演算法 - 排序（最全讲解+Python代码实现？）

原创

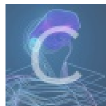
[壮壮不太胖^QwQ](#) 于 2020-03-24 22:21:31 发布 401 收藏 8

分类专栏： 演算法 文章标签： 排序算法 python

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/weixin_46072771/article/details/105074795

版权

 演算法 专栏收录该内容

10 篇文章 1 订阅

订阅专栏

文章目录

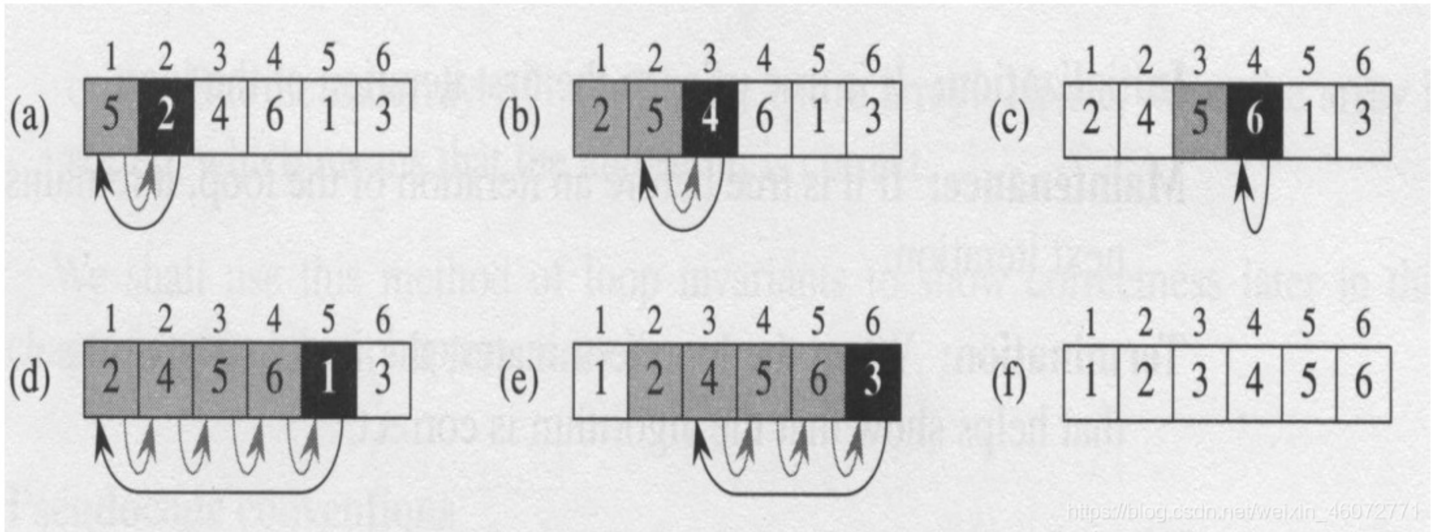
一，插入排序（Insertion-Sort）
1，直接插入排序
2，折半插入排序
3，希尔排序
二，交换排序
1，冒泡排序
2，快速排序（改进的交换排序）
三，选择排序（Selection Sort）
1，简单选择排序
2，什么是堆排序
堆的定义
堆排序的原理
堆的调整
堆的建立
3，堆排序的代码实现
4，算法分析
四，归并排序（Merge Sort）
五，基数排序（Radix sort）
六，计数排序（Counting sort）
七，桶排序（Bucket sort）
八，各种排序方法比较
1，时间性能
2，空间性能
3，排序方法的稳定性能
4，关于“排序方法的时间复杂度下线”

一，插入排序（Insertion-Sort）

1，直接插入排序

[视频素材 - bilibili]

工作原理



代码实现

```python
def Insertion_Sort(a):
    for i in range(len(a)):
        key = a[i]
        j = i - 1
        while i > 0 and a[i - 1] > key:
            a[i] = a[i - 1]
            i -= 1
        a[i] = key
    return a
```

```python
a = [2, 3, 5, 2, 1, 9, 7, 5, 2]
b = Insertion_Sort(a)
print(b)
>>>[1, 2, 2, 2, 3, 5, 5, 7, 9]
```

算法分析

Insertion-sort(A)	cost	times
1 **for** $j \leftarrow 2$ **to** length[A]	c_1	n
2 **do** key$\leftarrow$A[j]	c_2	$n-1$
3 ∗Insert $A[j]$ into the sorted sequence $A[1..j-1]$	0	
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i>0$ and $A[i]>$key	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum^{n} (t_j-1)$

7 $i \leftarrow i-1$ c_7 $\sum_{j=2}^{n}(t_j-1)$

8 $A[i+1] \leftarrow key$ c_8 $n-1$

https://blog.csdn.net/weixin_46072771

时间复杂度为

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

https://blog.csdn.net/weixin_46072771

最佳的情况：已经由小到大排好序
第5行 则只会被执行 n-1 次
第6,7行 则不会被执行

- The **best case** occurs if the array is already sorted
- $t_j = 1$, for j = 2,3,…,n

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

We can express this running time as an **an+b** for *constants* a and b that depend on the statement costs c_i;
T(n) is a ***linear function*** of n.

https://blog.csdn.net/weixin_46072771

最差情况：列表为逆序
第5行 对任何一个 a[i] 进行排序时 都会被执行 i 次（共有n个a[i]）
第6,7行 随同 行5，每一次排序会被执行 i-1 次（进行n次排序）

- $t_j = j$ for j = 2,3,…,n : quadratic function on n.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) +$$

$$c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1)$$

$$= (\frac{c_5 + c_6 + c_7}{2})n^2 - (c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

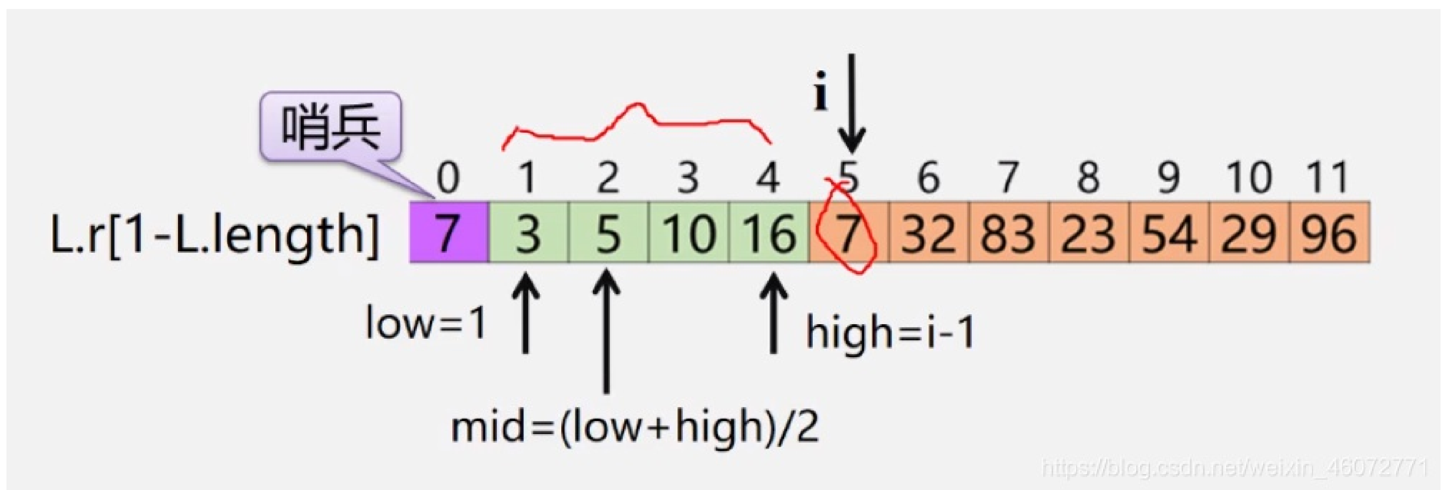
https://blog.csdn.net/weixin_46072771

时间复杂度【最好情况–最坏情况–平均情况】–空间复杂度–稳定性

直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定

2，折半插入排序

视频素材 - bilibili
工作原理



代码实现

```
def BinaryInsertSort(list):
    for i in range(2, len(list)):
        list[0] = list[i]
        low = 1
        high = i - 1
        while low <= high:
            m = int((low + high) / 2)  # 折半
            if list[0] < list[m]:  # 插入点在低半区
                high = m - 1
            else:  # 插入点在高半区
                low = m + 1

        j = i - 1  # 记录后移
        while j >= high + 1:
            list[j + 1] = list[j]
            j -= 1
        list[high + 1] = list[0]
```

其中[0]=-1这一位置是暂存单元，不会参与排序

```
a = [-1, 2, 3, 5, 2, 1, 9, 7, 5, 2]
BinaryInsertSort(a)
print(a)

>>>[2, 1, 2, 2, 2, 3, 5, 5, 7, 9]
```

算法分析

> 与直接插入排序法相比：
> 折半插入减少了比较次数，但没有减少移动次数（平均性能更优）
> 当数据量 n 较大时，且数据越乱，越适合用折半排序
> 而在最佳情况下时，直接排序（只用比较一次）反而优于折半排序

