前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

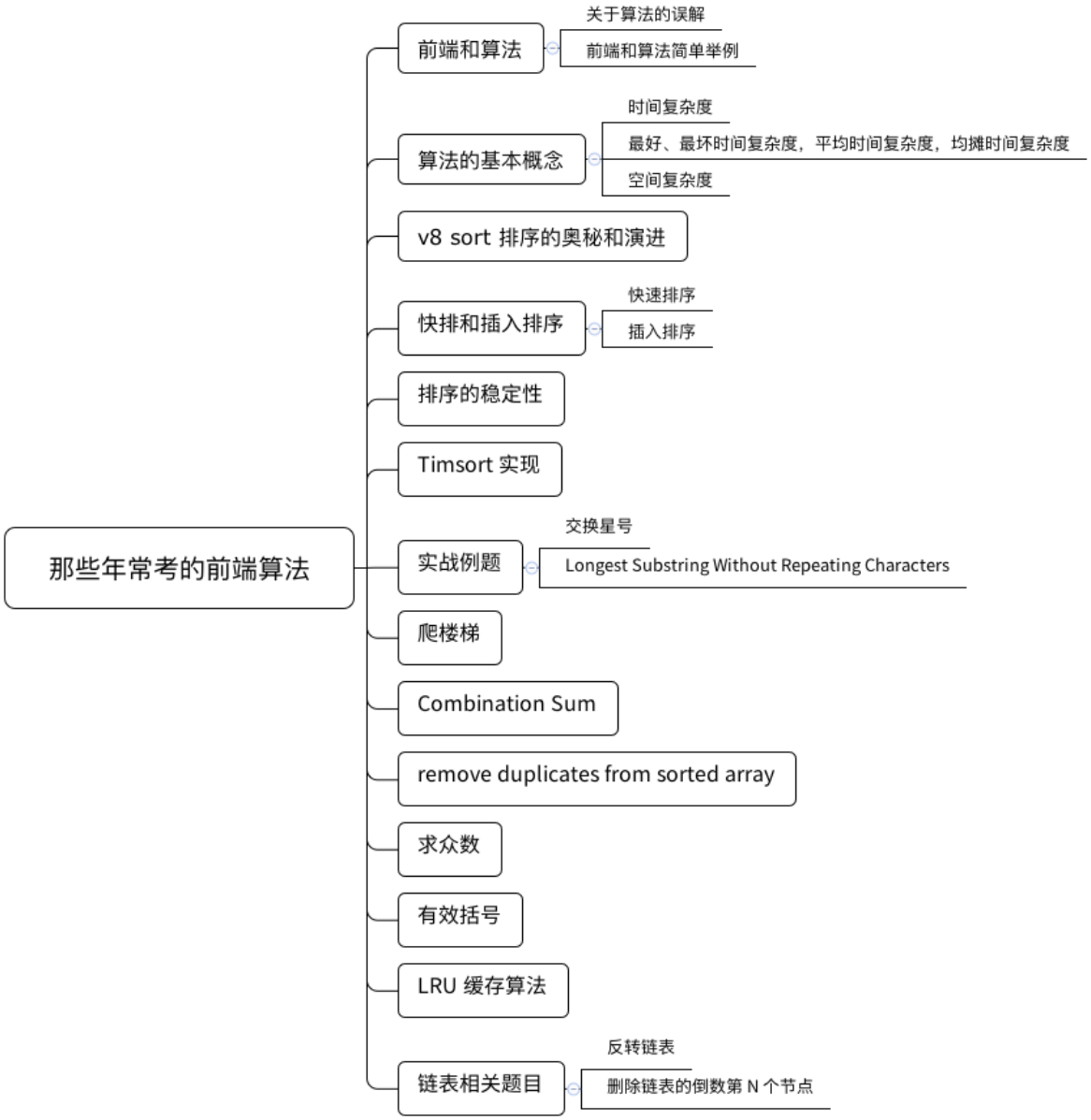
查看详情 >

那些年常考的前端算法（3）

前面课程，我们总结了前端和算法的关系，在上一讲中，也已经通过两道题目开启了「刷算法」的热身。算法是面试中必不可少的部分，尤其对于高阶职位来说，算法题目是面试环节的「最难」和「最关键」的环节。

算法说难也不难，我们大可不必「谈虎色变」，有策略地「刷算法题」将会使你更有信心。我认为在课程中一味地「秀算法」，找最高深最偏的算法分析没有任何意义。这里我总结出一些经典的算法题目，我常用来考察候选者以及我作为面试者遇到的一些题目来讲解。

主要内容如下：



爬楼梯

题目：假设我们需要爬一个楼梯，这个楼梯一共有 N 阶，可以一步跨越 1 个或者 2 个台阶，那么爬完楼梯一共有多少种方式?

