精细地实现程序——浅谈OI竞赛中的常数优化

天津南开中学 何琦

摘 要

程序的运行时间主要由算法复杂度决定，算法复杂度描述了随着数据规模增大时程序运行时间的变化情况。程序常数指的是乘在瓶颈复杂度前的系数，不随数据规模增大而变化，对程序运行时间的影响相对较小。然而，对程序常数的优化，在相同数据规模的情况下可以导致程序运行时间成倍缩短，从而使得程序更加高效。本文主要介绍OI竞赛中对算法复杂度并无影响，但对程序运行时间有较为明显影响的常数处理问题，涉及相同复杂度的算法选择，实现细节优化等方面。

1　同一类型的算法常数比较与选择

对于同一道题目，往往存在多种不同算法，它们的时间复杂度都是最优或很优的，但它们的实现方式不尽相同，从而程序常数也有所不同。由于常数受实现方式和细节影响较大，以下我们采用部分不会变化的算法骨架作为常数的比较方式，会与实际情况有所出入。

1.1　复杂度相同的算法

以树状数组，线段树，平衡树等为例进行比较。复杂度均为为$O(n\log n)$。

1.1.1　使用2-分治的理由

注意到描述复杂度时我们叙述的是$O(n\log n)$而不是$O(n\log_2 n)$，这是因为对于任意常数x,y，

$$\frac{n\log_x n}{n\log_y n} = \log_x y = C$$

由此，底数不影响算法复杂度，但会对常数产生影响。

通常情况下，分治型数据结构通常采用二分，此时合并常数可以看作2，而若采用三分，合并常数为3，而$\log_2 3 > \frac{3}{2}$，似乎采用三分常数更小一些。

然而，事实上三分实现效果不如二分好，这是因为在计算均分位置时，除法运算占了瓶颈，若采p用三分，需要执行两次÷3操作，而由于操作系统为二进制，二分时只需要采用位运算即可。从而，二分常数实际上远小于三分

同时，平衡树系列数据结构也均采用二分作为旋转操作的基础，若采用三分，单是从分析角度就已经无法承受。

1.1.2 线段树

通常情况下，线段树采用自顶向下的实现方式，并可以支持动态建点，可持久化等操作。

为了通过位运算加速寻找子结点的过程，通常编号x的两个子结点分别为$2x$和$2x + 1$。

这种线段树对单点进行操作时，每层只会访问到一个节点，由于需要递归返回，常数可以看作2。而对区间进行操作时，每层最多访问4个节点，最多提取两个节点，我们以访问节点数量为常数，则这种线段树常数为4。

而一种自底向上线段树[1]，将整个数据段补齐至长度$2^k - 2$，并按照之前的编号规则，从$2^k + 1$位置填入底层数据，并向上合并。对区间进行操作时，转为开区间，直接取出底层的两个节点向上递归。

这种数据结构的优势在于，如果没有修改型懒标记的存在，每一层只需要考虑被提取的两个节点，常数为2。而如果有修改型懒标记的存在，就必须考虑到开区间的两个点的懒标记，常数增大为4。

可以看到，使用这种线段树，没有懒标记时比之前的自顶向下线段树快了一倍，究其原因则是不必自顶向下查找所需的节点位置，省去了前一半的时间。

然而，即便有修改型懒标记，这种线段树依然比自顶向下的线段树要快。这则是因为我们分析复杂度时只考虑了访问节点数量，而没有考虑自顶向下时进行的包含判断。p 综合考虑，自底向上线段树常数较优。

[1]最早由清华大学的张昆伟提出，参见参考文献[1]

