**分析优化区间各类操作的数据结构和算法**

2053385 高逸轩

**摘要：** 基于笔者中学时信息学基础，以及数据结构课程的学习知识，为实现大区间大规模各类操作（如随机访问、删除、插入、查询、子区间求和、子区间进行运算、维护子区间最值）且保证优秀时间复杂度，对线性表、分块、树状数组、线段树等数据结构与算法进行了分析。在本文中，对以上几种提到的数据结构与算法进行了实现原理的介绍与时间复杂度的计算，并对关键过程进行了代码实现的展示。同时，为验证时间复杂度理论的计算，构造了随机生成的区间，对比进行了上述数据结构和算法的运算时间对比。

**关键词：** 时间复杂度；线性表；分块；树状数组；线段树

在数据结构课程上，在数据结构课程上，我们学习了线性表的两种存储方式：顺序存储、链式存储。并分析了两种存储方式在处理不同操作时的优缺点。在信息学竞赛中，利用高级数据结构和算法对大区间进行高效率的大规模各项操作，也是入门且热门的学习内容。为此，我整理了线性表、分块、树状数组、线段树等数据结构和算法的相关内容，并对优缺点和效率对比进行了展示。在本文中，**全部的实现代码来自于笔者高中时期的实现成果，数据结构原理内容的解释来自于笔者高中时期的博客内容与OI-wiki。**

**1 线性表**

在数据结构课程上，我们学习了线性表的两种存储方式：顺序存储、链式存储。由于在课程中，对于线性表的知识已经进行了细致的学习，并且在课程设计的作业中进行了代码的实现，故**在此对线性表定义、实现方式等基础知识不再详细进行介绍，仅对各类操作的时间复杂度进行了分析。**

* 1. **顺序存储**

在C++语言中，数组即一种常见的顺序存储。其可以方便地寻找并读取数据。

利用数组的下标，容易得到，在随机访问中，操作次数是 。

当删除、插入某个元素时，由于在这个元素之后的全部元素下标都需要改变，所以所需要的操作次数为 次。

当对数组的某个子区间求和时，我们需要对这个区间进行遍历，所以所需要的操作次数是 次。

当对数组的某个子区间的全部元素进行某个运算（如全部加一，全部除以2等）时，我们需要对这个区间进行遍历的同时进行计算，所以所需要的操作次数是 次。

当维护数组的某个子区间的最值时，我们需要对这个区间进行遍历的同时记录并比较当前元素与最值的关系，并相应进行更新。所以所需要的操作次数是 次。

* 1. **链式存储**

链表是常见的链式存储结构。其通过指针域可以方便得到前驱和后继节点。

由于不能像数组一样拥有下标的访问方式，在随机访问中，我们需要对；链表进行遍历，操作次数是 次。

当删除、插入某个元素时，通过指针域的指向，仅改变对应节点的后继前驱节点信息即可，不必对整个链表的元素顺序进行调整，所以所需要的操作次数为 次。

当对链表的某个子区间求和时，我们需要对这个区间进行遍历，所以所需要的操作次数是 次。

当对链表的某个子区间的全部元素进行某个运算（如全部加一，全部除以2等）时，我们需要对这个区间进行遍历的同时进行计算，所以所需要的操作次数是 次。

当维护链表的某个子区间的最值时，我们需要对这个区间进行遍历的同时记录并比较当前元素与最值的关系，并相应进行更新。所以所需要的操作次数是 次。

* 1. **Vector矢量**

在C++的STL标准库中，提供了矢量Vector这种数据结构。其是一种动态大小的数组，可以实现一些操作，同时具有链表的动态大小、数组的下标访问的特点，在用户使用时具有两者的特点，看似包含了两者的优点。

