数据结构和算法

（Python描述）

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

内排序算法

内排序算法概述

法国圣十字湖

什么是内排序

- 在内存中进行的排序，叫内排序，简称排序

  复杂度不可能优于O(nlog(n))

- 对外存（硬盘）上的数据进些排序，叫外排序

如何评价排序算法

- 时间复杂度

  平均复杂度

  最坏情况复杂度

  最好情况复杂度

- 空间复杂度

  需要多少额外辅助空间

- 是否稳定:

  同样大小的元素，排序前和排序后是否先后次序不变

各种排序方法总结

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

Python的排序函数

Timsort，蒂姆排序

- 一种混合了归并排序和插入排序的算法
- 稳定
- 最坏时间复杂度O(nlog(n))
- 最好时间复杂度接近O(n)
- 额外空间：最坏O(n)，但通常较少
- 是目前为止最快的排序算法

插入排序

法国小镇Eze

插入排序

➢基本思想

1） 将序列分成有序的部分和无序的部分。有序的部分在左边，无序的部分在右边。开始有序部分只有1个元素

2） 每次找到无序部分的最左元素（设下标为i），将其插入到有序部分的合适位置(设下标为k,则原下标为k到i-1的元素都右移一位)，有序部分元素个数+1

3） 直到全部有序

插入排序

```
def insertionSort(a):
    for i in range(1,len(a)): #每次把a[i]插入到合适位置
        e,j = a[i],i
        while j > 0 and e < a[j-1]: #比写 a[j-1]>e 适应性强
            a[j] = a[j-1]       #(1)
            j -= 1
        a[j] = e
```

- 最坏情况(倒序)：语句(1)执行1+2+3+...+(n-1)次，总复杂度O(n^2)
- 平均情况：对每个i，语句（1）平均执行i/2次，总复杂度O(n^2)
- 最好情况（基本有序)：对每个i,语句(1)不执行或执行很少次，总复杂度O(n)
- 稳定性：稳定
- 额外空间：O(1)

插入排序

改进：二分法找插入位置

寻找插入位置时，用二分法。总体复杂度量级没有改进，因为要移动元素。且为了保证稳定性，插入位置必须是最后一个相等元素的后面。

插入排序常在其它高效排序算法中使用，比如改进的快速排序算法，在待排序区间很小的时候，改用插入排序

插入排序

改进： 可以自定义比较器

```
def insertionSort(a,key=lambda x,y:x<y):
    for i in range(1,len(a)):
        e,j = a[i],i
        while j > 0 and key(e,a[j-1]):
            a[j] = a[j-1]
            j -= 1
        a[j] = e

a = [5,47,74,3,4,12,8]
insertionSort(a,key = lambda x,y:x>y)
print(a) #>>[74, 47, 12, 8, 5, 4, 3]
```

插入排序

- 规模很小的排序可优先选用(比如，元素个数10以内)
- 特别适合元素基本有序的情况(复杂度接近O(n))
- 许多算法会在上述两种情况下采用插入排序。例如改进的快速排序算法、归并排序算法，在待排序区间很小的时候就不再递归快排或归并，而是用插入排序

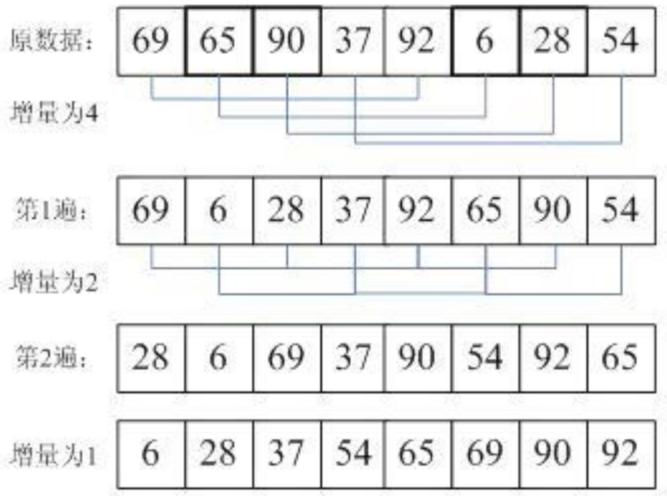
一种改进的插入排序: 希尔(shell)排序

1) 选取增量(间隔)为D，根据增量将列表分为多组，每组分别插入排序:

第一组： A_0 , A_{0+D} , $A_{0+2D,}$ ……
第二组： A_1 , A_{1+D} , $A_{1+2D,}$ ……
第三组： A_2 , A_{2+D} , $A_{2+2D,}$ ……

若D==1，则插入排序后，整个排序结束

2) D = D//2 ，转1



初始增量D可以为 n//2, n是元素总数
也许D还可以有别的选取法

最好：O(n)，平均$O(n^{1.5})$，最坏$O(n^2)$

选择排序

夏纳海滨

选择排序

➢基本思想

1） 将序列分成有序的部分和无序的部分。有序的部分在左边，无序的部分在右边。开始有序部分没有元素

2） 每次找到无序部分的最小元素（设下标为i），和无序部分的最左边元素（设下标为j)交换。有序部分元素个数+1。

3） 2)做n-1次，排序即完成

选择排序

```
def selectionSort(a):
    n = len(a)
    for i in range(n-1):
        minPos = i  #最小元素位置
        for j in range(i+1,n):
            if a[j] < a[minPos]: #(1)
                minPos = j
        if minPos != i:
            a[minPos],a[i] = a[i],a[minPos]
```

选择排序

```
def selectionSort(a):
    n = len(a)
    for i in range(n-1):
        minPos = i #最小元素位置
        for j in range(i+1,n):
            if a[j] < a[minPos]: #(1)
                minPos = j
        if minPos != i:
            a[minPos],a[i] = a[i],a[minPos]
```

- 无论最好、最坏、平均，语句(1)必定执行(n-1)+...+3+2+1次，复杂度O(n^2)
- 稳定性：不稳定，因a[i]被交换时，可能越过了其后面一些和它相等的元素
- 额外空间：O(1)

平均效率低于插入排序，没啥实际用处

冒泡排序

巴黎凡尔赛宫

冒泡排序

➢基本思想

1） 将序列分成有序的部分和无序的部分。有序的部分在右边，无序的部分在左边。开始有序部分没有元素

2） 每次从左到右，依次比较无序部分相邻的两个元素。如果右边的小于左边的，则交换它们。做完一次后，无序部分最大元素即被换到无序部分最右边，有序部分元素个数+1。

3） 2)做n-1次，排序即完成

冒泡排序

```
def bubbleSort(a):
    n = len(a)
    for i in range(1,n):
        for j in range(n-i):
            if a[j+1] < a[j]: #(1)
                a[j+1],a[j] = a[j],a[j+1]
```

- 无论最好、最坏、平均，语句(1) 必定执行(n-1)+...+3+2+1次，复杂度O(n^2)
- 稳定性：稳定
- 额外空间：O(1)

冒泡排序

改进：最好情况，即基本有序时，可以做到o(n)

如果发现某一轮扫描时，没有发生元素交换的情况，则说明已经排好序了，就不用再扫描了

```
def bubbleSort(a):
    n = len(a)
    for i in range(1,n):
        done = True
        for j in range(n-i):
            if a[j+1] < a[j]: #(1)
                a[j+1],a[j] = a[j],a[j+1]
                done = False
        if done:
            break
```

排序的稳定性

1. 假设线性表中每个元素有两个数据项key1和key2，现对线性表按以下规则进行排序：先根据数据项key1的值进行非递减排序；在key1值相同的情况下，再根据数据项key2的值进行非递减排序。满足这种要求的排序方法是（　　）。

A) 先按key1值进行冒泡排序，再按key2值进行直接选择排序
B) 先按key2值进行冒泡排序，再按key1值进行直接选择排序
C) 先按key1值进行直接选择排序，再按key2值进行冒泡排序
D) 先按key2值进行直接选择排序，再按key1值进行冒泡排序