示例：输入 2（标注 N = 2，一共是 2 级台阶）；

输出：2 （爬完一共两种方法：一次跨两阶 + 分两次走完，一次走一阶）

示例：输入 3；输出 3（1 阶 + 1 阶 + 1 阶；1 阶 + 2 阶；2 阶 + 1 阶）

思路：最直接的想法其实类似 Fibonacci 数列，使用递归比较简单。比如我们爬 N 个台阶，其实就是爬 N − 1 个台阶的方法数 + 爬 N − 2 个台阶的方法数。

解法：

```
const climbing = n => {
   if (n == 1) return 1
   if (n == 2) return 2
   return climbing(n - 1) + climbing(n - 2)
}
```

我们来分析一下时间复杂度：递归方法的时间复杂度是高度为 n-1 的不完全二叉树节点数，因此近似为 O(2^n)，具体数学公式不再展开。

我们来尝试进行优化。实际上，上述的计算过程肯定都包含了不少重复计算，比如 climbing(N) + climbing(N − 1) 后会计算 climbing(N − 1) + climbing(N − 2)，而实际上 climbing(N − 1) 只需要计算一次就可以了。

优化方案：

```
const climbing = n => {
   let array = []
   const step = n => {
       if (n == 1) return 1
       if (n == 2) return 2
       if (array[n] > 0) return array[n]

       array[n] = step(n - 1) + step(n - 2)
       return array[n]
   }
   return step(n)
}
```

我们使用了一个数组 array 来储存计算结果，时间复杂度为 O(n)。

另外一个优化方向是：所有递归都可以用循环来代替。

```
const climbing = n => {
   if (n == 1) return 1
```

```
    if (n == 2) return 2

    let array = []
    array[1] = 1
    array[2] = 2

    for (let i = 3; i<= n; i++) {
        array[i] = array[i - 1] + array[i - 2]
    }
    return array[n]
}
```

时间复杂度仍然为 O(n)，但是我们优化了内存的开销。

因此这道题看似「困难」，其实就是一个 Fibonacci 数列。很多算法题目都是类似的，也许第一次读题会觉得没有思路，但是隐藏在题目后边的解决方案，其实就是我们常见的知识。

Combination Sum

这个算法，让我们来聚焦「回溯」这两个字，题目出处 Combination Sum。

题目：给定一组不含重复数字的非负数组和一个非负目标数字，在数组中找出所有数加起来等于给定的目标数字的组合。

示例：输入

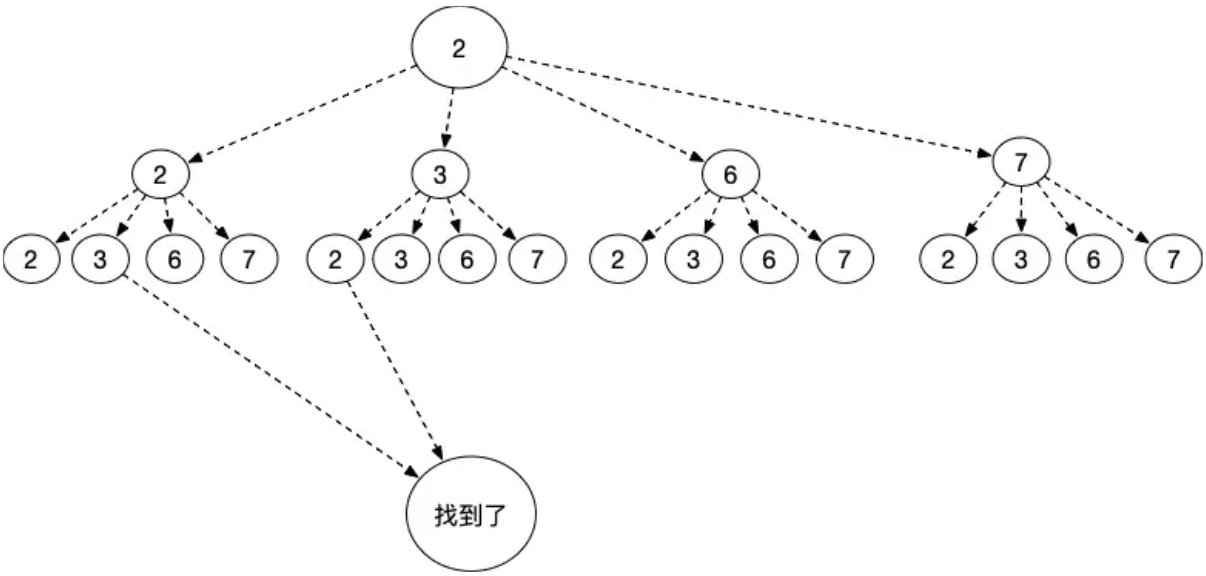
```
const array = [2, 3, 6, 7]
const target = 7
```

