信息学中的分块思想

成都七中 2013 级 13 班 王 迪

摘要：本文主要研究了分块思想在信息学竞赛的数论和数据结构两方面的应用。

关键词：信息学 分块 数据结构 块状数组 块状链表

目录

○ 前言

Pascal 之父尼古拉斯·沃斯凭借这样一句话获得图灵奖：“算法+数据结构 = 程序。”对于数据结构，从链表、栈、队列、二叉堆、二叉查找树，到二项堆、左偏树、斐波那契堆、AVL、SBT、Treap、伸展树、红黑树、线段树、矩形树，形形色色，花样百出，让人应接不暇。而随着数据结构的发展，我们不仅能够更好的驾驭计算机解决现实问题，还能更加深刻的感受到算法的博大与优美。

我的这篇论文从算法和数据结构两方面介绍了分块的思想，着重介绍分块思想在数据结构中的应用。所谓分块，就是将规模为n的问题划分为k块，每块规模为s，那么对块内的操作和整个范围内的操作的复杂度要平均，即令$k = s = \sqrt{n}$。通过这个思想可以把$O(n)$的复杂度降到$O(\sqrt{n})$，达到优化算法的目的。

○ 分块思想在数论中的应用

【问题 1】求解$a^x \equiv b(mod\ m)$的最小非负整数解。(数据保证$1 \le a,b,m \le 10^9$，m为质数)

若a为m的倍数，那么无解；若a不为m的倍数，由m为质数我们知道a与m互质，那么$a^0, a^1, \dots, a^{m-1}$构成模m的完全剩余系。这样x的范围缩小到$0 \le x < m$，通过枚举可以在$O(m)$的时间内得到解，但因为m的范围很大，暴力枚举会超时。

设$x = pt + s(0 \le s < p)$，那么原来的同余方程可转化为$a^{pt} \equiv b \times a^{-s}(mod\ m)$，$a^{-s}$的含义是乘以$a^s$的乘法逆元。通过枚举$t$的取值将$a^{pt}(mod\ m)$的值用 Hash 映射到$pt$，即$hash\left(a^{pt}(mod\ m)\right) = pt$，再通过枚举$s$进行判断。通过观察可以知道，若$p$的值过小，那么枚举$t$的时间复杂度偏高；若$p$的值过大，那么枚举$s$的时间开销又太大。那么只有**分得尽量均匀**才好。那么可以列出下面这个式子：

$$p - 1 = \max s = \max t = \lfloor (m-1)/p \rfloor$$

通过上式我们发现，令$p = \lfloor \sqrt{m-1} \rfloor$是个不错的选择。这样一来，枚举$t$和$s$的时间复杂度都降到约$O(\sqrt{m})$，于是我们通过分块的思想使整个算法的复杂度从$O(m)$降到$O(\sqrt{m})$，而算法的核心仍然是枚举，可见分块思想的**精巧**与**强大**。

○ 分块思想在数据结构中的应用

信息学竞赛中有一类**动态维护数列**的问题，直接模拟的每次操作的复杂度达到$O(n)$，而使用**线段树**或**平衡树**能将每一次的复杂度降到$O(\log n)$。线段树和平衡树的特征是用空间换时间，而且能处理的问题有一定局限性。通过分块的思想，可以在较优的空间复杂度下，将每次操作的复杂度降低到$O(\sqrt{n})$，以较低的编码复杂度解决问题。

○ 块状数组

【问题 2】给出长度为n的一个正整数数列，完成m次操作，操作分为 2 种：$Q\ x\ y$表示询问$[x,y]$上的最大值，$M\ x\ y$表示将第x个数字修改为y。($1 \le n \le 10^5, 1 \le m \le 10^5$)

这是个经典问题，用线段树可以做到$O(m \log n)$。不过我将要说明的是通过

分块降低复杂度。

设数列为$\{a_n\}$，设块的大小$s=\lfloor\sqrt{n}\rfloor$，块的数目$t=\lceil n/s\rceil$，即a_0~a_{s-1}为第一块，a_s~a_{2s-1}为第二块，并以此类推。设数列$\{b_n\}$，$b_i=\max\{b_j|i\le j\le rb\}$，其中$rb$为$a_i$所在块的右端点。定义函数$maintain(x)$表示维护$x$所在的块的$b_i$值：

```
void maintain(int x) {

    int lb = (x / s) * s, rb = min(n, lb + s) - 1;
    if (x == rb)  // this chunk is [lb, rb]
        b[x] = a[x], x --;
    for ( ; x >= lb; x --)
        b[x] = max(b[x + 1], a[x]);
}
```