但以插入操作(**push\_back**)为例，该函数首先检查是否当前的Vector容器是否还有备用空间，如果有就直接在备用空间上构造元素，这时的时间复杂度为。如果没有备用空间了，就扩充空间，重新配置、移动数据，释放原空间。​但在重新配置的过程中，需要遍历原来的全部数据才能复制过去。这时的时间复杂度为。所以Vector并不能实现综合链式和顺序存储方式优点。

* 1. **小结**

可以看到，顺序存储仅在随机访问操作、链式存储仅在删除、插入操作上具有优秀的表现，在其他的操作中都无法避免遍历操作从而导致时间复杂度为。这显然无法满足当数据规模足够大时，其他操作的时间效率。

1. **分块**
   1. **思想介绍**

分块是一种思想，而不是一种数据结构。其基本思想是，通过对原数据的适当划分，并在划分后的每一个块上预处理部分信息，从而较一般的暴力算法取得更优的时间复杂度。并且分块非常灵活，相较于树状数组和线段树，其通用性更好，可以维护很多树状数组和线段树无法维护的信息。著名的离线算法莫队算法也是通过分块的思想实现。

* 1. **时间复杂度计算**

分块的时间复杂度主要取决于分块的块长，一般可以通过均值不等式求出某个问题下的最优块长，以及相应的时间复杂度。

例如，对长度为n的序列执行区间 [ l , r]修改、求和操作：

我们将序列按每 s个元素一块进行分块，**并记录每块的区间和** 。最后一个块可能是不完整的（因为 n很可能不是 s的倍数），但是这对于我们的讨论来说并没有太大影响。

查询操作：

* 若 l和 r在同一个块内，直接暴力求和即可，因为块长为s ，因此最坏复杂度为 。
* 若 l 和 r 不在同一个块内，则答案由三部分组成：以 l 开头的不完整块，中间几个完整块，以 r 结尾的不完整块。对于不完整的块，仍然采用上面暴力计算的方法，对于完整块，则直接利用已经求出的区间和求和即可。这种情况下，最坏复杂度为 。

修改操作：

* 若l和r 在同一个块内，直接暴力修改即可，因为块长为 s，因此最坏复杂度为  。
* 若l和 r 不在同一个块内，则需要修改三部分：以l 开头的不完整块，中间几个完整块，以r结尾的不完整块。对于不完整的块，仍然是暴力修改每个元素的值（同时更新区间和 ），对于完整块，则直接修改区间和即可。这种情况下，最坏复杂度和仍然为  。

利用均值不等式可知，当 ，即 时，单次操作的时间复杂度最优，为 。

其他操作：当利用分块的思想，解决插入或删除数字，查询一个数是否属于这个集合，以及查找第k大的数等。必须将数字按递增顺序存储，并分割成多个块，每个块中包含 个数字。每次添加或删除一个数字时，必须通过在相邻块的边界移动数字来重新分块。

* 1. **核心代码实现**

// id 表示块的编号, len=sqrt(n) , sqrt的时候时间复杂度最优

int id[N], len = sqrt(n);

// a 数组表示数据数组, b 表示区间和, s 表示块长,因为块可能不是完整的所以要开数组

long long a[N], b[N], s[N];

// 区间加法（减法等可以同样根据此思想编写）

void add(int l, int r, long long x) {

  int start\_id = id[l], end\_id = id[r];

  // 在一个块中

  if (start\_id == end\_id) {

    for (int i = l; i <= r; i++)

        a[i] += x, s[start\_id] += x;

    return;

  }

  for (int i = l; id[i] == start\_id; i++)

    a[i] += x, s[start\_id] += x;

  for (int i = start\_id + 1; i < end\_id; i++)

    b[i] += x, s[i] += len \* x;  // 更新区间和数组(完整的块)

  for (int i = r; id[i] == end\_id; i--)

    a[i] += x, s[end\_id] += x;

}

// 区间求和%p

long long query(int l, int r, long long p) {

  int start\_id = id[l], end\_id = id[r];

  long long ans = 0;

