kruscal

熟悉unordered_map

二、图论

- 单源最短路
- 多源汇最短路

1、单源最短路

1.1 Dijkstra()算法

算法步骤

S为确定了到源点最短距离的点 算法当中用st数组标志为true，只有当那个点是更新的点里面最短点出队以后才算确定

U为没有确定最短距离的点

```
(1) 初始时，S只包含起点s；U包含除s外的其他顶点，且U中顶点的距离为"起点s到该顶点的距离"[例如，U中顶点v的距离为(s,v)的长度，然后s和v不相邻，则v的距离为0x3f3f3f3f。

(2) 从U中选出"距离最短的顶点k"，并将顶点k加入到S中；同时，从U中移除顶点k。 st置为true

(3) 更新U中各个顶点到起点s的距离。之所以更新U中顶点的距离，是由于上一步中确定了k是求出最短路径的顶点，从而可以利用k来更新其它顶点的距离；例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。普通dijkstra是更新所有边，堆优化更新相邻的边

(4) 重复步骤(2)和(3)，直到遍历完所有顶点 即n次循环，每次确定一个最短点(出队的时候)
```

算法模板

1061109567

1.1.1朴素版O（n^2)

适用于稠密图

```
int dijkstra(){
    memset(dist,0x3f,sizeof dist);
    dist[1] = 0;
    for(int i = 0;i<n;i++){ //n次松弛
        //t表示最短的点 初始并未找到这个点
        int t = -1;
        for(int j = 1;j<=n;j++)
            if(!st[j] && (t==-1||dist[t]>dist[j]))
                t = j;//t是由目前所有点得到的最短的那个点

        st[t] = true;//t点被更新为最短点

        //更新临近的点
        for(int j = 1;j<=n;j++)
```

```
            dist[j] = min(dist[j],dist[t]+g[t][j]); //1 -> t -> j
        }
        if(dist[n] == 0x3f3f3f3f) return -1;
        return dist[n];
    }
```

1.1.2 堆优化版 O(mlogn)

适用于稀疏图

- 小根堆的定义`priority_queue<int, vector<int>, greater<int>>`
- 大根堆的定义`priority_queue<int>` 默认的

```
int dijkstra(){
    memset(dist,0x3f,sizeof dist);
    dist[1]  = 0;
    priority_queue< PII,vector<PII>,greater<PII> > heap;
    heap.push({0,1});//距离 点的编号
    while(heap.size()){
        auto t = heap.top();
        heap.pop();
        int ver = t.second,num = t.first;//点的编号 点的距离
        if(st[ver]) continue;
        st[ver] = true;
        for(int i = h[ver];i != -1;i = ne[i]){
            int j = e[i];//这里的i只是邻接表当中的一个点的编号
            if(dist[j] > dist[ver] + w[i]){
                dist[j] = dist[ver] + w[i];
                heap.push({dist[j],j});
            }
        }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    else return dist[n];
}
```

1.2 bellman-ford算法

限制多少条边

注意 因为bellman_ford算法是遍历所有的边，所以可以先定义一个结构体存下所有的边然后再进行遍历

```
typedef struct{
    int a,b,w;
}Edge;
```

或者

```
struct edge{
    int a,b;
    int w;
} edges[M];//注意M不能是数字
```

bellman_ford算法核心代码

```
void bellman_ford(){
    memset(dist,0x3f,sizeof dist);
    dist[1] = 0;//从1号点开始 自己到自己的距离为0
    for(int i = 0;i<k;i++){//k条边循环k次
        memcpy(tmp,dist,sizeof dist);//防止串联更新
        for(int j = 0;j<m;j++){
            int a = edges[j].a,b = edges[j].b,w = edges[j].w;
            dist[b] = min(dist[b],tmp[a] + w);//这里的tmp就是防止本次更新串联更新了
        }
    }
}
```