这个操作的复杂度为$O(\sqrt{n})$。那么初始化函数可以$O(n)$完成：

```
void init() {

    for (int i = 1; i <= t; i ++)
        maintain(min(n, i * s) - 1);
}
```

此时我们可以写出修改某个数的值的函数，同样是$O(\sqrt{n})$的：

```
void modify(int x, int y) {

    a[x] = y;
    maintain(x); // just need to start with x, not the right bound
}
```

最后的问题在于查询操作。这里我们分情况讨论：如果x和y隶属于同一块，那么直接枚举就行；如果它们不在同一块，那么中间的每一块的最大值可以由数列$\{b_n\}$得到，即第i块的最大值为b_{i*s}。代码如下：

```
int query(int x, int y) {

    int x_pos = x / s, y_pos = y / s;
    int ret = 0;
    if (x_pos == y_pos) {

        for (int i = x; i <= y; i ++)
            ret = max(ret, a[i]);
    }
```

```
    else {

        ret = b[x];
        for (int i = x_pos + 1; i < y_pos; i ++)
            ret = max(ret, b[i * s]);
        for (int i = y_pos * s; i <= y; i ++)
            ret = max(ret, a[i]);
    }
    return ret;
}
```

由于$query(x,y)$至多只会在一个块中进行枚举，其他的每一个块中的最大值可以$O(1)$得到，而块数有不超过$\sqrt{n}$，一块的大小也不超过$\sqrt{n}$，所以这个操作的复杂度也做到了$O(\sqrt{n})$。

到此我们利用分块的思想，以$O(m\sqrt{n})$的复杂度解决了这个问题。虽然这比起线段树的确有些不值得，但下面这个问题，块状数组就比线段树方便多了。

【问题3】给出长度为n的一个正整数数列，完成m次操作，操作分为两种：$Q\ x\ y\ k$表示询问$[x,y]$上第k小的值，$M\ x\ y$表示将第x个数修改为y。（$1 \le n \le 10^5, 1 \le m \le 10^5$）

这个经典问题的经典做法是线段树套平衡树，代码难度较大。其实可以用块状数组实现，代码量较小，代码难度也较低。

先进行分块，a_i表示原数列第i个数，b_i表示一个序列，是第i块数据排序后的序列，sz_i表示第i块数据包含的数据个数。排序可以使用插入排序，理论可以做到做到$O(\sqrt{n})$：

```
void modify(int x, int y) {

    int x_pos = x / s;
    (*lower_bound(b[x_pos], b[x_pos] + sz[x_pos], a[x])) = y;
    sort(b[x_pos], b[x_pos] + sz[x_pos]);
    a[x] = y;
}
```

这里实现的代码复杂度是$O(\sqrt{n}\log\sqrt{n})$，但这不影响整个算法的复杂度，因为这个算法的复杂度集中在查询操作上。对于询问第k小的值，我们可以先进行

二分答案，在$[x,y]$上统计比它小的数的个数，若恰好有$k-1$的比它小的数，那

```
int query(int x, int y, int k) {

    int lb = 0, rb = INF + 1, mid;
    while (lb + 1 < rb) { // [lb, rb)

        mid = (lb + rb) >> 1;
        if (count_lower(x, y, mid) >= k)
            rb = mid;
        else
            lb = mid;
    }
    return lb;
}
```

么它就是这个区间上第k小的数。

函数$count_lower(x,y,num)$返回区间$[x,y]$上比num小的数的个数。同上一道题，如果x和y隶属于同一块，直接枚举；否则在x和y所在的块中分别枚举，对于中间的块i，就在对应的b_i中进行二分查找，这样$query(x,y)$的复杂度做到了$O(\sqrt{n}\log\sqrt{n}\log RANGE_LEN)$，$RANGE_LEN$为二分答案的区间的长度。

