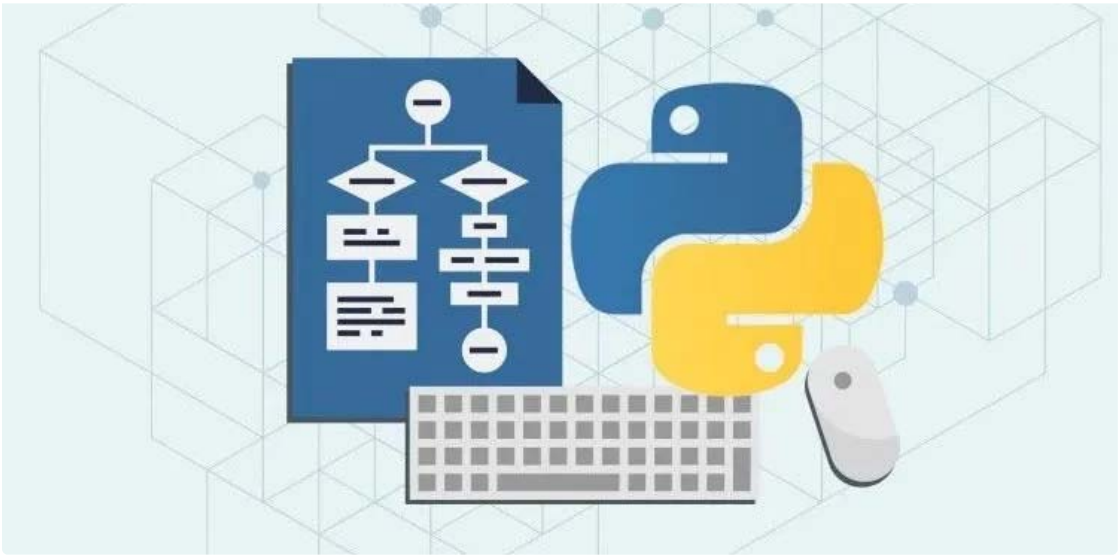
用Python手写五大经典排序算法，看完这篇终于懂了！

菜鸟学Python 5月16日

以下文章来源于Python数据科学 ， 作者Python数据科学

Python数据科学
以Python为核心语言，专攻于「数据科学」领域，文章涵盖数据分析，数据挖掘，机器…



算法作为程序员的必修课，是每位程序员必须掌握的基础。作为Python忠实爱好者，本篇东哥将通过**Python来手撕5大经典排序算法**，结合例图剖析内部实现逻辑，对比每种算法各自的优缺点和应用点。相信我，耐心看完绝对有收获。

前戏准备

大家都知道从理论上讲，我们一般会使用**大O表示法**测量算法的**运行时复杂度。"大O表示法"表示程序的执行时间或占用空间随数据规模的增长趋势。**

但为了测算具体的时间，本篇将使用`timeit`模块来衡量实现的运行时间。下面自己写一个

对算法测试时间的函数。

```
from random import randint
from timeit import repeat

def run_sorting_algorithm(algorithm, array):
    # 调用特定的算法对提供的数组执行。
    # 如果不是内置sort()函数，那就只引入算法函数。
    setup_code = f"from __main__ import {algorithm}" \
        if algorithm != "sorted" else ""

    stmt = f"{algorithm}({array})"

    # 十次执行代码，并返回以秒为单位的时间
    times = repeat(setup=setup_code, stmt=stmt, repeat=3, number=10)

    # 最后，显示算法的名称和运行所需的最短时间
    print(f"Algorithm: {algorithm}. Minimum execution time: {min(times)}")
```

这里用到了一个骚操作，通过**f-strings魔术方法**导入了算法名称，不懂的可以自行查看使用方法。

> **注意：应该找到算法每次运行的平均时间，而不是选择单个最短时间。**由于系统同时运行其他进程，因此时间测量是受影响的。最短的时间肯定是影响最小的，是这样才使其成为算法时间最短的。

