一 排序

1 插入排序

给予一组长度为n的序列A，将其按从小到大的顺序进行排序，伪代码如下：

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

运行的时间时间复杂度如下：

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

其中的tj根据实际的序列情况各有不同，最好的情况下（序列已经从小到大排好序了），tj=1，这个时候

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

最坏的情况为输入序列是从大到小的序列，这个时候tj=j-1，有

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

取最坏情况下的时间复杂度作为该算法的时间复杂度，可知插入排序的时间复杂度为$O(n^2)$

2 合并排序

合并排序的基本思想是使用分治法对序列进行排序，将该序列拆分成2个子序列，然后对子序列进行排序，最后将两个排好序的子序列进行合并，最后就可以得到排好序的完整序列。
分治法的特点是它采用递归的处理方式，将序列拆分到足够小（一般长度为1）后进行合并，伪代码如下：

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)

MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Merge过程将序列A中下标[p,q]和[q+1,r]的两个序列进行合并，以一个被拆分后的最小序列为例，Merge-Sort(A,0,1)，那么它里面的过程则为：

```
if(0<1)
{
        int mid = (0+1)/2 ;//==>mid = 0
        Merge-Sort(A,0,0);//直接返回
        Merge-Sort(A,1,1);//直接返回
        Merge(A,0,0,1);          //对子序列[0,0]和[1,1]，即两个下标为0和1的元素进行排序
}
```

时间复杂度

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

当进行排序的序列长度小于一个常量c时，分治排序的时间是一个常量，用O(1)表示。假设把原问题分解成a个子问题，每个子问题的大小是原来的1/b。（对于合并排序，a和b都是2，但在许多分治法中，a≠b）。D(n)为分解的时间（这一步仅算出序列的中间位置，需要的时间为常量，也即时间复杂度为O(1)），C(n)为合并的时间（为Merge过程的时间复杂度，参看其伪代码可知复杂度为O(n)）。
取三项中的最大项，得到时间复杂度为O($nl_g n$)，这里的$l_g n$为$\log_2 n$。具体过程参考书本。

第三章 函数的增长

Θ记号

对于给定的函数g(n)，用Θ(g(n))来表示函数g(n)的集合。
f(n) = Θ(g(n))，指的是存在正常数c1、c2、n0，使得对于所有的n>n0，有 0<= c1g(n) <= f(n) <= c2g(n)。从这里可以看出Θ记号渐进的给出了一个函数的上界和下界。

对于函数f(n) = $an^2 + bn + c$，其中a、b、c都是常数且a>0，则根据集合的定义，可知f(n)= Θ(n^2)，也即f(n)是函数n^2的集合。

O记号

f(n) = O(g(n))，指的是存在正常数c和n0，使得对于所有的n>n0，有 0<= f(n) <= cg(n)。O(g(n))在一个常数因子内给出了某函数的一个上界。

O记号用来表示算法最坏情况时运行时间的上界

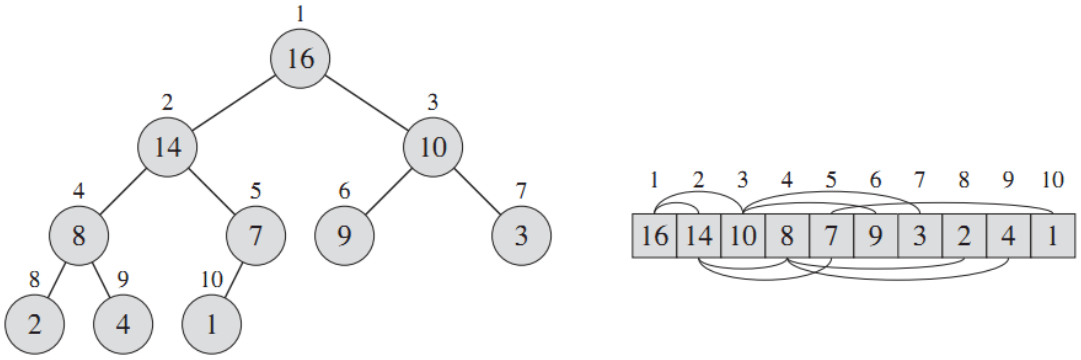
上取整⌈⌉和下取整⌊⌋

第六章 堆排序

堆排序集合了插入排序和合并排序的优点：插入排序的原地排序特点（任意时刻，数组中只有常数个元素存储在输入数组以外），合并排序的效率（时间复杂度也为$O(nlg n)$）。