1.3 spfa()算法

注意

- spfa()算法是对bellman_ford算法的优化 dist[b] = min(dist[b],tmp[a] + w);不是每一个都成立

  即只对有更新的算数
- 这里的st数组是为了防止队列中的元素重复入队

  dijkstra()的那个数组是为了确认有没有成为最短值

```
void spfa(){
    memset(dist,0x3f,sizeof dist);
    queue<int> q;
    dist[1] = 0;
    st[1] = true;
    q.push(1);
    while(q.size()){
        int t = q.front();
        q.pop();
        st[t] = false;//这里的st数组不同于dij算法，这里是为了标记它不在队列当中
        for(int i = h[t];i!=-1;i=ne[i]){
            int j = e[i];
            if(dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if(!st[j]) {
                    q.push(j);
                    st[j] = true;//已经在队列当中就不用重新放了
                }
            }
        }

    }
    if(dist[n] >= 0x3f3f3f3f / 2) puts("impossible");
    else cout<<dist[n]<<endl;

}
```

1.4 floyd（）多源汇最短路

O(n^3)

核心代码

注意

- 这里的距离用邻接矩阵存储

```
void floyd()
{
    //因为算第k层时 k-1层必须算好 所以是把k放在最外面
    //这里的d[i][j] 就是i、j点的距离
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

邻接矩阵的初始化和读入

```
//初始化
for(int i = 1;i<=n;i++)
     for(int j = 1;j<=n;j++){
        if(j==i) d[i][j] = 0;//self distance = 0;
        else d[i][j] = INF;
     }
//读入
while(m--){
       int x,y,z;
       scanf("%d%d%d",&x,&y,&z);
       d[x][y] = min(d[x][y],z);//有重边
}
```

三、DP

能不能拿一等就看dp写得怎么样了

状态表示 分情况讨论

dp的精髓在于如何用一个数代表一类物品

只有01背包问题一维优化版和有依赖的背包问题是从小到大枚举体积

其他的都是从小到大枚举体积

动态规划就是怎么拿上一步的结果推出来这一步的结果

3.1 背包问题

给一堆东西选出来一个最佳值

3.1.0 记忆化搜索

记忆化搜索 = 深搜的形式 + 动态规划的思想

即 每次搜索的时候，都将子问题的最优解比如`dp[][]`记录下来,每次开始搜索的时候，如果当前值已经被搜索过就可以直接返回，而不用重复搜索相关的子问题

```
int dfs(int x,int y)
{
    //代表已经搜索过了 不用重复计算
    if(dp[x][y] != 0) return dp[x][y];
    //别忘了深搜的边界
    if(边界条件) return;
    //迭代搜索子过程
    dfs();
    //记录搜索的结果
    dp[x][y] = t;
    return t;
}
```

3.1.1 01背包问题

每个物品选或者不选 事件复杂度o(n^2)

从前i个物品当中选，体积不超过j的所有选法的集合，属性为体积的最大值

只有当更新的时候是从[i-1]转移到[i]的时候才需要逆序去枚举体积

```
//0-1背包问题 从前i个物品当中选择或者不选择 属性是什么
//体积不超过 体积恰好为多少 体积至少是 体积最多是
//对应的属性不一定就是恰好 根据具体情况有多种bia'ni
for(int i = 1;i<=n;i++)
      for(int j = V;j >= v[i];j--)
       {
           //f[i][j] = f[i-1][j];
           //if(j >= v[i]) f[i][j] = max(f[i-1][j-v[i]]+w[i],f[i][j]);
           //逆序j - v[i] < j 算f[j]的时候f[j-v[i]]还是上一层的结果 故而要逆序
           //如果不是逆序的话本来应该用i-1层的结果来更新结果用了第i层
           f[j] = max(f[j],f[j-v[i]]+w[i]);
       }
```

3.1.2 完全背包问题

每个物品有无限多个

同01背包问题的区别：完全背包的i轮状态是由i轮状态本身更新过来的 所以不需要逆序

`f[i][j]` 含义同上