Python中的冒泡排序算法

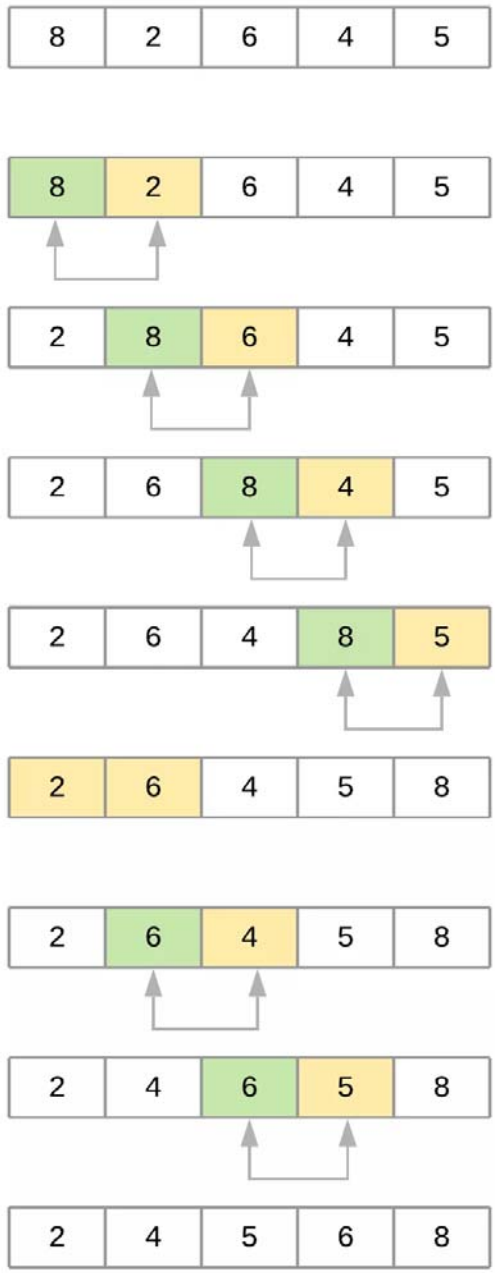
冒泡排序是最直接的排序算法之一。它的名称来自算法的工作方式：每经过一次新的遍历，列表中最大的元素就会“冒泡”至正确位置。

在Python中实现冒泡排序

这是Python中冒泡排序算法的实现：

```
def bubble_sort(array):
    n = len(array)

    for i in range(n):
        # 创建一个标识，当没有可以排序的时候就使函数终止。
        already_sorted = True

        # 从头开始逐个比较相邻元素，每一次循环的总次数减1，
        # 因为每次循环一次，最后面元素的排序就确定一个。
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                # 如果此时的元素大于相邻后一个元素，那么交换。
                array[j], array[j + 1] = array[j + 1], array[j]

                # 如果有了交换，设置already_sorted标志为False算法不会提前停止
                already_sorted = False

        # 如果最后一轮没有交换，数据已经排序完毕，退出
        if already_sorted:
            break

    return array
```

为了正确分析算法的工作原理，看下这个列表[8， 2， 6， 4， 5]。假设使用bubble_sort()排序，下图说明了算法每次迭代时数组的实际换件情况：



冒泡排序过程

测算冒泡算法的大O运行复杂度

冒泡排序的实现由两个嵌套for循环组成，其中算法先执行*n-1*个比较，然后进行*n-2*个比较，依此类推，直到完成最终比较。因此，总的比较次数为 *(N - 1) + (N - 2) + (N - 3) + ... + 2 + 1 = N (N-1) / 2*，也可以写成 $½n^2 - ½n$。

去掉不会随数据规模n而变化的常量，可以将符号简化为 $n^2 - n$。由于 n^2的增长速度快于n，

因此也可以舍弃最后一项，使冒泡排序的平均和最坏情况下的时间复杂度为 $O(n^2)$ 。

在算法接收到已排序的数组的情况下，运行时间复杂度将降低到更好的$O(n)$，因为算法循环一遍没有任何交换，标志是`true`，所以循环一遍比较了N次直接退出。因此，$O(n)$是冒泡排序的最佳情况运行时间复杂度。但最好的情况是个例外，比较不同的算法时，应该关注平均情况。

冒泡排序的时间运行测试

使用`run_sorting_algorithm()`测试冒泡排序处理具有一万个元素的数组所花费的时间。

```
ARRAY_LENGTH = 10000


if __name__ == "__main__":
    # 生成包含“ ARRAY_LENGTH”个元素的数组，元素是介于0到999之间的随机整数值
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

    # 使用排序算法的名称和刚创建的数组调用该函数
    run_sorting_algorithm(algorithm="bubble_sort", array=array)
```

现在运行脚本来获取`bubble_sort`的执行时间：

```
$ python sorting.py
Algorithm: bubble_sort. Minimum execution time: 73.21720498399998
```

分析冒泡排序的优缺点

冒泡排序算法的主要优点是它的**简单性**，理解起来非常简单。但也看到了冒泡排序的缺点是**速度慢**，运行时间复杂度为$O(n^2)$。因此，一般对大型数组进行排序的时候，不会考虑使用冒泡排序。

Python中的插入排序算法

像冒泡排序一样，插入排序算法也易于实现和理解。但是与冒泡排序不同，它通过将每个元素与列表的其余元素进行比较并将其插入正确的位置，来一次构建一个排序的列表元素。此“插入”过程为算法命名。

