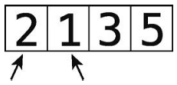
4 运用大 O 来给代码提速

4.1 冒泡排序

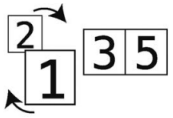
1. 冒泡排序图解

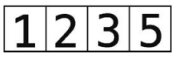
冒泡排序是一种很基本的排序算法，步骤如下。

(1) 指向数组中两个相邻的元素（最开始是数组的头两个元素），比较它们的大小。



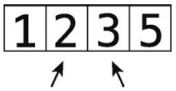
(2) 如果它们的顺序错了（即左边的值大于右边），就互换位置。





如果顺序已经是正确的，那这一步就什么都不用做。

(3) 将两个指针右移一格。



重复第(1)步和第(2)步，直至指针到达数组末尾。

(4)重复第(1)至(3)步，直至从头到尾都无须再做交换，这时数组就排好序了。

这里被重复的第(1)至(3)步是一个**轮回**，也就是说，这个算法的主要步骤被“轮回”执行，直到整个数组的顺序正确。

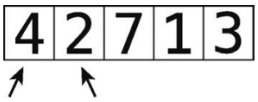
4.2 冒泡排序实战

1. 一个冒泡排序的例子 [4,2,7,1,3]
2. 第一次轮回

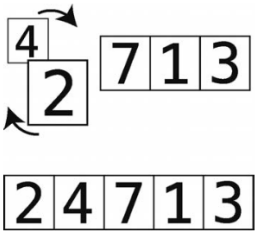
开始第 1 次轮回。

数组一开始如下图所示。

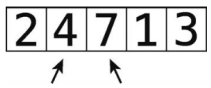
第 1 步：首先，比较 4 和 2。如图可见它们的顺序是错的。



第 2 步：交换它们的位置。

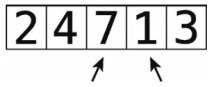


第 3 步：比较 4 和 7。

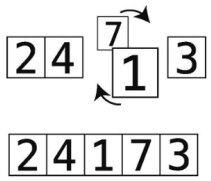


它们的顺序正确，所以不用做什么交换。

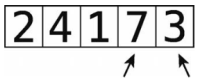
第 4 步：比较 7 和 1。



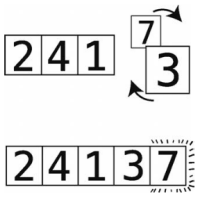
第 5 步：顺序错误，于是进行交换。



第 6 步：比较 7 和 3。



第 7 步：顺序错误，于是进行交换。



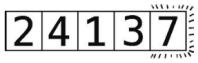
因为我们一直把较大的元素换到右边，所以现在最右侧的 7 正处于其正确位置上。我将那个格子用虚线圈起来了。

这也正是此种算法名为**冒泡排序**的原因：每一次轮回过后，未排序的值中最大的那个都会“冒”到正确的位置上。

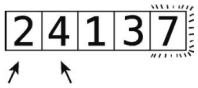
3. 第二次轮回

下面来第 2 次轮回。

此时 7 已经在正确的位置上了。

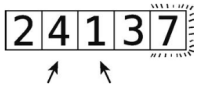


第 8 步：从比较 2 和 4 开始。

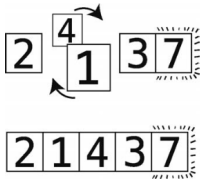


它们已经按顺序排好了，所以直接进行下一步。

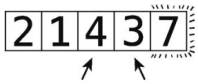
第 9 步：比较 4 和 1。



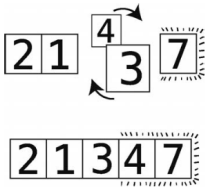
第 10 步：它们的顺序错误，于是交换。



第 11 步：比较 4 和 3。



第 12 步：顺序错误，进行交换。

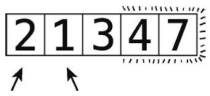


因为 7 已经在上一次轮回里排好了，所以无须比较 4 和 7。此外，4 移到了正确的位置，本次轮回结束。因为这次轮回也做了不止一次的交换，所以得继续轮回。

4. 第三次轮回

下面来第 3 次轮回。

第 13 步：比较 2 和 1。



第 14 步：顺序错误，进行交换。

第 15 步：比较 2 和 3。

顺序正确，不用交换。

这时 3 也“冒”到其正确位置了。因为这次轮回做了不止一次的交换，所以还要继续。

5. 第四次轮回

于是开始第 4 次轮回。

第 16 步：比较 1 和 2。

顺序正确，不用交换。而且剩下的元素也都排好序了，轮回结束。

因为刚才的轮回没有任何交换，可知整个数组都已排好序。

4.3 冒泡排序的实现

```
def bubble_sort(list):
    unsorted_until_index = len(list) - 1
    sorted = False

    while not sorted:
        sorted = True
        for i in range(unsorted_until_index):
            if list[i] > list[i+1]:
                sorted = False
                list[i], list[i+1] = list[i+1], list[i]
        unsorted_until_index = unsorted_until_index - 1

list = [65, 55, 45, 35, 25, 15, 10]
bubble_sort(list)
print list
```