- 时间复杂度为 $O(n^2)$
- 空间复杂度为 $O(1)$
- 是一种稳定的排序方法

https://blog.csdn.net/weixin_46072771

3，希尔排序

视频素材 - bilibili
基本思想

先将整个待排记录序列分割成若干子序列，分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。

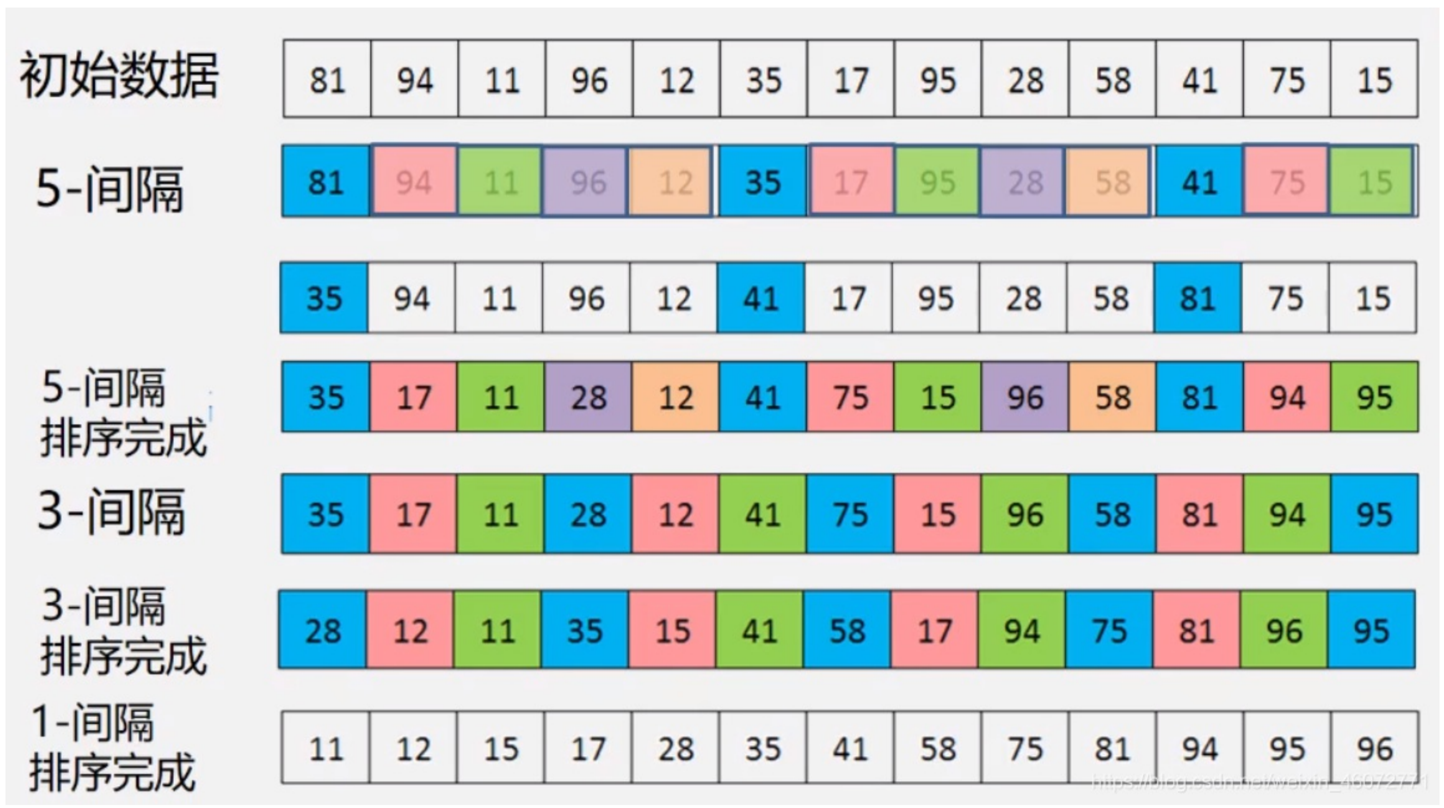
https://blog.csdn.net/weixin_46072771

希尔排序的算法特点：

- 缩小增量
- 多遍插入排序
- 一次移动，移动位置较大，跳跃式的接近排序后的最终位置
- 最后一次只需要少量移动
- 增量序列必须是递减的，最后一个必须是“1”
- 增量序列应该是互质的（互质是公约数只有1的两个整数，叫做互质整数。）

排序思路：

增量 5 > 3 > 1



1. 定义增量序列D_K： $D_M > D_{M-1} > \ldots > D_1 = 1$

   - 刚才的例子中：$D_3=5$，$D_2=3$，$D_1=1$

2. 对每个D_K进行“D_K-间隔”插入排序（k=M，M-1，...1）

https://blog.csdn.net/weixin_46072771

代码实现

```python
def shell_Sort(alist):
    sublistcount = len(alist) // 2  # 除2后取不超过结果的最大整数
    while sublistcount > 0:

        for startposition in range(sublistcount):
            InsertionSort(alist, startposition, sublistcount)

        print("增量：", sublistcount,
              "此时列表：", alist)

        sublistcount = sublistcount // 2  # 增量每次减小一半


def InsertionSort(alist, start, gap):
    """
    :param alist: 需要排序的列的表
    :param start: 开始位置
    :param gap: 增量
    :return: 排序后列表
    """
    for i in range(start + gap, len(alist), gap):
        currentvalue = alist[i]
        position = i

        while position >= gap and alist[position - gap] > currentvalue:
            alist[position] = alist[position - gap]
            position = position - gap

        alist[position] = currentvalue
```

```python
alist = [23, 45, 11, 67, 44, 98, 69, 57, 26, 58]
shell_Sort(alist)
print(alist)
>>>增量： 5 此时列表： [23, 45, 11, 26, 44, 98, 69, 57, 67, 58]
   增量： 2 此时列表： [11, 26, 23, 45, 44, 57, 67, 58, 69, 98]
   增量： 1 此时列表： [11, 23, 26, 44, 45, 57, 58, 67, 69, 98]
   最终结果：[11, 23, 26, 44, 45, 57, 58, 67, 69, 98]
```

代码理解

原列表：[23.45,11,67],44,98,69,57,26.58] len=10.

① shell_Sort( ):

从0位置依次遍历至mid位置.
增量初始为 mid = len/2，之后每次 mid = mid/2

② InsertionSort( ): 主函数.

https://blog.csdn.net/weixin_48072771

算法分析

希尔排序算法效率与**增量序列的取值**有关

时间效率↓

- Hibbard增量序列
  - $D_k=2^{k-1}$——相邻元素互质
  - 最坏情况：$T_{worst}=O(n^{3/2})$
  - 猜想：$T_{avg}=O(n^{5/4})$
- Sedgewick增量序列
  - {1,5,19,41,109,...}

    ——$9*4^i-9*2^i+1$或4^i-3*2^i+1
  - 猜想：$T_{avg}=O(n^{7/6})$ $T_{worst}=O(n^{4/3})$

https://blog.csdn.net/weixin_46072771

希尔排序是**不稳定**的排序算法

例如： $d=2$

3	5	10	8	7	2	8	1	20	6

3	1	7	2	8	5	10	6	20	8

相对位置发生改变

希尔排序是不稳定的

https://blog.csdn.net/weixin_46072771

总结

- 时间复杂度是n和d的函数：

  $O(n^{1.25})$ ～ $O(1.6n^{1.25})$ —经验公式
- 空间复杂度为 $O(1)$
- 是一种不稳定的排序方法

✓如何选择最佳d序列，目前尚未解决

✓最后一个增量值必须为1，无除了1之外的公因子

✓不宜在链式存储结构上实现

https://blog.csdn.net/weixin_46072771

二，交换排序

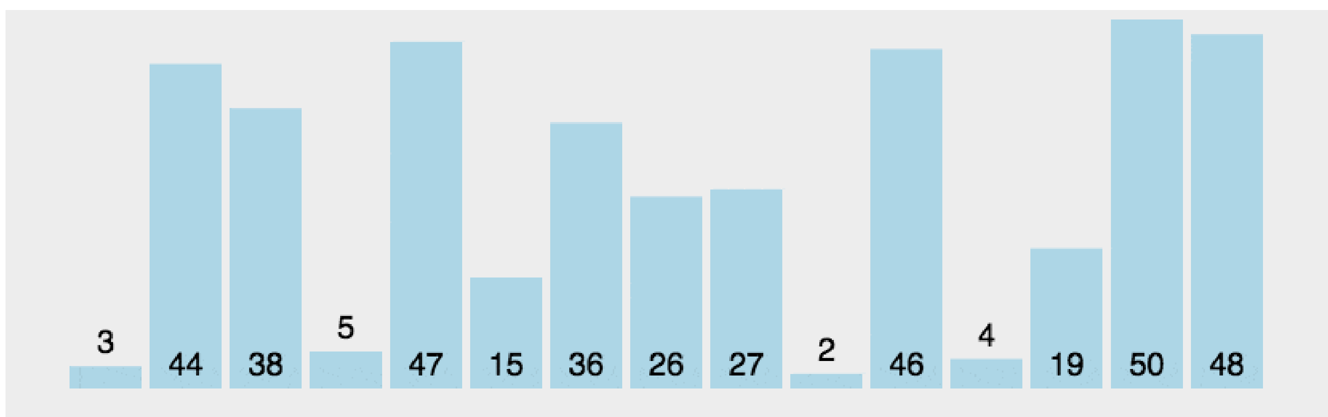
基本思想：
两两比较，如果发生逆序则交换，直到所有记录都排好序为止。
常见的交换排序：

- 冒泡排序 O(n^2)
  — 优点:比较简单，空间复杂度较低，是稳定的；
  — 缺点:时间复杂度太高，效率慢；
- 快速排序 O($nlog_2^n$)
  — 优点：一轮比较只需要换一次位置；
  — 缺点：效率慢，不稳定（举个例子5，8，5，2，9 我们知道第一遍选择第一个元素5会和2交换，那么原序列中2个5的相对位置前后顺序就破坏了）。

先来总结下他们的区别了（**划重点**）：

1. 冒泡排序是比较相邻位置的两个数，而选择排序是按顺序比较，找最大值或者最小值；
2. 冒泡排序每一轮比较后，位置不对都需要换位置，选择排序每一轮比较都只需要换一次位置；
3. 冒泡排序是通过数去找位置，选择排序是给定位置去找数；

1，冒泡排序



基于简单交换思想：
每趟不断将记录两两比较，并按照“前小后大”规则排序

举个例子：
初始：[21, 25, 49, 25*, 16, 08] n = 6

- 第一趟
- 位置0，1 进行比较 —— 判断 —— 不交换 —— 结果：21，25，49，25*，16，08
- 位置1，2 进行比较 —— 判断 —— 不交换 —— 结果：21，25，49，25*，16，08
- 位置2，3 进行比较 —— 判断 —— 交换 —— 结果：21，25，25*，49，16，08
- 位置3，4 进行比较 —— 判断 —— 交换 —— 结果：21，25，25*，16，49，08
- 位置4，5 进行比较 —— 判断 —— 交换 —— 结果：21，25，25*，16，08，49