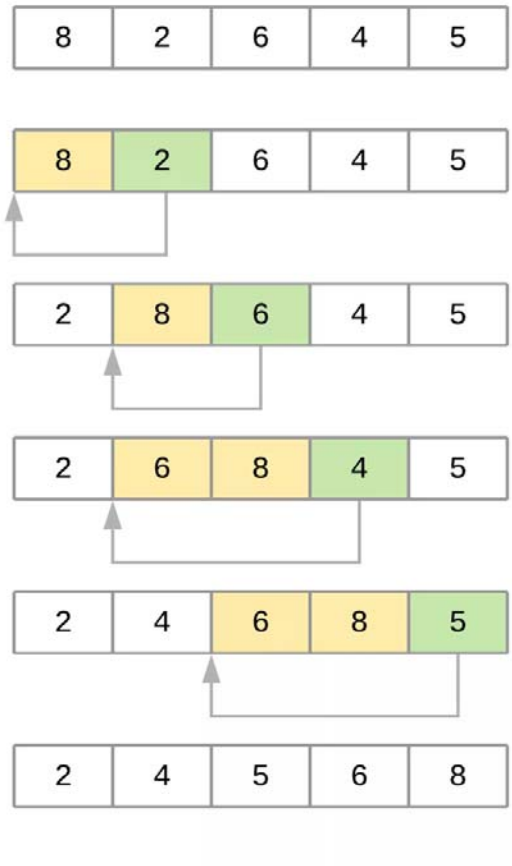
一个例子，就是对一副纸牌进行排序。将一张卡片与其余卡片进行逐个比较，直到找到正确的位置为止，然后重复进行直到您手中的所有卡都被排序。

在Python中实现插入排序

插入排序算法的工作原理与纸牌排序完全相同，Python中的实现：

```
def insertion_sort(array):
  # 从数据第二个元素开始循环，直到最后一个元素
  for i in range(1, len(array)):
      # 这个是我们想要放在正确位置的元素
      key_item = array[i]

      # 初始化变量，用于寻找元素正确位置
      j = i - 1

      # 遍历元素左边的列表元素，一旦key_item比被比较元素小，那么找到正确位置插入。
      while j >= 0 and array[j] > key_item:
          # 把被检测元素向左平移一个位置，并将j指向下一个元素（从右向左）
          array[j + 1] = array[j]
          j -= 1

      # 当完成元素位置的变换，把key_item放在正确的位置上
      array[j + 1] = key_item

  return array
```

下图显示了对数组进行排序时算法的不同迭代[8, 2, 6, 4, 5]：



插入排序过程

测量插入排序的大O时间复杂度

与冒泡排序实现类似，插入排序算法具有两个嵌套循环，遍历整个列表。内部循环非常有效，因为它会遍历列表，直到找到元素的正确位置为止。

最坏的情况发生在所提供的数组以相反顺序排序时。在这种情况下，内部循环必须执行每个比较，以将每个元素放置在正确的位置。这仍然给您带来O（n^2）运行时复杂性。

最好的情况是对提供的数组进行了排序。这里，内部循环永远不会执行，导致运行时复杂度为O（n），就像冒泡排序的最佳情况一样。

尽管冒泡排序和插入排序具有相同的大O时间复杂度，但实际上，插入排序比冒泡排序有效得多。如果查看两种算法的实现，就会看到插入排序是如何减少了对列表进行排序的比较次

数的。

插入排序时间测算

为了证明插入排序比冒泡排序更有效，可以对插入排序算法进行计时，并将其与冒泡排序的结果进行比较。调用我们写好的测试函数。

```
if __name__ == "__main__":
    # 生成包含“ ARRAY_LENGTH”个元素的数组，元素是介于0到999之间的随机整数值
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

    # 使用排序算法的名称和刚创建的数组调用该函数
   run_sorting_algorithm(algorithm="insertion_sort", array=array)
```

执行脚本：

```
  $ python sorting.py
Algorithm: insertion_sort. Minimum execution time: 56.71029764299999
```

可以看到，插入排序比冒泡排序实现少了17秒，即使它们都是O（n 2）算法，插入排序也更加有效。

分析插入排序的优点和缺点

就像冒泡排序一样，插入排序算法的实现也很简单。尽管插入排序是O（n 2）算法，但在实践中它也比其他二次实现（例如冒泡排序）更有效。

有更强大的算法，包括合并排序和快速排序，但是这些实现是递归的，在处理小型列表时通常无法击败插入排序。如果列表足够小，可以提供更快的整体实现，则某些快速排序实现甚至在内部使用插入排序。Timsort还在内部使用插入排序对输入数组的一小部分进行排序。

也就是说，插入排序不适用于大型阵列，这为可以更有效地扩展规模的算法打开了大门。

Python中的合并排序算法

合并排序是一种非常有效的排序算法。它基于**分治法**，这是一种用于解决复杂问题的强大算法技术。

要正确理解分而治之，应该首先了解**递归**的概念。递归涉及将问题分解成较小的子问题，直到它们足够小以至于无法解决。在编程中，递归通常由调用自身的函数表示。

分而治之算法通常遵循相同的结构：

原始输入分为几个部分，每个部分代表一个子问题，该子问题与原始输入相似，但更为简单。每个子问题都递归解决。所有子问题的解决方案都组合成一个整体解决方案。在合并排序的情况下，分而治之方法将输入值的集合划分为两个大小相等的部分，对每个一半进行递归排序，最后将这两个排序的部分合并为一个排序列表。

