中。

数组是最简单的容器。如果数组没有排序，可以用 Partition 函数找出数组中的中位数。在没有排序的数组中插入一个数字和找出中位数的时间复杂度是 O(1)和 O(n)。

我们还可以往数组里插入新数据时让数组保持排序，这是由于可能要移动 O(n)个数，因此需要 O(n)时间才能完成插入操作。在已经排好序的数组中找出中位数是一个简单的操作，只需要 O(1)时间即可完成。

排序的链表时另外一个选择。我们需要 O(n)时间才能在链表中找到合适的位置插入新的数据。如果定义两个指针指向链表的中间结点（如果链表的结点数目是奇数，那么这两个指针指

向同一个结点），那么可以在 O（1）时间得出中位数。此时时间效率与及基于排序的数组的时间效率一样。

如果能够保证数据容器左边的数据都小于右边的数据，这样即使左、右两边内部的数据没有排序，也可以根据左边最大的数及右边最小的数得到中位数。如何快速从一个容器中找出最大数？用最大堆实现这个数据容器，因为位于堆顶的就是最大的数据。同样，也可以快速从最小堆中找出最小数。

因此可以用如下思路来解决这个问题：用一个最大堆实现左边的数据容器，用最小堆实现右边的数据容器。往堆中插入一个数据的时间效率是 O(logn)。由于只需 O(1)时间就可以得到位于堆顶的数据，因此得到中位数的时间效率是 O(1)。

- **你知道哪些排序算法，这些算法的时间复杂度分别是多少，解释一下快排?**

参考回答：

排序方式	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
希尔排序	$O(n^{1.3})$			O(1)	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	简单
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定	较复杂
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定	较复杂
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(r)	稳定	较复杂

快排：快速排序有两个方向，左边的i下标一直往右走（当条件a[i] <= a[center_index]时），其中center_index是中枢元素的数组下标，一般取为数组第0个元素。

而右边的j下标一直往左走（当a[j] > a[center_index]时）。

如果i和j都走不动了，i <= j, 交换a[i]和a[j],重复上面的过程，直到i>j。交换a[j]和a[center_index]，完成一趟快速排序。

1.6 | 堆与栈

- **请你解释一下，内存中的栈(stack)、堆(heap) 和静态区(static area) 的用法。**

参考回答：

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过new关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直接书写的100、”hello”和常量都是放在静态区中。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用。

String str = new String(”hello”);

上面的语句中变量str放在栈上，用new创建出来的字符串对象放在堆上，而”hello”这个字面量放在静态区。

- **说一说，heap和stack有什么区别。**

参考回答：

（1）Java的堆是一个运行时数据区，类的对象从中分配空间。通过比如：new等指令建立，不需要代码显式的释放，由垃圾回收来负责。

优点：可以动态地分配内存大小，垃圾收集器会自动回收垃圾数据。

缺点：由于其优点，所以存取速度较慢。

（2）栈：

其数据项的插入和删除都只能在称为栈顶的一端完成，后进先出。栈中存放一些基本类型的变量 和 对象句柄。

优点：读取数度比堆要快，仅次于寄存器，栈数据可以共享。

缺点：比堆缺乏灵活性，存在栈中的数据大小与生存期必须是确定的。

举例：

String是一个特殊的包装类数据。可以用：

String str = new String(”geek”);

String str = ”geek”;

两种的形式来创建，第一种是用new()来新建对象的，它会在存放于堆中。每调用一次就会创建一个新的对象。而第二种是先在栈中创建一个对String类的对象引用变量str，然后查找栈中有没有存放”geek”，如果没有，则将”geek”存放进栈，并令str指向”abc”，如果已经有”geek”则直接令str指向”geek”。

- **最小的k个数**

> 输入整数数组 arr ，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。
> 示例 1：
> 输入：arr = [3,2,1], k = 2
> 输出：[1,2] 或者 [2,1]
> 示例 2：
>
> 输入：arr = [0,1,2,1], k = 1
> 输出：[0]
>
> 限制：
> 0 <= k <= arr.length <= 10000
> 0 <= arr[i] <= 10000
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/zui-xiao-de-kge-shu-lcof
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

大根堆(前 K 小) / 小根堆（前 K 大),Java中有现成的 PriorityQueue，实现起来最简单：O(NlogK)O(NlogK)
本题是求前 K 小，因此用一个容量为 K 的大根堆，每次 poll 出最大的数，那堆中保留的就是前 K 小啦（注意不是小根堆！小根堆的话需要把全部的元素都入堆，那是 O(NlogN)O(NlogN)😂，就不是 O(NlogK)O(NlogK)啦～～）
这个方法比快排慢，但是因为 Java 中提供了现成的 PriorityQueue（默认小根堆），所以实现起来最简单，没几行代码～

```
// 保持堆的大小为K，然后遍历数组中的数字，遍历的时候做如下判断：
// 1. 若目前堆的大小小于K，将当前数字放入堆中。
// 2. 否则判断当前数字与大根堆堆顶元素的大小关系，如果当前数字比大根堆堆顶还大，这个数就直接跳过；
//    反之如果当前数字比大根堆堆顶小，先poll掉堆顶，再将该数字放入堆中。
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if (k == 0 || arr.length == 0) {
            return new int[0];
        }
        // 默认是小根堆，实现大根堆需要重写一下比较器。
        Queue<Integer> pq = new PriorityQueue<>((v1, v2) -> v2 - v1);
        for (int num: arr) {
            if (pq.size() < k) {
                pq.offer(num);
            } else if (num < pq.peek()) {
                pq.poll();
                pq.offer(num);
            }
        }

        // 返回堆中的元素
        int[] res = new int[pq.size()];
        int idx = 0;
        for(int num: pq) {
            res[idx++] = num;
        }
        return res;
    }
}
作者：sweetiee
链接：https://leetcode-cn.com/problems/zui-xiao-de-kge-shu-lcof/solution/3chong-jie-fa-miao-sha-topkkuai-pai-dui-er-cha-sou/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

- **滑动窗口最大值**

> 给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例 1：

输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置 最大值
--- -----
[1 3 -1] -3 5 3 6 7 3
1 [3 -1 -3] 5 3 6 7 3
1 3 [-1 -3 5] 3 6 7 5
1 3 -1 [-3 5 3] 6 7 5
1 3 -1 -3 [5 3 6] 7 6
1 3 -1 -3 5 [3 6 7] 7
示例 2：

输入：nums = [1], k = 1
输出：[1]
示例 3：

输入：nums = [1,-1], k = 1
输出：[1,-1]
示例 4：

输入：nums = [9,11], k = 2
输出：[11]
示例 5：

输入：nums = [4,-2], k = 2
输出：[4]

提示：

1 <= nums.length <= 105
-104 <= nums[i] <= 104
1 <= k <= nums.length

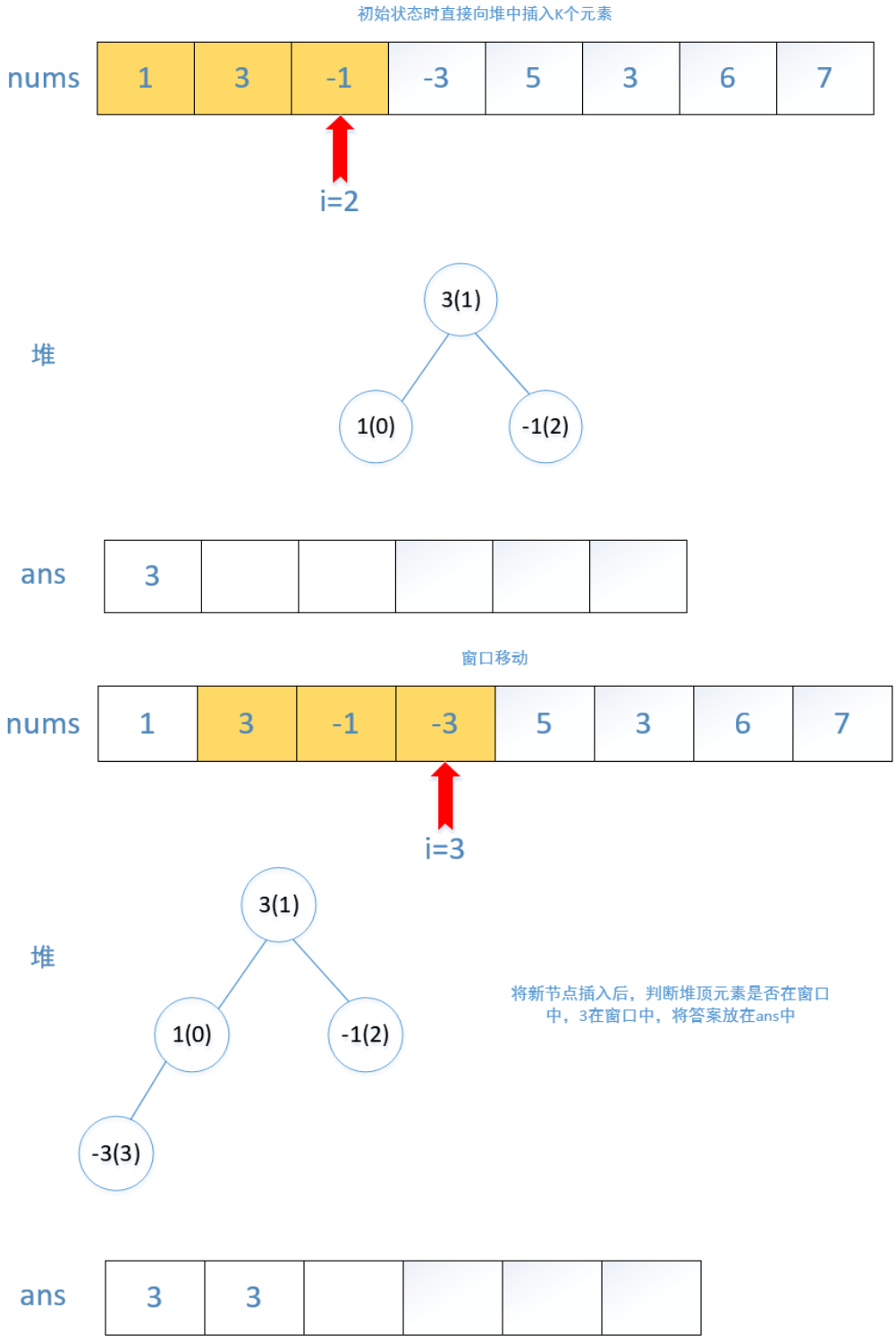
来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/sliding-window-maximum
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

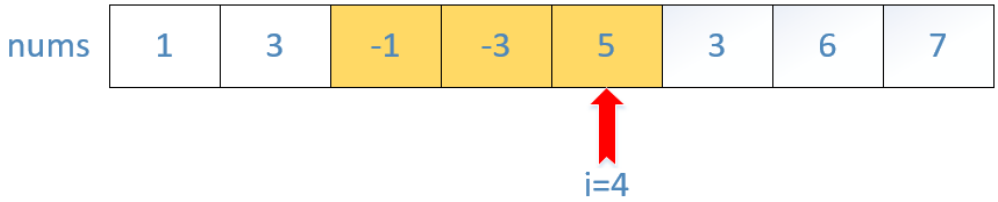
最大堆(优先队列)

我们可以采用最大堆的数据结构来保存元素，堆顶元素即为当前堆的最大值，并判断当前堆顶元素这是否在窗口中，在则直接返回，不在则删除堆顶元素并调整堆。

我们以nums = [1,3,−1,−3,5,3,6,7], k = 3为例来模拟堆的过程。

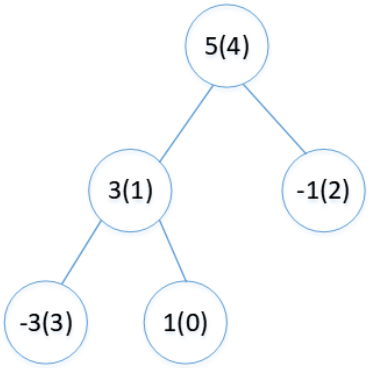


窗口移动



i=4

堆



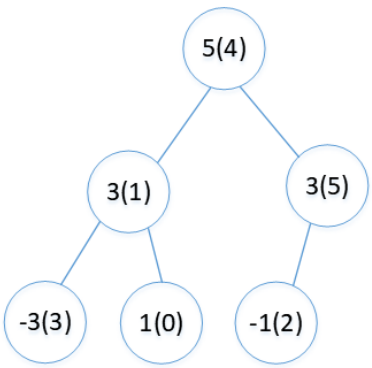
将新节点插入后，判断堆顶元素是否在窗口中，5在窗口中，将答案放在ans中

ans

窗口移动

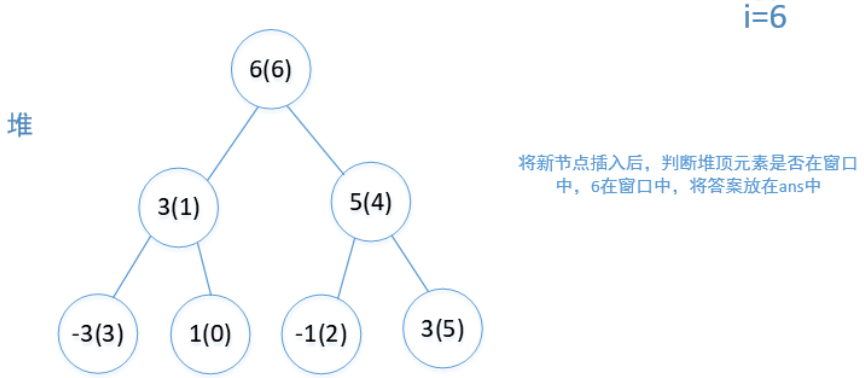
i=5

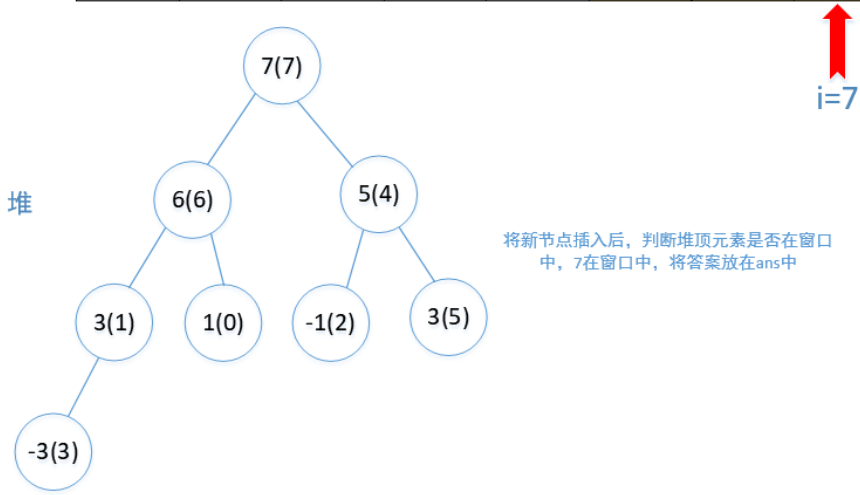
堆



将新节点插入后，判断堆顶元素是否在窗口中，5在窗口中，将答案放在ans中

ans





上述如何创建堆和调整堆暂不讨论,感兴趣可自行百度。在Java中的优先队列(PriorityQueue)就是堆的数据结构。代码如下：

```
public int[] maxSlidingWindow(int[] nums, int k) {
    int n = nums.length;
    //这里我们传入了一个比较器，当两者的值相同时，比较下标的位置，下标大的在前面。
    PriorityQueue<int[]> queue = new PriorityQueue<>((p1, p2) -> p1[0] != p2[0] ? p2[0] - p1[0] : p2[1] - p1[1]);
    //初始化前K的元素到堆中
    for (int i = 0; i < k; i++) {
        queue.offer(new int[]{nums[i], i});
```

```
    }
    //有n-k+1个
    int[] ans = new int[n - k + 1];
    //将第一次答案加入数据
    ans[0] = queue.peek()[0];
    for (int i = k; i < n; i++) {
        //将新元素加入优先队列
        queue.offer(new int[]{nums[i], i});
        //循环判断当前队首是否在窗口中，窗口的左边界为i-k
        while (queue.peek()[1] <= i - k) {
            queue.poll();
        }
        //在窗口中直接赋值即可
        ans[i - k + 1] = queue.peek()[0];
    }
    return ans;
}
作者：jiang-hui-4
链接：https://leetcode-cn.com/problems/sliding-window-maximum/solution/you-xian-dui-lie-zui-da-dui-dan-diao-dui-dbn9/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