而自底向上线段树的局限性在于不能处理“无法直接找到底端节点”的题目，例如在线段树上二分。所以，在允许的情况下，使用自底向上线段树效果更佳。

1.1.3 树状数组

树状数组结构与二项树较为类似，但修改和查询方式比较奇怪，也没有明显的“分治”算法的特征，实现效果通常比线段树好。现在仔细分析一下它的原理。

考虑大小为2^{k+1}的树状数组，实质是由两个大小为2^k的树状数组，其中一个的根接到另一个的根上所形成的。

再考虑线段树，大小为2^{k+1}的线段树是由两个大小为2^k的树状数组，都将根接到一个新节点上形成的。

从而，每次合并时树状数组都比线段树少用了一个节点，而少的那个节点，实质是右子树的根

实际上我们注意到，树状数组支持的操作是查询区间$[1, x]$的值，若包含某个右区间，则一定包含整个区间，只需调用父亲节点即可，所以右节点其实永远不会被用到。

树状数组修改时的位运算，作用是跳到它的最近一个存在的父亲，而查询时的位运算，作用是跳到它父亲的左兄弟以继续查询。

整个算法比线段树节约了一半的空间，但由于省去的节点不均匀，最坏常数仍是1 (这里依然是以访问节点数量为常数，对修改而言最坏情况是修改1号节点，查询而言最坏情况是查询$[1, 2^k - 1]$)

而树状数组对随机数据的期望常数为0.5，较线段树而言更优一些。

1.1.4 平衡树

平衡树的实现方式多种多样，大致分为基于旋转的平衡树和基于重构的平衡树两种。这里，我们以访问的节点数量(可以近似看做深度)+修改的指针数量定义常数。

基于旋转的平衡树，若记录父亲，每次旋转重构6个指针。

AVL深度常数为1，每平衡一层需要旋转最多两次，常数可以看作$2 \times 6 + 1 =$

12。

splay势能分析中自带3的常数，每1单位势能支持旋转1次，访问次数和旋转次数相同，常数可以看作21。

基于旋转的treap，其旋转次数有一个期望的常数界[2]，所以瓶颈在于深度，而treap的实际深度常数约在1.5到2之间。因而看起来非常优秀。实际上，由于非瓶颈部分常数较大(第二小节会描述这部分的影响)，且需要处理随机数发生器的问题(第二节会讲述)，实现效果并没有预期的好，但这仍是一个常数不大的优秀算法。

基于重构的平衡树，不必记录父亲指针，重构常数为2×重构大小。

替罪羊树，以平衡因子0.7计，深度常数大致为$\log_{\frac{10}{7}} 2$，重构常数约为$2\log_{1.2} 2$，势能总常数=深度常数×重构常数，约为16。

基于重构的treap，每次重构期望大小为深度，常数较基于旋转的treap增大2，一般只有不支持旋转时才会使用。

特别的，即便按照这种方式计算常数，与实际运行时间的比例差距还是很大，因为访问和修改指针并不是平衡树的全部操作。

而我们可以看到，平衡树系列算法常数都不小，其中属splay算法常数最大。因此，可行时最好采用预留位置的静态线段树代替平衡树。

1.2 复杂度相近的算法

有些情况下，同一个问题的不同算法中，有一些算法，它们的复杂度不是最优的，但与最优复杂度的算法相近，但常数较小，使得在现有能承受的数据规模下实际效果可以与最优算法相比。

1.2.1 堆

堆的实现有很多种，应用比较广的是数组实现的二叉堆，需要支持合并时常用的有左偏树，二项堆，还有一种时间复杂度较优的算法“斐波那契堆”。

这里同样采用访问节点数量和修改指针数量来描述常数，其中数组实现的

[2]期望旋转不超过2次，见参考文献[2]

二叉堆只需要用到前一部分。

数组实现的二叉堆，时间复杂度除Min之外全部为$O(\log n)$，但由于其采用了和线段树类似的父亲——儿子对应关系(x号节点子结点为2x和2x+1)，从而通过位运算大大加速，常数只有1。

左偏树，支持合并的二叉堆，时间复杂度与数组实现的二叉堆基本相同。由于有交换儿子操作，常数增大为2。而虽然不平衡，但操作只涉及较浅一侧的深度(DecreaseKey除外)，深度常数为1。