归并排序

分治的典型应用：归并排序

- 数组排序任务可以如下完成：

  1） 把前一半排序

  2）把后一半排序

  3）把两半归并到一个新的有序数组，然后再拷贝回原数组，排序完成。

分治的典型应用：归并排序

```
[47] [18] [14] [96] [79] [24] [13]

[18   47] [14   96] [24   79] [13]

[14   18   47   96] [13   24   79]

[13   14   18   24   47   79   96]
```

分治的典型应用：归并排序

```
def Merge(a,s,m, e,tmp):
#将数组a的局部a[s,m]和a[m+1,e]合并到tmp,并保证tmp有序，然后再拷贝回a[s,e]
#归并操作时间复杂度：O（e-m+1）,即O（n）
    pb = 0
    p1 ,p2 = s,m+1
    while p1 <= m and p2 <= e:
        if a[p1] < a[p2]:
            tmp[pb] = a[p1]
            p1 += 1
        else:
            tmp[pb] = a[p2]
            p2 += 1
        pb += 1
```

分治的典型应用：归并排序

```
while p1 <= m:
    tmp[pb] = a[p1]
    pb+= 1
    p1 += 1
while p2 <= e:
    tmp[pb] = a[p2]
    pb += 1
    p2 += 1
for i in range(e-s+1):
    a[s+i] = tmp[i]
```

```
def MergeSort(a,s,e,tmp):
#将a[s:e+1]归并排序，用tmp做缓存
    if s < e :
        m = s + (e-s)//2
        MergeSort(a,s,m,tmp)
        MergeSort(a,m+1,e,tmp)
        Merge(a,s,m,e,tmp)


lst = [1,41,7,98,7,12,3]
lst2 = [0] * len(lst)
MergeSort(lst,0,len(lst)-1,lst2)
print(lst)
```

归并排序的时间复杂度

对n个元素进行排序的时间:

```
T(n)    = 2*T(n/2) + a*n
        = 2*(2*T(n/4)+a*n/2)+a*n
        = 4*T(n/4)+2a*n
        = 4*(2*T(n/8)+a*n/4)+2*a*n
        = 8*T(n/8)+3*a*n
        ...
```

$= 2^k *T(n/2^k)+k*a*n$

(a是常数，具体多少不重要)

一直做到 $n/2^k = 1$ **(此时** $k = \log_2 n$**)**,

$T(n) = 2^k *T(1)+k*a*n = 2^k *T(1)+k*a*n = 2^k+k*a*n$

$= n+a*(\log_2 n)*n$

复杂度O(nlogn)

归并排序

- 无所谓最坏、最好或平均情况，复杂度都是： O(nlog(n))
- 稳定性：稳定。只要归并两个区间时，碰到相同元素，总是先取左区间元素即可
- 额外空间：

  O(n)
  归并用额外空间O(n) + 栈空间O(log(n))

例题
求排列的逆序数

例题：求排列的逆序数

考虑1,2,…,n (n <= 100000)的排列i_1，i_2，…，i_n，如果其中存在j,k，满足 j < k 且 $i_j > i_k$， 那么就称(i_j,i_k)是这个排列的一个逆序。

一个排列含有逆序的个数称为这个排列的逆序数。例如排列 263451 含有8个逆序(2,1),(6,3),(6,4),(6,5),(6,1),(3,1),(4,1),(5,1)，因此该排列的逆序数就是8。

现给定1,2,…,n的一个排列，求它的逆序数。

例题：求排列的逆序数

笨办法：$O(n^2)$

分治$O(n\log n)$:

1） 将数组分成两半，分别求出左半边的逆序数和右半边的逆序数

2） 再算有多少逆序是由左半边取一个数和右半边取一个数构成(要求O(n)实现)

