线段树

均为数组实现

求和线段树

支持区间修改，如果不需要记得把pushdown和segupdate相关的都删了，别的不用改。

```
#include <bits/stdc++.h>
typedef long long ll;
const int maxn = 2e5+300;
const int inf = 0x3f3f3f3f;
using namespace std;
int a[maxn],Sum[maxn<<2],pos[maxn],lazy[maxn];
//更新当前节点
void pushup(int rt){
    Sum[rt] = Sum[rt<<1] + Sum[rt<<1|1];
}
//下传函数
void pushdown(int l,int r,int rt){
    //区间改值
    if(lazy[rt]){
        int m = (l+r) >> 1;
        lazy[rt<<1] = lazy[rt];
        lazy[rt<<1|1] = lazy[rt];
        Sum[rt<<1] = lazy[rt] * (m-l+1);
        Sum[rt<<1|1] = lazy[rt] * (r-m);
        lazy[rt] = 0;
    }
    //区间增减
    /*if(lazy[rt]){
        int m = (l+r) >> 1;
        lazy[rt<<1] += lazy[rt];
        lazy[rt<<1|1] += lazy[rt];
        Sum[rt<<1] += lazy[rt] * (m-l+1);
        Sum[rt<<1|1] += lazy[rt] * (r-m);
        lazy[rt] = 0;
    }*/
}
//l:当前节点的左端点 r: 当前节点的右端点 rt:当前节点的编号
void build(int l,int r,int rt){
    if(l == r){
        pos[l] = rt;
        Sum[rt] = a[l];
        return;
    }
    int m = (l+r) >> 1;
    build(l,m,rt<<1);
    build(m+1,r,rt<<1|1);
    pushup(rt);
```

```
}
//l:当前节点的左端点 r: 当前节点的右端点 rt:当前节点的编号 [L,R]查询的区间
int query(int L,int R,int l,int r,int rt){
    if(L <= l && R >= r) return Sum[rt];
    int m = (l+r) >> 1;
    int res = 0;
    pushdown(l,r,rt);
    if(L <= m) res += query(L,R,l,m,rt<<1);
    if(R > m) res += query(L,R,m+1,r,rt<<1|1);
    return res;
}
//l:当前节点的左端点 r: 当前节点的右端点 rt:当前节点的编号 将L的值改为V
void update(int L,int V,int l,int r,int rt){
    if(l==r){Sum[rt]=V;return;}
    int m = (l+r) >> 1;
    pushdown(l,r,rt);
    if(L <= m) update(L,V,l,m,rt<<1);
    else update(L,V,m+1,r,rt<<1|1);
    pushup(rt);
}
void segupdate(int L,int R,int l,int r,int rt,int lzy){
    if(L <= l && R >= r){
        //区间改值
        lazy[rt]=lzy;
        Sum[rt] = (r-l+1) * lzy;
        //区间加减
        /*lazy[rt]+=lzy;
        sum[rt]+=(r-l+1) * lzy;*/
        return;
    }
    int m = (l+r) >> 1;
    pushdown(l,r,rt);
    if(L <= m) segupdate(L,R,l,m,rt<<1,lzy);
    if(R > m) segupdate(L,R,m+1,r,rt<<1|1,lzy);
    pushup(rt);
    return;
}
```

最大值线段树

没有加区间修改，需要的话用上面的模版改改吧。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 200000 + 5;
const int inf = 0x3f3f3f3f;
//mark是原始数据
int mark[maxn],Max[maxn<<2];
//更新当前节点
void pushup(int rt){
    Max[rt] = max(Max[rt<<1],Max[rt<<1|1]);
```