堆

堆采用的完全二叉树数据结构，以一个数组来实现。堆中的元素按照二叉树的层从左到右进行存放，如下图所示。



对于一个下标为i的元素，定义三个过程分别获取它的父节点、左子节点和右子节点

```
PARENT(i)
  return i/2;
LEFT(i)
  return 2i;
RIGHT(i)
  return 2i + 1;
```

二叉堆有两种：最大堆和最小堆。最大堆的特点是，除了根以外的任意结点i，有
$A[PARENT(i)] >= A[i]$
最小堆则相反，除了根以外的任意结点i，有
$A[PARENT(i)] <= A[i]$

堆排序使用的是最大堆

结点在堆中的高度定义为从本节点到叶子的最长简单下降路径上边的数目，这样的话堆的高度即为根的高度。

Note
堆的一些性质

- 高度为N的完全二叉树总的元素的个数 $\sum_{i=0}^{N} 2^i = 2^{N+1}-1$
- 元素个数为n的堆的高度为⌊lgn⌋（⌊lg(2^(N+1)-1)⌋ == N）
- 高度为h的结点，它距根节点的深度为k = ⌊lgn⌋ - h，则该深度的元素个数为2^k=2^(⌊lgn⌋ - h) = 2^⌊lgn⌋/(2^h)，可得最大值为⌈$n/2^{h+1}$⌉（最后一步结论请细细分析感受下）

保持堆性质

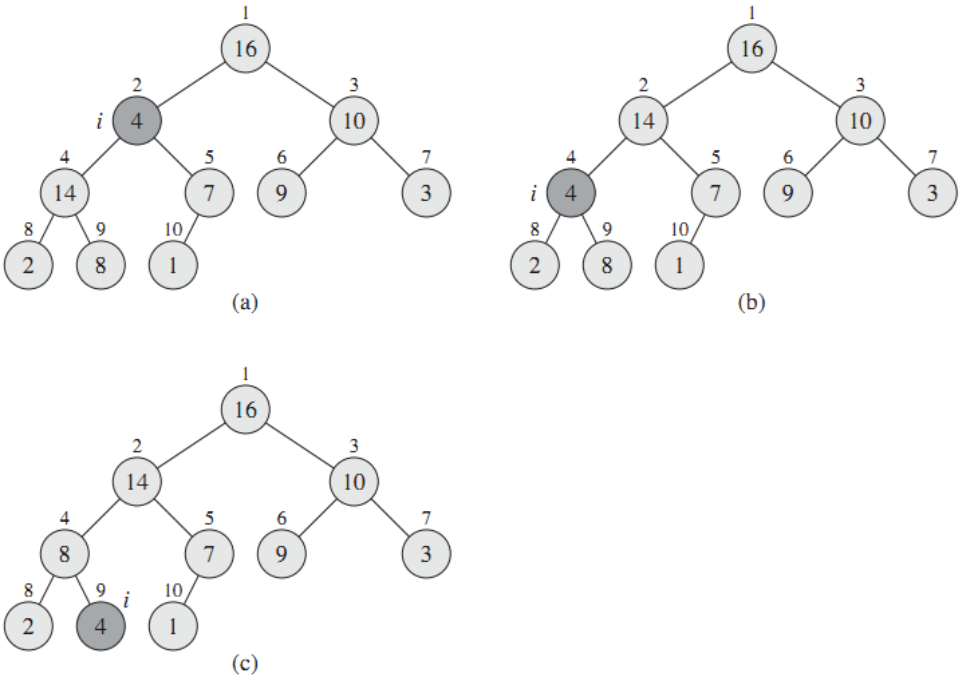
因为堆排序使用的是最大堆，而最大堆的性质是除了根以外的任意结点i，有A[PARENT(i)] >= A[i]

