各位老铁，从本节开始，我们进入排序算法的世界。

对于前端来说，排序算法在应用方面似乎始终不是什么瓶颈——JS 天生地提供了对排序能力的支持，很多时候，我们实现排序只需要这样寥寥数行的代码：

```
arr.sort((a,b) => {
    return a - b
})
```

以某一个排序算法为“引子”，顺藤摸瓜式地盘问，可以问出非常多的东西，这也是排序算法始终热门的一个重要原因——面试官可以通过这种方式在较短的时间里试探出候选人算法能力的扎实程度和知识链路的完整性。因此排序算法在面试中的权重不容小觑。

以面试为导向来看，需要大家着重掌握的排序算法，主要是以下5种：

- 基础排序算法：
  - 冒泡排序
  - 插入排序
  - 选择排序
- 进阶排序算法
  - 归并排序
  - 快速排序

我们的学习安排就按照这个从基础到进阶的次序来。

和以往不同的是，本专题的讲解线索不再是“题目”，而是排序算法本身：针对每一种算法，我都会首先介绍其思想，然后为大家逐步示范一遍真实的排序过程，接着为大家做编码教学。最后，别忘了，排序算法的时间复杂度也是一个不能忽视的考点，“编码复盘”部分我们不见不散。

> 注意：考虑到排序类题目在未经特别声明的情况下，都默认以“从小到大排列”为有序标准。因此下文中所有“有序”的描述指代的都是“从小到大排列”。

冒泡排序

基本思路分析

冒泡排序的过程，就是从第一个元素开始，**重复比较相邻的两个项**，若第一项比第二项更大，则交换两者的位置；反之不动。
每一轮操作，都会将这一轮中最大的元素放置到数组的末尾。假如数组的长度是 n ，那么当我们重复完 n 轮的时候，整个数组就有序了。

真实排序过程演示

下面我们基于冒泡排序的思路，尝试对以下数组进行排序：

```
[5, 3, 2, 4, 1]
```

首先，将第一个元素 5 和它相邻的元素 3 作比较，发现5 比 3 大，故将 5 和 3 交换：

```
[3, 5, 2, 4, 1]
 ↑  ↑
```

将第二个元素 5 和第三个元素 2 作比较，发现 5 比 2大，故将 5 和 2 交换：

```
[3, 2, 5, 4, 1]
    ↑  ↑
```

将第三个元素 5 和第四个元素 4 作比较，发现 5 比 4 大，故将 5 和 4 交换：

```
[3, 2, 4, 5, 1]
       ↑  ↑
```

将第四个元素 5 和第五个元素 1 作比较，发现 5 比 1 大，故将 5 和 1 交换：

```
[3, 2, 4, 1, 5]
          ↑  ↑
```

至此我们就完成了一轮排序，此时，五个数中最大的数字 5 仿佛气泡浮出水面一样，被"冒"到了数组顶部。这也是冒泡排序得名的原因。

重复上面的操作，我们继续从第一个元素开始看起。比较 3 和 2，发现 3 比 2 大，交换两者：

```
[2, 3, 4, 1, 5]
 ↑  ↑
```

比较 3 和 4，发现 3 比 4 小，符合从小到大排列的原则，故保持不动：

```
[2, 3, 4, 1, 5]
    ↑  ↑
```

比较 4 和 1，发现 4 比 1 大，交换两者：

```
[2, 3, 1, 4, 5]
       ↑  ↑
```

比较 4 和 5，发现 4 比 5 小，符合从小到大排列的原则，故保持不动：

```
[2, 3, 1, 4, 5]
          ↑  ↑
```

以上我们完成了第二轮排序，至此，五个数中第二大的数字 4 也被"冒"到了数组相对靠后的位置。
沿着这个思路往下走，仍然是从第一个元素开始，比较 2 和 3。发现 2 比 3 小，符合排序原则，故保持不动：

```
[2, 3, 1, 4, 5]
 ↑  ↑
```

接着走下去，比较 3 和 1，发现 3 比 1 大，交换两者：

```
[2, 1, 3, 4, 5]
    ↑  ↑
```

比较 3 和 4，发现 3 比 4 小，符合排序原则，故保持不动：

```
[2, 1, 3, 4, 5]
       ↑  ↑
```

比较 4 和 5，发现 4 比 5 小，符合排序原则，故保持不动：

```
[2, 1, 3, 4, 5]
          ↑  ↑
```

以上我们完成了第二轮排序，至此，五个数中第三大的数字 3 被"冒"到了倒数第三个的位置。
继续我们的循环，从当前的第一个元素 2 开始，比较 2 和相邻元素 1，发现 2 比 1 大，交换两者：

```
[1, 2, 3, 4, 5]
 ↑  ↑
```