  //在一个块里直接暴力求和

  if (start\_id == end\_id) {

    for (int i = l; i <= r; i++)

        ans = (ans + a[i] + b[start\_id]) % p;

    return ans;

  }

  for (int i = l; id[i] == start\_id; i++)

    ans = (ans + a[i] + b[start\_id]) % p;

  for (int i = start\_id + 1; i < end\_id; i++)

    ans = (ans + s[i]) % p;

  for (int i = r; id[i] == end\_id; i--)

    ans = (ans + a[i] + b[end\_id]) % p;

  return ans;

}

1. **树状数组**
   1. **数据结构介绍**

树状数组是一个查询和修改复杂度都为的数据结构。主要用于数组的单点修改或区间求和。也常用于计算前缀和、逆序对数。另外一个拥有类似功能的是线段树，两者的具体区别和联系如下：

1.两者在复杂度上同级, 但是**树状数组的常数明显优于线段树**, 其编程复杂度也远小于线段树。

2.树状数组的作用被线段树完全涵盖, 凡是可以使用树状数组解决的问题, 使用线段树一定可以解决, 但是线段树能够解决的问题树状数组未必能够解决。

3.树状数组的突出特点是其编程的**极端简洁性**, 使用lowbit技术可以在很短的几步操作中完成树状数组的核心操作，其代码效率远高于线段树。

* 1. **Lowbit**

由于补码的性质可得，一个数与它的相反数做与操作时，会返回其二进制下最右边的1的位置。例如： 6&-6=2。将6变成二进制：110。其中最右侧的1是：1**1**0，则返回的是二进制下10的值：2

long long lowbit(long long num){

    return num & -num;

}

* 1. **建立树状数组与单点修改**

利用lowbit的性质建立树状数组：

数组中第k位的值为原数组中的一段区间和，这个区间的长度是lowbit(k),终点是k。 比如：输入一个数组，那么我们所建立的对应的树状数组：

第一位（1在二进制下=**1** 二进制下的1=1）的值为输入的数组的第一位往前的一位的和，也就是第一位。

第二位（2在二进制下=**1**0 二进制下的10=2）的值为输入的数组的第二位往前两位的和，第一位和第二位。

第三位（3在二进制下=1**1** 二进制下的1=1）的值为输入的数组的第三位往前一位的和，也就是第三位。

第四位的值（4在二进制下=**1**00 二进制下的100=4）的值为输入的数组的第四位往前四位的和，也就是第一位，第二位，第三位以及第四位。

依次类推，我们建立了树状数组，代码实现如下：

void build(long long s,long long num){

    for(long long i = s;i <= n;i += lowbit(i))

        tree[i]+=num; //当s在i的范围内,第num位数组加上num

}

下面给出一个树状数组建立的样例辅助理解：

假设输入的n=5，输入的数为1 5 4 2 3（样例）

输入第一个数时s=1，num=1。加上的树状数组数组位数为第一位，第二位，第四位。树状数组为： 1 1 0 1 0

输入第二个数时s=2，num=5，加上的位置为第二位，第四位。数组为1 6 0 6 0

输入第三个数时s=3，num=4，加上的位置为第三位。数组为1 6 4 10 0

输入第四个数时s=4，num=2，加上的位置为第四位。数组为1 6 4 12 0

输入第五个数时s=5，num=3，加上的位置为第五位。数组为1 6 4 12 3

至此，建树操作已经完成了。可以发现这个操作的本质就是修改树状数组的值，所以它**也是单点修改的函数**。

* 1. **区间求和**

区间求和时，我们采用两个前缀和相减的方式来间接求解。求前缀和的思想与反向建树相似，代码如下：

long long ask(long long s){

    long long ans=0;

    for(ll i=s;i>=1;i-=lowbit(i))

        ans+=tree[i]; //建树的反操作

    return ans;