例题：求排列的逆序数

2) 的关键：左半边和右半边都是排好序的。比如，都是从大到小排序的。这样，左右半边只需要从头到尾各扫一遍，就可以找出由两边各取一个数构成的逆序个数

i				j			
10	8	7	3	12	11	5	2

例题：求排列的逆序数

2) 的关键：左半边和右半边都是排好序的。比如，都是从大到小排序的。这样，左右半边只需要从头到尾各扫一遍，就可以找出由两边各取一个数构成的逆序个数

i					j		
10	8	7	3	12	11	5	2

例题：求排列的逆序数

2) 的关键：左半边和右半边都是排好序的。比如，都是从大到小排序的。这样，左右半边只需要从头到尾各扫一遍，就可以找出由两边各取一个数构成的逆序个数

i

j

10	8	7	3	12	11	5	2

例题：求排列的逆序数

2) 的关键：左半边和右半边都是排好序的。比如，都是从大到小排序的。这样，左右半边只需要从头到尾各扫一遍，就可以找出由两边各取一个数构成的逆序个数

	i					j	
10	8	7	3	12	11	5	2

例题：求排列的逆序数

2) 的关键：左半边和右半边都是排好序的。比如，都是从大到小排序的。这样，左右半边只需要从头到尾各扫一遍，就可以找出由两边各取一个数构成的逆序个数

		i				j	
10	8	7	3	12	11	5	2

例题：求排列的逆序数

2) 的关键：左半边和右半边都是排好序的。比如，都是从大到小排序的。这样，左右半边只需要从头到尾各扫一遍，就可以找出由两边各取一个数构成的逆序个数

			i			j	
10	8	7	3	12	11	5	2

例题：求排列的逆序数

2) 的关键：左半边和右半边都是排好序的。比如，都是从大到小排序的。这样，左右半边只需要从头到尾各扫一遍，就可以找出由两边各取一个数构成的逆序个数

			i				j
10	8	7	3	12	11	5	2

例题：求排列的逆序数

总结：

由归并排序改进得到，加上计算逆序的步骤

MergeSortAndCount: 归并排序并计算逆序数

快速排序

挪威奥勒松

分治的典型应用：快速排序

- 数组排序任务可以如下完成：

  1）设k=a[0], 将k挪到适当位置，使得比k小的元素都在k左边,比k大的元素都在k右边，和k相等的，不关心在k左右出现均可 （O（n)时间完成)

  2）把k左边的部分快速排序

  3） 把k右边的部分快速排序

K = 7

i							j
7	1	3	8	12	11	2	9

K = 7

i						j	
7	1	3	8	12	11	2	9

K = 7

i j

2	1	3	8	12	11	7	9

K = 7

	i					j	
2	1	3	8	12	11	7	9

K = 7

		i				j	
2	1	3	8	12	11	7	9

K = 7

			i			j	
2	1	3	8	12	11	7	9

K = 7

			i			j	
2	1	3	7	12	11	8	9

K = 7

			i		j		
2	1	3	7	12	11	8	9

K = 7

i j

2	1	3	7	12	11	8	9

分治的典型应用：快速排序

```
def quickSort(a,s,e): #将a[s:e+1]排序
    if s >= e:
        return
    i,j = s,e
    while i != j:
        while i < j and a[i] <= a[j]:
            j -= 1
        a[i],a[j] = a[j],a[i]
        while i < j and a[i] <= a[j]:
            i += 1
        a[i], a[j] = a[j], a[i]
    #处理完后，a[i] = k
    quickSort(a,s,i-1)
    quickSort(a, i+1,e)
```

快速排序

- 最坏情况（已经基本有序或倒序）：$O(n^2)$
- 平均情况：O(nlog(n))
- 最好情况：O(nlog(n))
- 稳定性：不稳定
- 额外空间：

  两次递归的普通写法：最坏情况需要递归n层，需要n层栈空间，复杂度O(n)。
  最好情况和平均情况递归log(n)层，复杂度O(log(n))

快速排序

➢ 时间优化

如何避免最坏情况的发生

办法1） 排序前o(n)随机打乱

办法2） 若待排序段为a[s,e]，则选a[s],a[e],a[(s+e)/2]三者中的中位数作为分隔基准元素

快速排序

➢ 额外空间的优化

采用一次递归的尾递归优化写法，只递归排序短的那一半，可以做到最坏情况O(log(n))，最好O(1)

