分而治之的艺术

递归与分治算法

分治——分而治之

基本思想

将原问题划分成m个规模较小，而结构与原问题相似的子问题，递归的解决这些子问题，然后再合并其结果，就得到原问题的解。

所谓递归就是对子问题也要进行划分，直到子问题容易的求解

基本步骤

分解
（Divide）

将原问题分解为规模更小的子问题

解决
（Conquer）

如果问题足够小，可以直接解决；否则调用相同的方法求解

合并
（Combine）

将子问题的结果合并为原问题的解

例子：二分查找

- 问题:给定已按升序排好序的n个元素a[o:n-1]，现要在这n个元素中查找一特定元素x
- 分析:
  - 当数组小到一个元素的时候就可以判断
  - 可以选取数组中的中间位置的值a[n/2]，如果
    - X< a[n/2],则在a[o – (n/2-1)]中查找
    - X> a[n/2],则在a[(n/2+1) – n-1]中查找
    - X = a[n/2],已找到，返回
  - 不需要最后的合并步骤

能否解决的问题

该问题可以分解为若干个规模较小的相同问题

该问题的规模缩小到一定的程度就可以容易地解决

利用该问题分解出的子问题的解可以合并为该问题的解

分解
（Divide）

解决
（Conquer）

合并
（Combine）

子问题独立性

该问题所分解出的各个子问题是相互独立的
即子问题之间不包含公共的子问题

问题：归并排序

- 问题：对一列数字进行排序
- 分析
  - 分解：可以把一列数字分成多段进行排序
  - 解决：当每个序列只剩一个元素的时候自然有序
  - 合并：对于已经有序的两个序列，如何进行合并
  - 子问题独立性：每个问题的子问题都是相互独立的
- 基本步骤
  - 分解：将n个元素分成各含n/2个元素的子序列
  - 解决：对每个子序列进行归并排序（当每个子列中只有一个元素时，那么这个序列已经是有序了）
  - 合并：合并两个已排序的子序列得到排列结果

归并两个有序数列

将两个有序的数列合并为一个有序的数列

时间复杂度O(n)

递归过程

14 12 15 13 11 16
14 12 15
14 12
14 12 15
12 14
12 14 15
13 11 16
13 11
13 11 16
11 13
11 13 16
11 12 13 14 15 16

问题：快速排序(QUICKSORT)

- 基本步骤
  - 分解：任取一个元素x，将数组分为小于x的部分和大于等于x的部分
  - 解决：对每个子序列应用快速排序算法
  - 合并：无需合并，自然有序

两种方法比较

	归并排序	快速排序
分解（Divide）	✓	★
解决（Conquer）	✓	✓
合并（Combine）	★	✓

问题：棋盘覆盖

- 在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

$2^{k-1}X2^{k-1}$
$2^{k-1}X2^{k-1}$
$2^{k-1}X2^{k-1}$
$2^{k-1}X2^{k-1}$

再来回顾一下排序

冒泡排序：O(n²)

插入排序：O(n²)

快速排序：O(nlgn)

归并排序：O(nlgn)

Name	Average	Worst	Memory	Stable
Binary tree sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Yes
Bubble sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Cocktail sort	—	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Comb sort	—	—	$\mathcal{O}(1)$	No
Gnome sort	—	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	No
In-place merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	No
Insertion sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Introsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	No
Library sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	Yes
Merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Yes
				No
				No
				es
				No
Smoothsort	—	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	No
Strand sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	Yes

使用分治思想设计的排序算法的时间复杂度降低了

http://en.wikipedia.org/wiki/Algorithms

时间复杂度的意义?

π = 3.14159265358979323846264338327950288419716939937510...

e = 2.7 ... 52353602874 ... 4709369995...

2^n vs n^2

e^n vs n^e

18446744073709551616

$1.844674407 \times 10^{19}$

2^{64}

	2	3	4	5	6	7	8	9	10
x^2	4	9	16	25	36	49	64	81	100
2^x	4	8	16	32	64	128	256	512	1024

	2	5	10
x^2	4	25	100
$5x^2$	20	125	500
x^3	8	125	10^3

时间复杂度反应的是随着问题规模的增长，计算量增长的速度的快慢

有前面的讨论可以发现，之后次数最高的项才能真正体现速度的快慢，非最高项以及所有的系数在反应增长速度方面意义都不大

算法计算复杂度的度量

f(N)和g(N)是定义在正数集上的正函数

如果存在
正常数C和自然数N_0

当$N>N_0$时，
有$f(N)<=Cg(N)$

当$N>N_0$时，
有$f(N)>=Cg(N)$

g(N)是f(N)的上界

g(N)是f(N)的下界

记做
$f(N)=O(g(N))$

记做
$f(N)=\Omega(g(N))$

$f(N)=O(g(N))$

+

$f(N)=\Omega(g(N))$

→

$f(N)=\theta(g(N))$

举例

- $3n=O(n)$
- $n+1024 = O(n)$
- $2n^2+11n+5=O(n^2)$
- $n^2=O(n^3)$
- $3n= \Omega(n)$
- $2n^2+11n+5= \Omega(n^2)$
- $n^3=\Omega(n^2)$
- $2n^2+11n+5=\Theta(n^2)$
- $3n= \Theta(n)$

分治算法的时间复杂度

把原规模为n的问题分解为a个规模为n/b子问题，T(n)代表求解规模为n的问题所需的时间， f(n)代表分解和合并所需的时间为

$$T(n) = aT(n/b) + f(n)$$

如何计算
T(n)

归并排序： $T(n) = 2T(n/2) + cn$

递归树法

$$T(n) = 3T(n/4) + cn^2$$

$$3^{\log_4 n} = n^{\log_4 3}$$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1 - \frac{3}{16}} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$= \mathrm{O}(n^2)$$

分析实例

- 用递归树法分析归并排序的时间复杂度
- 假设一个数列用快速排序，每次都能将其分解为10:1的两段，试用递归树法分析在此种情况下算法的时间复杂度