}

再求解区间[ l , r]的区间和时，用ask(r)-ask(l-1)即可得到答案。

以建树时所举的数组为例子，求第三项和第五项之间的和。 本质上就是求5的前缀和与2的前缀和的差。

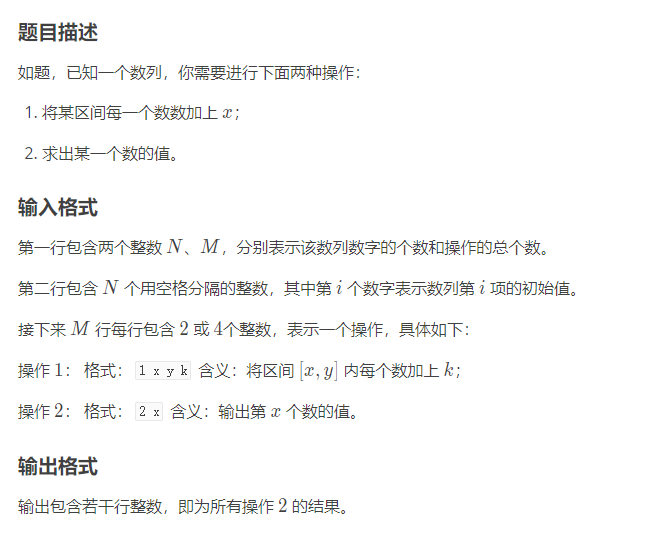
先求5的前缀和。所求的就是第5（10**1**）项+第4（**1**00）项就是12+3=15。

再求2的前缀和。所求的就是第2（**1**0）项就是6。

最后作差就是15-6=9。

检验得到，3到5位的求和就是4+2+3=9。求解正确。

* 1. **代码实现**

另外，为实现快速进行区间加等操作，还可以利用差分的思想，维护树状数组的前缀和。下附树状数组解决此问题的经典例题及其代码：

    #include <iostream>

    #include <algorithm>

    #include <cstdio>

    #include <cstring>

    #include <cmath>

    #include <queue>

    using namespace std;

    int n,m;

    int input[500010];

    int tree[500100];

    int lowbit(int x){

        return x & -x;

}

//区间加

    void add(int x,int k){

        while(x<=n){

            tree[x]+=k;

            x+=lowbit(x);

        }

}

//查询

    int search(int x){

        int ans=0;

        while(x!=0){

            ans+=tree[x];

            x-=lowbit(x);

        }

        return ans;

    }

    int main(){

        cin>>n>>m;

        for(int i=1;i<=n;i++)

            cin>>input[i];

        for(int i=1;i<=m;i++){

            int a;

            scanf("%d",&a);

            if(a==1){

                int x,y,z;

                scanf("%d%d%d",&x,&y,&z);

//转换为两个区间的差分

                add(x,z);

                add(y+1,-z);

            }

            if(a==2){

                int x;

                scanf("%d",&x);

//利用差分单点查询

                printf("%d\n",input[x]+search(x));

            }

        }

    }

1. **线段树**
   1. **数据结构介绍**

线段树是一种[二叉搜索树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%90%9C%E7%B4%A2%E6%A0%91)，与[区间树](https://baike.baidu.com/item/%E5%8C%BA%E9%97%B4%E6%A0%91)相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。线段树是[平衡二叉树](https://baike.baidu.com/item/%E5%B9%B3%E8%A1%A1%E4%BA%8C%E5%8F%89%E6%A0%91)，最后的子节点数目为N，即整个线段区间的长度。线段树之所以称为“树”，是因为其具有树的结构特性，其本身是专门用来处理区间问题的。使用线段树可以快速的进行查询、区间求和、区间加、区间乘、维护区间最值等一系列操作，[时间复杂度](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6)为。而未优化的[空间复杂度](https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6)为，因此有时需要离散化让空间压缩。

* 1. **与分块对比**

对于线段树中的每一个子节点而言，都表示整个序列中的一段子区间；对于每个叶子节点而言，都表示序列中的单个元素信息；子节点不断向自己的父亲节点传递信息，而父节点存储的信息则是他的每一个子节点信息的整合。这种思想与分块相同。

