Bit Tricks and Applications

Yuhao Du

IIS, Tsinghua University

January 20, 2016

Word-RAM Model

在word-RAM模型中，一个数据结构被放在字长为w的RAM中。
我们假设一些1字长的运算可以在常数时间内解决，包括算术运算，位移运算，以及位与或非运算。
并且假设$w = \Omega(\log n)$。
注意：一些C++中的运算比如“__builtin_popcount()”不包括在这些运算中。
乘法和除法在这里是被允许的。

AC^0 Circuit

AC^0 circuit是一系列由与门和或门构成的常数深度，多项式大小的电路。
整数加法减法可以被AC^0计算，而乘法不可以。
一个二进制数中1的个数奇偶性也不能被AC^0计算。

"__builtin_popcount()"

如何使用word-RAM中的操作实现这个函数？
如果不能预处理？

Comparison

称(M,f)-representation为将M个$f-1$ bit的整数压入一个word内，每个整数占f个bit。
有两个(M,f)-representation的word怎么并行比较每一个大小？

Index of the Leftmost Nonzero Element

假设$f \geq \log M + 2$，怎么求这样的(M, f)-representation的最左非零位的位置？
如果是只是w个bit呢？

Merging Two Words

假设有两个有序的长度为k的列表，被表示成了一个word。如何快速地合并？

一个序列是bitonic的当它在循环意义下能被表示成两个不增或者不降的序列连接起来。

对一个bitonic的序列$x_0, x_1, \ldots, x_{h-1}$可以使用分治进行排序，令$m_i = \min\{x_i, x_{i+h/2}\}, M_i = \max\{x_i, x_{i+h/2}\}$，对于$m_i$和$M_i$ 分别排序，然后连起来。

比较可以并行完成，所以这个算法可以在$O(\log k)$时间内完成。

Conclusion

Word operation的实质是将w个bit并行处理。
这就给了一些问题能带来复杂度上的优化。
在OI中可以体现为常数的优化。

Description

两个$n * n$的布尔矩阵A, B，求矩阵乘法。
可以通过矩阵乘法算法做到$O(n^{2.38})$，然而并不能在实践中用。
Conjecture: 布尔矩阵乘法组合算法不能做到$O(n^{3-\epsilon})$
Upper Bound: $O(n^3 * poly(\log\log n)/(\log n)^4)$,[Huacheng Yu' 2015].

Description

有一个$n * n$的布尔矩阵，要求完成类似高斯消元的过程。
或者是这样一个更困难的问题：每次添加一个长度为n的布尔向量，然后维护一组基。
高斯消元可以做到和矩阵乘法一样的复杂度，更严谨地称为矩阵的LUP 分解。
下一个问题理论界进展不明。
我们来考虑这个更加困难的问题。

Solution

一个经典的做法就是维护上三角的一组基，然后位运算加速，时间复杂度$O(n^2/w)$。
这里时间复杂度指加入一个向量维护这组基的代价。
过程就是这个向量经过了$O(n)$次消去，然后以$O(1)$的代价插入了这组基中。

Solution

考虑平衡这两部分的代价。假设将连续m个基设成一块。
假设这个向量通过1次消去就能去掉这块对应的bit，那么只需经过$O(n/m)$次即可。
所以对于一个块要维护额外的信息，也就是知道了这m位就要得出这m个基的组合对应的向量。
也就是要预处理出2^m个基的组合对应的向量。
然后插入到这组基中，需要将这个基对应的块的向量组重新处理，需要2^m次操作。
去$m = 0.5 \log n$，得到复杂度$O(n^2/w \log n)$。

Description

两两之间的最短路。
Conjecture: APSP不能做到$O(n^{3-\epsilon})$。
Upper Bound: $n^3/2^{\Omega(\sqrt{\log n})}$ [Ryan Williams' 2014]

Min-Plus Product

A, B是两个$n * n$的实数矩阵，$C = A * B$有$c_{ij} = \min\{a_{ik} + b_{kj}\}$
又叫Distance Matrix Multiplication。
令APSP的复杂度为$\mathrm{APSP}(n)$，Min-plus product的复杂度为$\mathrm{MPP}(n)$。
显然有$\mathrm{MPP}(n) = O(\mathrm{APSP}(n))$，并且有$\mathrm{APSP}(n) = O(\mathrm{MPP}(n) \log n)$。
右面的式子可以通过快速幂得到。
可以证明$\mathrm{APSP}(n) = O(\mathrm{MPP}(n))$。