接下来仍然会对剩余的元素进行相邻元素比较，但由于不再发生交换，所以我们这里简写一下每一步对应的相邻元素关系：

```
[1, 2, 3, 4, 5]
    ↑  ↑
```

```
[1, 2, 3, 4, 5]
       ↑  ↑
```

```
[1, 2, 3, 4, 5]
       ↑  ↑
```

```
[1, 2, 3, 4, 5]
          ↑  ↑
```

经过第四轮冒泡，整个数组已经完全达到了有序状态。不过，冒泡排序的逻辑并不会因为数组有序就立刻停下来——"从头到尾遍历数组，对比+交换每两个相邻元素"这套逻辑到底要执行多少次，按照我们目前的基本思路来看，是完全由数组中元素的个数来决定的：每一次从头到尾的遍历都只能定位到一个元素的位置，因此元素有多少个，总的循环就要执行多少轮。
在这个例子中，总的元素有 5 个，因此理论上来说还有一轮从头到尾的循环要走。
相信大家已经隐约感觉到了哪里不对，不过没关系，掌握了基本思路，优化啥的都好说。我们先按照这个思路来编码：

基本冒泡思路编码实现

```
function bubbleSort(arr) {
    // 缓存数组长度
    const len = arr.length
    // 外层循环用于控制从头到尾的比较+交换到底有多少轮
    for(let i=0;i<len;i++) {
```

```
            // 内层循环用于完成每一轮遍历过程中的重复比较+交换
            for(let j=0;j<len-1;j++) {
                // 若相邻元素前面的数比后面的大
                if(arr[j] > arr[j+1]) {
                    // 交换两者
                    [arr[j], arr[j+1]] = [arr[j+1], arr[j]]
                }
            }
        }
        // 返回数组
        return arr
    }
```

基本冒泡思路的改进

在上面的示例中，我们已经初步分析出了这样一个结论：在冒泡排序的过程中，有一些"动作"是不太必要的。比如数组在已经有序的情况下，为什么还要强行再从头到尾再对数组做一次遍历？

这背后的根本原因是，我们忽略了这样一个事实：随着外层循环的进行，数组尾部的元素会渐渐变得有序——当我们走完第1轮循环的时候，最大的元素被排到了数组末尾；走完第2轮循环的时候，第2大的元素被排到了数组倒数第2位；走完第3轮循环的时候，第3大的元素被排到了数组倒数第3位……以此类推，走完第 n 轮循环的时候，数组的后 `n` 个元素就已经是有序的。

楼上基本冒泡思路的问题在于，没有区别处理这一部分已经有序的元素，而是把它和未排序的部分做了无差别的处理，进而造成了许多不必要的比较。

为了避免这些冗余的比较动作，我们需要规避掉数组中的后 `n` 个元素，对应的代码可以这样写：

改进版冒泡排序的编码实现

```
function betterBubbleSort(arr) {
    const len = arr.length
    for(let i=0;i<len;i++) {
        // 注意差别在这行，我们对内层循环的范围作了限制
        for(let j=0;j<len-1-i;j++) {
            if(arr[j] > arr[j+1]) {
                [arr[j], arr[j+1]] = [arr[j+1], arr[j]]
            }
        }
```

```
    }
    return arr
}
```

面向“最好情况”的进一步改进

很多同学反映说，在不少教材里都看到了“冒泡排序时间复杂度在最好情况下是 O(n)”这种说法，但是横看竖看，包括楼上示例在内的各种冒泡排序主流模板似乎都无法帮助我们推导出 `O(n)` 这个结果。
实际上，你的想法是对的，冒泡排序最常见的写法（也就是楼上的编码示例）在最好情况下对应的时间复杂度确实不是 `O(n)` ，而是 `O(n^2)` 。
那么是 `O(n)` 这个说法错了吗？其实也不错，因为冒泡排序通过进一步的改进，确实是可以做到最好情况下 `O(n)` 复杂度的，这里我先把代码给大家写出来（注意解析在注释里）：

```
function betterBubbleSort(arr) {
    const len = arr.length

    for(let i=0;i<len;i++) {
        // 区别在这里，我们加了一个标志位
        let flag = false
        for(let j=0;j<len-1-i;j++) {
            if(arr[j] > arr[j+1]) {
                [arr[j], arr[j+1]] = [arr[j+1], arr[j]]
                // 只要发生了一次交换，就修改标志位
                flag = true
            }
        }

        // 若一次交换也没发生，则说明数组有序，直接放过
        if(flag == false)  return arr;
    }
    return arr
}
```

