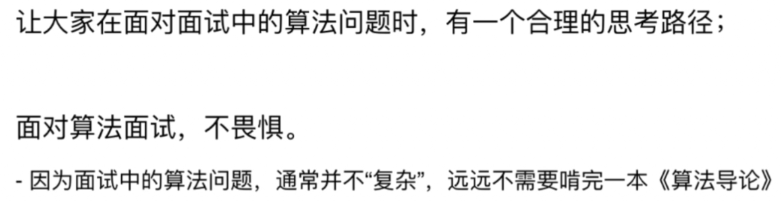
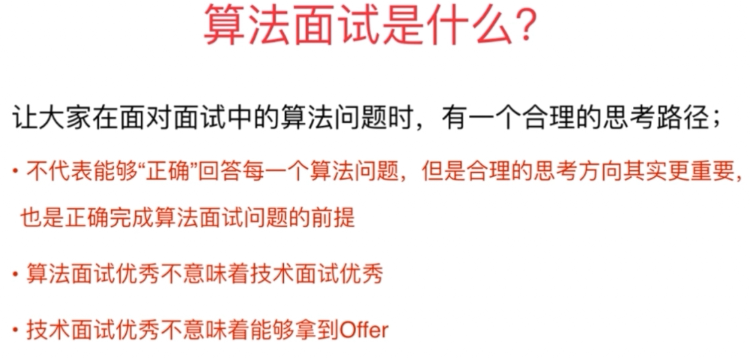
# 算法入门基础

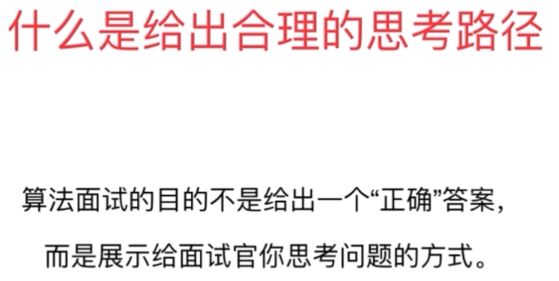
## 第1章 算法面试到底是什么鬼?

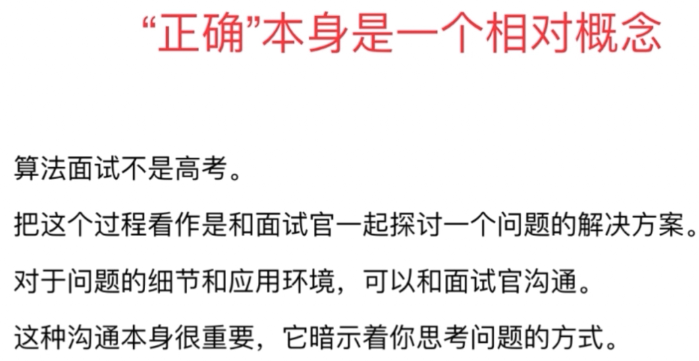
一提起算法面试，很多同学就会心有余悸。可其实，大多数企业的算法面试，并没有那么可怕。并不是一定要啃完整本《算法导论》，才能玩儿转算法面试；也并不是只有ACM参赛选手，才能笑傲算法面试。恰恰相反，大多数算法面试关注的算法思维，其实很基础。在这一章，和大家聊一聊，算法面试，到底是什么鬼？...