在Python中实现合并排序

合并排序算法的实现需要两个不同的部分：

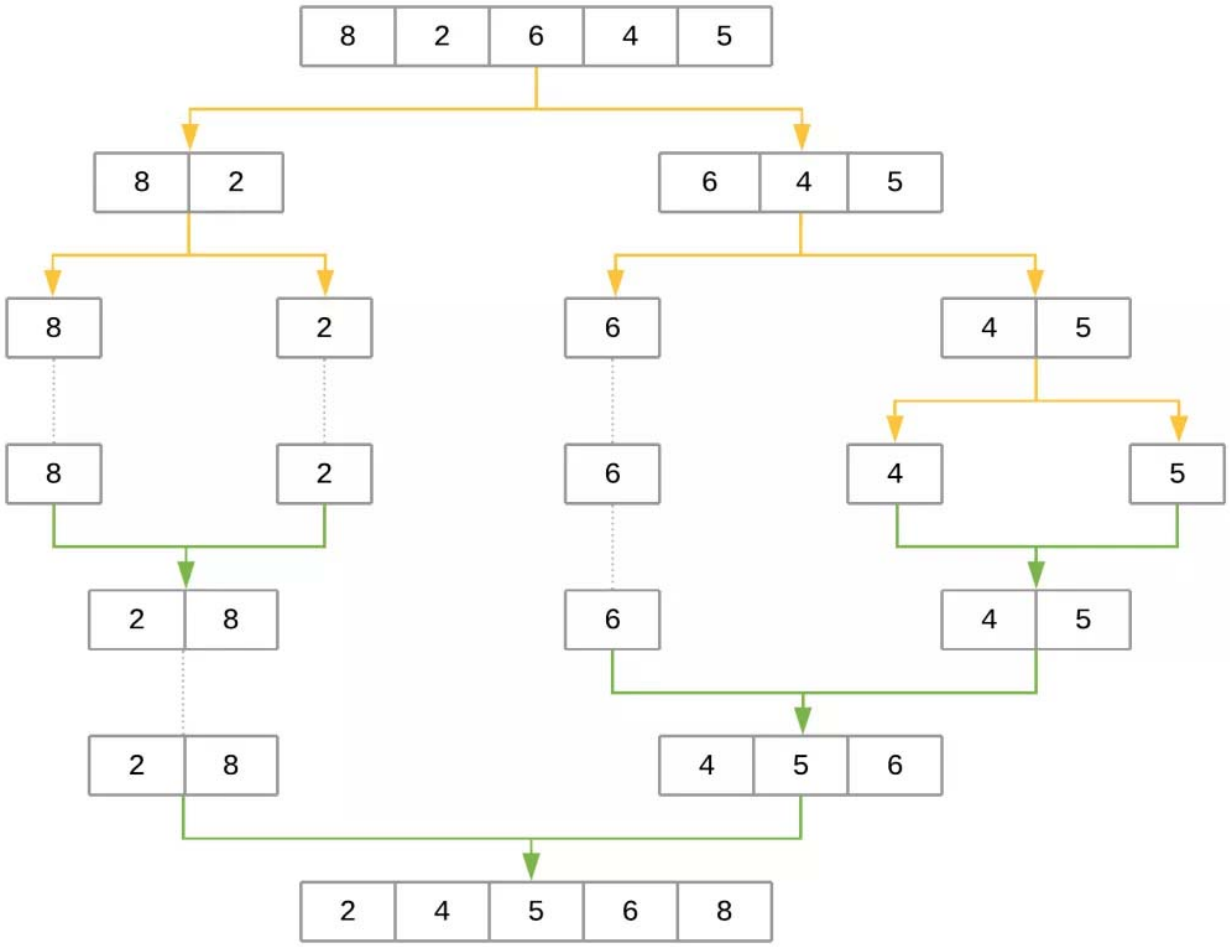
递归地将输入分成两半的函数合并两个半部的函数，产生一个排序数组这是合并两个不同数组的代码：

```
def merge(left, right):
    # 如果第一个数组为空，那么不需要合并，
    # 可以直接将第二个数组返回作为结果
    if len(left) == 0:
        return right

    # 如果第二个数组为空，那么不需要合并，
    # 可以直接将第一个数组返回作为结果
    if len(right) == 0:
        return left

    result = []
    index_left = index_right = 0

    # 查看两个数组直到所有元素都装进结果数组中
    while len(result) < len(left) + len(right):
        # 这些需要排序的元素要依次被装入结果列表，因此需要决定将从
        # 第一个还是第二个数组中取下一个元素
        if left[index_left] <= right[index_right]:
            result.append(left[index_left])
            index_left += 1
        else:
            result.append(right[index_right])
            index_right += 1

        # 如果哪个数组达到了最后一个元素，那么可以将另外一个数组的剩余元素
        # 装进结果列表中，然后终止循环
        if index_right == len(right):
            result += left[index_left:]
            break

        if index_left == len(left):
            result += right[index_right:]
            break

    return result
```