快速排序

- 额外空间的尾递归优化

```
def tailRecursiveQuickSort(a,s,e):
    while s < e:
        i,j = s,e
        while i != j: #分两半的操作
            while i < j and a[i] <= a[j]:
                j -= 1
            a[i],a[j] = a[j],a[i]
            while i < j and a[i] <= a[j]:
                i += 1
            a[i], a[j] = a[j], a[i]
        if i-s < e - i: #每次选短的那一半进行递归快排，递归层数就不会超过log(n)
            tailRecursiveQuickSort(a,s,i-1)
            s = i + 1
        else:
            tailRecursiveQuickSort(a,i+1,e)
            e = i - 1
```

线性递归和迭代

- 求n!的第一种递归算法：
  n! = n * (n-1)!

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

- 线性递归，时间和空间复杂度都为线性

(factorial 6) 计算过程：

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

需要在栈中存 6 5 4 3 2 1

线性递归和迭代

- 求n!的第二种算法，反复做:

  product = counter · product
  counter = counter + 1

从counter=1 做到counter = n

```
def factorial(n):
    def fact_iter(product,counter):
        if counter > n:
            return product
        return fact_iter(product *
                counter,counter + 1)
    return fact_iter(1,1)
print(factorial(6))
```

(factorial 6) 计算过程:

```
(factorial 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

只需要在栈中存三个参数(后面的可以覆盖前面的)

- 迭代，时间复杂度线性，**空间复杂度常数**

尾递归优化

如果一个函数中某个递归调用出现在函数的返回语句，且递归调用的返回值不属于包含局部变量的表达式的一部分时，这个递归调用就是尾递归。

进行尾递归调用时，因为栈中存放的内容已经不再有用，所以可以不必让栈加深，只在原位置进行下一层函数调用即可。

尾递归

```
def factorial(n):
	if n == 0:
		return 1
	return n * factorial(n-1)
```

不是尾递归，因为栈中要存 n

尾递归

- 一些编译器或解释器会进行尾递归优化，以节约空间甚至降低空间复杂度。如何做实验判断编译器或解释器有无尾递归优化?

尾递归

- 一些编译器或解释器会进行尾递归优化，以节约空间甚至降低空间复杂度。如何做实验判断编译器或解释器有无尾递归优化?

```
def f(n):
        f(n)