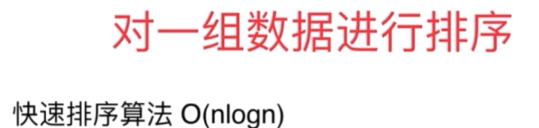
### 1.1算法面试不仅仅是正确的回答问题

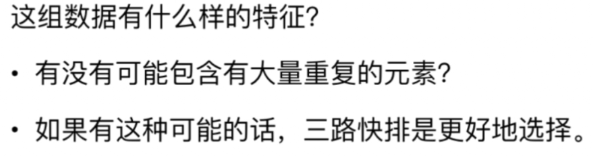


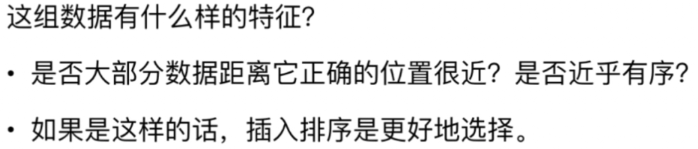


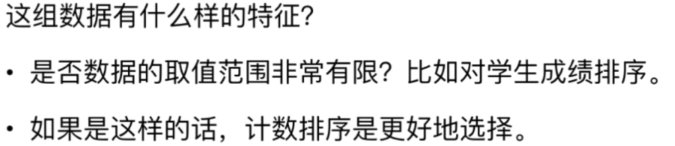


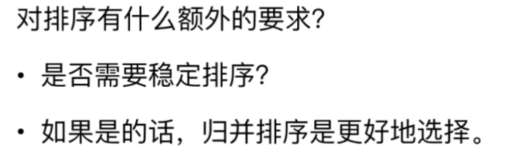


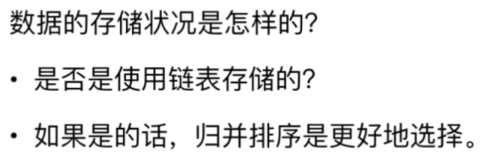




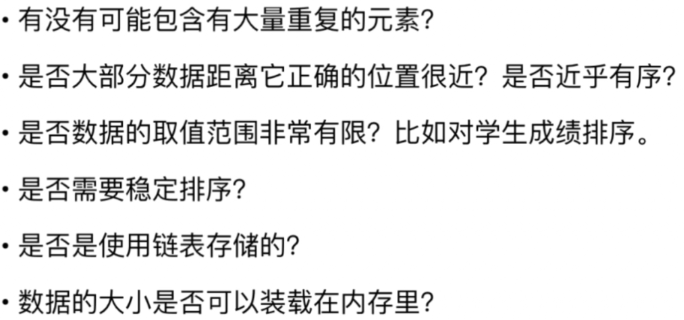


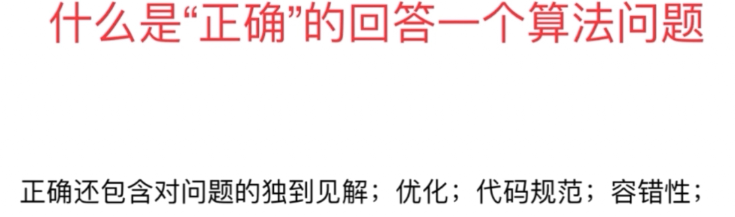






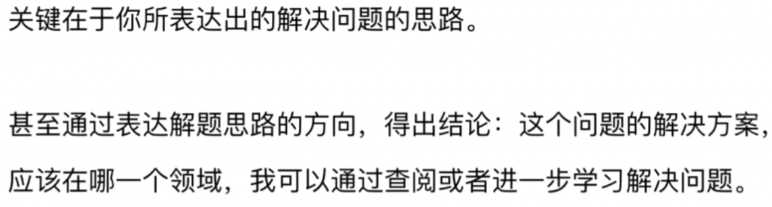




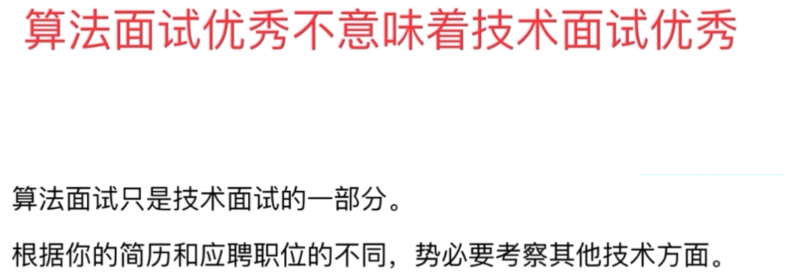


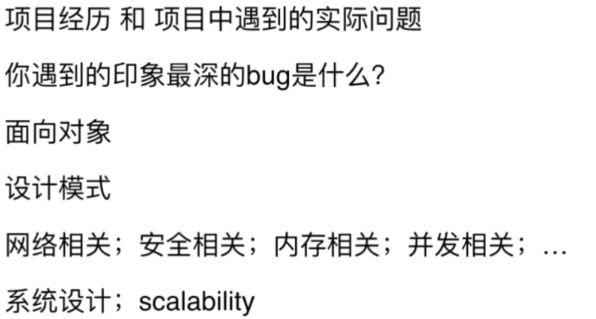