输出：

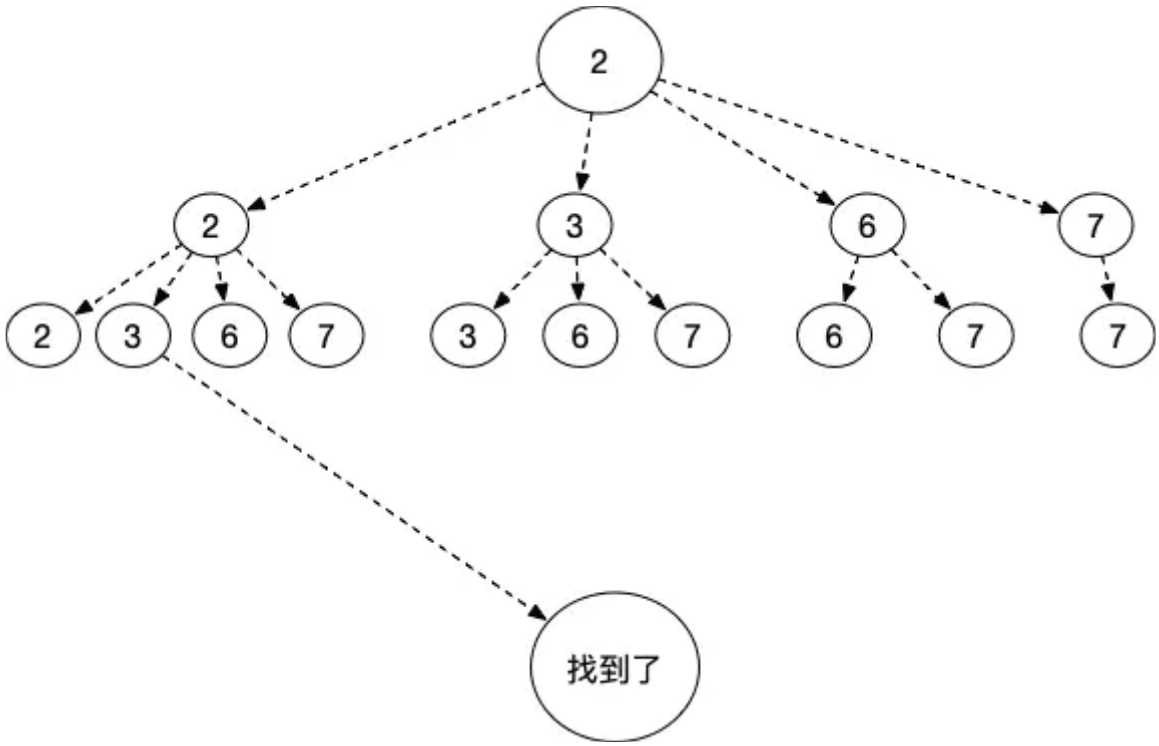
```
[
 [7],
 [2,2,3]
]
```

我们直接来看优化后的思想：回溯解决问题的套路就是先用「笨办法」，遍历所有的情况来找出问题的解，在这个遍历过程当中，以深度优先的方式搜索解空间，并且在搜索过程中用剪枝函数避免无效搜索。

回到这个问题，我们先通过图来遍历所有情况：



对于这个题目，事实上我们思考，数组 [2, 2, 3] 和 [2, 3, 2] 实际是重复的，因此可以删除掉重复的项，优化递归树为：



我们该如何用代码描述上述过程呢？这时候需要一个临时数组 tmpArray，进入递归前 push 一个结果，

最终答案：

```
const find = (array, target) => {
    let result = []

    const dfs = (index, sum, tmpArray)  => {
        if (sum === target) {
            result.push(tmpArray.slice())
        }

        if (sum > target) {
            return
        }

        for (let i = index; i < array.length; i++) {
            tmpArray.push(array[i])

            dfs(i, sum + array[i], tmpArray)

            tmpArray.pop()
        }
    }

    dfs(0, 0, [])

    return result
}
```

