数据结构与算法

Data Structure and Algorithm

IV. 搜索与排序

授课人：Kevin Feng

翻译　：梁 少 华

Review

回顾

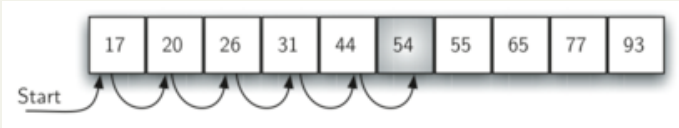
- 数据结构与算法
- 数学回顾
  - 大O表示法
  - 时间复杂度
  - 空间复杂度
- 数组
- 数组列表

概述

- 顺序查找
- 二分搜索
- 冒泡排序
- 选择排序
- 插入排序
- 计数排序
- **归并排序**
- **快速排序**
- 堆排序 / 二叉排序树
- Python Lib
- 应用

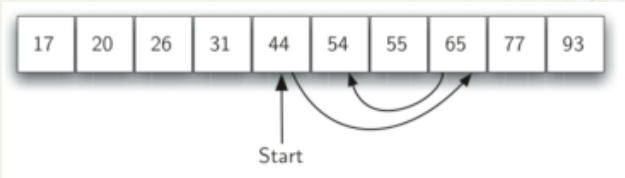
搜索

- 顺序查找



Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

- 折半搜索



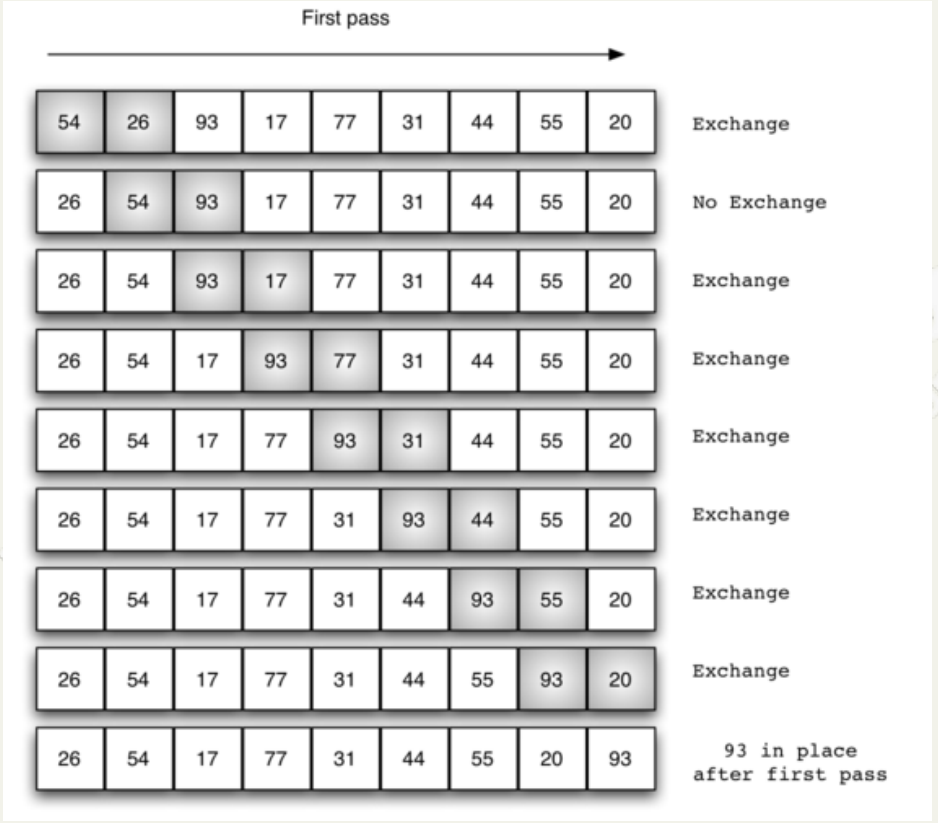
Comparisons	Approximate Number of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	
i	$\frac{n}{2^i}$