二项堆，记录Min时复杂度与之前的算法相同，同样采用指针记录，采用左儿子右兄弟结构，相同大小二项树合并时常数为2，最坏情况下需要将两个大小为2^k-1的二项堆合并，需合并$2\log n$次，总常数为4。在平均情况下只需涉及$\log n$棵二项树的合并(类似于树状数组)，期望常数为2。

斐波那契堆除Extract-Min和DeleteMin外复杂度均为O(1)[3]，其中每一次合并或分割带来的双向链表修改常数为4，而度数常数为$\log_{\frac{1+\sqrt{5}}{2}} 2$，从而，整个算法常数约为5.5。

综上，斐波那契堆的优越性是不可否认的，但在实际应用中，insert和extract通常差距不大，这导致另外几种实现甚至比它更快，因此通常我们使用前三种堆算法，而只有当insert比extract系列操作多达一定数量级时，才使用斐波那契堆。

1.2.2 后缀数组

后缀数组的构造分为倍增算法和DC3算法。由于这部分算法主要用到基数排序，而基数排序又由循环构成，因此以循环数量定义常数。(一次基数排序为2 个长度为n的循环和2个长度为c的循环)。

倍增算法，倍增内包含两个基数排序(常数4)+ 新rank生成(常数2)，最坏情况下要倍增全部$\log n$次才能出解，复杂度$O(n\log n)$，常数10。后续的RMQ运用经典的$O(n\log n)-O(1)$算法，预处理常数为1，查询常数为一个log运算的复杂度。

DC3算法，递归常数3，每轮递归前三关键字排序合并常数$3\times2\times(1+\frac{2}{3})=10$，递归后一次双关键字排序常数$2\times2\times(1+\frac{1}{3})=\frac{16}{3}$，归并常数3，总复杂度O(n)，常数高达55。同时，后续RMQ需要采用标准的O(n)-O(1)算法，因为过于繁琐难

[3]见参考文献[2]

以适应之前的常数定义，不予计算。

后缀数组构造的两种算法复杂度相差一个log，而两种算法常数都不小，实际运行中，DC3算法略胜一筹，但差距不大。

1.2.3 树上链系操作

对树上的某条链进行操作，通常可以采用树链剖分和动态树两种做法。[4]

这两种做法的骨架不同，所以常数单位也不同，只能大致估算，偏离实际情况较远。

动态树分析复杂度与splay类似，常数和splay 相同，以21计。

树链剖分，找LCA复杂度$O(\log n)$常数为2，线段树复杂度$O(\log n)$常数为4，总复杂度$O(\log^2 n)$常数以8计。但是注意到两个log很难同时取到，最坏情况下为$1 + 2 + ... + \log_2 \frac{n}{2}$，有常数$\frac{1}{2}$，最坏情况下总常数为4，平均情况下甚至远小于1。

综上，虽然树链剖分复杂度多一个$\log n$，但实现效果与动态树不相上下。

2 实现常数优化

在同一种算法中，也有一些细节可以决定整个程序常数的大小。这部分主要介绍在算法实现过程中进行的常数优化。

2.1 整块中的无用部分

在一些算法中，需要用到数组和循环，但并不是数组的每个位置都有意义。我们可以借此优化算法。

2.1.1 动态规划与记忆化搜索

在高维动态规划中，常常有一些状态是不合法或没有意义的，我们可以通过不计算它们来优化算法常数。

例题：NOIP2008 传纸条方格中要求找出两条不相交的左上到右下的最短路

[4] 2007年集训队杨哲曾提出过一种全局平衡二叉树，复杂度为$O(n \log n)$且常数较小，见参考文献[3]

径使得路径经过点权值之和最小。方格200×200。

最容易想到的方法：dp[x1][y1][x2][y2]表示两条路径分别走到(x1,y1)和(x2,y2)时的最小权值和。时间复杂度$O(n^4)$，本来无法通过全部数据。但是注意到不合法的状态其实有很多，采用记忆化搜索能够避开这些不合法状态，从而通过这题的全部数据。

