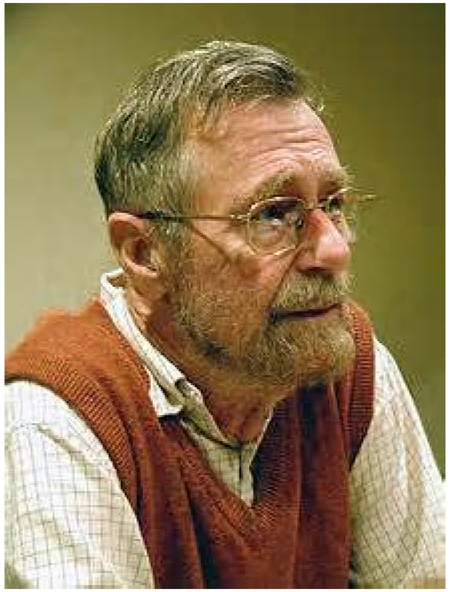
面向科学问题求解的编程实践

全局变量

介绍全局变量这个主题时，我们是心怀忐忑的。从20世纪70年代开始，正统计算机科学家都强烈反对使用全局变量，因为随意使用全局变量会引发很多问题。使程序清晰易读的关键就是局部性。人们一次只能阅读一段程序，理解这段程序所需的上下文越少，效果就越好。因为全局变量可以在程序中的很多地方被修改或读取，所以草率地使用全局变量会破坏局部性。尽管如此，全局变量有时真的很有用。



Edsger Wybe Dijkstra(1930—2002)

何谓算法

- 计算的方法，用计算的方式解决问题（不一定是计算问题）的方法，利用计算机的高效计算能力解决问题的方法
- 例如：十进制的加、减、乘、除。
- 例如：根号2的计算。
- 例如：Fibonacci数列的计算问题。

Fibonacci数列的计算问题

约1175-约1250

- 数列中每个数都是其前一个数和更前面一个数的和

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F(n-1) + F(n-2),\ n \geq 2 \end{cases}$$

- 例如：0,1,1,2,3,5,8,13,21,34，…
- 随n增大，Fibonacci数列呈指数增长。 $F_n \approx 2^{0.694n}$
- 问题：给定任意一个n，F_n是多少？F_{100}？F_{200}？
  - 计算机出现以前，人们只能够算出n不大的F_n

一个指数时间算法

- 算法如下

```
function fib1(n)
if n = 0:   return 0
if n = 1:   return 1
return fib1(n - 1) + fib1(n - 2)
```

- 考虑三个问题：
  - 1）正确性。正确，就是Fibonacci数列的定义
  - 2）执行时间。？？
  - 3）能否改进。？？

- 计算时间T(n)比Fibonacci数列本身增长得还快！
  - 由于Fibonacci指数增长，因此算法是一个指数时间算法
  - 使用目前最快的计算机，完成F_{200}的计算需要10^{19}年！
- 指数时间算法虽然正确，但是无用。

一个多项式时间算法

- 算法使用一个数列保存中间结果，避免重复计算
  - 当计算f[i]时，f[i-1]和f[i-2]已经获得。

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

- 正确性？直接使用Fibonacci数列定义，正确
- 算法2的执行时间？
  - 计算fib2(n)主要包含一个执行n-1步的循环，所以算法的执行时间是线性的，称为多项式时间算法。
    - 计算时间可以表示为输入规模n的多项式表达式
  - 计算F_{200}非常快，计算$F_{20000000}$也非常快

Euler Project

- https://projecteuler.net/

Leonhard Euler
（1707—1783）

“读读欧拉，他是所有人的老师。”

“计算机的数学就是欧拉的数学。”

Euler Project

- Problem 80: Square root digital expansion
  - It is well known that if the square root of a natural number is not an integer, then it is irrational. The decimal expansion of such square roots is infinite without any repeating pattern at all.
  - The square root of two is 1.41421356237309504880..., and the digital sum of the first one hundred decimal digits is 475.
  - For the first one hundred natural numbers, find the total of the digital sums of the first one hundred decimal digits for all the irrational square roots.