可以认为，线段树就是分块思想的树化，或者说是对于信息处理的二进制化——用于达到级别的处理速度。而分块的思想，则是可以用一句话总结为：通过将整个序列分为有穷个小块，对于要查询的一段区间，总是可以整合成k个所分块与m个单个元素的信息的并。但普通的分块不能高效率地解决很多问题，所以作为级别的数据结构，线段树应运而生。

虽然线段树的时间效率要高于分块,但是实际上分块的总合并次数不会超过​，但是线段树在最坏情况下的合并次数显然是要大于这个时间效率的。

分块的应用范围也广于线段树。根据离散数学课程所学的代数系统方面知识，我们知道，线段树维护的信息在很多时候可以认为是满足（幺）半群的性质的信息。幺半群具有以下性质：封闭性、结合律、存在幺元。我们观察到线段树上的信息一般满足这样的性质，例如自然数域上的加法与乘法。

事实上线段树虽然效率较高，但是只能维护具有结合律性质的的信息，比如区间/、*，*等，但是不具有结合律的信息就不能维护；而分块则灵活得多，可以维护很多别的东西。可以认为，分块的本质就是**优雅的暴力**。

* 1. **与树状数组对比**

时间复杂度上，它们都是级别。但是在查询时，树状数组最坏情况是，但是线段树是所有情况都是，稍慢于树状数组。且线段树常数巨大，甚至超过了一般规模的。所以在时间效率方面相比，树状数组优于线段树。

空间复杂度上，树状数组完胜于线段树。普通的线段树要开2倍到4倍内存（推荐4倍，防止访问越界），但是树状数组一倍大小空间即可。

在适用范围方面，线段树则比树状数组适用于更多的情况。不仅仅是区间、单点的查询修改，还有标记等等，可以用于模拟、DP等等。而且空间经过离散化以后也可以相对压缩，所以适用范围线段树更加广一些。

* 1. **父子关系构建**

创建结构体node，记录每个线段树上每个节点的l,r 位置，其中的区间和sum，加运算的lazy标记add，乘运算的lazy标记mul。（lazy标记后续会介绍）

1. struct node{
2. long long l,r,sum,add,mul;
3. }tree[4\*N+5];

由于二叉树的自身特性，对于每个父亲节点的编号i,他的两个儿子的编号分别是2i和2i+1，以宏定义来取左右两个儿子：

#define lson root<<1

#define rson root<<1|1

位运算操作<<1即为\*2, <<1|1为\*2+1.

* 1. **push\_up与线段树的建立**

push\_up操作的目的是为了**维护父子节点之间的逻辑关系**。当我们递归建树或者进行区间修改（后续介绍）时，对于每一个节点我们都需要遍历一遍。在这里的**递归实际意义是先向树的底层叶子节点递归，然后从底层向上回溯**。所以开始递归之后，必然是**先去整合子节点的信息**，再向它们的祖先回溯整合之后的信息。在这里可以观察得到，实际上push\_up是在合并两个子节点的信息到父节点，所以需要其维护的信息满足结合律。实现代码如下：

1. void push\_up(long long root){
2. tree[root].sum=tree[lson].sum+tree[rson].sum;
3. }

对于建树，由于二叉树自身的父子节点之间的可传递关系，所以可以考虑递归建树，并且在建树的同时，我们应该维护父子节点的关系，在初始化树的各节点的同时，不断将区间二分，向下递归构造节点。实现代码如下：

void build(long long l,long long r,long long root){

    if(l==r){

        tree[root].add=0;

        tree[root].mul=1;

        tree[root].l=l;

        tree[root].r=r;

        scanf("%lld",&tree[root].sum);

        return;

    }

    tree[root].add=0;

    tree[root].mul=1;

    tree[root].l=l;

    tree[root].r=r;

    long long mid=(l+r)>>1;

    build(l,mid,lson);

    build(mid+1,r,rson);

    push\_up(root);