```
    for(int i = 1;i<=n;i++)
      for(int j = v[i];j<=V;j++){
        //f[i][j] = f[i-1][j];
        //同0-1背包相比 只是第i轮的状态还是由i轮更新过来
        //if(j >= v[i]) f[i][j] = max(f[i][j],f[i][j-v[i]]+w[i]);
        //这里不用逆序是因为f[j-v[i]]就是本轮更新过来的 逆序了反而会错误
        f[j] = max(f[j],f[j-v[i]]+w[i]);
        //f[j]  = max(f[j],f[j-v[i]]+w[i]);//直接写这个肯定漏了
}
```

3.1.3 多重背包问题

物品个数有限制

暴力版

实在不记得了再用

```
    //与前面两个相比较多了一个数量第i个物品数量k的限制
    //三重循环 依次枚举前i个 体积 数量
    for(int i = 1;i<=n;i++)
      for(int j = 0;j<=V;j++)
        for(int k = 0;k <= num[i] && k*v[i] <= j;k++)
          f[i][j] = max(f[i][j],f[i-1][j-k*v[i]]+k*w[i]);
```

二进制优化版

优化思路

对每组的物品进行二进制化 比如一组物品有五个 可以分成 1 2 4 三种情况

选的时候可以凑出 0-5当中任意一个情况，把 1 2 4 三种物品打包放入到新的背包中，所有物品处理完成之后

只要对新的背包进行01背包问题处理就可以了

```
    for(int i = 1;i<=n;i++){
        int v,w,s;
        cin>>v>>w>>s;
        //对每个物品组进行二进制处理
        for(int i = 1;i<=s;i*=2){
            g.push_back({i*v,i*w});
            s -= i;
        }
        //除了刚好而二进制以外还剩下的
        if(s) g.push_back({s*v,s*w});
    }
    //剩下的就是对g当中的物品进行01背包问题的处理
    for(int i = 1;i<=g.size();i++){
        int v = g[i-1].first,w = g[i-1].second;
        for(int j = V;j >= v;j--){
            f[j] = max(f[j],f[j-v] + w);
        }
    }
```

3.1.4 分组背包问题

从小到大枚举体积

有 N组物品和一个容量是 V 的背包。

分组背包问题的核心就在于组内的物品都是相互独立的，所以后面 开心的金明那题可以转换为分组背包来处理

每组物品有若干个，同一组内的物品最多只能选一个。

```
//维度优化版  注意分组背包的时候  还是要判断一下体积
for(int i = 1;i<=n;i++)
       for(int j = m;j>=0;j--)
       {
          for(int k = 1;k<=cnt[i];k++)
             if(j >= v[i][k]) f[j] = max(f[j],f[j-v[i][k]]+w[i][k]);
       }
```

总结

对于0-1背包、分组背包、完全背包维度优化之后，都不用处理不选的情况，因为f[j] = f[j]相当于自动考虑过了

```
//三重循环  前i组  体积j   第i组里面选择哪个
for(int i = 1;i<=n;i++){
       //枚举体积
        for(int j = 1;j<=V;j++){
            f[i][j] = f[i-1][j];
            for(int k = 1;k<=s[i];k++){
                //对组内的物品选或者不选进行分析
                if(j >= v[i][k]) f[i][j] = max(f[i][j],f[i-1][j-v[i][k]] +
w[i][k]);
            }
        }
    }
```

3.1.5 混合背包问题

```
//混合背包问题  是在二进制化里面进行体积的m
for(int i = 1;i<=n;i++)
{
    if(s[i] == 0)
    {
        for(int j = v[i];j<=m;j++)
            f[j] = max(f[j],f[j-v[i]]+w[i]);
    }
    else{
        //单独处理s[i] == -1 和 s[i] > 0的情况
        if(s[i] == -1) s[i] = 1;
        for(int k = 1;k<=s[i];k*=2)
        {
            for(int j = m;j>=k*v[i];j--)
            {
                f[j] = max(f[j],f[j-k*v[i]] + k*w[i]);
            }
```

```
            s[i] -= k;
        }
        if(s[i]){
            for(int j = m;j>=s[i]*v[i];j--)
            {
                f[j] = max(f[j],f[j-s[i]*v[i]] + s[i]*w[i]);
            }
        }
    }
}
```

3.1.6 有依赖的背包问题

父节点从大到小枚举体积，并且要预留一部分给父节点

子节点从小到大枚举体积

选某个物品必须连着某个物品一起选

Acwing 10.有依赖的背包问题