比较 5 次
第1趟，结束后：[21, 25, 25*, 16, 08, **49**]

- 第2趟:
  - 位置0，1 进行比较 —— 判断 —— 不交换 —— 结果：21，25，25*，16，08，49
  - 位置1，2 进行比较 —— 判断 —— 不交换 —— 结果：21，25，25*，16，08，49
  - 位置2，3 进行比较 —— 判断 —— 交换 —— 结果：21，25，16，25*，08，49
  - 位置3，4 进行比较 —— 判断 —— 交换 —— 结果：21，25，16，08，25*，49

比较 4 次
第2趟，结束后：[21, 25, 16, 08, **25***, **49**]
之后以此类推…
第3趟，结束后：[21, 16, 08, **25**, **25***, **49**] 比较3次
第4趟，结束后：[16, 08, **21**, **25**, **25***, **49**] 比较2次
第5趟，结束后：[08, **16**, **21**, **25**, **25***, **49**] 比较1次

总结：

- n 个记录，总共需要 n-1 **趟**
- 第 m 趟需要比较 n-m **次**

代码实现
传送门：菜鸟教程 - 冒泡排序

```
（非最优版本）
def bubbleSort(arr):
    n = len(arr)

    # 遍历所有数组元素
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n - i - 1):

            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

优点：每趟结束时，不仅能挤出一个最大值到最后位置，还能同时部分理顺其他元素

那么还有没有优化空间呢？
有的，若果再一趟中，完全没有发生交换，就可以说是已经排列好，所以可以加一个 变量change **判断是否发生过交换**，若没有则 break

```
（优化版本，增加了change判断变量）
def bubbleSort(arr):
    n = len(arr)

    # 遍历所有数组元素
    for i in range(n):
        change = False

        # Last i elements are already in place
        for j in range(0, n - i - 1):

            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                change = True  # 发生交换

        # 如果未发生交换则跳出回圈
        if not change:
            break
```

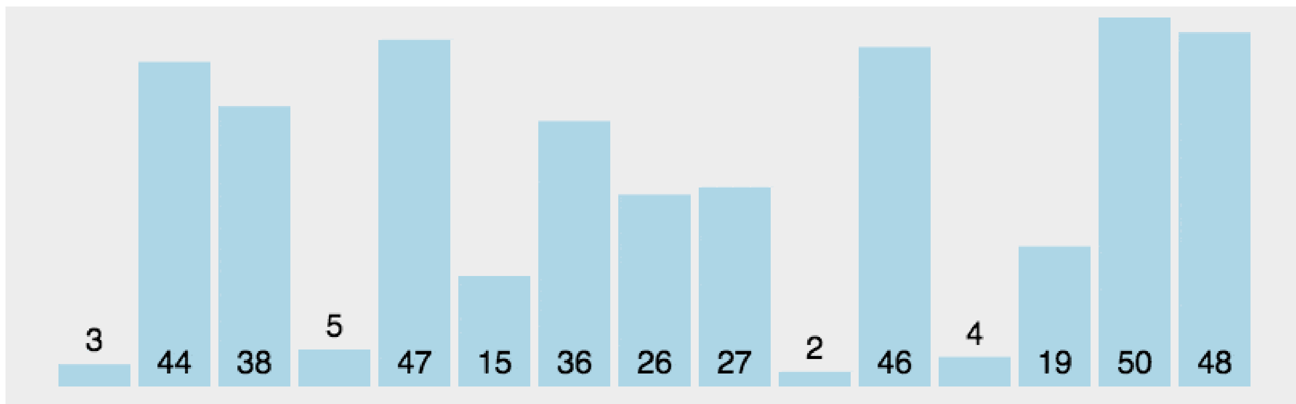
算法分析
时间复杂度

- 最优情况（正序）
  比较次数：n-1
  移动次数：0
- 最坏情况（逆序）
  比较次数：$\Sigma(n-i) = (1/2)(n^2 - n)$ 其中 i ⟵[1, n-1]
  移动次数：$3\Sigma(n-i) = (3/2)(n^2 - n)$ 其中 i ⟵[1, n]

- 冒泡排序**最好**时间复杂度是$O(n)$
- 冒泡排序**最坏**时间复杂度为$O(n^2)$
- 冒泡排序**平均**时间复杂度为$O(n^2)$
- 冒泡排序算法中增加一个辅助空间temp，辅助空间为$S(n)=O(1)$
- 冒泡排序是**稳定**的

https://blog.csdn.net/weixin_46072771

2，快速排序（改进的交换排序）



视频素材 - bilibili

基本思想（使用递归）：

> 任取一个元素（如：第一个）**为中心**（pivot，**枢轴**）
>
> 所有比他**小**的元素一律前放，比他大的元素一律后放，形成左右两个子表
> -（【小】（pivot）【大】）
>
> 对各子表中心选择中心元素并**依此规则调整**
>
> 直到每个子表的元素**只剩**一个

快速排序演示

https://blog.csdn.net/weixin_46072771

每个子表的形成都是采用从两头向中间交替式逼近法；
由于每趟中对各个子表的操作都相似，可采用**递归算法**。

代码实现
传送门：菜鸟教程 - 快速排序

```
def partition(arr, low, high):
    i = (low - 1)  # 最小元素索引
    pivot = arr[high]

    for j in range(low, high):

        # 当前元素小于或等于 pivot
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)


# arr[] --> 排序数组
# low  --> 起始索引
# high  --> 结束索引

# 快速排序函数
def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)

        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

这边要**注意**一下 n 的取值

```
n = len(list) - 1
quickSort(list, 0, n)
print(list)
>>>[2, 5, 66, 99, 77, 52, 33, 2, 1, 0, 44, 5, 10]
```

算法分析
时间复杂度

- 可以证明，平均计算时间是O($n\log_2 n$)。
  - Qsort( ): O($\log_2 n$)
  - Partition( ): O(n)
- 实验结果表明：就平均计算时间而言，快速排序是我们所讨论的所有内排序方法中最好的一个。

https://blog.csdn.net/weixin_46072771

空间复杂度

快速排序不是原地排序

由于程序中使用了递归，需要递归调用栈的支持，而栈的长度取决于递归调用的深度。（即使不用递归，也需要用用户栈）

- 在平均情况下：需要O(logn)的栈空间
- 最坏情况下：栈空间可达O(n)。

https://blog.csdn.net/weixin_46072771

稳定性

快速排序是一种不稳定的排序方法。

例如：49，38，49*，20，97，76

一次划分后：20，38，49*，49，97，76

总结

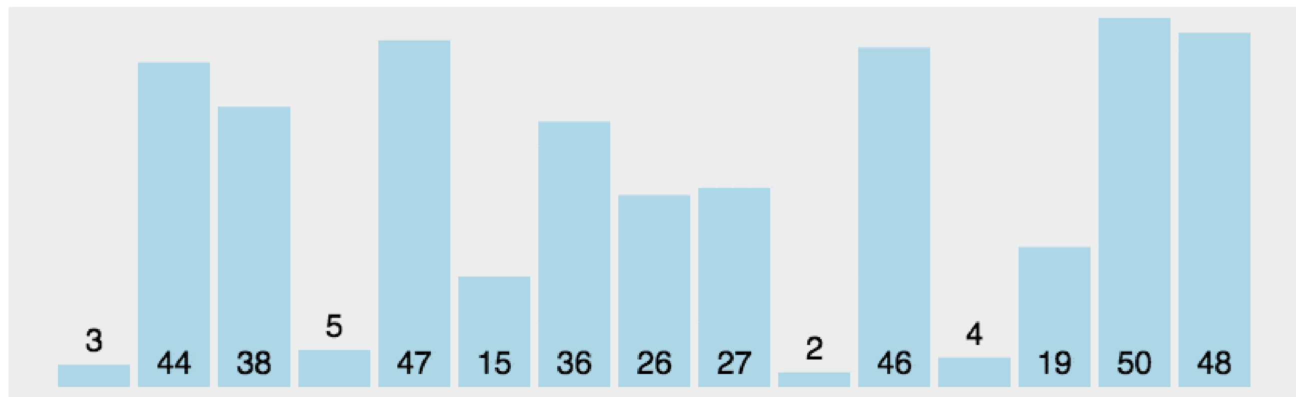
- 划分元素的选取是影响时间性能的关键
- 输入数据次序越乱，所选划分元素值的随机性越好，排序速度越快，快速排序不是自然排序方法。
- 改变划分元素的选取方法，至多只能改变算法平均情况的下的世界性能，无法改变最坏情况下的时间性能。即最坏情况下，快速排序的时间复杂性总是O(n^2)