- **丑数**

> 我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。
>
> 示例:
>
> 输入: n = 10
> 输出: 12
> 解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
> 说明:
>
> 1 是丑数。
> n 不超过1690。
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/chou-shu-lcof
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

解题思路

跟面试题 17.09. 第k个数一样的做法，任何丑数乘以2、3、5，其结果也是丑数（证明略），我们可以利用小根堆，然后1作为第一个丑数，每次从小根堆弹出最小的丑数，然后记录已弹

出丑数的个数，如果count>=n,返回当前弹出的元素，否则继续乘以2、3、5，（注意：放入堆里的元素需要排除重复值）。

代码

```
class Solution {
    private int[] uglyNumber = {2,3,5};
    public int nthUglyNumber(int n) {
        Set<Long> set = new HashSet<>();
        //创建小根堆，每次出堆的都是最小值
        PriorityQueue<Long> queue = new PriorityQueue<>();
        queue.add(1L);
        //记录出堆的个数，出堆的元素完全按照从小到大排序
        int count = 0;
        while (! queue.isEmpty()){
            long cut = queue.poll();
            //如果出堆的个数>=n,当前cut就是第n个丑数
            if(++count >= n){
                return (int) cut;
            }
            for(int num : uglyNumber){
                //排除重复的数字
                if(! set.contains(num * cut)){
                    queue.add(num * cut);
                    set.add(num * cut);
                }
            }
        }
        return -1;
    }
}
作者：YanShaoJiangHu
链接：https://leetcode-cn.com/problems/chou-shu-lcof/solution/li-yong-xiao-gen-dui-wan-mei-jie-jue-by-yanshaojia/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

- **前K个高频元素**

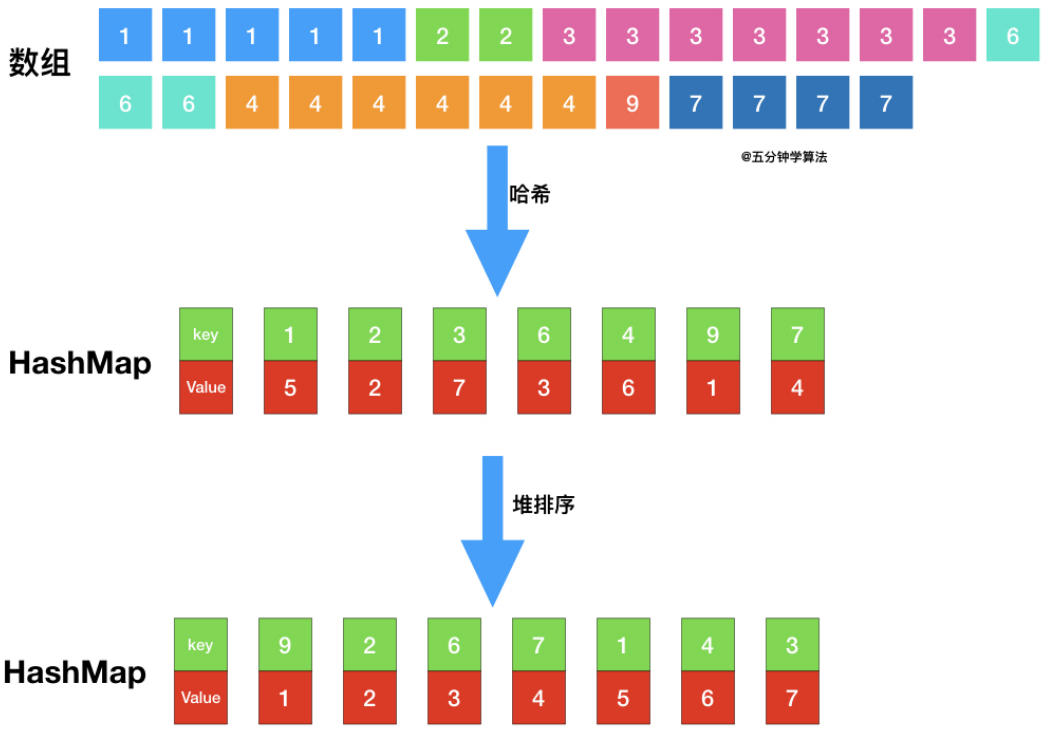
> 给定一个非空的整数数组，返回其中出现频率前 k 高的元素。
>
> 示例 1:
>
> 输入: nums = [1,1,1,2,2,3], k = 2
> 输出: [1,2]
> 示例 2:
>
> 输入: nums = [1], k = 1
> 输出: [1]

> 提示：
>
> 你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
> 你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小。
> 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。
> 你可以按任意顺序返回答案。
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/top-k-frequent-elements
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

最小堆

题目最终需要返回的是前 kk 个频率最大的元素，可以想到借助堆这种数据结构，对于 kk 频率之后的元素不用再去处理，进一步优化时间复杂度。



具体操作为：

借助 哈希表 来建立数字和其出现次数的映射，遍历一遍数组统计元素的频率
维护一个元素数目为 kk 的最小堆
每次都将新的元素与堆顶元素（堆中频率最小的元素）进行比较
如果新的元素的频率比堆顶端的元素大，则弹出堆顶端的元素，将新的元素添加进堆中
最终，堆中的 kk 个元素即为前 kk 个高频元素
https://pic.leetcode-cn.com/b548a3796066fa7072baa2b1e06e0d54641a7913d87c88c61d73b6b9ad0e90db-file_1561712388100

代码如下：

```
class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        // 使用字典，统计每个元素出现的次数，元素为键，元素出现的次数为值
        HashMap<Integer,Integer> map = new HashMap();
        for(int num : nums){
            if (map.containsKey(num)) {
               map.put(num, map.get(num) + 1);
             } else {
                map.put(num, 1);
             }
        }
        // 遍历map，用最小堆保存频率最大的k个元素
        PriorityQueue<Integer> pq = new PriorityQueue<>(new
Comparator<Integer>() {
            @Override
            public int compare(Integer a, Integer b) {
                return map.get(a) - map.get(b);
            }
        });
        for (Integer key : map.keySet()) {
            if (pq.size() < k) {
                pq.add(key);
            } else if (map.get(key) > map.get(pq.peek())) {
                pq.remove();
                pq.add(key);
            }
        }
        // 取出最小堆中的元素
        List<Integer> res = new ArrayList<>();
        while (!pq.isEmpty()) {
            res.add(pq.remove());
        }
        return res;
    }
}
```

复杂度分析

时间复杂度：O(nlogk)O(nlogk)，nn 表示数组的长度。首先，遍历一遍数组统计元素的频率，这一系列操作的时间复杂度是 O(n)O(n)；接着，遍历用于存储元素频率的 map，如果元素的频率大于最小堆中顶部的元素，则将顶部的元素删除并将该元素加入堆中，这里维护堆的数目是 kk，所以这一系列操作的时间复杂度是 O(nlogk)O(nlogk) 的；因此，总的时间复杂度是 O(nlogk)O(nlogk)。

空间复杂度：O(n)O(n)，最坏情况下（每个元素都不同），map 需要存储 nn 个键值对，优先队列需要存储 kk 个元素，因此，空间复杂度是 O(n)O(n)。

作者：MisterBooo

链接：https://leetcode-cn.com/problems/top-k-frequent-elements/solution/leetcode-di-347-hao-wen-ti-qian-k-ge-gao-pin-yuan-/

来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **有效的括号**

> 给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。
>
> 有效字符串需满足：
>
> 左括号必须用相同类型的右括号闭合。
> 左括号必须以正确的顺序闭合。
>
> 示例 1：
>
> 输入：s = "()"
> 输出：true
> 示例 2：
>
> 输入：s = "()[]{}"
> 输出：true
> 示例 3：
>
> 输入：s = "(]"
> 输出：false
> 示例 4：
>
> 输入：s = "([)]"
> 输出：false
> 示例 5：
>
> 输入：s = "{[]}"
> 输出：true
>
> 提示：
>
> 1 <= s.length <= 104
> s 仅由括号 '()[]{}' 组成
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/valid-parentheses
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
解题思路：
算法原理

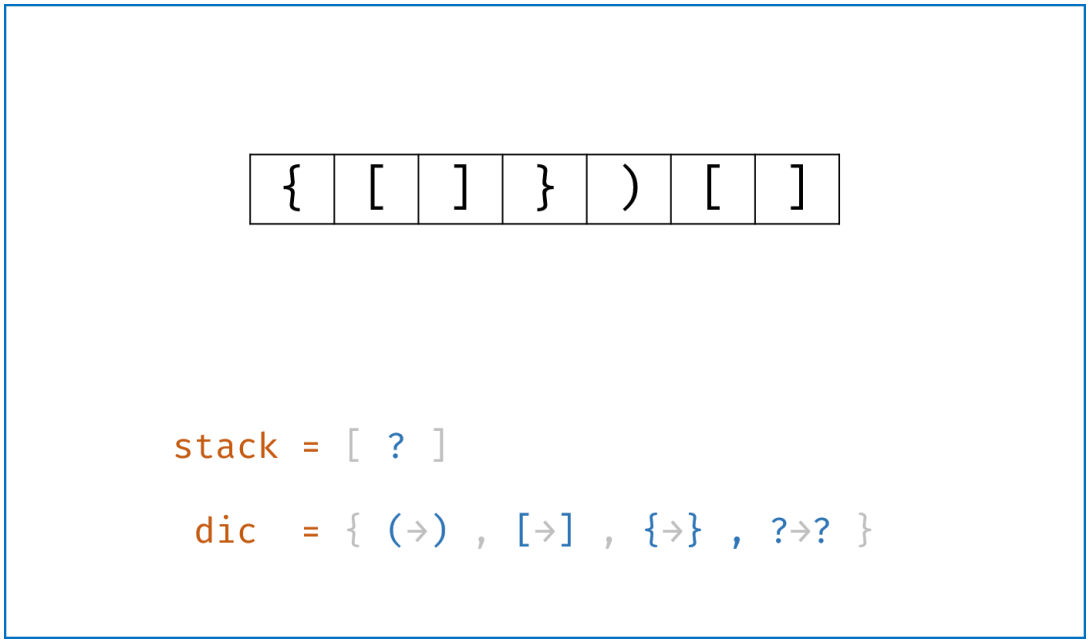
栈先入后出特点恰好与本题括号排序特点一致，即若遇到左括号入栈，遇到右括号时将对应栈顶左括号出栈，则遍历完所有括号后 stack 仍然为空；

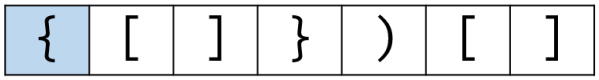
建立哈希表 dic 构建左右括号对应关系：keykey 左括号，valuevalue 右括号；这样查询 22 个括号是否对应只需 O(1)O(1) 时间复杂度；建立栈 stack，遍历字符串 s 并按照算法流程一一判断。
算法流程

如果 c 是左括号，则入栈 pushpush；
否则通过哈希表判断括号对应关系，若 stack 栈顶出栈括号 stack.pop() 与当前遍历括号 c 不对应，则提前返回 falsefalse。
提前返回 falsefalse

提前返回优点： 在迭代过程中，提前发现不符合的括号并且返回，提升算法效率。
解决边界问题：
栈 stack 为空： 此时 stack.pop() 操作会报错；因此，我们采用一个取巧方法，给 stack 赋初值 ?? ，并在哈希表 dic 中建立 key: '?'，value:'?'key:
'
?
'
，value:
'
?
'
的对应关系予以配合。此时当 stack 为空且 c 为右括号时，可以正常提前返回 falsefalse；
字符串 s 以左括号结尾： 此情况下可以正常遍历完整个 s，但 stack 中遗留未出栈的左括号；因此，最后需返回 len(stack) == 1，以判断是否是有效的括号组合。
复杂度分析

时间复杂度 O(N)O(N)：正确的括号组合需要遍历 11 遍 s；
空间复杂度 O(N)O(N)：哈希表和栈使用线性的空间大小。

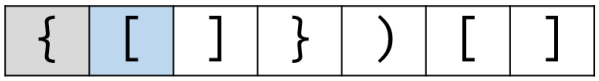




c

stack = [? , {]

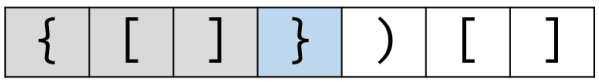
dic = { (→) , [→] , {→} , ?→? }



c

stack = [? , { , [ ]

dic = { (→) , [→] , {→} , ?→? }

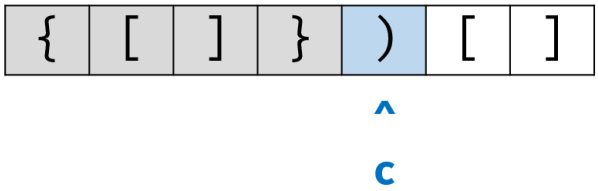


c

stack = [?]

dic = { (→) , [→] , {→} , ?→? }

符合



stack = [ ]

dic = { (→) , [→] , {→} , ?→? }

? 与) 不对应，直接返回false

代码：

```
class Solution {
    private static final Map<Character,Character> map = new
HashMap<Character,Character>(){{
        put('{','}'); put('[',']'); put('(',')'); put('?','?');
    }};
    public boolean isValid(String s) {
        if(s.length() > 0 && !map.containsKey(s.charAt(0))) return false;
        LinkedList<Character> stack = new LinkedList<Character>() {{
add('?'); }};
        for(Character c : s.toCharArray()){
            if(map.containsKey(c)) stack.addLast(c);
            else if(map.get(stack.removeLast()) != c) return false;
        }
        return stack.size() == 1;
    }
}
作者：jyd
链接：https://leetcode-cn.com/problems/valid-parentheses/solution/valid-
parentheses-fu-zhu-zhan-fa-by-jin407891080/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

- **最小栈**

> 设计一个支持 push ， pop ， top 操作，并能在常数时间内检索到最小元素的栈。
>
> push(x) —— 将元素 x 推入栈中。
> pop() —— 删除栈顶的元素。
> top() —— 获取栈顶元素。
> getMin() —— 检索栈中的最小元素。

> 示例:
>
> 输入:
> ["MinStack","push","push","push","getMin","pop","top","getMin"]
> [[],[-2],[0],[-3],[],[],[],[]]
>
> 输出:
> [null,null,null,null,-3,null,0,-2]
>
> 解释:
> MinStack minStack = new MinStack();
> minStack.push(-2);
> minStack.push(0);
> minStack.push(-3);
> minStack.getMin(); --> 返回 -3.
> minStack.pop();
> minStack.top(); --> 返回 0.
> minStack.getMin(); --> 返回 -2.
>
> 提示:
>
> pop、top 和 getMin 操作总是在 非空栈 上调用。
>
> 来源:力扣(LeetCode)
> 链接:https://leetcode-cn.com/problems/min-stack
> 著作权归领扣网络所有。商业转载请联系官方授权,非商业转载请注明出处。

参考回答:
这道题的思想很简单:“以空间换时间”,使用辅助栈是常见的做法。

思路分析:
在代码实现的时候有两种方式:

1、辅助栈和数据栈同步

特点:编码简单,不用考虑一些边界情况,就有一点不好:辅助栈可能会存一些“不必要”的元素。

2、辅助栈和数据栈不同步

特点:由“辅助栈和数据栈同步”的思想,我们知道,当数据栈进来的数越来越大的时候,我们要在辅助栈顶放置和当前辅助栈顶一样的元素,这样做有点“浪费”。基于这一点,我们做一些“优化”,但是在编码上就要注意一些边界条件。

(1)辅助栈为空的时候,必须放入新进来的数;

（2）新来的数小于或者等于辅助栈栈顶元素的时候，才放入，特别注意这里“等于”要考虑进去，因为出栈的时候，连续的、相等的并且是最小值的元素要同步出栈；

（3）出栈的时候，辅助栈的栈顶元素等于数据栈的栈顶元素，才出栈。

总结一下：出栈时，最小值出栈才同步；入栈时，最小值入栈才同步。

对比：个人觉得“同步栈”的方式更好一些，因为思路清楚，因为所有操作都同步进行，所以调试代码、定位问题也简单。“不同步栈”，虽然减少了一些空间，但是在“出栈”、“入栈”的时候还要做判断，也有性能上的消耗。

