

用Python 實現十大經典排序算法

AI科技大本營 2022-03-09 18:15

以下文章來源於Python數據之道，作者我是陽哥



Python數據之道

點擊領取《Python知識手冊》高清電子版，回複數字“600”獲取。「Python數據之道...

洞见AI前沿, 打开未来!



作者| 我是陽哥

來源| Python數據之道

今天，詳細的跟大家分享下10 種經典排序算法。

10種經典排序算法包括冒泡排序、選擇排序、快速排序、歸併排序、堆排序、插入排序、希爾排序、計數排序、桶排序、基數排序等。

當然，還有一些其他的排序算法，大家可以繼續去研究下。

01 冒泡排序

冒泡排序（Bubble Sort）是一種比較簡單的排序算法，它重複地走訪過要排序的元素，依次比較相鄰兩個元素，如果它們的順序錯誤就把他們調換過來，直到沒有元素再需要交換，排序完成。



注：上圖中，數字表示的是數據序列原始的索引號。

算法過程

- 比較相鄰的元素，如果前一個比後一個大，就把它們兩個對調位置。
- 對排序數組中每一對相鄰元素做同樣的工作，直到全部完成，此時最後的元素將會是本輪排序中最大的數。
- 對剩下的元素繼續重複以上的步驟，直到沒有任何一個元素需要比較。

冒泡排序每次找出一個最大的元素，因此需要遍歷 $n-1$ 次（ n 為數據序列的長度）。

算法特點

什麼時候最快（Best Cases）：當輸入的數據已經是正序時。

什麼時候最慢（Worst Cases）：當輸入的數據是反序時。

Python代碼

```
def bubble_sort(lst):  
    n = len(lst)  
    for i in range(n):  
        for j in range(1, n - i):  
            if lst[j - 1] > lst[j]:  
                lst[j - 1], lst[j] = lst[j], lst[j - 1]  
    return lst
```

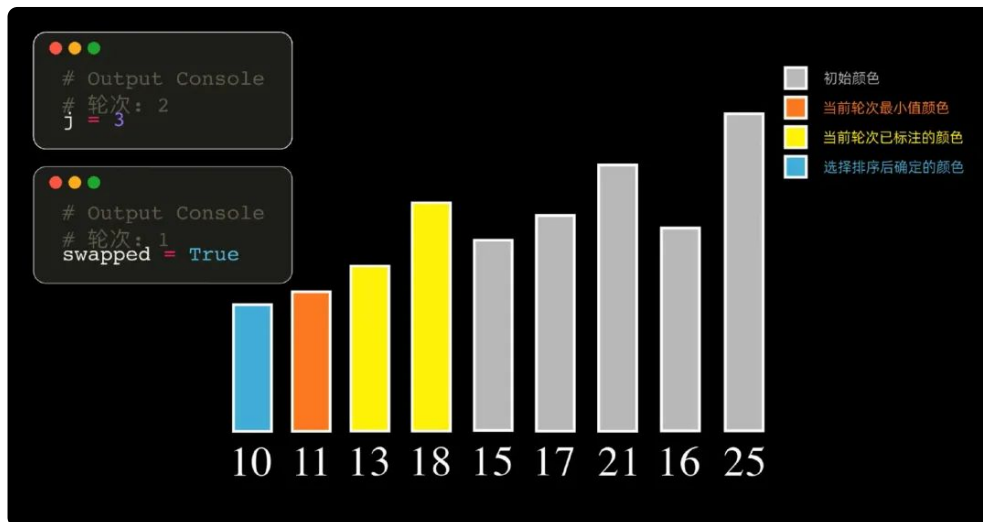
02 選擇排序

選擇排序原理

選擇排序 (Selection Sort) 的原理，每一輪從待排序的記錄中選出最小的元素，存放在序列的起始位置，然後再從剩餘的未排序元素中尋找到最小元素，然後放到已排序的序列的末尾。以此類推，直到全部待排序的數據元素的個數為零。得到數值從小到達排序的數據序列。

也可以每一輪找出數值最大的元素，這樣的話，排序完畢後的數組最終是從大到小排列。

選擇排序每次選出最小（最大）的元素，因此需要遍歷 $n-1$ 次。



Python代碼

```
def selection_sort (lst) :  
    for i in range(len(lst) - 1):  
        min_index = i  
        for j in range(i + 1 , len(lst)):  
            if lst[j] < lst[min_index]:  
                min_index = j  
        lst[i], lst[min_index] = lst[min_index], lst[i]  
    return lst
```

03 快速排序

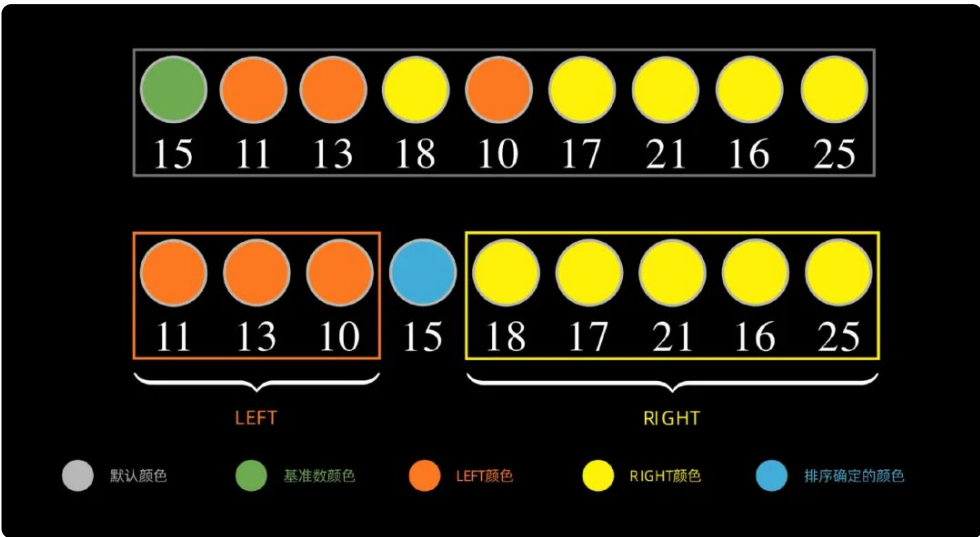
快速排序（Quick Sort），是在上世紀60年代，由美國人東尼·霍爾提出的一種排序方法。這種排序方式，在當時已經是非常快的一種排序了。因此在命名上，才將之稱為“快速排序”。

算法過程

- 1. 先從數據序列中取出一個數作為基準數（baseline，習慣取第一個數）。
- 2. 分區過程，將比基準數小的數全放到它的左邊,大於或等於它的數全放到它的右邊。
- 3. 再對左右區間遞歸(recursive)重複第二步，直到各區間只有一個數。

因為數據序列之間的順序都是固定的。最後將這些子序列一次組合起來，整體的排序就完成了。

如下圖，對於數據序列，先取第一個數據 15 為基準數，將比 15 小的數放在左邊，比 15 大（大於或等於）的數放在右邊



接下来，对于左边部分，重复上面的步骤，如下图，取左边序列的第一个数据 11 为基准数，将比 11 小的数放在左边，比 11 大（大于或等于）的数放在右边。



继续递归重复上述过程，直到每个区间只有一个数。这样就会完成排序

Python代码

```
def quick_sort(lst):
    n = len(lst)
    if n <= 1:
        return lst
    baseline = lst[0]
    left = [lst[i] for i in range(1, len(lst)) if lst[i] < baseline]
    right = [lst[i] for i in range(1, len(lst)) if lst[i] >= baseline]
    return quick_sort(left) + [baseline] + quick_sort(right)
```

04 归并排序

算法思想

归并排序（Merge Sort）是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用，归并排序将两个已经有序的子序列合并成一个有序的序列。

算法流程

主要两步(拆分，合并)

- 步骤 1：进行序列拆分，一直拆分到只有一个元素；
- 步骤 2：拆分完成后，开始递归合并。

思路：假设我们有一个没有排好序的序列，那么我们首先使用拆分的方法将这个序列分割成一个个已经排好序的子序列（直到剩下一个元素）。然后再利用归并方法将一个个有序的子序列合并成排好序的序列。

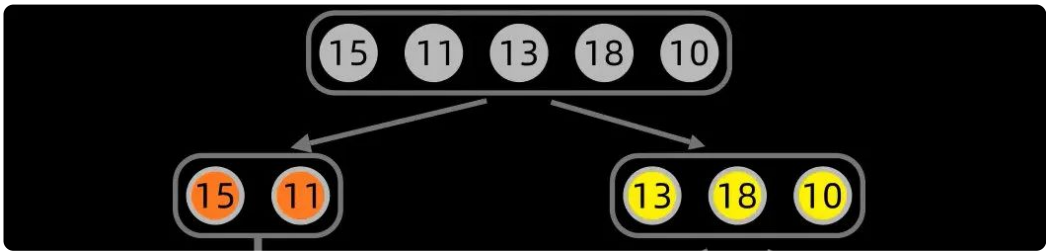
图解算法

拆分

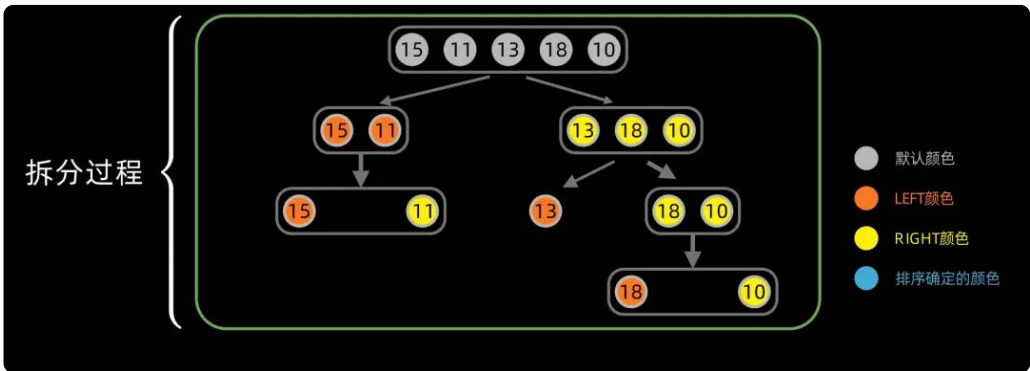
对于数据序列 [15,11,13,18,10] ,我们从首先从数据序列的中间位置开始拆分，中间位置的设置为

