NC78 反转链表

输入一个链表，反转链表后，输出新链表的表头。

```python
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    # 返回ListNode
    def ReverseList(self, pHead):
        # write code here
        if pHead is None or pHead.next is None:
            return pHead

        previous, current, next = None, pHead, None
        while current is not None:
            next = current.next
            current.next = previous
            previous = current
            current = next

        return previous
```

NC140 排序

给定一个数组，请你编写一个函数，返回该数组排序后的形式。

排序算法	时间复杂度	是否基于比较
冒泡、插入、选择	$O(n^2)$	✅
快排、归并	$O(n\log n)$	✅
桶、计数、基数	$O(n)$	❌

如何分析一个“排序算法”？

执行效率

1. 最好情况、最坏情况、平均情况时间复杂度
2. 时间复杂度的系数、常数、低阶
3. 比较次数和交换（或移动）次数

内存消耗

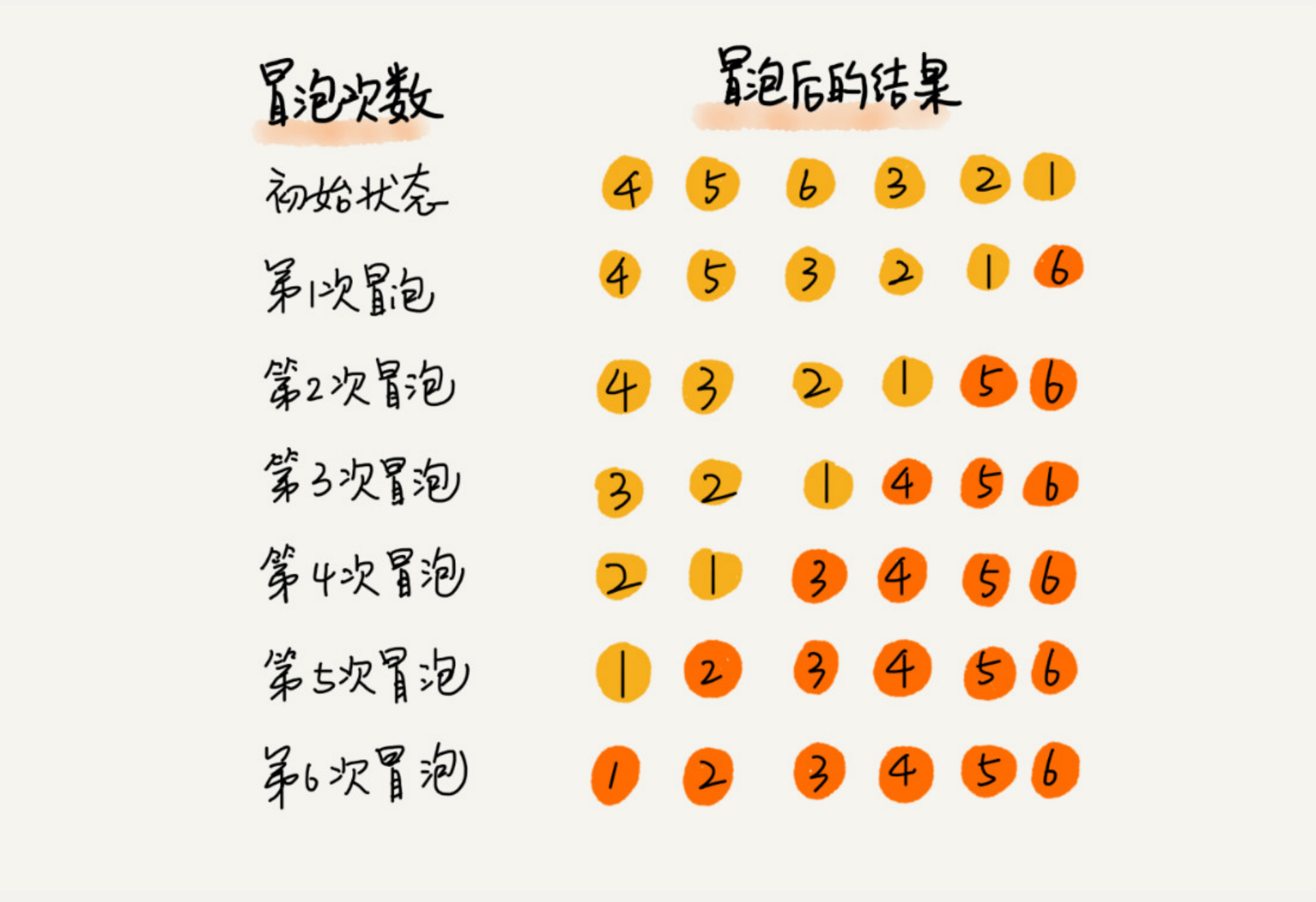
算法的内存消耗可以通过空间复杂度来衡量，排序算法也不例外。不过，针对排序算法的空间复杂度，我们还引入了一个新的概念，**原地排序（Sorted in place）**。原地排序算法，就是特指空间复杂度是 $O(1)$ 的排序算法。