```
void dfs(int u)
{
    //分组背包问题
    for(int i = v[u];i<=m;i++) f[u][i] = w[u];
       //对当前结点的边进行遍历
       for(int i = h[u];i!=-1;i = ne[i]){
        //e数组的值是当前边的终点，即儿子结点
        int son = e[i];
        dfs(son);
        //省略了一维i 所以要从大到小枚举 因为默认了加父节点 所以j要大于v[u]
        for(int j = m;j>=v[u];j--){
            //去遍历子节点的组合
            for(int k = 0;k<=j-v[u];k++){
                //这里的f[u][j-k]就相当于除了当前节点son以外，其余的最大值
                f[u][j] = max(f[u][j],f[u][j-k]+f[son][k]);
            }
        }
    }
}
```

Acwing 1074 二叉苹果树

```
void dfs(int u,int fa)
{
    for(int i = h[u];i!=-1;i=ne[i])
    {
        if( e[i] == fa) continue;
        dfs(e[i],u);
        for(int j = m;j>=1;j--)
           for(int k = 0;k<j;k++)
              f[u][j] = max(f[u][j],f[u][j-k-1] + f[e[i]][k]+w[i]);//除了给当前分支 还要留一点给其它分支
    }
}
```

3.1.7 背包问题达到最大价值的时候求方案数

求达到最大价值，有多少种可选方案

//原来的有向无环图是 `f[1][] -> f[2][] -> f[3][]...->f[i-1][]->f[i][j]` 所以正常求路径是从后往前求

最小字典序 将有向无环图改成 `f[i][j] -> f[i-1][] -> f[i-2][]->....->f[2][]->f[1][]` 也是从后往前求 但这时候就是最小字典序

```
//体积为0的时候还是有一种方案数的
g[0] = 1;
for(int i = 1;i<=n;i++)
    for(int j = m;j>=v[i];j--)
    {
        int cnt = 0;
        int maxv = max(f[j],f[j-v[i]]+w[i]);
        if(f[j] == maxv) cnt += g[j] % mod;
        if(f[j-v[i]] + w[i] == maxv) cnt += g[j-v[i]]%mod;
        g[j] = cnt % mod;
        f[j] = maxv;
    }
int t = 0;
for(int i = 1;i<=m;i++)
{
    t = max(t,f[i]);
}
for(int i = 0;i<=m;i++)
{
    if(f[i] == t){
        ans = (ans%mod + g[i] % mod)%mod;
    }
}
```

3.1.8 记录最小字典序路径

```
    for(int i = n;i>=1;i--)
      for(int j = 0;j <= m;j++)
      {
          f[i][j] = f[i+1][j];
          if(j >= v[i]) f[i][j] = max(f[i][j],f[i+1][j-v[i]]+w[i]);
      }

    int k = m;
    int cnt = 0;
    for(int i = 1;i <= n;i++)
    {
         if(k >= v[i] && f[i][k] == f[i+1][k-v[i]] + w[i])
         {
             cout<<i<<" ";
             k -= v[i];
         }
    }
```

3.2 线性DP

3.2.1 数字三角形

从起始点 按照 上下左右/左下右下等顺序走到终点得到的最大值

一般`f[i][j]`就表示起始点走到（i，j）的所有路程的集合，然后对集合进行分析以求出状态转移方程，从而得到最后的结果

898.数字三角形

https://www.acwing.com/problem/content/900/

3.2.2 最长上升子序列模型

题意 给定一个序列 求最长的上升子序列的长度

设`f[i]`为以第i个点结尾的上升子序列 属性为最长值

注意

f[i]前面最长的不一定就是f[i-1]，f[i-1]只是代表以a[i-1]结尾的最长的一个

暴力双重循环版

