哈希

```
//字符串哈希模板

##### typedef unsigned long long ull;
ull h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64
const int P=13331;
初始化
p[0]=1;
for(int i=1;i<=n;i++)
{
    h[i]=h[i-1]*P+str[i];
    p[i]=p[i-1]*P;
}

// 计算子串str[l~r]的哈希值,l和r范围为1~n***
ull get(int l, int r)//画图理解  进制运算 得到子串哈希值
{
    return h[r]-h[l-1]*p[r-l+1];
}
```

```
#include<iostream>
using namespace std;
const int MAX=100005;            //数组的最大长度（即图中点个数的最大值）
int m,n;                         //当前图的长宽规格
int pre[MAX];                    //用于存放每个点的根节点
void init(int n) {           //初始化函数
    for(int i=1; i<=n; i++)
        pre[i]=i;
}
int find(int x) {//递归
    if (x != pre[x]) pre[x] = find_r(pre[x]);//
    return pre[x];
}
//循环
int _find(int x) {
    while(x != pre[x]) { //如果x元素的父亲指向的不是自己，说明x并不是集合中的根元素，还需要一直向上查找和路径压缩
        //在find查询中嵌入一个路径压缩操作
        pre[x]=pre[pre[x]];//区别
        //x元素不再选择原来的父亲节点，而是直接选择父亲节点的父亲节点来做为自己新的一个父亲节点
        //这样的操作使得树的层数被压缩了
        x=pre[x];//x压缩完毕后且x并不是根节点，x变成x新的父节点继续进行查找和压缩的同时操作
    }
    return x;//经过while循环后，x=pre[x],一定是一个根节点，且不能够再进行压缩了,我们返回即可
}
void merge(int x,int y) {       //合并函数
    int rx=find_r(x);
    int ry=find_r(y);
    if(rx!=ry) pre[rx]=ry;
}
```

字典树

```
#include <bits/stdc++.h>
#define LOCAL
using namespace std;
int t;
//统计不同单词数
//flag =1 结点数
const int MAX =2e6+5;//如果是64MB可以开到2e6+5，尽量开大
int tree[MAX][30];//tree[i][j]表示节点i的第j个儿子的节点编号
int flag[MAX];//表示以该节点结尾是一个单词
int sum[MAX];
int tot=0;//总节点数
int ans=0;
void insert_(string s)
{
   int  len=s.size();
   int root=0;
   for(int i=0;i<len;i++)
   {
       int id=s[i]-'0';
       if(!tree[root][id]) tree[root][id]=++tot;//新子树 编号tot+1
       sum[tree[root][id]]++;//
       root=tree[root][id];
   }
   if(!flag[root]){flag[root]=1;ans++;}//未出现
}
int  find_(string s)//查询操作，按具体要求改动
{
    int len=s.size();
    int root=0;
    for(int i=0;i<len;i++)
    {
        int id=s[i]-'0';
        if(!tree[root][id]) return 0;
        root=tree[root][id];
    }
    return sum[root];////返回当前字符串结尾节点的访问次数，也就是作为前缀的出现次数
}
void init()//最后清空，节省时间
{
    ans=0;
    for(int i=0;i<=tot+5;i++)//?
    {
       flag[i]=false;
       sum[i]=0;
       for(int j=0;j<35;j++)
           tree[i][j]=0;

    }
   tot=0;
}

int main(){
std::ios::sync_with_stdio(false);
cin.tie(0);
```

```
cout.tie(0);

string s;
while(getline(cin,s)){
    if(s=="#"){
    break;
    }
    string s1="";
    for(int i=0;i<s.size();i++){//注意输入
        if(i>0&&s[i]==' '&&s[i-1]!=' '){insert_(s1);s1="";}//
        else if(s[i]!=' '){
            s1+=s[i];
            if(i==s.size()-1&&s1!=" ")insert_(s1);//最后也要
             }
    }
    cout<<ans<<endl;
    init();
}
    return 0;
}
```