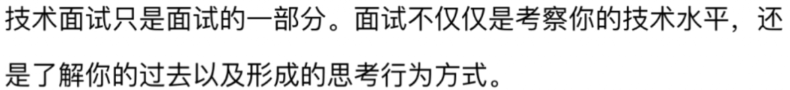


### 1.2算法面试只是面试的一部分

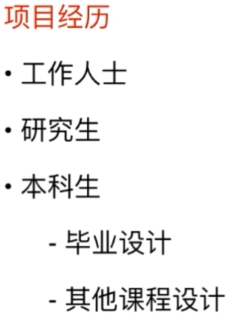


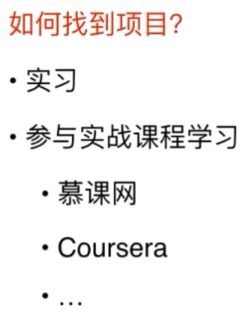


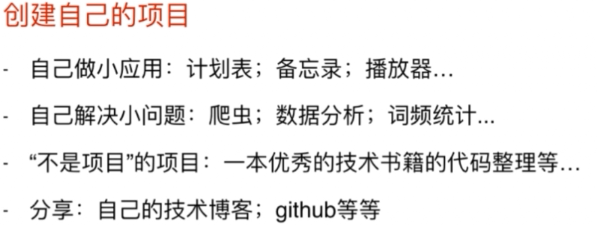


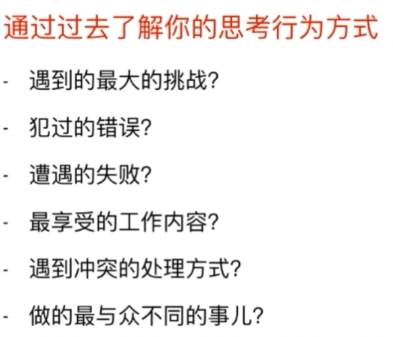




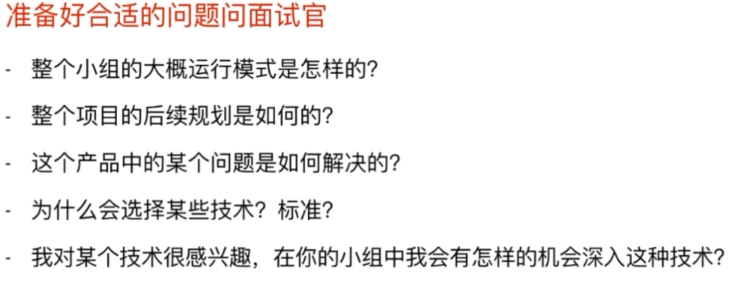




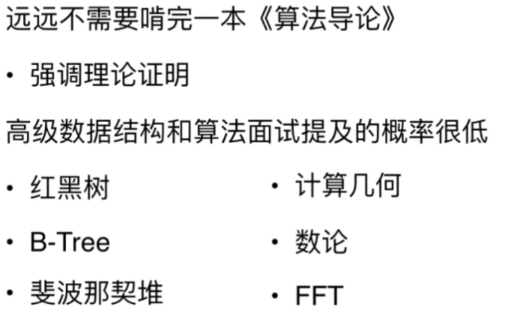




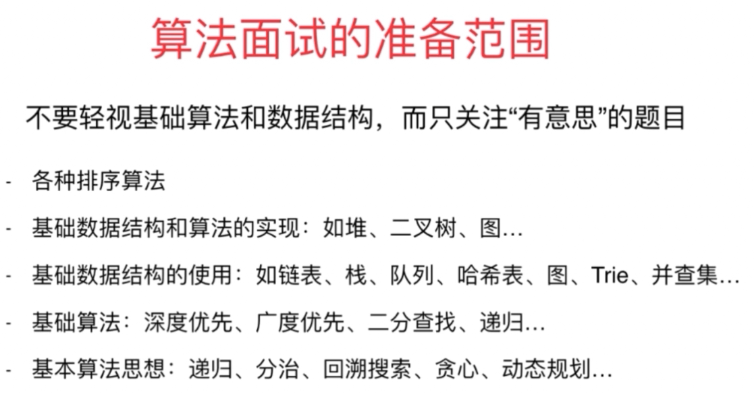
**和项目挂钩**

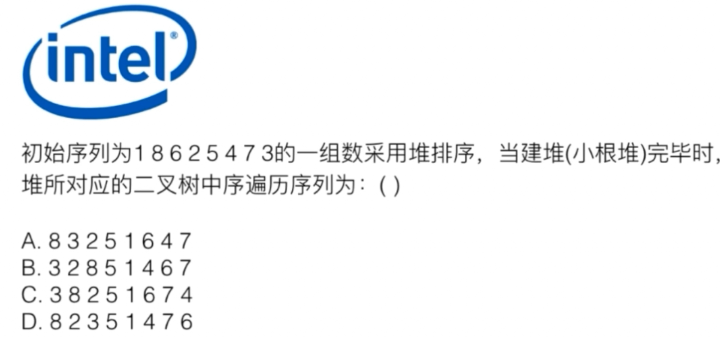


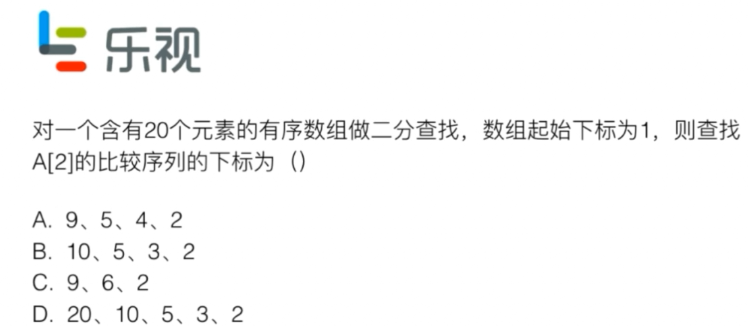
### 1.3如何准备算法面试





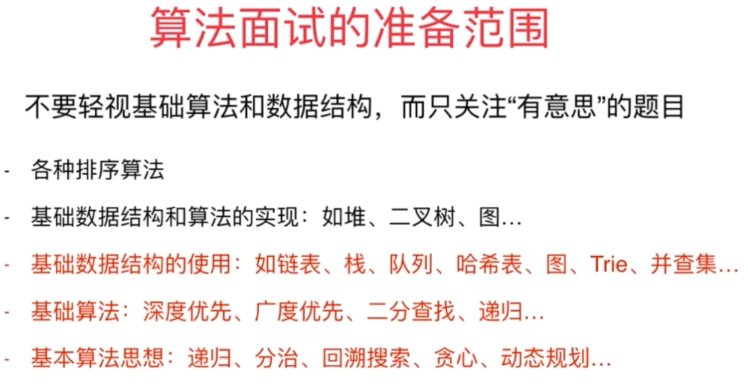
