$mid = n/2$

首次拆分如下：



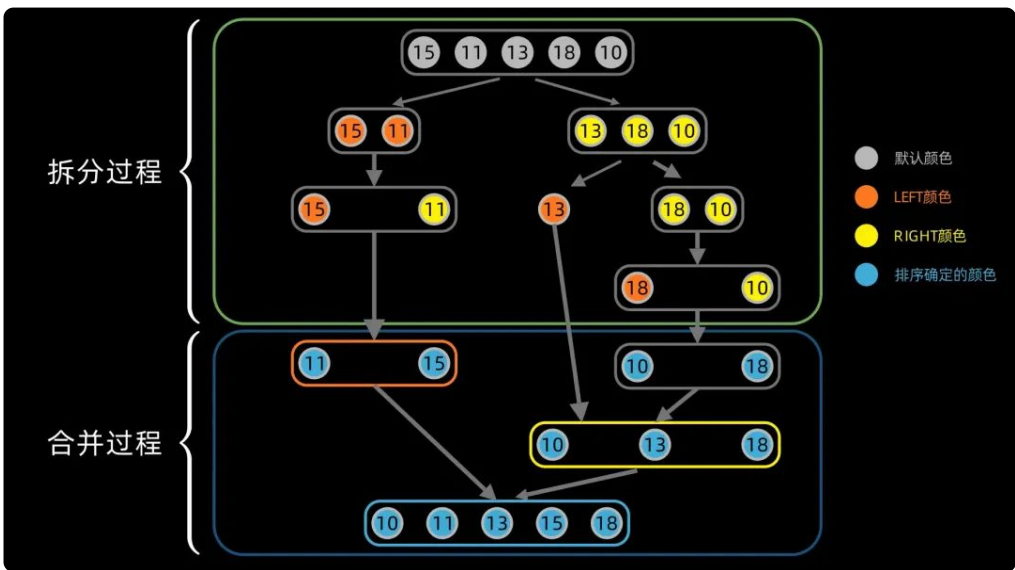
第一次拆分后，依次对子序列进行拆分，拆分过程如下：



合并

合并过程中，对于左右分区及其子区间，递归使用合并方法。先从左边最小的子区间开始，对于每个区间，依次将最小的数据放在最左边，然后对右边区间也执行同样的操作。

合并过程的完整图示如下：



Python代码

```
def merge_sort(lst):
    def merge(left, right):
        i = 0
        j = 0
        result = []
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result = result + left[i:] + right[j:]
        return result
    n = len(lst)
    if n <= 1:
        return lst
    mid = n // 2
    left = merge_sort(lst[:mid])
    right = merge_sort(lst[mid:])
    return merge(left, right)
```

05 堆排序

要理解堆排序（Heap Sort）算法，首先要知道什么是“堆”。

堆的定义

对于 n 个元素的数据序列 $\{k_0, k_1, k_2, \dots, k_{n-1}\}$ ，当且仅当满足下列情形之一时，才称之为堆：

情形1：

$$\text{最小化堆或小顶堆} \begin{cases} k_i \leq k_{2i+1} \\ k_i \leq k_{2i+2} \end{cases}$$

其中 $i=0,1,2,\dots,n//2$

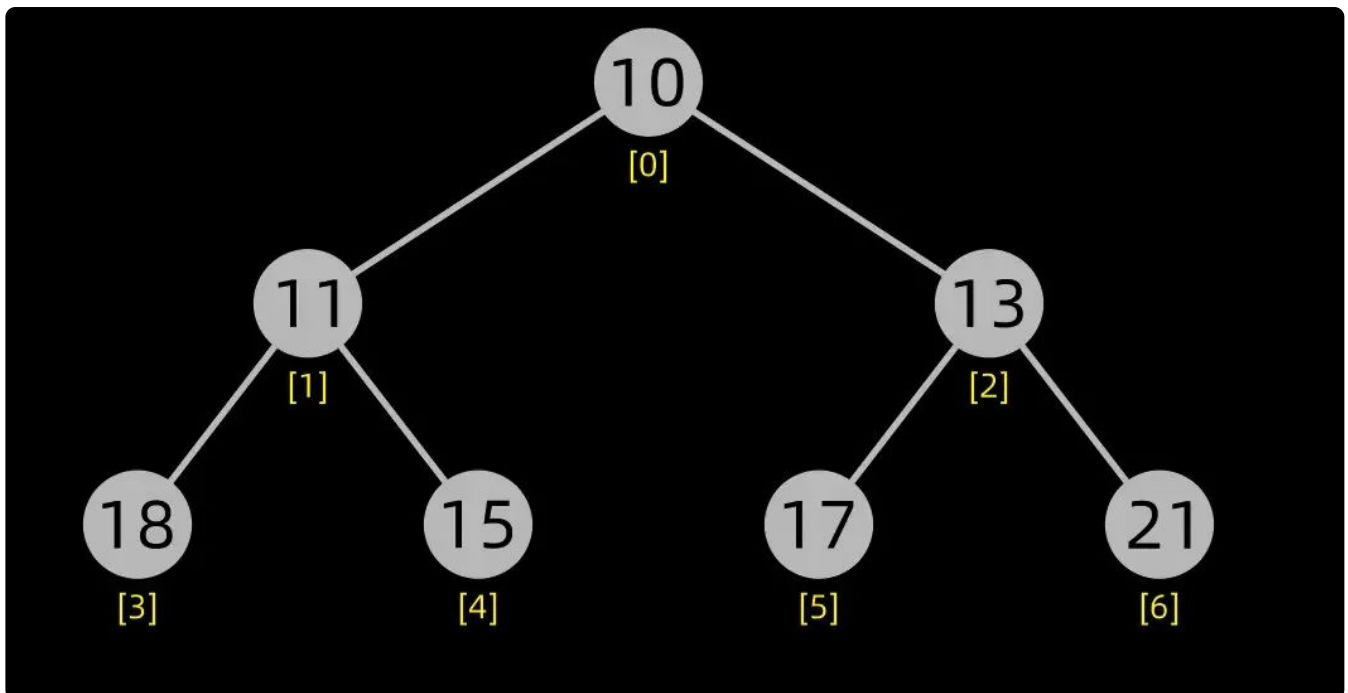
情形2:

最大化堆或大顶堆 $\begin{cases} k_i \geq k_{2i+1} \\ k_i \geq k_{2i+2} \end{cases}$

其中 $i=0,1,2,\dots,n//2$

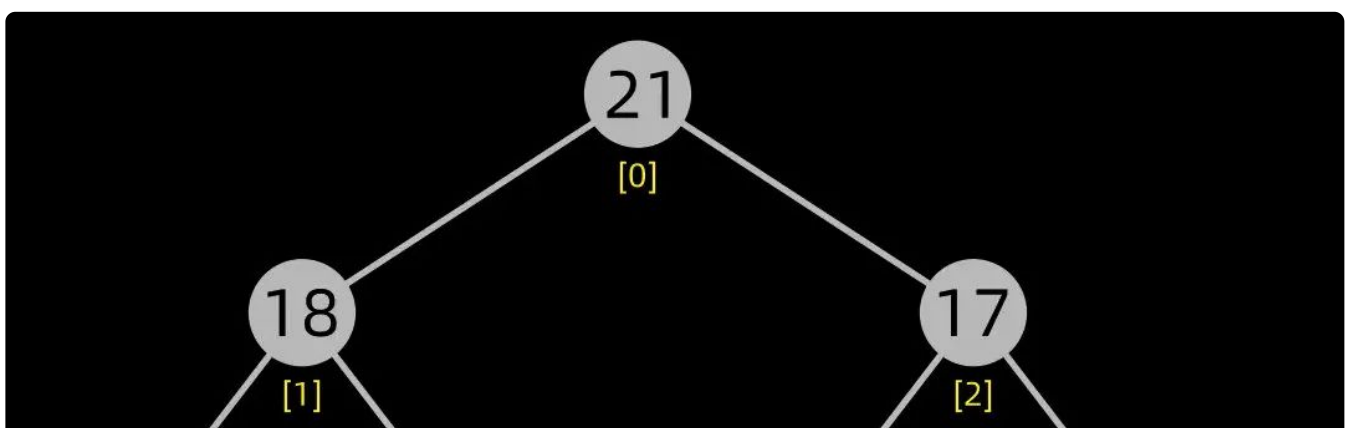
若序列 $\{k_0, k_1, k_2, \dots, k_{n-1}\}$ 是堆, 则堆顶元素必为序列中 n 个元素的最小值或最大值。

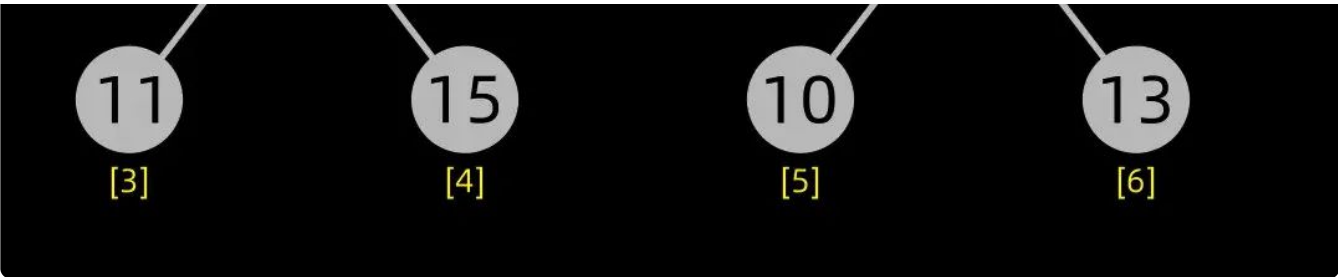
小顶堆 如下图所示:



小顶堆

大顶堆 如下图所示:





大顶堆

若在输出堆顶的最小值（或最大值）之后，使得剩余n-1个元素的序列重又建成一个堆，则得到n个元素的次小值（或次大值）。如此反复执行，便能得到一个有序序列，这个过程称之为 **堆排序**。