现在还缺少的部分是将输入数组递归拆分为两个并用于merge()产生最终结果的函数：

```
def merge_sort(array):
  # 如果输入数组包含元素不超过两个，那么返回它作为函数结果
  if len(array) < 2:
      return array

  midpoint = len(array) // 2

  # 对数组递归地划分为两部分，排序每部分然后合并装进最终结果列表
  return merge(
      left=merge_sort(array[:midpoint]),
      right=merge_sort(array[midpoint:]))
```

看一下合并排序将对数组进行排序的步骤[8, 2, 6, 4, 5]:



合并排序过程

该图使用黄色箭头表示在每个递归级别将数组减半。绿色箭头表示将每个子阵列合并在一起。

衡量合并排序的大O复杂度

要分析合并排序的复杂性，可以分别查看其两个步骤：

`merge()`具有线性运行时间。它接收两个数组，它们的组合长度最多为n（原始输入数组的长度），并且通过最多查看每个元素一次来组合两个数组。这导致运行时复杂度为O（n）。

第二步以递归方式拆分输入数组，并调用`merge()`每一部分。由于将数组减半直到剩下单个元素，因此此功能执行的减半运算总数为log 2 n。由于`merge()`每个部分都被调用，因此总运行时间为O（n log 2 n）。

对合并排序进行测算时间

同样通过之前时间测试函数：

```
if __name__ == "__main__":
     # 生成包含“ ARRAY_LENGTH”个元素的数组，元素是介于0到999之间的随机整数值
     array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

     # 使用排序算法的名称和刚创建的数组调用该函数
    run_sorting_algorithm(algorithm="merge_sort", array=array)
```

执行时间：

```
$ python sorting.py
Algorithm: merge_sort. Minimum execution time: 0.6195857160000173
```

与冒泡排序和插入排序相比，合并排序实现非常快，可以在一秒钟内对一万个数组进行排序了！

分析合并排序的优点和缺点

由于其运行时复杂度为O（n log 2 n），因此合并排序是一种非常有效的算法，可以随着输入数组大小的增长而很好地扩展。并行化也很简单，因为它将输入数组分成多个块，必要时可以并行分配和处理这些块。

缺点是对于较小的列表，递归的时间成本就较高了，冒泡排序和插入排序之类的算法更快。例如，运行包含十个元素的列表的实验的时间：

```
Algorithm: bubble_sort. Minimum execution time: 0.00001877499999998654
Algorithm: insertion_sort. Minimum execution time: 0.000029786000000000395
Algorithm: merge_sort. Minimum execution time: 0.00016983000000000276
```

合并排序的另一个缺点是，它在递归调用自身时会创建数组。它还在内部创建一个新列表，这使得合并排序比气泡排序和插入排序使用更多的内存。

Python中的快速排序算法

就像合并排序一样，快速排序算法采用分而治之的原理将输入数组分为两个列表，第一个包含小项目，第二个包含大项目。然后，该算法将对两个列表进行递归排序，直到对结果列表进行完全排序为止。

划分输入列表称为对列表进行**分区**。快排首先选择一个`pivot`元素，然后将列表划分为pivot，然后将每个较小的元素放入`low`数组，将每个较大的元素放入`high`数组。