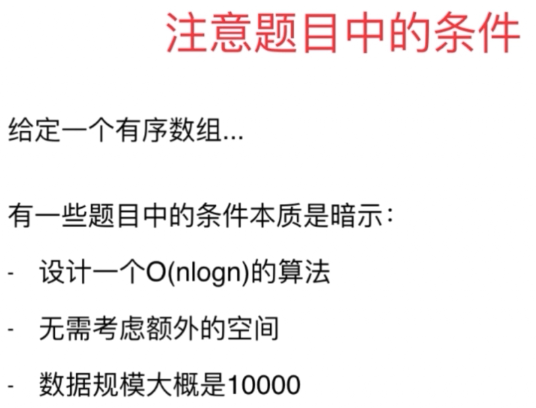




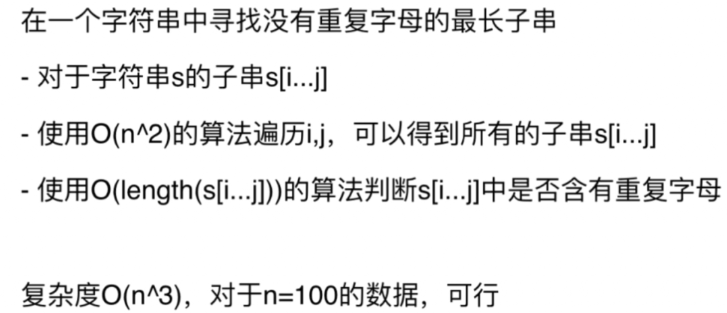


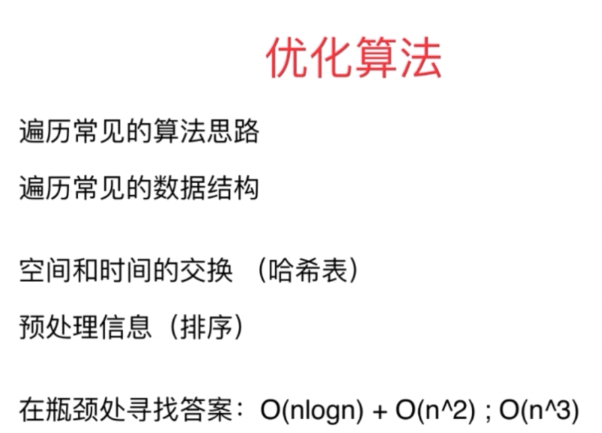
### 1.4 如何回答算法面试问题

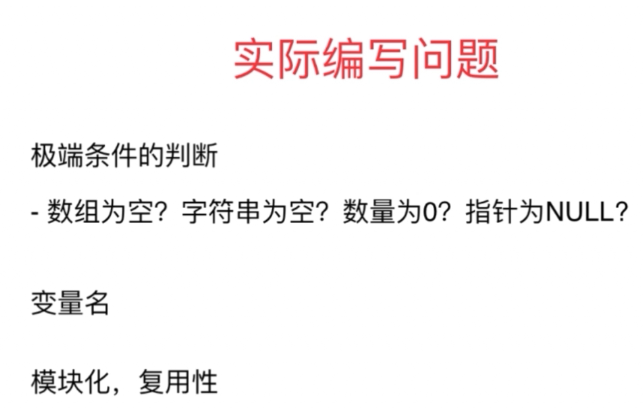










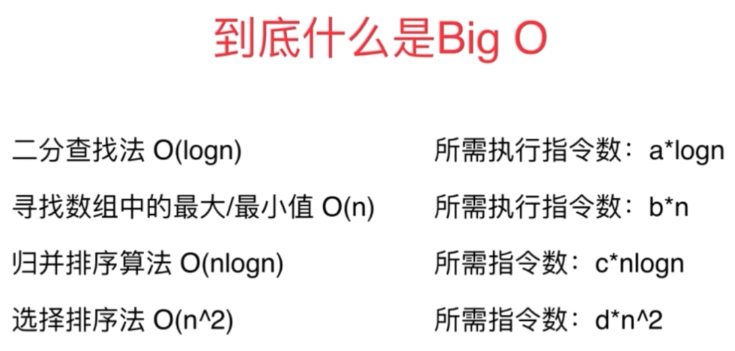




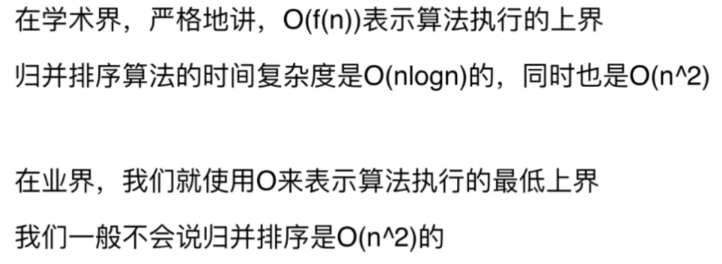
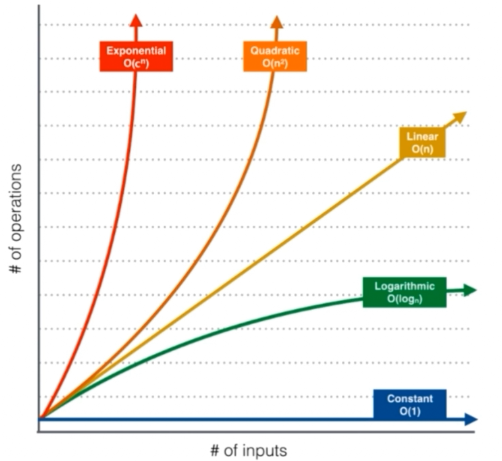
## 第2章 面试中的复杂度分析

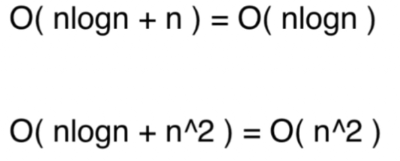
很多同学一提起复杂度分析就头疼，马上想起了《算法导论》中复杂的数学推导。但其实在一般的企业面试中，对复杂度的分析要求并没有那么高，但也是绕不过去的坎儿。在这一章，和大家介绍一下，面试中需要掌握的复杂度分析。...