排序的一个核心操作是保持堆的这个性质，使用了过程MAX-HEAPIFY(A,i)，输入为一个数组A和下标i，当MAX-HEAPIFY(A,i)被调用时，假定以LEFT(i)和RIGHT(i)为根的两颗二叉树都是最大堆，但现在有可能A[i]小于LEFT(i)和RIGHT(i)，所以假如A[i]更小的话需要将其下沉。

伪代码如下：

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

下图显示了一个例子，黑色节点调用MAX-HEAPIFY下沉的过程



当调用MAX-HEAPIFY作用在一棵以节点i为根节点，大小为n的子树上时，i节点的子树大小至多为2n/3（**Todo:看不明白**）。运行时间为

$$T(n) \le T(2n/3) + \Theta(1) .$$

建堆

自底向上使用MAX-HEAPIFY就可以构建一个最大堆，伪代码如下

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

（从A.length/2开始是因为前面讲到一个节点i的父节点下标是i/2，而MAX-HEAPIFY应从最底端的非叶子节点开始操作）

BUILD-MAX-HEAP时间分析
MAX-HEAPIFY的复杂度是O(lgn)，共有O(n)次调用，故运行时间是O(nlgn)。但是这个分析不精确，结合前面note部分的知识可知，对于高度为h的结点，MAX-HEAPIFY的时间是O(h)，调用的次数（也就是该层叶子节点的个数）是$\lceil n/2^{h+1} \rceil$，可得整个过程的时间为：

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

又有：

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

故得出：

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

这说明可以在线性时间内，将一个数组建成最大堆。

堆排序

先用BUILD-MAX-HEAP将输入数组A[1..n](n=length(A))够建成最大堆，此时A[1]为最大元素。然后将A[1]与A[n]互换，接着又继续对数组A[1..n-1]构建最大堆，如此进行后即可完成排序。

伪代码如下

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT时间：BUILD-MAX-HEAP时间为O(n)，然后调用MAX-HEAPIFY(O(lgn))的次数为n-1，总体时间代价为O(nlgn)。

优先级队列

优先级队列以队列中元素关键字的值作为优先级，然后可以根据这个优先级进行一些操作。例如在作业调度过程中，优先级最高的作业被优先调度，调度完之后从队列弹出，同时可以在任意时刻往队列中添加新的作业。

以最大优先级队列为例，它支持如下操作：

- INSERT(S,key)：把key插入几何S中
- MAXIMUM(S)：取S中最大元素
- EXTRACT-MAX(S)：去掉并返回S中最大元素
- INCREASE-KEY(S,x,k)：将元素x下标的值增加到k，这里k不能小于S[x]