单调队列

滑动窗口

```
#include <bits/stdc++.h>
//# pragma GCC optimize(3)
#define int long long
#define endl "\n"
using namespace std;

const int N = 2e6+ 5;
int T, n, k,a[N];
int q[N]; //单调队列  存的是下标  也可以多开一个数组存下标
void solve(){
    cin>>n>>k;
    //最小值;单调递增
    for(int i=1;i<=n;i++)cin>>a[i];
    int head=1,tail=0;  //   -1和0; 0,0也可
    for(int i=1;i<=n;i++){// 每次移动一个元素入队
        if(head<=tail&&i-k+1>q[head])head++; //队首已不在窗口内
        while(head<=tail&&a[q[tail]]>=a[i])tail--; //pop 掉队尾大于a[i]的 (在前面且
小于)
        q[++tail]=i ; //新元素入队
        if(i>=k)cout<<a[q[head]]<<" "; //输出队首元素max
    }
    cout<<endl;
    //最大值同理
    head=1,tail=0;
    for(int i=1;i<=n;i++){// 每次一个元素入队
        if(head<=tail&&i-k+1>q[head])head++; //队首已不在窗口内
        while(head<=tail&&a[q[tail]]<=a[i])tail--; //pop 掉队尾小于a[i]的 (在前面且
小于)
        q[++tail]=i ; //新元素入队
        if(i>=k) //窗口已进入
            cout<<a[q[head]]<<" "; // 输出队首元素max
```

```
    }
}
signed main() {
  std::ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
      solve();
  return 0;
}
```

树状数组

1. 单点修改区间查

```
#include <bits/stdc++.h>
#define LOCAL
using namespace std;//模板题 单点修改 + 区间查询
//动态维护树状数组 c[i]代表a[i]及之前共lowbit[i]个元素
int c[50005],n,t;
int lowbit(int x){
    return x& -x;
}
void update(int i,int val){//单点更新:每次加自身lowbit的元素改变
    while(i<=n){
        c[i]+=val;
        i+=lowbit(i);
    }
}//更新(从小到大)时查询(从大到小)的逆过程
int sum(int i){//求前缀和/
    int ret =0;
    while(i>0){
        ret+=c[i];
        i-=lowbit(i);
    }
    return ret;
}
int main(){
cin>>t;
int Case=0;
while(t--){
    memset(c,0,sizeof c);
    Case++;
printf("Case %d:\n",Case);
scanf("%d",&n);
for(int i=1;i<=n;i++){
    int val;
    scanf("%d",&val);
    update(i,val);
}
string s;
while(cin>>s){
    if(s=="End")break;
    int a,b;
    scanf("%d %d",&a,&b);
    int ans=0;
```

```
    if(s=="Query"){
        ans=sum(b)-sum(a-1);
       printf("%d\n",ans);
    }
    else if(s=="Sub"){
        update(a,-b);
    }
    else if(s=="Add"){
        update(a,b);
    }
}
}
    return 0;
}
```

2.区间修改，单点查

差分

```
#include <bits/stdc++.h>
#define LOCAL//由于本题是先修改最后按顺序查询 所以直接用普通数组差分实现区间修改之后按序维护输出前缀和更快(n>nlogn)  ,但如果边改变查或随机查询效率显然不如本方法
using namespace std;//nlog
//树状数组实现 单点查询 区间修改
//   用差分树状数组 d[]实现
//用差分数组将区间修改转化为单点修改    单点查询转化为求前缀和
//区间修改只需改端点的差分值
//  单点查询只需求d[i]的前缀和
//  d[0]=0,d[1]=a[1];d[i]==a[i]-a[i-1],
int n,d[100005];
int lowbit(int i){
    return i& -i;
}
void update(int i,int val){//树状数组单点修改，每次对差值更新
    while(i<=n)d[i]+=val,i+=lowbit(i);
}
int sum(int i){//求树状数组前缀和***//求sum d[i]实现单点查询
    int ret =0;
    while(i>0){
    ret+=d[i];
    i-=lowbit(i);
    }
    return ret;
}
void  range_update(int l,int r,int x){//差分 实现区间修改
    update(l,x);update(r+1,-x);//
}

int main(){
#ifdef LOCAL
    freopen("data.in","r",stdin);
    freopen("data.out","w",stdout);
#endif
while(cin>>n,n){
    memset(d,0,sizeof d);
    for(int i=1;i<=n;i++){
        int a,b;
```

```
        cin>>a>>b;
        range_update(a,b,1);//区间修改
    }
    for(int i=1;i<=n;i++){
        cout<<sum(i);
        if(i!=n)cout<<" ";
    }
    cout<<endl;
}
    return 0;
}
```