f(2)
```

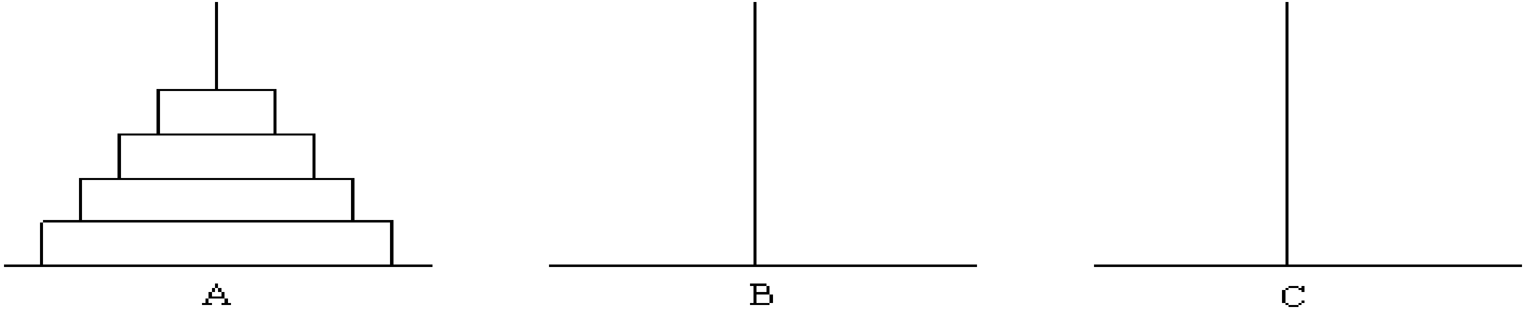
看死循环还是爆栈

```
RecursionError: maximum recursion depth exceeded
```

说明没优化

汉诺塔问题(Hanoi)

古代有一个梵塔，塔内有三个座A、B、C，A座上有64个盘子，盘子大小不等，大的在下，小的在上（如图）。有一个和尚想把这64个盘子从A座移到C座，但每次只能允许移动一个盘子，并且在移动过程中，3个座上的盘子始终保持大盘在下，小盘在上。在移动过程中可以利用B座，要求输出移动的步骤。



汉诺塔问题(Hanoi)

```
def Hanoi(n, src,mid,dest):
       #将src座上的n个盘子，以mid座为中转，移动到dest座
       if( n == 1) : #只需移动一个盘子
              # 直接将盘子从src移动到dest即可
              print(src + "->" + dest)
              return #递归终止
       Hanoi(n-1,src,dest,mid)  #先将n-1个盘子从src移动到mid
       print(src + "->" + dest) #再将一个盘子从src移动到dest
       Hanoi(n-1,mid,src,dest)  #最后将n-1个盘子从mid移动到d

n = int(input())
Hanoi(n, 'A', 'B', 'C');
```

n = 3

A->C
A->B
C->B
A->C
B->A
B->C
A->C

n个盘子的汉诺搬家，需要移动盘子2^n-1次

T(n) = 2 * T(n-1) + 1

汉诺塔的尾递归优化

```
def tailRecursiveHanoi(n,a,b,c):
#n个盘子从a移动到c,用b中转
    while n > 0:
        tailRecursiveHanoi(n-1,a,c,b)
        print(a,"->",c)
        n -= 1
        a,b,c = b,a,c
```

堆排序

普罗旺斯

堆的应用：堆排序

1) 将待排序列表a变成一个堆(O(n))
2) 将a[0]和a[n-1]交换，然后对新a[0]做下移，维持前n-1个元素依然是堆。此时优先级最高的元素就是a[n-1]
3) 将a[0]和a[n-2]交换，然后对新a[0]做下移， 维持前n-2个元素依然是堆。此时优先级次高的元素就是a[n-2]

……

直到堆的长度变为1，列表a就按照优先级从低到高排好序了。

整个过程相当不断删除堆顶元素放到a的后部。堆顶元素依次是优先级最高的、次高的....

一共要做n次下移，每次下移O(log(n))，因此总复杂度O(nlog(n))

堆排序

如果用递归实现，需要O(log(n))额外栈空间(递归要进行log(n)层)。

如果不用递归实现，需要O(1)额外空间。

用Python自带的堆来排序

```
import heapq
def heapSorted(iterable): #iterable是个序列
#函数返回一个列表，内容是iterable中元素排序的结果，不会改变iterable
        h = []
        for value in iterable:
                h.append(value)
        heapq.heapify(h)   #将h变成一个堆
        return [heapq.heappop(h) for i in range(len(h))]
a = (2,13,56,31,5)
print(heapSorted(a)) #>>[2, 5, 13, 31, 56]
print(heapq.nlargest(3,a)) #>>[56, 31, 13]
print(heapq.nlargest(3,a,lambda x:x%10))
#>>[56, 5, 13] 取个位数最大的三个
print(heapq.nsmallest(3,a,lambda x:x%10))
#>>[31, 2, 13] 取个位数最小的三个
```

堆排序

```
def heapSort(a,key = lambda x:x): #对列表a进行排序
    def makeHeap(): #建堆
        i = (heapSize - 1 - 1) // 2 #i是最后一个叶子的父亲
        for k in range(i,-1,-1):
            shiftDown(k)
    def shiftDown(i): #a[i]下移
        while i * 2 + 1 < heapSize:   #只要a[i]有儿子就做
            L,R = i * 2 + 1, i * 2 + 2
            if R >= heapSize or key(a[L]) < key(a[R]):
                s = L
            else:
                s = R
            if key(a[s]) < key(a[i]):
                a[i],a[s] = a[s],a[i]
                i = s
            else:
                break
```

堆排序

