线段树上机报告

刘浚哲
北京大学物理学院 *1500011370*

October 15, 2017

1. 线段树简要介绍

线段树，类似区间树，是一个完全二叉树，它在各个节点保存一条线段（数组中的一段子数组），主要用于高效解决连续区间的动态查询问题，由于二叉结构的特性，它基本能保持每个操作的复杂度为 $O(\log N)$. 性质：父亲的区间是 $[a,b]$,$(mid=(a+b)/2)$, 那么左儿子的区间是 $[a,mid]$，右儿子的区间是 $[mid+1,b]$. 构建一个线段树需要的空间为原数组大小的四倍.

例如对于数组 $[2,5,1,4,9,3]$ 可以构造如下的二叉树:

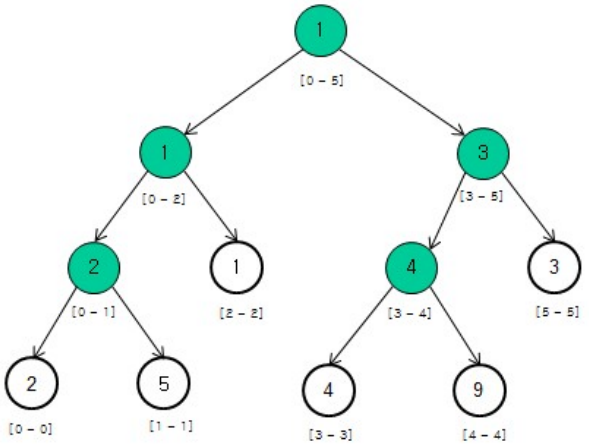

图 1：线段树示意

由于线段树的父节点区间是平均分割到左右子树，因此线段树是完全二叉树，对于包含 n 个叶子节点的完全二叉树，它一定有 $n-1$ 个非叶节点，总共 $2n-1$ 个节点，因此存储线段是需要的空间复杂度是 $O(n)$. 一般利用数组来存储线段树, 利用下标变换可以迅速准确地找到节点的左右子节点. 若数根节点的下标为 0, 则对于某一节点 i, 其左子节点下标为 $2i+1$, 右子节点下标为 $2i+2$; 若数根节点的下标为 1, 则对于某一节点 i, 其左子节点下标为 $2i$, 右子节点下标为 $2i+1$.

2.A Simple Problem with Integers

题目描述:

给定 N 个数的序列，有 Q 个操作. 操作分两种：1. 将某个区间内的所有数都加上某个数.2. 计算某个区间的所有数之和并输出.

题目分析:

这是十分典型的线段树类型的题目: 给定一组数, 我们用数组来存储. 对于两种操作:1. 计算某区间内所有数之和,2. 某区间内所有数都加上某个数来说, 如果利用普通的暴力解法, 即主动查找区间并依次求和, 或加上某一个值, 最复杂的情况会达到 $O(n)$ 的量级. 因此更好的方式是构建一个线段树. 节点类型为一个结构体, 内部保存的信息为:

```
typedef struct node
{
    int left, right;
    int64_t value; //对叶子节点来说它是原数列的元素, 对非叶子节点为线段中元素之和
    int flag; //延时标记,0为无标记, 1表示当前节点以及整棵子树都被标记
}segment;
```

叶节点的 value 值为其所保存的区间对应原数组中的值, 非叶节点的 value 值为两个子节点的 value 值之和.left 和 right 为其左右区间. flag 是所谓的" 延迟标记", 它表示: 若进行的操作是对一段区间进行的, 那么我们只访问到刚好被包含的区间线段, 而不再向下进行遍历.

若某一节点记上了延迟标记, 表示当前节点以及它的子树全都被标记, 即一段区间内所有的数都被处以某种操作. 这种思想的精髓在于标记节点以下的信息暂时不更新. 换句话说, 若某一节点被标记, 则只更新当前被标记的节点, 被标记点以下的节点信息保持不变. 这样做的好处是: 我们没有必要在要求区间操作的情况下去处理单个数据的细节, 比如我们求某一区间的所有数之和, 我们在囊括了这一区间的节点进行和的处理即可, 而不需要再向下去修改单个叶节点的值. 在修改了本节点的信息之后, 标记自身; 这样在下一次需要访问其两个子节点的时候, 就可以进一步修改子节点的信息.