方法一：辅助栈和数据栈同步
参考代码 1：

```
import java.util.Stack;
public class MinStack {
    // 数据栈
    private Stack<Integer> data;
    // 辅助栈
    private Stack<Integer> helper;
    /**
     * initialize your data structure here.
     */
    public MinStack() {
        data = new Stack<>();
        helper = new Stack<>();
    }
    // 思路 1：数据栈和辅助栈在任何时候都同步
    public void push(int x) {
        // 数据栈和辅助栈一定会增加元素
        data.add(x);
        if (helper.isEmpty() || helper.peek() >= x) {
            helper.add(x);
        } else {
            helper.add(helper.peek());
        }
    }
    public void pop() {
        // 两个栈都得 pop
        if (!data.isEmpty()) {
            helper.pop();
            data.pop();
        }
    }
    public int top() {
        if(!data.isEmpty()){
            return data.peek();
        }
        throw new RuntimeException("栈中元素为空，此操作非法");
```

```
    }
    public int getMin() {
        if(!helper.isEmpty()){
            return helper.peek();
        }
        throw new RuntimeException("栈中元素为空，此操作非法");
    }
}
```

复杂度分析：

时间复杂度：O(1)O(1)，“出栈”、“入栈”、“查看栈顶元素”的操作不论数据规模多大，都只是有限个步骤，因此时间复杂度是：O(1)O(1)。
空间复杂度：O(N)O(N)，这里 NN 是读出的数据的个数。
方法二：辅助栈和数据栈不同步
参考代码 2：

```
import java.util.Stack;
public class MinStack {
    // 数据栈
    private Stack<Integer> data;
    // 辅助栈
    private Stack<Integer> helper;
    /**
     * initialize your data structure here.
     */
    public MinStack() {
        data = new Stack<>();
        helper = new Stack<>();
    }
    // 思路 2：辅助栈和数据栈不同步
    // 关键 1：辅助栈的元素空的时候，必须放入新进来的数
    // 关键 2：新来的数小于或者等于辅助栈栈顶元素的时候，才放入（特别注意这里等于要考虑进去）
    // 关键 3：出栈的时候，辅助栈的栈顶元素等于数据栈的栈顶元素，才出栈，即"出栈保持同步"就可以了
    public void push(int x) {
        // 辅助栈在必要的时候才增加
        data.add(x);
        // 关键 1 和 关键 2
        if (helper.isEmpty() || helper.peek() >= x) {
            helper.add(x);
        }
    }
    public void pop() {
```

```
        // 关键 3：data 一定得 pop()
        if (!data.isEmpty()) {
            // 注意：声明成 int 类型，这里完成了自动拆箱，从 Integer 转成了 int，因此下面的比较可以使用 "==" 运算符
            // 参考资料：https://www.cnblogs.com/GuoYaxiang/p/6931264.html
            // 如果把 top 变量声明成 Integer 类型，下面的比较就得使用 equals 方法
            int top = data.pop();
            if(top == helper.peek()){
                helper.pop();
            }
        }
    }
    public int top() {
        if(!data.isEmpty()){
            return data.peek();
        }
        throw new RuntimeException("栈中元素为空，此操作非法");
    }
    public int getMin() {
        if(!helper.isEmpty()){
            return helper.peek();
        }
        throw new RuntimeException("栈中元素为空，此操作非法");
    }
}
```

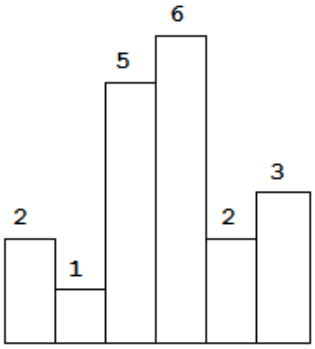
复杂度分析：

时间复杂度：O(1)O(1)，“出栈”、“入栈”、“查看栈顶元素”的操作不论数据规模多大，都只有有限个步骤，因此时间复杂度是：O(1)O(1)。
空间复杂度：O(N)O(N)，这里 NN 是读出的数据的个数。

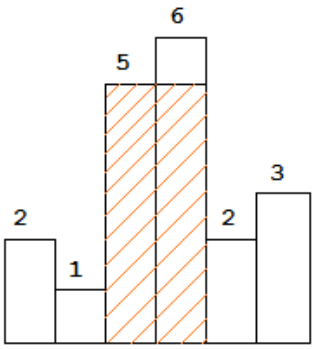
作者：liweiwei1419
链接：https://leetcode-cn.com/problems/min-stack/solution/shi-yong-fu-zhu-zhan-tong-bu-he-bu-tong-bu-python-/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **柱状图中最大的矩形**

> 给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。
>
> 求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2,1,5,6,2,3]。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

示例:

输入: [2,1,5,6,2,3]
输出: 10

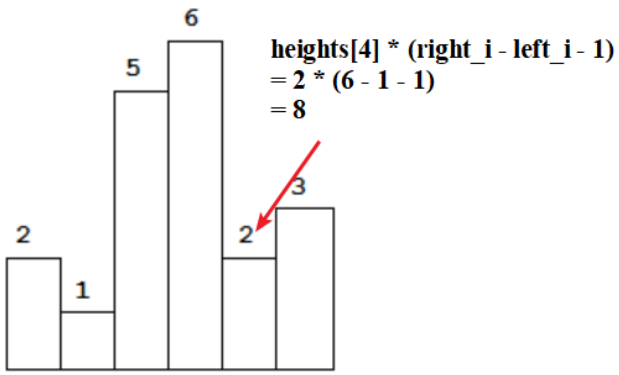
来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/largest-rectangle-in-histogram
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答:

思路:

首先，要想找到第 i 位置最大面积是什么?

是以i 为中心，向左找第一个小于 heights[i] 的位置 left_i；向右找第一个小于于 heights[i] 的位置 right_i，即最大面积为 heights[i] * (right_i – left_i –1)，如下图所示:



所以，我们的问题就变成如何找 right_i 和 left_i?

最简单的思路就是，就是暴力法，直接分别在 i 左右移动。
但是，这是一个时间复杂度为 O(n^2)O(n
2
)，超时。

接下来想办法优化。

思路一：
当我们找 i 左边第一个小于 heights[i] 如果 heights[i−1] >= heights[i] 其实就是和 heights[i−1] 左边第一个小于 heights[i−1] 一样。依次类推，右边同理。

思路二：栈
利用单调栈

维护一个单调递增的栈，就可以找到 left_i 和 right_i。

代码：
思路一：

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        if (heights == null || heights.length == 0) return 0;
        int n = heights.length;
        int[] left_i = new int[n];
        int[] right_i = new int[n];
        left_i[0] = -1;
        right_i[n - 1] = n;
        int res = 0;
        for (int i = 1; i < n; i++) {
            int tmp = i - 1;
            while (tmp >= 0 && heights[tmp] >= heights[i]) tmp = left_i[tmp];
            left_i[i] = tmp;
        }
        for (int i = n - 2; i >= 0; i--) {
            int tmp = i + 1;
            while (tmp < n && heights[tmp] >= heights[i]) tmp = right_i[tmp];
            right_i[i] = tmp;
        }
        for (int i = 0; i < n; i++) res = Math.max(res, (right_i[i] - left_i[i] - 1) * heights[i]);
```

```
        return res;
    }
}
```

思路二：

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        int res = 0;
        Deque<Integer> stack = new ArrayDeque<>();
        int[] new_heights = new int[heights.length + 2];
        for (int i = 1; i < heights.length + 1; i++) new_heights[i] =
heights[i - 1];
        //System.out.println(Arrays.toString(new_heights));
        for (int i = 0; i < new_heights.length; i++) {
            //System.out.println(stack.toString());
            while (!stack.isEmpty() && new_heights[stack.peek()] >
new_heights[i]) {
                int cur = stack.pop();
                res = Math.max(res, (i - stack.peek() - 1) *
new_heights[cur]);
            }
            stack.push(i);
        }
        return res;
    }
}
作者：powcai
链接：https://leetcode-cn.com/problems/largest-rectangle-in-
histogram/solution/zhao-liang-bian-di-yi-ge-xiao-yu-ta-de-zhi-by-powc/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

1.7 | 高级算法

- **请你讲讲LRU算法的实现原理?**

参考回答：

①LRU（Least recently used，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也很高”，反过来说“如果数据最近这段时间一直都没有访问,那么将来被访问的概率也会很低”，两种理解是一样的；常用于页面置换算法，为虚拟页式存储管理服务。

②达到这样一种情形的算法是最理想的：每次调换出的页面是所有内存页面中最迟将被使用的；这可以最大限度的推迟页面调换，这种算法，被称为理想页面置换算法。可惜的是，这种算法是无法实现的。

为了尽量减少与理想算法的差距，产生了各种精妙的算法，最近最少使用页面置换算法便是其中一个。LRU 算法的提出，是基于这样一个事实：在前面几条指令中使用频繁的页面很可能在后面的几条指令中频繁使用。反过来说，已经很久没有使用的页面很可能在未来较长的一段时间内不会被用到 。这个，就是著名的局部性原理——比内存速度还要快的cache，也是基于同样的原理运行的。因此，我们只需要在每次调换时，找到最近最少使用的那个页面调出内存。

算法实现的关键

命中率：

当存在热点数据时，LRU的效率很好，但偶发性的、周期性的批量操作会导致 LRU 命中率急剧下降，缓存污染情况比较严重。

复杂度：

实现起来较为简单。

存储成本：

几乎没有空间上浪费。

代价：

命中时需要遍历链表，找到命中的数据块索引，然后需要将数据移到头部。

- **为什么要设计 后缀表达式，有什么好处?**

参考回答：

后缀表达式又叫逆波兰表达式，逆波兰记法不需要括号来标识操作符的优先级。

- **请你设计一个算法，用来压缩一段URL?**

参考回答：

该算法主要使用MD5 算法对原始链接进行加密（这里使用的MD5 加密后的字符串长度为 32 位），然后对加密后的字符串进行处理以得到短链接的地址。

- **谈一谈，id全局唯一且自增，如何实现?**

参考回答：

SnowFlake雪花算法

雪花ID生成的是一个64位的二进制正整数，然后转换成10进制的数。64位二进制数由如下部分组成：

snowflake id生成规则

1位标识符：始终是0，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0。

41位时间戳：41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 – 开始时间截)得到的值，这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的。

10位机器标识码：可以部署在1024个节点，如果机器分机房（IDC）部署，这10位可以由 5位机房ID + 5位机器ID 组成。

12位序列：毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号

优点

简单高效，生成速度快。

时间戳在高位，自增序列在低位，整个ID是趋势递增的，按照时间有序递增。

灵活度高，可以根据业务需求，调整bit位的划分，满足不同的需求。

- **最后一个单词的长度**

> 给你一个字符串 s，由若干单词组成，单词之间用空格隔开。返回字符串中最后一个单词的长度。如果不存在最后一个单词，请返回 0 。
>
> 单词 是指仅由字母组成、不包含任何空格字符的最大子字符串。
>
> 示例 1：
>
> 输入：s = "Hello World"
> 输出：5
> 示例 2：
>
> 输入：s = " "
> 输出：0
>
> 提示：
>
> 1 <= s.length <= 104
> s 仅有英文字母和空格 ' ' 组成
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/length-of-last-word
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

思路

标签：字符串遍历

从字符串末尾开始向前遍历，其中主要有两种情况

第一种情况，以字符串”Hello World”为例，从后向前遍历直到遍历到头或者遇到空格为止，即为最后一个单词”World”的长度5

第二种情况，以字符串”Hello World ”为例，需要先将末尾的空格过滤掉，再进行第一种情况的操作，即认为最后一个单词为”World”，长度为5

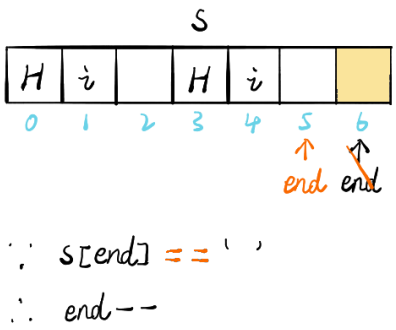
所以完整过程为先从后过滤掉空格找到单词尾部，再从尾部向前遍历，找到单词头部，最后两者相减，即为单词的长度

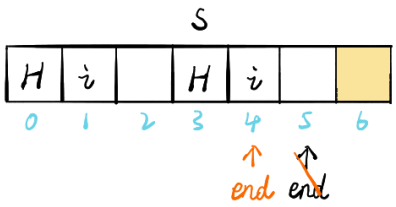
时间复杂度：O(n)，n为结尾空格和结尾单词总体长度

代码

```
class Solution {
    public int lengthOfLastWord(String s) {
        int end = s.length() - 1;
        while(end >= 0 && s.charAt(end) == ' ') end--;
        if(end < 0) return 0;
        int start = end;
        while(start >= 0 && s.charAt(start) != ' ') start--;
        return end - start;
    }
}
```

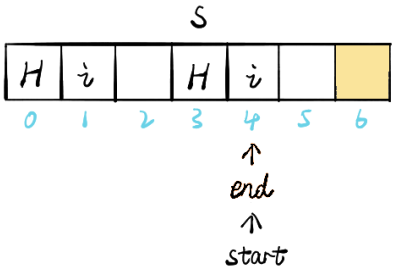
画解





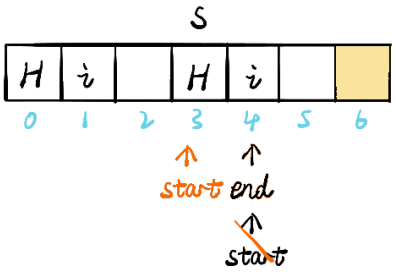
∵ s[end] == ' '

∴ end--



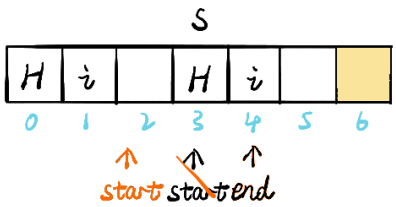
∵ s[end] != ' '

∴ start = end



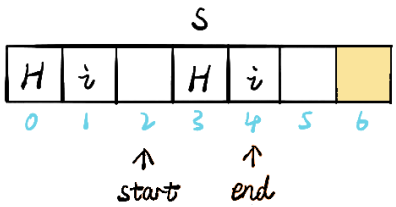
∵ s[start] != ' '

∴ start--



∵ s[start] != ' '

∴ start--



∵ s[start] == ' '

∴ end - start = 4 - 2 = 2

作者：guanpengchn

链接：https://leetcode-cn.com/problems/length-of-last-word/solution/hua-jie-suan-fa-58-zui-hou-yi-ge-dan-ci-de-chang-d/

来源：力扣（LeetCode）

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

1.8 | 动态规划

- **斐波那契数**

> 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
> 给你 n ，请计算 F(n) 。
>
> 示例 1：

> 输入：2
> 输出：1
> 解释：F(2) = F(1) + F(0) = 1 + 0 = 1
> 示例 2：
>
> 输入：3
> 输出：2
> 解释：F(3) = F(2) + F(1) = 1 + 1 = 2
> 示例 3：
>
> 输入：4
> 输出：3
> 解释：F(4) = F(3) + F(2) = 2 + 1 = 3
>
> 提示：
> 0 <= n <= 30
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/fibonacci-number
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

```
public int fib(int n) {
    if(n<2) return n;
    int[] f= new int[n+1];
    f[0] = 0;
    f[1]= 1;
    for(int i = 2;i<=n;i++) f[i] =f[i-1]+f[i-2];
    return f[n];
}
作者：a-fei-8
链接：https://leetcode-cn.com/problems/fibonacci-number/solution/chang-you-mian-shi-zhong-de-dong-tai-gui-e1ph/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

- **不同路径**

> 一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start”）。
>
> 机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。
>
> 问总共有多少条不同的路径?