}

* 1. **Lazy标记与push\_down进行区间修改**

在线段树上，我们不讨论单点修改，因为单点修改就是区间修改的一个子问题，即区间长度为1时进行的区间修改操作。

对于区间操作，我们考虑引入“lazytag”（懒标记）——之所以称其“lazy”，是因为原本区间修改需要通过先改变叶子节点的值，然后不断地向上递归修改祖先节点直至到达根节点，时间复杂度最高可以到达的级别。但当我们引入了懒标记之后，区间更新的期望复杂度就降到了的级别且甚至会更低。

## （1）首先先来从分块思想上解释如何区间修改：

分块的思想是**通过将整个序列分为有穷个小块，对于要查询的一段区间，总是可以整合成k个所分块与m个单个元素的信息的并**()

那么我们可以反过来思考这个问题：对于一个要修改的、长度为l的区间来说，总是可以看做由一个长度为和剩下的元素（或者小区间组成）。那么我们就可以先将其拆分成线段树上节点所示的区间，之后分开处理：**如果单个元素被包含就只改变自己，如果整个区间被包含就修改整个区间.**在线段树里，无论是元素还是区间都是线段树上的一个节点，**所以我们不需要区分区间还是元素**。

## （2）懒标记

首先，懒标记的作用是记录每次、每个节点要更新的值，也就是，但线段树的优点不在于全记录（全记录的速度依然很慢），而在于**传递式记录**：

若整个区间都被操作，记录在公共祖先节点上；只修改了一部分，那么就记录在这部分的公共祖先上；如果只修改了自己的话，那就只改变自己。在这知乎，如果我们采用上述的优化方式的话，我们就需要在每次区间的查询修改时push\_down一次，以免重复或者冲突。

## （3）push\_down

push\_down操作，其实就是push\_up的逆向思维(但不是逆向操作)：当进行区间修改时，因为修改信息存在父节点上，所以要由父节点向下传导，对其子节点的lazy tag进行更新*。*

在进行push\_down操作时，值得注意的是，首先要在开始回溯时执行push\_up。因为是向上传导信息；那我们如果要让它向下更新，就调整顺序，在向下递归的时候push\_down*。*

代码如下：（注意运算符的优先级）

void push\_down(long long root){

    tree[lson].sum=tree[lson].sum\*tree[root].mul+tree[root].add\*(tree[lson].r-tree[lson].l+1);

    tree[rson].sum=tree[rson].sum\*tree[root].mul+tree[root].add\*(tree[rson].r-tree[rson].l+1);

    tree[lson].add=tree[lson].add\*tree[root].mul+tree[root].add;

    tree[rson].add=tree[rson].add\*tree[root].mul+tree[root].add;

    tree[lson].mul=tree[lson].mul\*tree[root].mul;

    tree[rson].mul=tree[rson].mul\*tree[root].mul;

    tree[root].add=0;

    tree[root].mul=1;

}

//执行区间加

void add(long long l,long long r,long long root,long long k){

    if(tree[root].l==l&&tree[root].r==r){

        tree[root].add=(tree[root].add+k);

        tree[root].sum=(tree[root].sum+(r-l+1)\*k);

        return;

    }

    long long mid=(tree[root].l+tree[root].r)>>1;

    push\_down(root);

    if(r<=mid) add(l,r,lson,k);

    else if(l>mid) add(l,r,rson,k);

    else add(l,mid,lson,k),add(mid+1,r,rson,k);

    push\_up(root);

}

// 执行区间乘

void mul(long long l,long long r,long long root,long long k){

    if(tree[root].l==l&&tree[root].r==r){

        tree[root].mul=tree[root].mul\*k;

        tree[root].add=tree[root].add\*k;

        tree[root].sum=tree[root].sum\*k;

        return;