线段树

```
#include <bits/stdc++.h>
#define LOCAL
using namespace std;
// l,r 是大（总）区间 ， L，R是操作/查询区间
const int N =1e5+7;//元素总个数
int a[N];//原数组 可不用
struct segtreenode{
  int val;
  int lazy ;//懒惰标记
       //其他元素
}segtree[N<<2];//定义原数组大小四倍的线段数组
void pushup(int rt){
  segtree[rt].val=segtree[rt<<1].val+segtree[rt<<1|1].val;//线段树写法
}//将左右字数的总值加到根节点

void build(int l,int r,int rt){// 递归构造线段树
  segtree[rt].lazy=0;//      初始化
  if(l==r){//出口  左右相等 为叶子节点则停止向下递归
      segtree[rt].val=a[l];//叶子节点的l,r即位置下标
      return;
  }
  int mid=(l+r)/2;
  build(l,mid, rt*2);   //递归构造左子树 根序号为2*rt ,2*tr+1
  build(mid+1,r,rt*2+1);  //递归构造右子树
  pushup(rt);            //** 回溯,当左右子树都构造完后向上加到根节点
}

//单点更新，假设 a[t]+=c    类似二进制，每层只需更新一个   从上到下
void updateNode(int t,int c,int l,int r,int rt){//l,r 表示当前节点区间，rt表示当前根节点编号
  if(l==r){
    segtree[rt].val+=c;//     叶子节点   直接修改
    return;
  }
  int mid=(l+r)/2;
  if(t<=mid)  updateNode(t,c,l,mid,rt<<1);  //更新左子树
  else    updateNode(t,c,mid+1,r,rt<<1|1);
  pushup(rt);                            //回溯向上更新，相加
}

//区间查询(区间a[L....R]的和)   [L,R]为操作区间,[l,r]为当前区间，rt为节点编号
int query(int L,int R,int l,int r,int rt){
```

```
  if(L<=l&&r<=R)
    return  segtree[rt].val;//当前区间被包含 则直接返回(整个被加)
  // 并且不再向下递归
  if(L>r||R<l)    //当前区间全部不重和，则返回0，而且其子区间也不会包含
  return 0;
//否则部分包含  向下递归
  int mid=(l+r)/2;
  return  query(L,R,l,mid,rt<<1)+query(L,R,mid+1,r,rt<<1|1);
}

//ln,rn 分别为左右子区间大小
void pushdown(int rt,int ln,int rn){
  if(segtree[rt].lazy){//有懒惰标记
    segtree[rt<<1].lazy+=segtree[rt].lazy;//更新左右子区间的值和懒惰标记
    segtree[rt<<1|1].lazy+=segtree[rt].lazy;
    segtree[rt<<1].val+=segtree[rt].lazy*ln;
    segtree[rt<<1|1].val+=segtree[rt].lazy*rn;

    segtree[rt].lazy=0;//** 清除标记
  }
}
//区间更新-->延迟操作 ***(eg. a[L,R]+=C)  [L,R]为操作区间,[l,r]为当前区间，rt为节点编号
// 结果：将完全包含于[L,R]的子区间更新并存lazy，其余等查询后再下推
void updateRange(int L,int R,int l,int r,int c,int rt){
    if(L<=l&&r<=R){//只有当前区间被完全包含才更新自己及子区间  ,部分包含先不更新，减少操作次
数
        segtree[rt].val+=c*(r-l+1);      //更新区间总和
        segtree[rt].lazy+=c;      //根据不同操作更新懒惰标记
        return ;
    }
    int mid=(l+r)/2;
    //只做了lg级的pushdowm，其余用懒惰记录，查询时修改
    pushdown(rt,mid-l+1,r-mid);// 一次下推操作，才能准确 更新左右子节点 (非必要，不下退)
    if(L<=mid) updateRange(L,R,l,mid,c,rt<<1);//更新左子区间
    if(R>mid) updateRange(L,R,mid+1,r,c,rt<<1|1);   //更新右子区间
    pushup(rt);
}

//区间更新的区间查询（单点）
int queryRange(int L,int R,int l,int r,int rt){
  if(L<=l&&r<=R)
    return  segtree[rt].val;

  if(L>r||R<l)
  return 0;

  int mid=(l+r)/2;
  pushdown(rt,mid-l+1,r-mid);//唯一不同,也是精华所在 ，查询到，必要时下推
  return  queryRange(L,R,l,mid,rt<<1)+queryRange(L,R,mid+1,r,rt<<1|1);
}
```