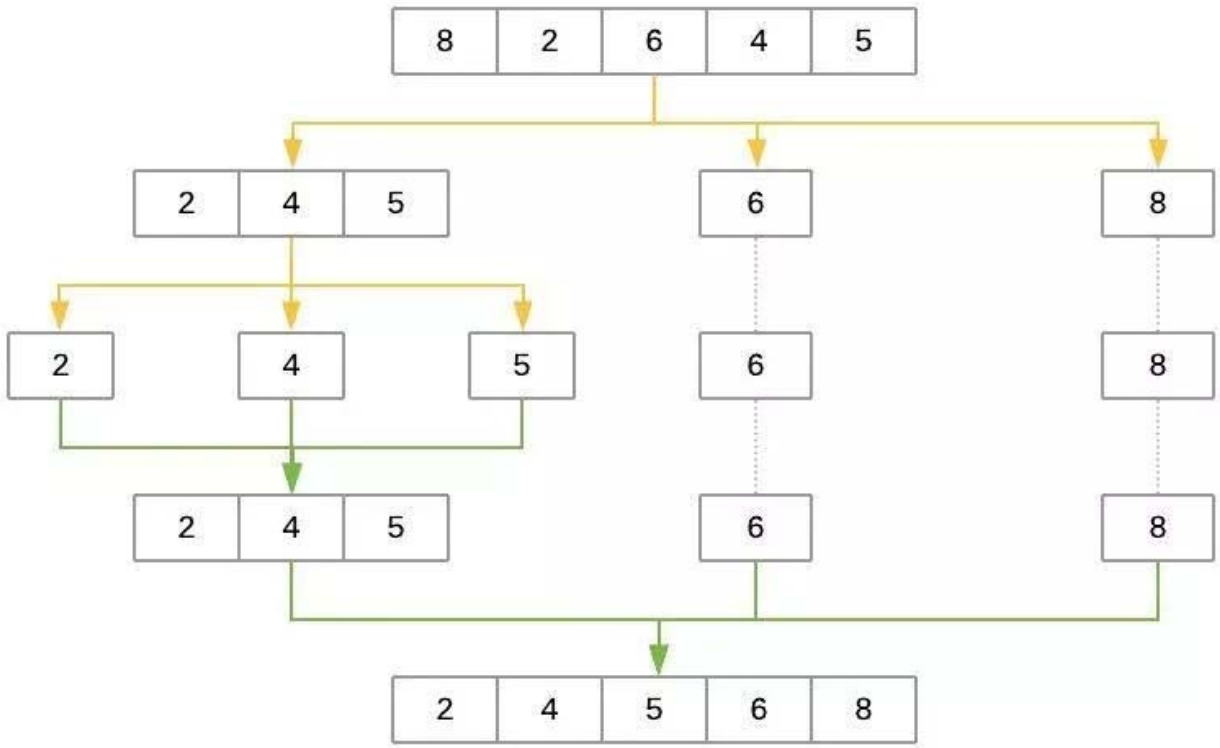
将low列表中的每个元素放在列表的左侧，列表中的pivot每个元素high放在右侧，将其pivot精确定位在最终排序列表中的确切位置。这意味着该函数现在可以递归地将相同的过程应用于low，然后high对整个列表进行排序。

在Python中实现快排

这是快排的一个相当紧凑的实现：

```
from random import randint

def quicksort(array):
    # 如果第一个数组为空，那么不需要合并，
    # 可以直接将第二个数组返回作为结果
    if len(array) < 2:
        return array

    low, same, high = [], [], []

    # 随机选择 pivot 元素
    pivot = array[randint(0, len(array) - 1)]

    for item in array:
        # 元素小于pivot元素的装进low列表中，大于piviot元素值的装进high列表中
        # 如果和pivot相等，则装进same列表中
        if item < pivot:
            low.append(item)
        elif item == pivot:
            same.append(item)
        elif item > pivot:
            high.append(item)

    # 最后的结果列表包含排序的low列表、same列表、hight列表
    return quicksort(low) + same + quicksort(high)
```

以下是快排对数组进行排序的步骤的说明[8, 2, 6, 4, 5]：



快排排序过程

快速排序流程黄线表示阵列的划分成三个列表：`low`，`same`，`high`。绿线表示排序并将这些列表放在一起。

选择pivot元素

为什么上面的实现会pivot随机选择元素？选择输入列表的第一个或最后一个元素会不会一样？

由于快速排序算法的工作原理，递归级别的数量取决于pivot每个分区的结尾位置。在最佳情况下，算法始终选择中值元素作为pivot。这将使每个生成的子问题恰好是前一个问题的一半，从而导致最多log 2 n级。

另一方面，如果算法始终选择数组的最小或最大元素作为pivot，则生成的分区将尽可能不相等，从而导致n-1个递归级别。对于快速排序，那将是最坏的情况。

如你所见，**快排的效率通常取决于pivot选择。**如果输入数组未排序，则将第一个或最后一

个元素用作，pivot将与随机元素相同。但是，如果输入数组已排序或几乎已排序，则使用第一个或最后一个元素作为pivot可能导致最坏的情况。**pivot随机选择使其更有可能使快排选择一个接近中位数的值并更快地完成。**

另一个选择是找到数组的中值，并强制算法将其用作pivot。这可以在O（n）时间内完成。尽管该过程稍微复杂一些，但将中值用作pivot快速排序可以确保您拥有最折中的大O方案。

衡量快排的大O复杂度

使用快排，将输入列表按线性时间O（n）进行分区，并且此过程将平均递归地重复log 2 n次。这导致最终复杂度为**O（n log 2 n）**。

当所选择的pivot是接近阵列的中位数，最好的情况下会发生**O（n）**，当pivot是阵列的最小或最大的值，最差的时间复杂度为**O（n 2）**。

从理论上讲，如果算法首先专注于找到中值，然后将其用作pivot元素，那么最坏情况下的复杂度将降至O（n log 2 n）。数组的中位数可以在线性时间内找到，并将其用作pivot保证代码的快速排序部分将在O（n log 2 n）中执行。