示例 1：

输入：m = 3, n = 7
输出：28
示例 2：

输入：m = 3, n = 2
输出：3
解释：
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下
示例 3：

输入：m = 7, n = 3
输出：28
示例 4：

输入：m = 3, n = 3
输出：6

提示：

1 <= m, n <= 100
题目数据保证答案小于等于 2 * 109

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/unique-paths
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/unique-paths
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
我们令 dp[i][j] 是到达 i, j 最多路径

动态方程：dp[i][j] = dp[i-1][j] + dp[i][j-1]

注意，对于第一行 dp[0][j]，或者第一列 dp[i][0]，由于都是在边界，所以只能为 1

时间复杂度：O(m*n)O(m∗n)

空间复杂度：O(m * n)O(m∗n)

优化：因为我们每次只需要 dp[i−1][j],dp[i][j−1]

所以我们只要记录这两个数，直接看代码吧！

代码
思路：

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        for (int i = 0; i < n; i++) dp[0][i] = 1;
        for (int i = 0; i < m; i++) dp[i][0] = 1;
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m - 1][n - 1];
    }
}
```

优化1：空间复杂度 O(2n)

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[] pre = new int[n];
        int[] cur = new int[n];
        Arrays.fill(pre, 1);
        Arrays.fill(cur,1);
        for (int i = 1; i < m;i++){
            for (int j = 1; j < n; j++){
                cur[j] = cur[j-1] + pre[j];
            }
            pre = cur.clone();
        }
        return pre[n-1];
    }
}
```

优化2：空间复杂度 O(n)

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[] cur = new int[n];
        Arrays.fill(cur,1);
        for (int i = 1; i < m;i++){
            for (int j = 1; j < n; j++){
                cur[j] += cur[j-1] ;
            }
        }
        return cur[n-1];
    }
}
```

作者：powcai

链接：https://leetcode-cn.com/problems/unique-paths/solution/dong-tai-gui-hua-by-powcai-2/

来源：力扣（LeetCode）

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **爬楼梯**

> 假设你正在爬楼梯。需要 n 阶你才能到达楼顶。
>
> 每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?
>
> 注意：给定 n 是一个正整数。
>
> 示例 1:
>
> 输入： 2
> 输出： 2
> 解释： 有两种方法可以爬到楼顶。
> 1. 1 阶 + 1 阶
> 2. 2 阶
> 示例 2:
>
> 输入： 3
> 输出： 3
> 解释： 有三种方法可以爬到楼顶。
> 1. 1 阶 + 1 阶 + 1 阶
> 2. 1 阶 + 2 阶
> 3. 2 阶 + 1 阶
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/climbing-stairs
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

本问题其实常规解法可以分成多个子问题，爬第n阶楼梯的方法数量，等于 2 部分之和

爬上 n−1n−1 阶楼梯的方法数量。因为再爬1阶就能到第n阶

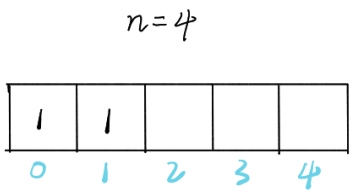
爬上 n−2n−2 阶楼梯的方法数量，因为再爬2阶就能到第n阶

所以我们得到公式 dp[n] = dp[n−1] + dp[n−2]dp[n]=dp[n−1]+dp[n−2]

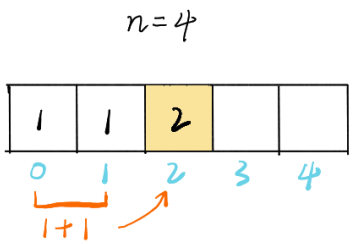
同时需要初始化 dp[0]=1dp[0]=1 和 dp[1]=1dp[1]=1

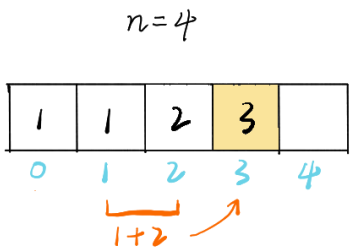
时间复杂度：O(n)O(n)

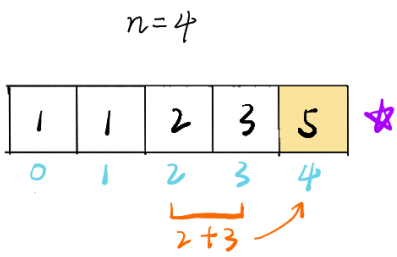
画解



初始化







代码

```
class Solution {
    public int climbStairs(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        for(int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

作者：guanpengchn
链接：https://leetcode-cn.com/problems/climbing-stairs/solution/hua-jie-suan-fa-70-pa-lou-ti-by-guanpengchn/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **零钱兑换**

> 给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
>
> 你可以认为每种硬币的数量是无限的。
>
> 示例 1：
>
> 输入：coins = [1, 2, 5], amount = 11
> 输出：3
> 解释：11 = 5 + 5 + 1
> 示例 2：
>
> 输入：coins = [2], amount = 3
> 输出：-1
> 示例 3：
>
> 输入：coins = [1], amount = 0
> 输出：0
> 示例 4：
>
> 输入：coins = [1], amount = 1
> 输出：1
> 示例 5：
>
> 输入：coins = [1], amount = 2
> 输出：2
>
> 提示：
>
> 1 <= coins.length <= 12
> 1 <= coins[i] <= 231 - 1
> 0 <= amount <= 104
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/coin-change
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

看题目的问法，只问最优值是多少，没有要我们求最优解，一般情况下可以用「动态规划」解决。

动态规划与记忆化递归
思路：分析最优子结构。根据示例 1：

输入: coins = [1, 2, 5], amount = 11
凑成面值为 1111 的最少硬币个数可以由以下三者的最小值得到：

凑成面值为 1010 的最少硬币个数 + 面值为 1 的这一枚硬币；
凑成面值为 99 的最少硬币个数 + 面值为 2 的这一枚硬币；
凑成面值为 66 的最少硬币个数 + 面值为 5 的这一枚硬币。
即 dp[11] = min (dp[10] + 1, dp[9] + 1, dp[6] + 1)。

可以直接把问题的问法设计成状态。

第 1 步：定义「状态」。dp[i]：凑齐总价值 i 需要的最少硬币个数；
第 2 步：写出「状态转移方程」。根据对示例 1 的分析：

dp[amount] = min(dp[amount], 1 + dp[amount − coins[i]]) for i in [0, len − 1] if coins[i] <= amount
说明：感谢 @paau 朋友纠正状态转移方程。

注意：

单枚硬币的面值首先要小于等于 当前要凑出来的面值；
剩余的那个面值也要能够凑出来，例如：求 dp[11] 需要参考 dp[10]。如果不能凑出 dp[10]，则 dp[10] 应该等于一个不可能的值，可以设计为 11 + 1，也可以设计为 -1 ，它们的区别只是在编码的细节上不一样。
再次强调：新状态的值要参考的值以前计算出来的「有效」状态值。因此，不妨先假设凑不出来，因为求的是小，所以设置一个不可能的数。

参考代码 ：

注意：要求的是「恰好凑出面值」，所以初始化的时候需要赋值为一个不可能的值：amount + 1。只有在有「正常值」的时候，「状态转移」才可以正常发生。

```
import java.util.Arrays;
public class Solution {
    public int coinChange(int[] coins, int amount) {
        // 给 0 占位
        int[] dp = new int[amount + 1];
```

```
        // 注意：因为要比较的是最小值，这个不可能的值就得赋值成为一个最大值
        Arrays.fill(dp, amount + 1);
        // 理解 dp[0] = 0 的合理性，单独一枚硬币如果能够凑出面值，符合最优子结构
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (i - coin >= 0 && dp[i - coin] != amount + 1) {
                    dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
                }
            }
        }
        if (dp[amount] == amount + 1) {
            dp[amount] = -1;
        }
        return dp[amount];
    }
}
```

复杂度分析：

时间复杂度：O(N \times amount)O(N×amount)，这里 NN 是可选硬币的种类数，amountamount 是题目输入的面值；
空间复杂度：O(amount)O(amount)，状态数组的大小为 amountamount。
「动态规划」是「自底向上」求解。事实上，可以 直接面对问题求解 ，即「自顶向下」，但是这样的问题有 重复子问题，需要缓存已经求解过的答案，这叫 记忆化递归。

作者：liweiwei1419
链接：https://leetcode-cn.com/problems/coin-change/solution/dong-tai-gui-hua-shi-yong-wan-quan-bei-bao-wen-ti-/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **打家劫舍**

> 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。
>
> 给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。
>
> 示例 1：
>
> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1)，然后偷窃 3 号房屋 (金额 = 3)。
> 偷窃到的最高金额 = 1 + 3 = 4 。
> 示例 2：

> 输入：[2,7,9,3,1]
> 输出：12
> 解释：偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
> 偷窃到的最高金额 = 2 + 9 + 1 = 12 。
>
> 提示：
> 0 <= nums.length <= 100
> 0 <= nums[i] <= 400
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/house-robber
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

思路

标签：动态规划

动态规划方程：dp[n] = MAX(dp[n-1], dp[n-2] + num)

由于不可以在相邻的房屋闯入，所以在当前位置 n 房屋可盗窃的最大值，要么就是 n-1 房屋可盗窃的最大值，要么就是 n-2 房屋可盗窃的最大值加上当前房屋的值，二者之间取最大值

举例来说：1 号房间可盗窃最大值为 33 即为 dp[1]=3，2 号房间可盗窃最大值为 44 即为 dp[2]=4，3 号房间自身的值为 22 即为 num=2，那么 dp[3] = MAX(dp[2], dp[1] + num) = MAX(4, 3+2) = 5，3 号房间可盗窃最大值为 55

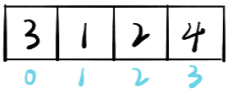
时间复杂度：O(n)O(n)，nn 为数组长度

代码

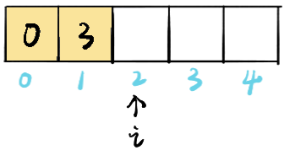
```
class Solution {
    public int rob(int[] nums) {
        int len = nums.length;
        if(len == 0)
            return 0;
        int[] dp = new int[len + 1];
        dp[0] = 0;
        dp[1] = nums[0];
        for(int i = 2; i <= len; i++) {
            dp[i] = Math.max(dp[i-1], dp[i-2] + nums[i-1]);
        }
        return dp[len];
    }
}
```

画解

nums

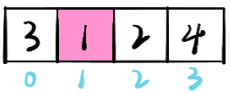


dp

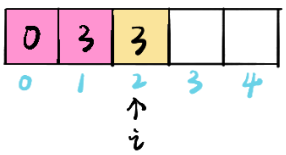


$dp[0] = 0 \quad dp[1] = nums[0]$

nums



dp

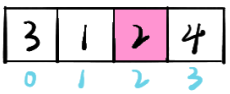


$dp[i] = MAX(dp[i-1], dp[i-2] + nums[i-1])$

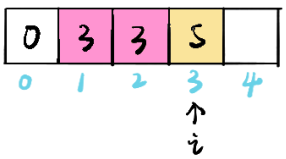
$dp[2] = MAX(dp[1], dp[0] + nums[1])$

$= MAX(3, 0+1) = 3$

nums



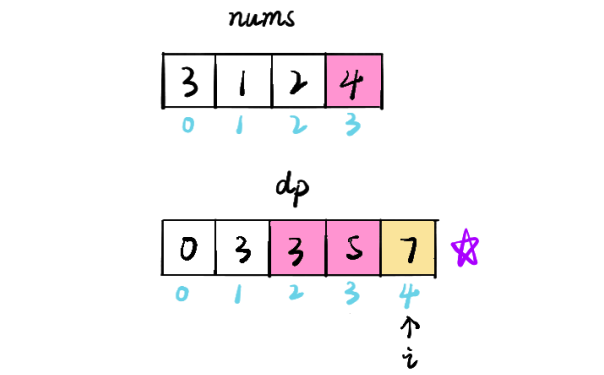
dp



$dp[i] = MAX(dp[i-1], dp[i-2] + nums[i-1])$

$dp[3] = MAX(dp[2], dp[1] + nums[2])$

$= MAX(3, 3+2) = 5$



作者：guanpengchn
链接：https://leetcode-cn.com/problems/house-robber/solution/hua-jie-suan-fa-198-da-jia-jie-she-by-guanpengchn/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **编辑距离**

> 给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数。
>
> 你可以对一个单词进行如下三种操作：
>
> 插入一个字符
> 删除一个字符
> 替换一个字符
>
> 示例 1：
>
> 输入：word1 = "horse", word2 = "ros"
> 输出：3
> 解释：
> horse -> rorse (将 'h' 替换为 'r')
> rorse -> rose (删除 'r')
> rose -> ros (删除 'e')
> 示例 2：
>
> 输入：word1 = "intention", word2 = "execution"
> 输出：5
> 解释：
> intention -> inention (删除 't')
> inention -> enention (将 'i' 替换为 'e')

> enention –> exention (将 'n' 替换为 'x')
> exention –> exection (将 'n' 替换为 'c')
> exection –> execution (插入 'u')
>
> 提示：
>
> 0 <= word1.length, word2.length <= 500
> word1 和 word2 由小写英文字母组成
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/edit-distance
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

动态规划：

dp[i][j] 代表 word1 到 i 位置转换成 word2 到 j 位置需要最少步数

所以，

当 word1[i] == word2[j]，dp[i][j] = dp[i-1][j-1];

当 word1[i] != word2[j]，dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1

其中，dp[i-1][j-1] 表示替换操作，dp[i-1][j] 表示删除操作，dp[i][j-1] 表示插入操作。

注意，针对第一行，第一列要单独考虑，我们引入 '' 下图所示：

	' '	r	o	s
' '	0	1	2	3
h	1			
o	2			
r	3			
s	4			
e	5			

第一行，是 word1 为空变成 word2 最少步数，就是插入操作

第一列，是 word2 为空，需要的最少步数，就是删除操作

再附上自顶向下的方法，大家可以提供 Java 版吗？

代码：
自底向上

```
class Solution {
    public int minDistance(String word1, String word2) {
        int n1 = word1.length();
        int n2 = word2.length();
        int[][] dp = new int[n1 + 1][n2 + 1];
        // 第一行
        for (int j = 1; j <= n2; j++) dp[0][j] = dp[0][j - 1] + 1;
        // 第一列
        for (int i = 1; i <= n1; i++) dp[i][0] = dp[i - 1][0] + 1;
        for (int i = 1; i <= n1; i++) {
            for (int j = 1; j <= n2; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) dp[i][j] = dp[i - 1][j - 1];
                else dp[i][j] = Math.min(Math.min(dp[i - 1][j - 1], dp[i][j - 1]), dp[i - 1][j]) + 1;
            }
        }
        return dp[n1][n2];
    }
}
作者：powcai
链接：https://leetcode-cn.com/problems/edit-distance/solution/zi-di-xiang-shang-he-zi-ding-xiang-xia-by-powcai-3/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

2 | C++ 工程师

2.1 | 数组

- **请你回答一下Array&List， 数组和链表的区别**

参考回答：
数组的特点：
数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。数组的插入数据和删除数据效率低，插入数据时，这个位置后面的数据在内存中都要向后移。删除数据时，这个数据后面的数据都要往前移动。但数组的随机读取效率很高。因为数组是连续的，知道每一个数据的内存地址，可以直接找到给地址的数据。如果应用需要快速访问数据，很少或不插入和删除元素，就应该用数组。数组需要预留空间，在使用前要先申

请占内存的大小，可能会浪费内存空间。并且数组不利于扩展，数组定义的空间不够时要重新定义数组。

链表的特点：

链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。比如：上一个元素有个指针指到下一个元素，以此类推，直到最后一个元素。如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表数据结构了。不指定大小，扩展方便。链表大小不用定义，数据随意增删。

各自的优缺点

数组的优点：

1. 随机访问性强

2. 查找速度快

数组的缺点:

1. 插入和删除效率低

2. 可能浪费内存

3. 内存空间要求高，必须有足够的连续内存空间。

4. 数组大小固定，不能动态拓展

链表的优点:

1. 插入删除速度快

2. 内存利用率高，不会浪费内存

3. 大小没有固定，拓展很灵活。

链表的缺点:

不能随机查找，必须从第一个开始遍历，查找效率低

- **一组有序数（从小到大排列），有负有正，找出绝对值最小值**

参考回答：

```
int i = 0;
while( a[i] < 0){
    i++;
}
if(a[i] + a[i-1] <= 0)
```

```
    return a[i];
else
    return a[i-1];
```

链接：
https://www.nowcoder.com/questionTerminal/bde7945561b546baa3b79b8d5850422b
来源：牛客

- **数组中重复的数字**

在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中第一个重复的数字。 例如，如果输入长度为7的数组[2,3,1,0,2,5,3]，那么对应的输出是第一个重复的数字2。没有重复的数字返回-1。

示例1
输入

```
[2,3,1,0,2,5,3]
```

输出

```
2
```

参考回答：
直接使用map，遍历一遍数组，当第一次出现重复的值时，此时map值大于1，直接返回得到答案

```
class Solution {
public:
    /**
     * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
     *
     *
     * @param numbers int整型vector
     * @return int整型
     */
    int duplicate(vector<int>& numbers) {
        if(numbers.size()<=1) return -1;
        // write code here
        unordered_map<int,int> map;
        for(int i = 0;i<numbers.size();i++){
            map[numbers[i]]++;
            if(map[numbers[i]]>1) return numbers[i];