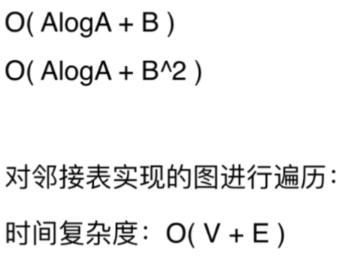
### 2.1 究竟什么是大O（Big O）

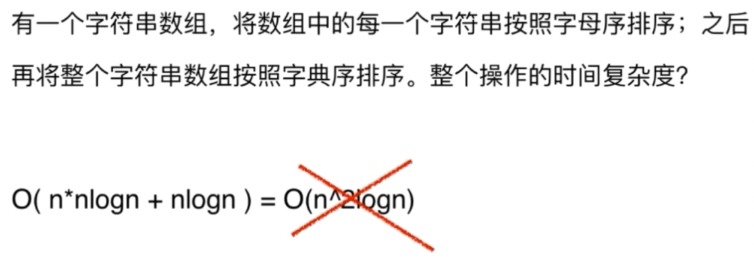


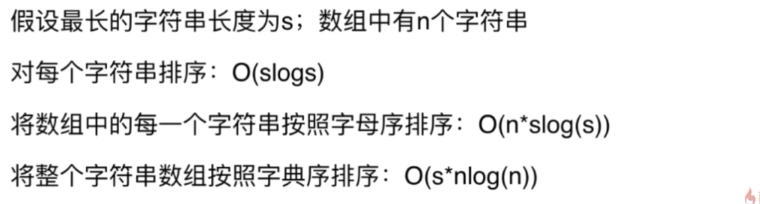


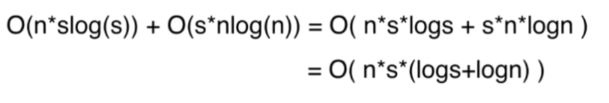


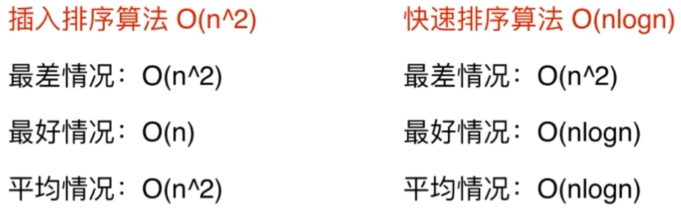






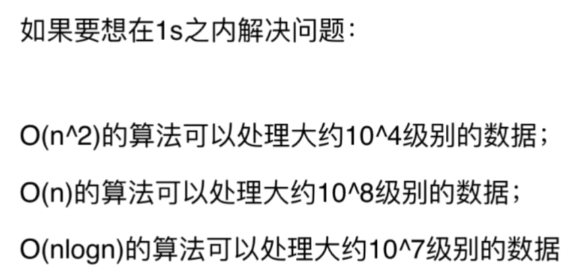


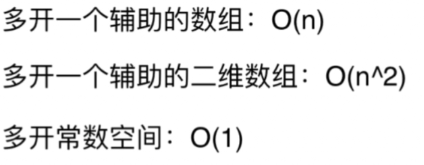




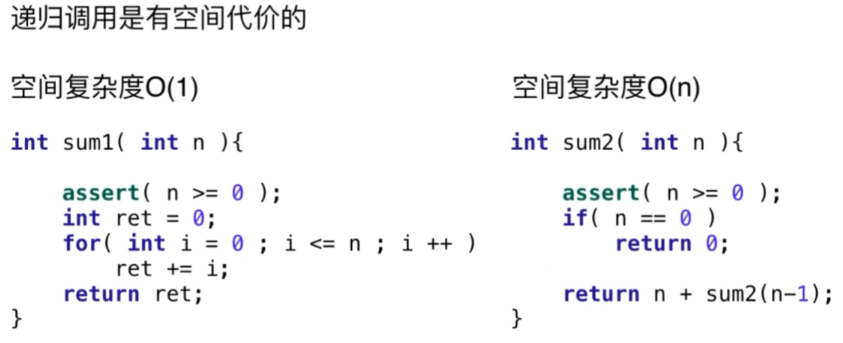
### 2.2 对数据规模有一个概念

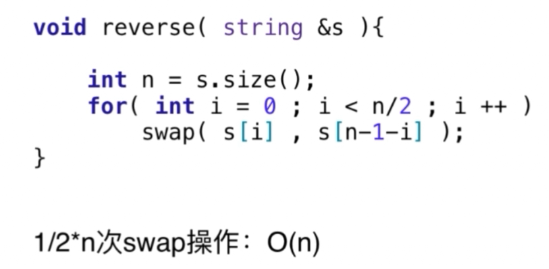


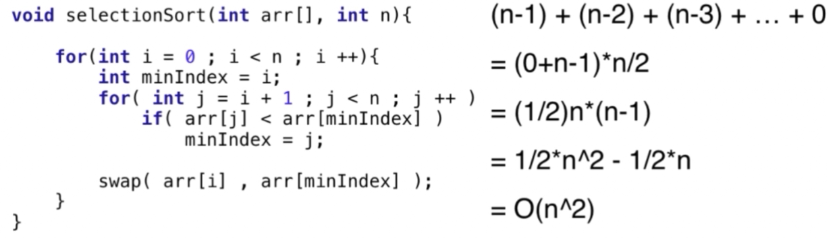


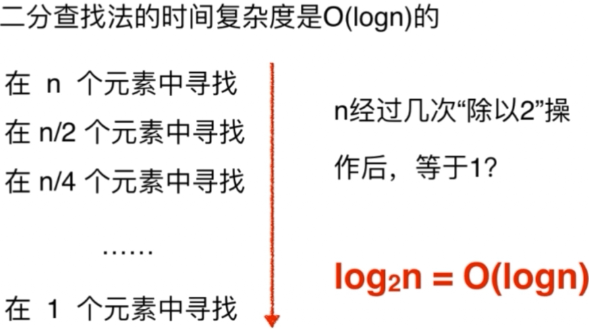


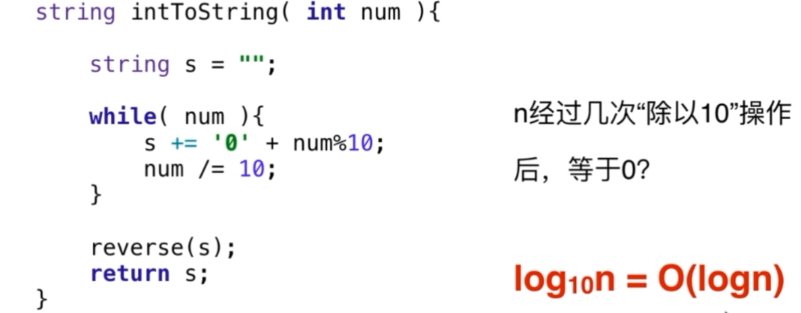
### 2.3 简单的复杂度分析

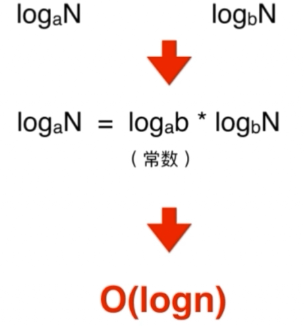


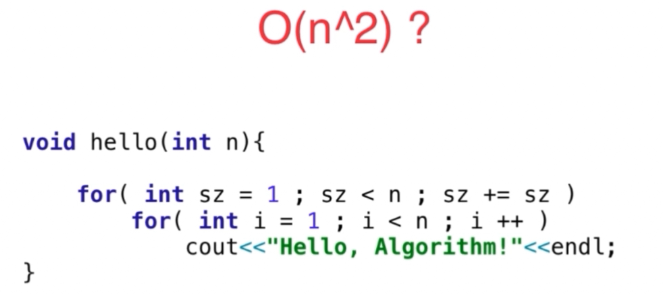


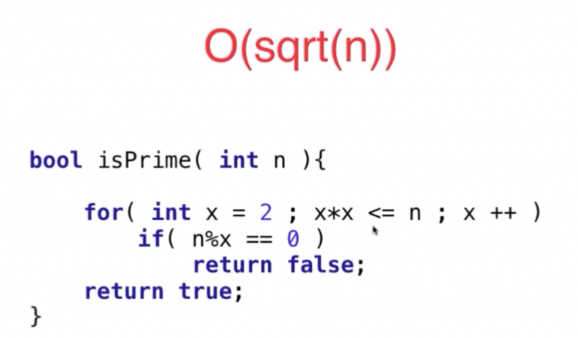




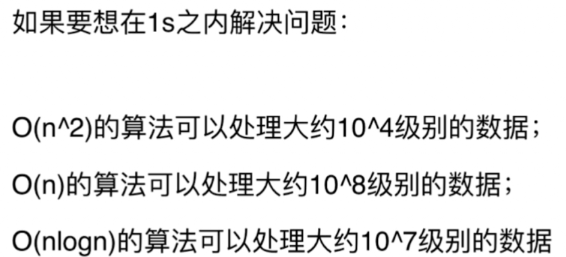


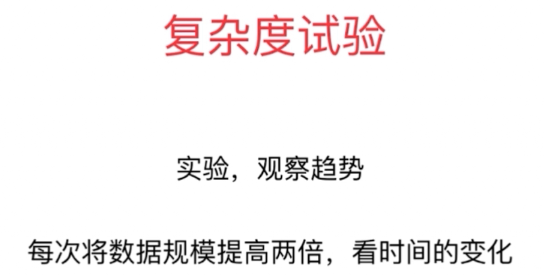




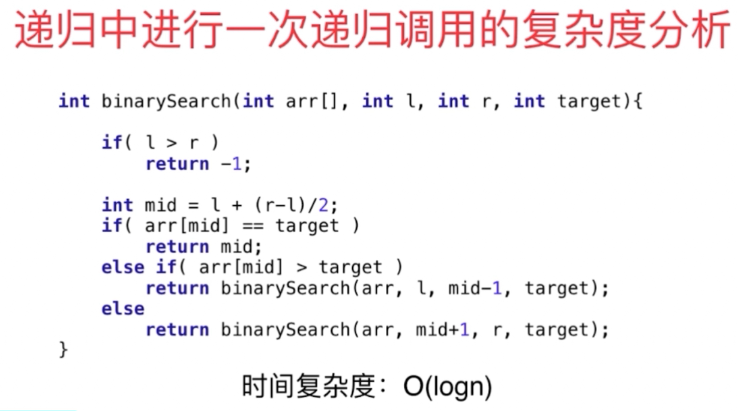


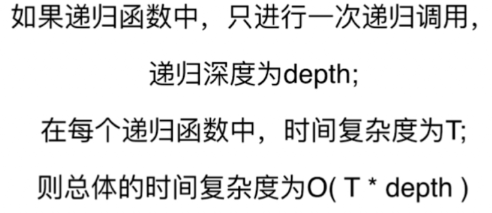
### 2.4 亲自试验自己算法的时间复杂度

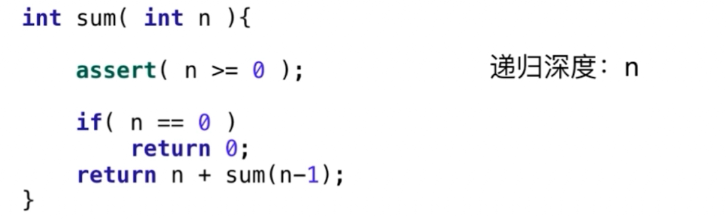




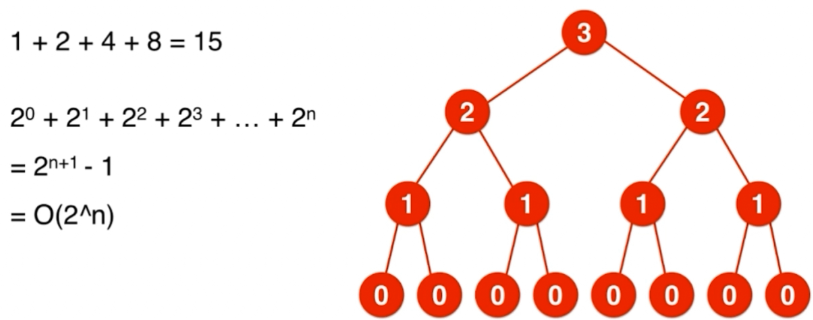