算法时间复杂度

- 运行一个程序需要多长时间？
- 做一次加法、乘法、除法需要多长时间？

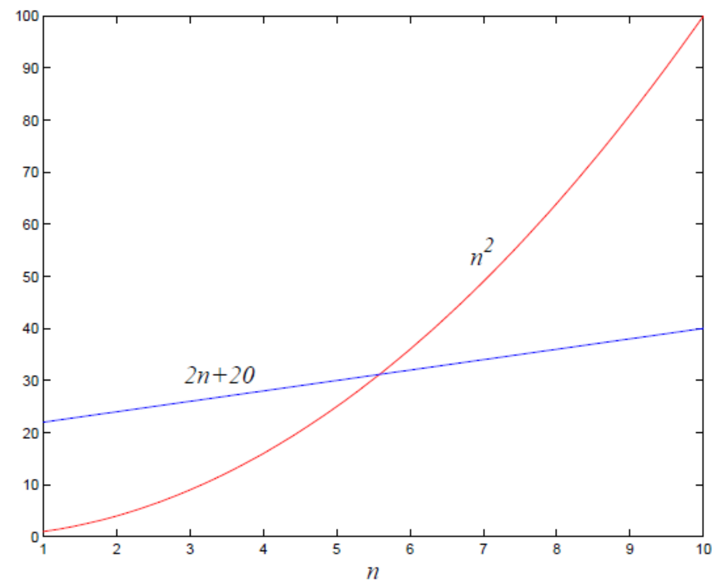
大O表示法

- 算法的确切执行时间取决于计算机硬件、编译器、执行环境等若干因素。不同的机器有不同的配置，对于研究算法本身而言，难以也没有必要确定这些信息。通常我们仅需要知道算法的大致执行时间。
  - 例如，某算法执行时间为$5n^3+4n+3$，仅考虑支配项，忽略次要项和常系数项，称该算法的执行时间为$O(n^3)$

- 定义：令f (n)和g(n)是两个从正整数到正实数的函数，如果存在常数c，使得对任意n>0，有f(n) ≤ c·g(n)，则称f=O(g)。
- 可以把f=O(g)初略地理解为“f≤g”
  - 10n=O(n)，取c=11
  - $f_1(n)=n^2$，$f_2(n)=2n+20$。根据定义，$f_2=O(f_1)$。

$$\frac{f_2(n)}{f_1(n)}=\frac{2n+20}{n^2}\le 22$$

- 当n≤5时，f_1实际上小于f_2。



- 再考虑$f_3(n)=n+1$，显然$f_3=O(f_2)$
- 注意到 $\dfrac{f_2(n)}{f_3(n)}=\dfrac{2n+20}{n+1}\le 20$，所以 $f_2=O(f_3)$也成立