```
  for(int i = 1;i<=n;i++) f[i] = 1;
    for(int i = 1;i<=n;i++)
    {
      for(int j = 1;j<i;j++){
          if(a[i] > a[j]){
              f[i] = max(f[i],f[j]+1);
          }
      }
    }
```

895.最长上升子序列模型

https://www.acwing.com/problem/content/897/

单调栈优化

其实这题就是一道数据结构

因为要求的是最长上升的那个队列，所以可以用一个单调上升的栈来维护，只有当栈里面的元素越小，这个栈才有可能维护得越大，所以对于一个元素有下面两种情况来解决

- 如果`a[i] > stack.top()` `a[i]`入栈
- 找到一个比`a[i]`大于等于的数 进行替换即可

二分nlogn

```
int find(int x){
    int l = 0,r = tt-1;
    while(l < r){
```

```
        int mid = l + r >> 1;
        if(s[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return l;
}
int main(){
    cin>>n;
    for(int i = 0;i<n;i++)
       scanf("%d",&a[i]);

    s[tt++] = a[0];
    for(int i = 1;i<n;i++)
    {
        if(a[i] > s[tt-1]) s[tt++] = a[i];
        else{
            int x = find(a[i]);//第一个大于等于a[i]的数
            s[x] = a[i];
        }
    }
    printf("%d",tt);
    return 0;
}
```

3.3 状态机模型

3.4 区间dp

- 链式区间dp O(n3)
- 环形区间dp O(n3)

```
//迭代式 常用
//len一般从2开始
//一般情况下len为表示状态可以计算的最小值
for(int len = 可以计算的最小长度; len<=用到的最大的长度;len++)
    for(int l = 1;l+len-1<=2n或者n;l++)
    {
        int r = l + len - 1;
        for(int k = l + 可以使得f数组有意义的最小长度;k<r;k++)//注意这里是k<r!!!不能取到r
            //是k还是k+1根据具体题目的定义来
            f[l][r] = max/min(f[l][r],f[l][k]+f[k+1][r]+具体情况计算的值)
    }
//对于不同的开头 结果可能也不同
int ans = 0;
for(int i = 1;i<=n;i++)
     ans = max(f[i][i+n+x],ans);
//记忆化搜索
碰到已经搜索过的状态则return
```

数论

质数埃氏筛法

O nloglogN 接近线性时间复杂度

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6+10;
int prime[N];
bool st[N];
int cnt;
void primes(int n)
{
    for(int i = 2;i<=n;i++)
    {
        if(prime[i]) continue;//是合数
        cnt++;//否则是质数 个数++
        //只要利用合数 = 质数 * 倍数
        //x * x以前的数如(x-1)*x肯定会被x-1筛掉 所以从x^2开始筛
        for(int j = i;j<=n/i;j++) prime[i*j] = true;
    }
    cout<<cnt<<endl;
}
int main(){
    int n;cin>>n;
    primes(n);
    return 0;
}
```

质数的线性筛法

```
//线性筛法 利用最小质因子
//若v[i] = i说明是质数 存下来
//扫描小于v[i]的每个质数p 令v[i*p] = p
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6+10;
int cnt;
int prime[N],v[N];//v每个数的最小质因子
int n;
void get_primes(int n)
{
    for(int i = 2;i<=n;i++)
    {
        if(v[i] == 0)  v[i] = i,prime[++cnt] = i;
        //给当前数i乘上一个质因子
        for(int j = 1;j<=cnt;j++)
        {
            //利用最小的 当大于v[i]说明不是最小的
            或者当primes[j]超过n的范围的时候 退出循环
            if(prime[j] > v[i] || prime[j] > n/i) break;
            //prime[j]是最小质因子
            v[i*prime[j]] = prime[j];
        }
    }
    cout<<cnt<<endl;
}
```