Min-Plus Product

和矩阵乘法一样，要做到$o(n^3)$的复杂度就是分成足够的小块。然后使用一些预先处理好的表加速运算。然而MPP比矩阵乘法困难得多。
可以首先把矩阵分成$m*m$个小块，一共$(n/m)*(n/m)$个。
假设小矩阵的乘积可以在$T(m)$内被算出，那么整个算法的复杂度为$O((n/m)^3T(m)+n^3/m)$。
如果$T(m)=O(m^{2.5})$，那么总的时间复杂度为$O(n^3/\sqrt{m})$。

Min-Plus Product

接下来考虑$T(m)$的值，就是$m*m$的小矩阵的MPP，
令$l=\sqrt{m}$。计算$C=A*B$，
首先将A划分成l个$m*l$的小矩阵$(A_1, A_2, \ldots, A_l)$，然后将B划分成l个$l*m$的小矩阵$(B_1, B_2, \ldots, B_l)^T$。
那么$C=\min\{A_k B_k\}$。如果$A_k B_k$能在$O(l^2 m)$内被算出，那么C就能在$O(m^3/l+lm^2)=O(m^{2.5})$内被算出。

Min-Plus Product

对于所有的$1 \le r < s \le l$，如果$(a_{1r} - a_{1s}, \ldots, a_{mr} - a_{ms})$, $(b_{s1} - b_{r1}, \ldots, b_{sm} - b_{rm})$ 都是排完序的，那么可以在$O(m)$ 的时间复杂度内把每个表合并起来，总共$O(l^2)$个表，总的时间复杂度$O(l^2 m)$。

记$H_{rs}[i]$为在合并的表中$a_{ir} - a_{is}$的排名，$L_{rs}[j]$ 为在合并的表中$b_{sj} - b_{rj}$的排名。

$a_{ir} + b_{rj} \le a_{is} + b_{sj} \Leftrightarrow a_{ir} - a_{is} \le b_{sj} - b_{rj} \Leftrightarrow H_{rs}[i] \le L_{rs}[j]$。

记$H[i] = H_{12}[i] H_{13}[i] \ldots H_{l-1\ l}[i]$，$L[j] = L_{12}[j] L_{13}[j] \ldots L_{l-1\ l}[j]$，那么如果知道了$H[i], L[j]$ 就能知道确定哪个k 使得$a_{ik} + b_{kj}$最小。

于是我们打一下表，$A_k B_k$就能在$O(l^2 m)$的时间内被算出了。

Min-Plus Product

考虑我们要压的表为$T[H[i], L[j]]$，为$l(l-1)/2$个1到$2m$之间的数，并且计算的时间复杂度为$O(l^2)$，那么总的时间复杂度为$O(l^2(2m)^{l(l-1)}) = O(c^{m\log m})$。
令$m = \log n/(\log c \log logn)$，那么预处理表时间复杂度为$O(n)$。
然后上面对于$(a_{1r} - a_{1s}, \ldots, a_{mr} - a_{ms})$的排序可以预处理，时间复杂度为$O(n^{2.5}\log n)$。
所以总的时间复杂度为$O(n^3(\log\log n/\log n)^{1/2})$。

Introduction

位运算或者Four Russian Method在一些多项式级别的算法上只能做到$O(\log n)$的优化，并没有太大本质上的改进。
然而在一些本来时间复杂度就是有$O(\log n)$数据结构问题中，使用这些技巧，就能有很大的进步。

