



线段树（朴素）

问题

区间信息的维护与查询

写有哪些运算可以使用线段树来维护？

线段树是一种非常强大的数据结构，它的核心功能是高效地处理区间查询和区间更新操作。线段树能够维护的运算必须满足一个关键性质：结合律。

核心要求：运算满足结合律

对于一个二元运算 \oplus ，它必须满足结合律： $(a \oplus b) \oplus c = a \oplus (b \oplus c)$

只有满足结合律的运算，我们才能将一个大区间 $[L, R]$ 的信息，通过将其分割成两个（或多个）子区间 $[L, M]$ 和 $[M + 1, R]$ ，然后合并这两个子区间的結果 $info_{left} \oplus info_{right}$ 来高效地计算得到。线段树节点存储的就是其对应区间的运算结果。

常见的可以被线段树维护的运算 / 信息

1. 区间和 (+):

- 查询：`query(l, r) = a[1] + a[1+1] + ... + a[r]`
- 更新：点更新（修改单个元素值），区间更新（给区间内每个元素加 / 减一个值 - 需要懒惰标记）
- 满足结合律： $(a + b) + c = a + (b + c)$

2. 区间最小值 (min):

- 查询：`query(l, r) = min(a[1], a[1+1], ..., a[r])`
- 更新：点更新（修改单个元素值），区间更新（将区间内每个元素设为某个值 - 需要懒惰标记）。注意：区间加 / 减一个值也能维护最小值（因为最小值是线性变化的）。
- 满足结合律： $\min(\min(a, b), c) = \min(a, \min(b, c)) = \min(a, b, c)$

3. 区间最大值 (max):

- 查询：`query(l, r) = max(a[1], a[1+1], ..., a[r])`
- 更新：类似最小值（点更新，区间设值，区间加 / 减）。
- 满足结合律： $\max(\max(a, b), c) = \max(a, \max(b, c)) = \max(a, b, c)$

4. 区间乘积 (\times):

- 查询：`query(l, r) = a[1] * a[1+1] * ... * a[r]`
- 更新：点更新，区间乘一个值（需要懒惰标记）。注意零值需要特殊处理（如果元素可能为 0）。
- 满足结合律： $(a \times b) \times c = a \times (b \times c)$

5. 区间最大公约数 (gcd):

- 查询：`query(l, r) = gcd(a[1], a[1+1], ..., a[r])`
- 更新：点更新（修改单个元素值）。区间更新（如区间加）维护 GCD 比较复杂，通常结合差分或其他技巧。
- 满足结合律： $\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c)) = \gcd(a, b, c)$

6. 区间异或 (`xor` 或 `^`):

- 查询：`query(l, r) = a[1] ^ a[1+1] ^ ... ^ a[r]`
- 更新：点更新，区间异或一个值（需要懒惰标记）。
- 满足结合律： $(a ^ b) ^ c = a ^ (b ^ c)$

7. 区间按位与 (&):

- 查询: `query(l, r) = a[1] & a[1+1] & ... & a[r]`
- 更新: 点更新, 区间与一个值 (需要懒惰标记, 但逻辑可能复杂)。
- 满足结合律: `(a & b) & c = a & (b & c)`

8. 区间按位或 (|):

- 查询: `query(l, r) = a[1] | a[1+1] | ... | a[r]`
- 更新: 点更新, 区间或一个值 (需要懒惰标记, 逻辑可能复杂)。
- 满足结合律: `(a | b) | c = a | (b | c)`

9. 区间最值差 (`max - min`):

- 查询: `query(l, r) = (max in [l, r]) - (min in [l, r])`
- 更新: 点更新 (修改单个元素值), 区间加 / 减 (因为最值差在区间加 / 减下不变, 只需更新 max 和 min)。
- 需要同时维护区间最大值和最小值信息, 查询时计算差值。结合律体现在如何合并子区间的 max 和 min 上。

10. 区间第 K 小 / 大数:

- 通常使用权值线段树或 主席树 (可持久化线段树) 来实现。这需要离散化数据, 并在每个节点存储其值域范围内元素出现的总次数。结合律体现在子区间计数值的加和上。

11. 矩阵运算 (广义结合律):

- 如果区间中的每个元素是一个矩阵, 并且你关心的运算是矩阵乘法 (或其他满足结合律的矩阵运算), 那么线段树也可以维护区间矩阵乘积。这在处理动态线性变换问题时非常有用。
- 满足结合律: 矩阵乘法满足结合律 `(A * B) * C = A * (B * C)`。

12. 自定义满足结合律的运算:

- 只要你能定义一个二元运算 `⊕` 并证明它满足结合律, 你就可以用线段树来维护这个运算在任意区间上的计算结果。例如:
 - 字符串连接 (如果连接是需要的操作, 但注意字符串长度增长很快, 效率可能不高)。
 - 模意义上的乘法或加法。
 - 结构体 (如记录最大连续子段和、区间总和、前缀最大和、后缀最大和等信息, 通过定义 合适的合并规则 来维护区间最大连续子段和)。