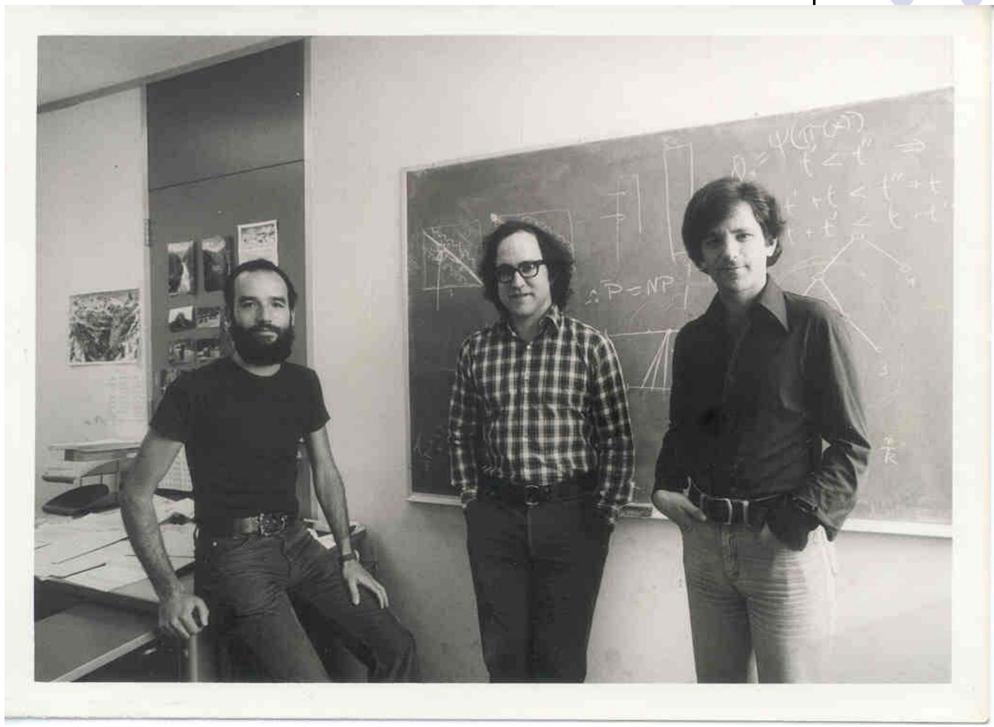
- 几个定义：
  - 如果$g=O(f)$，则称$f=\Omega(g)$，可以理解为“≥”
  - 如果$f=O(g)$并且$g=O(f)$，则称$f=\Theta(g)$
    - 上面例子中，$f_2=\Theta(f_3)$

- 大O表示法的一些原则
  - 忽略常系数，例如$14n^2$就是n^2
  - 如果a>b，则n^a支配n^b，例如n^2支配n
  - 指数项支配多项式项，例如3^n支配n^5
  - 多项式项支配对数项，例如n支配$(\log n)^3$，n^2支配$n\log n$

RSA

- 大数因子分解

- N = P * Q



Ronald L. Rivest, Adi Shamir and
Leonard M. Adleman @MIT

分治法：算法思想

- 分治法采用“分而治之”的思想解决问题，大致过程如下：
  - 将原问题分解为一组子问题，子问题是和原问题类型相同的问题，但是问题的输入规模变小
  - 递归地解决这些子问题
  - 采用合适的方法合并子问题的解，获得原问题的解

大数相乘

- 两个长为*n*-bit的数*x*和*y*相乘。将数分为长为*n/2*-bit的两部分，分别相乘

$$x = \boxed{\quad x_L \quad}\boxed{\quad x_R \quad} = 2^{n/2}x_L + x_R$$

$$y = \boxed{\quad y_L \quad}\boxed{\quad y_R \quad} = 2^{n/2}y_L + y_R.$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

两个*n*-bit数相乘的问题变为4个*n/2*-bit数相乘的和3个*n*-bit数相加，以及两个移位操作。

- *n*-bit相加可以在O(n)时间内完成，移位操作在O(1)时间内完成

- 令$T(n)$表示n-bit相乘的计算时间，则

  $$T(n) = 4T(n/2) + O(n)$$

- 通过一些方法（后面将介绍），得知 $T(n) = O(n^2)$ ，与我们在小学里学到的方法耗时相同，同时可以发现n-bit相乘比n-bit相加耗时大得多
  - 是否正确？正确！
  - 能否提高效率？可以。

- 高斯在研究复数相乘时发现，两个复数相乘可以通过三个而不是四个实数相乘来完成
  - 通常的计算过程 $(a+bi)(c+di)=ac-bd+(bc+ad)i$
  - 高斯发现：
    仅需要完成(a+b)(c+d)，ac和bd三次乘法操作

$$bc+ad=(a+b)(c+d)-ac-bd$$

Carolus Fridericus Gauss
(1777—1855)