        }
        return -1;
    }
};
```

链接：https://www.nowcoder.com/questionTerminal/6fe361ede7e54db1b84adc81d09d8524
来源：牛客网

- **一个长度为N的整形数组，数组中每个元素的取值范围是[0,n-1],判断该数组否有重复的数，请说一下你的思路并手写代码**

参考回答：

把每个数放到自己对应序号的位置上，如果其他位置上有和自己对应序号相同的数，那么即为有重复的数值。时间复杂度为O(N),同时为了节省空间复杂度，可以在原数组上进行操作，空间复杂度为O(1)

```
bool IsDuplicateNumber(int *array, int n)
{
if(array==NULL) return false;
int i,temp;
for(i=0;i<n;i++)
{
while(array[i]!=i)
{
if(array[array[i]]==array[i])
return true;
temp=array[array[i]];
array[array[i]]=array[i];
array[i]=temp;
}
}
return false;
}
```

2.2 | 排序

- **手写一下快排的代码**

参考回答：

```
int once_quick_sort(vector<int> &data, int left, int right)
{
int key = data[left];
while (left < right)
{
while (left < right && key <= data[right])
{
right--;
}
if (left < right)
{
data[left++] = data[right];
}
while (left < right && key > data[left])
{
left++;
```

```
}
if (left < right)
{
data[right--] = data[left];
}
}
data[left] = key;
return left;
}
int quick_sort(vector<int> & data, int left, int right)
{
if (left >= right )
{
return 1;
}
int middle = 0;
middle = once_quick_sort(data, left, right);
quick_sort(data, left, middle-1);
quick_sort(data, middle + 1, right);
};
```

- **介绍一下各种排序算法及其复杂度**

参考回答：

插入排序：对于一个带排序数组来说，其初始有序数组元素个数为1，然后从第二个元素，插入到有序数组中。对于每一次插入操作，从后往前遍历当前有序数组，如果当前元素大于要插入的元素，则后移一位；如果当前元素小于或等于要插入的元素，则将要插入的元素插入到当前元素的下一位中。

希尔排序：先将整个待排序记录分割成若干子序列，然后分别进行直接插入排序，待整个序列中的记录基本有序时，在对全体记录进行一次直接插入排序。其子序列的构成不是简单的逐段分割，而是将每隔某个增量的记录组成一个子序列。希尔排序时间复杂度与增量序列的选取有关，其最后一个值必须为1.

归并排序：该算法采用分治法；对于包含m个元素的待排序序列，将其看成m个长度为1的子序列。然后两两合归并，得到n/2个长度为2或者1的有序子序列；然后再两两归并，直到得到1个长度为m的有序序列。

冒泡排序：对于包含n个元素的带排序数组，重复遍历数组，首先比较第一个和第二个元素，若为逆序，则交换元素位置；然后比较第二个和第三个元素，重复上述过程。每次遍历会把当前前n-i个元素中的最大的元素移到n-i位置。遍历n次，完成排序。

快速排序：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

选择排序：每次循环，选择当前无序数组中最小的那个元素，然后将其与无序数组的第一个元素交换位置，从而使有序数组元素加1，无序数组元素减1.初始时无序数组为空。

堆排序：堆排序是一种选择排序，利用堆这种数据结构来完成选择。其算法思想是将带排序数据构造一个最大堆（升序）/最小堆（降序），然后将堆顶元素与待排序数组的最后一个元素交换位置，此时末尾元素就是最大/最小的值。然后将剩余n-1个元素重新构造成最大堆/最小堆。

各个排序的时间复杂度、空间复杂度及稳定性如下：

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性	复杂性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(nlog_2n)$	$O(n^2)$	$O(n)$	$O(1)$	不稳定	较复杂
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定	较复杂
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定	较复杂

- **稳定排序有哪几种?**

参考回答：

基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序

- **请问求第k大的数的方法以及各自的复杂度是怎样的，另外追问一下，当有相同元素时，还可以使用什么不同的方法求第k大的元素**

参考回答：

首先使用快速排序算法将数组按照从大到小排序，然后取第k个，其时间复杂度最快为O(nlogn)

使用堆排序，建立最大堆，然后调整堆，知道获得第k个元素，其时间复杂度为O(n+klogn)

首先利用哈希表统计数组中个元素出现的次数，然后利用计数排序的思想，线性从大到小扫描过程中，前面有k-1个数则为第k大的数

利用快排思想，从数组中随机选择一个数i，然后将数组分成两部分Dl,Dr，Dl的元素都小于i,Dr的元素都大于i。然后统计Dr元素个数，如果Dr元素个数等于k-1,那么第k大的数即为k，如果Dr元素个数小于k,那么继续求Dl中第k-Dr大的元素；如果Dr元素个数大于k,那么继续求Dr中第k大的元素。

当有相同元素的时候，

首先利用哈希表统计数组中个元素出现的次数，然后利用计数排序的思想，线性从大到小扫描过程中，前面有k-1个数则为第k大的数，平均情况下时间复杂度为O(n)

- **请问海量数据如何去取最大的k个**

参考回答：

1.直接全部排序（只适用于内存够的情况）

当数据量较小的情况下，内存中可以容纳所有数据。则最简单也是最容易想到的方法是将数据全部排序，然后取排序后的数据中的前K个。

这种方法对数据量比较敏感，当数据量较大的情况下，内存不能完全容纳全部数据，这种方法便不适应了。即使内存能够满足要求，该方法将全部数据都排序了，而题目只要求找出top K个数据，所以该方法并不十分高效，不建议使用。

2.快速排序的变形（只使用于内存够的情况）

这是一个基于快速排序的变形，因为第一种方法中说到将所有元素都排序并不十分高效，只需要找出前K个最大的就行。

这种方法类似于快速排序，首先选择一个划分元，将比这个划分元大的元素放到它的前面，比划分元小的元素放到它的后面，此时完成了一趟排序。如果此时这个划分元的序号index刚好等于K，那么这个划分元以及它左边的数，刚好就是前K个最大的元素；如果index ＞ K，那么前K大的数据在index的左边，那么就继续递归的从index-1个数中进行一趟排序；如果index ＜ K，那么再从划分元的右边继续进行排序，直到找到序号index刚好等于K为止。再将前K个数进行排序后，返回Top K个元素。这种方法就避免了对除了Top K个元素以外的数据进行排序所带来的不必要的开销。

3.最小堆法

这是一种局部淘汰法。先读取前K个数，建立一个最小堆。然后将剩余的所有数字依次与最小堆的堆顶进行比较，如果小于或等于堆顶数据，则继续比较下一个；否则，删除堆顶元素，并将新数据插入堆中，重新调整最小堆。当遍历完全部数据后，最小堆中的数据即为最大的K个数。

4.分治法

将全部数据分成N份，前提是每份的数据都可以读到内存中进行处理，找到每份数据中最大的K个数。此时剩下N*K个数据，如果内存不能容纳N*K个数据，则再继续分治处理，分成M

份，找出每份数据中最大的K个数，如果M*K个数仍然不能读到内存中，则继续分治处理。直到剩余的数可以读入内存中，那么可以对这些数使用快速排序的变形或者归并排序进行处理。

5.Hash法

如果这些数据中有很多重复的数据，可以先通过hash法，把重复的数去掉。这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间。处理后的数据如果能够读入内存，则可以直接排序；否则可以使用分治法或者最小堆法来处理数据。

- **请问快排的时间复杂度最差是多少？什么时候时间最差**

参考回答：

O（N2),元素本来倒序排列用时最多

- **请你介绍一下快排算法；以及什么是稳定性排序，快排是稳定性的吗；快排算法最差情况推导公式**

参考回答：

1、快排算法

根据哨兵元素，用两个指针指向待排序数组的首尾，首指针从前往后移动找到比哨兵元素大的，尾指针从后往前移动找到比哨兵元素小的，交换两个元素，直到两个指针相遇，这是一趟排序，经常这趟排序后，比哨兵元素大的在右边，小的在左边。经过多趟排序后，整个数组有序。

稳定性：不稳定

平均时间复杂度：O(nlogn)

2、稳定排序

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。

快排算法是不稳定的排序算法。例如：

待排序数组:int a[] ={1, 2, 2, 3, 4, 5, 6};

若选择a[2]（即数组中的第二个2）为枢轴，而把大于等于比较子的数均放置在大数数组中，则a[1]（即数组中的第一个2）会到pivot的右边， 那么数组中的两个2非原序。

若选择a[1]为比较子，而把小于等于比较子的数均放置在小数数组中，则数组中的两个2顺序也非原序。

3、快排最差情况推倒

在快速排序的早期版本中呢，最左面或者是最右面的那个元素被选为枢轴，那最坏的情况就会在下面的情况下发生啦：

1）数组已经是正序排过序的。（每次最右边的那个元素被选为枢轴）

2）数组已经是倒序排过序的。（每次最左边的那个元素被选为枢轴）

3）所有的元素都相同（1、2的特殊情况）

因为这些案例在用例中十分常见，所以这个问题可以通过要么选择一个随机的枢轴，或者选择一个分区中间的下标作为枢轴，或者（特别是对于相比更长的分区）选择分区的第一个、中间、最后一个元素的中值作为枢轴。有了这些修改，那快排的最差的情况就不那么容易出现了，但是如果输入的数组最大（或者最小元素）被选为枢轴，那最坏的情况就又来了。

快速排序，在最坏情况退化为冒泡排序，需要比较O(n2)次（n(n - 1)/2次）。

2.3 | 动态规划

- **手写代码：最长公共连续子序列**

参考回答：

```
int substr(string & str1, string &str2)
{
int len1 = str1.length();
int len2 = str2.length();
vector<vector<int>>dp(len1,vector<int>(len2,0));
for (int i = 0; i < len1; i++)
{
dp[i][0] = str1[i]==str1[0]?1:0;
}
for (int j = 0; j <= len2; j++)
{
dp[0][j] = str1[0]==str2[j]?1:0;
}
for (int i = 1; i < len1; i++)
{
for (int j = 1; j < len2; j++)
{
if (str1[i] == str2[j])
{
dp[i][j] = dp[i - 1][j - 1]+1;
}
}
```

```
}
int longest = 0;
int longest_index = 0;
for (int i = 0; i < len1; i++)
{
for (int j = 0; j < len2; j++)
{
if (longest < dp[i][j])
{
longest = dp[i][j];
longest_index = i;
}
}
}
//字符串为从第i个开始往前数longest个
for (int i = longest_index-longest+1; i <=longest_index; i++)
{
cout << str1[i] << endl;
}
return longest;
}
```

- **手写代码：求一个字符串最长回文子串**

参考回答：

```
int LongestPalindromicSubstring(string & a)
{
int len = a.length();
vector<vector<int>>dp(len, vector<int>(len, 0));
for (int i = 0; i < len; i++)
{
dp[i][i] = 1;
}
int max_len = 1;
int start_index = 0;
for (int i= len - 2; i >= 0; i--)
{
for (int j = i + 1; j < len; j++)
{
if (a[i] == a[j])
{
if (j - i == 1)
```

```
{
dp[i][j] = 2;
}
else
{
if (j - i > 1)
{
dp[i][j] = dp[i + 1][j - 1] + 2;
}
}
if (max_len < dp[i][j])
{
max_len =  dp[i][j];
start_index = i;
}
}
else
{
dp[i][j] = 0;
}
}
}
cout << "max len is " << max_len << endl;
cout << "star index is" << start_index << endl;
return max_len;
}

```

- **手写代码：求最大子序和**

> 给定一个整数数组 nums ， 找到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。
>
> 示例 1：
>
> 输入：nums = [−2,1,−3,4,−1,2,1,−5,4]
> 输出：6
> 解释：连续子数组 [4,−1,2,1] 的和最大，为 6 。
> 示例 2：
>
> 输入：nums = [1]
> 输出：1
> 示例 3：
>
> 输入：nums = [0]
> 输出：0

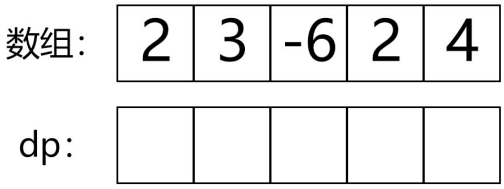
> 示例 4：
>
> 输入：nums = [-1]
> 输出：-1
> 示例 5：
>
> 输入：nums = [-100000]
> 输出：-100000
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/maximum-subarray
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

dp[i]表示nums中以nums[i]结尾的最大子序和
dp[i] = max(dp[i - 1] + nums[i], nums[i]);
dp[i]时当前数字，要么是与前面的最大字序和的和

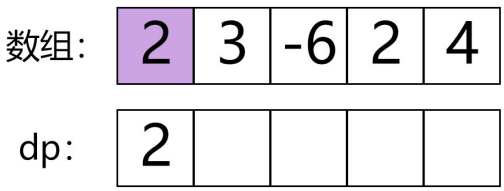
时间复杂度O(n)
空间复杂度O(n) 可优化至O(1)

动态规划法



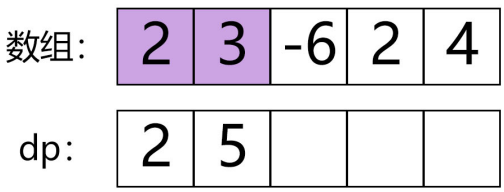
dp[i]表示
nums中以nums[i]结尾的最大子序和
dp[0] = nums[0]
dp[i] = max(dp[i - 1] + nums[i], nums[i]);

动态规划法



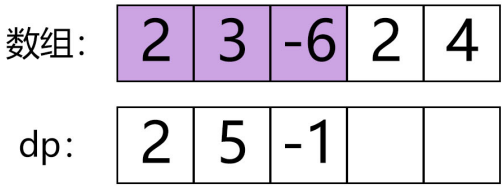
dp[i]表示
nums中以nums[i]结尾的最大子序和
dp[0] = nums[0]
dp[i] = max(dp[i - 1] + nums[i], nums[i]);

动态规划法



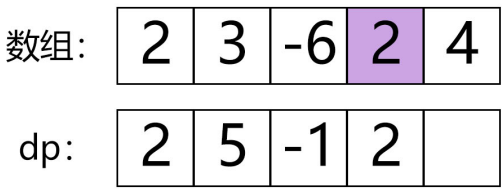
dp[i]表示
nums中以nums[i]结尾的最大子序和
dp[0] = nums[0]
dp[i] = max(dp[i - 1] + nums[i], nums[i]);

动态规划法



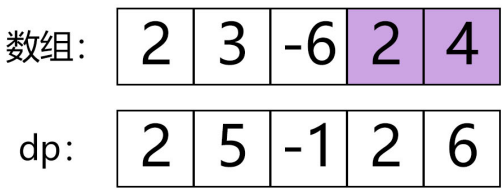
dp[i]表示
nums中以nums[i]结尾的最大子序和
dp[0] = nums[0]
dp[i] = max(dp[i - 1] + nums[i], nums[i]);

动态规划法



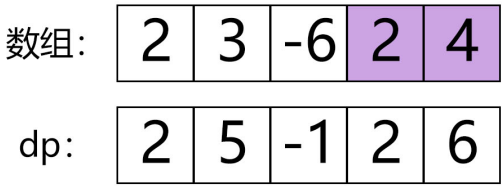
dp[i]表示
nums中以nums[i]结尾的最大子序和
dp[0] = nums[0]
dp[i] = max(dp[i - 1] + nums[i], nums[i]);

动态规划法



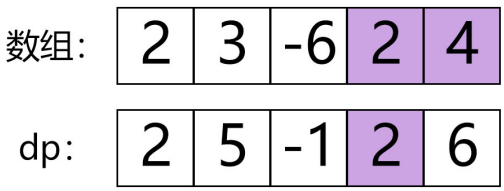
dp[i]表示
nums中以nums[i]结尾的最大子序和
dp[0] = nums[0]
dp[i] = max(dp[i - 1] + nums[i], nums[i]);

动态规划法



dp中最大值为6

动态规划法



因为我们只用到dp中的前一项
我们可以用int代替一维数组，时间复杂度为O(1)

```
class Solution
{
public:
    int maxSubArray(vector<int> &nums)
    {
        //类似寻找最大最小值的题目，初始值一定要定义成理论上的最小最大值
        int result = INT_MIN;
        int numsSize = int(nums.size());
        //dp[i]表示nums中以nums[i]结尾的最大子序和
        vector<int> dp(numsSize);
        dp[0] = nums[0];
        result = dp[0];
        for (int i = 1; i < numsSize; i++)
        {
            dp[i] = max(dp[i - 1] + nums[i], nums[i]);
            result = max(result, dp[i]);
        }
```