这个原本要写树套树的问题，现在我们通过分块的思想，用数组这种简单的数据结构，以$O(m\sqrt{n}\log\sqrt{n}\log RANGE_LEN)$的复杂度解决了，在实际评测中的速度还不错。

```
int count_lower(int x, int y, int num) {

    int x_pos = x / s, y_pos = y / s;
    if (x_pos == y_pos) {

        for (int i = x; i <= y; i ++)
            ret += a[i] < num;
    }
    else {

        for (int i = x, rb = x_pos * s + sz[x_pos] - 1, i <= rb; i ++)
            ret += a[i] < num;
        for (int i = y_pos * s; i <= y; i ++)
            ret += a[i] < num;
        for (int i = x_pos + 1; i < y_pos; i ++)
```

```
            ret += lower_bound(b[i], b[i] + sz[i], num) - b[i];
    }
    return ret;
}
```

块状数组可以简洁地实现一些数列维护的问题。而有些数列维护的问题，加入了**动态插入、删除数据**的功能，这时块状数组处理起来就比较吃力了。于是，我们引入了另外一种同样利用分块思想的数据结构：块状链表。

○ **块状链表**

【问题 4】给一个初始为空的数列，完成以下几种操作：$I\ x\ n$，在第x个数前面插入n个数；$D\ x\ n$，从第x个数开始删除连续的n个数；$Q\ x\ n$，输出从x个数开始的连续n个数的和。

这个问题是 NOI2003 文本编辑器的简化版本。这个数列维护的问题涉及删除，用数组模拟的话是$O(n)$的；于是可以联想到链表，因为链表的删除可以做到$O(1)$，但是链表的定位却又是$O(n)$的。如何将链表的定位复杂度降下来呢？通过前面的例子我们不难想到解决方案：链表的每个节点存储一块数据，这块数据直接用数组存储，而块的大小正是$\lfloor\sqrt{n}\rfloor$，这样链表中的结点数达到$O(\sqrt{n})$，定位操作也就降到$O(\sqrt{n})$。

现在我们可以定义链表中的结点类型$node_t$，它应该包含数组num，数组中的元素个数sz，整块数据的和sum，以及指向下一个结点的指针$next$：

```
struct node_t {

    int num[MAX_S + 1], sz, sum;
    node_t *next;

    node_t() : sz(0), next(0) {

    }

    void refresh();
} *head = NULL; // refer to the head of the list
```

其中MAX_S表示每一块的容量上限，$refresh()$函数表示对这一块数据的和

进行更新，其实就是$O(\sqrt{n})$的循环加起来。现在我们考虑如何处理块的大小，因为这个问题不同于之前，总的元素个数在不断的变化，理论最优的分块大小也不断变化；如果要动态调整，难度较大。我们不妨退一步，设MAX_N为整个过程中可能出现的最多的数据个数，那么就令MAX_S为$\lfloor\sqrt{MAX_N}\rfloor$，并固定它为块的大小，在整个算法中不再改变。这比理论最优复杂度高，但大致还是$O(\sqrt{n})$级别的。

首先定义函数$get_pos(x)$返回第x个元素所在块的指针，并将x修改为它在它所在块的数组中的下标，这一步可以$O(\sqrt{n})$实现：

```
node_t *get_pos(int &x) {

    node_t *ret = head -> next;
    while (ret -> next && x > ret -> sz)
        x -= ret -> sz, ret = ret -> next;
    return ret;
}
```

现在我们考虑插入操作，假设定位后要在指针pos指向的块中，在数组下标为x的前面插入一段数据，那么我们可以这样做：把pos以x为分界线分裂为两块，且前一块保留$x-1$个数据，这时插入操作就相当于在这两块的中间新建一些块；又因为链表的插入操作是$O(1)$的，这样插入操作的复杂度可以做到与读入数据的复杂度同阶。插入结束后，链表中的一些块可能没有饱和，如果不将较小的块合并起来，会造成空间的浪费。这时我们需要$maintain()$函数来维护链表的长度为$O(\sqrt{n})$：从头开始遍历链表，如果有相邻两个结点的sz之和不超过MAX_S，就将它们合并。合并函数$merge(pos)$表示合并pos和它的后继结点：

```
bool merge(node_t *pos) {

    if (pos -> next -> sz <= MAX_S) {

        memmove(pos -> num + pos -> sz + 1, pos -> next -> num + 1,
sizeof(int) * (pos -> next -> sz));
        pos -> sz += pos -> next -> sz, pos -> refresh();
        pos -> next = pos -> next -> next;
        return 1;
    }
    return 0;
}
```