堆的存储

一般用数组来表示堆，若根结点存在序号 0 处， i 结点的父结点下标就为 $(i-1)/2$ 。 i 结点的左右子结点下标分别为 $2*i+1$ 和 $2*i+2$ 。

对于上面提到的小顶堆和大顶堆，其数据存储情况如下：

```
graph TD; 10((10)) --- 11((11)); 10 --- 13((13)); 11 --- 18((18)); 11 --- 15((15)); 13 --- 17((17)); 13 --- 21((21));
```

数据存储

Index	Array
[0]	10
[1]	11
[2]	13
[3]	18
[4]	15
[5]	17
[6]	21

小顶堆

```
graph TD; 21((21)) --- 18((18)); 21 --- 17((17)); 18 --- 11((11)); 18 --- 15((15)); 17 --- 10((10)); 17 --- 13((13));
```

数据存储

Index	Array
[0]	21
[1]	18
[2]	17
[3]	11
[4]	15
[5]	10
[6]	13

大顶堆

每幅图的右边为其数据存储结构，左边为其逻辑结构。

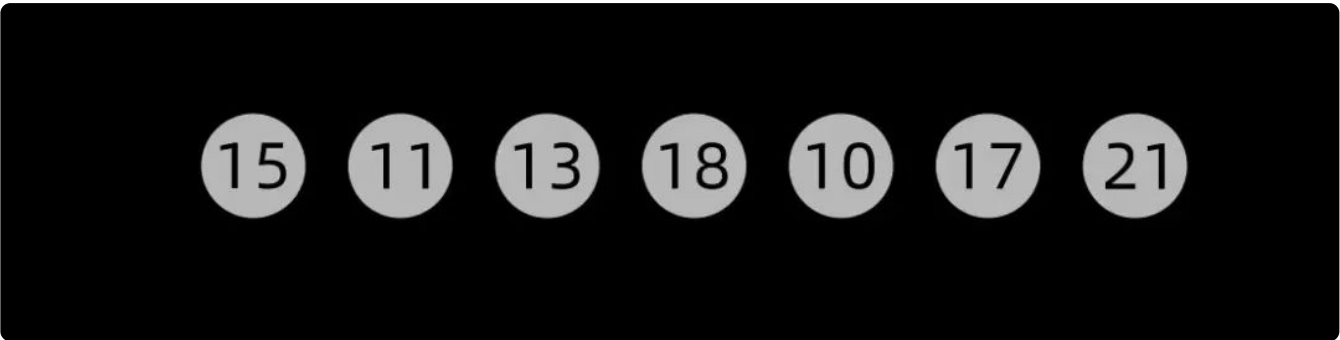
堆排序

实现堆排序需要解决两个问题：

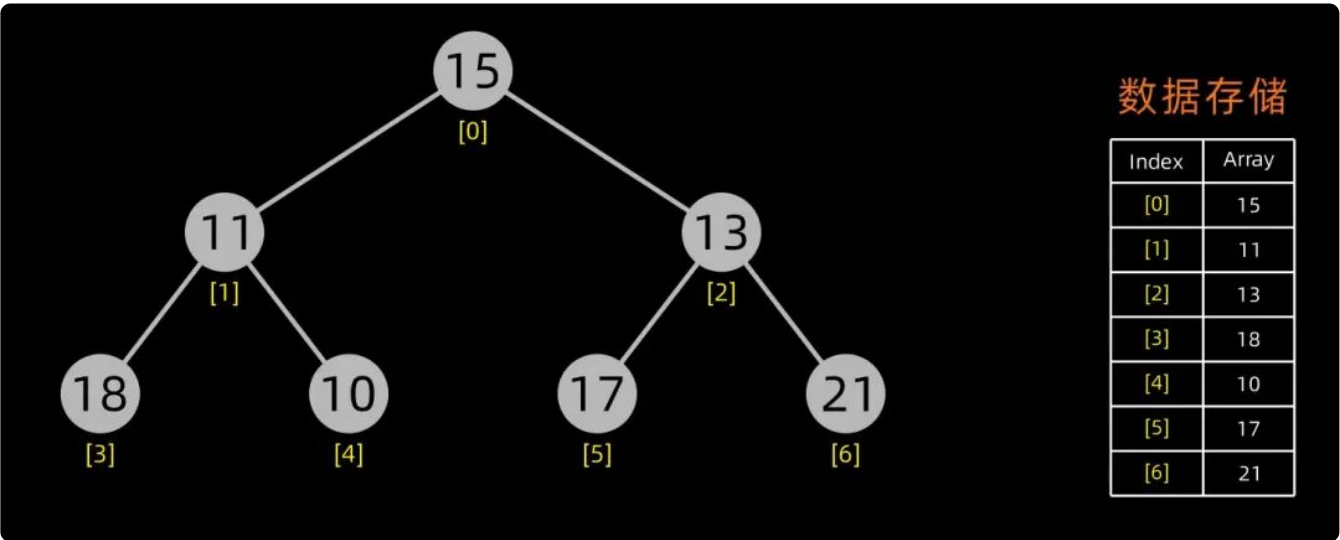
- 1. 如何由一个无序序列建成一个堆？
- 2. 如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

堆的初始化

第一个问题实际上就是堆的初始化，下面来阐述下如何构造初始堆，假设初始的数据序列如下：



咱们首先需要将其以树形结构来展示，如下：

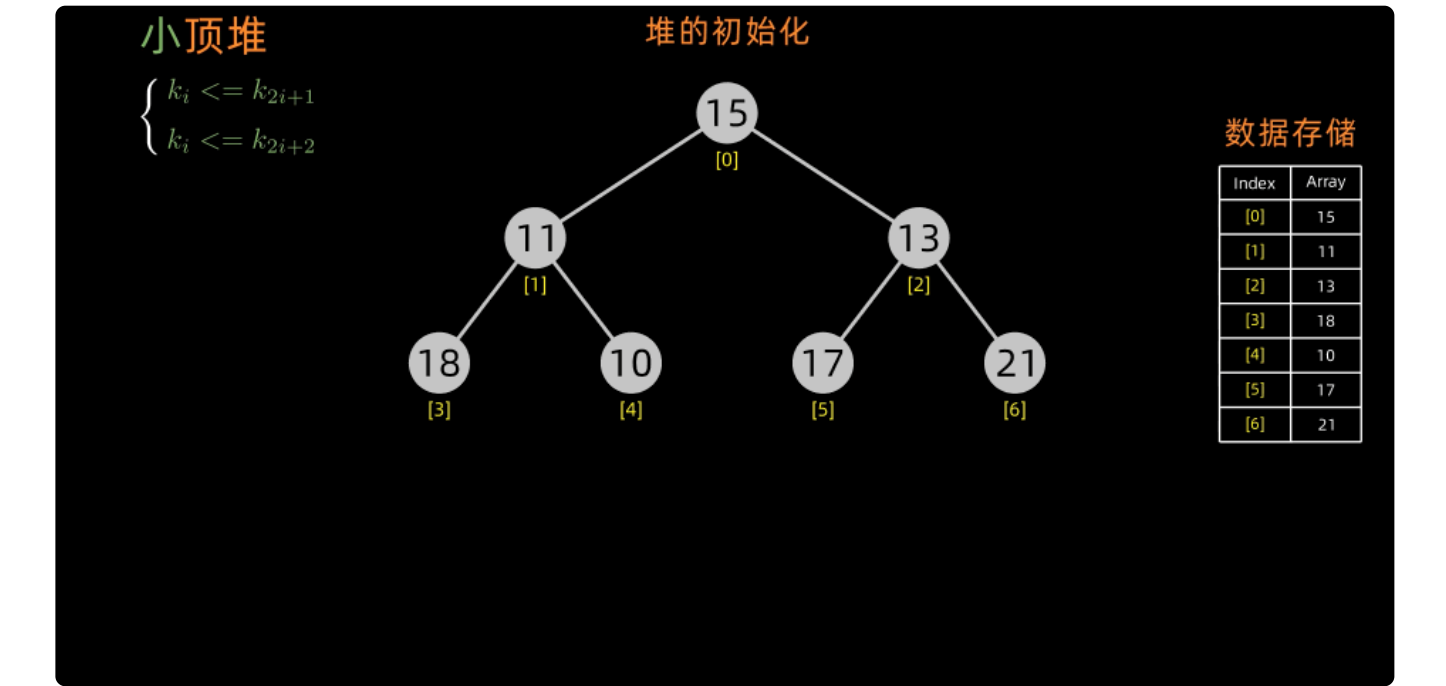


初始化堆的时候是对所有的非叶子结点进行筛选。

最后一个非终端元素的下标是 $\lfloor n/2 \rfloor$ 向下取整，所以筛选只需要从第 $\lfloor n/2 \rfloor$ 向下取整个元素开始，从后往前进行调整。

从最后一个非叶子结点开始，每次都是从父结点、左边子节点、右边子节点中进行比较交换，交换可能会引起子结点不满足堆的性质，所以每次交换之后需要重新对被交换的子结点进行调整。