```
        return result;
    }
};
```

```
class Solution
{
public:
    int maxSubArray(vector<int> &nums)
    {
        //类似寻找最大最小值的题目，初始值一定要定义成理论上的最小最大值
        int result = INT_MIN;
        int numsSize = int(nums.size());
        //因为只需要知道dp的前一项，我们用int代替一维数组
        int dp(nums[0]);
        result = dp;
        for (int i = 1; i < numsSize; i++)
        {
            dp = max(dp + nums[i], nums[i]);
            result = max(result, dp);
        }
        return result;
    }
};
作者：pinku-2
链接：https://leetcode-cn.com/problems/maximum-subarray/solution/zui-da-zi-xu-he-cshi-xian-si-chong-jie-fa-bao-li-f/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

2.4 | 链表

- **请你手写代码，如何合并两个有序链表**

参考回答：

```
class Solution {
public:
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
if(l1 == NULL)
{
return l2;
}
```

```
if(l2 == NULL)
{
return l1;
}
if(l1->val < l2->val)
{
l1->next=mergeTwoLists(l1->next,l2);
return l1;
}
else
{
l2->next=mergeTwoLists(l1,l2->next);
return l2;
}
}
};
```

- **手写代码：反转链表**

参考回答：

```
Void reversal_list(mylist * a_list)
{
mylist * forward_node = nullptr;
mylist * cur_node = a_list->next;
mylist* next_node = cur_node->next;
if(cur_node == nullptr)
{
return ;
}
while(1)
{
cur_node->next = forward_node;
forward_node = cur_node;
cur_node = next_node;
if(cur_node == nullptr)
{
break;
}
next_node = cur_node->next;
}
a_list->next = forward_node;
}
```

- **判断一个链表是否为回文链表，说出你的思路并手写代码**

参考回答：

思路：使用栈存储链表前半部分，然后一个个出栈，与后半部分元素比较，如果链表长度未知，可以使用快慢指针的方法，将慢指针指向的元素入栈，然后如果快指针指向了链表尾部，此时慢指针指向了链表中间

```
bool is_palindromic_list2(mylist *a_list)
{
if(a_list == nullptr)
{
return false;
}
stack<int>list_value;
mylist * fast =a_list;
mylist *slow =a_list;
while(fast->next!=nullptr && fast->next->next!=nullptr)
{
list_value.push(slow->next->value);
slow = slow->next;
fast = fast->next->next;
}
cout<<"middle elem value is "<<slow->next->value<<endl;
if(fast->next != nullptr)
{
cout<<"the list has odd num of node"<<endl;
slow =slow->next;
}
int cur_value;
while(!list_value.empty())
{
cur_value = list_value.top();
cout<<"stack top value is"<<cur_value<<endl;
cout<<"list value is "<<slow->next->value<<endl;
if(cur_value != slow->next->value)
{
return false;
}
list_value.pop();
slow = slow->next;
}
return true;
```

```
}
```

- **请问什么是单向链表，如何判断两个单向链表是否相交**

参考回答：

考察点：数据结构，算法

公司：百度

1、单向链表

单向链表（单链表）是链表的一种，其特点是链表的链接方向是单向的，对链表的访问要通过顺序读取从头部开始；链表是使用指针进行构造的列表；又称为结点列表，因为链表是由一个个结点组装起来的；其中每个结点都有指针成员变量指向列表中的下一个结点。

列表是由结点构成，head指针指向第一个成为表头结点，而终止于最后一个指向nuLL的指针。

2、判断两个链表是否相交

1）方法1：

链表相交之后，后面的部分节点全部共用，可以用2个指针分别从这两个链表头部走到尾部，最后判断尾部指针的地址信息是否一样，若一样则代表链表相交！

2）方法2：

可以把其中一个链表的所有节点地址信息存到数组中，然后把另一个链表的每一个节点地址信息遍历数组，若相等，则跳出循环，说明链表相交。进一步优化则是进行hash排序，建立hash表。

2.5 | 高级算法

- **什么是LRU缓存**

参考回答：

LRU(最近最少使用)算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高

实现：使用一个链表保存缓存数据，将新数据插入到头部，每当缓存命中时，则将命中的数据移动到链表头部，当链表满的时候，将链表尾部的数据丢弃。

- **请你说一说洗牌算法**

参考回答：

1、Fisher-Yates Shuffle算法

最早提出这个洗牌方法的是 Ronald A. Fisher 和 Frank Yates，即 Fisher–Yates Shuffle，其基本思想就是从原始数组中随机取一个之前没取过的数字到新的数组中，具体如下：

1）初始化原始数组和新数组，原始数组长度为n(已知)。

2）从还没处理的数组（假如还剩k个）中，随机产生一个[0, k)之间的数字p（假设数组从0开始）。

3）从剩下的k个数中把第p个数取出。

4）重复步骤2和3直到数字全部取完。

5）从步骤3取出的数字序列便是一个打乱了的数列。

时间复杂度为O(n*n)，空间复杂度为O(n)。

2）Knuth-Durstenfeld Shuffle

Knuth 和 Durstenfeld 在Fisher 等人的基础上对算法进行了改进，在原始数组上对数字进行交互，省去了额外O(n)的空间。该算法的基本思想和 Fisher 类似，每次从未处理的数据中随机取出一个数字，然后把该数字放在数组的尾部，即数组尾部存放的是已经处理过的数字。

算法步骤为：

1. 建立一个数组大小为 n 的数组 arr，分别存放 1 到 n 的数值；

2. 生成一个从 0 到 n - 1 的随机数 x；

3. 输出 arr 下标为 x 的数值，即为第一个随机数；

4. 将 arr 的尾元素和下标为 x 的元素互换；

5. 同2，生成一个从 0 到 n - 2 的随机数 x；

6. 输出 arr 下标为 x 的数值，为第二个随机数；

7. 将 arr 的倒数第二个元素和下标为 x 的元素互换；

……

如上，直到输出m 个数为止

时间复杂度为O(n)，空间复杂度为O(1)，缺点必须知道数组长度n。

2.6 | 字符串

- **给你一个字符串，找出第一个不重复的字符，如“abbbabcd”，则第一个不重复就是c**

参考回答：

使用哈希的思想，建立256个bool数组array，初始都为false,从头开始扫描字符串，扫到一个，将以其ascii码为下标的元素置true。例如扫描到A的时候，执行：array['A']=true。第二边扫描，扫到一个字母就以其ascii码为下标，去array数组中看其值，如果是true,返回改字母，如果是false，继续扫描下一个字母。

- **最长公共前缀**

> 编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

示例 1：

输入：strs = ["flower","flow","flight"]
输出："fl"
示例 2：

输入：strs = ["dog","racecar","car"]
输出：""
解释：输入不存在公共前缀。

提示：

0 <= strs.length <= 200
0 <= strs[i].length <= 200
strs[i] 仅由小写英文字母组成

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/longest-common-prefix
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
解题思路
先找出数组中字典序最小和最大的字符串，最长公共前缀即为这两个字符串的公共前缀
下面是 C++17 的代码

```
class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        if(strs.empty()) return "";
        // c++17 结构化绑定
        // str0, str1 分别是一个 pair<string, string> 的 first 和 second
        const auto [str0, str1] = minmax_element(strs.begin(),
strs.end());
        for(int i = 0; i < str0->size(); ++i)
            if(str0->at(i) != str1->at(i)) return str0->substr(0, i);
        return *str0;
    }
};
```

等同的 C++11 代码如下

```
class Solution {
public:
```

```
    string longestCommonPrefix(vector<string>& strs) {
        if(strs.empty()) return "";
        const auto p = minmax_element(strs.begin(), strs.end());
        for(int i = 0; i < p.first->size(); ++i)
            if(p.first->at(i) != p.second->at(i)) return p.first->substr(0, i);
        return *p.first;
    }
};
```

作者：you-yuan-de-cang-qiong
链接：https://leetcode-cn.com/problems/longest-common-prefix/solution/zi-dian-xu-zui-da-he-zui-xiao-zi-fu-chuan-de-gong-/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **有效的字母异位词**

> 给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。
>
> 示例 1:
>
> 输入: s = "anagram", t = "nagaram"
> 输出: true
> 示例 2:
>
> 输入: s = "rat", t = "car"
> 输出: false
> 说明:
> 你可以假设字符串只包含小写字母。
>
> 进阶:
> 如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/valid-anagram
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

方法一：sort

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        sort(s.begin(), s.end());
        sort(t.begin(), t.end());
        return s == t;
    }
};
```

方法二：map1

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if(s.size() != t.size()) return false;
        unordered_map<char, int> map;
        for(char c : s) map[c] ++;
        for(char c : t)
            if(-- map[c] == -1) return false;
        return true;
    }
};
```

方法三：map2

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        unordered_map<char, int> map;
        for(int i = 0; i < max(s.size(), t.size()); i ++)
            map[s[i]] ++, map[t[i]] --;
        for(auto it : map)
            if(it.second != 0) return false;
        return true;
    }
};
```

方法四：数组1(推荐)

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if(s.size() != t.size()) return false;
        int hash[26] = {0};
        for(char c : s) hash[c - 'a'] ++;
        for(char c : t)
            if(-- hash[c - 'a'] == -1) return false;
        return true;
    }
};
```

方法五：数组2

```
class Solution {
public:
    bool isAnagram(string s, string t) {
```

```
        if(s.size() != t.size()) return false;
        int hash[26] = {0};
        for(int i = 0; i < s.size(); i ++)
            hash[s[i] - 'a'] ++, hash[t[i] - 'a'] --;
        for(int i = 0; i < 26; i ++)
            if(hash[i] != 0) return false;
        return true;
    }
};
```

作者：OrangeMan

链接：https://leetcode-cn.com/problems/valid-anagram/solution/cjian-ji-dai-ma-duo-chong-fang-fa-by-orangeman-10/

来源：力扣（LeetCode）

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

3 | Golang 工程师

3.1 | 递归&回溯

- **手写代码：两数相加**

> 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 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，数组中同一个元素不能使用两遍。
>
> 你可以按任意顺序返回答案。
>
> 示例 1：
>
> 输入：nums = [2,7,11,15], target = 9
> 输出：[0,1]
> 解释：因为 nums[0] + nums[1] == 9 ， 返回 [0, 1] 。
> 示例 2：
>
> 输入：nums = [3,2,4], target = 6
> 输出：[1,2]
> 示例 3：
>
> 输入：nums = [3,3], target = 6
> 输出：[0,1]
>
> 提示：
>
> 2 <= nums.length <= 103
> −109 <= nums[i] <= 109

> -109 <= target <= 109
> 只会存在一个有效答案
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/two-sum
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
思路
定义辅助函数addTwoNumber(l1 *ListNode, l2 *ListNode, add int)，add指进位值，向右进位。
l1、l2为空，进位值为0未递归边界条件。
计算add = l1.Val + l2.Val + add，add % 10为当前节点值，add / 10为下次递归进位值，相邻递归次数之间节点连接起来即为所求。
复杂度
时间复杂度：O(Max(M,N))
代码实现

```
func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    return addTwoNumber(l1, l2, 0)
}
func addTwoNumber(l1 *ListNode, l2 *ListNode, add int) *ListNode{
    if l1 == nil && l2 == nil && add == 0 {
        return nil;
    }
    if l1 != nil {
        add += l1.Val
        l1 = l1.Next
    }
    if l2 != nil {
        add += l2.Val
        l2 = l2.Next
    }
    node := ListNode{
        Val: add % 10,
        Next: addTwoNumber(l1, l2, add / 10),
    }
    return &node
}
```

作者：somnus-23
链接：https://leetcode-cn.com/problems/add-two-numbers/solution/di-gui-shi-xian-by-somnus-23/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **手写代码：括号生成**

> 数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

示例 1：

输入：n = 3
输出：[”((()))”,”(()())”,”(())()”,”()(())”,”()()()”]
示例 2：

输入：n = 1
输出：[”()”]

提示：

1 <= n <= 8

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/generate-parentheses
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
解题思路

使用递归解决问题

递归结束条件：当进入字符串的右括号多于已经进入的左括号就结束，当两个同时为零时，存储下结果
代码

```
func generateParenthesis(n int) []string {
    s := make([]string,0,0)
    t := " "
    brackets(n,n,&s,t[:0])
    return s
}
func brackets(l int , r int ,s *[]string ,t string){
    if l ==0 && r==0{
        *s = append(*s,t)
    }
    if l > r || l<0 || r<0{
        return
    }
    brackets(l-1,r,s,t+"(")
    brackets(l,r-1,s,t+")")
}
```

作者：xiao-xiao-xiao-niao-2
链接：https://leetcode-cn.com/problems/generate-parentheses/solution/di-gui-de-jie-jue-fang-an-by-xiao-xiao-x-vkhi/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **手写代码：验证二叉搜索树**

> 给定一个二叉树，判断其是否是一个有效的二叉搜索树。
>
> 假设一个二叉搜索树具有如下特征：
>
> 节点的左子树只包含小于当前节点的数。
> 节点的右子树只包含大于当前节点的数。
> 所有左子树和右子树自身必须也是二叉搜索树。
> 示例 1:
>
> 输入:
> 2
> / \
> 1 3
> 输出: true
> 示例 2:
>
> 输入:
> 5
> / \
> 1 4
> / \
> 3 6
> 输出: false
> 解释: 输入为: [5,1,4,null,null,3,6]。
> 根节点的值为 5 ，但是其右子节点值为 4 。
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/validate-binary-search-tree
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
思路
遇到树基本来说递归跑不掉了

简洁实现！！！

代码

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func isValidBST(root *TreeNode) bool {
    return validBST(root,math.MinInt64,math.MaxInt64)
}
```

```
func validBST(root *TreeNode,min,max int) bool{
    //递归结束条件
    if root==nil{
        return true
    }
    // 判断节点的值是不是在区间呢，不是的话就false结束
    if root.Val<=min || root.Val>=max{
        return false
    }
    //左递归 最大值改为当前节点值
    //右递归 最小值改为当前节点值
    return validBST(root.Left,min,root.Val) &&
validBST(root.Right,root.Val,max)
}
作者：feng-a
链接：https://leetcode-cn.com/problems/validate-binary-search-
tree/solution/98-yan-zheng-er-cha-sou-suo-shu-4ms-chao-ji-shao-d/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

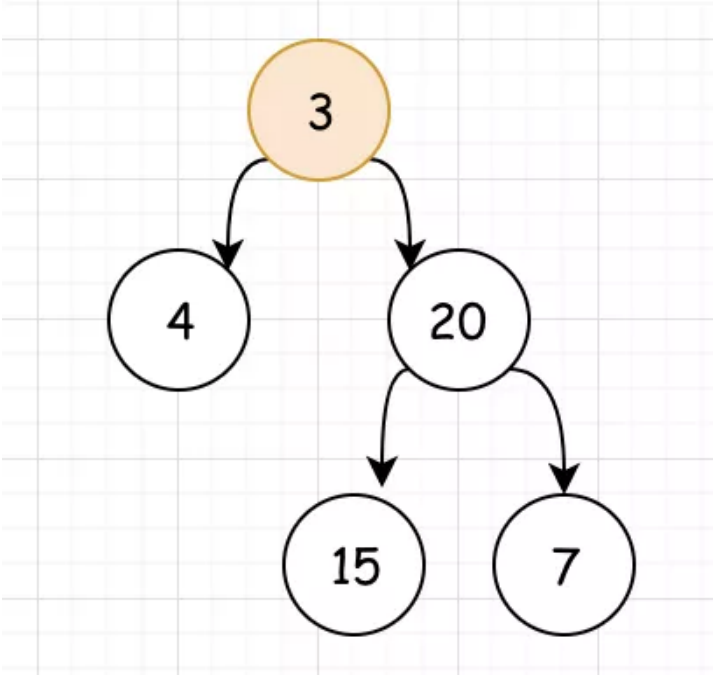
- **二叉树的最大深度**

参考回答：
递归求解
我们知道，每个节点的深度与它左右子树的深度有关，且等于其左右子树最大深度值加上 1。
即：

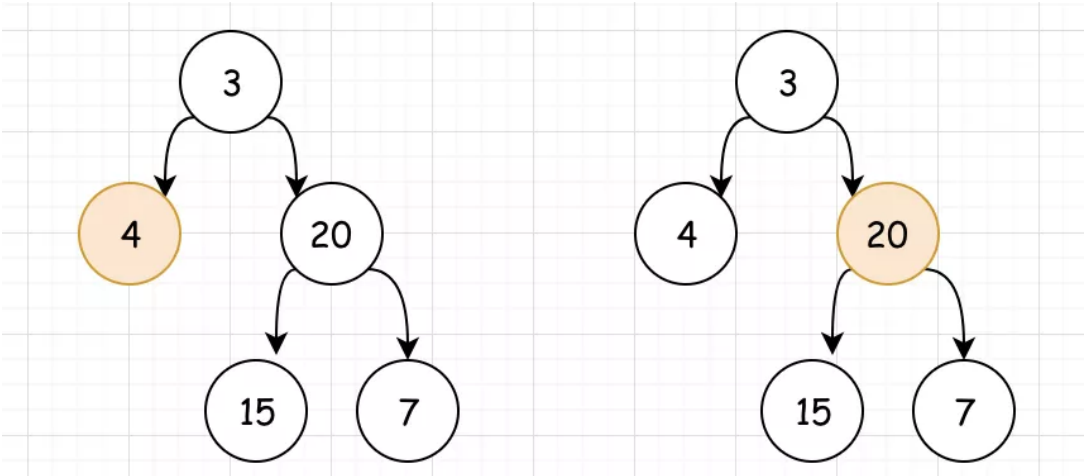
maxDepth(root) = max(maxDepth(root.left),
maxDepth(root.right)) + 1