这时$maintain()$操作也可以实现了：

```
void maintain() {

    node_t *pos = head -> next;
    while (pos && pos -> next) {

        while (pos && pos -> next && merge(pos))
            ;
        pos = pos -> next;
    }
}
```

这样可以保证链表中的结点不超过$2\sqrt{n}$，证明如下：调用$maintain()$后，我么定义相邻两块为一个单元，每一个单元的sz必定超过$\sqrt{n}$，单元数便不超过$\sqrt{n}$，这样块的数目也控制在$2\sqrt{n}$以内了。

现在我们来实现插入操作，$ins(x,n)$表示在第x个数前面插入n个数。根据前面的分析我们知道插入时需要实现将一块数据一分为二的操作。定义函数$split(pos,x)$为以数组下标x为界限将pos指向的块一分为二，前一块保留$x-1$个数据，并将pos指针指向前一块：

```
void split(node_t *pos, int x) {

    node_t *tmp = pos -> next;
    pos -> next = node_alloc(); // use node_alloc() to alloc a new node
    pos -> next -> next = tmp;
    memmove(pos -> next -> num + 1, pos -> num + x, sizeof(int) * (pos
-> sz - x + 1));
    pos -> next -> sz = pos -> sz - x + 1, pos -> next -> refresh();
    pos -> sz = x - 1, pos -> refresh();
}
```

现在我们可以实现插入操作了：

```
void ins(int x, int n) {

    node_t *pos = get_pos(x);
    split(pos, x);
    for (int i = 1; i <= n; i ++) {

        if (pos -> sz == MAX_S) {
```

```
            node_t *tmp = pos -> next;
            pos -> next = node_alloc();
            pos -> next -> next = tmp;
            pos -> refresh(), pos = pos -> next();
        }
        scanf("%d", &(pos -> num[++ pos -> sz]));
    }
    pos -> refresh();
    maintain();
}
```

核心部分在新建块：如果当前块饱和了，那么更新这一块的sum，并在它的后面插入新的一块，再将pos指针后移。现在我们来处理删除操作。定义$del(x,n)$表示从x开始删除连续的n个数。令$y=x+n$，那么转化为删除$[x,y)$区间上的所有数。如果x和y隶属于同一块，那么直接在这一块的数组中进行删除，复杂度为$O(\sqrt{n})$；否则，令$x_pos=get_pos(x)$，$y\_pos=get\_pos(y)$，将x_pos以x为界限裂开，将y_pos以y界限断开，这样一来，x_pos的前一块的最后一个数据相当于$x-1$，y_pos的后一块的第一个数据相当于y，将这两块在链表中直接连起来就行了：

```
void del(int x, int n) {

    int y = x + n;
    node_t *x_pos = get_pos(x), *y_pos = get_pos(y);
    if (x_pos == y_pos) {

        memmove(x_pos -> num + x, x_pos -> num + y, sizeof(int) * (x_pos
-> sz - y + 1));
        x_pos -> sz -= n, x_pos -> refresh();
    }
    else {

        split(x_pos, x), split(y_pos, y);
        x_pos -> next = y_pos -> next;
    }
    maintain();
}
```

最后一个操作是求和$sum(x,n)$，返回从第x个数开始的n个数的和。这个操作和块状数组的操作类似，不再赘述，代码如下：

```
int sum(int x, int n) {

    int ret = 0, y = x + n; // return sum of [x, y)
    node_t *x_pos = get_pos(x), *y_pos = get_pos(y);
    if (x_pos == y_pos)
        for (int i = x; i < y; i ++)
            ret += x_pos -> num[i];
    else {

        for (int i = x; i <= x_pos -> sz; i ++)
            ret += x_pos -> num[i];
        for (node_t *pos = x_pos -> next; pos != y_pos; pos = pos -> next)
            ret += pos -> sum;
        for (int i = 1; i < y; i ++)
            ret += y_pos -> num[i];
    }
    return ret;
}
```