```
        heapSize = len(a)
        makeHeap()
        for i in range(len(a)-1,0,-1):
                a[i],a[0] = a[0],a[i]
                heapSize -= 1
                shiftDown(0)
        n = len(a)
        for i in range(n//2):  #颠倒a
                a[i],a[n-1-i] = a[n-1-i],a[i]

a = [3,21,4,76,12,3]
heapSort(a)
print(a) #>>[3, 3, 4, 12, 21, 76]
```

堆排序

- 最坏情况：O(nlog(n))
- 平均情况：O(nlog(n))
- 最好情况：O(nlog(n))
- 稳定性：不稳定
- 额外空间：O(1) （可以用非递归写法，或编译器、解释器自动尾递归优化）

桶排序
基数排序

俯瞰巴黎

分配排序(桶排序)

- 如果待排序元素只有m种不同取值，且m很小（比如考试分数只有0-100），可以采用桶排序

- 设立m个桶，分别对应m种取值。桶和桶可以比大小，桶的大小就是其对应取值的大小。把元素依次放入其对应的桶，然后再按先小桶后大桶的顺序，将元素都收集起来，即完成排序

- 复杂度O(n+m)，且稳定。n是待排序元素个数。

- 额外空间：O(n+m)

例如：将考试分数分到0-100这101个桶里面，然后按照0、1、2...100的顺序收集桶里的分数，即完成排序

桶排序