实际上，只有到起点距离相同的两个节点才能成为dp状态，所以标程的做法为dp[距离][x1][x2]，时间复杂度为$O(n^3)$。而实际上记忆化搜索访问的合法状态也为$O(n^3)$，这个算法为我们省去了一些思考难度。

2.1.2 矩阵表示的变化量

在一些题目中，我们常常采用矩阵乘法或倍增的方式优化算法，而整个矩阵也并不是每个位置都有意义或计算的必要。

例题：NOI2013 Day2 P1 要求按一定顺序迭代y=ax+b和y=cx+d，求最终答案对p取模的值。

很容易想到利用矩阵描述迭代并倍增加速：

$$\begin{pmatrix} x & 1 \end{pmatrix} \times \begin{pmatrix} a & 0 \\ b & 1 \end{pmatrix} = \begin{pmatrix} y & 1 \end{pmatrix}$$

p而采用这个算法，矩阵乘法的常数为$2 \times 2 \times 2 = 8$次带模乘法运算，会超时。而注意到这个矩阵无论如何做乘法，右侧一列的0和1都不会变，所以我们可以把常数优化至2次带模乘法和2次带模加法，可以通过该题。

本题标算提供了这种算法[5]和一种采用等比数列模意义求和的算法。采用这种算法减少了思维难度。

2.2 自定义常数选择

在一些算法中，有一些自定义常数，它们对整个程序复杂度并无影响，但对常数有较大影响。

[5]命题人提到“若你采用这种算法，需要实现得非常精细”，这也是本文标题的来源。

2.2.1 分块算法

分块算法中，经常需要微调块的大小以减小常数。

例题：小Z的袜子给定一个长度为n的序列，m 个询问，从区间$[l_i, r_i]$中任选2个数，求相同的概率。

解法：区间端点移动1需要花费1的常数，按左端点分x块，每块按右端点排序后依次调整。左端点总移动长度$m \times n \div x$，右端点总移动长度$x \times n$。所以应当选择块数$x = \sqrt{m}$而不是通常情况下的$\sqrt{n}$。

实际测试表明，n=65536，m=262144时，选择512分块比选择256分块快了1.5倍。

2.2.2 树的分治

树上分治中比较通用的算法为基于重心二分。那么涉及两个常数问题：1、如何划分子树。2、分治到多大时应当转而使用暴力法。

常见划分算法：按任意顺序划分，过半时的那一棵子树划归到较小的一半。由于重心性质保证最大子树不超过总点数一半，这种方法最坏情况划分比例为$1 : 3(\frac{1}{4} + \frac{1}{2} + \frac{1}{4})$。

改进：按任意顺序划分，可分成三份，均不过半，此时将较小两份合做一半。显然合成的这一份不会超过$\frac{2}{3}$，而另一份不超过$\frac{1}{2}$，最坏情况下划分比例为$1 : 2$。而注意到有三棵相同大小子树的情况，不存在比1:2更优的算法，因此这种算法常数足够小。

分治大小下界：假设s是树的大小，首先一遍BFS求出size和重心，之后子树划分，两部分各BFS一遍求出各自的列表，最后合并，大约要进行6～8次遍历，并需要将子树按最坏1：2比例进行暴力。暴力时需要进行s 次遍历，则暴力优于分治的情况：$s^2 \le \left(\frac{s}{3}\right)^2 + (\frac{2s}{3})^2 + 8s$，解得$s \le 18$。

p实测结果：约20规模时暴力，程序效果最佳。

2.3 一些实现细节

实现代码时，养成注意细节的习惯，可以减小程序常数。

2.3.1 位运算

对布尔变量的整体操作，可以采用位运算加速，注意到即使是64位整型，位运算速度也是相当快的，这通常能够给程序带来$\frac{1}{64}$的常数优化，有时能使得算法能力提升一个数量级。