Description

维护一个集合，支持插入一个数，删除一个数，查询一个数，查询最大值最小值前趋后继。
集合里为0到$U-1$的整数。
如果没有后面的操作可以使用hash table在期望$O(1)$插入删除，最坏$O(1)$的时间内查询。
使用平衡树能达到$O(\log n)$的时间复杂度，只允许使用比较操作最优。

vEB Tree

设权值为0到$U-1$，分成$\sqrt{U}$棵子树，然后用一棵树维护这$\sqrt{U}$子树，称这棵树为summary structure，然后分别递归这个结构。插入一个数要在子树内和summary structure内插入，这样的复杂度为$T(U)=2T(\sqrt{U})+O(1)$，复杂度为$T(U)=O(\log U)$，并没有改进复杂度。

vEB Tree

于是对每个节点，额外维护最大值和最小值，并且当前的最大值或者最小值不继续递归插入到下面的结构中。
所以只需要$O(1)$的代价新建一棵vEB tree。
插入一个数如果这个子树存在，那么不需要更改summary structure的内容，只需要递归到子树内，否则需要递归更改summary structure，然后新建一棵子树。
于是这样复杂度变为$T(U) = T(\sqrt{U}) + O(1)$。
可以算出$T(U) = O(\log\log U)$，删除同理。
查询一个数的后继时，首先看对应子树的最大值，如果大于这个值，就查summary structure的中后继的最小值，否则就递归进入这棵子树，时间复杂度为$O(\log\log U)$。
内存$O(U)$，使用hash table，内存可以变为$O(n)$。

x-Fast Trie

考虑一棵Trie，那么上述的复杂度都为$O(\log U)$。
插入删除的代价都是$O(\log U)$，考虑如何加速前趋后继。最大值最小值就是$+\infty$的前趋和$-\infty$的后继。
一个数插入这棵树中和这棵树的最长公共前缀可以二分，并且对每一层可以使用hash table记录，所以可以在$O(\log\log U)$的时间内解决。

x-Fast Trie

这样找到的位置是一个叶节点或者是只有一个儿子的节点。
如果是叶节点前趋后继可以用双向链表解决，否则对于这样的只有一个儿子的内部节点，使用一个指针直接连向一个儿子节点。
如果缺少了1儿子，那么就连向0子树内最大的叶子，否则就连向1内最小的叶子。
那样可以在$O(1)$内查找前趋后继。
然而这样的时间复杂度还是不优，空间复杂度为$O(n \log U)$。

y-Fast Trie

底层的$\Theta(\log U)$个数字组成一块，用平衡树维护，这步复杂度为$O(\log\log U)$。
然后每个块选出一个数，用一棵x-fast trie维护这些代表元。
一个块的大小超过$2\log U$或者小于$0.5\log U$，考虑分裂或和相邻的块的合并，并更新对应的x-fast trie。
$\Omega(\log U)$次操作才会引发上层x-fast trie的操作，所以均摊下来在x-fast trie上插入的代价是$O(1)$的。
所以插入删除的复杂度为$O(\log\log U)$。
x-fast trie部分空间复杂度$O(n/\log U) * \log U = O(n)$，下面每块平衡树空间也为$O(n)$。

Trie+Bit Tricks

还是考虑使用Trie解决这个问题，把二叉改为w叉，恰好可以一个word表示这个summary structure。
树的深度为$O(\log_w U) = O(\log U / \log w)$，插入删除的代价为$O(\log U / \log w)$。
查询前驱后继的时候按线段树的做法，然后使用一些word operation快速找到summary structure中下一个1的位置。
总的时间复杂度是$O(\log U / \log w)$，空间是$O(U)$。
使用hash table空间变为$O(n \log U / \log w)$。

Trie+Bit Tricks

假设$\log U = O(w)$，那么上面的时间复杂度为$O(w/\log w)$，
而vEB tree和y-fast trie的复杂度为$O(\log w)$。
随着w增加时间复杂度将会变高。

Fusion Tree

一种查询能做到$O(\log_w n)$的数据结构，也就是$O(\log n/\log w)$，并且空间是线性的。
结合上面的算法能得到$O(\sqrt{\log n})$的复杂度。
已经证明线性空间下，静态前趋后缀问题，使用vEB tree或者Fusion tree的复杂度是最优的。

Description

就是整数排序的意思，每个整数都能被一个word表示。
Upper Bound: $O(n\sqrt{\log\log n})$,[Han and Thorup' 2002]
确定性算法: $O(n\log\log n)$,[Han' 2002]
Main open problem: 整数排序能不能做到线性？