在引入延迟标记之后, 可以进行对一个区间的操作. 对于求给定区间内的所有数之和, 只需要查找符合给定区间的 value 值即可, 没找到则向左右子树分别进行递归. 而若是将给定区间内的所有数都加上某一值, 则也可以按照对左右子树分别递归处理的方法进行. 其中需要判断当前进行的节点是否已经被标记, 被标记说明该节点之前已经被处理过, 但还未进行信息的更新, 此时需要进行相应的更新操作, 并将标记继续向左右节点进行传递.

数据结构与算法:

节点类型为一个结构体, 内部保存的信息为:

```
typedef struct node
{
    int left, right;
    int64_t value; //对叶子节点来说它是原数列的元素, 对非叶子节点为线段中元素之和
    int flag; //延时标记,0为无标记, 1表示当前节点以及整棵子树都被标记
}segment;
```

其中 value 为 $int64\ t$ 类型, 是一个 64 位的整数, 因为考虑到根节点的 Value 值可能很大 (区间内所有数的和), 超过了 32 位整数能够表示的范围, 因此将其定义为 long long 型. 声明线段树与原数组为:

```
#define maxsize 1000
segment segtree[maxsize];
int integers[maxsize];
```

根据原数组进行的建树函数:

```
void buildsegtree(int id, int l, int r) //根节点下标为 id, 对应区间为[i,j],建立一个线段树。若从原数组建树, 则调用build(1,1,n)
{
    segtree[id].left=l;
    segtree[id].right=r; //初始化为其表示的区间
    segtree[id].flag=0; //初始无标记
    if (l==r)
        cin>>segtree[id].value;

    else
    {
        int mid=(l+r)/2;
        buildsegtree(id*2, l, mid); //建左子树
        buildsegtree(id*2+1, mid+1, r); //建右子树
        segtree[id].value=segtree[id*2].value+segtree[id*2+1].value;//各非叶节点value值为两子节点 value 值之和,也即相应区间总和
    }
}
```

延迟标记函数: 判断当前节点是否被标记, 若被标记, 说明两个字节点都需要进行操作, 则对两个子节点进行相应的信息更新, 并将标记传给左右子节点, 最后消除本身的标记.

```
void pushdown(int root) //延迟标记函数
{
    if (segtree[root].flag!=0) //若被标记
    {
        segtree[root*2].flag+=segtree[root].flag; //因为该节点可能被标记了不止一次,因此用+=,根据被标记的次数进行多次信息更新处理
        segtree[root*2+1].flag+=segtree[root].flag;
        segtree[root*2].value+=(int64_t)segtree[root].flag*(segtree[root*2].right-segtree[root*2].left+1); //直接根据区间的大小修改和的信息
        segtree[root*2+1].value+=(int64_t)segtree[root].flag*(segtree[root*2+1].right-segtree[root*2+1].left+1);
        segtree[root].flag=0;
    }
}
```