    }

    long long mid=(tree[root].l+tree[root].r)>>1;

    push\_down(root);

    if(r<=mid) mul(l,r,lson,k);

    else if(l>mid) mul(l,r,rson,k);

    else mul(l,mid,lson,k),mul(mid+1,r,rson,k);

push\_up(root);

}

对于复杂度而言，由于完全二叉树的深度不超过，那么单点修改显然是的。区间修改的话，由于我们进行操作的总区间至多分为个子区间，又因为对于每个子区间的查询是的，所以复杂度自然是*。* 但由于每次对子区间查询时会执行多项操作，所以线段树的常数通常比较大。

**4.7 区间求和查询**

由于是信息的整合，所以还是要用到分块思想，具体步骤与之前类似，不再详细介绍。下附代码：

long long query(long long l,long long r,long long root){

    if(tree[root].l==l&&tree[root].r==r){

        return tree[root].sum;

    }

    long long mid=(tree[root].l+tree[root].r)>>1;

    push\_down(root);

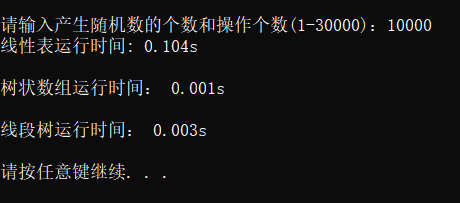
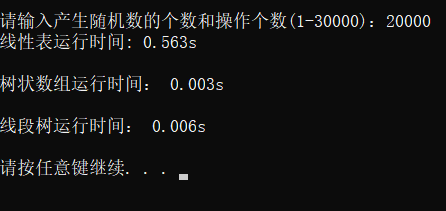
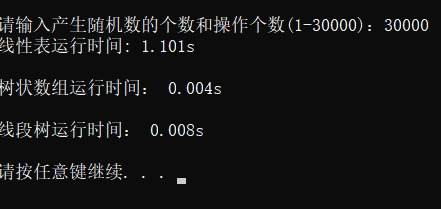
    if(r<=mid) return query(l,r,lson);

    else if(l>mid) return query(l,r,rson);

    else return query(l,mid,lson)+query(mid+1,r,rson);

}

**5 效率对比**

在以上的内容中，我们着重介绍了树状数组、线段树两种高级数据结构的原理，并对其所能执行的功能和时间复杂度进行了分析。下面我通过编写程序实际测试了线性表、树状数组、线段树的工作效率。由于其所能实现功能不同，所以在测试时，我们仅测试了区间求和、区间加、随机访问这三种操作。用随机数来生成初始的区间序列，并产生对应的随机操作，部分测试结果如下：

**可以发现，对于输入的数据n，大致符合了我们对线性表操作时间复杂度为，树状数组和线段树操作时间复杂度为 的预估。其中，由于线段树的常数较大，并且树状数组没有采取递归的方式，线段树的时间效率一般为树状数组的2倍左右。**

为优化此问题，山西晋中的**张昆玮**用自己的方式改进了线段树，命名为**zkw线段树**，是OI界的一个重要的创新。**zkw线段树采用的写法非递归，代码简短，结合位运算速度快,常数小（尤其在差分区间更新时）**，它采用的是堆结构，构造一棵满二叉树。在此不再具体给出zkw线段树的实现，但其代码的简短精悍，让人在阅读和理解时候啧啧称叹。如老师有需求，可阅读以下链接：

<https://www.cnblogs.com/Judge/p/9514862.html>

笔者自身水平有限，目前不能像金牌和姚班的牛人一样用自己的方式改进某些已经存在已久的算法或数据结构。但希望在未来的学习过程中，我能通过沿着前人的脚步一步步向前，找到自己的所得所获。

在本文中，**全部的实现代码来自于笔者高中时期的实现成果，数据结构原理内容的解释来自于笔者高中时期的博客内容与OI-wiki。**

**12/28/2021 11:31:10 PM**