Solution 1

基于比较的排序，$O(n \log n)$。

Solution 2

基数排序，$O(nw/\log n)$。

Solution 3

使用Fusion tree或vEB tree得到$O(n\sqrt{\log n})$复杂度。

Solution 4

令$T(n,b)$表示对n个数bbit的整数排序的复杂度。一个word能存$k=O(w/b)$个整数。
那么使用归并排序的复杂度能在$O(n\log n)$的基础上利用乘上$\log k/k$的因子。
如果$k=\Theta(\log n\log\log n)$，那么就能在$O(n)$完成排序。
接着我们要证明$T(n,2b)\le T(n,b)+O(n)$，也就是在$O(n)$的时间内将数的位长缩小一半。
那么$O(\log\log n)$次后，数字的长度将会不超过$w/\log n\log\log n$，也就可以使用上述算法进行$O(n)$排序。
时间复杂度为$O(n\log\log n)$。

接着考虑n个长度为$2b$的整数排序问题怎么在$O(n)$的时间内将数的长度缩小一半。
首先使用2^b个bucket，对于数字x，将$x \bmod 2^b$丢入$\lfloor x/2^b \rfloor$这个桶中。用2^b个bucket辅助存储答案。
对于一个桶y，将桶内最大的数，记为m，放入答案桶y中，然后将这个数记作$(y, 2^b)$，将桶内其他数z，记作(z, y)。
这样得到了n个二元组，按照第一维排序，也就是n个长度为b的整数。
然后倒着遍历这些数字，对于(y, z)，将y放入答案桶z中。这时候答案桶中的数字有序。
2^b桶中的按顺序存放着所有桶编号，所以遍历2^b中编号，然后遍历每个桶，就能得到一个有序的序列。

Description

二维平面上有n个红点和蓝点，问有多少个红蓝点对红点两维坐标都小于蓝点。
首先假设是整点，且坐标两两不同。
由于上述排序可以在很低复杂度内完成，并且可以向浮点数推广，所以不会成为瓶颈。
于是可以离散化坐标都是在1到n之内。

Solution 1

分治，类似于归并排序的方法。
线段树或者树状数组。
时间复杂度$O(n \log n)$。

Solution 2

考虑n个点，每个点的y坐标在0到2^l-1之间。这样求点对的时间复杂度为$T(n,l)$.
使用分治，将它分成2^h个横条，第i条内点的y坐标范围为$i*2^{l-h}$到$(i+1)*2^{l-h}-1$。
然后记P_i为第i条内的点集。然后把每个点y坐标向下取整到所在的条的编号，得到点集P'。
于是P_i中点的y坐标范围为0到$2^{l-h}-1$，P'中点的y坐标范围为0到2^h-1。
我们只要把这几部分分别计算然后加起来即可。
选择一个参数L，使用不同的算法。

Solution 2

如果$l \le L$，那么它的y坐标和颜色只要$L+1$bits可以记录，也就是记录n个点需要$O(nL/w)$ 个bits。
这个时候选择$h=1$，也就是分成两部分，P_0和P_1分别计算。
计算P'时，有若干个word，$z_1, z_2, \ldots$，每个里面包含了$O(w/L)$个点。那么可以预处理出来每个word内部有多少个红点y坐标为0，蓝点的y 坐标为1，和内部的红蓝点对，可以在$O(nL/w)$的时间复杂度内解决。
$T(n,l) = T(n_0, l-1) + T(n_1, l-1) + O(nL/w)$，其中$n_0 + n_1 = n, l \le L$。
所以$T(n,l) = O(nL^2/w)$。

Solution 2

如果$l > L$，此时选择$h = L$，那么分成2^h个部分，$P_0, \ldots, P_{2^h-1}$。
对于计算P'，首先要$O(n)$的时间把它们压倒若干个整数内，然后使用上述算法在$O(nL^2/w)$内计算。
所以得到$T(n, l) = \sum T(n_i, l - L) + O(n(1 + L^2/w))$，其中$\sum n_i = n$。
于是$T(n, l) = O(n(l/L)(1 + L^2/w))$，取$L = \sqrt{w}$，那么有$T(n, l) = O(nl/\sqrt{w})$.
预处理时间复杂度为$2^{O(w)}$，取$w = 0.5 \log n$即能得到$O(n\sqrt{\log n})$的算法。