经典折半搜索

- 找到目标的第一个位置,如果找不到则返回-1
- 最后的位置, 任何的位置怎么样?
- 递归与迭代
- 模板

冒泡排序

- 重复列表进行排序,比较每对相邻的项目,如果它们的顺序错误,则交换它们
- 在每次通过时，未排序的最大元素已被“冒泡”到阵列末端的合适位置
- 重复列表直到不需要交换，这表明列表已被排序
- 属性:
  - 稳定
  - O(1) 额外的空间
  - $O(n^2)$ 比较和交换
  - 适应性: O(n) 接近排序时



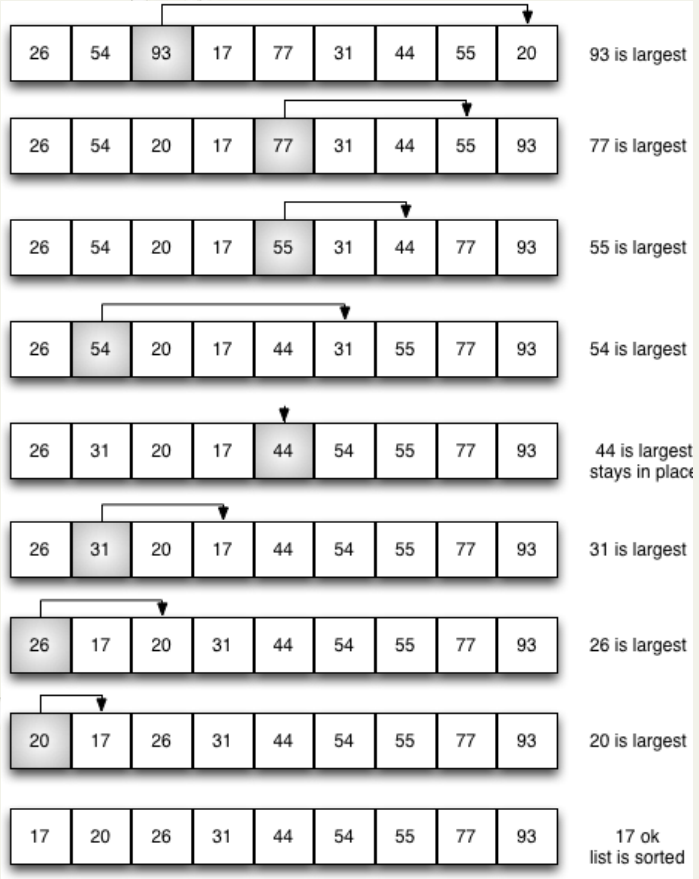
冒泡排序

```python
def _bubble_sort(nums: list, reverse=False):
    start = time.time()
    for i in range(len(nums)):
        # Get (i+1) largest in the correct position
        for j in range(len(nums) - i - 1):
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
    if reverse:
        #nums = nums[::-1]  # why this is not working?
        nums.reverse()
    t = time.time() - start
    return len(nums), t


def bubble_sorted(nums: list, reverse=False) -> list:
    """Bubble Sort"""
    nums_copy = list(nums)
    _bubble_sort(nums_copy, reverse=reverse)
    return nums_copy
```

Sort	Best	Average	Worst	Memory	Stable*	Method
Bubble	n	n^2	n^2	1	Yes	Exchanging

选择排序

- 列表上的2个线性传递
- 在每次通过时,它会选择最小的值
- 用最后一个未分类元素交换它
- 属性
  - 不稳定
  - O(1) 额外的空间
  - $O(n^2)$ 对比
  - O(n) 互换
  - 费适应性