区间增值函数: 对某一区间 $[i, j]$, 所有数都加上某一值 addnum. 可利用递归方法进行操作, 先判断当前节点所保存的区间是否与给定区间有重合部分, 若没有交集则返回, 若完全被包含在内则进行增值处理, 若重合部分为当前节点的左子树, 则递归操作左子树, 否则递归操作右子树. 最后, 由于子节点的改动, 需要在最后回溯更新对应的父节点

```
void updateSegtree(int id,int l, int r, int addnum) //区间[i,j]内所有数都加上 addnum, 当前节点下标为 id
{
    if (segtree[id].left>r||segtree[id].right<l) //如果所给区间与id包含区间没有交集, 则返回
        return;

    if (segtree[id].left>=l&&segtree[id].right<=r) //若当前线段被包含于所给区间
    {
        segtree[id].flag+=addnum;
        segtree[id].value+=(int64_t)addnum*(segtree[id].right-segtree[id].right+1); //直接根据区间的大小修改和的信息
        return;
```

```
    }

    pushdown(id); //判断该节点是否已经被标记过,有则进行处理,并将节点的标记传给两个子节点。

    int mid=(segtree[id].left+segtree[id].right)/2; //如果当前节点横跨两个子树,则对左右子树进行递归处理
    if (l<=mid) //若与左子节点有重合部分则操作左子树
        updateSegtree(id*2, l, r, addnum);
    if (r>=mid) //若与右子节点有重合部分则操作右子树
        updateSegtree(id*2+1, l, r, addnum);
    segtree[id].value=segtree[id*2].value+segtree[id*2+1].value; //回溯更新父节点
}
```

求和函数: 给定区间 $[i, j]$ 所有数之和, 和上边一样进行递归操作. 首先判断当前当前节点所保存的区间是否与给定区间有重合部分, 若没有交集则返回, 若完全被包含在内则返回该节点保存的 value 值 (即区间的和), 若重合部分为当前节点的左子树, 则递归操作左子树, 否则递归操作右子树. 由于不存在修改操作, 故不需回溯修改父节点:

```
int64_t searchSumSegtree(int id, int l, int r) //查询某区间[i,j]内部元素之和
{
    if (segtree[id].left>=l&&segtree[id].right<=r) //如果包含在区间内
        return segtree[id].value;

    else
    {
        pushdown(id); //判断该节点是否已经被标记过,有则进行处理,并将节点的标记传给两个子节点。
        int64_t result=0;
        int mid=(segtree[id].left+segtree[id].right)/2;
        if (r>mid) //若区间位于左子树内
            result+=searchSumSegtree(id*2+1, l, r); //搜寻左子树
        if (l<=mid) //若区间位于右子树内
            result+=searchSumSegtree(id*2, l, r); //搜寻右子树
        return result;
    }
}
```

主函数:

```
int main()
{
    int N,Q;
    cin>>N>>Q;

    buildsegtree(1,1,N);

    while (Q--)
    {
        int x,y;
        char c;
        cin>>c>>x>>y;
        if (c=='C')
        {
            int z;
            cin>>z;
            updateSegtree(1, x, y, z);
```

```
        }
        else if (c=='Q')
        {
            int64_t result=searchSumSegtree(1, x, y);
            cout<<result<<'\n';
        }
    }
    return 0;
}
```

运行结果:

```
10 5
1 2 3 4 5 6 7 8 9 10
Q 4 4
4
Q 1 10
55
Q 2 4
9
C 3 6 3
Q 2 4
15
Program ended with exit code: 0
```

图 2: A Simple Problem with Integers 运行结果示意

经验与体会

这虽然是线段树中最普通的题目, 但是其所蕴含的" 延迟标记" 思想却是十分重要的线段树处理方法, 也许是线段树最重要的特点.

3.The K-th numbers

题目描述:

给定一个数列 $a_1, a_2, ..., a_n$ 和 m 个三元组表示的查询。对于每个查询 (i, j, k), 输出 $a_i, a_{i+1}, ..., a_j$ 的升序排列中第 k 个数。

题目分析:

若采用暴力解法来解这道题, 则需从原数组中抽出 $[i, j]$ 区间, 进行排序, 排序完之后, 再搜索其中第 k 大的数. 这样排序的操作时间复杂度可以达到 $O(n^2)$ 级别, 若采用一些归并排序或快速排序的算法, 时间复杂度可以达到 $O(\log n)$, 但搜索时间复杂度仍有 $O(n)$. 并且若进行多次查找, 则上一次已经进行过的排序结果不能为本次查找操作所用, 也就是说每一次查找都要经历一次排序和搜索过程, 这样做必定会损失很长时间.