```
}
//l:当前节点的左端点 r：当前节点的右端点 rt:当前节点的编号
void build(int l,int r,int rt){
    if(l == r){Max[rt] = mark[l];return;}
    int m = (l+r) >> 1;
    build(l,m,rt<<1);
    build(m+1,r,rt<<1|1);
    pushup(rt);
}
//l:当前节点的左端点 r：当前节点的右端点 rt:当前节点的编号 [L,R]查询的区间
int query(int L,int R,int l,int r,int rt){
    if(L <= l && R >= r) return Max[rt];
    int m = (l+r) >> 1;
    int res = 0;
    if(L <= m) res = max(res,query(L,R,l,m,rt<<1));
    if(R > m) res = max(res,query(L,R,m+1,r,rt<<1|1));
    return res;
}
//l:当前节点的左端点 r：当前节点的右端点 rt:当前节点的编号 将L的值改为V
void update(int L,int V,int l,int r,int rt){
    if(l==r){Max[rt]=V;return;};
    int m = (l+r) >> 1;
    if(L <= m) update(L,V,l,m,rt<<1);
    else update(L,V,m+1,r,rt<<1|1);
    pushup(rt);
}
```

权值线段树和主席树

权值线段树

有点简单，而且估计用不到。

```
#include <bits/stdc++.h>
using namespace std;
int tree[1000];
void pushup(int rt){
    tree[rt] = tree[rt<<1] + tree[rt<<1|1];
}
//插入一个数字
void Insert(int x,int l, int r, int rt){
    if(l==r){tree[rt]++;return;}
    int m = (l+r) >> 1;
    if(x <= m) Insert(x,l,m,rt<<1);
    else Insert(x,m+1,r,rt<<1|1);
    pushup(rt);
}
//查询某个数字出现的次数
int querynum(int x,int l,int r,int rt){
    if(l==r){return tree[rt];}
    int m = (l+r) >> 1;
```

```
    if(x <= m) return querynum(x,l,m,rt<<1);
    else return querynum(x,m+1,r,rt<<1|1);
}
//查询某个区间中数字出现的次数
int queryseg(int L,int R,int l,int r,int rt){
    if(L <= l && R >= r) return tree[rt];
    int res = 0;
    int m = (l+r) >> 1;
    if(L <= m) res += queryseg(L,R,l,m,rt<<1);
    if(R >= m+1) res += queryseg(L,R,m+1,r,rt<<1|1);
    return res;
}
//查询全体第k大，注意是全体，没卵用
int kth(int k,int l,int r,int rt){
    if(l==r) return l;
    int m = (l+r) >> 1;
    if(tree[rt<<1|1] >= k) return kth(k,m+1,r,rt<<1|1);
    else return kth(k-tree[rt<<1|1],l,m,rt<<1);
}
```

主席树（占坑）

还没写。

素数筛和莫比乌斯筛

这两放一起了，莫比乌斯筛把mu相关删了就是快速筛了。

```
#include <bits/stdc++.h>
typedef long long ll;
const int maxn = 2e5 + 50;
const int mod = 1e9+7;
using namespace std;

ll mu[maxn],not_prime[maxn],prime[maxn];

void getprime(){
    mu[1] = 1;
    not_prime[1] = 1;
    ll cnt = 0;
    for(int i = 2; i <= maxn; ++i){
        if(!not_prime[i]){
            prime[cnt++] = i;
            mu[i]=-1;
        }
        for(int j = 0; j<cnt && prime[j]*i <= maxn; j++){
            ll x = prime[j]*i;
            not_prime[x] = 1;
            if(i % prime[j] == 0)
                break;
```

```
                mu[x] = -mu[i];
            }
        }
    }
```

快速幂和矩阵快速幂

快速幂

很常用的函数，也很简单。其实已经背出来了