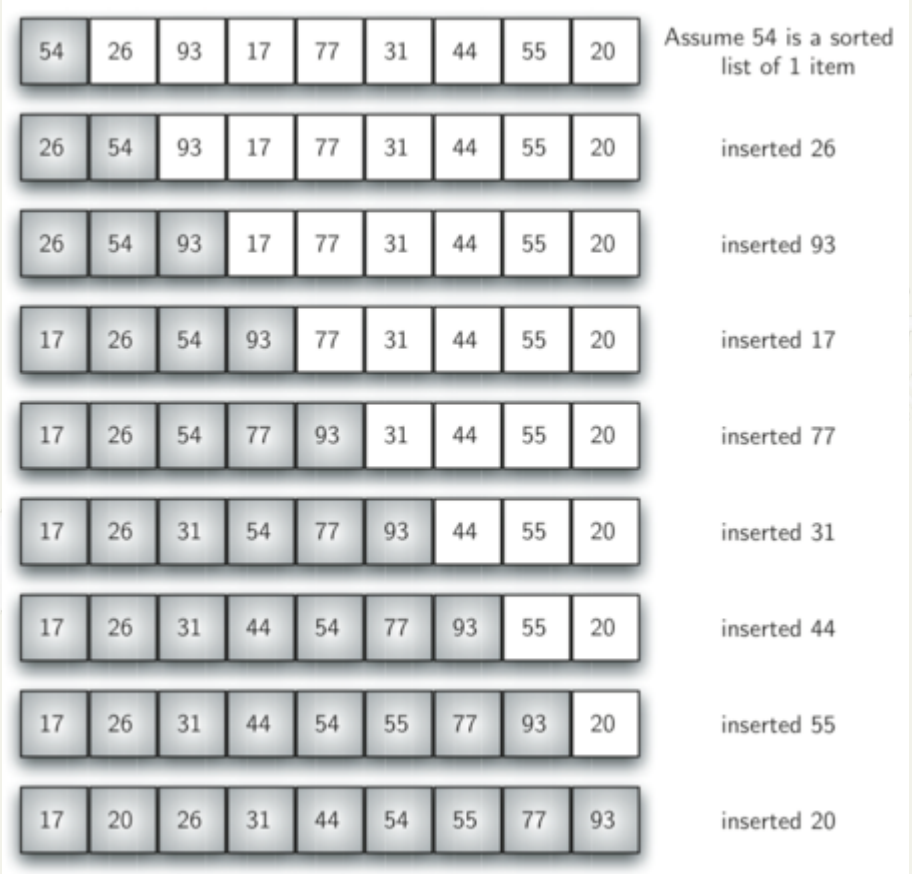
选择排序

```
def selection_sort(items):
    start = time.time()
    for i in range(len(items)):    # n
        pos_min = i   #idx
        for j in range(i + 1, len(items)):  # n
            if (items[j] < items[pos_min]):
                pos_min = j

        items[i], items[pos_min] = items[pos_min], items[i]
    t = time.time() - start
    return len(items), t
```

Sort	Best	Average	Worst	Memory	Stable*	Method
Selection	n^2	n^2	n^2	1	No	Selection

插入排序

- 在每次通过时，当前项目被插入列表的排序部分
- 它从排序列表的最后一个位置开始,向后移动直到找到当前项目的正确位置
- 然后将该项插入该位置，之后所有项目都被拖曳到左边以适应它
- 折半插入排序
  - 不是每次对正确的位置进行线性搜索，而是进行二进制搜索，这是 *O(log n)*而不是*O(n)*
  - 这带来了最佳铸造成本为*O(n log n)*
  - 唯一的问题是，即使项目处于当前位置，它也必须执行二分搜索
  - 由于有可能将所有其他元素从每一道传递到列表中，所以最坏情况下的运行时间保持在 $O(n^2)$



插入排序

```
def insert_sort(items):
    start = time.time()
    for sort_inx in range(1,len(items)):
        unsort_inx = sort_inx
        while unsort_inx > 0 and items[unsort_inx-1] > items[unsort_inx]:
            items[unsort_inx-1], items[unsort_inx] = items[unsort_inx], items[unsort_inx-1]
            unsort_inx = unsort_inx-1
    t = time.time() - start
    return len(items), t
```

Sort	Best	Average	Worst	Memory	Stable*	Method
Insertion	n	n^2	n^2	1	Yes	Insertion
Binary Insertion	$n \log n$	n^2	n^2	1	Yes	Insertion