我们采用线段树来解决这个问题. 我们仍利用数组来构造这个线段树. 比较特殊的是, 每一个线段树的节点保存的是其所对应的区间 $[i, j]$ 的线段, 因此可以说这个线段树实际上是一个二维数组:

```
#define maxn 100000
vector<int> segtree[4*maxn]; //线段树的数据,实际上是一个2维数组, 横坐标有4*maxn行, 每一行为一个动态数组
```

例如节点 1, 保存了区间 $[1,n]$ 信息, 因此内部存储的是线段:$a_1,...,a_n$ 的所有点. 且所有点已经按照升序进行排列.8 个点的线段树示意图如下:

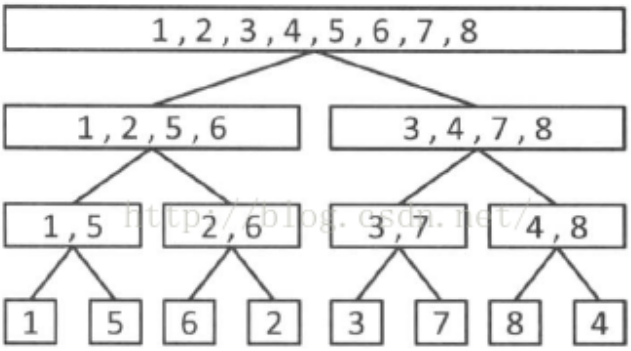

图 3: 线段树例子示意图

可以看见每一个节点都是有序排列过的, 而每一个父节点均为两个子节点序列合在一起再经过排列的序列. 这么做的好处在于, 建树的时候就进行相应的排序, 这样整个程序就只需要进行这一次排序就够了, 之后的查询操作都针对于线段树来进行. 而建立线段树过程的复杂度是 O(logn), 就可以减少时间.

查找某一序列中第 k 大的数, 由于给定的是某个基数 k, 因此免不了要进行计数, 即一个一个数直到第 k 个数. 但我们可以用另外一种想法: 取一个参考值 x, 求区间内小于 x 的个数, 记为 num, 若 $num < k$, 说明第 k 个数要大于 x; 反之若 $num > k$, 则有第 k 个数小于 x, 这样持续操作就可以逼近第 k 个数. 这样就需要两个函数: 一个用于判断区间内小于某个值 x 的个数, 另外一个是求区间内第 k 个数; 第二个函数要用到前一个函数. 由于对应同一深度的节点最多只访问常数个, 因此可以在 $O(\log 2n)$ x $O(n\log n + m\log 3n)$.

数据结构与算法:

数据结构:

```
#define maxn 100000
vector<int> segtree[4*maxn]; //线段树的数据,实际上是一个2维数组, 横坐标有4*maxn行, 每一行为一个动态数组
int integers[maxn];       //存放数据的原数组
int n,q   //数据个数与搜索次数
```

建树函数: 同前一题, 但要注意, 每一个节点需进行对两个子节点的排序, 因此函数的最后需加上一句排序语句:

```
void build(int k, int l, int r) //构建线段树.k是节点的编号, 和区间[l, r)对应
{
    if (r==l)
        segtree[k].push_back(integers[l]); //如果是叶子节点, 就将数组数据输入叶节点, 加到第k行末尾 (每一行都是一个动态数组)
    else
    {
        build(2*k, l, (l + r) / 2); //建左子树
        build(2*k+1, (l + r) / 2+1, r); //建右子树

        segtree[k].resize(r - l+1); //k节点的大小需要改变, 因为每一个k节点实际上都是一个动态数组, 存的是(l,r)区间已排序的线段
        merge(segtree[2*k].begin(), segtree[2*k].end(), segtree[2*k+1].begin(), segtree[2*k+1].end(),segtree[k].begin());
```

```
    } //利用STL的merge函数把两个儿子的数列合并，父节点为两个子节点所含线段的归并排序线段
}
```