如果读者存在理解问题，建议打断点调试一下。回溯是一个非常常见的思想，这也是一个典型的回溯常考题目。

另外，该题有另一个变种：

从一个数组中找出 N 个数，其和为 M 的所有可能。

这里我们指定数组元素个数的和，需要这个和为指定值。

举例：从数组 [1, 2, 3, 4] 中选取 2 个元素，求和为 5 的所有可能。答案是两组组合: [1, 4] 和 [2, 3]。

这里我们介绍一种借助「二进制」实现的解法，可以用 0 和 1 来表示数组中相应的元素是否被选中。因此，对于一个长度为 4 的数组来说：

0000 表示没有选择数组中的任何元素

0100 表示选择了数组中第 1 位元素

以此类推，数组长度为 4，那么上述情况一共有 16 种可能（Math.pow(length, 2)）。

而这道题目中，只需要选择指定数组元素个数的和，还是对于数组长度为 4 的情况：只需要考虑 0011 等 1 的个数累加为 0 case，而不需要考虑类似 0111 这样的 case。

针对符合个数的所有情况，我们进行数组项目的求和，判断是否等于指定值的情况即可：

```
const find = (array, target, sum) => {
 const len = array.length
 let result = []

 for (let i = 0; i < Math.pow(2, len); i++) {
   if (getCount(i) == target) {
     let s = 0
     let temp = []
     for (let j = 0; j < len; j++) {
       if (i & 1 << (len - 1 -j)) {
         s += array[j]
         temp.push(array[j])
       }
     }
```

```
      if (s == sum) {
        result.push(temp)
      }
    }
  }
  return result
}

function getCount(i) {
  let count = 0;
  while (i) {
    if (i & 1){
      ++count
    }
    i >>= 1
  }
  return count
}
```

remove duplicates from sorted array

题目：对一个给定一个排序数组去重，同时返回去重后数组的新长度。

难点：这道题并不困难，但是需要临时加一些条件，即需要原地操作，在使用 O(1) 额外空间的条件下完成。

示例：

输入：

```
let array = [0,0,1,1,1,2,2,3,3,4]
```

输出：

```
console.log(removeDuplicates(array))
// 5
```

```
console.log(array)
// 0, 1, 2, 3, 4
```

这道题既然规定 in-place 的操作，那么可以考虑算法中的另一个重要思想：双指针。

使用快慢指针：

开始时，快指针和慢指针都指向数组中的第一项

如果快指针和慢指针指的数字相同，则快指针向前走一步

如果快指针和慢指针指的数字不同，则两个指针都向前走一步，同时快指针指向的数字赋值给慢指针指向的数字

当快指针走完整个数组后，慢指针当前的坐标加 1 就是数组中不同数字的个数

代码很简单：

```
const removeDuplicates = array => {
   const length = array.length
```

```
    let slowPointer = 0

    for (let fastPointer = 0; fastPointer < length;
fastPointer ++) {
        if (array[slowPointer] !== array[fastPointer]) {
            slowPointer++
            array[slowPointer] = array[fastPointer]
        }
    }
}
```

这道题目如果不要求 O(n) 的时间复杂度， O(1) 的空间复杂度，那么会非常简单。如果进行空间复杂度要求，尤其是 in-place 操作，开发者往往可以考虑双指针的思路。

求众数

这也是一道简单的题目，关键点在于如何优化。

题目：给定一个大小为 N 的数组，找到其中的众数。众数是指在数组中出现次数大于 N/2 的元素。

可能大家都会想到使用一个额外的空间，记录元素出现的次数，我们往往用一个 map 就可以轻易地实现。那优化点在哪里呢？答案就是投票算法。

```
const find = array => {
    let count = 1
    let result = array[0]

    for (let i = 0; i < array.lenght; i++) {
        if (count === 0) result = array[i]

        if (array[i] === result) {
            count++
        }
        else {
```

```
            count--
        }
    }

    return result
}
```

有效括号

有效括号这个题目和前端息息相关，在之前课程模版解析时，其实都需要类似的算法进行模版的分析，进而实现数据的绑定。我们来看题目：

举例：输入 "()"