以 [3,4,20,null,null,15,7] 为例：

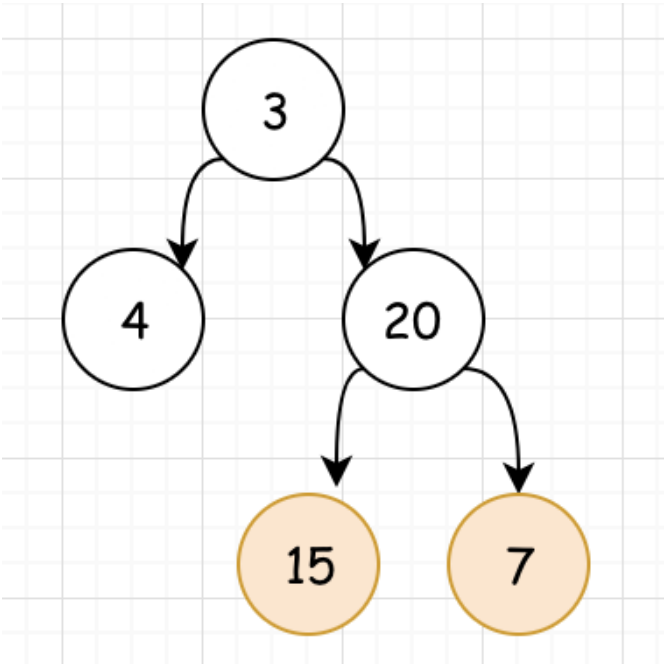
<1>我们要对根节点的最大深度求解，就要对其左右子树的深度进行求解



<2>我们看出。以4为根节点的子树没有左右节点，其深度为 1 。而以 20 为根节点的子树的深度，同样取决于它的左右子树深度。



<3>对于15和7的子树，我们可以一眼看出其深度为 1 。



<4>由此我们可以得到根节点的最大深度为:

```
maxDepth(root-3)
```

```
=max(**maxDepth**(sub-4),**maxDepth**(sub-20))+1
=max(1,max(**maxDepth**(sub-15),**maxDepth**(sub-7))+1)+1
=max(1,max(1,1)+1)+1
=max(1,2)+1
=3
```

根据分析，我们通过**递归**进行求解代码如下：

```
func maxDepth(root *TreeNode) int {
    if root == nil {
        return 0
    }
    return max(maxDepth(root.Left), maxDepth(root.Right)) + 1
}
func max(a int, b int) int {
    if a > b {
        return a
    }
    return b
}
```

作者：ivan1
链接：https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/solution/duo-jie-fa-qiu-zui-da-shen-du-di-gui-di-gui-dfs-fe/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **二叉树的最近公共祖先**

> 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。
>
> 百度百科中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”
>
> 示例 1：
>
> 输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
> 输出：3
> 解释：节点 5 和节点 1 的最近公共祖先是节点 3 。
> 示例 2：
>
> 输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
> 输出：5
> 解释：节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定义最近公共祖先节点可以为节点本身。
> 示例 3：
>
> 输入：root = [1,2], p = 1, q = 2

> 输出：1
>
> 提示：
>
> 树中节点数目在范围 [2, 105] 内。
> -109 <= Node.val <= 109
> 所有 Node.val 互不相同 。
> p != q
> p 和 q 均存在于给定的二叉树中。
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
思路
我们想一下，对于根节点 root，p、q 的分布，有两种可能：

p、q 分居 root 的左右子树，则 LCA 为 root。
p、q 在 root 的同一个子树中，问题转为规模小一点的相同问题。
定义递归函数
递归函数：返回当前子树中 p 和 q 的 LCA。如果没有 LCA，就返回 null。

从根root开始递归搜索，递归搜到各个子树……

如果遍历到 p 或 q，比方说 p，则 LCA 要么是当前的 p（q 在 p 的子树中），要么是 p 之上的节点（q 不在 p 的子树中），不可能是 p 之下的节点。不用继续往下遍历，返回当前的 p。

当遍历到 null 节点，空树不存在 p 和 q，不存在 LCA，返回 null。

当遍历的节点 root 不是 p 或 q 或 null，则递归搜寻 root 的左右子树：

如果左右子树的递归都有结果，说明 p 和 q 分居 root 的左右子树，返回 root。
如果只是一个子树递归调用有结果，说明 p 和 q 都在这个子树，返回该子树递归结果。
如果两个子树递归结果都为 null，说明 p 和 q 都不在这俩子树中，返回 null。
代码

```
func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
    if root == nil {
        return nil
    }
    if root == q || root == p {
        return root
    }
    left := lowestCommonAncestor(root.Left, p, q)
    right := lowestCommonAncestor(root.Right, p, q)
    if left != nil && right != nil {
        return root
```

```
    }
    if left == nil {
        return right
    }
    return left
}
```

作者：xiao_ben_zhu
链接：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/jsersi-lu-hao-li-jie-by-hyj8/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

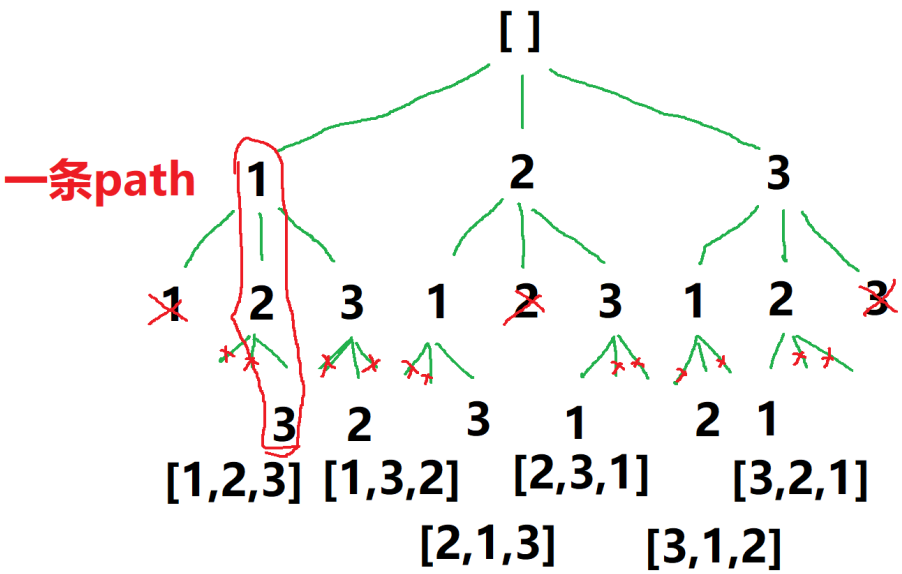
- **全排列**

参考回答：
思路
每一位都有3种选择：1、2、3。

每一次都做选择，展开出一棵空间树，如下图。

利用约束条件「不能重复选」，做剪枝，剪去不会产生正确解的选项（分支）。

用一个 hashMap，记录选过的数，下次遇到相同的数，跳过。
这样就不会进入「不会得出解的分支」，做无效的搜索。



怎么写递归函数
我们要在这个包含解的空间树上，用 DFS（递归）的方式搜索出所有的解。

dfs 函数做的事：基于当前的 path，继续选数，直到构建出合法的path，加入解集。

递归的入口：dfs 执行传入空 path，什么都还没选。

函数体内，用 for 循环，枚举出当前所有的选项，并写 if 语句跳过剪枝项。

每一轮迭代，作出一个选择，基于它，继续选（递归调用）。
递归的出口：当构建的 path 数组长度等于 nums 长度，就选够了，加入解集。

为什么要回溯
我们不是找到一个排列就完事，要找出所有满足条件的排列。

递归结束时，结束的是当前的递归分支，还要去别的分支继续搜。

所以，要撤销当前的选择，回到选择前的状态，去选下一个选项，即切入下一个分支。

注意，往map添加的当前选择也要同时撤销。代表没有做这个选择。

退回来，把路走全，才能在一棵空间树中，回溯出所有的解。

代码

```
func permute(nums []int) [][]int {
    res := [][]int{}
    visited := map[int]bool{}
    var dfs func(path []int)
    dfs = func(path []int) {
        if len(path) == len(nums) {
            temp := make([]int, len(path))
            copy(temp, path)
            res = append(res, temp)
            return
        }
        for _, n := range nums {
            if visited[n] {
                continue
            }
            path = append(path, n)
            visited[n] = true
            dfs(path)
            path = path[:len(path)-1]
            visited[n] = false
        }
    }
    dfs([]int{})
    return res
}
```

作者：xiao_ben_zhu
链接：https://leetcode-cn.com/problems/permutations/solution/chou-xiang-cheng-jue-ce-shu-yi-ge-pai-lie-jiu-xian/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

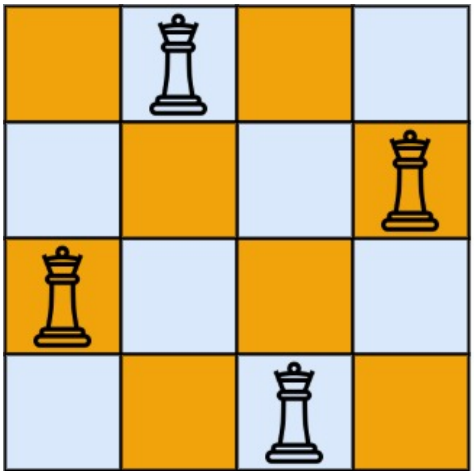
- N皇后

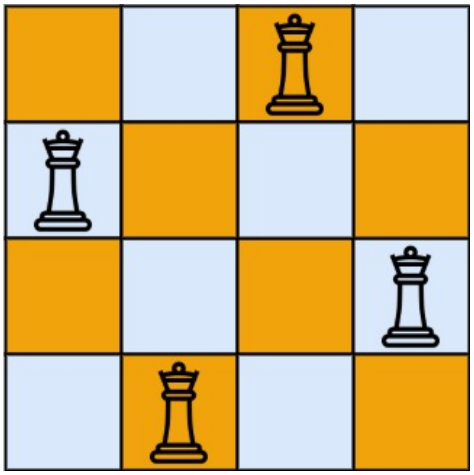
n 皇后问题 研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n ，返回所有不同的 n 皇后问题 的解决方案。

每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

示例 1：





输入：n = 4
输出：[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
解释：如上图所示，4 皇后问题存在两个不同的解法。
示例 2：

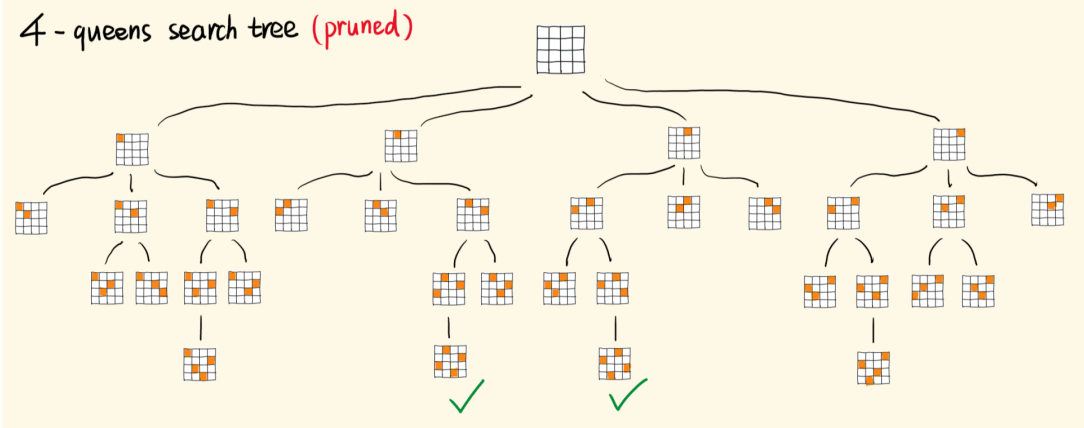
输入：n = 1
输出：[["Q"]]

提示：

1 <= n <= 9
皇后彼此不能相互攻击，也就是说：任何两个皇后都不能处于同一条横行、纵行或斜线上。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/n-queens
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
我起初的思路
以 4 皇后为例，我画出一个搜索树，初始时棋盘的格子都是"."。



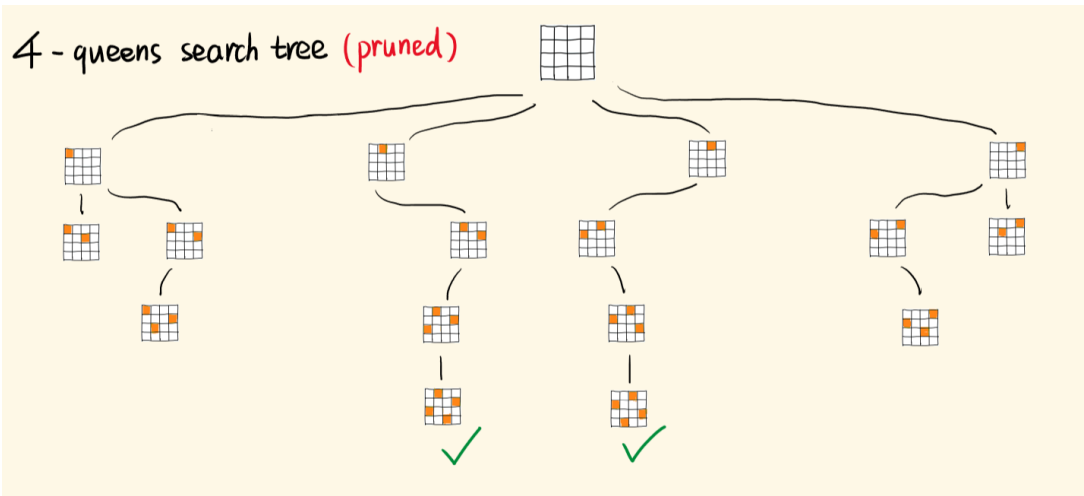
每一行，选一个格子置为”Q”，一行行往下选，第一行有四种选择。
在选下一行的皇后时，为了避免列的冲突，有三种选择。
继续选下去，可能会遇到对角线冲突，继续选下去没有意义，得不出合法的解。需要回溯。
回溯的套路（可硬记）：

遍历枚举出所有可能的选择。
依次尝试这些选择：作出一种选择，并往下递归。
如果这个选择产生不出正确的解，要撤销这个选择（将当前的 ”Q” 恢复为 ”.”），回到之前的状态，并作出下一个可用的选择。
选择、探索、撤销选择。识别出死胡同，就回溯，尝试下一个点，不做无效的搜索。

思路修正
我在枚举选择时，规避了行和列的冲突，前面画的搜索树已经是修剪后的。

但我还是想复杂了，对角线冲突和行列冲突一起作为约束就好，直接进行充分剪枝。

即遍历之前的行，如果当前的格子和之前的皇后们 同列或同对角线，则跳过该点（这需要优化，后面会讲）



- 你看上图左边两个叶子节点，下一行怎么放都冲突，可选的选项都被剪完了，
- 当所有可选的选择迭代完，当前递归分支就结束，撤销最后的选择，回到上一层，切入另一个分支。
- 当填完第四行，如上图的绿钩，生成了一个解，加入解集，并返回（这里不返回也行，因为已经做了充分的剪枝，不返回就会走一遍迭代，递归也结束），开始回溯，继续寻找完整解。

回溯的三要点

1. 选择，决定了搜索空间，决定了搜索空间有哪些节点。
2. 约束，用来剪枝，避免进入无效的分支。
3. 目标，决定了什么时候捕获有效的解，提前结束递归，开始回溯。