### 2.5 递归算法的复杂度分析



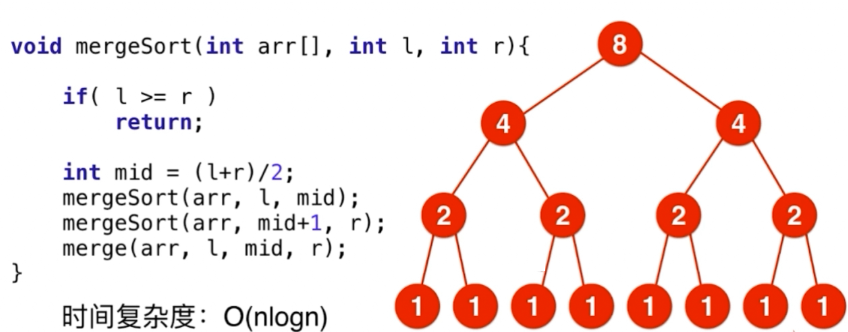








指数问题：通过剪枝、动态规划降低时间复杂度



### 2.6 均摊时间复杂度分析（Amortized Time Analysis）

|  |
| --- |
| // 平均复杂度为 O(1)  public void push\_back(Item e){  if(size == capacity)  resize(2 \* capacity);  data[size++] = e;  }  // 平均复杂度为 O(1)  public Item pop\_back(){  if(size <= 0)  throw new IllegalArgumentException("can not pop back for empty vector.");  Item ret = data[size-1];  size --;  // 在size达到静态数组最大容量的1/4时才进行resize  // resize的容量是当前最大容量的1/2  // 防止复杂度的震荡  if(size == capacity / 4)  resize(capacity / 2);  return ret;  } |
| // 复杂度为 O(n)  private void resize(int newCapacity){  assert newCapacity >= size;  Item[] newData = (Item[])new Object[newCapacity];  for(int i = 0 ; i < size ; i ++)  newData[i] = data[i];  data = newData;  capacity = newCapacity;  } |

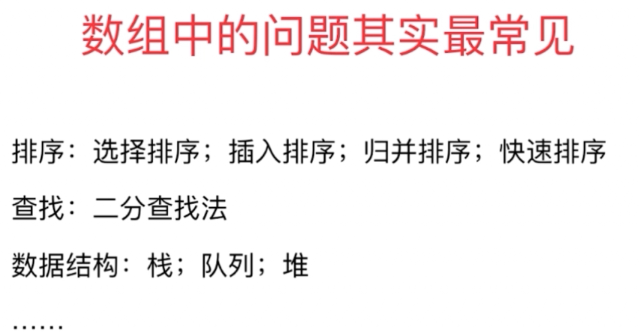
但前n次操作都是O(1)的，只有第n+1次操作是O(1)，总的来说是O(1)的

### 2.7 避免复杂度的震荡

## 第3章 数组中的问题其实最常见

面试中的算法问题，有很多并不需要复杂的数据结构支撑。就是用数组，就能考察出很多东西了。其实，经典的排序问题，二分搜索等等问题，就是在数组这种最基础的结构中处理问题的。在这一章中，我们学习常见的数组中处理问题的方法。...