通过使用中值作为pivot，最终运行时间为O（n）+ O（n log 2 n）。你可以将其简化为O（n log 2 n），因为对数部分的增长快于线性部分。

随机选择pivot使最坏情况发生的可能性很小。这使得随机pivot选择对于该算法的大多数实现都足够好。

对快排测量运行时间

调用测试函数：

```
if __name__ == "__main__":
    # 生成包含“ ARRAY_LENGTH”个元素的数组，元素是介于0到999之间的随机整数值
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

    # 使用排序算法的名称和刚创建的数组调用该函数
    run_sorting_algorithm(algorithm="quicksort", array=array)
```

执行脚本：

```
$ python sorting.py
Algorithm: quicksort. Minimum execution time: 0.11675417600002902
```

快速排序不仅可以在不到一秒钟的时间内完成，而且比合并排序（0.11几秒钟对0.61几秒钟）要快得多。如果增加ARRAY_LENGTH从10,000到数量1,000,000并再次运行脚本，合并排序最终会在97几秒钟内完成，而快排则仅在10几秒钟内对列表进行排序。

分析快排的优势和劣势

顾名思义，快排非常快。尽管从理论上讲，它的最坏情况是O（n 2），但在实践中，快速排序的良好实现胜过大多数其他排序实现。而且，就像合并排序一样，快排也很容易并行化。

快排的主要缺点之一是缺乏保证达到平均运行时复杂度的保证。尽管最坏的情况很少见，但是某些应用程序不能承受性能不佳的风险，因此无论输入如何，它们都选择不超过O（n log 2 n）的算法。

就像合并排序一样，快排也会在内存空间与速度之间进行权衡。这可能成为对较大列表进行排序的限制。

通过快速实验对十个元素的列表进行排序，可以得出以下结果：

```
Algorithm: bubble_sort. Minimum execution time: 0.0000909000000000014
Algorithm: insertion_sort. Minimum execution time: 0.00006681900000000268
Algorithm: quicksort. Minimum execution time: 0.0001319930000000004
```

结果表明，当列表足够小时，快速排序也要付出递归的代价，完成时间比插入排序和冒泡排序都要长。

Python中的Timsort算法

最后这个算法就有意思了！

所述Timsort算法被认为是一种混合的排序算法，因为它采用插入排序和合并排序的最佳的两个世界级组合。Timsort与Python社区也很有缘，它是由Tim Peters于2002年创建的，被用作Python语言的标准排序算法。我们使用的内置sorted函数就是这个算法。

Timsort的主要特征是它利用了大多数现实数据集中存在的已排序元素。这些称为natural runs。然后，该算法会遍历列表，将元素收集到运行中，然后将它们合并到一个排序的列表中。

在Python中实现Timsort

本篇创建一个准系统的Python实现，该实现说明Timsort算法的所有部分。如果有兴趣，也可以查看Timsort的原始C实现。

实施Timsort的第一步是修改`insertion_sort()`的实现：

```
def insertion_sort(array, left=0, right=None):
    if right is None:
        right = len(array) - 1

    # 从指示的left元素循环，直到right被指示
    for i in range(left + 1, right + 1):
        # 这个是我们想要放在正确位置的元素
        key_item = array[i]

        # 初始化变量，用于寻找元素正确位置
        j = i - 1
        # 遍历元素左边的列表元素，一旦key_item比被比较元素小，那么找到正确位置插入
        while j >= left and array[j] > key_item:
            # 把被检测元素向左平移一个位置，并将j指向下一个元素（从右向左）
            array[j + 1] = array[j]
            j -= 1

        # 当完成元素位置的变换，把key_item放在正确的位置上
        array[j + 1] = key_item

    return array
```

此修改后的实现添加了两个参数left和right，它们指示应该对数组的哪一部分进行排序。这使Timsort算法可以对数组的一部分进行排序。修改功能而不是创建新功能意味着可以将其同时用于插入排序和Timsort。

现在看一下Timsort的实现：

```
def timsort(array):
    min_run = 32
    n = len(array)

    # 开始切分、排序输入数组的小部分，切分在`min_run`定义
    for i in range(0, n, min_run):
        insertion_sort(array, i, min((i + min_run - 1), n - 1))

    # 现在可以合并排序的切分块了
    # 从`min_run`开始，每次循环都加倍直到超过数组的长度
    size = min_run
    while size < n:
        # Determine the arrays that will
        # be merged together
        for start in range(0, n, size * 2):
            # 计算中点（第一个数组结束第二个数组开始的地方）和终点(第二个数组结束的地方)
            midpoint = start + size - 1
            end = min((start + size * 2 - 1), (n-1))

            # 合并两个子数组
            # left数组应该从起点到中点+1，right数组应该从中点+1到终点+1
            merged_array = merge(
                left=array[start:midpoint + 1],
                right=array[midpoint + 1:end + 1])

            # 最后，把合并的数组放回数组
            array[start:start + len(merged_array)] = merged_array

        # 每次迭代都应该让数据size加倍
        size *= 2

    return array
```