输出：true

举例：输入 "()[]{}"

输出：true

举例：输入 "{[]}"

输出：false

举例：输入 "([)]"

输出：false

这道题目的解法非常典型，就是借助栈实现，将这些括号自右向左看做栈结构。我们把成对的括号分为左括号和右括号，需要左括号和右括号一一匹配，通过一个 Object 来维护关系：

```
let obj = {
    "]": "[",
    "}": "{",
```

```
    ")": "(",
}
```

如果编译器中在解析时，遇见左括号，我们就入栈；如果是右括号，就取出栈顶元素检查是否匹配。如果匹配，就出栈；否则，就返回 false。

```
const isValid = str => {
    let stack = []
    var obj = {
        "]": "[",
        "}": "{",
        ")": "(",
    }

    for (let i = 0; i < str.length; i++) {
        if(str[i] === "[" || str[i] === "{" || str[i] === "
(") {
            stack.push(str[i])
        }
        else {
            let key = stack.pop()
            if(obj[key] !== str[i]) {
                return false
            }
        }
    }

    if (!stack.length) {
        return true
    }

    return false
};
```

LRU 缓存算法

看了这么多小算法题目，我们来换一个口味，现在看一个算法的实际应用。

LRU（Least Recently Used）算法是缓存淘汰算法的一种。简单地说，由于内存空间有限，需要根据某种策略淘汰不那么重要的数据，用以释放内存。LRU 的策略是最早操作过的数据放最后，最晚操作过的放开始，按操作时间逆序，如果达到上限，则淘汰末尾的项。

整个 LRU 算法有一定的复杂度，并且需要很多功能扩展。因此在生产环境中建议直接使用成熟的库，比如 npm 搜索 lru-cache。

这里我们尝试实现一个微型体统级别的 LRU 算法：

运用你所掌握的数据结构，设计和实现一个 LRU（最近最少使用）缓存机制。它应该支持以下操作：获取数据 get 和 写入数据 put 。

获取数据 get(key) – 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 –1。

写入数据 put(key, value) – 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

我们先来整体思考：尽量满足 O(1) 的时间复杂度中完成获取和写入的操作，那么可以使用一个 Object 来进行存储，如果 key 不是简单类型，可以使用 Map 实现：

```
const LRUCache = function(capacity) {
 // ...
 this.map = {};
 // ...
};
```

在这个算法中，最复杂的应该是淘汰策略，淘汰数据的时间复杂度必须是 O(1) 的话，我们一定需要额外的数据结构来完成 O(1) 的淘汰策略。那应该用什么样的数据结构呢？答案是双向链表。

链表在插入与删除操作上，都是 O(1) 时间的复杂度，唯一有问题的查找元素过程比较麻烦，是 O(n)。但是这里我们不需要使用双向链表实现查找逻辑，因为 map 已经很好的弥补了缺陷。

赘述一下：**我们在写入值的时候，判断缓存容量是否已经达到上限，如果缓存容量达到上限时，应该删除最近最少使用的数据值，从而为以后的新的数据值留出空间。**

结合链表的话，我们将刚刚写入的目标值设置为链表的首项，超过限制，就删除链表的尾项。

最终实现：

```
const LRUCache = function(capacity) {
 this.map = {}
 this.size = 0
 this.maxSize = capacity

 // 链表初始化，初始化只有一个头和尾
 this.head = {
   prev: null,
   next: null
 }
 this.tail = {
   prev: this.head,
   next: null
 }

 this.head.next = this.tail
};

LRUCache.prototype.get = function(key) {
```

```
  if (this.map[key]) {
    const node = this.extractNode(this.map[key])

    // 最新访问，将该节点放到链表的头部
    this.insertNodeToHead(node)

    return this.map[key].val
  }
  else {
    return -1
  }
}

LRUCache.prototype.put = function(key, value) {
  let node

  if (this.map[key]) {
    // 该项已经存在，更新值
    node = this.extractNode(this.map[key])
    node.val = value
  }
  else {
    // 如该项不存在，新创造节点
    node = {
      prev: null,
      next: null,
      val: value,
      key,
    }

    this.map[key] = node
    this.size++
  }

  // 最新写入，将该节点放到链表的头部
  this.insertNodeToHead(node)
```

```
  // 判断长度是否已经到达上限
  if (this.size > this.maxSize) {
    const nodeToDelete = this.tail.prev
    const keyToDelete = nodeToDelete.key
    this.extractNode(nodeToDelete)
    this.size--
    delete this.map[keyToDelete]
  }
};

// 插入节点到链表首项
LRUCache.prototype.insertNodeToHead = function(node) {
  const head = this.head
  const lastFirstNode = this.head.next

  node.prev = head
  head.next = node
  node.next = lastFirstNode
  lastFirstNode.prev = node

  return node
}

// 从链表中抽取节点
LRUCache.prototype.extractNode = function(node) {
  const beforeNode = node.prev
  const afterNode = node.next

  beforeNode.next = afterNode
  afterNode.prev = beforeNode

  node.prev = null
  node.next = null

  return node
}
```