思路

非常直观, 不做赘述

每个节点表示原数组的一个区间。

懒标记: 在修改区间时, 不直接更新叶子节点, 而是在区间上进行暂存, 查询时再把懒标记下放给叶子结点, 避免大规模的数据存储与修改。

详见 [OI Wiki](#)

code

线段树的运用很多, 为了增加线段树的可扩展性, 使用 OPP 的思想来处理不同的模块。代码分段给出……

区间

我们知道线段树每一个节点都表示一个区间。

使用了重载运算符完成四个操作:

1. 读入

2. 合并
3. 包含
4. 分割

```

struct Rge{
    int l,r,mid,len;
    Rge():l(0),r(0),mid(0),len(0){};
    Rge(int l_,int r_):l(l_),r(r_),mid((l_+r_->>1),len(r_-l_-+1){};
    friend istream& operator >> (istream& stream,Rge& rg){
        int l_,r_;
        stream>>l_->>r_;
        rg = Rge(l_,r_);//使用初始化函数，初始化所有可能用到的变量
        return stream;
    }
    friend Rge operator + (Rge a, Rge b) {
        return Rge(a.l, b.r);
    }
    friend bool operator <= (Rge a,Rge b){//判断区间的包含关系
        return b.l <= a.l && a.r <= b.r;
    }
    friend pair<Rge,Rge> operator / (Rge x,const int p){//分成两个子区间
        return make_pair(Rge(x.l,x.mid),Rge(x.mid+1,x.r));
    }
};
```

信息

节点存储的内容

1. 管理的区间
2. 管理的信息对象（参见后文）
3. 管理的方式方法
4. 管理信息的初始化

```

struct Msg {
    Rge rg;
    struct Sum {
        int res, tag;
        friend Sum operator + (Sum a, Sum b) {
            return Sum{a.res + b.res, 0}; //区间合并时tag一定为0
        }
    } sum;//以区间和为例
    friend Msg operator + (Msg a, Msg b) {
        return Msg{a.rg + b.rg, a.sum + b.sum};
    }
    void update(int val){
        sum.res += val * rg.len;
        sum.tag += val;
    }
    void init(int val){
        sum.res = val;
    }
```

};

节点

记录左儿子和右儿子的编号。

```
struct Node {
    int ls, rs;
    Msg msg;
} node[maxn];
```

关于节点的创建与储存：

不使用 传统的 4 倍空间存储法，在空间上能省则省。

不使用 容易坏且内存开销巨大的 *vector* 存储。

朴素而简单的 堆+数组，把空间复杂度降到 $O(n)$ 。

线段树基本操作

分为两类：

1. 修改类 (*modify - mdf*)
2. 查询类 (*query - qry*)

这两类代码有着类似的运行逻辑。

```
function f(int k,/*other arguments*/){
    if(node[k].msg.rg <= inrg){//在本区间内
        /*Do sth.*/
        return;
    }
    pushdown(k); //下传标记 & 新建节点
    if(inrg.l <= node[k].msg.rg.mid){//处理左侧区间
        /*Do sth.*/
        f(node[k].ls); //递归
    }
    if(inrg.r > node[k].msg.rg.mid){//处理右侧区间
        /*Do sth.*/
        f(node[k].rs); //递归
    }
    return;
}
```

MODIFY

```
void mdf(int k,int val){
    if(node[k].msg.rg <= inrg){
```

```

    node[k].msg.update(val);
    return;
}
pushdown(k);
if(inrg.l <= node[k].msg.rg.mid)mdf(node[k].ls,val);
if(inrg.r > node[k].msg.rg.mid)mdf(node[k].rs,val);
node[k].msg = node[node[k].ls].msg + node[node[k].rs].msg;
}

```

QUARY

```

int qry(int k){
    if(node[k].msg.rg <= inrg){
        return node[k].msg.sum.res;
    }
    pushdown(k);
    int ret = 0;
    if(inrg.l <= node[k].msg.rg.mid)ret += qry(node[k].ls);
    if(inrg.r > node[k].msg.rg.mid)ret += qry(node[k].rs);
    return ret;
}