以小顶堆为例，构造初始堆的过程如下：

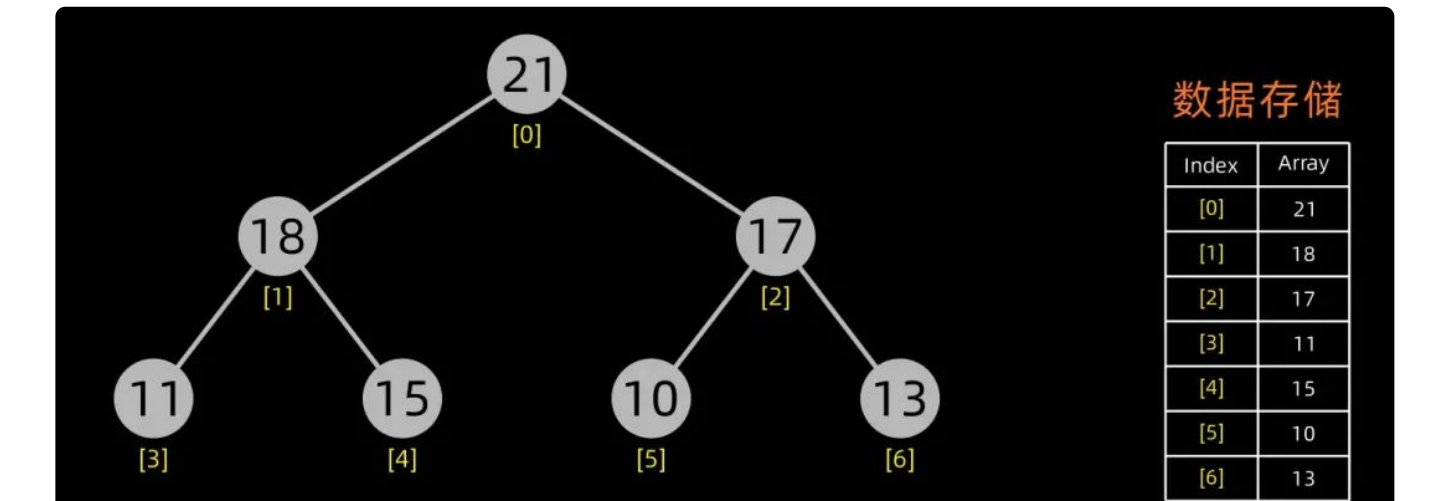


进行堆排序

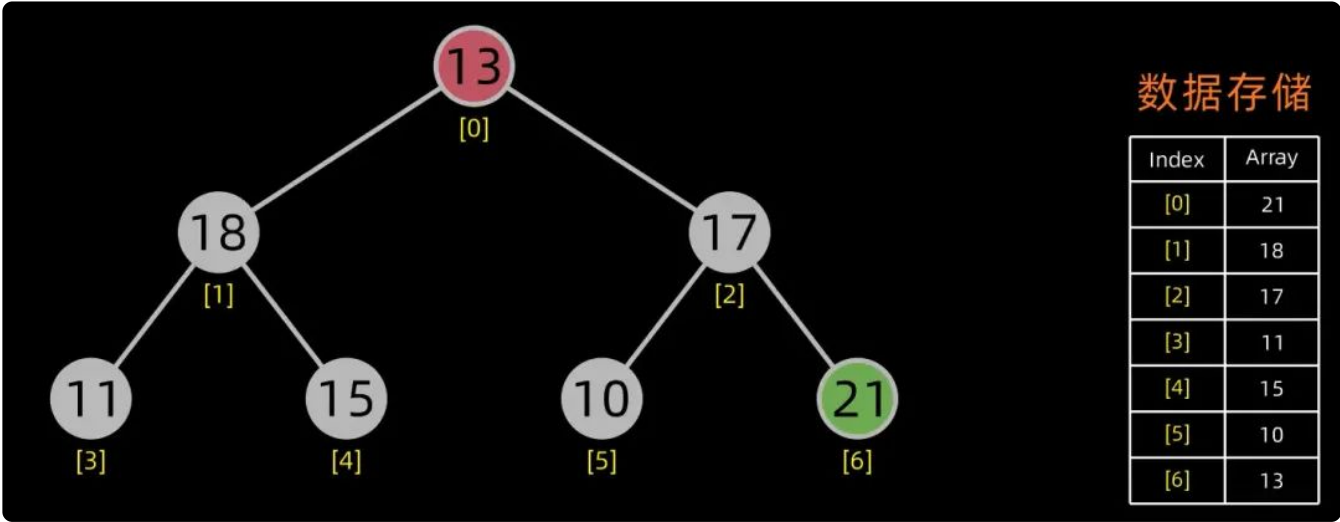
有了初始堆之后就可以进行排序了。

堆排序是一种选择排序。建立的初始堆为初始的无序区。

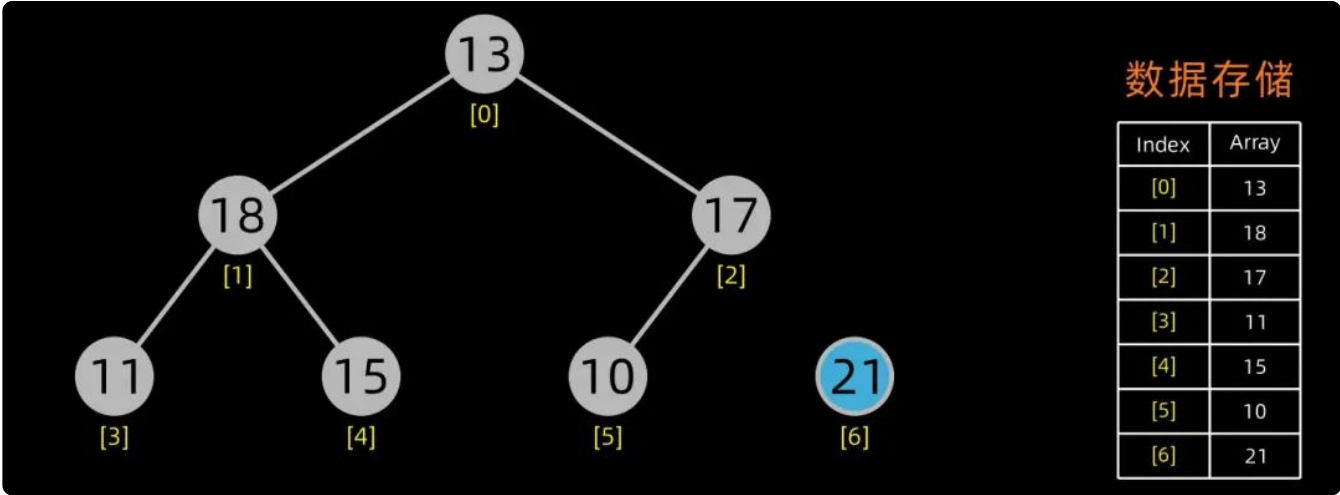
排序开始，首先输出堆顶元素（因为它是最值），将堆顶元素和最后一个元素交换，这样，第n个位置（即最后一个位置）作为有序区，前n-1个位置仍是无序区，对无序区进行调整，得到堆之后，再交换堆顶和最后一个元素，这样有序区长度变为2。。。



大顶堆



交换堆顶元素和最后的元素



无序区-1，有序区+1

不断进行此操作，将剩下的元素重新调整为堆，然后输出堆顶元素到有序区。每次交换都导致无序区-1，有序区+1。不断重复此过程直到有序区长度增长为n-1，排序完成。

Python代码

```
def heap_sort(lst):
    def adjust_heap(lst, i, size):
        left_index = 2 * i + 1
        right_index = 2 * i + 2
        largest_index = i
        if left_index < size and lst[left_index] > lst[largest_index]:
            largest_index = left_index
        if right_index < size and lst[right_index] > lst[largest_index]:
            largest_index = right_index
        if largest_index != i:
            lst[i], lst[largest_index] = lst[largest_index], lst[i]
            adjust_heap(lst, largest_index, size)
```

```
if largest_index != i:
    lst[largest_index], lst[i] = lst[i], lst[largest_index]
    adjust_heap(lst, largest_index, size)

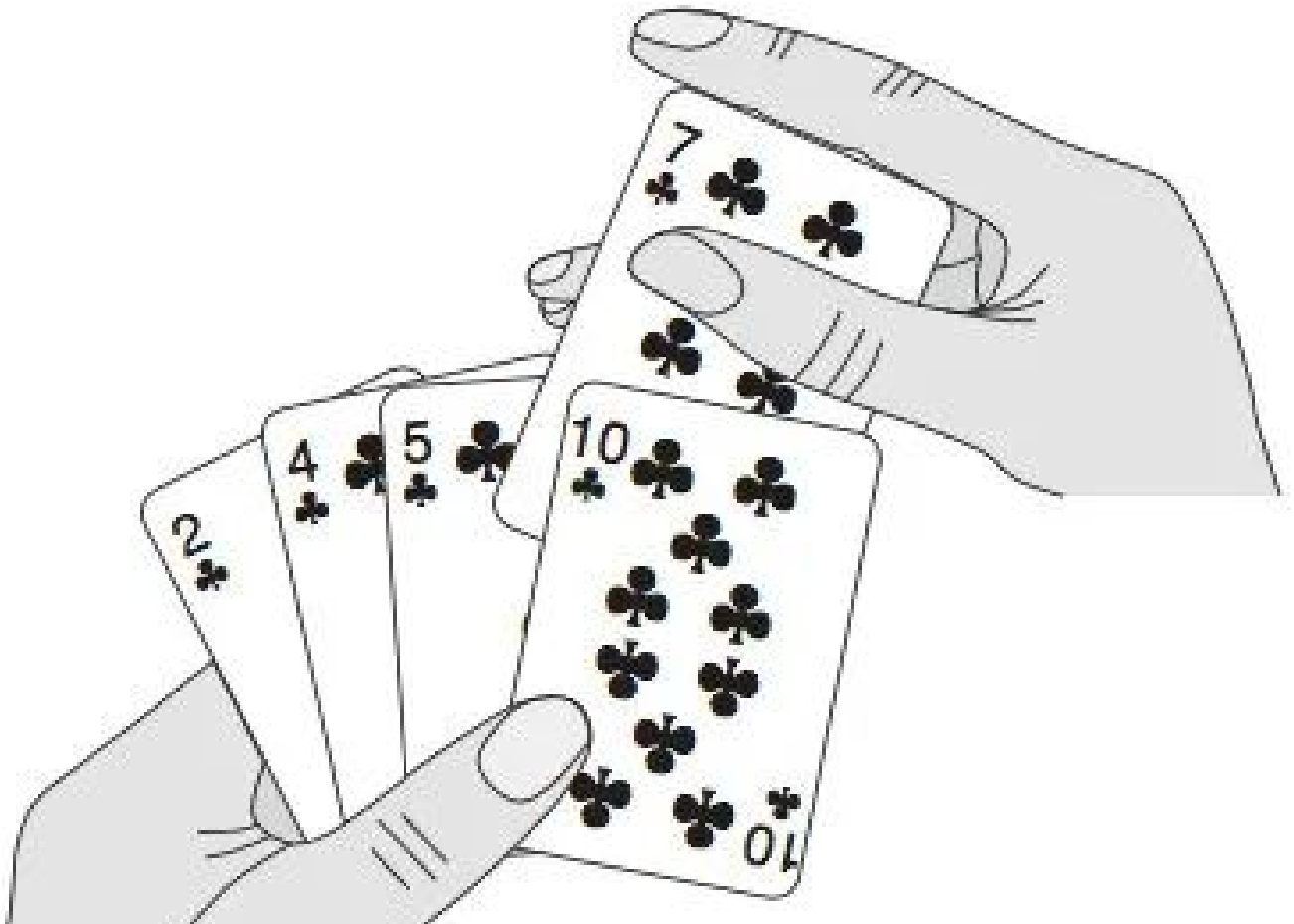
def built_heap(lst, size):
    for i in range(len(lst)//2[::-1]):
        adjust_heap(lst, i, size)

size = len(lst)
built_heap(lst, size)
for i in range(len(lst))[::-1]:
    lst[0], lst[i] = lst[i], lst[0]
    adjust_heap(lst, 0, i)
return lst
```