链表相关题目

在之前的课程中，我们介绍了链表这种数据结构。链表应用非常广泛，这里来熟悉两个常见的对链表的操作算法。

反转链表

题目：对一个单链表进行反转

输入：1→2→3→4→5→NULL

输出：5→4→3→2→1→NULL

最直观的解法是使用三个指针，把头节点变成尾节点点，进行遍历：下一个节点拼接到当前节点的头部，以此类推。这种方法的实现我们不再手写，而是重点关注一下递归解法。

递归解法就要先判断递归终止条件，当下一个节点为 null，找到尾节点时，将其返回。我们从后往前进行：

```
const reverseList = head => {
  // 到了尾节点，则返回尾节点
  if (head == null || head.next == null) {
      return head
  }
  else {
      let newhead = reverseList(head.next)
      // 将当前节点的一下节点的 next 指向，指向为当前节点
      head.next.next = head
      // 暂时情况当前节点的 next 指向
      head.next = null

      return newhead
  }
}
```

删除链表的倒数第 N 个节点

题目：给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

输入：1→2→3→4→5，和 n = 2

输出：1→2→3→5

这道题目的关键是如何优雅地找到倒数第 N 个节点。

我们当然可以使用两次循环，第一次循环得到整个链表的长度 L，那么需要删除的节点就位于 L - N + 1 位置处，第二次遍历到相关位置进行操作即可。

这道题其实是可以用一次遍历来解决的。我们需要使用双指针，快指针 fast 先前进 N，找到需要删除的节点；然后慢指针 slow 从 head 开始，和快指针 fast 一起前进，直到 fast 走到末尾。 此时 slow 的下一个节点就是要删除的节点，也就是倒数第 N 个节点。需要注意的是，如果快指针移动 N 步之后，已经到了尾部，那说明需要删除的就是头节点。

```
const removeNthFromEnd = (head, n) => {
    if (head === null) {
        return head
    }

    if (n === 0) {
        return head
    }

    let fast = head
    let slow = head

    // 快指针前进 N 步
    while (n > 0) {
        fast = fast.next
        n--
    }

    // 快指针移动 N 步之后，已经到了尾部，那说明需要删除的就是头节点
```

```
    if (fast === null) {
        return head.next
    }

    while (fast.next != null ){
        fast = fast.next
        slow = slow.next
    }

    slow.next=slow.next.next
    return head
}
```

这两道关于链表的题目都重点考察了对你链表结构的理解，其中是用到了多个指针，这也是解决链表题目的关键。

算法学习

本节课内容到这里，我们只是列举了一些算法题目，也算不上「题海战术」，但问题都比较典型。可是面对这些相对零散的内容，我们应该如何入手学习呢？只是一味的刷题，似乎效率低下而无趣。

我认为对于算法的学习，需要做到「分门别类」，按照不同类别的算法思想，遵循循序渐进的进步路线，才会「越来越有感觉」。我把算法的一些基础思想进行了归并：

- 枚举
- 模拟
- 递归/分治
- 贪心
- 排序

二分

倍增

构造

前缀和/差分

我们来简单总结一下这些算法基础思想。

枚举

枚举是基于已有知识来猜测，印证答案的一种问题求解策略。当拿到一道题目时，枚举这种「暴力解法」最容易想到。这其中重点是：

建立简洁的数学模型

想清楚枚举哪些要素

尝试减少枚举空间

举个例子：

一个数组中的数互不相同，求其中和为 0 的数对的个数

最笨的方法：

```
for (int i = 0; i < n; ++i)
 for (int j = 0; j < n; ++j)
   if (a[i] + a[j] == 0) ++ans;
```