对应的伪代码：

```
HEAP-MAXIMUM(A)
1  return A[1]

HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max

HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)

MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = -∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

其中的INCREASE-KEY(S,x,k)，由于S原本是最大堆，且k>=S[x]，那么k插入到x处之后，x节点以下的仍然是最大堆，这个时候只需校验x以上的节点即可。

第七章 快速排序

快速排序采用了合并排序一样的分治模式，但它具有原地排序的特点。
相比于堆排序，快速排序的平均性能相当好。

伪代码：

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

快速排序的核心是分割函数PARTITION，它选取A[r]作为主元，得到分割的边界q，并将主元换到A[q]，同时边界q将原先的序列分割成两段，其中序列[p,q-1]小于A[q]，序列[q+1,r]大于A[q]。

性能分析

最坏划分

当输入的序列是一个从小到大排好序的序列使，分割出来的序列为最不平衡的序列，假如初始长度为n，分割一次得到序列[1,n-1]和 [n+1,n]（这个为空序列），第二次得到序列[1,n-2]与另一个空序列，运行时间可以递归的表示为：

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

使用迭代法可得 T(n) = $\Theta(n^2)$

Note：

这种情形下，需要分割n次，每次的PARTITION的复杂度又为$\Theta(n)$，所以可得$\Theta(n^2)$ 的运行时间

最优划分

假如PARTITION每一次的划分都可以得到两个平衡的序列，第一次划分后一个序列大小为⌊n/2⌋，另一个为⌈n/2⌉-1，其运行时间可以递归为：

$$T(n) = 2T(n/2) + \Theta(n) .$$

可知划分的次数为O(lgn),即可得运行时间为 $\Theta(nlg_n)$

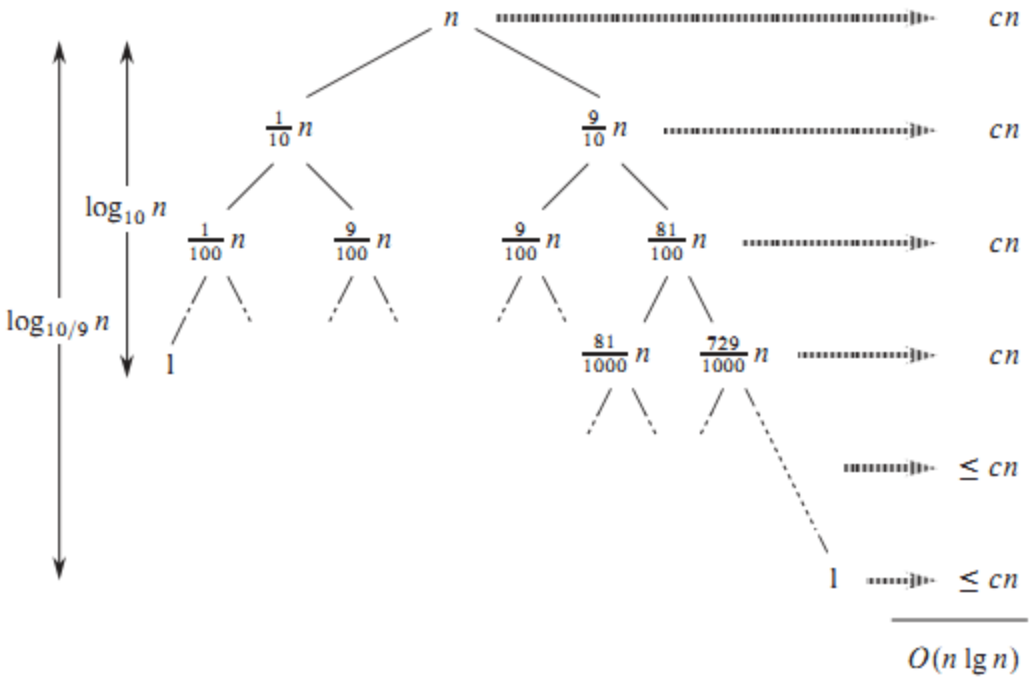
这里可以得到最优的性能，是因为在每次都可以进行平衡划分的时候，划分的次数最少，为lgn

平衡划分

假设划分过程总是以9:1进行划分，可得运行时间递归式

$$T(n) = T(9n/10) + T(n/10) + cn$$

以此为例的一棵递归树如下



可以看出划分出的小序列最先完成，该序列对应的高度为$\log_{10}n$，而大序列经过了$\log_{10/9}n$次才划分完成，在没有出现序列划分完成的时候，树上的每一层所有节点对应划分过程的时间为cn(所

有节点对应的序列的元素之和为n)，当出现有的小序列划分完成之后，每一层的时间就<=cn了。

这里递归树的深度为$\log_{10/9}n = O(\lg n)$，同时可以发现按常数比例（就算是9999:1）进行的划分都会产生深度为 O(lgn)的递归树，其中每一层的代价为O(n)

这里就说明了为什么快速排序为什么会有很好的平均性能了。

对数的换底公式

$$\log_\alpha x = \frac{\log_\beta x}{\log_\beta \alpha}$$。