https://blog.csdn.net/weixin_46072771

三，选择排序（Selection Sort）

– （演算法 2，Sorting.pdf ）

1，简单选择排序



视频素材 - bilibili

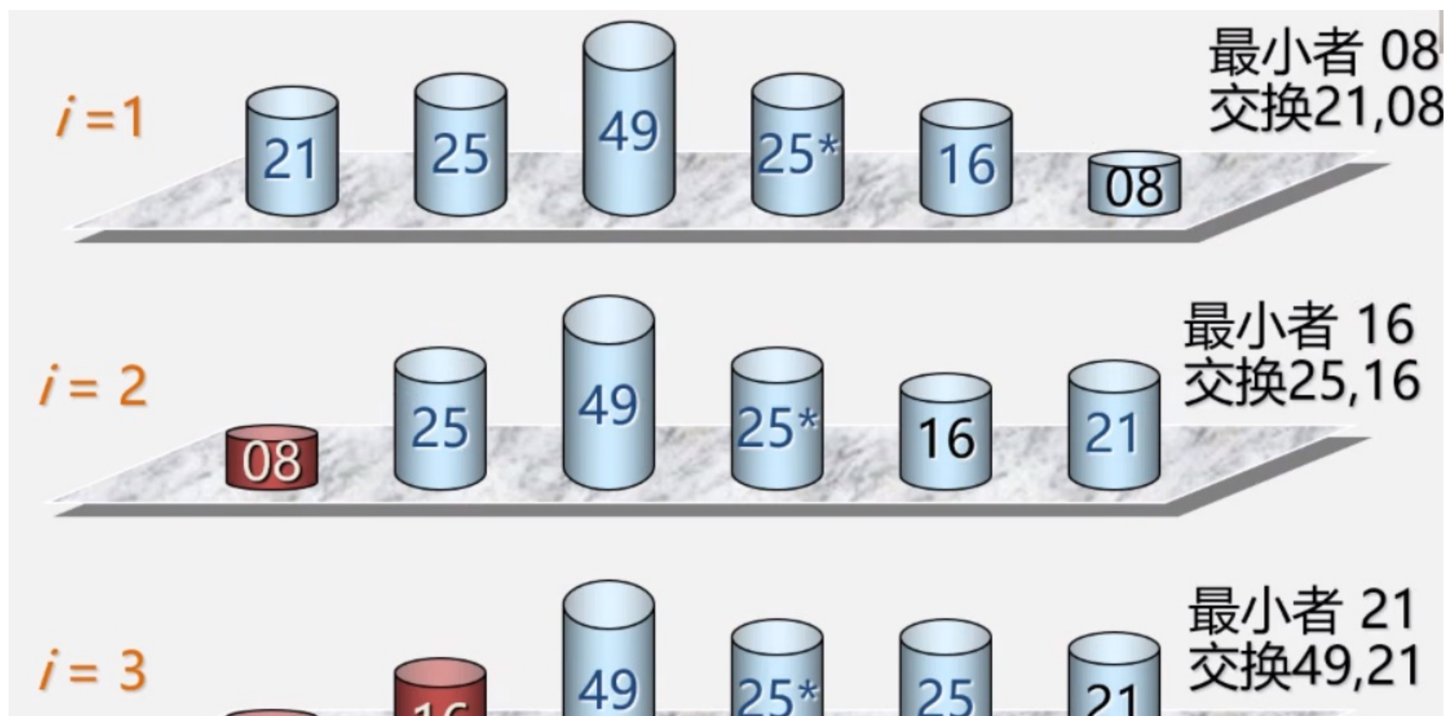
算法原理

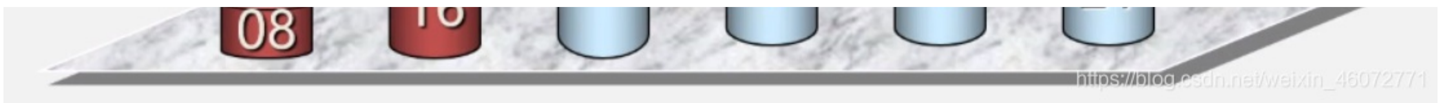
基本操作：

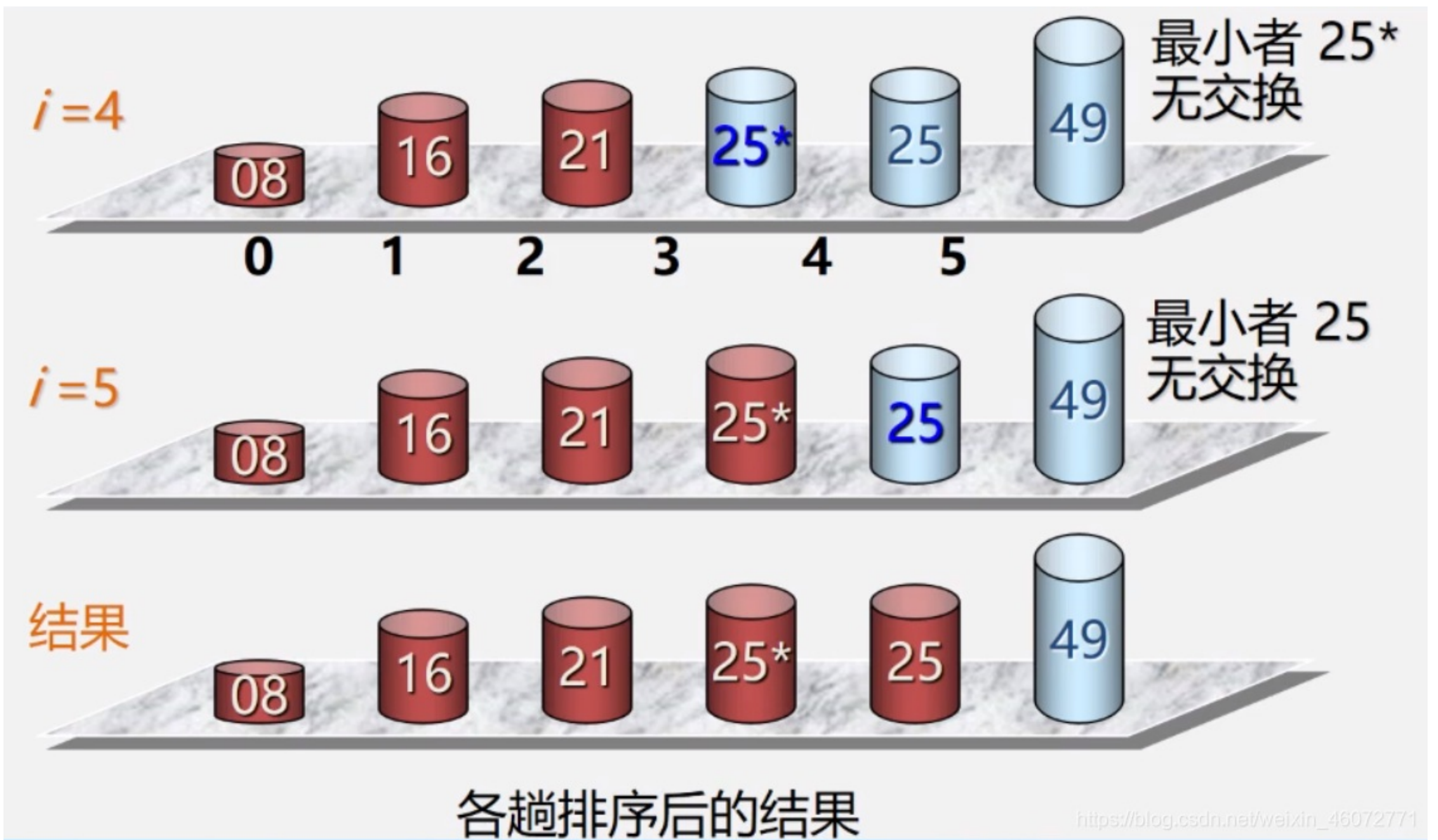
1. 首先通过n-1次关键字比较，从n个记录中找出关键字最小的记录，将它与第一个记录交换
2. 再通过n-2次比较，从剩余的n-1个记录中找出关键字次小的记录，将它与第二个记录交换
3. 重复上述操作，共进行n-1趟排序后，排序结束

https://blog.csdn.net/weixin_46072771

精髓在于：找最小值（要注意每一次找最小值的范围）







代码实现

```
def Normal_Select_Sort(a):
    for i in range(len(a)):
        # 找最小值的位置（index）
        key = i
        for j in range(i, len(a)):
            if a[key] > a[j]:
                key = j
        # 将最小值与a[i]进行交换
        if key != i:
            key_value = a[key]
            a[key] = a[i]
            a[i] = key_value
```

```
Normal_Select_Sort(list)
print(list)
>>>[0, 1, 2, 2, 5, 5, 10, 33, 44, 52, 66, 77, 99]
```

算法分析

时间复杂度

- 记录移动次数
  - ✡最好情况：0
  - ✡最坏情况：3(n-1)
- 比较次数：无论待排序列处于什么状态，选择排序所需进行的"比较" 次数都相同

$$\sum_{i=1}^{n-1}(n-i)=\frac{n}{2}(n-1)$$

算法稳定性

- 简单选择排序是不稳定排序

例：8 5 8* 7 9

第1次：5 8 8* 7 9

第2次：5 7 8* 8 9

https://blog.csdn.net/weixin_46072771

时间复杂度【最好情况–最坏情况–平均情况】–空间复杂度–稳定性

直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定

2，什么是堆排序

@五分钟学算法之堆排序

堆的定义

视频素材 - bilibili

堆的定义：

若n个元素的序列$\{a_1\ \ a_2\ \ldots\ a_n\}$满足

$$\begin{cases} a_i \le a_{2i} \\ a_i \le a_{2i+1} \end{cases} \quad 或 \quad \begin{cases} a_i \ge a_{2i} \\ a_i \ge a_{2i+1} \end{cases}$$

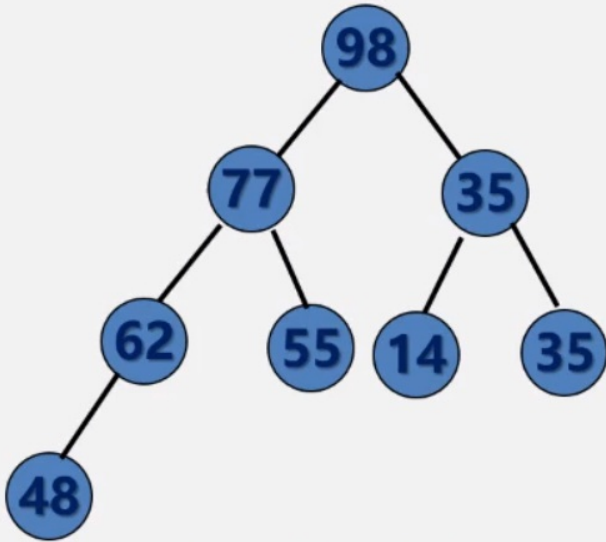
则分别称该序列$\{a_1\ \ a_2\ \ldots\ a_n\}$为小根堆和大根堆。