06 插入排序

插入排序(Insertion Sort)就是每一步都将一个需要排序的数据按其大小插入到已经排序的数据序列中的适当位置，直到全部插入完毕。

插入排序如同打扑克牌一样，每次将后面的牌插到前面已经排好序的牌中。





Python代码

```
def insertion_sort(lst):  
    for i in range(len(lst) - 1):  
        cur_num, pre_index = lst[i+1], i  
        while pre_index >= 0 and cur_num < lst[pre_index]:  
            lst[pre_index + 1] = lst[pre_index]  
            pre_index -= 1  
        lst[pre_index + 1] = cur_num  
    return lst
```

07 希尔排序

基本原理

希尔排序(Shell Sort)是插入排序的一种更高效率的实现。

希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。

这里以动态间隔序列为例来描述。初始间隔（gap值）为数据序列长度除以2取整，后续间隔以 前一个间隔数值除以2取整为循环，直到最后一个间隔值为 1 。

对于下面这个数据序列，初始间隔数值为5

15 81 13 18 10 17 21 16 25 35 39

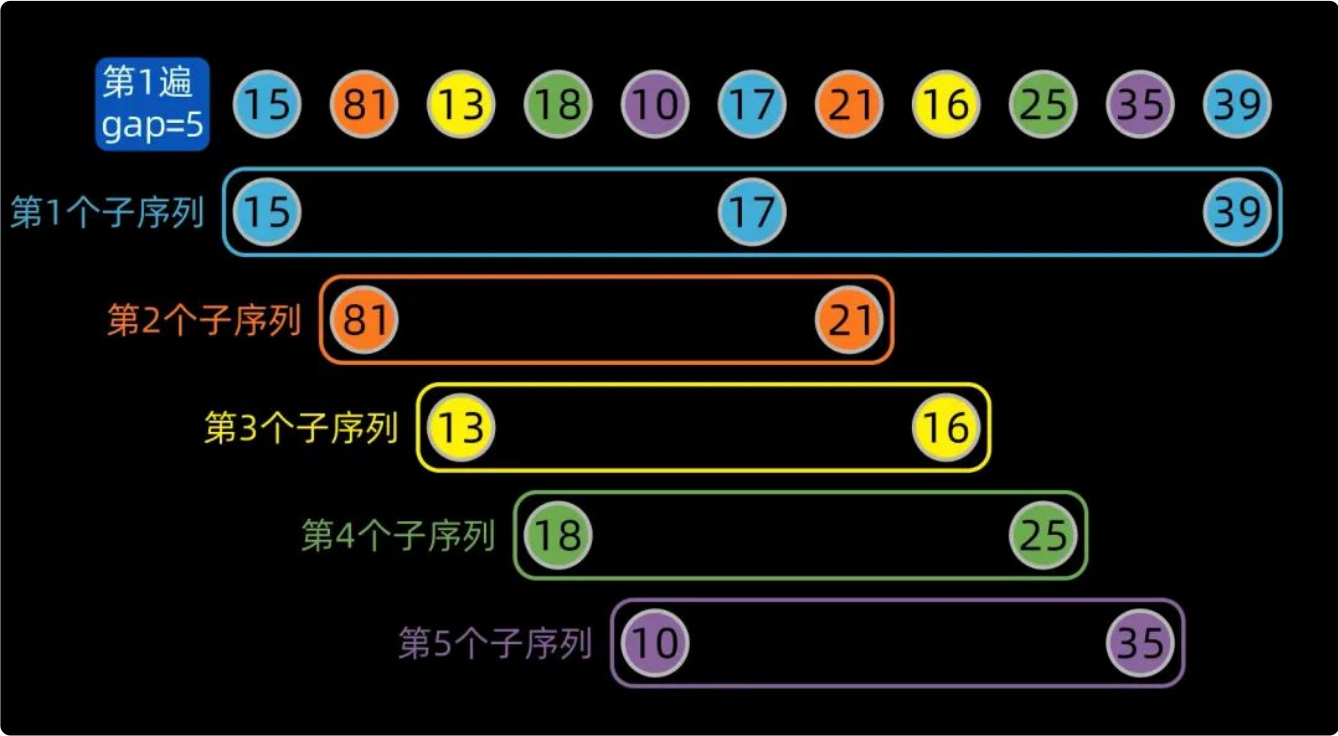
初始间隔

$$gap = len(lst) // 2 \quad gap = 11 // 2$$
$$gap = 5$$

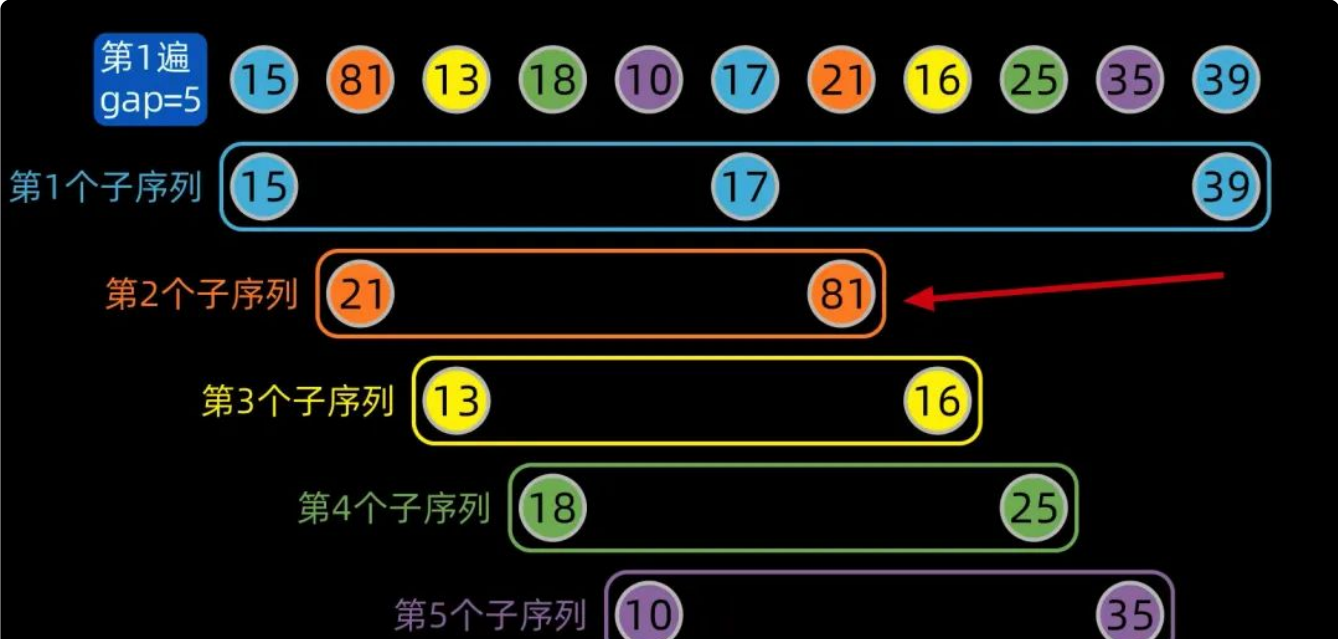
先将数据序列按间隔进行子序列分组，第一个子序列的索引为[0,5,10]，这里分成了5组。