判断区间内小于某个值 x 的个数: 同样地可以用递归算法进行简化: 若节点区间与给定区间完全没有重合, 则返回, 若节点区间完全被给定区间包含, 则返回节点区间内小于 x 的个数, 否则的话, 此时节点区间与给定区间只可能是不完全重合关系, 给定区间内小于 x 的个数等于两个子区间各自小于 x 的个数之和, 这样就可以利用递归算法了:

```
int query(int i, int j, int x, int k, int l, int r) //计算[i, j]中不超过x的数的个数。k是节点的编号，和区间[l, r)对应
{
    if (j < l || i>r) //两个区间完全不相交
        return 0;
    else if (i <= l&&j>= r) //若(l,r)完全包含在(i,j)里面
        return (upper_bound(segtree[k].begin(), segtree[k].end(), x) - segtree[k].begin()); //返回在k节点保存的有序线段中，大于x的第一个元素的下标位置

    else //若区间不完全重合
    {
        int lcn = query(i, j, x, k*2, l, (l + r) / 2);
        int rcn = query(i, j, x, k*2+1, (l + r) / 2+1, r);
        return lcn+rcn; //对左右子树进行递归计算，父节点线段中不超过x的数的个数等于左右子树不超过x的数的个数之和
    }
}
```

求给定区间第 k 个数的函数: 按照上面的思路, 我们应设立一个参考值 reference, 利用上一个函数, 搜索在区间内小于 reference 的个数 num, 将它和 k 对比, 再去修改 reference 值, 这样就可以用 reference 逼近给定区间内的第 k 个数. 但这样头疼的地方就在于循环的出口在哪里. 我们的做法是: 设立一个初始很大的区间 $[l, r]$, 用其中位数 mid 作为参考值 reference, 根据比较结果来决定是取它的左半区间还是右半区间, 最后直到 $l == r == mid$ 的时候, 循环结束, 此时肯定有 mid 等于给定区间 $[x, y]$ 中的第 k 个数:

```
int search(int x, int y, int k) //二分法搜寻区间[x,y]排序后的第k个数
{
    int l = -inf; //考虑到数据中负数的存在,因此 mid 初始设立为0
    int r = inf; //设定一个区间[l,r],使参考值 mid 可以改动
    while (l < r) //逐渐逼近 k 值,当 l==r 时循环退出
    {
        int mid = (l + r)/2; //参考值mid,在区间[x,y]内小于 mid 的值由 num 记录,将 num 与 k 比较
        int num = query(x, y, mid, 1, 1, n); //在整颗树的范围内搜索,num 记录在区间[x,y]内小于 mid 的个数
        if (k <= num) //与 num 相比,k若小于则在左半区间中搜索,若大于则在右半区间中搜索
            r = mid;
        else
            l = mid + 1;
    }
    return l;
}
```

主函数:

```
int main()
{
```

```
    scanf("%d%d", &n, &q);
    for (int i = 1; i <= n; i++)
        scanf("%d", integers + i);

    build(1, 1, n);
    int li, ri, ki;
    for (int i = 0; i < q; i++)
    {
        scanf("%d%d%d", &li, &ri, &ki);
        printf("%d\n", search(li, ri, ki));
    }

    return 0;
}
```

运行结果:

```
7 3
1 5 2 6 3 7 4
2 5 3
5
4 4 1
6
1 7 3
3
Program ended with exit code: 0
```

图 4: K-th number 运行结果示意

经验与体会

这道题里数据结构的选择很巧妙, 每一个节点都存放着一个动态数组, 能够保存相应区间内所有的点, 之后直接在这个节点内将这一段区间进行排序, 保证每一个节点都是升序排列的顺序, 这样就避免了每查找一次都需要排一次序的麻烦. 同时利用一个动态区间去逼近第 k 个数的做法也很巧妙, 在 $O(\log n)$ 的查找时间内就可以找到.

4.Lost cows

题目描述:

有 N 头奶牛，编号为 $1 \sim N$，乱序排成一列，现在已知每头牛前面有多少头牛比它的编号小，求队伍从前往后数每头牛的编号.