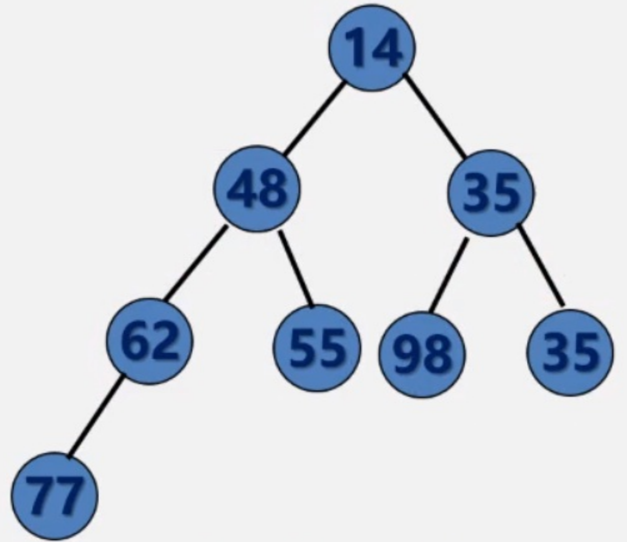
（堆实质是满足如下性质的**完全二叉树**：*二叉树中任意非叶子节点均小于（大于）它的孩子节点*）

光看定义肯定会比较晕，那么来看一下下边的例子吧~

{ 98 77 35 62 55 14 35 48 }

{ 14 48 35 62 55 98 35 77 }

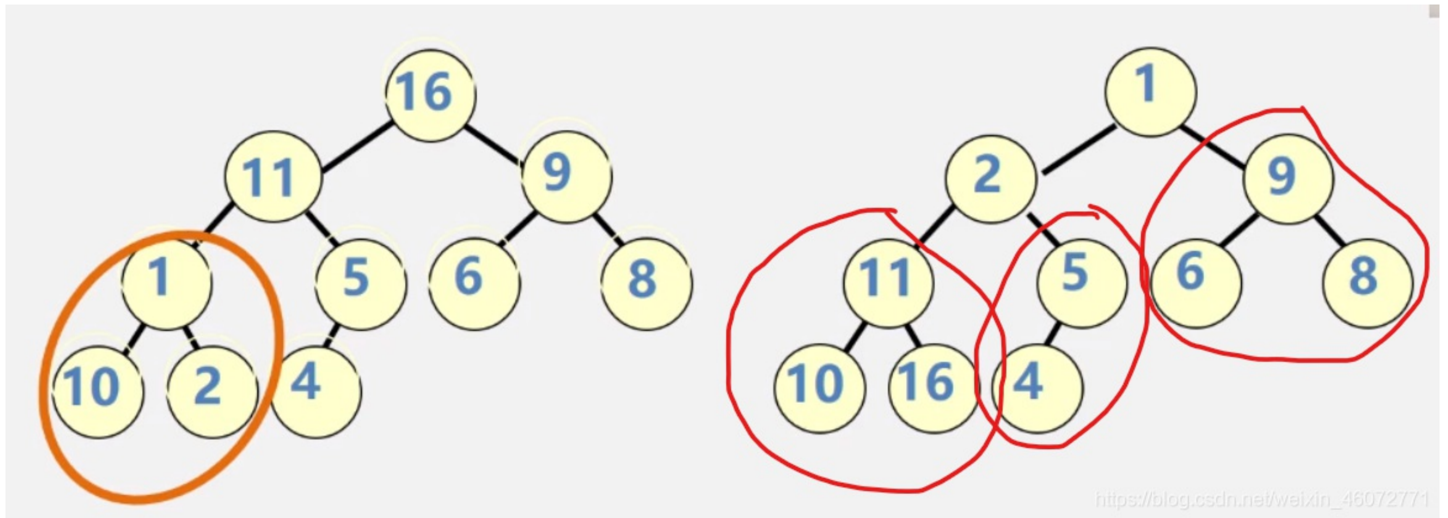




是一个大根堆 是一个小根堆

https://blog.csdn.net/weixin_46072771

再来看两个错误的例子



https://blog.csdn.net/weixin_46072771

堆排序的原理

若在输出堆顶的最小值（最大值）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（次大值）......如此反复，便能得到一个有序序列，这个过程称之为堆排序。

https://blog.csdn.net/weixin_46072771

实现堆排序需要解决的两个问题：

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后，调整剩余元素为新的堆。

堆的调整

视频素材 - bilibili

下边不理解的话，上边视频从3:50开始看一下

小根堆:

1. 输出堆顶元素之后，以堆中最后一个元素替代之；
2. 然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；
3. 重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”

https://blog.csdn.net/weixin_46072771

（大根堆 交换大者）

堆的建立

视频素材 - bilibili

通过反复的筛选，将一个无序的序列建成一个堆

单结点的二叉树是堆；

在完全二叉树中所有以叶子结点（序号i > n/2）为根的子树是堆。

这样，我们只需依次将以序号为n/2，n/2 - 1，……，1的结点为根的子树均调整为堆即可。

https://blog.csdn.net/weixin_46072771

具体的操作：

从最后一个非叶子结点开始，以此向前调整：

① 调整从第n/2个元素开始，将以该元素为根的二叉树调整为堆

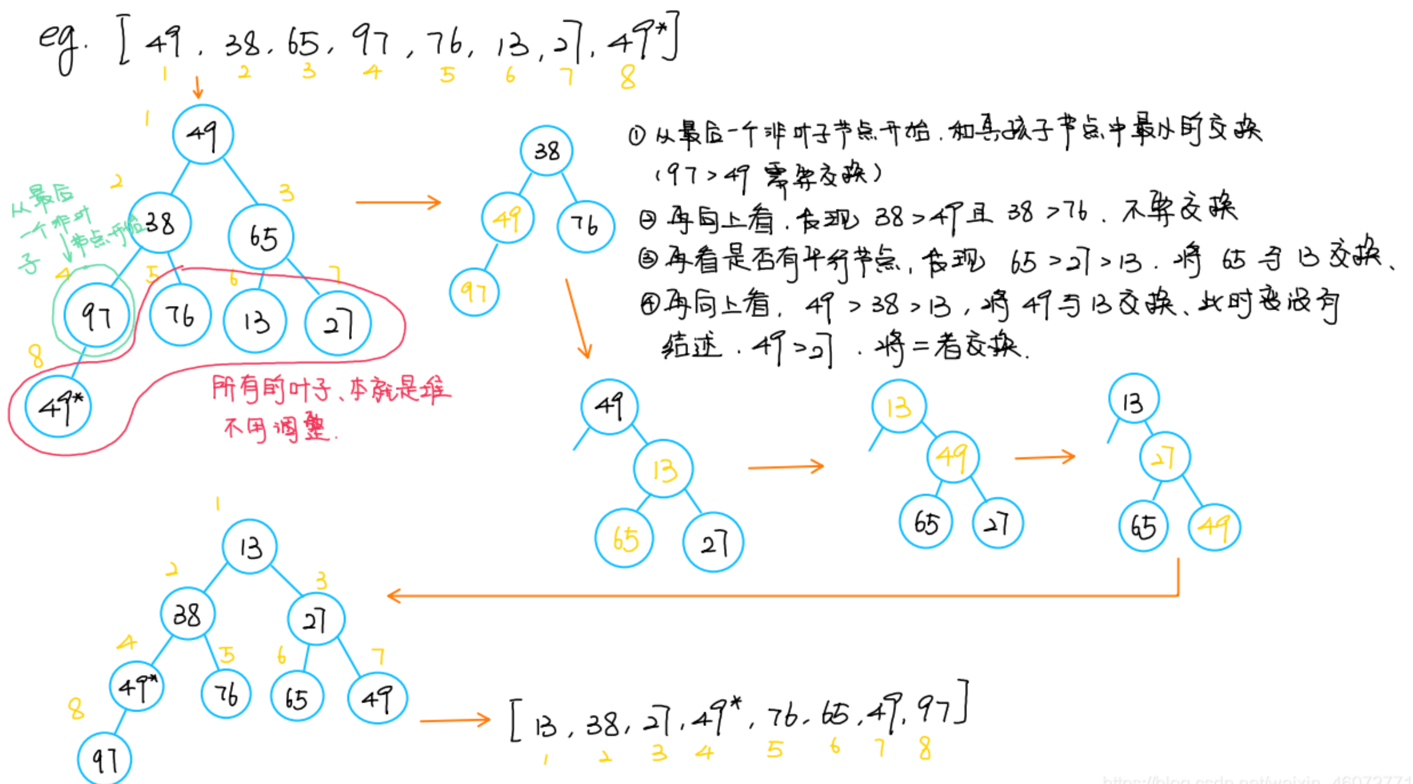
② 将以序号为n/2 - 1的结点为根的二叉树调整为堆；

③ 再将以序号为n/2 - 2的结点为根的二叉树调整为堆；

④ 再将以序号为n/2 - 3的结点为根的二叉树调整为堆；

https://blog.csdn.net/weixin_46072771

如果还是一脸懵逼就来看一下这个例子吧~



3，堆排序的代码实现

视频素材 - bilibili

~~还没学二叉树，所以下边的代码我自己看也是云里雾里，补完二叉树会回来进行更改和补充的（希望早日给这两行加上删除线）~~

传送门：演算法 - 树

```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1  # left = 2*i + 1
    r = 2 * i + 2  # right = 2*i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # 交换

        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)

    # 建立一个最大堆
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # 一个个交换元素
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # 交换
        heapify(arr, i, 0)
```

```python
heapSort(list)
print(list)
>>>[0, 1, 2, 2, 5, 5, 10, 33, 44, 52, 66, 77, 99]
```