这样删除操作和求和操作都做到$O(\sqrt{n})$，而插入操作的复杂度与读入数据的复杂度同阶。整个算法的复杂度大致是$O(m\sqrt{n})$。

【问题5】题干同【问题4】，并新增一种操作：$M\ x\ n\ v$，将从第x个数开始的n个数都修改为v。

新增的操作实质上是区间修改，这里我们可以借鉴线段树中懒标记的思想。令$y = x + n$，对x和y所在的块直接修改，中间的块直接更新sum并打上标记$mark$，表示这一块的数据都应该修改为$mark$，但我偷懒并没有修改。当下一个操作用到这些打了标记的块时，先将这一块的数据修改为$mark$，清除标记后再进行下一步操作。

```
struct node_t {
    …
    int mark;
    bool mark_exist;

    void check();
    …
```

为$node_t$添加一些成员和方法，$check()$函数在$mark_exist$为真的情况下将

num数组修改为$mark$的值。我们现在分析什么时候会用到$check()$。

首先看插入操作：在x的前面插入一段数，只有x所在的块需要$check()$，而这一块又被裂开了，所以$split$操作时要先进行$check()$；插入结束后会调用$maintain()$，合并两个块时也需要对这两个块先进行$check()$。综上$split$和$merge$中需要加入$check()$：

```
void split(node_t *pos, int x) {

    pos -> check();
    …
}

bool merge(node_t *pos) {

    if (pos -> sz + pos -> next -> sz <= MAX_S) {

        pos -> check(), pos -> next -> check();
        …
}
```

然后看求和操作$sum(x,n)$：令$y=x+n$，因为要在x和y所在的块中暴力统计，所以这时要先进行$check()$。

```
int sum(int x, int n) {

    int ret = 0, y = x + n; // return sum of [x, y)
    node_t *x_pos = get_pos(x), *y_pos = get_pos(y);
    x_pos -> check(), y_pos -> check();
    …
}
```

最后我们添加$modify(x,n,v)$函数实现修改操作。同样令$y=x+n$，对x和y所在的块进行暴力修改，中间的块只是修改sum并打上标记，这样可以保证$O(\sqrt{n})$的复杂度：

```
void modify(int x, int n, int v) {

    int y = x + n;
    node_t *x_pos = get_pos(x), *y_pos = get_pos(y);
    x_pos -> check(), y_pos -> check();
    if (x_pos == y_pos) {
```

```
        for (int i = x; i < y; i ++)
            x_pos -> num[i] = v;
        x_pos -> refresh();
    }
    else {

        for (int i = x; i <= x_pos -> sz; i ++)
            x_pos -> num[i] = v;
        x_pos -> refresh();
        for (node_t *pos = x_pos -> next; pos != y_pos; pos = pos -> next)
            pos -> mark_exist = 1, pos -> mark = v, pos -> sum = pos ->
sz * v;
        for (int i = 1; i < y; i ++)
            y_pos -> num[i] = v;
        y_pos -> refresh();
    }
}
```

到此我们用块状链表解决了这个数列的动态维护问题！

块状链表还可以实现更加复杂的功能，如区间翻转等。另外，为了节约内存，可以将删除掉的结点进行回收利用，时空效率更好。笔者在 NOI2005 的维修数列一题中尝试了块状链表，在衡阳八中 OJ 上我的程序排到 Rank 1，而该题的标准做法是伸展树。由此我们再次发现了分块思想的**强大**，而分块思想又是那么的**简单**而**容易理解**！

○ 总结

通过研究学习分块思想在数论和数据结构中的应用，我们发现对于一些问题，一个已有但时间效率不够优秀的算法，通过分块的思想可以优化到一个比较低的复杂度。而且在考察数据结构的问题上，分块思想可以应用到比较多的方面上，是一种比较**普适**而**强大**的算法。在一些问题上，块状算法可以直接拿到满分，且可以较好的时空复杂度实现；而另外的一些问题上，虽然$O(\sqrt{n})$的复杂度很有压力，但可以拿到比较高的分数。所以在 OI 比赛中，当想不到正确做法时，分块思想是个很不错的**拿分策略**。所以，分块算法具有**较高的性价比**，而分块思想在信息学竞赛中是一种**实用**的算法思想。

○ 参考文献

【1】国家集训队 2005 年论文集：蒋炎武《数据结构的联合——块状链表》

【2】国家集训队 2008 年论文集：苏煜《对块状链表的一点研究》

○ 特别感谢

感谢 2012 级 15 班的尹茂帆学长在块状链表维护性质的问题上给予的帮助。