高级数据结构

1,树状数组的应用

单点修改和区间查询

动态得维护一个数组，可以用于求某个数前面有多少个比它大/小的，或者后面有多少个比它大/小的

同时还可以动态删除某个数并进行动态的查询工作

```
//树状数组用来动态得维护已经插入的点哪些比它大 哪些比它小
//同理也可以用来维护逆序对 对已插入的点通过看哪些点比它大就可以算出来逆序对
//动态的单点修改和区间查询
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;
typedef long long LL;
const int N = 2e5+10;
int tr1[N*4];
int tr2[N*4];
int w[N];
int h1[N],h2[N];
int low1[N],low2[N];
int n;
int lowbit(int x)
{
    return x & -x;
}
//主要操作1 从后往前query
int query(int tr[],int x)
{
    int res = 0;
    for(int i = x;i>0;i-=lowbit(i))
        res += tr[i];
    return res;
}
//主要操作2 从当前往后面进行查询工作
void add(int tr[],int x,int v)
{
    for(int i = x;i<=200000+10;i+=lowbit(i))
         tr[i] += v;
}
int main()
{
    scanf("%d",&n);
    for(int i = 1;i<=n;i++)
    {
        scanf("%d",&w[i]);
    }
    //顺着做一遍
    for(int i = 1;i<=n;i++)
    {
```

```
        //先算左边的
        h1[i] = query(tr1,200000+10) - query(tr1,w[i]);
        low1[i] = query(tr1,w[i]-1);//在他前面比它小
        add(tr1,w[i],1);
    }
    for(int i = n;i>=1;i--)
    {
        h2[i] = query(tr2,200000+10) - query(tr2,w[i]);
        low2[i] = query(tr2,w[i]-1);
        add(tr2,w[i],1);
    }
    LL res1 = 0;
    LL res2 = 0;
    for(int i = 2;i<=n-1;i++)
    {
         res1 += h1[i]*(LL)h2[i];
         res2 += low1[i]*(LL)low2[i];
    }
    printf("%lld %lld",res1,res2);
    return 0;
}
```

维护一个差分数组，以实现区间修改和单点查询操作

```
//区间修改 单点查询
//可以通过维护差分数组来实现
#include <iostream>
#include <algorithm>
using namespace std;
typedef long long LL;
const int N = 1e5+10;
int n,m;
int tr[N*4];
int w[N];
int lowbit(int x)
{
    return x & -x;
}
LL query(int x)
{
    LL res = 0;
    for(int i = x;i>0;i-=lowbit(i))
        res += tr[i];
    return res;
}
void add(int x,int v)
{
    for(int i = x;i<=100000+10;i+=lowbit(i))
       tr[i] += v;
}
int main()
{
    scanf("%d %d",&n,&m);
    for(int i = 1;i<=n;i++)
    {
        scanf("%d",&w[i]);
        add(i,w[i]-w[i-1]);
```

```
    }
    char op[2];
    int k,x,v;
    while(m--)
    {
        scanf("%s",op);
        if(*op == 'Q')
        {
            scanf("%d",&x);
            printf("%lld\n",query(x));
        }else{
            scanf("%d %d %d",&k,&x,&v);
            add(k,v);
            add(x+1,-v);
        }
     }
    return 0;
}

```

2. 线段树的应用

支持区间修改和区间查询，可以维护很多信息

单点修改和区间查询，只要用到pushup操作

```
struct node{
    int l,r;
    LL sum;//区间和
    LL d;//区间最大公约数
}tr[N*4];
LL w[N];
int n,m;
LL gcd(LL a,LL b)
{
    if(!b) return a;
    else return gcd(b,a%b);
}
//函数的重载
void pushup(node &u,node &l,node &r)
{
    u.sum = l.sum + r.sum;
    u.d = gcd(l.d,r.d);
}
void pushup(int u)
{
    pushup(tr[u],tr[u<<1],tr[u<<1|1]);
}
void build(int u,int l,int r)
{
    if(l == r){
        LL t = w[r] - w[r-1];//差分数组
        tr[u] = {l,r,t,t};
    }else{
        tr[u] = {l,r};
```

```
        int mid = l + r >> 1;
        build(u<<1,l,mid),build(u<<1|1,mid+1,r);
        pushup(u);
    }
}
void modify(int u,int x,LL v)
{
    if(tr[u].l == x && tr[u].r == x)
    {
        LL t =  tr[u].sum + v;
        tr[u] = {x,x,t,t};
    }else{
        int mid = tr[u].l + tr[u].r >> 1;
        if(x <= mid)
        {
            modify(u<<1,x,v);
        }else{
            modify(u<<1|1,x,v);
        }
        pushup(u);
    }
}
//当查询的时候 如果子节点对父节点会有影响 这时候我们要返回的就是一个结构体类型
node query(int u,int l,int r)
{
    if(tr[u].l >= l && tr[u].r <= r) return tr[u];
    else{
        int mid = tr[u].l + tr[u].r >> 1;
        if(r <= mid) return query(u<<1,l,r);
        else if(l > mid) return query(u<<1|1,l,r);
        else{
            node left = query(u<<1,l,r);
            node right = query(u<<1|1,l,r);
            node res;
            pushup(res,left,right);
            return res;
        }
    }
}
```