### 3.1 从二分查找法看如何写出正确的程序





|  |
| --- |
| **public class** BinarySearch {   **private** BinarySearch(){}   **public static int** binarySearch(Comparable[] arr, **int** n, Comparable target){   **int** l = 0, r = n - 1; *// 在[l...r]的范围里寻找target* **while**(l <= r){ *// 当 l == r时,区间[l...r]依然是有效的* **int** mid = l + (r - l) / 2;  **if**(arr[mid].compareTo(target) == 0) **return** mid;  **if**(target.compareTo(arr[mid]) > 0)  l = mid + 1; *// target在[mid+1...r]中; [l...mid]一定没有target* **else** *// target < arr[mid]* r = mid - 1; *// target在[l...mid-1]中; [mid...r]一定没有target* }   **return** -1;  }   **public static void** main(String[] args) {   **int** n = (**int**)Math.*pow*(10, 7);  Integer data[] = Util.*generateOrderedArray*(n);   **long** startTime = System.*currentTimeMillis*();  **for**(**int** i = 0 ; i < n ; i ++)  **if**(i != *binarySearch*(data, n, i))  **throw new** IllegalStateException(**"find i failed!"**);  **long** endTime = System.*currentTimeMillis*();   System.***out***.println(**"Binary Search test complete."**);  System.***out***.println(**"Time cost: "** + (endTime - startTime) + **" ms"**);  } } |
| **public class** Util {   **private** Util(){}   **public static** Integer[] generateRandomArray(**int** n, **int** rangeL, **int** rangeR) {   **assert** n > 0 && rangeL <= rangeR;   Integer[] arr = **new** Integer[n];  **for** (**int** i = 0; i < n; i++)  arr[i] = (**int**)(Math.*random*() \* (rangeR - rangeL + 1)) + rangeL;  **return** arr;  }   **public static** Integer[] generateOrderedArray(**int** n) {   **assert** n > 0;   Integer[] arr = **new** Integer[n];   **for** (**int** i = 0; i < n; i++)  arr[i] = i;  **return** arr;  } } |
| Binary Search test complete.  Time cost: 1397 ms |

### 3.2 改变变量定义，依然可以写出正确的算法

[o...k) 和[0...k]

### 3.3 在LeetCode上解决第一个问题 Move Zeros

|  |
| --- |
| **package** MoveZores;  **class** Solution {  **private void** swap(**int**[] nums, **int** a, **int** b){  **int** tmp = nums[a];  nums[a] = nums[b];  nums[b] = tmp;  }  **public void** moveZeroes(**int**[] nums) {  **int** k = 0; *// nums中，[0...k)的元素均为非0元素* **for** (**int** i = 0; i < nums.**length**; i++) {  **if** (nums[i] != 0)  **if** (i != k)  swap(nums,k++,i);  **else** k++;  }  }   **public static void** main(String[] args) {  **int**[] nums = {1,0,3,0,5};  **new** Solution().moveZeroes(nums);  **for** (**int** i = 0; i < nums.**length**; i++) {  System.***out***.print(nums[i] + **" "**);  }  } } |
|  |
| 1 3 5 0 0 |

|  |
| --- |
| **package** RemoveElements;  **class** Solution {  **public int** removeElement(**int**[] nums, **int** val) {  **int** k = 0; *// [0...k)为非val值的元素* **for** (**int** i = 0; i < nums.**length**; i++) {  **if** (nums[i] != val)  **if** (k!=i){  **int** tmp = nums[k];  nums[k++] = nums[i];  nums[i] = tmp;  } **else** {  k ++;  }  }  **return** k;  } } |

|  |
| --- |
| **package** RemoveDuplicatedElements;  **class** Solution {  **public int** removeDuplicates(**int**[] nums) {  **int** k = 1; *// [0,k)存放非重复元素* **for** (**int** i = 1; i < nums.**length**; i++) {  **if** (nums[i] != nums[k-1]){  **int** tmp = nums[k];  nums[k++] = nums[i];  nums[i] = tmp;  }  }  **return** k;  }   **public static void** main(String[] args) {  **int**[] nums = {0,0,1,1,1,2,2,3,3,4};  System.***out***.println(**new** Solution().removeDuplicates(nums));  **for** (**int** i = 0; i < nums.**length**; i++) {  System.***out***.print(nums[i] + **" "**);  }  } } |
| **5**  **0 1 2 3 4 0 2 1 3 1** |

### 3.4 即使简单的问题，也有很多优化的思路

|  |
| --- |
| **package** RemoveDuplicatedElements2;  **class** Solution {  **public int** removeDuplicates(**int**[] nums) {  **int** k = 0; *// [0,k)存放非重复元素* **for** (**int** i = 0; i < nums.**length**; i++) {  **if** (k < 2 || nums[i] > nums[k-2]){  **if** (k!=i)  nums[k++] = nums[i];  **else** k++;  }  }  **return** k;  }   **public static void** main(String[] args) {  **int**[] nums = {0,0,1,1,1,2,2,3,3,4};  System.***out***.println(**new** Solution().removeDuplicates(nums));  **for** (**int** i = 0; i < nums.**length**; i++) {  System.***out***.print(nums[i] + **" "**);  }  } } |
| 9  0 0 1 1 2 2 3 3 4 4 |