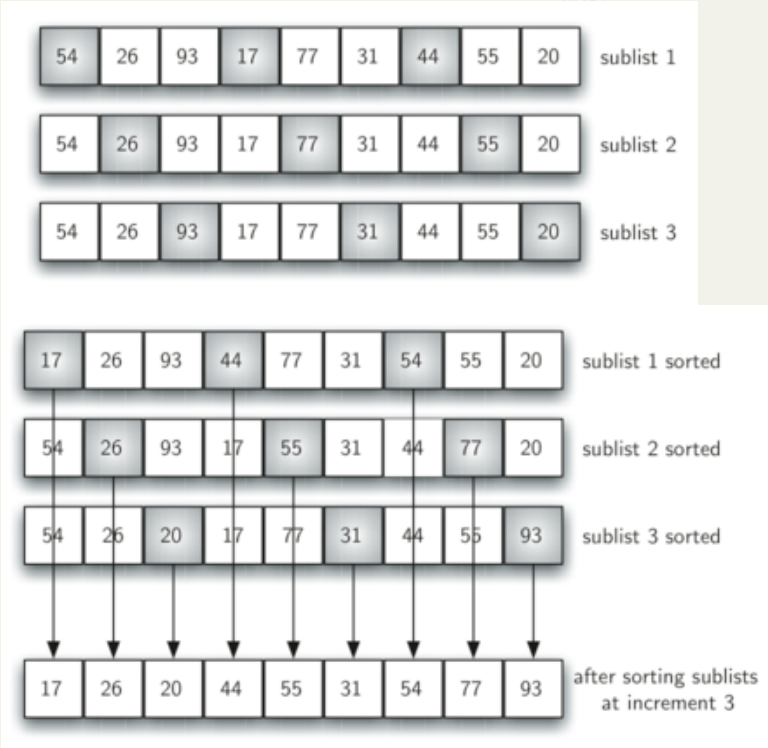
希尔排序

- 插入排序的简单扩展，通过允许相隔很远的元素交换来获得速度
- 逐步缩小要比较的元素之间的差距
- 从相距甚远的元素开始，可以将一些不适合的元素移动到比简单的最近邻居交换更快的位置
- 例如: 带间隙 5, 3 和 1

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
input data:	62	83	18	53	07	17	95	86	47	69	25	28
after 5-sorting:	17	28	18	47	07	25	83	86	53	69	62	95
after 3-sorting:	17	07	18	47	28	25	69	62	53	83	86	95
after 1-sorting:	07	17	18	25	28	47	53	62	69	83	86	95

- 第一步,5次排序,对单独的子阵列 (a1, a6, a11), (a2, a7, a12), (a3, a8), (a4, a9), (a5, a10)进行插入排序.
- 它将子阵列 (a1,a6, a11) 从 (62, 17, 25) 更改为 (17, 25, 62)
- 下一步,3次排序,对子阵列 (a1, a4, a7, a10), (a2, a5, a8, a11), (a3, a6, a9, a12)进行插入排序.
- 最后一步,1次排序,是整个数组的普通插入排序 (a1,..., a12)

希尔排序




```python
def shell_sort(nums):
    start = time.time()

    gap = len(nums)
    length = len(nums)

    while (gap > 0):
        for i in range(gap, length):
            for j in range(i, gap - 1, -gap):
                if (nums[j - gap] > nums[j]):
                    nums[j], nums[j - gap] = nums[j - gap], nums[j]

        if (gap == 2):
            gap = 1
        else:
            gap = gap // 2

    t = time.time() - start
    return len(nums), t
```