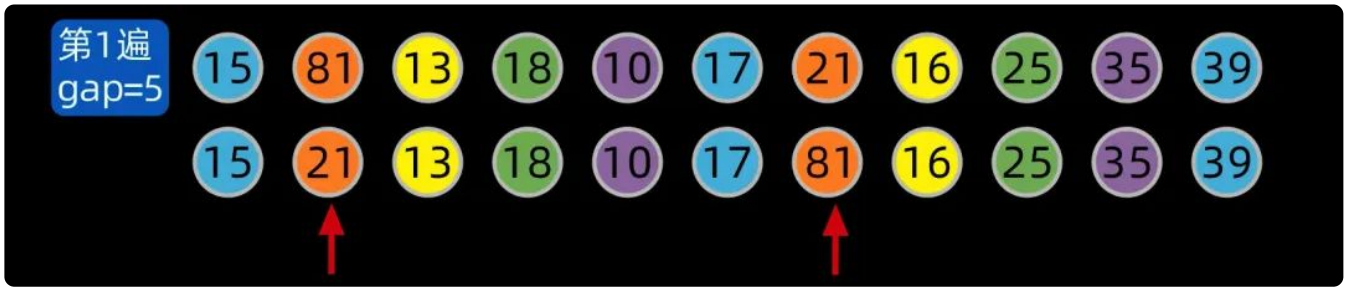


为大家方便区分不同的子序列，对同一个子序列标注相同的颜色，分组情况如下：



分组结束后，子序列内部进行插入排序，gap为5的子序列内部排序后如下：





注：红色箭头标注的地方，是子序列内部排序后的状态

接下来选取第二个间隔值，按照间隔值进行子序列分组，同样地，子序列内部分别进行插入排序；

如果数据序列比较长，则会选取第3个、第4个或者更多个间隔值，重复上述的步骤。

后续的间隔

$$\begin{cases} gap = gap // 2 \\ gap > 0 \end{cases}$$

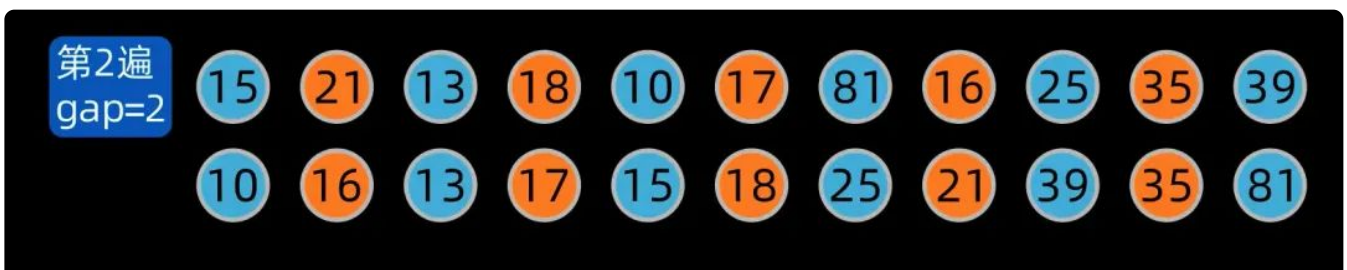
$$gap = 5 // 2$$

$$gap = 2$$

$$gap = 2 // 2$$

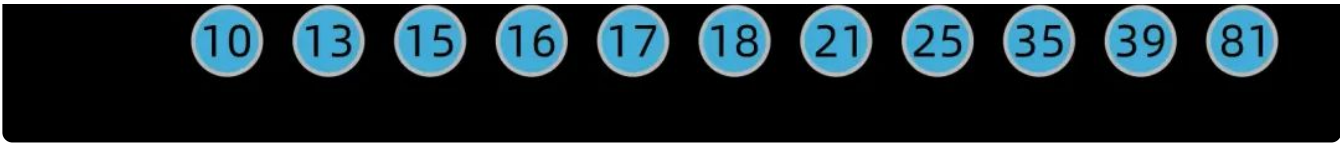
$$gap = 1$$

gap为2的排序情况前后对照如下：



最后一个间隔值为1，这一次相当于简单的插入排序。但是经过前几次排序，序列已经基本有序，因此最后一次排序时间效率就提高了很多。





Python代码

```
def shell_sort(lst):
    n = len(lst)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            for j in range(i, gap - 1, -gap):
                if lst[j] < lst[j - gap]:
                    lst[j], lst[j - gap] = lst[j - gap], lst[j]
            else:
                break
        gap //= 2
    return lst
```

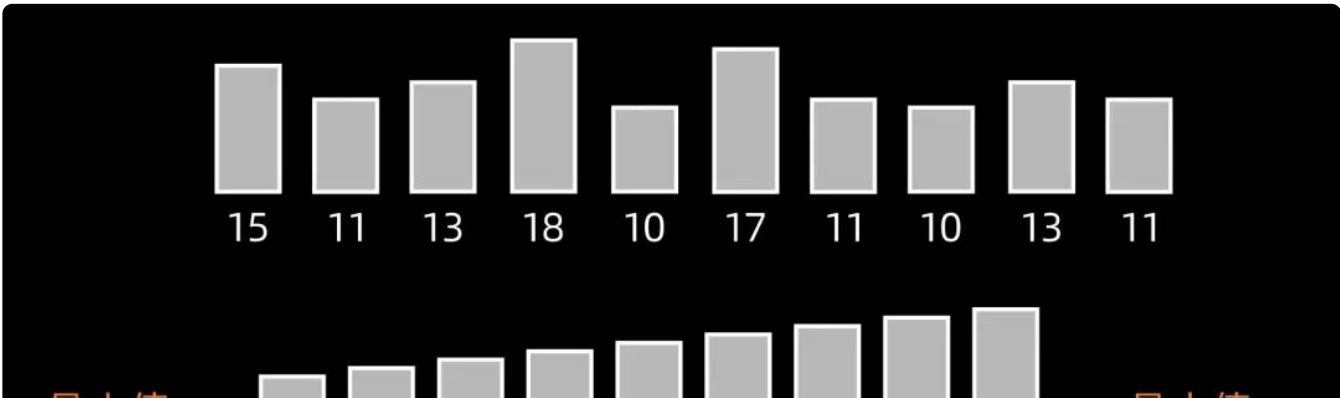
08 计数排序

基本原理

计数排序（Counting Sort）的核心在于将输入的数据值转化为键，存储在额外开辟的数组空间中。计数排序要求输入的数据必须是有确定范围的整数。

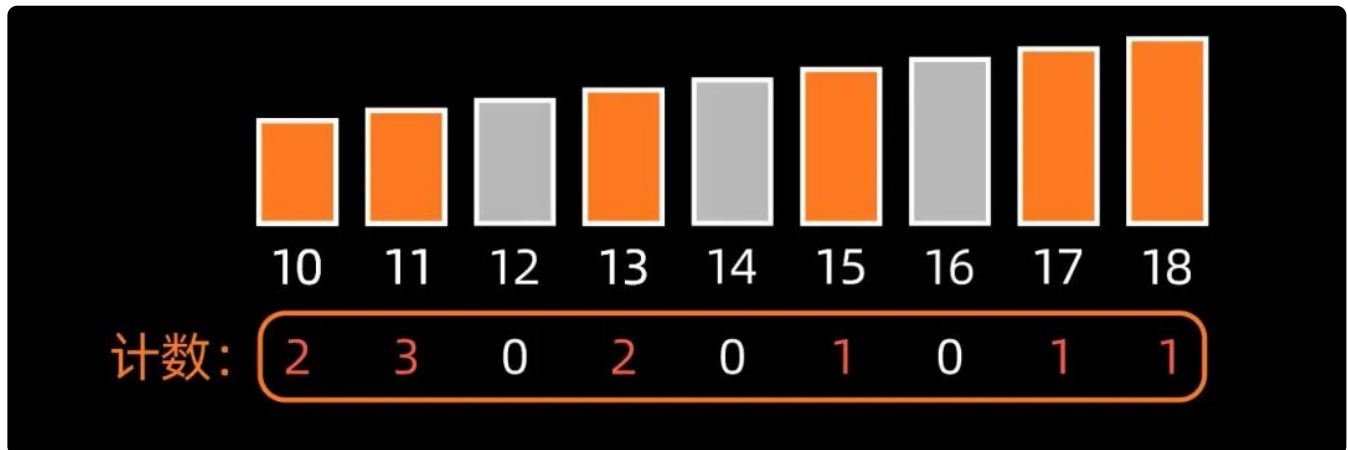
算法的步骤如下：

先找出待排序的数组中最大和最小的元素，新开辟一个长度为 **最大值-最小值+1** 的数组；

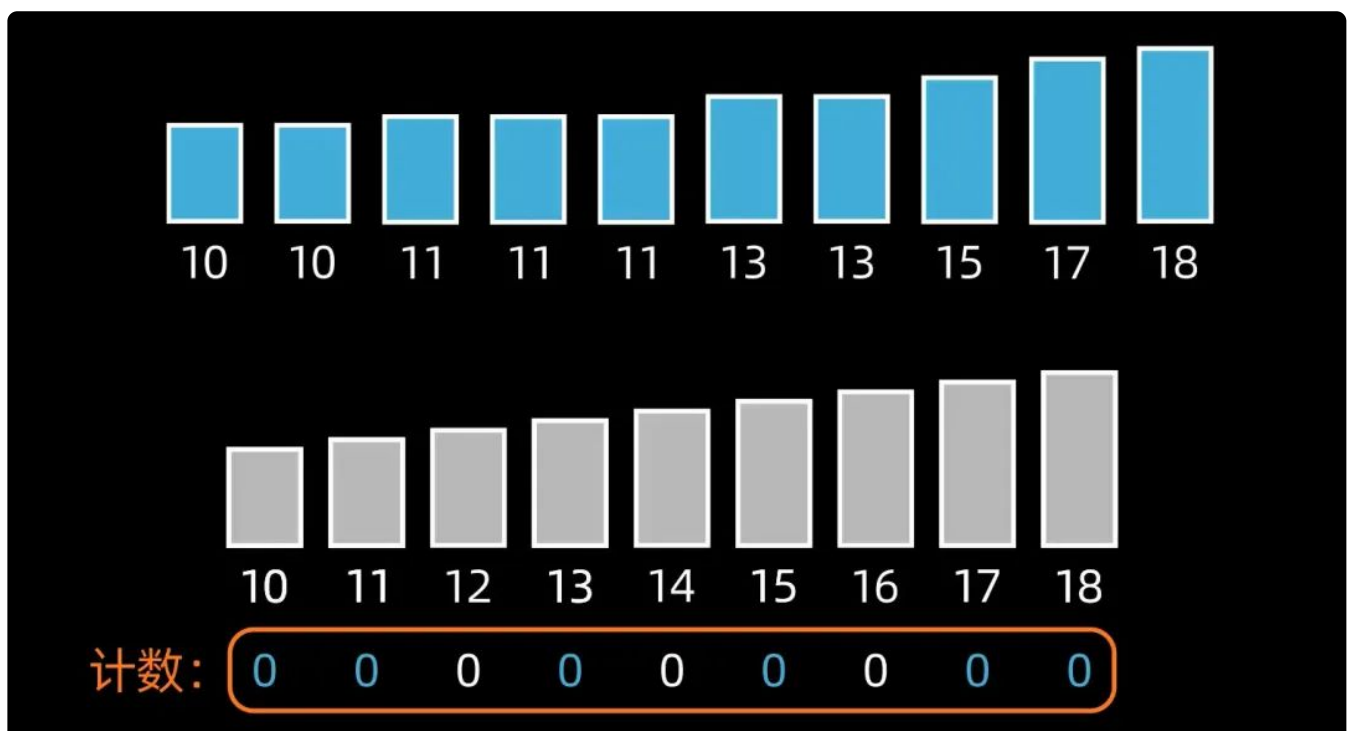




然后，统计原数组中每个元素出现的次数，存入到新开辟的数组中；



接下来，根据每个元素出现的次数，按照新开辟数组中从小到大的秩序，依次填充到原来待排序的数组中，完成排序。



Python代码

```
def counting_sort(lst):
```

```
    nums_min = min(lst)
```

https://mp.weixin.qq.com/s/dOWFfv_8Q6vy5usX0UB9DA

```
bucket = [0] * (max(lst) + 1 - nums_min)

for num in lst:
    bucket[num - nums_min] += 1

i = 0
for j in range(len(bucket)):
    while bucket[j] > 0:
        lst[i] = j + nums_min
        bucket[j] -= 1
        i += 1

return lst
```