尽管实现比以前的算法要复杂一些，但是我们可以通过以下方式快速总结一下：

之前已经了解到，插入排序在小列表上速度很快，而Timsort则利用了这一优势。Timsort使用新引入的`left`和`right`参数在`insertion_sort()`对列表进行适当排序，而不必像

merge sort和快排那样创建新数组。

定义min_run = 32作为值有两个原因：

1. 使用插入排序对小数组进行排序非常快，并且min_run利用此特性的价值很小。使用min_run太大的值进行初始化将无法达到使用插入排序的目的，并使算法变慢。

2. 合并两个平衡列表比合并不成比例的列表要有效得多。min_run在合并算法创建的所有不同运行时，选择一个为2的幂的值可确保更好的性能。

结合以上两个条件，可以提供几种min_run选择。本教程中的实现min_run = 32是其中一种可能性。

衡量Timsort的大O时间复杂性

平均而言，Timsort的复杂度为O（n log 2 n），就像合并排序和快速排序一样。对数部分来自执行每个线性合并操作的运行大小加倍。

但是，Timsort在已排序或接近排序的列表上表现特别出色，从而导致了O（n）的最佳情况。在这种情况下，Timsort明显胜过合并排序，并与快速排序的最佳情况相匹配。但是，对于Timsort来说，最糟糕的情况也是O（n log 2 n）。

对Timsort测量运行时间

调用时间运行测试函数：

```
if __name__ == "__main__":
    # 生成包含“ ARRAY_LENGTH”个元素的数组，元素是介于0到999之间的随机整数值
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

    # 使用排序算法的名称和刚创建的数组调用该函数
    run_sorting_algorithm(algorithm="timsort", array=array)
```

执行时间：

```
$ python sorting.py
Algorithm: timsort. Minimum execution time: 0.5121690789999998
```

0.51秒，Timsort比合并排序整整快0.1秒，即17%。它也比插入排序快11,000%！

现在，尝试使用这四种算法对已经排序的列表进行排序，然后看看会发生什么。

```
if __name__ == "__main__":
    # 生成包含“ ARRAY_LENGTH”个元素的数组
    array = [i for i in range(ARRAY_LENGTH)]

    # 调用每个函数
    run_sorting_algorithm(algorithm="insertion_sort", array=array)
    run_sorting_algorithm(algorithm="merge_sort", array=array)
    run_sorting_algorithm(algorithm="quicksort", array=array)
    run_sorting_algorithm(algorithm="timsort", array=array)
```

现在执行脚本，那么所有算法都将运行并输出相应的执行时间：

```
Algorithm: insertion_sort. Minimum execution time: 53.5485634999991
Algorithm: merge_sort. Minimum execution time: 0.372304601
Algorithm: quicksort. Minimum execution time: 0.24626494199999982
Algorithm: timsort. Minimum execution time: 0.23350277099999994
```

这次，Timsort的速度比合并排序快了37%，比快排快了5%，从而增强了利用已排序运行的能力。

请注意，Timsort如何从两种算法中受益，这两种算法单独使用时速度要慢得多。Timsort的神奇之处在于将这些算法结合起来并发挥其优势，以获得令人印象深刻的结果。

分析Timsort的优势和劣势

Timsort的主要缺点是它的复杂性。尽管实现了原始算法的非常简化的版本，但由于它同时依赖于`insertion_sort()`和`merge()`，因此仍需要更多代码。

Timsort的优点之一是其能够以O（n log 2 n）的方式执行预测，而与输入数组的结构无关。与快排相比，快排可以降级为O（n 2）。对于小数组，Timsort也非常快，因为该算法变成了单个插入排序。

对于现实世界中的使用（通常对已经具有某些预先存在的顺序的数组进行排序），Timsort是一个不错的选择。它的适应性使其成为排序任何长度的数组的绝佳选择。

结论

排序是任何Pythonista工具包中必不可少的工具。了解Python中不同的排序算法以及如何最大程度地发挥它们的潜力，你就可以实现更快，更高效的应用程序和程序！

参考：

1. https://realpython.com/sorting-algorithms-python/
2. https://blog.csdn.net/weixin_38483589/article/details/84147376
3. https://mp.weixin.qq.com/s/OHoe6TTX--Ys5G-5yR6juA

近期热门：

- 我的神！用Python竟然还能做一个文字套娃
- 我珍藏的一些好的Python代码，技巧|上篇
- 爬取300本Python书籍，用Python告诉你哪家强？
- 卧槽！Pdf转Word用Python轻松搞定！
- 我打赌，学会这6招，谁再敢笑你的Python程序慢！

阅读原文