* 希尔排序的运行时间很大程度上取决于它使用的间隙顺序
* 对于许多实际的变量，确定它们的时间复杂度仍然是一个公开的问题

计数排序

- 计数排序的输入由n个项的集合组成
- 每个集合都有一个非负整数键其最大值最多为k

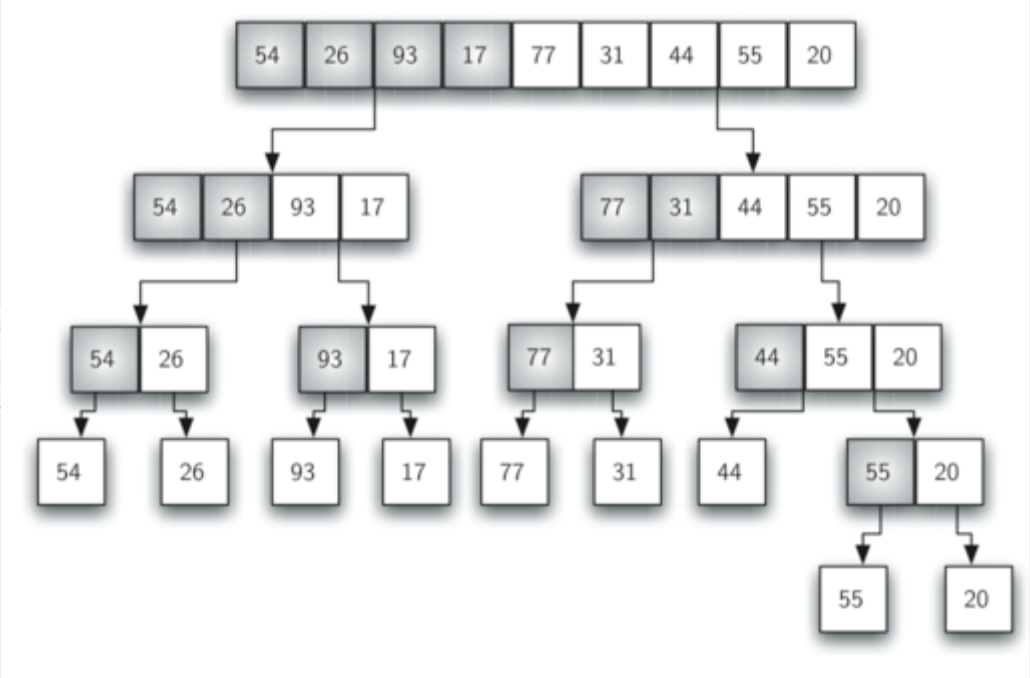
```
def count_sort(items):
    start = time.time()
    mmax, mmin = items[0], items[0]
    for i in range(1, len(items)):
        if (items[i] > mmax): mmax = items[i]
        elif (items[i] < mmin): mmin = items[i]
    print(mmax)
    nums = mmax - mmin + 1
    counts = [0] * nums
    for i in range (len(items)):
        counts[items[i] - mmin] = counts[items[i] - mmin] + 1

    pos = 0
    for i in range(nums):
        for j in range(counts[i]):
            items[pos] = i + mmin
            pos += 1

    t = time.time() - start
    return len(items), t
```

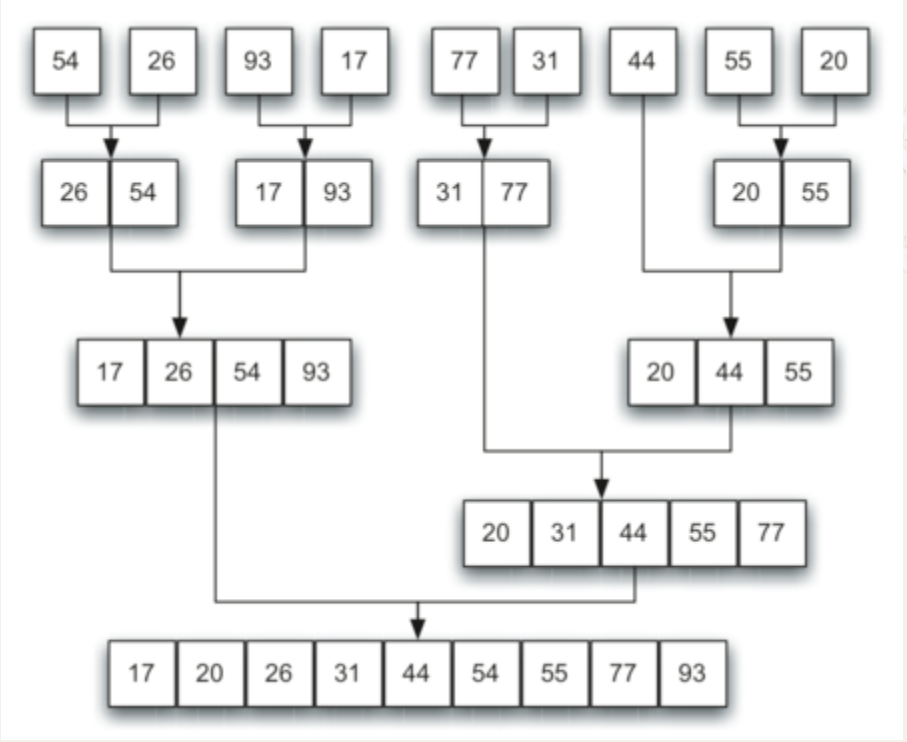
Sort	Best	Average	Worst	Memory	Stable*	Method
Count	n	n	n	k	No	Counting