09 桶排序

基本思想

简单来说，桶排序（Bucket Sort）就是把数据分组，放在一个个的桶中，对每个桶里面的数据进行排序，然后将桶进行数据合并，完成桶排序。

该算法分为四步，包括划分桶、数据入桶、桶内排序、数据合并。

桶的划分过程

这里详细介绍下桶的划分过程。

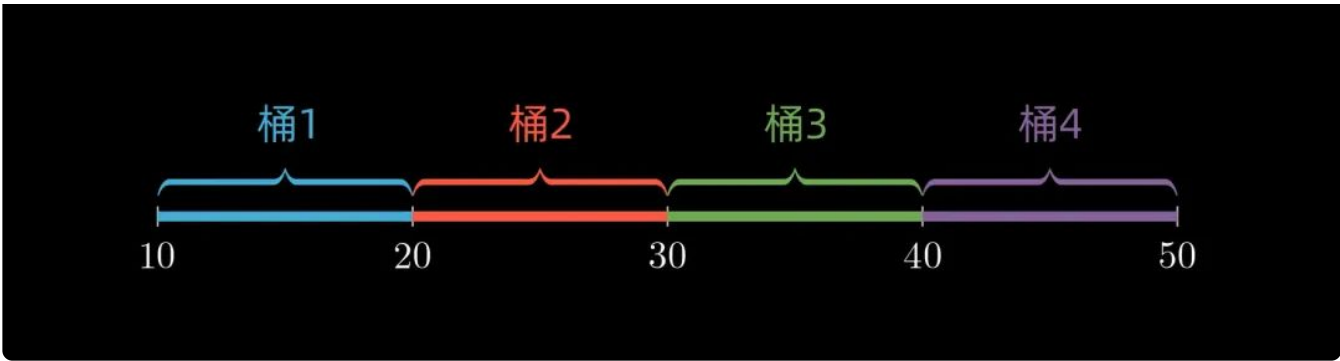
对于一个数值范围在10到 49范围内的数组，我们取桶的大小为10（`defaultBucketSize = 10`），则第一个桶的范围为 10到20，第二个桶的数据范围是20到30，依次类推。最后，我们一共需要4个桶来放入数据。

34 13 18 10 31 29 20 20 31 42 30 39 24 36 27



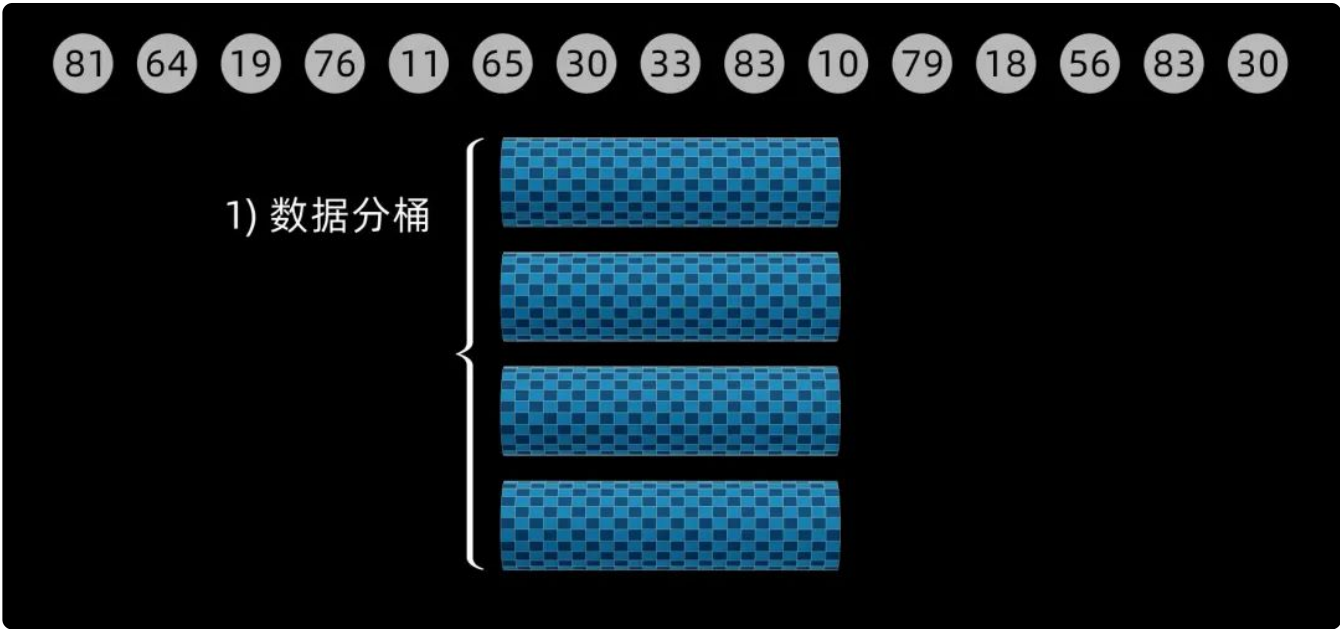
```
defaultBucketSize = 10
```

34 13 18 10 31 29 20 20 31 42 30 39 24 36 27

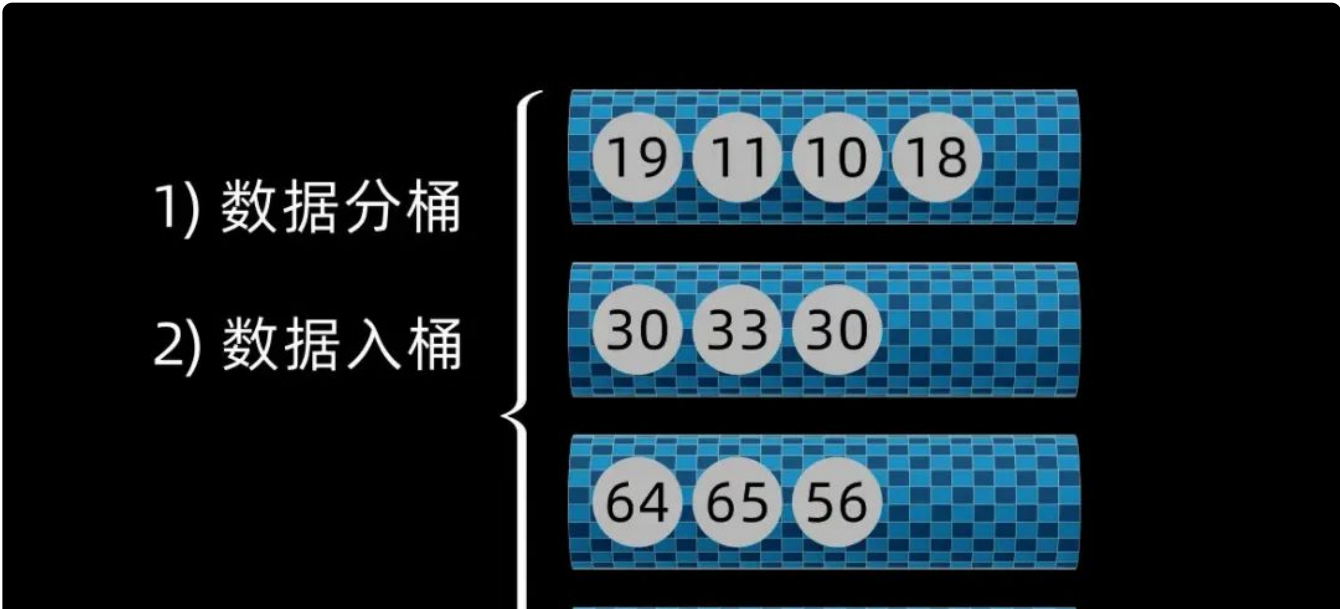


排序过程

对于下面这个数据序列，初始设定桶的大小为 20 （ defaultBucketSize = 20 ），经计算，一共需要4个桶来放入数据。



然后将原始数组按数值大小放入到对应的桶中，完成数据分组。





对于桶内的数据序列，这时可以用冒泡排序、选择排序等多种排序算法来对数据进行排序。这些算法，在之前的视频里已有介绍，大家可以去了解。

这里，我选用 **冒泡排序** 来对桶内数据进行排序。

1) 数据分桶

2) 数据入桶

3) 桶内排序



桶内排序完成后，将数据按桶的顺序进行合并，这样就得到所有数值排好序的数据序列了



Python代码

```
def bucket_sort(lst, defaultBucketSize=4):  
    maxVal, minVal = max(lst), min(lst)  
    bucketSize = defaultBucketSize  
    bucketCount = (maxVal - minVal) // bucketSize + 1
```

```
buckets = [[] for i in range(bucketCount)]  
for num in lst:  
    buckets[(num - minVal) // bucketSize].append(num)  
lst.clear()  
for bucket in buckets:  
    bubble_sort(bucket)  
    lst.extend(bucket)  
return lst
```

10 基数排序

基数排序（radix sort）属于“分配式排序”（distribution sort），它是透过键值的部份信息，将要排序的元素分配至某些“桶”中，以达到排序的作用。

基数排序适用于所有元素均为正整数的数组。

基本思想

排序过程分为“分配”和“收集”。

排序过程中，将元素分层为多个关键码进行排序（一般按照数值的个位、十位、百位、..... 进行区分），多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序逐次排序。