```
def bucketSort(s,m,key=lambda x:x):
    buckets = [[] for i in range(m)]
    for x in s:
        buckets[key(x)].append(x)
    i = 0
    for bkt in buckets:
        for e in bkt:
            s[i] = e
            i += 1
lst = [2, 3, 4, 8, 9, 12, 3, 2, 4, 12]
bucketSort(lst, 13)
print(lst) #>>[2, 2, 3, 3, 4, 4, 8, 9, 12, 12]
lst = [(-2, "Jack"), (8, "Mike"), (2, "Jane"), (2, "John")]
bucketSort(lst, 12, lambda x: x[0]+2)   # 取值范围写大一些也无妨
print(lst) #>>[(-2, 'Jack'), (2, 'Jane'), (2, 'John'), (8, 'Mike')]
lst = ["Jack", "Mike", "Lany", "Ada"]
bucketSort(lst, 26, lambda x: ord(x[0])-ord("A"))
print(lst) #>>['Ada', 'Jack', 'Lany', 'Mike']
```

多轮分配排序（基数排序）

- 有时也称为桶排序

- 思想

1）将待排序元素看作由相同个数的原子构成的元组（$e_1, e_2 ... e_n$）。长度不足的元素，用最小原子补齐左边空缺的部分。

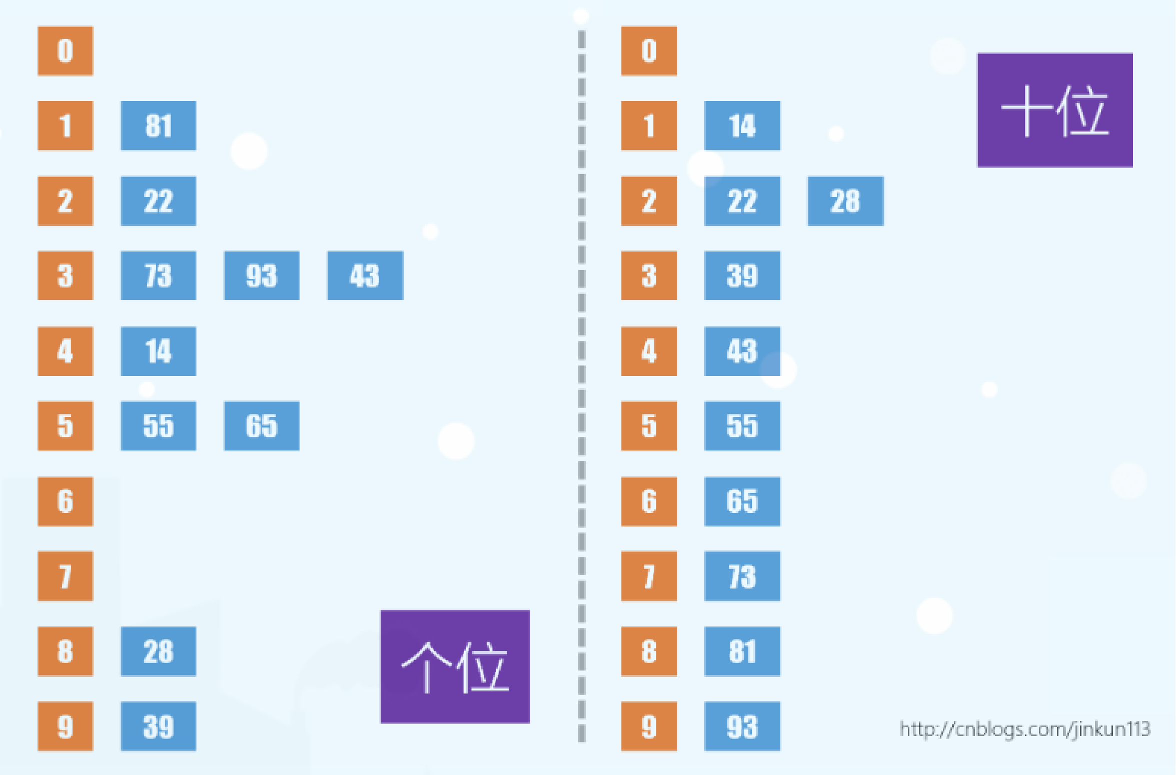
2）原子种类必须很少。有n种原子，就设立n个桶

3）先按e_n将所有元素分配到各个桶里，然后从小桶到大桶收集所有元素，得到序列1，然后将序列1按e_{n-1}分配到各个桶里再收集成为序列2.....直到按e_0分配到桶再完成收集得到序列n，序列n就是最终排序结果。

多轮分配排序（基数排序）

- **对序列 73,22,93,43,55,14,28,65,39,81 排序**

- 根据个位数，将每个数分配到0到9号桶
- 按桶号由小到大收集这些数，得到序列1：
81,22,73,93,43,14,55,65,28,39
- 按顺序将序列1中的每个数，根据十位数，重新分配到0到9号桶
- 按桶号由小到大收集这些数，得到序列2：
14,22,28,39,43,55,65,73,81,93



多轮分配排序（基数排序)

- **基数排序的复杂度是 O(d*(n+radix))**

  n：要排序的元素的个数(假设每个元素由若干个原子组成)
  radix: 桶的个数，即组成元素的原子的种类数
  d: 元素最多由多少个原子组成

对序列 73,22,93,43,55,14,28,65,39,81 排序:
n = 10, d = 2, radix = 10(或9)

- 一共要做 d 轮分配和收集
- 每一轮, 分配的复杂度 O(n)，收集的复杂度O(radix)
  (一个桶里的元素可以用链表存放，便于快速搜集)
- 总复杂度 O(d * (n + radix))

基数排序


```
def radixSort(s, m, d, key):
    #key(x,k)可以取元素x的第k位原子
    for k in range(d):
        buckets = [[] for j in range(m)]
        for x in s:
            buckets[key(x, k)].append(x)
        i = 0
        for bkt in buckets:
            for e in bkt:
                s[i] = e
                i += 1
```

基数排序


```
def getKey(x, i):
    #取非负整数x的第i位。个位是第0位
    tmp = None
    for k in range(i + 1):
        tmp = x % 10
        x //= 10
    return tmp

lst = [123,21,48,745,143,62,269,87,300,6]
radixSort(lst, 10, 3, getKey)
print(lst) #>>[6, 21, 48, 62, 87, 123, 143, 269, 300, 745]
```