归并排序

- 分而治之
- 分
  - 递归地拆分数组,直到它被分成两对单个元素数组为止.
  - 然后,将这些单个元素中的每一个与它的对合并,然后将这些对与它们的对等合并,直到整个列表按照排序顺序合并为止.
- 治
  - 将2个排序列表合并为另一个排序列表是很简单的.
  - 简单地通过比较每个列表的头,删除最小的,以加入新排序的列表.
  - *O(n)* 操作

归并排序


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20


54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

分治排序

- 数学分析.在最坏的情况下,合并排序使得~ *N lg N*比较并且运行时间是线性的.
- 二次线性方程式的鸿沟. N^2和*N lg N*之间的差异在实际应用中造成巨大的差异.
- 分而治之算法.相同的基本方法对许多重要问题都有效.
- 减少分拣.如果我们可以使用B的解来解答A,则问题A简化为问题B.
  - 例如,考虑确定数组中的元素是否都不同的问题.
  - 这个问题归结为排序,因为我们可以对数组进行排序,通过排序数组进行线性传递来检查是否有任何条目等于下一个条目（如果不是,则元素都不相同.

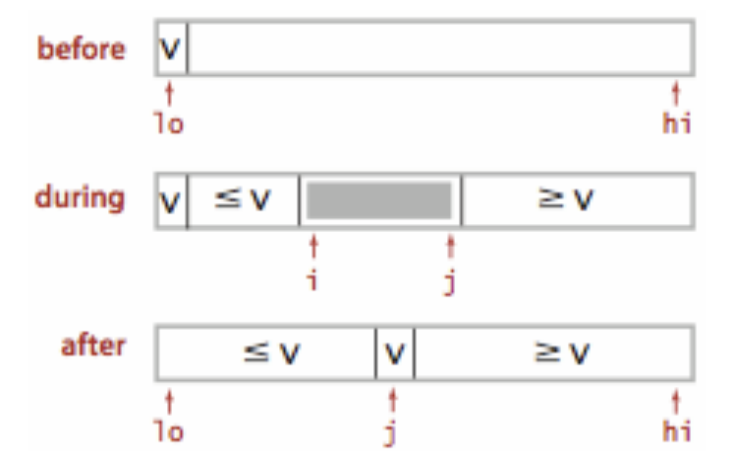
Sort	Best	Average	Worst	Memory	Stable*	Method
Merge	$n \log n$	$n \log n$	$n \log n$	Depends	Yes	Merging

改进

- 对小型子阵列使用插入排序
  - 我们可以通过不同的方式处理小案例来改进大多数递归算法.
  - 将典型合并排序执行的运行时间提高10％至15％.
- 测试数组是否已经按顺序排列
  - 如果[mid]小于或等于[mid+1] ,我们可以通过添加一个测试来跳过对merge( )的调用，从而将运行时间减少为已经按顺序排列的数组.
  - 通过这种改变,我们仍然可以进行所有的递归调用,但任何已经排序的子阵列的运行时间都是线性的.