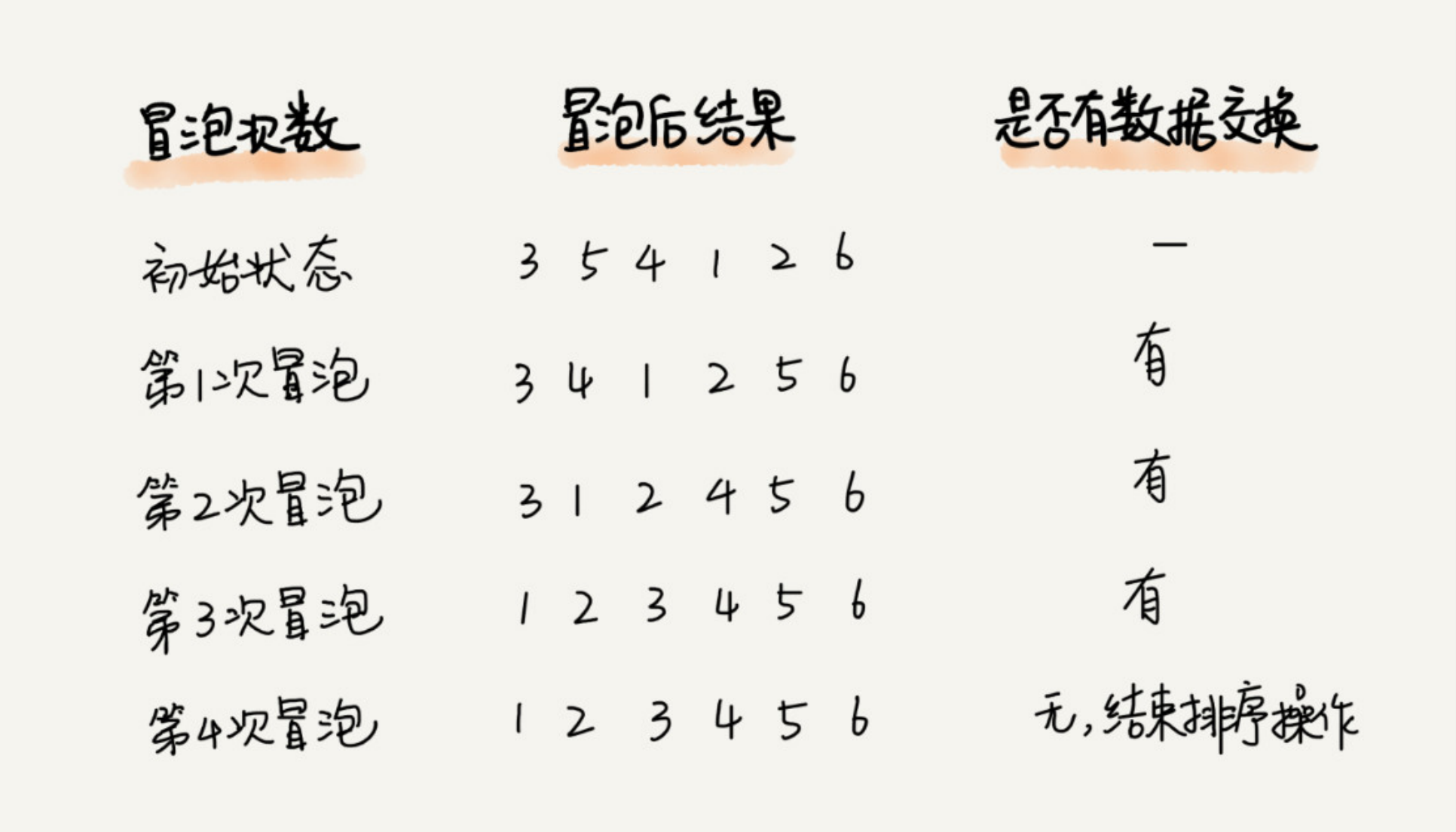
稳定性

仅仅用执行效率和内存消耗来衡量排序算法的好坏是不够的。针对排序算法，我们还有一个重要的度量指标，**稳定性**。这个概念是说，如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

冒泡排序（Bubble Sort）

冒泡排序只会操作相邻的两个数据。每次冒泡操作都会对相邻的两个元素进行比较，看是否满足大小关系要求。如果不满足就让它俩互换。一次冒泡会让至少一个元素移动到它应该在的位置，重复 n 次，就完成了 n 个数据的排序工作。





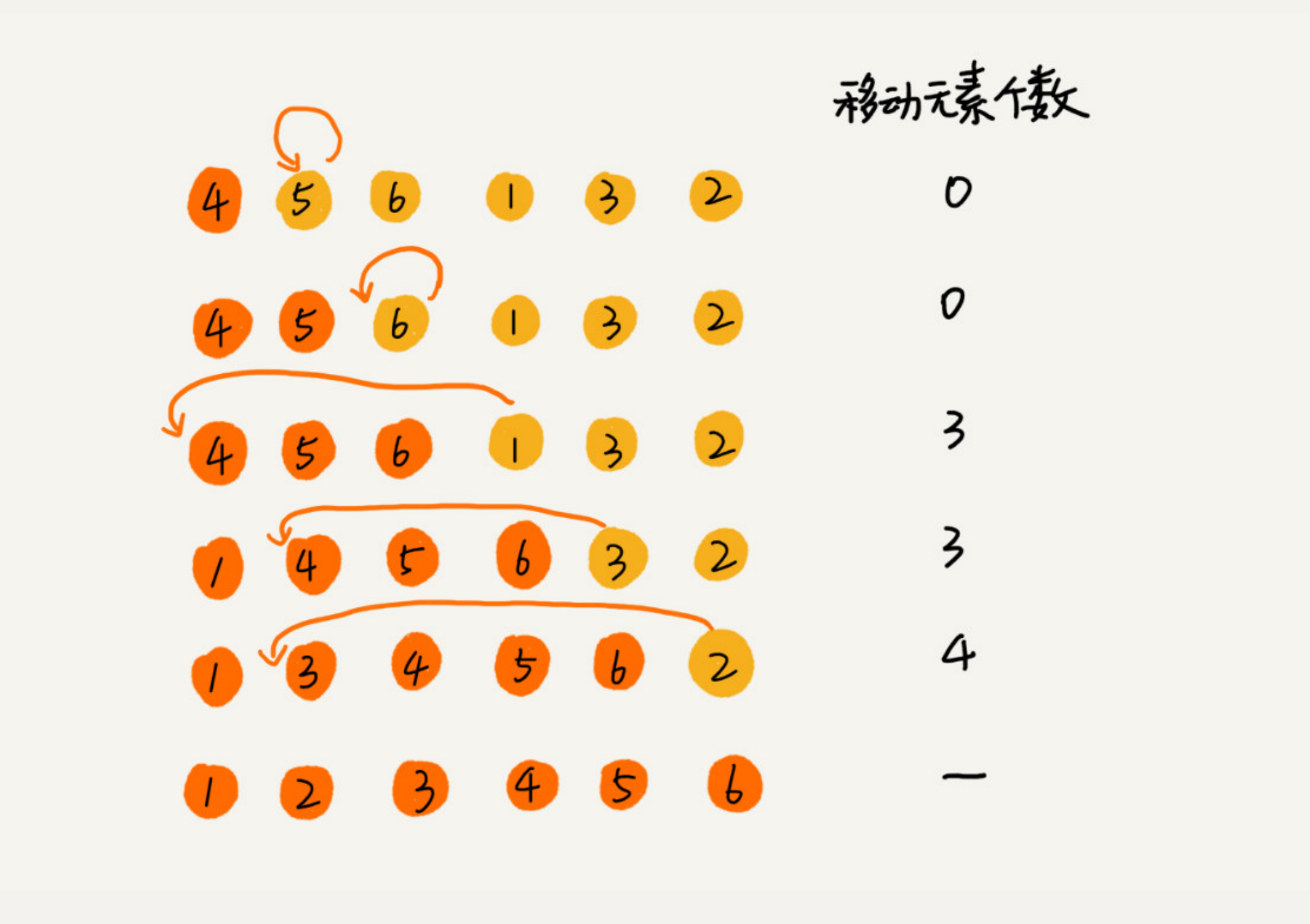
冒泡次数	冒泡后结果	是否有数据交换
初始状态	3 5 4 1 2 6	—
第1次冒泡	3 4 1 2 5 6	有
第2次冒泡	3 1 2 4 5 6	有
第3次冒泡	1 2 3 4 5 6	有
第4次冒泡	1 2 3 4 5 6	无，结束排序操作

```python
# 冒泡排序
def bubble_sort(a: List[int]):
    length = len(a)
    if length <= 1:
        return

    for i in range(length):
        made_swap = False
        for j in range(length - i - 1):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                made_swap = True
        if not made_swap:
            break
```