4.4 冒泡排序的效率

1. 冒泡排序的执行步骤可分为两种。**比较**：比较两个数看哪个更大。**交换**：交换两个数的位置以使它们按顺序排列。
2. 以5个元素为例

回顾之前那个 5 个元素的数组，你会发现在第 1 次轮回我们为 4 对元素进行了 4 次比较。

到了第 2 次轮回，则只做了 3 次比较。这是因为第 1 次轮回已经确定了最后一个格子的元素，所以不用再比较最后两个元素了。

第 3 次轮回，只比较 2 次；第 4 次，只比较 1 次。

算起来就是：

$4 + 3 + 2 + 1 = 10$ 次比较。

推广到 N 个元素，需要

$(N-1) + (N-2) + (N-3) + \cdots + 1$ 次比较。

分析过比较之后，再来看看**交换**。

如果数组不只是随机打乱，而是完全反序，在这种最坏的情况下，每次比较过后都得进行一次交换。因此 10 次比较加 10 次交换，总共 20 步。

现在把两种步骤放在一起来看。一个含有 10 个元素的数组，需要：

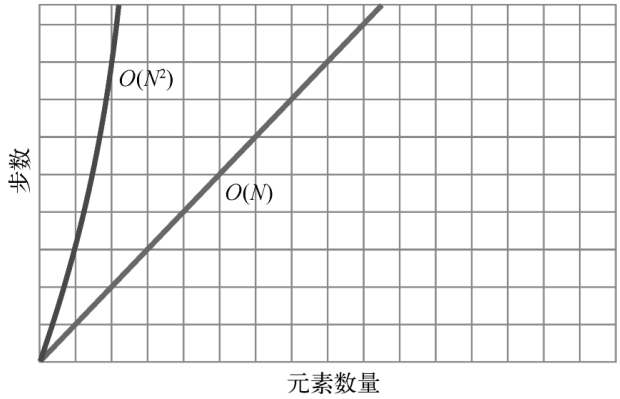
$9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45$ 次比较，以及 45 次交换，共 90 步。

20 个元素的话，就是：

$19 + 18 + 17 + 16 + 15 + 14 + 13 + 12 + 11 + 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 190$ 次比较，以及 190 次交换，共 380 步。

3. 冒泡排序 O(N^2)

$O(N^2)$算法是比较低效的，随着数据量变多，其步数也剧增，如下图所示。



4. O(N^2) **二次时间**

4.5 二次问题

1. 检查数组中是否有重复(hasDuplicateValue)

```
function hasDuplicateValue(array) {
    for(var i = 0; i < array.length; i++) {
        for(var j = 0; j < array.length; j++) {
            if(i !== j && array[i] == array[j]) {
                return true;
            }
        }
    }
    return false;
}
```

2. 传入一个含有 N个元素的数组，最坏情况下需要多少步才能完成。
    1. 该函数只有一种步骤，就是**比较**。重复地比较 i 和 j 所指的值，看它们是否相等。
    2. 复值。最坏的情况就是没有重复，这将使我们跑遍内外两层循环，比较完所有 i、j 组合，才返回 false。
3. 由此可知N个元素需要比较N^2次。因为外层循环需要 N步来遍历数组，而这里的每 1 步会发起内层循环去用 N步遍历数组。所以是O(N^2)算法。
4. 加入步数跟踪的代码

```
function hasDuplicateValue(array) {
    var steps = 0;
    for(var i = 0; i < array.length; i++) {
        for(var j = 0; j < array.length; j++) {
            steps++;
            if(i !== j && array[i] == array[j]) {
                return true;
            }
        }
    }
    console.log(steps);
    return false;
}
```

4.6 线性解决

1. hasDuplicateValue 的另一种实现 （没有嵌套循环）

```
function hasDuplicateValue(array) {
    var existingNumbers = [];
    for(var i = 0; i < array.length; i++) {
        if(existingNumbers[array[i]] === undefined) {
            existingNumbers[array[i]] = 1;
        } else {
            return true;
        }
    }
    return false;
}
```

2.