- 回到大数相乘问题

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

因此 $xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$

$$= 2^n x_L y_L + 2^{n/2}\left((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R\right) + x_R y_R$$

- 采用新算法：
  - 每次递归地将问题分解为三个子问题，算法执行时间减为 $O(n^{1.59})$

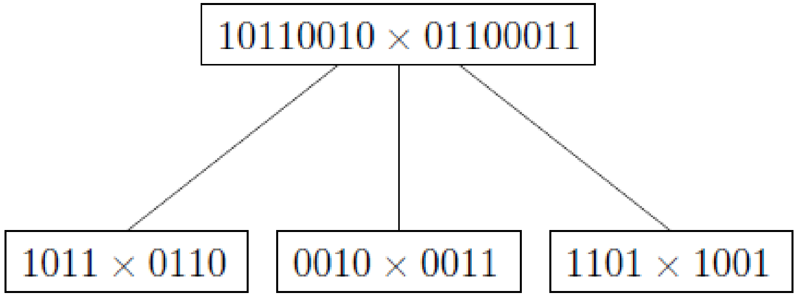
$$T(n) = 3T(n/2) + O(n)$$

- *n*-bit大数相乘算法

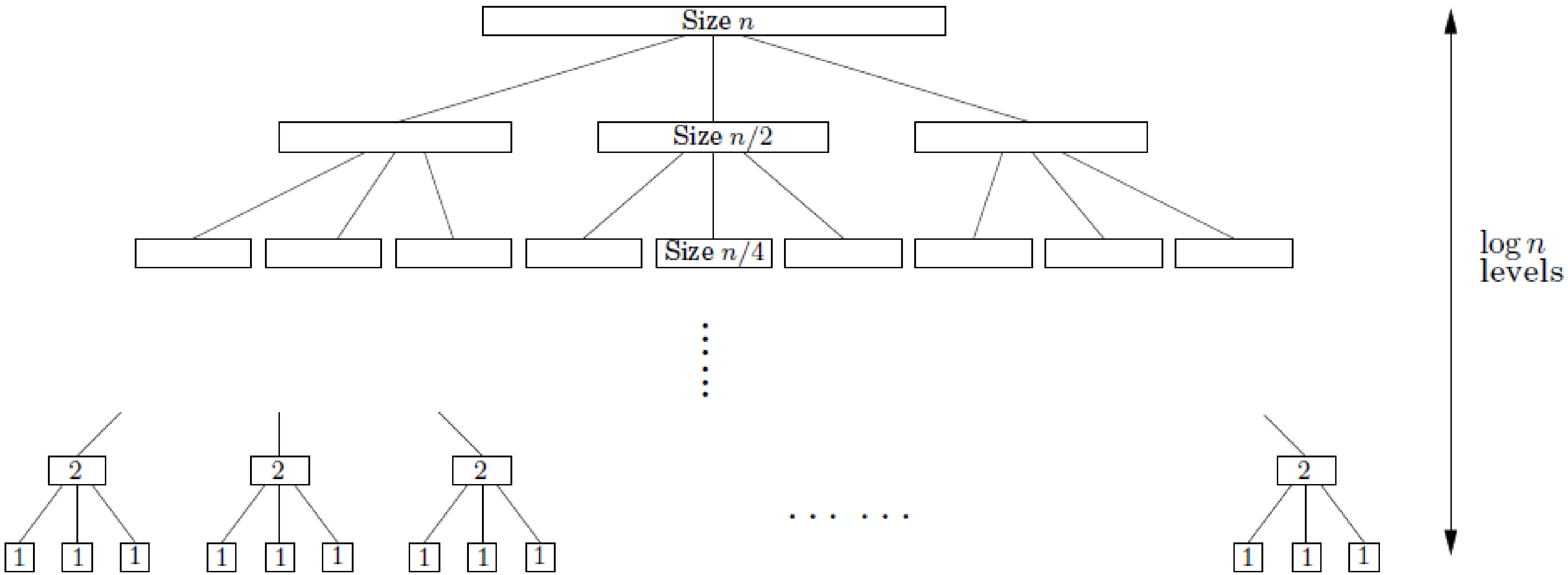
```
function multiply(x, y)
Input:  Positive integers x and y, in binary
Output:  Their product

n = max(size of x, size of y)
if n = 1:  return xy

x_L, x_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of x
y_L, y_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of y

P1 = multiply(x_L, y_L)
P2 = multiply(x_R, y_R)
P3 = multiply(x_L + x_R, y_L + y_R)
return P1 × 2^n + (P3 − P1 − P2) × 2^(n/2) + P2
```