4，算法分析

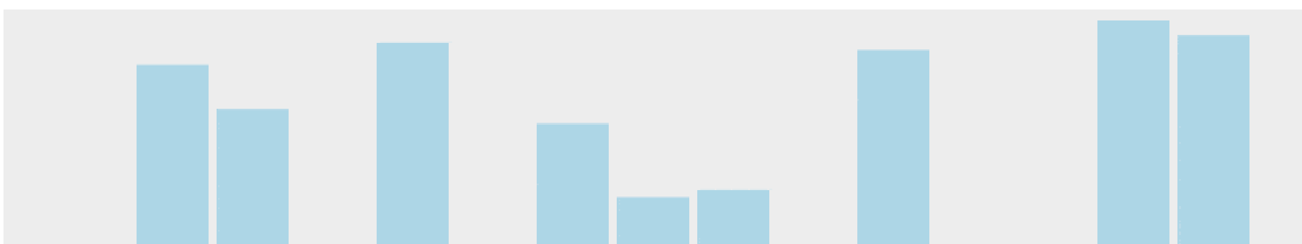
堆排序的时间主要耗费在建初始堆和调整建新堆时的反复及筛选上。
对排序的**最大优点**即是：最坏和最好的情况下，时间复杂度均为↓

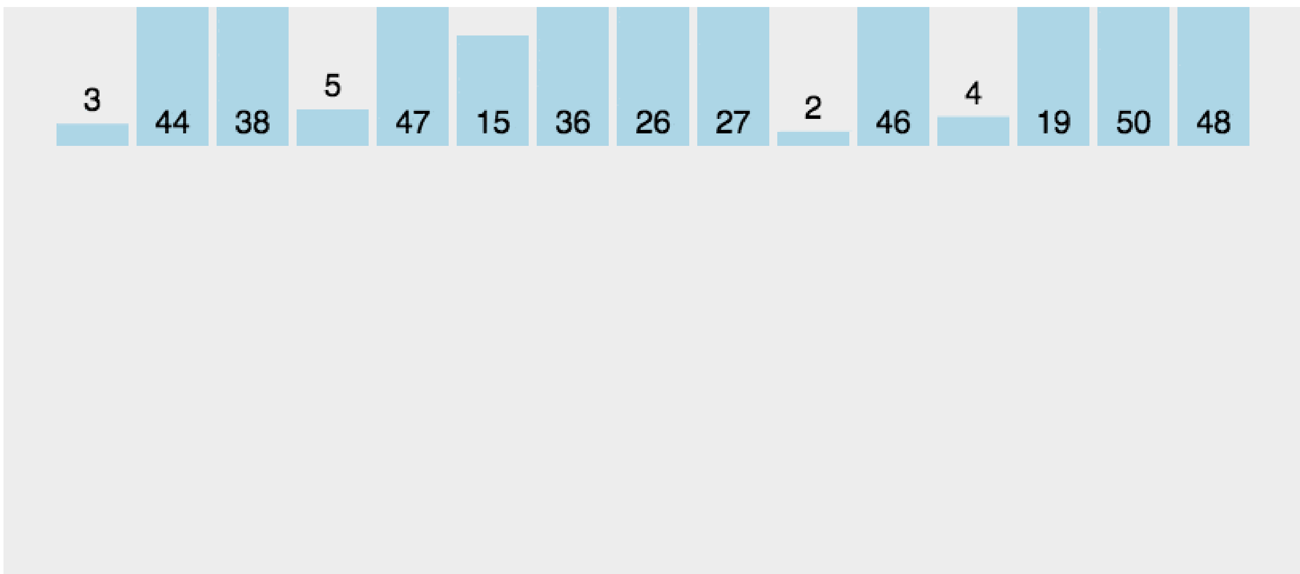
$$O(n\log_2 n)$$

无论待排序列中的记录是正序还是逆序，都不会使得堆排序处于“最好”或“最坏”的状态。
另外，堆排序仅需一个记录大小供交换用的辅助存储空间。
然而，堆排序是一种不稳定的排序方法，不适用于排序记录个数 n 较小的情况，但对于 n 较大的文件还是很有效的。

四，归并排序（Merge Sort）

视频素材 - bilibili
工作原理

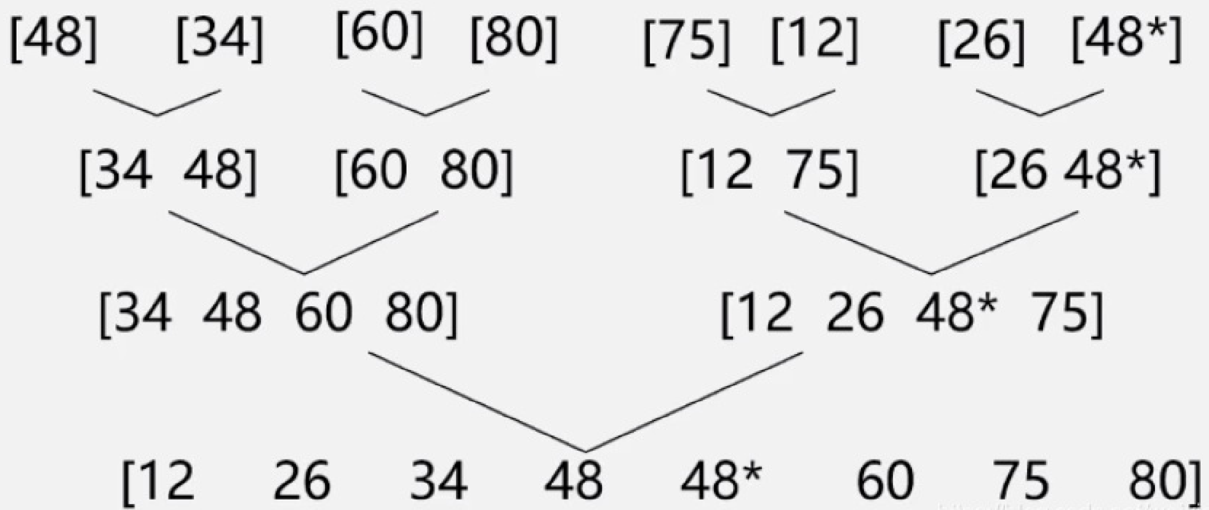


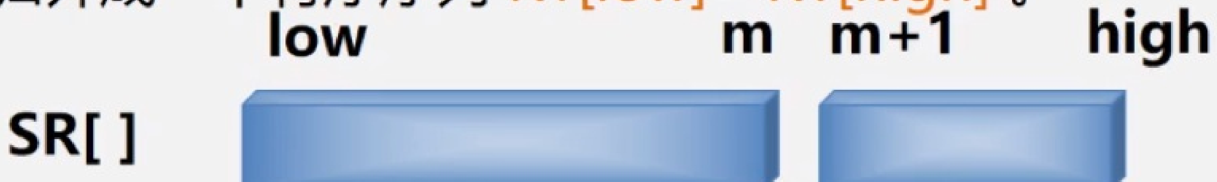


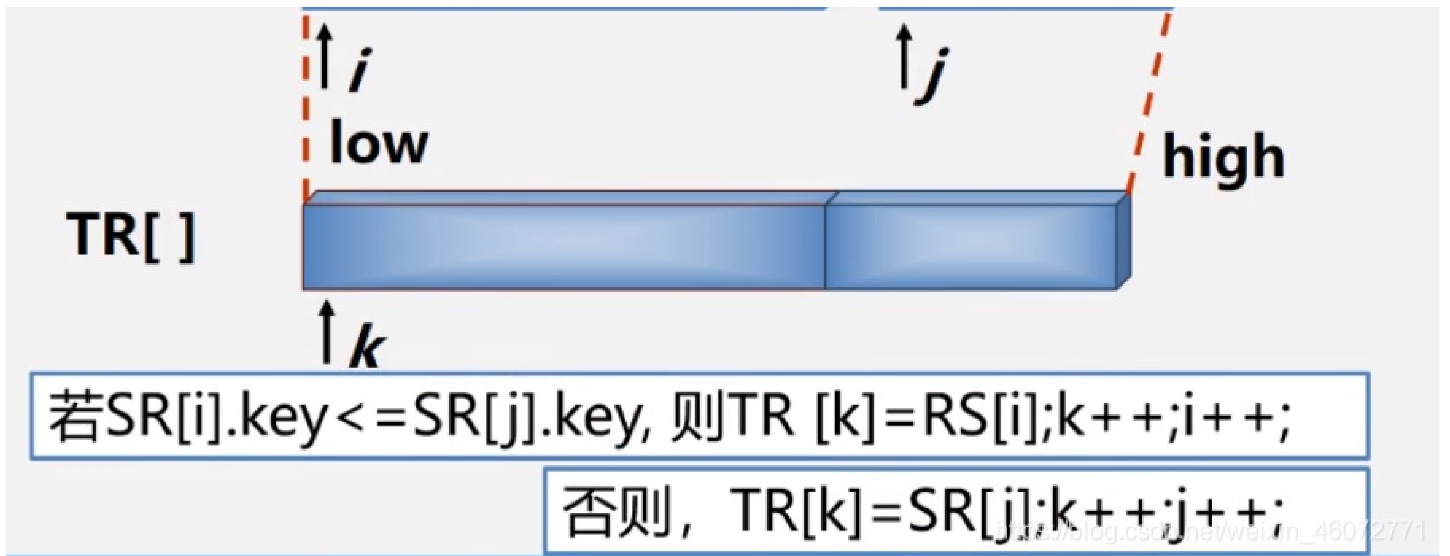
将两个有序或两个以上**有序的**子序列，归并为一个有序序列

通常采用 2-路归并排序

即：将两个相邻的有序子序列 L[1 … n1+1] & R[1 … n2+1]，归并为一个序列







代码实现

参考文献 — 讲的很详细

```python
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # 创建临时数组
    L = [0] * (n1)
    R = [0] * (n2)

    # 拷贝数据到临时数组 arrays L[] 和 R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

        # 归并临时数组到 arr[l..r]
    i = 0  # 初始化第一个子数组的索引
    j = 0  # 初始化第二个子数组的索引
    k = l  # 初始归并子数组的索引

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # 拷贝 L[] 的保留元素
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # 拷贝 R[] 的保留元素
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# 进行递归
def mergeSort(arr, start_index, end_index):
    if start_index < end_index:
        mid = int((start_index + (end_index - 1)) / 2)

        mergeSort(arr, start_index, mid)
        mergeSort(arr, mid + 1, end_index)

        merge(arr, start_index, mid, end_index)
```