标志位可以帮助我们在第一次冒泡的时候就定位到数组是否完全有序，进而节省掉不必要的判断逻辑，将最好情况下的时间复杂度定向优化为 `O(n)` 。

> 注意，以上几种写法中，改进后的版本可以视为标准的冒泡排序。但笔者更推荐大家把两个思路都记住，尤其是要理解从基本思路到改进思路的演进过程和优化依据。

编码复盘——冒泡排序的时间复杂度

我们分最好、最坏和平均来看：

- **最好时间复杂度**：它对应的是数组本身有序这种情况。在这种情况下，我们只需要作比较（n-1 次），而不需要做交换。时间复杂度为 **O(n)**
- **最坏时间复杂度**： 它对应的是数组完全逆序这种情况。在这种情况下，每一轮内层循环都要执行，重复的总次数是 n(n-1)/2 次，因此时间复杂度是 **O(n^2)**
- **平均时间复杂度**：这个东西比较难搞，它涉及到一些概率论的知识。实际面试的时候也不会有面试官摁着你让你算这个，这里记住平均时间复杂度是 **O(n^2)** 即可。

> 对于每一种排序算法的时间复杂度，大家对计算依据有了解即可，重点在于记忆。面试的时候不要靠现场推导，要靠直觉+条件反射。

选择排序

思路分析

选择排序的关键字是“**最小值**”：循环遍历数组，每次都找出当前范围内的最小值，把它放在当前范围的头部；然后缩小排序范围，继续重复以上操作，直至数组完全有序为止。

真实排序过程演示

下面我们尝试基于选择排序的思路，对如下数组进行排序：

```
[5, 3, 2, 4, 1]
```

首先，索引范围为 [0, n-1] 也即 [0,4] 之间的元素进行的遍历（两个箭头分别对应当前范围的起点和终点）：

```
[5, 3, 2, 4, 1]
 ↑              ↑
```

得出整个数组的最小值为 `1` 。因此把 `1` 锁定在当前范围的头部，也就是和 `5` 进行交换：

```
[1, 3, 2, 4, 5]
```

交换后，数组的第一个元素值就明确了。接下来需要排序的是 [1, 4] 这个索引区间：

```
[1, 3, 2, 4, 5]
    ↑        ↑
```

遍历这个区间，找出区间内最小值为 `2` 。因此区间头部的元素锁定为 `2` ，也就是把 `2` 和 `3` 交换。相应地，将需要排序的区间范围的起点再次后移一位，此时区间为 [2, 4]：

```
[1, 2, 3, 4, 5]
       ↑     ↑
```

遍历 [2,4] 区间，得到最小值为 `3` 。 `3` 本来就在当前区间的头部，因此不需要做额外的交换。以此类推， `4` 会被定位为索引区间 [3,4] 上的最小值，仍然是不需要额外交换的。

基于这个思路，我们来写代码：

编码示范

```
function selectSort(arr)  {
  // 缓存数组长度
  const len = arr.length
  // 定义 minIndex，缓存当前区间最小值的索引，注意是索引
  let minIndex
  // i 是当前排序区间的起点
  for(let i = 0; i < len - 1; i++) {
    // 初始化 minIndex 为当前区间第一个元素
    minIndex = i
    // i、j分别定义当前区间的上下界，i是左边界，j是右边界
    for(let j = i; j < len; j++) {
      // 若 j 处的数据项比当前最小值还要小，则更新最小值索引为 j
      if(arr[j] < arr[minIndex]) {
        minIndex = j
      }
    }
    // 如果 minIndex 对应元素不是目前的头部元素，则交换两者
    if(minIndex !== i) {
      [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]]
    }
```

```
    }
    return arr
  }
```

编码复盘——选择排序的时间复杂度

在时间复杂度这方面，选择排序没有那么多弯弯绕绕：最好情况也好，最坏情况也罢，两者之间的区别仅仅在于元素交换的次数不同，**但都是要走内层循环作比较的**。因此选择排序的三个时间复杂度都对应两层循环消耗的时间量级： O(n^2)。

插入排序

思路分析

插入排序的核心思想是“找到元素在它前面那个序列中的正确位置”。
具体来说，插入排序所有的操作都基于一个这样的前提：当前元素前面的序列是有序的。基于这个前提，从后往前去寻找当前元素在前面那个序列里的正确位置。