- 上述算法的执行情况可以用一个树状结构来表示：将问题表示为树中的节点，每次问题分解，问题规模减半，产生子节点。在第($\log_2 n$)层，问题规模为1，问题分解终止。

(a)


10110010 × 01100011
1011 × 0110
0010 × 0011
1101 × 1001

(b)


Size n
Size n/2
Size n/4
2
1
log n levels

- 算法分析
  - 树一共有 $\log_2 n$ 层；
  - 在第k层上，总共产生3^k个子问题，每个子问题的输入大小是$n/2^k$，即是两个$n/2^k$-bit数相乘
  - 在第k层上，算法需要线性时间将子问题划分为更小的子问题，因此该层的算法执行时间为 $3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n)$
  - 在顶层，k=0，执行时间为 $O(n)$
  - 在最后一层，$k = \log_2 n$，执行时间为 $O\left(3^{\log_2 n}\right)$，或 $O\left(n^{\log_2 3}\right)$
  - 在每层上算法执行时间呈指数上升，从 $O(n)$ 到 $O\left(n^{\log_2 3}\right)$，所有层相加，算法执行时间为

$$O\left(n^{\log_2 3}\right) \approx O(n^{1.59})$$

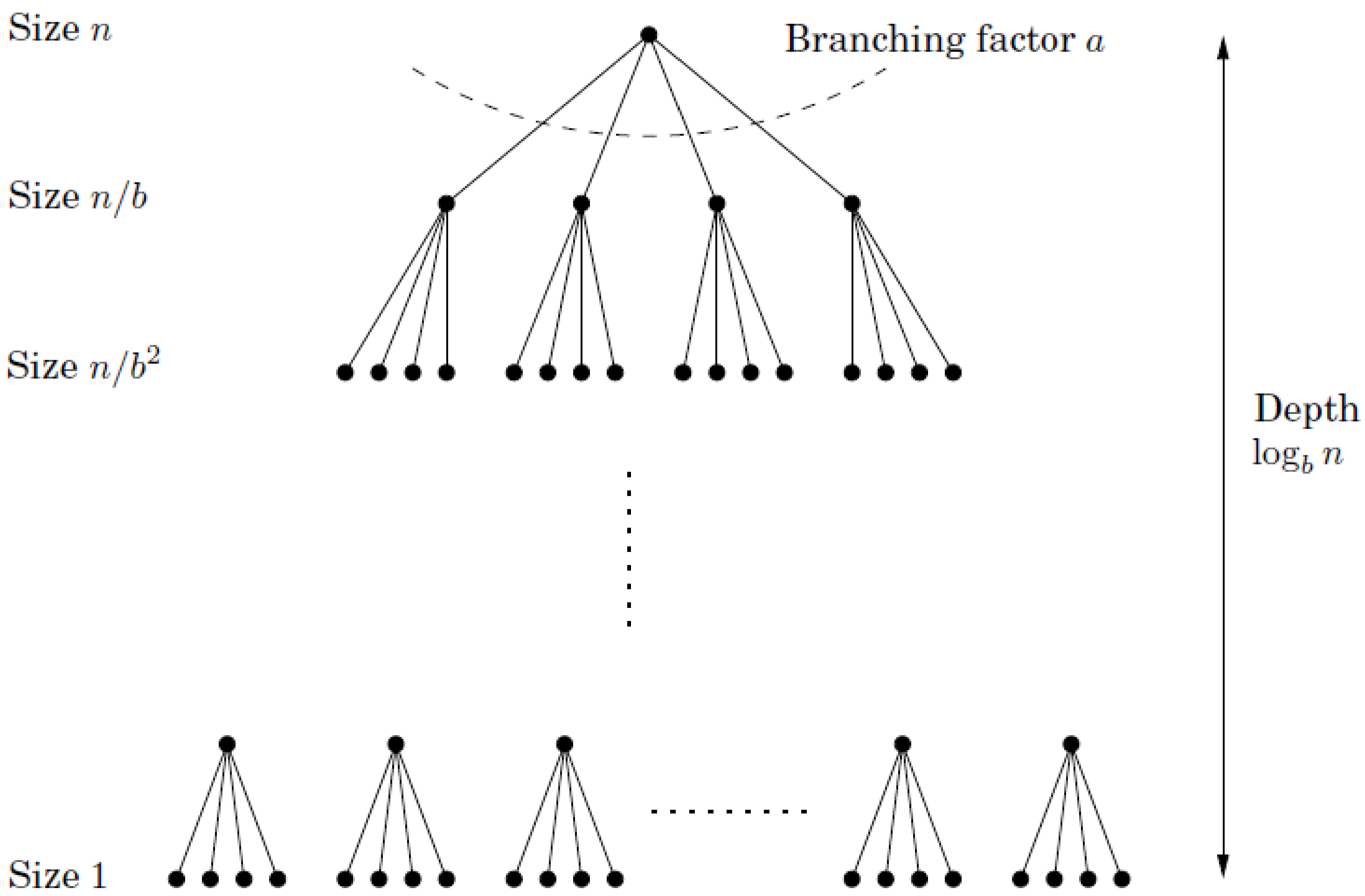
- 类似地，如果不采用高斯的方法，第k层算法时间 $4^k \times O\left(\frac{n}{2^k}\right) = 2^k \times O(n)$，
  - 第0层上执行时间$O(n)$