题目分析:

首先, 题目中有一个隐藏条件: 第一头牛前面肯定只有 0 头牛比它的编号小. 其次我们从后往前考虑, 对于最后一头牛来说，设它前面 $N-1$ 头牛编号比他大的牛的数目为 i_N, 说明整个队伍里头, 只有 i_N 头牛的编号要比它小, 也就是说:i_N+1 就是它本身的编号. 然后我们把它的编号移除, 这样, 对于倒数第二只牛, 它就在剩下的序列中排列第 $i_{N-1}+1$. 因此我们可以一直这样做来得到剩余牛的编号. 我们用线段树来纪录每个区间的长度 len，每从后向前查找一头牛的编号，我们将线段树保存的区间的 len 值 -1, 看

线段树内哪个区间内未被删除的数字个数能使当前要找的数成为第 $i+1$ 个, 能则递归左子树, 否则递归右子树, 那么最后的叶子节点的值就是这头牛的初始编号.

数据结构与算法:

普通的线段树结构体结构: 区间的左右端, 与其区间长度 len:

```
typedef struct node
{
    int l,r;
    int len;
}segment;
```

建树: 利用数组, 建一个普通的线段树, 区间从 $[1, N]$, 根节点下标设为 1:

```
#define maxsize 10000
segment tree[maxsize*4];   //线段树

void build(int i,int left, int right)
{
    tree[i].l=left;
    tree[i].r=right;
    tree[i].len=right-left+1; //区间长度
    if (left==right)
        return;
    else
    {
        int mid=(left+right)/2;
        build(2*i, left, mid);
        build(2*i+1, mid+1, right);
    }
}
```

查找函数: 查找 value 在根节点为 i 的线段树中的位置

```
int query(int i, int value) //查找 value 在根节点为 i 的线段树中的位置
{
    tree[i].len-=1; // 叶节点处: len=0, 每找到一个叶节点就被删除
    if (tree[i].l==tree[i].r)
        return tree[i].l;
    else if (value<=tree[2*i].len) //如果 value 小于等于左子树的区间长度,那么 value 肯定落在左子树中
        return query(2*i, value);
    else //如果 value 大于左子树的区间长度,那么 value 还有一部分在右子树当中
        return query(2*i+1, value-tree[2*i].len); // 在右子树当中的长度为value-tree[2*i].len
}
```

按照之前的思路: 最后查找到线段树的某一叶节点为止, 而叶节点的长度为 1, 找到之后区间长度变为零, 可视为被删除. 而若 value 小于等于左子树的区间长度, 那么 value 肯定落在左子树中, 如果 value 大于左子树的区间长度, 那么 value 还有一部分在右子树当中, 在右子树当中的长度为 value-tree[2*i].len, 分别对左右子树进行递归操作可得到最后结果.

主函数:

```cpp
int main()
{
    int pre[maxsize]; //乱序排列,每头牛之前编号小于自己的数目
    int ans[maxsize]; //乱序牛的编号
    int n;

    scanf("%d",&n);
    build(1,1,n); //建树,根节点下标为1,区间为[1,n]

    pre[1]=0; //隐藏条件:第一头牛前面有0个编号小于它

    for(int i=2;i<=n;i++)
        scanf("%d",&pre[i]);

    for(int i=n;i>0;i--)
        ans[i]=query(1,pre[i]+1); //从后往前查找,注意查找的 value 值为 pre[i]+1
    for(int i=1;i<=n;i++)
        printf("%d\n",ans[i]);

    return 0;
}
```

运行结果:

```
5
1
2
1
0
2
4
5
3
1
Program ended with exit code: 0
```

图 5: Lost cows 运行结果示意

经验与体会

本题的突破口在于, 从后往前数牛的时候, 它的编号等于前边编号小于它本身的牛的数目 +1, 之后删掉这个编号, 重新对倒数第二个牛重复这个操作就可以从后向前依次找到队伍的编号.

实现这种操作的函数构思也很巧妙, 对线段树仍采用递归操作, 只不过每一次调用此函数的时候都要将区间减一, 从树根递归运算到树叶节点时, 相当于将每个节点的长度都减去了一个. 若循环到叶节点时, 区间长度变为零, 直接就相当于将此节点删去了.