插入排序（Insertion Sort）

首先，我们将数组中的数据分为两个区间，**已排序区间和未排序区间**。初始已排序区间只有一个元素，就是数组的第一个元素。插入算法的核心思想是取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。重复这个过程，直到未排序区间中元素为空，算法结束。



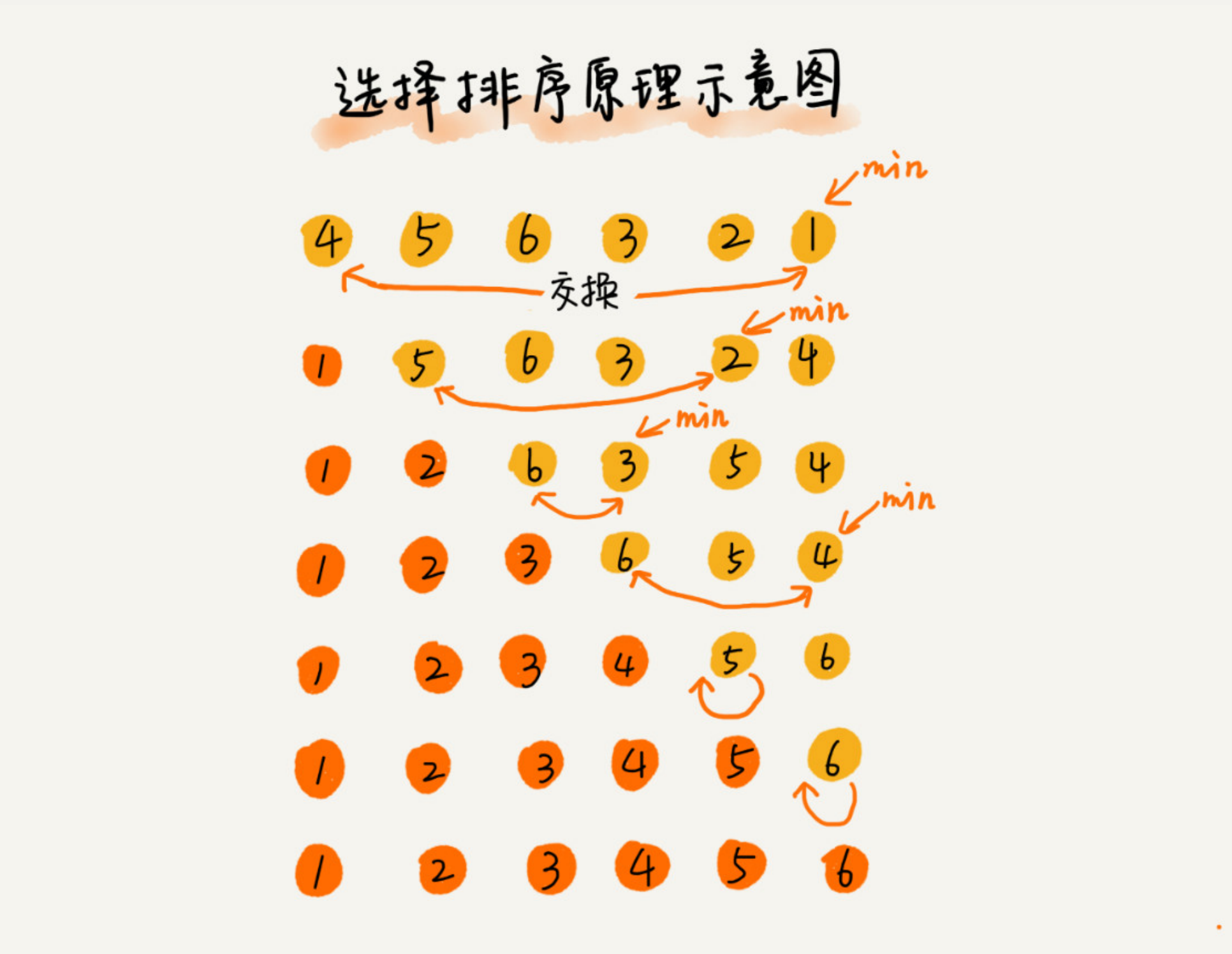
插入排序也包含两种操作，一种是**元素的比较**，一种是**元素的移动**。当我们需要将一个数据 a 插入到已排序区间时，需要拿 a 与已排序区间的元素依次比较大小，找到合适的插入位置。找到插入点之后，我们还需要将插入点之后的元素顺序往后移动一位，这样才能腾出位置给元素 a 插入。

```python
# 插入排序
def insertion_sort(a: List[int]):
    length = len(a)
    if length <= 1:
        return

    for i in range(1, length):
        value = a[i]
        j = i - 1
        while j >= 0 and a[j] > value:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = value
```

选择排序（Selection Sort）

选择排序算法的实现思路有点类似插入排序，也分已排序区间和未排序区间。但是选择排序每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾。