```

例题

P3373 线段树1

```

#include<bits/stdc++.h>
#define int long long
using namespace std;
constexpr int maxn = 1e6+7;
int n,m,opt,val,root=1;
namespace SGT {
    int cnt = 1,a[maxn];
    struct Rge {
        int l, r, mid, len;
        Rge(): l(0), r(0), mid(0), len(0) {};
        Rge(int l_, int r_): l(l_), r(r_), mid((l_ + r_) >> 1), len(r_ - l_ + 1) {};
        friend istream& operator >> (istream& stream, Rge& rg) {
            int l_, r_;
            stream >> l_ >> r_;
            rg = Rge(l_, r_);
            return stream;
        }
        friend Rge operator + (Rge a, Rge b) {
            return Rge(a.l, b.r);
        }
        friend bool operator <= (Rge a, Rge b) {
            return b.l <= a.l && a.r <= b.r;
        }
        friend pair<Rge, Rge> operator / (Rge x, const int p) {
            return make_pair(Rge(x.l, x.mid), Rge(x.mid + 1, x.r));
        }
    };
}

```

```

}inrg;
struct Msg {
    Rge rg;
    struct Sum {
        int res, tag;
        friend Sum operator + (Sum a, Sum b) {
            return Sum{a.res + b.res, 0}; //区间合并时tag一定为0
        }
    } sum;
    friend Msg operator + (Msg a, Msg b) {
        return Msg{a.rg + b.rg, a.sum + b.sum};
    }
    void update(int val){
        sum.res += val * rg.len;
        sum.tag += val;
    }
};

struct Node {
    int ls, rs;
    Msg msg;
} node[maxn];
void pushdown(int k){
    if(node[k].msg.sum.tag){
        node[node[k].ls].msg.update(node[k].msg.sum.tag);
        node[node[k].rs].msg.update(node[k].msg.sum.tag);
        node[k].msg.sum.tag = 0;
    }
}
void mdf(int k,int val){
    if(node[k].msg.rg <= inrg){
        node[k].msg.update(val);
        return;
    }
    pushdown(k);
    if(inrg.l <= node[k].msg.rg.mid)mdf(node[k].ls,val);
    if(inrg.r > node[k].msg.rg.mid)mdf(node[k].rs,val);
    node[k].msg = node[node[k].ls].msg + node[node[k].rs].msg;
}
int qry(int k){
    if(node[k].msg.rg <= inrg){
        return node[k].msg.sum.res;
    }
    pushdown(k);
    int ret = 0;
    if(inrg.l <= node[k].msg.rg.mid)ret += qry(node[k].ls);
    if(inrg.r > node[k].msg.rg.mid)ret += qry(node[k].rs);
    return ret;
}
void build(int k,Rge rg){
    node[k].msg.rg = rg;
    if(rg.len == 1){
        node[k].msg.sum.res = a[rg.l];
        return;
    }
    auto div = rg/2;
}

```

```

node[k].ls = ++cnt;
build(node[k].ls,div.first);
node[k].rs = ++cnt;
build(node[k].rs,div.second);
node[k].msg = node[node[k].ls].msg + node[node[k].rs].msg;
}
}

signed main() {
ios::sync_with_stdio(0),cin.tie(0),cout.tie(0);
cin>>n>>m;
for(int i=1;i<=n;i++)cin>>SGT::a[i];
SGT::build(root,SGT::Rge(1,n));
while(m--){
cin>>opt>>SGT::inrg;
if(opt==1){
cin>>val;
SGT::mdf(root,val);
}else cout<<SGT::qry(root)<<'\n';
}
return 0;
}

```

线段树（动态开点）

写在前面

首先要了解动态开点线段树的使用情景

情况	处理方法	说明
初始全为0	最简单	不存在的节点直接返回0，逻辑自然。
初始全为同一个值 <code>val</code>	修改 <code>pushdown</code> 逻辑	在创建新节点时，将其值初始化为 <code>val * rg.len</code> ，而不是0。
初始为一个（稀疏）数组	视情况通过 <code>update</code> 初始化	只为那些非默认值的元素创建节点。如果数组密集，则不适合用动态开点。
有值节点极多	不建议使用	退化成普通线段树

因为上述的板子很方便了，只需要稍作修改即可

新增

```

int create(Rge rg){
++cnt;
node[cnt].msg.rg = rg;

```

```
    return cnt;
}
```

修改

1. 新增检查和新建节点的模块

```
void pushdown(int k){
    auto div = node[k].msg.rg/2;
    if(!node[k].ls)node[k].ls = create(div.first);
    if(!node[k].rs)node[k].rs = create(div.second);
    if(node[k].msg.sum.tag){
        node[node[k].ls].msg.update(node[k].msg.sum.tag);
        node[node[k].rs].msg.update(node[k].msg.sum.tag);
        node[k].msg.sum.tag = 0;
    }
}
```

2. 删去无用的 `build` 函数。

Problem

1. Code

使用数组写法。

2. Code

使用指针优化写法。在保留原有代码的极高扩展性的情况下使代码时空效率更高。

应用



踩坑记录

- 记得判断输入区间是否合法，若没有声明 `l < r`，则默认进行交换操作。
- 不要滥用动态开点，新增节点会带来更多的时间复杂度。