采用状态压缩动态规划时，尽可能采用2，4，8进制表示状态，因为通过位运算可以很快的取出需要用到的位，从而加速算法。若有多余状态，参考“整块中的无用部分”优化。

乘法和除法操作通常比加法和减法慢，于是将常用的×2，÷2转为位运算能够小幅提速。[6]

2.3.2 读入

仔细分析后我们发现，读入时需要将十进制转换为二进制，这导致读入时需要做$n\log C$次乘法，这在一些复杂度为$O(n)$或常数不大的$O(n\log n)$算法中尤其明显，注意读入一个包含10^6个整数的文件，即使使用scanf也需要1秒，输入流就更不用说了。

实际上，系统的读入函数由于考虑了全部可能出现的情况，所以确保稳定，但常数相对较大。竞赛中可以认为输入数据是符合题目叙述的，则可以忽略其中的一些情况，所以可以考虑自写读入代码，利用系统的读单个字符函数来完成读入。实现效果通常比系统读入快60 % 到80 % 。

2.3.3 随机数

系统的随机生成函数很慢，通常随机5×10^5次就需要大约1s。

自己写随机函数时，可以考虑线性迭代，二次迭代，多个函数轮流取值等方法，在基本不影响随机性的前提下尽可能减小随机函数的常数。

2.3.4 动态内存静态化

系统的内存分配函数多次调用时速度非常慢，对于主席树等动态内存开销较大的数据结构，使用动态内存可能会使得程序变慢10～20倍。解决方案是估

[6]注意运算符优先级!

计需要的内存量并提前开辟内存池，手工实现内存分配。若有必要，可以采用栈进行内存回收等操作。

2.3.5 内存访问连续性

这种问题通常在大规模矩阵乘法时体现，主要原因是数组的第一维连续变化导致调用内存位置变动较大，看以下一段代码：

```
for(int i=1; i<=n; ++i)
    for(int j=1; j<=n; ++j)
        for(int k=1; k<=n; ++k)
            c[i][j]+=a[i][k]*b[k][j];
```

这段代码在内层的k循环时b数组的调用就出现了上述问题，导致该类型矩阵乘法在200×200规模的矩阵就达到了1s。改进方案：

```
for(int i=1; i<=n; ++i)
    for(int j=1; j<=n; ++j)
        for(int k=1; k<=n; ++k)
            c[i][k]+=a[i][j]*b[j][k];
```

这段代码可以处理400×400规模矩阵的乘法。

结论：尽可能防止数组非最后一维的连续变动，如果需要，可以交换数组的两维。

3 一道经典题目的算法比较

题意：n个整数，和不超过C，现要将其分成两部分使得和尽可能接近，求最小差值。n和C规模相同。

3.1 算法一：01背包算法

$dp[i][j]$表示用前i个数能否凑出和为j的一部分。

$$dp[0][0] = 1$$

$$dp[i][j] = dp[i-1][j] | (j \geq c_i \ \&\& \ dp[i-1][j-c_i])$$

时间复杂度$O(nC)$，可以支持10^4规模的数据。

3.2 算法二：位运算优化

注意到dp值实质为布尔变量，采用bitset将每个dp值压至一个二进制位。

$$dp[0] = 1$$

$$dp[i] = dp[i-1] | (dp[i-1] << c_i)$$

时间复杂度依然为$O(nC)$，常数缩小为$\frac{1}{32}$ 至$\frac{1}{64}$，可以支持5×10^4甚至10^5规模的数据。

3.3 算法三：分块FFT

该算法由上海交通大学的郭晓旭提出。

n个数中，超过$\sqrt{C}$的数至多$\sqrt{C}$个。

不超过$\sqrt{C}$的数，可以分成至多2$\sqrt{C}$块，每块和不超过$\sqrt{C}$ 显然每一块的个数也不超过$\sqrt{C}$，应用算法1，可以在$O(C)$的时间计算出某一块可以凑出的数，则可以在$O(C\sqrt{C})$的时间内计算出所有块可以凑出的数。