代码

Runtime: 4 ms, faster than 92.71% of Go online submissions for N-Queens.

```
func solveNQueens(n int) [][]string {
    bd := make([][]string, n)
    for i := range bd {
        bd[i] = make([]string, n)
        for j := range bd[i] {
            bd[i][j] = "."
        }
    }
    res := [][]string{}
    helper(0, bd, &res, n)
    return res
}
func helper(r int, bd [][]string, res *[][]string, n int) {
    if r == n {
        temp := make([]string, len(bd))
        for i := 0; i < n; i++ {
            temp[i] = strings.Join(bd[i], "")
        }
        *res = append(*res, temp)
        return
    }
    for c := 0; c < n; c++ {
        if isValid(r, c, n, bd) {
            bd[r][c] = "Q"
            helper(r+1, bd, res, n)
            bd[r][c] = "."
        }
    }
}
func isValid(r, c int, n int, bd [][]string) bool {
    for i := 0; i < r; i++ {
        for j := 0; j < n; j++ {
            if bd[i][j] == "Q" && (j == c || i+j == r+c || i-j == r-c) {
                return false
            }
        }
    }
    return true
}
```

还可以优化

本题必须记录之前放置皇后的位置，才能结合约束条件去做剪枝。

我每次都调用 isValid 遍历一遍前面的格子，效率是不优的。

最好是用三个数组或 Set 去记录出现过皇后的列们、正对角线们、反对角线们，用空间换取时间。

优化后的代码
Runtime: 4 ms, faster than 92.71% of Go online submissions for N-Queens.

```
func solveNQueens(n int) [][]string {
    bd := make([][]string, n)
    for i := range bd {
        bd[i] = make([]string, n)
        for j := range bd[i] {
            bd[i][j] = "."
        }
    }
    cols := map[int]bool{}
    diag1 := map[int]bool{}
    diag2 := map[int]bool{}
    res := [][]string{}
    helper(0, bd, &res, n, cols, diag1, diag2)
    return res
}
func helper(r int, bd [][]string, res *[][]string, n int, cols, diag1, diag2 map[int]bool) {
    if r == n {
        temp := make([]string, len(bd))
        for i := 0; i < n; i++ {
            temp[i] = strings.Join(bd[i], "")
        }
        *res = append(*res, temp)
        return
    }
    for c := 0; c < n; c++ {
        if !cols[c] && !diag1[r+c] && !diag2[r-c] {
            bd[r][c] = "Q"
            cols[c] = true
            diag1[r+c] = true
            diag2[r-c] = true
            helper(r+1, bd, res, n, cols, diag1, diag2)
            bd[r][c] = "."
            cols[c] = false
            diag1[r+c] = false
            diag2[r-c] = false
        }
    }
}
作者：xiao_ben_zhu
链接：https://leetcode-cn.com/problems/n-queens/solution/shou-hua-tu-jie-cong-jing-dian-de-nhuang-hou-wen-t/
来源：力扣（LeetCode）
```

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

3.2 | 并查集

- **手写代码：省份数量**

> 有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直接相连，且城市 b 与城市 c 直接相连，那么城市 a 与城市 c 间接相连。
>
> 省份 是一组直接或间接相连的城市，组内不含其他没有相连的城市。
>
> 给你一个 n x n 的矩阵 isConnected ，其中 isConnected[i][j] = 1 表示第 i 个城市和第 j 个城市直接相连，而 isConnected[i][j] = 0 表示二者不直接相连。
>
> 返回矩阵中 省份 的数量。
>
> 示例 1：
>
> 输入：isConnected = [[1,1,0],[1,1,0],[0,0,1]]
> 输出：2
> 示例 2：
>
> 输入：isConnected = [[1,0,0],[0,1,0],[0,0,1]]
> 输出：3
>
> 提示：
>
> 1 <= n <= 200
> n == isConnected.length
> n == isConnected[i].length
> isConnected[i][j] 为 1 或 0
> isConnected[i][i] == 1
> isConnected[i][j] == isConnected[j][i]
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/number-of-provinces
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

```go
func findCircleNum(isConnected [][]int) int {
    n := len(isConnected)
    if n == 1 {
        return 1
    }
```

```
    uf := generate(n)
    for i := 0; i < n; i++ {
        for j := i; j < n; j++ {
            if isConnected[i][j] == 1 {
                uf.union(i, j)
            }
        }
    }
    return uf.count
}
type unionFind struct {
    count int
    parent []int
}
func generate(n int) (uf *unionFind) {
    uf = &unionFind{}
    uf.count = n
    uf.parent = make([]int, n)
    for i := 0; i < n; i++ {
        uf.parent[i] = i
    }
    return
}
func (uf *unionFind) union(p, q int) {
    if p == q {
        return
    }
    rootP := uf.find(p)
    rootQ := uf.find(q)
    if rootP == rootQ {
        return
    }
    uf.parent[rootP] = rootQ
    uf.count--
}
func (uf *unionFind) find(t int) (root int) {
    root = t
    for root != uf.parent[root] {
        root = uf.parent[root]
    }
    for t != uf.parent[t] {
        x := uf.parent[t]
        uf.parent[t] = root
        t = x
    }
    return
}
作者：lzhlyle
```

链接：https://leetcode-cn.com/problems/number-of-provinces/solution/bing-cha-ji-by-lzhlyle-zh6l/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **手写代码：岛屿数量**

> 给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。
>
> 岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。
>
> 此外，你可以假设该网格的四条边均被水包围。
>
> 示例 1：
>
> 输入：grid = [
> ["1","1","1","1","0"],
> ["1","1","0","1","0"],
> ["1","1","0","0","0"],
> ["0","0","0","0","0"]
>]
> 输出：1
> 示例 2：
>
> 输入：grid = [
> ["1","1","0","0","0"],
> ["1","1","0","0","0"],
> ["0","0","1","0","0"],
> ["0","0","0","1","1"]
>]
> 输出：3
>
> 提示：
>
> m == grid.length
> n == grid[i].length
> 1 <= m, n <= 300
> grid[i][j] 的值为 '0' 或 '1'
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/number-of-islands
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

```
type UnionFindSet struct {
    Parents []int // 每个结点的顶级节点
```

```
    SetCount int // 连通分量的个数
}
func (u *UnionFindSet) Init(grid [][]byte) {
    row := len(grid)
    col := len(grid[0])
    count := row*col
    u.Parents = make([]int, count)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            u.Parents[i*col+j] = i*col+j
            if grid[i][j] == '1' {
                u.SetCount++
            }
        }
    }
}
func (u *UnionFindSet) Find(node int) int {
    if u.Parents[node] == node {
        return node
    }
    root := u.Find(u.Parents[node])
    u.Parents[node] = root
    return root
}
func (u *UnionFindSet) Union(node1 int, node2 int) {
    root1 := u.Find(node1)
    root2 := u.Find(node2)
    if root1 == root2 {
        return
    }
    if root1 < root2 {
        u.Parents[root1] = root2
    } else {
        u.Parents[root2] = root1
    }
    u.SetCount--
}
// 心得：并查集是一种搜索算法（针对聚合的）
func numIslands(grid [][]byte) int {
    // 创建并初始化并查集
    u := &UnionFindSet{}
    row := len(grid)
    col := len(grid[0])
    u.Init(grid)
    // 根据grid建立相应的并查集，并统计连通分量个数【每连接一次进行减一】
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            if grid[i][j] == '1' {
                // 如果周边四个方向也是1就进行union
```

```
                if i - 1 >= 0 && grid[i-1][j] == '1' {
                    u.Union(i*col+j, (i-1)*col+j)
                }
                if i + 1 < row && grid[i+1][j] == '1' {
                    u.Union(i*col+j, (i+1)*col+j)
                }
                if j - 1 >= 0 && grid[i][j-1] == '1' {
                    u.Union(i*col+j, i*col+(j-1))
                }
                if j + 1 < col && grid[i][j+1] == '1' {
                    u.Union(i*col+j, i*col+(j+1))
                }
                grid[i][j] = '0'
            }
        }
    }
    // 返回结果
    return u.SetCount
}
作者：bryson-2
链接：https://leetcode-cn.com/problems/number-of-islands/solution/bing-cha-ji-go-by-bryson-2/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

- **手写代码：最长连续序列**

> 给定一个未排序的整数数组 nums ， 找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。
>
> 进阶：你可以设计并实现时间复杂度为 O(n) 的解决方案吗?
>
> 示例 1：
>
> 输入：nums = [100,4,200,1,3,2]
> 输出：4
> 解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。
> 示例 2：
>
> 输入：nums = [0,3,7,2,5,8,4,6,0,1]
> 输出：9
>
> 提示：
>
> 0 <= nums.length <= 104
> −109 <= nums[i] <= 109
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/longest-consecutive-sequence

> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

1. 用map记录一下数字和索引
2. 用例有重复数字，需要处理一下
3. 看看左右数字是否在map中，有的话，连通它们的索引
4. 遍历并查集的size数组，看看规模最大的一个分量有多大，就是最长连续序列

```
func longestConsecutive(nums []int) int {
    //key : nums[i]; value : i
    uf := NewUnionFindSet(len(nums))
    m := make(map[int]int)
    for i := 0; i < len(nums); i++ {
        cur := nums[i]
        if _, ok := m[cur]; ok {
            continue
        }
        m[cur] = i
        if idx, ok := m[cur - 1]; ok {
            uf.Mix(i, idx)
        }
        if idx, ok := m[cur + 1]; ok {
            uf.Mix(i, idx)
        }
    }
    res := 0
    for _, v := range uf.sz {
        if v > res {
            res = v
        }
    }
    return res
}
type UnionFindSet struct {
   id    []int
   sz    []int
   count int
}
func NewUnionFindSet(n int) *UnionFindSet {
   id := make([]int, n)
   for i := range id {
      id[i] = i
   }
   sz := make([]int, n)
   for i := range sz {
      sz[i] = 1
   }
   return &UnionFindSet{
      id:    id,
      sz :   sz,
```

```
        count: n,
    }
}
func (u *UnionFindSet) Find(p int) int {
    for p != u.id[p] {
        p = u.id[p]
    }
    return p
}
func (u *UnionFindSet) Mix(p, q int) {
    i := u.Find(p)
    j := u.Find(q)
    if i == j {
        return
    }
    if u.sz[i] < u.sz[j] {
        u.id[i] = j
        u.sz[j] += u.sz[i]
    } else {
        u.id[j] = i
        u.sz[i] += u.sz[j]
    }
    u.count--
}
```

作者：rclove2k
链接：https://leetcode-cn.com/problems/longest-consecutive-sequence/solution/bing-cha-ji-lian-tong-lian-xu-shu-zi-de-n9t87/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

3.3 | 字符串

- **手写代码：转换成小写字母**

> 实现函数 ToLowerCase()，该函数接收一个字符串参数 str，并将该字符串中的大写字母转换成小写字母，之后返回新的字符串。
>
> 示例 1：
>
> 输入: "Hello"
> 输出: "hello"
> 示例 2：
>
> 输入: "here"
> 输出: "here"
> 示例 3：
>
> 输入: "LOVELY"
> 输出: "lovely"

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/to-lower-case
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

解题思路

Go里面无法对字符串修改的，需要转成字节数组

```
func toLowerCase(str string) string {
    b := []byte(str)
    for i:=0;i<len(b);i++{
        if str[i]>='A'&&str[i]<='Z'{
            b[i]=str[i]+32
        }
    }
    return string(b)
}
作者：ba-xiang
链接：https://leetcode-cn.com/problems/to-lower-case/solution/go-zhuan-zi-jie-shu-zu-ran-hou-bian-li-by-ba-xiang/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

- **手写代码：最长公共前缀**

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

示例 1：

输入：strs = ["flower","flow","flight"]
输出："fl"
示例 2：

输入：strs = ["dog","racecar","car"]
输出：""
解释：输入不存在公共前缀。

提示：

0 <= strs.length <= 200
0 <= strs[i].length <= 200
strs[i] 仅由小写英文字母组成

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/longest-common-prefix
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：

解题思路
先找到最短的元素，因为最长公共前缀的长度肯定小于等于这个元素的长度
接着遍历最短的元素，和数组中的每个元素的相同位置比较看是否相同
如果遇到不同的了，那就返回不同的之前的所有元素
如果遍历完了这个最短元素，说明这个最短元素就是最长公共前缀，直接返回

代码

```
func longestCommonPrefix(strs []string) string {
    // 最长的公共前缀的长度肯定小于等于数组中最短的元素
    // 所以从这个元素开始当基准
    short := findShortestInArray(strs)
    // 早发现早治疗
    if len(short) == 0 {
        return ""
    }
    // 遍历这个最短的每个位置元素，用来判断是不是相等
    for i, v := range short {
        // 要判断多少次，取决于数组strs中有多少个元素，所以用的len(strs)
        for j := 0; j < len(strs); j++ {
            // 数组的第j个元素的第i个位置不等于我们的short的第i个位置的元素
            // 写成strs[j][i] 是为了和short里面的每个元素一一对应比较
            if strs[j][i] != byte(v) {
                // 到了第[j][i]个没有匹配上，那么就说明之前的都匹配上了，所以直接返回[j][:i]
                return strs[j][:i]
            }
        }
    }
    // 遍历完short了，说明short就是最长的，直接返回
    return short
}
func findShortestInArray(s []string) string {
    // 空字符数组返回空
    if len(s) == 0 {
        return ""
    }
    // 临时定义最短为数组第一个
    shortest := s[0]
    // 遍历数组每个元素
    for _, v := range s {
        // 找到当前小于res
        if len(v) < len(shortest) {
            // 看看是否是空的，空的说明数组中有空字符，所以最长公共前缀肯定为空
            if len(v) == 0 {
                return ""
            }
            // 替换当前最小为当前遍历到的元素
            shortest = v
```

```
        }
    }
    return shortest
}
```

作者：DCCooper
链接：https://leetcode-cn.com/problems/longest-common-prefix/solution/gojie-fa-xiang-xi-zhu-shi-kan-bu-dong-da-si-wo-by-/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

- **手写代码：有效的字母异位词**

> 给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。
>
> 示例 1:
>
> 输入: s = ”anagram”, t = ”nagaram”
> 输出: true
> 示例 2:
>
> 输入: s = ”rat”, t = ”car”
> 输出: false
> 说明:
> 你可以假设字符串只包含小写字母。
>
> 进阶:
> 如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？
>
> 来源：力扣（LeetCode）
> 链接：https://leetcode-cn.com/problems/valid-anagram
> 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

参考回答：
思路
用 HashMap 来映射 s 的字符和对应的出现次数。
然后遍历 t ，将对应的字符在 map 中的值减 1，如果所有 value 都变为 0，则返回 true。

最多只有 26 个小写字符，所以也可以用长度为 26 的整型数组。

代码
下面几版的代码都是用这个思路，大同小异

```
func isAnagram(s string, t string) bool {
    hash := make([]int, 26)
    for _, c := range s {
        hash[c-'a']++  // int32类型的Unicode数值相减
    }
    for _, c := range t {
        if hash[c-'a'] == 0 { // s中没有出现过该字符，返回false
            return false
```

```
        }
        hash[c-'a']--  // 出现过，对应的出现次数-1
    }
    for _, v := range hash { // 如果有一个元素不为0，那就不满足
        if v != 0 {
            return false
        }
    }
    return true
}
```

代码2

```
func isAnagram(s string, t string) bool {
    if len(s) != len(t) {
        return false
    }
    hash := make([]int, 26)
    for i := 0; i < len(s); i++ { // s  t  长度相等
        hash[s[i]-'a']++
        hash[t[i]-'a']--
    }
    for _, v := range hash {
        if v != 0 {
            return false
        }
    }
    return true
}
```

代码3

```
func isAnagram(s string, t string) bool {
    hash := map[rune]int{}
    for _, c := range s {
        hash[c]++
    }
    for _, c := range t {
        if count, ok := hash[c]; !ok || count == 0 {
            return false
        }
        hash[c]-- // 匹配掉一个，就减一个
    }
    for _, v := range hash {
        if v != 0 {
            return false
        }
    }
    return true
}
```

代码4

```
func isAnagram(s string, t string) bool {
    if len(s) != len(t) {
        return false
    }
    arr1 := [26]int{}
    arr2 := [26]int{}
    for i := 0; i < len(s); i++ {
        arr1[s[i]-'a']++
        arr2[t[i]-'a']++
    }
    return arr1 == arr2
}
```

作者：xiao_ben_zhu
链接：https://leetcode-cn.com/problems/valid-anagram/solution/242-you-xiao-de-zi-mu-yi-wei-ci-by-xiao_ben_zhu/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。