```
ll qpow(ll x,ll y){
    ll res = 1;
    while(y){
        if(y&1) res = res * x % mod;
        y >>= 1;
        x = x * x % mod;
    }
    return res;
}
```

矩阵快速幂

把快速幂的乘法改成矩阵乘法就是矩阵快速幂了，连同主函数一起给出，通常用于快速递推数列第n项（如斐波那契数列）

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int mod=123456789;
struct matrix{
    ll a[11][11]; //begin with 1
    int r,c;
    matrix(int n,int m):r(n),c(m){memset(a,0,sizeof(a));}
    ll* operator[](int x){return a[x];}
    friend matrix operator*(matrix A,matrix B)
    {
        matrix C(A.r,B.c);
        for(int i=1;i<=A.r;i++)
            for(int j=1;j<=B.c;j++)
                for(int k=1;k<=A.c;k++){
                    C[i][j]+=(A[i][k]*B[k][j])%mod;
                    C[i][j]+=mod;
                    C[i][j]%=mod;
                }
        return C;
    }
```

```
};

matrix qpow(matrix A,ll m)//方阵A的m次幂
{
    matrix ans(A.r,A.c);
    for(int i=1;i<=A.r;i++) ans.a[i][i]=1; //单位矩阵
    while(m)
    {
        if(m&1)ans=ans*A;
        A=A*A;
        m>>=1;
    }
    return ans;
}

int main()
{
    ll T,n;
    for(cin>>T;T--;)
    {
        scanf("%lld",&n);
        matrix A(6,6);
        A[1][1]=1;
        A[1][2]=2;
        A[1][3]=1;
        A[1][4]=3;
        A[1][5]=3;
        A[1][6]=1;
        A[2][1]=1;
        A[3][3]=1;
        A[4][3]=1;
        A[4][4]=1;
        A[5][3]=1;
        A[5][4]=2;
        A[5][5]=1;
        A[6][3]=1;
        A[6][4]=3;
        A[6][5]=3;
        A[6][6]=1;
        matrix X2(6,1);
        X2[1][1]=2;
        X2[2][1]=1;
        X2[3][1]=1;
        X2[4][1]=2;
        X2[5][1]=4;
        X2[6][1]=8;
        matrix Xn=qpow(A,n-2)*X2;
        printf("%lld\n",Xn[1][1]);
    }
}
```

逆序对（归并排序）

主体是归并排序，但归并的过程中可以统计出逆序对。

```
#include <bits/stdc++.h>
typedef long long ll;
const int maxn = 1e5 + 200;
using namespace std;
//acnt为逆序对的数量
int a[maxn],b[maxn],acnt,i,j,cnt;

void Merge(int l,int m,int r){
    cnt = l,i=l,j=m+1;
    while(i  <= m && j <= r){
        if(a[i] <= a[j])
            b[cnt++] = a[i++];
        else{
            b[cnt++] = a[j++];
            //统计逆序对
            acnt += m-i+1;
        }
    }
    while(i <= m) b[cnt++] = a[i++];
    while(j <= r) b[cnt++] = a[j++];
    for(int i = l; i <= r; i++) a[i]=b[i];
}
void Mergesort(int l,int r){
    if(l < r - 1){
        Mergesort(l,(l+r)>>1);
        Mergesort(((l+r)>>1)+1,r);
    }
    Merge(l,(l+r)>>1,r);
    return;
}
```

并查集

数组实现，很简短。

```
#include <bits/stdc++.h>
const int maxn = 200;
using namespace std;
int father[maxn];
int Find(int a){
    if(father[a]==a) return a;
    return father[a]=Find(father[a]);
}
void Union(int a,int b){
    int f1=Find(a),f2=Find(b);
    father[f2] = f1;
```

```
}
void init(){
    for(int i = 1; i <= M; ++i) father[i]=i;
}
```

字符串

字典树

结构体实现。

```
#include <bits/stdc++.h>
typedef unsigned long long ull;
const int maxn = 11;
const int inf = 0x3f3f3f3f;
using namespace std;
struct trie
{
    trie *nxt[26];
    int cnt;
    trie()
    {
        cnt = 1;
        memset(nxt,NULL,sizeof(nxt));
    }
};
trie *root;//记得在函数开始前new trie
int i,id;
char S[maxn],s1[maxn];
//插入字符串
void Insert(char *s)
{
    trie *p = root;
    i = 0;
    while(s[i]){
        id = s[i] - 'a';
        if(p->nxt[id])
        {
            p = p->nxt[id];
            p -> cnt++;
        }
        else
        {
            p -> nxt[id] = new trie;
            p = p -> nxt[id];
        }
        i++;
    }
}
//查询字符串，功能可以自己改
```

```
int query(char* s)
{
    trie *p = root;
    i = 0;
    while(s[i])
    {
        id = s[i] - 'a';
        if(p -> nxt[id]) p = p -> nxt[id];
        else return 0;
        i++;
    }
    return p -> cnt;
}
//递归释放字典树
void Free(trie *p)
{
    for(i = 0; i < 26; ++i) if(p -> nxt[i] != NULL) Free(p->nxt[i]);
    delete(p);
    p = NULL;
}
```