快速排序

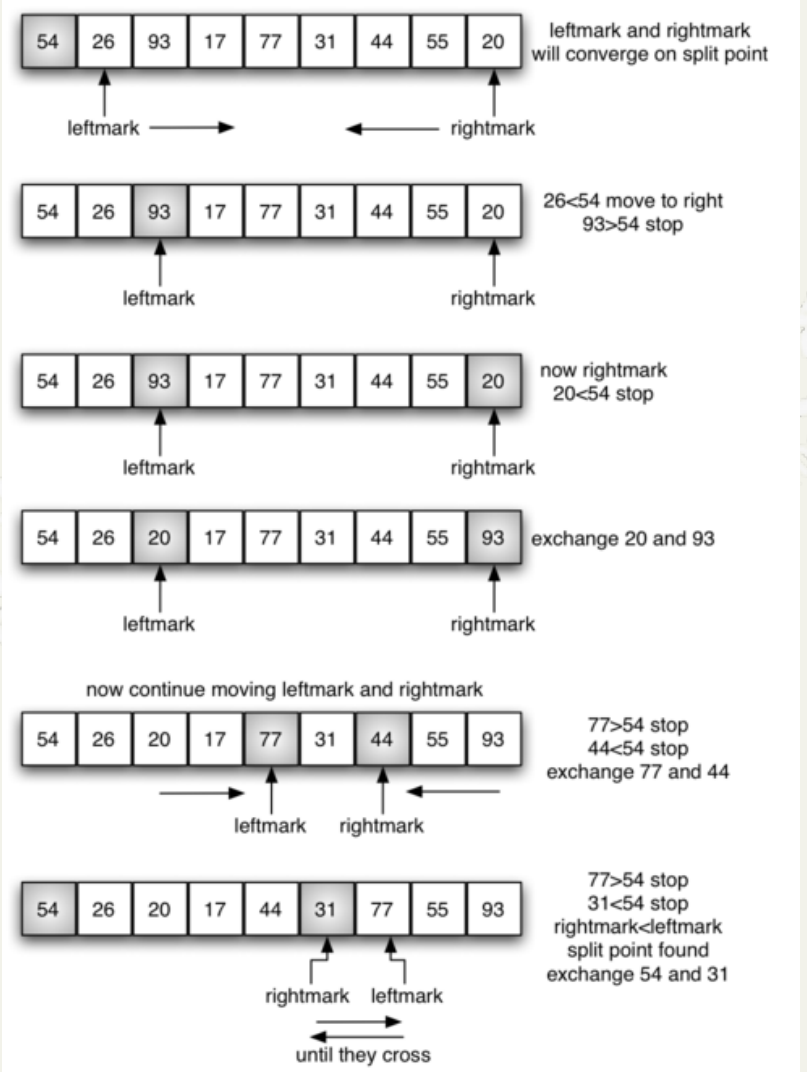
- 另一个分而治之
- 将数组划分为两个部分,然后独立地对部分进行排序
  - 首先选择一个数据透视,并从列表中删除(隐藏在最后)
  - 然后这些元素被分成两部分.一个小于枢轴,另一个大于枢轴. 这种分区是通过交换价值来实现的
  - 然后在中间恢复枢轴,并且这两个部分递归地快速排序.



Quicksort partitioning overview

快速排序

```python
def _quick_sorted(nums: list) -> list:
    if len(nums) <= 1:
        return nums

    pivot = nums[0]
    left_nums = _quick_sorted([x for x in nums[1:] if x < pivot])
    right_nums = _quick_sorted([x for x in nums[1:] if x >= pivot])
    return left_nums + [pivot] + right_nums


def quick_sorted(nums: list, reverse=False) -> list:
    """Quick Sort"""
    start = time.time()
    nums = _quick_sorted(nums)
    if reverse:
        nums = nums[::-1]
    t = time.time() - start
    return nums, len(nums), t
```



改进

- 切除插入排序
  - 与合并排序一样,切换到小数组的插入排序也是值得的
  - 截止值的最佳值取决于系统
  - 但是在大多数情况下,5和15之间的任何值都可能很好地工作.
- 三点中值算法
  - 提高快速排序性能的第二个简单方法是使用从阵列中提取的少量项目的中间值作为分区项目
  - 这样做会稍微改善分区,但是以计算中位数为代价
  - 事实证明,大多数可用的改进来自选择一个大小为3的样本(然后在中间项目上进行分区)

其他排序

- 其他排序算法
  - 二叉搜索树
  - 堆排序
  - 煎饼排序
    - 当锅铲可以插入堆叠中的任何一点并用于翻转它上面的所有煎饼时,按顺序排列一堆无序的薄饼
    - 1979年,比尔盖茨给出了5 / 3n的上限
    - 30年后,由德克萨斯大学达拉斯分校的研究人员组成的研究小组,由创始人Hal Sudborough教授领导,该研究小组在30年后改进到18 / 11n
- 应用
  - 对自定义对象进行排序：事务

总结

algorithm	stable?	inplace?	growth rate to sort N items		notes
			running time	extra space	
selection sort	no	yes	N^2	1	
insertion sort	yes	yes	between N and N^2	1	depends on order of input keys
shellsort	no	yes	$N^{6/5}$?	1	
quicksort	no	yes	$N \lg N$	$\lg N$	probabilistic guarantees, depend on distribution of input key values
3-way quicksort	no	yes	between N and $N \lg N$	$\lg N$	
mergesort	yes	no	$N \lg N$	N	
heapsort	no	yes	$N \lg N$	1	

Performance characteristics of sorting algorithms

数据结构与算法

Data Structure and Algorithm

IV. 搜索与排序结束

授课人：Kevin Feng

翻译　：梁少华