真实排序过程演示

下面我们尝试基于插入排序的思路，对如下数组进行排序：

```
[5, 3, 2, 4, 1]
```

首先，单个数字一定有序，因此数组首位的这个 `5` 可以看做是一个有序序列。在这样的前提下， 我们就可以选中第二个元素 `3` 作为当前元素，思考它和前面那个序列 `[5]` 之间的关系。很明显， `3` 比 `5` 小，注意这里按照插入排序的原则，靠前的较大数字要为靠后的较小数字腾出位置：

```
[暂时空出, 5, 2, 4, 1]
当前元素 3
```

再往前看，发现没有更小的元素可以作比较了。那么现在空出的这个位置就是当前元素 `3` 应该待的地方：

```
[3, 5, 2, 4, 1]
```

以上我们就完成了一轮插入。这一轮插入结束后，大家会发现，有序数组 [5] 现在变成了有序数组 [3, 5]——这正是插入排序的用意所在，**通过正确地定位当前元素在有序序列里的位置、不断扩大有序数组的范围，最终达到完全排序的目的。**

沿着这个思路，继续往下走，当前元素变成了紧跟[3, 5] 这个有序序列的 2 。对比 2 和 5 的大小，发现 2 比 5 小。按照插入排序的原则， 5 要往后挪，给较小元素空出一个位置：

```
[3, 暂时空出, 5, 4, 1]
当前元素 2
```

接着继续向前对比，遇到了 3 。对比 3 和 2 的大小，发现 3 比 2 小。按照插入排序的原则， 3 要往后挪，给较小元素空出一个位置：

```
[暂时空出, 3, 5, 4, 1]
当前元素 2
```

此时 2 前面的有序序列已经被对比完毕了。我们把 2 放到最终空出来的那个属于它的空位里去：

```
[2, 3, 5, 4, 1]
```

以上我们完成了第二轮插入。这一轮插入结束后，有序数组 [3, 5] 现在变成了有序数组 [2, 3, 5]。

继续往下走，紧跟有序数组 [2, 3, 5] 的元素是 4 。仍然是从后往前，首先对比 4 和 5 的大小，发现 4 比 5 小，那么 5 就要为更小的元素空出一个位置：

```
[2, 3, 暂时空出, 5, 1]
当前元素 4
```

向前对比，遇到了 3 。因为 4 比 3 大，符合从小到大的排序原则；同时已知当前这个序列是有序的， 3 前面的数字一定都比 3 小，再继续向前查找就没有意义了。因此当前空出的这个坑就是 4 应该待的地方：

```
[2, 3, 4, 5, 1]
```

以此类推，最后一个元素 1 会被拱到 [2, 3, 4, 5] 这个序列的头部去，最终数组得以完全排序：

```
[1, 2, 3, 4, 5]
```

分析至此，再来帮大家复习一遍插入排序里的几个关键点：

- 当前元素前面的那个序列是有序的

- “正确的位置”如何定义——所有在当前元素前面的数都不大于它，所有在当前元素后面的数都不小于它
- 在有序序列里定位元素位置的时候，是从后往前定位的。只要发现一个比当前元素大的值，就需要为当前元素腾出一个新的坑位。

基于这个思路，我们来写代码：

编码实现

```
function insertSort(arr) {
  // 缓存数组长度
  const len = arr.length
  // temp 用来保存当前需要插入的元素
  let temp
  // i用于标识每次被插入的元素的索引
  for(let i = 1;i < len; i++) {
    // j用于帮助 temp 寻找自己应该有的定位
    let j = i
    temp = arr[i]
    // 判断 j 前面一个元素是否比 temp 大
    while(j > 0 && arr[j-1] > temp) {
      // 如果是，则将 j 前面的一个元素后移一位，为 temp 让出位置
      arr[j] = arr[j-1]
      j--
    }
    // 循环让位，最后得到的 j 就是 temp 的正确索引
    arr[j] = temp
  }
  return arr
}
```

编码复盘——插入排序的时间复杂度

- **最好时间复杂度**：它对应的数组本身就有序这种情况。此时内层循环只走一次，整体复杂度取决于外层循环，时间复杂度就是一层循环对应的 **O(n)**。
- **最坏时间复杂度**：它对应的是数组完全逆序这种情况。此时内层循环每次都要移动有序序列里的所有元素，因此时间复杂度对应的就是两层循环的 O(n^2)
- **平均时间复杂度**：O(n^2)

小结

所谓基础排序算法，普遍符合两个特征：

1. 易于理解，上手迅速
2. 时间效率差

楼上的三个算法完美地诠释了这两个特征。对于基础排序算法，大家不要胡思乱想，你的目标就是默写，面试的时候考的最多的也是默写。
那么在此基础上，排序效率如何提升、排序算法如何与进阶的算法思想相结合？这就是我们下一节要讨论的问题了。