  - 第$\log_2 n$层上执行时间是$nO(n)$，即$O(n^2)$
  - 算法执行总时间是 $O(n^2)$

递推公式

- 分治法算法可以如下描述
  - 原问题的输入规模为*n*，通过递归的分治法，将问题分解为*a*个子问题，每个子问题的输入规模为*n/b*，并且将*a*个子问题的解组合为原问题的解需要耗时 $O(n^d)$ 。例如在大数相乘的问题中，*a=3*, *b=2*, *d=1*。
- 定理：如果 $T(n)=aT(\lceil n/b \rceil)+O(n^d)$ ，其中a>0，b>1，d≥0，则

$$T(n)=\begin{cases} O(n^d) & \text{如果} d>\log_b a \\ O(n^d \log n) & \text{如果} d=\log_b a \\ O(n^{\log_b a}) & \text{如果} d<\log_b a \end{cases}$$

- 证明：同样可以用一个树状图表示分治法的执行过程



- 顶层k=0，1个问题，规模为n，问题分解耗时O(n^d)
- 在第k层，a^k个子问题，每个规模为n/b^k，每个问题分解耗时O(n^d/b^{kd})，所有问题分解耗时 $O(n^d(a/b^d)^k)$

- 考虑公比a/b^d，分三种情况
  - 公比小于1，则每层耗时指数指数递减，算法总耗时O(n^d)，也就是第0层的耗时
  - 公比大于1，则每层耗时指数指数递增，算法总耗时
    $$n^d\left(\frac{a}{b^d}\right)^{\log_b n} = n^{\log_b a}$$
    也就是第$\log_b n$层的耗时
  - 公比等于1，每层耗时O(n^d)，所有层相加O($n^d\log n$)