试图合并两块时，设a_i和b_i分别表示两部分能否凑出i，c_i表示总体能否凑出i，则有

$$c_i = OR \ \ a_j \ \&\& \ b_{i-j} (0 \leq j \leq i)$$

将OR换成σ，则可以采用快速傅里叶变换(FFT)[7]在$O(ClogC)$时间内计算，总合并时间复杂度$O(C\sqrt{C}logC)$

之后，再将超过$\sqrt{C}$的部分按算法1转移，复杂度$O(C\sqrt{C})$ 总复杂度$O(C\sqrt{clogc})$[8]

瓶颈在于快速傅里叶变换(FFT)。由于不能采用位运算优化，只能做到2^15左右的数据规模，实现效果甚至不如算法二。

---

[7] 见参考文献[2]

[8] 通过合理分配块的大小可以做到$O(C\sqrt{ClogC})$，快速傅里叶的常数也可进入根号下，但实现效果仍不理想。

3.4 算法四：分治FFT

注意到上述算法中应用了两部分的合并，而支持合并时我们可以采用分治法解决。

和为C的若干个数，一定可以分为三部分，其中两部分的和不超过C/2，而第三部分只有一个数。递归求解两部分，合并时采用大小为C的FFT，并暴力将第三部分那一个数用O(C)的时间进行转移。

复杂度计算应用主定理，$T(C) = 2T(\frac{C}{2}) + O(C\log C) + O(C)$，得$T(C) = O(n\log^2 n)$

从而，我们得到了一个复杂度相当优秀的算法。

瓶颈依然在于快速傅里叶变换，因此常数依然很大，实现效果没有算法二好。即使在高达$5 * 10^5$的数据规模时，对n=C的情况该算法用时6s，算法2用时15s，依然没有拉开差距。

3.5 算法五：多重背包

在n很大时，有相当一部分数是相同的。那么，我们可能可以采用多重背包的算法进行加速。

最坏情况下，n个不同的数的和最小为$1 + 2 + ... + n = O(n^2)$，于是我们得出：至多有$O(\sqrt{C})$个不同的数(常数为2)。

采用多重背包的经典算法，令$dp[i][j]$表示用前i种数凑出j时，第i种数最少的使用数量。-1表示不能凑出。

$$dp[i][j] = \begin{cases} 0 & dp[i-1][j]! = -1 \\ dp[i][j-c_i]+1 & j \geq c_i \&\& 0 \leq dp[i][j-c_i] \leq num_i \\ p-1 & else \end{cases}$$

复杂度$O(C\sqrt{C})$，由于不能压位，常数为3(转移)*2(数量常数)=6，实现效果可以达到10^5的数据规模。

3.6 算法六：改进01背包

多重背包的另一个处理方式是将同样大小的包按二进制拆分，使得原来的x个相同数变为至多$O(\log x)$个不同的数。于是我们成功将n的规模缩减

为$O(\sqrt{C}logC)$[9]

而我们注意到，只要一个数的个数超过2，我们就可以继续拆分。于是我们从小到大进行拆分，最终保证每个数的个数不超过2，则最多有$O(\sqrt{C})$个数。

p进行01背包，最终复杂度$O(C\sqrt{C})$，采用算法二的位运算优化，常数为$\frac{1}{16}$到$\frac{1}{32}$，可以做到10^6规模的数据。

3.7 小结

该题目的各种算法中，复杂度最优的为算法四(分治FFT)，仅为$O(n\log^2 n)$，而实际效果最优的为算法六，复杂度为$O(n\sqrt{n})$。这是因为两种算法的常数相差了将近200倍。可见，关注算法常数可以大幅提高程序效率。

4 总结

做一个注重常数的OIer。

参考文献

[1] 张昆伟，《统计的力量——线段树全接触》

[2] Thomas H.Cormen、Charles E.Leiserson等，《Introduction to Algorithms》

[3] 杨哲，《对QTREE解法的一些研究》

[4] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》

[9]这里的复杂度实际不可能达到，所以常数非常小，已经与接下来的算法差距不大。