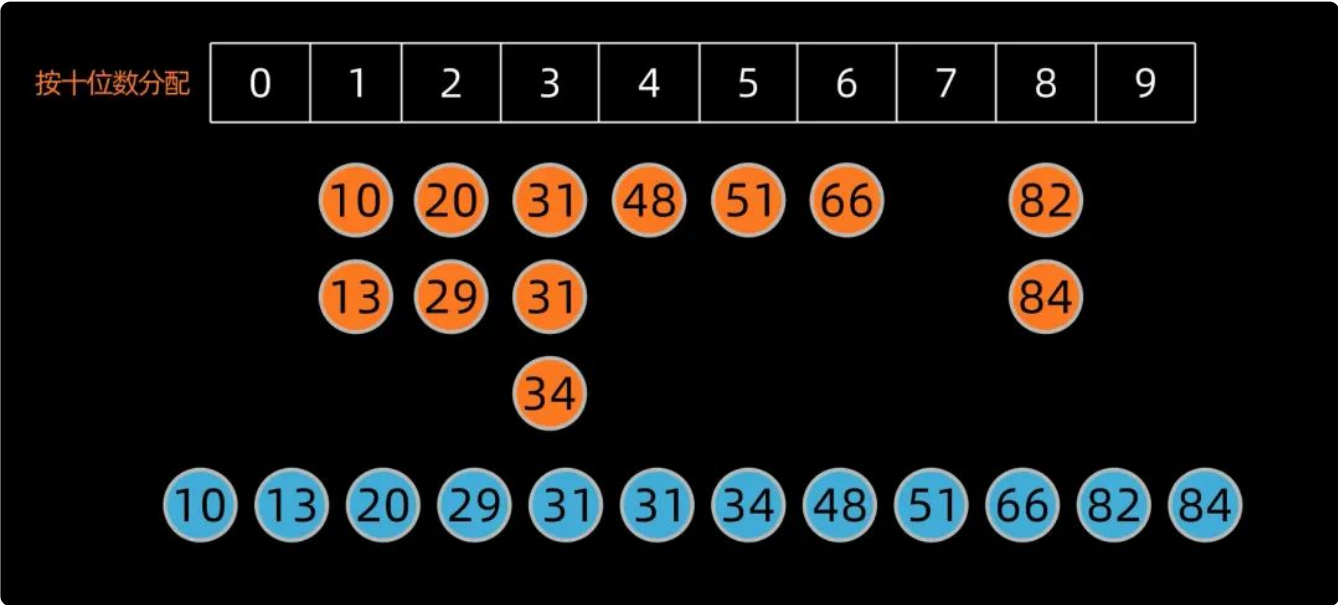
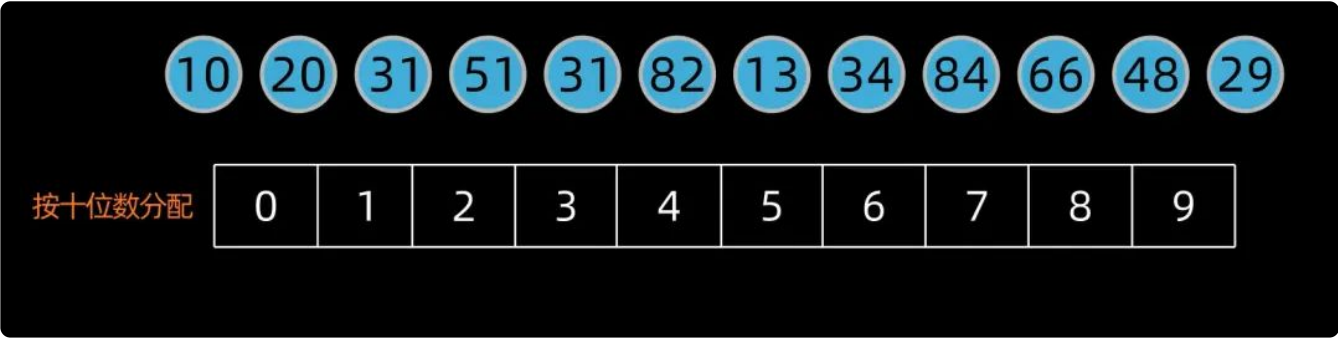
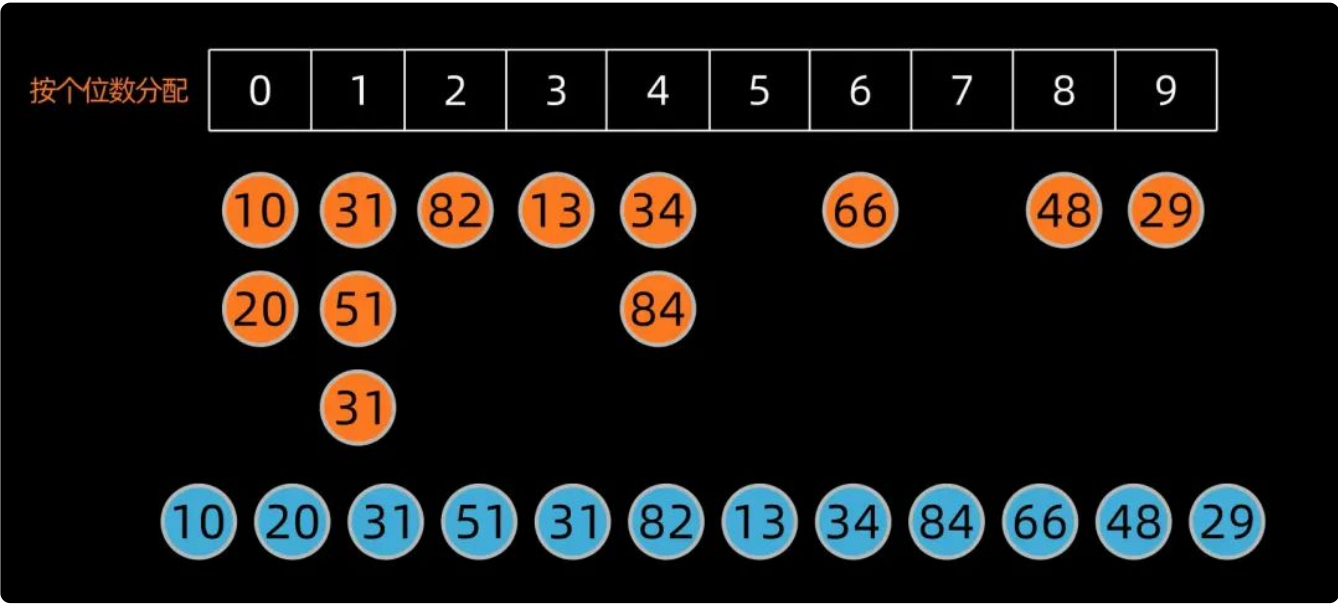
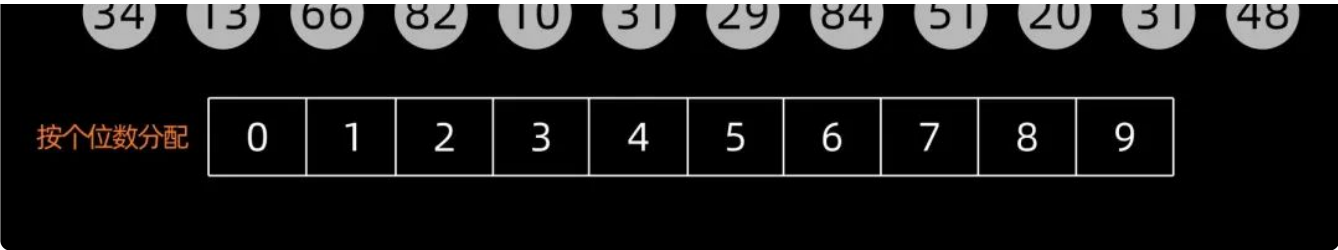
基数排序的方式可以采用最低位优先LSD（Least significant digital）法或最高位优先MSD（Most significant digital）法，LSD的排序方式由键值的最右边开始，而MSD则相反，由键值的最左边开始。

LSD的基数排序适用于位数小的数列，如果位数多的话，使用MSD的效率会比较好，MSD的方式恰与LSD相反，是由高位数为基底开始进行分配，其他的演算方式则都相同。

算法流程

这里以最低位优先LSD为例。

先根据个位数的数值，在扫描数值时将它们分配至编号0到9的桶中，然后将桶子中的数值串接起来。



如果排序的对象有三位数以上，则持续进行以上的动作直至最高位数为止。

Python代码

```
# LSD Radix Sort
def radix_sort(lst):
    mod = 10
    div = 1
    mostBit = len(str(max(lst)))
    buckets = [[] for row in range(mod)]
    while mostBit:
        for num in lst:
            buckets[num // div % mod].append(num)
        i = 0
        for bucket in buckets:
            while bucket:
                lst[i] = bucket.pop(0)
                i += 1
        div *= 10
        mostBit -= 1
    return lst
```

11 小结

以上就是用 Python 来实现10种经典排序算法的相关内容。

对于这些排序算法的实现，代码其实并不是最主要的，重要的是需要去理解各种算法的基本思想、基本原理以及其内部的实现过程。

对于每种算法，用其他编程语言同样是可以去实现的。

并且，对于同一种算法，即使只用 Python 语言，也有多种不同的代码方式可以实现，但其基本原理是一致的。

参考文档

- <https://www.jianshu.com/p/bbbab7fa77a2>

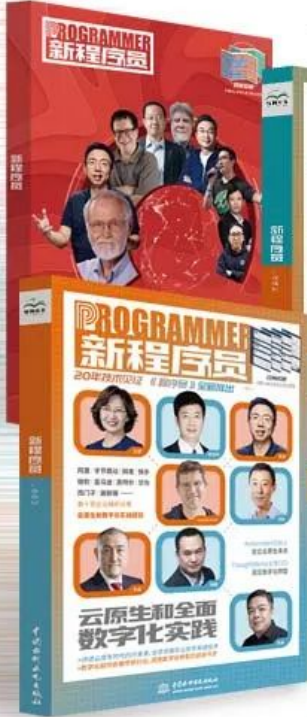
- <https://blog.csdn.net/MakerCloud/article/details/86182145>
- <https://www.cnblogs.com/mengdd/archive/2012/11/30/2796845.html>
- <https://www.cnblogs.com/jin-nuo/p/5293554.html>

THE END


《新程序员》001
开发者的黄金十年

《新程序员》002
新数据库时代&软件定义汽车

《新程序员》003
云原生和数字化实践



立即
订
阅



往 期 回 顾

技术

Pandas数据类型概述与转换实战

技术

Python版的故宫导游图，来袭

技术

快速实现Resnet残差模型实战

资讯

隐患：神经网络可以隐藏恶意软件

分享 点收藏

点点赞 点在看

喜欢此内容的人还喜欢

十大排序算法Python实现
ComputerVision SLAM Lab

听说 | 数据结构+ 算法
霖镜

十大排序算法C++實現
ComputerVision SLAM Lab