我们来看看如何操作进行优化。如果 (a, b) 是答案，那么 (b, a) 也是答案，因此对于这种情况只需统计一种顺序之后的答案，最后再乘 2 就好了。

```
for (int i = 0; i < n; ++i)
 for (int j = 0; j < i; ++j)
   if (a[i] + a[j] == 0) ++ans;
```

如此一来，就减少了 j 的枚举范围，减少了这段代码的时间开销。然而这还不是最优解。

我们思考：两个数是否都一定要枚举出来呢？其实枚举第一个数之后，题目的条件已经帮我们确定了其他的要素（另一个数），如果能找到一种方法直接判断题目要求的那个数是否存在，就可以省掉枚举后一个数的时间了。代码实现很简单，我们就不动手实现了。

模拟

模拟。顾名思义，就是用计算机来模拟题目中要求的操作，我们只需要按照题面的意思来写就可以了。模拟题目通常具有码量大、操作多、思路繁复的特点。

这种题目往往考察开发者的「逻辑转化为代码」的能力。一道典型题目是：魔兽世界。

递归 & 分治

递归的基本思想是某个函数直接或者间接地调用自身，这样就把原问题的求解转换为许多性质相同但是规模更小的子问题。

递归和枚举的区别在于：枚举是横向地把问题划分，然后依次求解子问题，而递归是把问题逐级分解，是纵向的拆分。比如请尝试回答这几个问题：

孙悟空身上有多少根毛？答：一根毛加剩下的毛。 你今年几岁？答：去年的岁数加一岁，1999 年我出生。

递归代码最重要的两个特征：结束条件和自我调用。

```
int func(传入数值) {
 if (终止条件) return 最小子问题解;
 return func(缩小规模);
}
```

写递归的技巧，「明白一个函数的作用并相信它能完成这个任务，千万不要试图跳进细节」。千万不要跳进这个函数里面企图探究更多细节，否则就会陷入无穷的细节无法自拔，人脑能压几个栈啊。

先举个最简单的例子：遍历二叉树。

```
void traverse(TreeNode* root) {
 if (root == nullptr) return;
 traverse(root->left);
 traverse(root->right);
}
```

这几行代码就足以遍历任何一棵二叉树了。对于递归函数 traverse(root)，我们只要相信：给它一个根节点 root，它就能遍历这棵树，因为写这个函数不就是为了这个目的吗?

那么遍历一棵 N 叉数呢?

```
void traverse(TreeNode* root) {
 if (root == nullptr) return;
 for (child : root->children) traverse(child);
}
```

总之，还是那句话：给它一个根节点 root，它就能遍历这棵树，不管你是几个叉。

典型题目：

给一棵二叉树，和一个目标值，节点上的值有正有负，返回树中和等于目标值的路径条数

这道题目解法很多，也比较典型。这里我们只谈思想，具体实现就不展开。

分治算法可以分三步走：分解 -> 解决 -> 合并。

分解原问题为结构相同的子问题

分解到某个容易求解的边界之后，进行递归求解

将子问题的解合并成原问题的解

归并排序是最典型的分治算法。

```
void mergeSort(一个数组) {
 if (可以很容易处理) return
 mergeSort(左半个数组)
 mergeSort(右半个数组)
 merge(左半个数组, 右半个数组)
}
```

分治算法的套路就是前面说的三步走：分解 -> 解决 -> 合并：先左右分解，再处理合并，回溯就是在退栈，就相当于后序遍历了。至于 merge 函数，相当于两个有序链表的合并。

LeetCode 有递归专题练习 LeetCode 上有分治算法的专项练习

贪心

贪心算法顾名思义就是只看眼前，并不考虑以后可能造成的影响。可想而知，并不是所有的时候贪心法都能获得最优解。

最常见的贪心有两种。一种是：「将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）处理」。另一种是：「我们每次都取 XXX 中最大/小的东西，

并更新 XXX」，有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护。这两种方式分别对应了离线的情况以及在线的情况。

相关题目：

工作调度 Work Scheduling

修理牛棚 Barn Repair

皇后游戏

二分

以二分搜索为例，它是用来在一个有序数组中查找某一元素的算法。它每次考察数组当前部分的中间元素，如果中间元素刚好是要找的，就结束搜索过程；如果中间元素小于所查找的值，那么左侧的只会更小，不会有所查找的元素，只需要到右侧去找就好了；如果中间元素大于所查找的值，同理，右侧的只会更大而不会有所查找的元素，所以只需要到左侧去找。