AC自动机

多模匹配算法，特点是用目标串在AC自动机上跑一遍以后，只要字典树上某一节点代表的字符串是目标串的子串，这一节点就会被遍历到。

```
#include <bits/stdc++.h>
typedef unsigned long long ull;
const int P = 1e9+7;
const int maxn = 5e5 + 200;
const int inf = 0x3f3f3f3f;
using namespace std;
struct trie
{
    trie *nxt[26];
    trie *fail;
    int cnt;//根据题意修改
    int flag;//根据提议修改
    trie()
    {
        cnt = 1;
        flag = 0;
        fail = NULL;
        memset(nxt,NULL,sizeof(nxt));
    }
};
trie *root;
int T,N,Q;
char S[maxn],s1[maxn],s2[maxn];
//插入字符串，根据题意修改函数中的cnt、flag
void Insert(char *s)
```

```
{
    trie *p = root;
    int len = strlen(s);
    for(int i = 0; i < len; i++)
    {
        int id = s[i] - 'a';
        if(p->nxt[id] != NULL)
        {
            p = p->nxt[id];
            p -> cnt++;
        }
        else
        {
            p -> nxt[id] = new trie;
            p = p -> nxt[id];
        }
    }
    p -> flag++;
}
//获取fail指针，一般不用动
void getFail()
{
    queue<trie*> q;
    q.push(root);
    trie *temp,*p;
    while(!q.empty())
    {
        p = q.front();
        q.pop();
        for(int i = 0; i < 26; ++i)
        {
            if(p -> nxt[i])
            {
                if(p == root)
                    p -> nxt[i] -> fail = root;
                else
                {
                    temp = p -> fail;
                    while(temp)
                    {
                        if(temp -> nxt[i])
                        {
                            p -> nxt[i] -> fail = temp -> nxt[i];
                            break;
                        }
                        temp = temp -> fail;
                    }
                    if(!temp) p -> nxt[i] -> fail = root;
                }
                q.push(p -> nxt[i]);
            }
        }
    }
}
```

```
int query(char* s)
{
    int i = 0,res = 0;
    trie *p = root;
    trie *temp;
    while(s[i])
    {
        int id = s[i] - 'a';
        while(!p -> nxt[id] && p != root) p = p -> fail;
        p = p -> nxt[id];
        if(p == NULL) p = root;
        temp = p;
        //这里是匹配和计算的过程，根据题意修改
        while(temp != root && temp -> flag != 0)
        {
            res += temp -> flag;
            temp -> flag = 0;
            temp = temp -> fail;
        }
        i++;
    }
    return res;
}
void Free(trie *p)
{
    for(int i = 0; i < 26; ++i)
    {
        if(p -> nxt[i] != NULL) Free(p->nxt[i]);
    }
    delete(p);
    p = NULL;
}
```

ST表

静态区间最值，比线段树快，不支持在线查询。

```
#include <bits/stdc++.h>
using namespace std;
//d为数据，mx[i][j]表示[i,i+2^j]区间内最大值
int mx[100][100],d[100] = {0},n;
//查询[l,r]内最大值
int askmx(int l,int r) {
    int k = log2(r-l+1);
    return max(mx[l][k],mx[r-(1<<k)+1][k]);
}
//初始化，数据输入完后调用
void init(){
    for(int i = 0; i <= n; ++i) mx[i][0] = d[i];
    for(int j = 1; (1<<j) <= n+1; ++j)
```

```
        for(int i = 0; i + (1<<j) <= n+1; ++i)
            mx[i][j] = max(mx[i][j-1],mx[i+(1 << (j-1))][j-1]);
}
```