### 3.5 三路快排partition思路的应用 Sort Color



|  |
| --- |
| **package** ColorClass75;  **import** java.util.Arrays;  **class** Solution {  **private void** swap(**int**[] nums, **int** a, **int** b){  **int** temp = nums[a];  nums[a] = nums[b];  nums[b] = temp;  }  **public void** sortColors(**int**[] nums) {  **int** zero = -1; *// [0...zero]存放0* **int** two = nums.**length**; *// [two...n-1]存放2* **for** (**int** i = 0; i < two;) {  **if** (nums[i] == 1)  i++;  **else if** (nums[i] == 2){  swap(nums, --two,i);  } **else** {  *// nums[i] == 0* **if** (nums[i]!=0)  **throw new** IllegalArgumentException(**"Not in [0,1,2]."**);  swap(nums, ++zero, i++);  }  }   }   **public static void** main(String[] args) {  **int**[] nums = {2,0,2,1,1,0};  **new** Solution().sortColors(nums);  System.***out***.println(Arrays.*toString*(nums));  } } |
| **[0, 0, 1, 1, 2, 2]** |

3.6 对撞指针 Two Sum II . Input Array is Sorted

11、成最多水的容器

|  |
| --- |
| **package** LargestCapacityOfWater11;  **class** Solution {  **public int** maxArea(**int**[] height) {  **int** l=0,r=height.**length**-1;  **int** maxArea=0;  **while** (l<r){  maxArea = Math.*max*(maxArea,Math.*min*(height[l],height[r]) \* (r-l));  **if** (height[l] < height[r])  l++;  **else** r--;   }  **return** maxArea;  }   **public static void** main(String[] args) {  **int**[] height = {1,8,6,2,5,4,8,3,7};  System.***out***.println(**new** Solution().maxArea(height));  } } |
| **49** |

3.7 滑动窗口 Minimum Size Subarray Sum

3.8 在滑动窗口中做记录 Longest Substring Without Repeating Characters

第4章 查找表相关问题

查找，是使用计算机处理问题时的一个最基本的任务，因此也是面试中非常常见的一类问题。很多算法问题的本质，就是要能够高效查找。学会使用系统库中的map和set，就已经成功了一半。

4.1 set的使用 Intersection of Two Arrays

4.2 map的使用 Intersection of Two Arrays II

4.3 set和map不同底层实现的区别

4.4 使用查找表的经典问题 Two Sum

4.5 灵活选择键值 4Sum II

4.6 灵活选择键值 Number of Boomerangs

4.7 查找表和滑动窗口 Contain Duplicate II

4.8 二分搜索树底层实现的顺序性 Contain Duplicate III

第5章 在链表中穿针引线

链表是一种特殊的线性结构，由于不能像数组一样进行随机的访问，所以和链表相关的问题有他自身的特点。我将之称为穿针引线。我们在这一章，就来看一看，如何在链表中穿针引线。

5.1 链表，在节点间穿针引线 Reverse Linked List

5.2 测试你的链表程序

5.3 设立链表的虚拟头结点 Remove Linked List Elements

5.4 复杂的穿针引线 Swap Nodes in Pairs

5.5 不仅仅是穿针引线 Delete Node in a Linked List

5.6 链表与双指针 Remove Nth Node Form End of List

第6章 栈，队列，优先队列

栈和队列虽然是简单的数据结构，但是使用这些简单的数据结构所解决的算法问题不一定简单。在这一章里，我们将来探索，和栈与队列相关的算法问题。

6.1 栈的基础应用 Valid Parentheses

6.2 栈和递归的紧密关系 Binary Tree Preorder, Inorder and Postorder Traversal

6.3 运用栈模拟递归

6.4 队列的典型应用 Binary Tree Level Order Traversal

6.5 BFS和图的最短路径 Perfect Squares

6.6 优先队列

6.7 优先队列相关的算法问题 Top K Frequent Elements

第7章 二叉树和递归

递归，是使用计算机解决问题的一种重要的思考方式。而二叉树由于其天然的递归结构，使得基于二叉树的算法，均拥有着递归性质。使用二叉树，是研究学习递归算法的最佳入门方式。在这一章里，我们就来看一看二叉树中的递归算法。...

7.1 二叉树天然的递归结构

7.2 一个简单的二叉树问题引发的血案 Invert Binary Tree

7.3 注意递归的终止条件 Path Sum

7.4 定义递归问题 Binary Tree Path

7.5 稍复杂的递归逻辑 Path Sum III

7.6 二分搜索树中的问题 Lowest Common Ancestor of a Binary Search Tree

第8章 递归和回溯法

回溯法是解决很多算法问题的常见思想，甚至可以说是传统人工智能的基础方法。其本质依然是使用递归的方法在树形空间中寻找解。在这一章，我们来具体看一下将递归这种技术使用在非二叉树的结构中，从而认识回溯这一基础算法思想。...

8.1 树形问题 Letter Combinations of a Phone Number

8.2 什么是回溯

8.3 排列问题 Permutations

8.4 组合问题 Combinations

8.5 回溯法解决组合问题的优化

8.6 二维平面上的回溯法 Word Search

8.7 floodfill算法，一类经典问题 Number of Islands.

8.8 回溯法是经典人工智能的基础 N Queens

第9章 动态规划基础

很多同学听到“动态规划”的名称可能会望而生畏，觉得动态规划的问题都很复杂。但其实，动态规划本质依然是递归算法，只不过是满足特定条件的递归算法。在这一章里，我们就来逐步解开动态规划的神秘面纱

9.1 什么是动态规划

9.2 第一个动态规划问题 Climbing Stairs

9.3 发现重叠子问题 Integer Break

9.4 状态的定义和状态转移 House Robber

9.5 0.1背包问题

9.6 0.1背包问题的优化和变种

9.7 面试中的0.1背包问题 Partition Equal Subset Sum

9.8 LIS问题 Longest Increasing Subsequence

9.9 LCS，最短路，求动态规划的具体解以及更多

第10章 贪心算法

通常同学们可能会认为贪心算法比较简单。确实，通常贪心算法的实现非常容易，但是，一个问题是否能够使用贪心算法，是一定要小心的。我们在这一章来看一看，贪心算法可能会有哪些坑。

10.1 贪心基础 Assign Cookies

10.2 贪心算法与动态规划的关系 Non.overlapping Intervals

10.3 贪心选择性质的证明

第11章 课程结语

看完整个课程，我不能保证所有的同学都能百分百地对每一个算法面试问题应答自如，但认真学习的同学对大部分问题都应该已经有了一个合理的思维路径。在最后一章，我们再来简单地总结一下，并祝每一位同学都能找到自己喜欢的工作，大展宏图：）...

11.1 结语