插入排序 vs 冒泡排序

插入排序和冒泡排序的时间复杂度相同，都是$O(n^2)$，在实际的软件开发里，为什么我们更倾向于使用插入排序算法而不是冒泡排序算法呢？

冒泡排序不管怎么优化，元素交换的次数是一个固定值，是原始数据的逆序度。插入排序是同样的，不管怎么优化，元素移动的次数也等于原始数据的逆序度。

但是，从代码实现上来看，冒泡排序的数据交换要比插入排序的数据移动要复杂，冒泡排序需要 3 个赋值操作，而插入排序只需要 1 个。

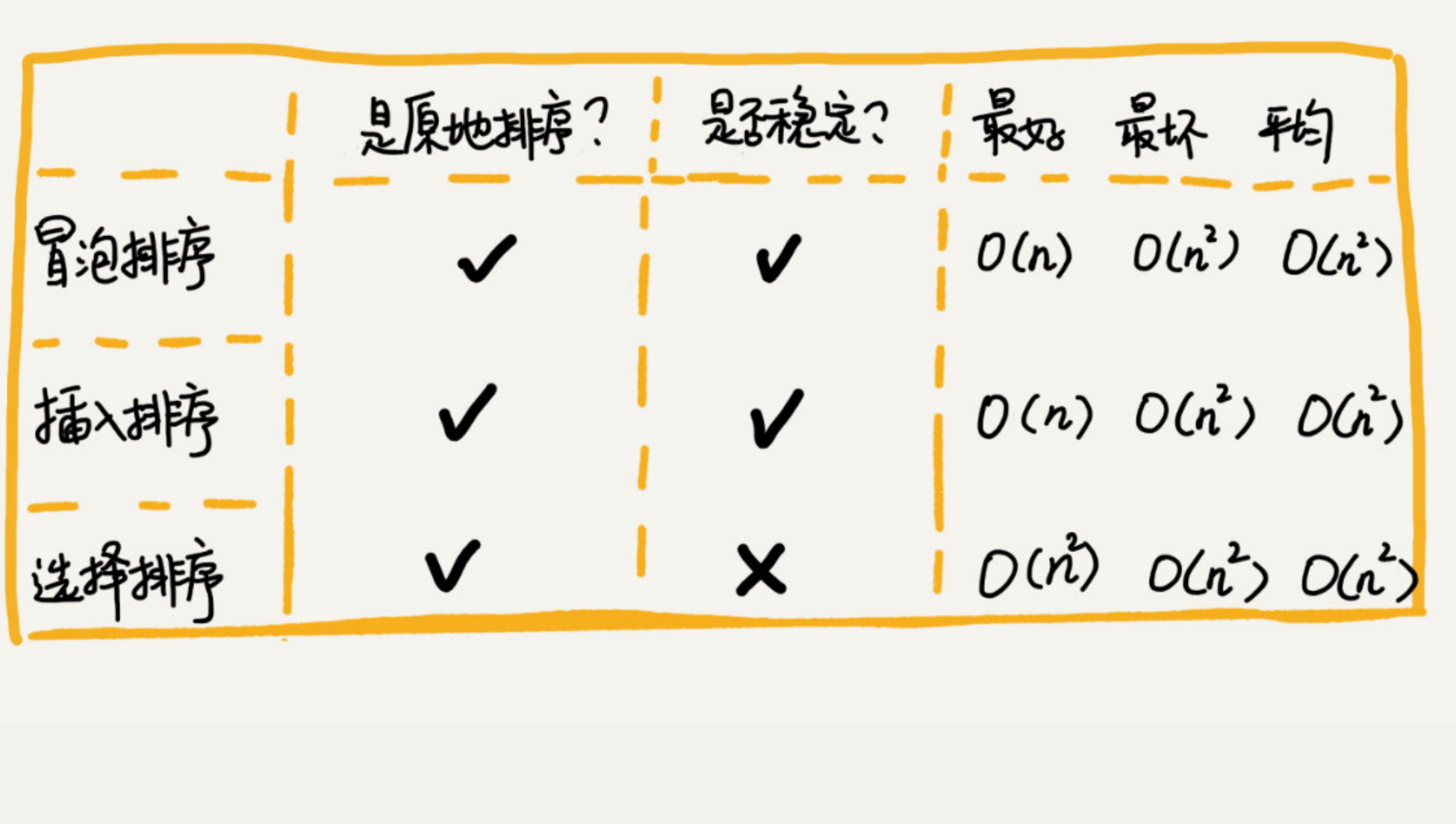
```
冒泡排序中数据的交换操作：
  if (a[j] > a[j+1]) { // 交换
    int tmp = a[j];
    a[j] = a[j+1];
    a[j+1] = tmp;
    flag = true;
  }

插入排序中数据的移动操作：
  if (a[j] > value) {
    a[j+1] = a[j]; // 数据移动
  } else {
    break;
  }
```

所以，虽然冒泡排序和插入排序在时间复杂度上是一样的，都是 $O(n^2)$，但是如果我们希望把性能优化做到极致，那肯定首选插入排序。插入排序的算法思路也有很大的优化空间，我们只是讲了最基础的一种。

小结

这三种时间复杂度为 $O(n^2)$的排序算法中，**冒泡排序和选择排序**可能就纯粹停留在理论的层面了，学习的目的也只是为了开拓思维，实际开发中应用并不多，但是插入排序还是挺有用的，有些编程语言中的排序函数的实现原理会用到插入排序算法。



	是原地排序？	是否稳定？	最好	最坏	平均
冒泡排序	✓	✓	$O(n)$	$O(n^2)$	$O(n^2)$
插入排序	✓	✓	$O(n)$	$O(n^2)$	$O(n^2)$
选择排序	✓	✗	$O(n^2)$	$O(n^2)$	$O(n^2)$