最小生成树

prim

码量小，点为主体，边多点少时效率高。

```
#include <bits/stdc++.h>
const int maxn = 200;
const int inf = 0x3f3f3f3f;
using namespace std;
int N,dis[maxn][maxn],vis[maxn],ans,mndis[maxn];
void prim(){
    //初始化，将1号点加入到生成树中
    ans = 0;
    int cnt = 1;
    vis[1] = 1;
    for(int i = 1;i <= N;++i) mndis[i]=dis[1][i];
    while(cnt != N){
        int mn = inf,id;
        //找出所有点中距离生成树最近的点
        for(int i = 1; i <= N; i++){
            if(!vis[i] && mndis[i] < mn){
                mn = mndis[i];
                id = i;
            }
        }
        //将找到的点加入生成树
        vis[id] = 1;
        ans += mn;
        cnt++;
        //更新剩余点到树的距离
        for(int i = 1; i <= N; i++) mndis[i] = min(mndis[i],dis[id][i]);

    }
}
```

Kruscal

码量较大，边为主体，边少点多效率高。

```
#include <bits/stdc++.h>
const int maxn = 200;
using namespace std;
struct edge{
```

```
    int from;
    int to;
    int val;
    edge(int a = 0,int b = 0,int c = 0){from=a;to=b;val=c;}
    friend bool operator > (edge a,edge b){
        return a.val > b.val;
    }
};
int N,M,father[maxn];
priority_queue< edge,vector<edge>,greater<edge> > Q;
int Find(int a){
    if(father[a]==a) return a;
    return father[a]=Find(father[a]);
}
void Union(int a,int b){
    int f1=Find(a),f2=Find(b);
    father[f2] = f1;
}
void init(){
    for(int i = 1; i <= M; ++i) father[i]=i;
    while(!Q.empty()) Q.pop();
}
void kruscal(){
    int ans = 0,cnt = 0;
    while(!Q.empty() && cnt != M-1){
        edge temp = Q.top();
        Q.pop();
        int f = temp.from,t = temp.to,v = temp.val;
        if(Find(t)!=Find(f)){
            Union(f,t);
            cnt++;
            ans+=v;
        }
    }
    //cnt小于M-1则没有连通，否则ans为最小生成树大小
    if(cnt != M-1) puts("?");
    else printf("%d\n",ans);
}
```

最短路

dijkstra

最常用的最短路，不支持负环。

```
#include <bits/stdc++.h>
const int maxn = 1000+50;
const int inf = 0x3f3f3f3f;
using namespace std;
struct edge{
```

```
    int to;
    int val;
    edge(int a = 0,int b = 0){to=a;val=b;}
};
struct nod{
    int pos;
    int d;
    nod(int a = 0,int b = 0){pos=a;d=b;}
    //优先队列重载大于符号
    friend bool operator > (nod a, nod b){
        return a.d>b.d;
    }
};
//邻接表
vector<edge> E[maxn];
int T,N,dis[maxn];//dis[i]为从X到i的最短距离，可以根据情况扩充为d[i][j][k].....
//加边，无向图时添加两条边
void add(int f,int t,int v){
    E[f].push_back(edge(t,v));
    E[t].push_back(edge(f,v));
    return;
}
void dij(){
    memset(dis,inf,sizeof(dis));
    priority_queue< nod,vector<nod>,greater<nod> > Q;
    Q.push(nod(N,0));//初始态,N为出发点
    while(!Q.empty()){
        nod temp = Q.top();
        int pos = temp.pos;
        int d = temp.d;
        Q.pop();
        if(d > dis[pos]) continue;
        dis[pos]=d;
        //遍历邻接表更新相邻点的最短距离
        for(int i = 0; i < E[pos].size(); ++i){
            int to = E[pos][i].to;
            int val = E[pos][i].val;
            int nd = d + val;
            if(nd < dis[to]){
                dis[to] = nd;
                Q.push(nod(to,nd));
            }
        }
    }
}
```