在二分搜索过程中，每次都把查询的区间减半，因此对于一个长度为 n 的数组，至多会进行 log(n) 次查找。

一定需要注意的是，这里的有序是广义的有序，如果一个数组中的左侧或者右侧都满足某一种条件，而另一侧都不满足这种条件，也可以看作是一种有序。

二分法把一个寻找极值的问题转化成一个判定的问题（用二分搜索来找这个极值）。类比枚举法，我们当时是枚举答案的可能情况，现在由于单调性，我们不再需要一个个枚举，利用二分的思路，就可以用更优的方法解决「最大值最小」、「最小值最大」。这种解法也成为是「二分答案」，常见于解题报告中。

比如：砍树问题，我们可以在 1 到 1000000000（10 亿）中枚举答案，但是这种朴素写法肯定拿不到满分，因为从 1 跑到 10 亿太耗时间。我们可以对答案进行 1 到 10 亿的二分，其中，每次都对其进行检查可行性（一般都是使用贪心法）。

依照此思想，我们还有三分法等展开算法。

倍增

倍增法，通过字面意思来看就是翻倍。这个方法在很多算法中均有应用，其中最常用的就是 RMQ 问题和求 LCA。

RMQ 是英文 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值。解决 RMQ 问题的主要方法有两种，分别是 ST 表和线段树，具体请参见 ST 表和 线段树内容。

构造

构造针对的问题的答案往往具有某种规律性，使得在问题规模迅速增大的时候，仍然有机会比较容易地得到答案。

这种思想我们接触的比较少，主要体现了数学解题方法啊。比较典型的有：

Luogu P3599 Koishi Loves Construction

Vladik and fractions

AtCoder Grand Contest 032 B

这里我们不再介绍，感兴趣的同学可以进行研究。

前缀和 & 差分

前缀和是一种重要的预处理，能大大降低查询的时间复杂度。我们可以简单理解为「数列的前 n 项的和」。其实前缀和几乎都是基于容斥原理。

比如这道题目：

有 N 个的正整数放到数组 A 里，现在要求一个新的数组 B，新数组的第 i 个数 B[i]是原数组 A 第 0 到第 i 个数的和。

对于这道题，我们有两种做法：

把对数组 A 的累加依次放入数组 B 中。

递推： B[i] = B[i-1] + A[i]

我们看第二种方法采用前缀和的思想，无疑更加优秀。

其他相关题目：

前缀和

前缀和的逆

最大の和

Subsequences Summing to Sevens

更复杂些，可以延伸出：基于 DP 计算高维前缀和，树上前缀和。

最后，差分是一种和前缀和相对的策略。这种策略是求相邻两数的差。相关题目：

树状数组 3： 区间修改，区间查询

地毯

最大流

思想归并

我列举了 9 中算法基本思想，并配上多到典型题目。实际上，读者可以根据自身情况酌情进行了解，在解题外更重要的是体会这些算法思想。比如我留一个小作业：在这三节课中所有讲到的算法中，你能按照这 9 种思想进行归类么？

请动手尝试，我认为我们可以有解不出来的题目，但是对于算法思想的理解至关重要。

总结

到此我们关于算法的三节课就结束了。整体来说，算法需要应试。算法就像弹簧一样，只要你有信心，态度正确，不畏难，一定就可以攻克它。

从今天起，下一个决心，制定一个计划，通过不断练习，提升自己解算法题的能力。当然学习数据结构和算法不仅仅对面试有帮助，对于程序的强健性、稳定性、性能来说，算法虽然只是细节，但却是最重要的一部分之一。比如 AVL 或者 B+ 树，可能除了在学校的大作业，一辈子也不会有机会实现一个出来，但你学会了分析和比较类似算法的能力, 有了搜索树的知识，你才能真正理解为什么 InnoDB 索引要用 B+ 树，你才能明白 like "abc%" 会不会使用索引，而不是人云亦云、知其然不知其所以然。

这一节课我挑选的典型算法都不算困难，但都能体现算法的思想闪光点，适合类推。但实话说，这节课的内容相对零散，算法的思想却是可以归类的，也留给大家一个作业，将上述算法进行思想归类，并在每个归类下再找一道题目进行扩充。这样的学习方法一定会让你有所收获，在全部课程结束后，我也会和大家针对这个「作业」，进行交流，也分享出我的更多算法心得。

点击查看下一节

分析一道「微信」面试题