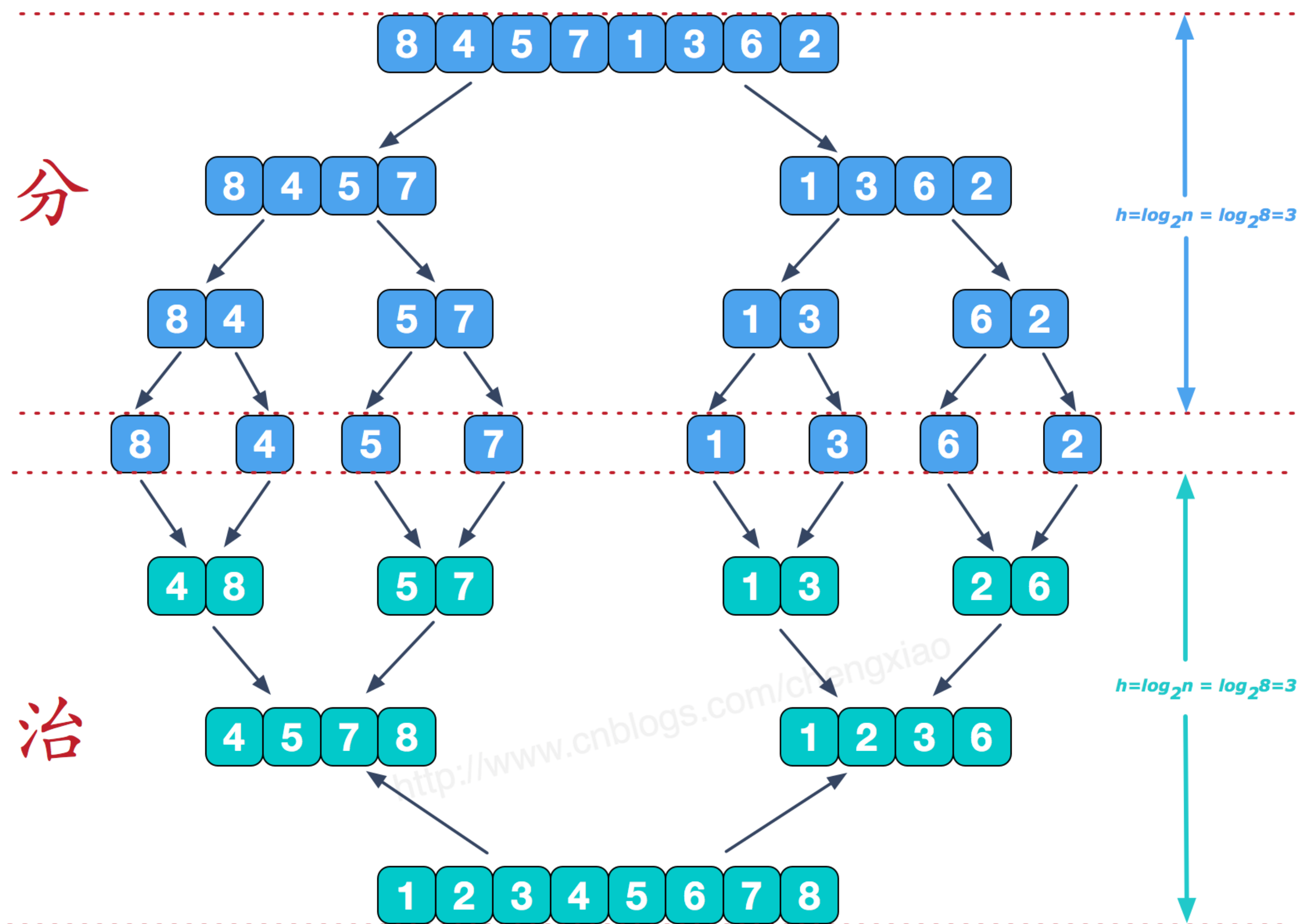
```python
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("给定的数组", arr)

mergeSort(arr, 0, n - 1)
print("排序后的数组", arr)
>>>给定的数组 [12, 11, 13, 5, 6, 7]
   排序后的数组 [5, 6, 7, 11, 12, 13]
```

算法分析

- 时间复杂度：O(nlog^{n})
- 空间复杂度：O(n)
- 稳定性：稳定

缺点：需要一个与原序列同样大小的辅助序列（L，R）

详细分析
传送门：分治法（Divide-and-Conquer）
归并排序中涉及到的分治思想（分解 - 排序 - 合并）



- Analyzing divide-and-conquer algorithms

$$T(n) = \begin{cases} \Theta(1) & if \quad n \le c \\ aT(n/b) + D(n) + c(n) & otherwise \end{cases}$$

See Chapter 4

- Analysis of merge sort

– Analysis of merge sort

$$T(n)=\begin{cases}\Theta(1) & if \quad n=1\\ 2T(n/2)+\Theta(n) & if \quad n>1\end{cases}$$

$$T(n)=\Theta(n\log n)$$

https://blog.csdn.net/weixin_46072771

也可以写作

$$T(n)=\begin{cases}c & if\, n=1\\ 2T(n/2)+cn & if\, n>1\end{cases}$$

其中常数c表示时间
需要解决大小为1的问题以及除法和合并步骤中每个数组元素的时间。

https://blog.csdn.net/weixin_46072771

五，基数排序（Radix sort）

视频素材 - bilibili基本思想：分配 + 收集
也叫**桶排序或箱排序**：设置若干个箱子，将关键字为 k 的记录放入第 k 个箱子，然后再按序号将非空的链接。

基数排序：数字是有范围的，均由0-9这十个数字组成，则只需要设置十个箱子，相继按照个、十、百...进行排序。

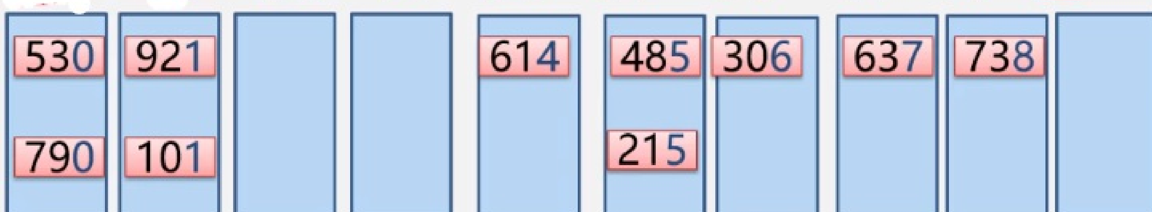
举个例子说明一下：

例：(614，738，921，485，637，101，215，530，790，306)

614 738 921 485 637 101 215 530 790 306

第一趟分配(按个位排)

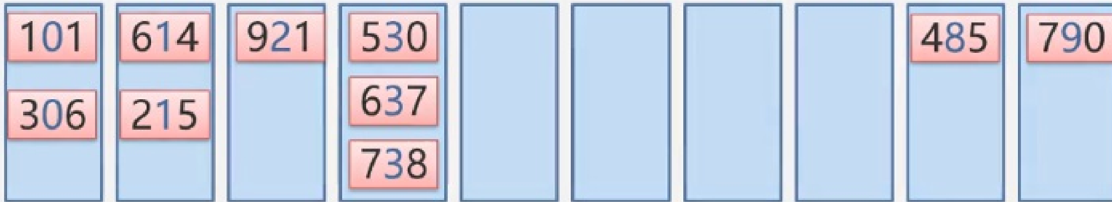
e[0] e[1] e[2] e[3] e[4] e[5] e[6] e[7] e[8] e[9]

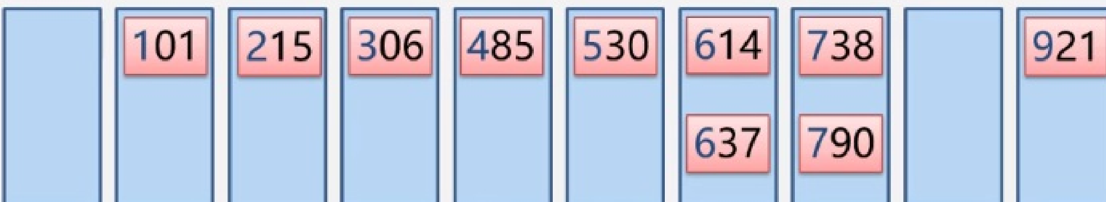


第一趟收集

530 790 921 101 614 485 215 306 637 738

第一趟的收集结果就已经将**个位**进行了排序，再在此结果的基础下进行第二趟。





（排序完成！）

算法分析

时间效率：O(k*(n+m)) - 线性阶
k：关键字个数（有几类桶）
m：关键字取值范围为m个值（桶的个数）
（进行k趟，每趟收集m次）

再举个例子：10000人按照生日排序
年（89个桶），月（12个桶），日（31个桶）

年 月 日 3个关键字

假设取值范围分别是：1930~2018 1~12 1~31

$O(n^2) \approx 10^8$

$O(n\log n) \approx 10^5$

$O(k*(n+m)) \approx (10000+89)+(10000+12)+(10000+31)$

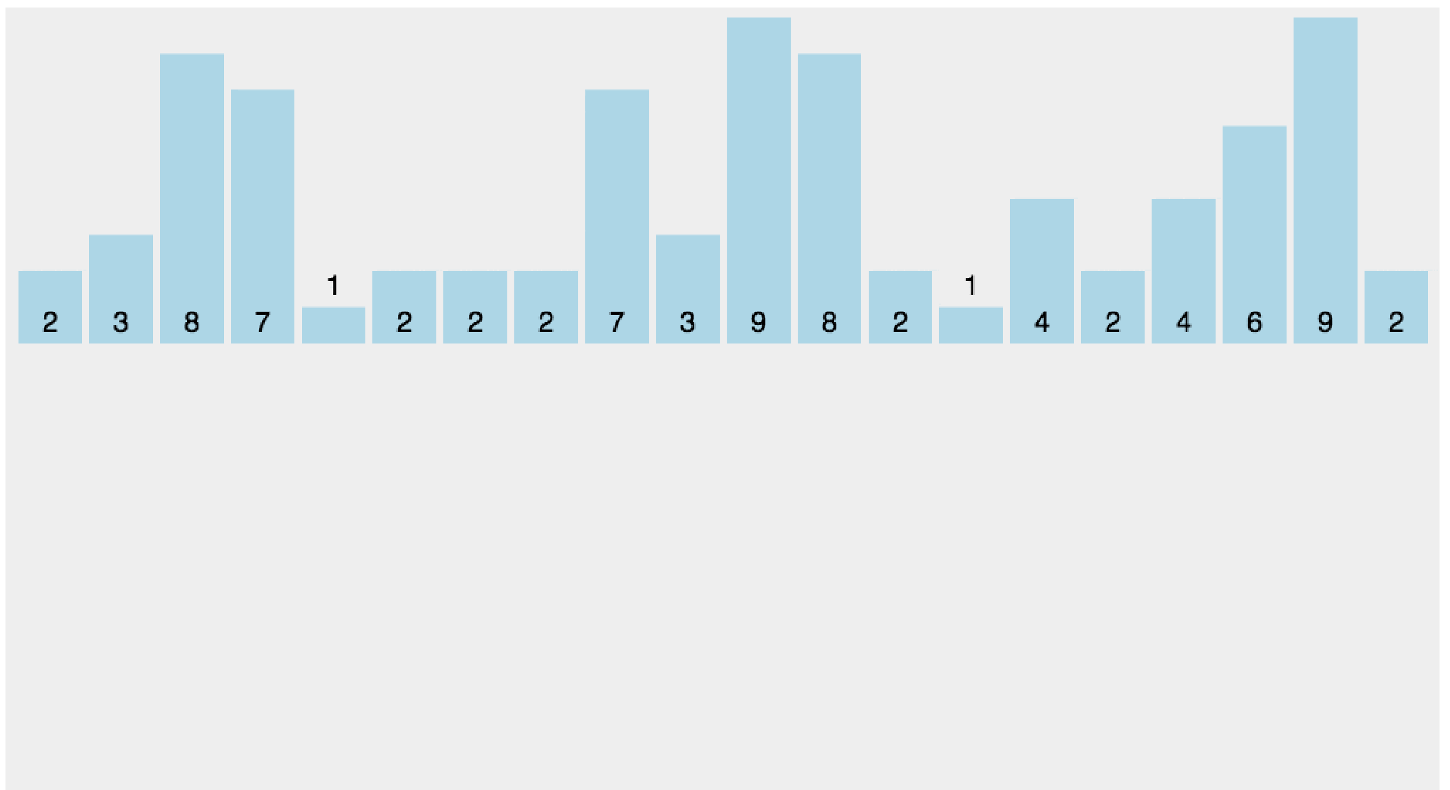
$\approx 10^4$

https://blog.csdn.net/weixin_46072771

六，计数排序（Counting sort）

计数排序的思想：

- 记录每一个数字出现过得次数，再进行排序



计数排序适用于：数据量大，但数据的取值（域）较小
例：

- **给全校学生成绩排序（学生很多，但成绩只属于[0, 100]之间）**

代码实现

```python
def CountingSort(arr):
    # 检查入参类型
    if not isinstance(arr, (list)):
        raise TypeError('error para type')

    # 获取arr中的最大值和最小值
    maxNum = max(arr)
    minNum = min(arr)

    # 以最大值和最小值的差作为中间数组的长度,并构建中间数组，初始化为0
    length = maxNum - minNum + 1
    tempArr = [0 for i in range(length)]

    # 创建结果List，存放排序完成的结果
    resArr = list(range(len(arr)))

    """
    索引 + minNum：数据
    tempArr[索引]：该数据出现次数
    统计每一个数据出现的次数"""
    for num in arr:
        tempArr[num - minNum] += 1

    """
    根据中间数组统计出的个数，一次加入结果List"""
    j = 0
    for i in range(len(tempArr)):
        while tempArr[i] > 0:
            resArr[j] = i + minNum
            j += 1
            tempArr[i] -= 1
    return resArr
```

```python
arr = [12, 25, -1, 13, 14, 25, 12, -1, 18, 14, -22, -13, -13]
print(CountingSort(arr))
>>>[-22, -13, -13, -1, -1, 12, 12, 13, 14, 14, 18, 25, 25]
```

空间复杂度和时间复杂度

与n个待排序数据，其中最大值和最小值的差值为m

- 时间复杂度：O(n+m)
- 空间复杂度：O(m)
- 稳定性：稳定

局限性

- 当数列的最大和最小值差距过大时，并不适用计数排序
- 当数列元素不是整数，并不适用计数排序

七，桶排序（Bucket sort）

参考文献：

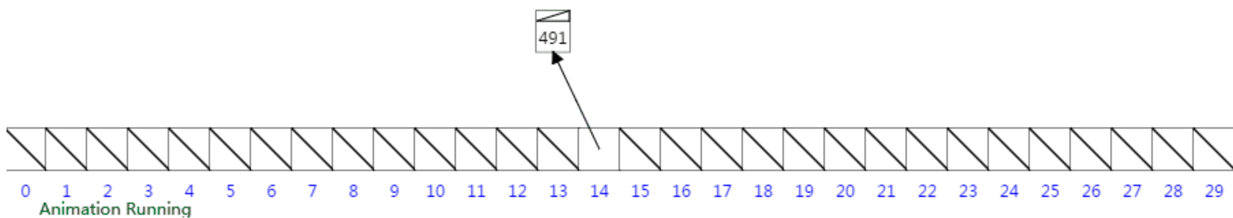
- 算法：排序算法之桶排序
- 关于python3.x的除法、向上向下取整及四舍五入的问题

桶排序原理

Randomize List | Bucket Sort

Skip Back | Step Back | pause | Step Forward | Skip Forward | Animation Speed | w: 1000 | h: 500 | Change Canvas Size | Move Controls

	568	948	47	702	758	321	958	282	77	912	152	920	69	891	170	529	574	675	575	489	24	803	249	113	33	357	611	576	602
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29



代码实现

```
import math  #用来进行除法向上取整
```

```
def Bucket_Sort(arr, interval=1000):
    """
    桶排序
    :param arr: 待排序数组
    :param interval: 每个桶的间隔
    :return: 排序好的数组
    """
    # 获取最大值和最小值
    maxNum = max(arr)
    minNum = min(arr)
    # 根据给出间隔算出桶个数
    bucket_num = math.ceil((maxNum - minNum) / interval)
    resArr = [[] for i in range(bucket_num + 1)]
    # 将数据依照间隔分入各桶中
    for date in arr:
        resArr[math.ceil((date - minNum) / interval)].append(date)

    # 对每一个桶进行排序，并将其汇总入最终数组
    sol = []
    for j in resArr:
        j.sort()
        sol += j
    return sol
```

```
arr = [63, 157, 189, -51, 101, 47, 141, -121, 157, 156, -194, 117, 98, 139, 67, 133, 181, 13, 28, 109]
print(Bucket_Sort(arr, 20)) # 间隔为20
>>>[-194, -121, -51, 13, 28, 47, 63, 67, 98, 101, 109, 117, 133, 139, 141, 156, 157, 157, 181, 189]
```

算法分析

n：待排序元素个数

m：桶的个数（对应上述代码中 bucket_num）

- 平均时间复杂度：O(n + m)
- 最佳时间复杂度：O(n + m)
- 最差时间复杂度：$O(n^2)$
- 空间复杂度：O(n * m)
- 稳定性：稳定

八，各种排序方法比较

类别	排序方法	时间复杂度			空间复杂度	稳定性
		最好情况	最坏情况	平均情况	辅助存储	
插入排序	直接插入排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
	希尔排序	O(n)	$O(n^2)$	$\sim O(n^{1.3})$	O(1)	不稳定
交换排序	冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
	快速排序	O(nlogn)	$O(n^2)$	O(nlogn)	O(nlogn)	不稳定
选择排序	直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
	堆排序	O(nlogn)	O(nlogn)	O(nlogn)	$O(n^2)$	不稳定
归并排序		O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定
基数排序 k:待排元素的维数，m为基数的个数		O(n+m)	O(k*(n+m))	O(k*(n+m))	O(n+m)	稳定

https://blog.csdn.net/weixin_46072771

1，时间性能

1，按平均的时间性能来分，有三类排序方法：

时间复杂度为 O(nlogn) 的方法有：

- 快速排序（最佳），堆排序和归并排序

时间复杂度为 O(n^2) 的有：

- 直接插入排序（最佳），冒泡排序和简单选择排序

时间复杂度为 O(n) 的排序方法只有：

- 基数排序

2，当待排记录序列按关键字**顺序有序**时，**直接插入法**和**冒泡排序法**能达到 **O(n)** 的时间复杂度；而对于**快速排序**而言，这是**最不好**的情况，此时的时间复杂度**退化为 O(n^2)**，因此是应该尽量避免的情况。

3，简单选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。

2，空间性能

指的是排序过程中所需的辅助空间大小。

1. 所有的简单排序方法（包括：直接插入、冒泡和简单选择）和堆排序的空间复杂度为 O(1)。
2. 快速排序为 O(logn)，为栈所需的辅助空间。
3. 归并排序所需辅助空间最多，其空间复杂度为 O(n)。
4. 链式基数排序需要附设队列首尾指针，则空间复杂度为 O(rd)。

3，排序方法的稳定性能

- 稳定的排序方法指的是，对于两个**关键字相等**的记录，他们在序列中的**相对位置**，在排序的前后**不会发生改变**。
- 当对多关键字的记录序列进行LSD方法排序时，必须采用稳定的排序方法。
- 对于不稳定的排序方法，只要能举出一个实例说明即可。
- 快速排序和堆排序是不稳定的排序方法。

4，关于“排序方法的时间复杂度下线”

- 本章讨论的各种排序方法，除基数排序，其他方法都是基于“**比较关键字**”进行排序的排序方法，可以证明，这类排序方法可能达到的**最快**的时间为 O(nlogn)。
（**基数排序**不是基于“比较关键字”的排序方法，所以它不受这个限制）
- 可以用一颗判断树来描述这类基于“比较关键字”进行排序的排序方法。