当涉及到区间修改的时候，这个时候我们就需要加入懒标记进行操作，懒标记代表的是以当前区间为根节点的所有子区间需要进行的操作

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int N = 1e5+10;
typedef long long LL;
struct node{
    int l,r;
    LL sum;//区间和
    LL mul;//乘
    LL add;//懒标记
    //先乘后加
```

```
}tr[N*4];
int w[N];
int n,mod;
int m;
void pushup(int u)
{
    tr[u].sum = (tr[u<<1].sum + tr[u<<1|1].sum) % mod;
}
void eval(node &t,int add,int mul)
{
    t.sum = ((LL)t.sum * mul + (LL)(t.r - t.l + 1)*add)%mod;
    t.mul = ((LL)t.mul*mul) % mod;
    t.add = ((LL)t.add*mul + add) % mod;
}
//lazy标记往下传
//这里要注意用父节点信息更新子节点信息的时候，一是子节点的信息要随着父节点的变化而改变
//二是传完之后记得将对应的懒标记进行更新
void pushdown(int u)
{
    //lazy标记往下面传
    node &root = tr[u], &l = tr[u<<1],&r = tr[u<<1|1];
//  eval(tr[u<<1],tr[u].add,tr[u].mul);
//  eval(tr[u<<1|1],tr[u].add,tr[u].mul);
    l.mul = ((LL)l.mul * root.mul)%mod;
    l.add = ((LL)l.add*root.mul + root.add)%mod;
    r.mul = ((LL)r.mul * root.mul)%mod;
    r.add = ((LL)r.add*root.mul + root.add)%mod;
    l.sum = ((LL)(l.sum*root.mul + (LL)(l.r-l.l + 1)*root.add))%mod;
    r.sum = ((LL)(r.sum*root.mul + (LL)(r.r-r.l+1)*root.add))%mod;
    tr[u].add = 0,tr[u].mul = 1;

}
//在build当中自己容易忘记else分支当中区间左右端点赋值
void build(int u,int l,int r)
{
    if(l == r) tr[u] = {l,r,w[l],1,0};
    else{
        tr[u] = {l,r,0,1,0};
        int mid = tr[u].l + tr[u].r >> 1;
        build(u<<1,l,mid),build(u<<1|1,mid+1,r);
        pushup(u);
    }
}
//在查询的时候如果碰到裂开的情况就要pushdown
LL query(int u,int l,int r)
{
    if(tr[u].l >= l && tr[u].r <= r)
    {
         return tr[u].sum;
    }else{
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        LL res = 0;
        if(l <= mid) res = (res+query(u<<1,l,r))%mod;
        if(r > mid)   res  = (res + query(u<<1|1,l,r))%mod;
        return res;
    }
}
```

```cpp
//这里的修改是区间修改 和前面单点修改有所不同
//对应的值由于懒标记的修改而修改 同时懒标记也要修改 相当于pushdown操作去掉了初始化父节点懒标记的操作
void modify(int u,int l,int r,int c,int d)
{
    if(tr[u].l >= l && tr[u].r <= r){
        tr[u].sum = ((LL)tr[u].sum * c) % mod;
        tr[u].sum = ((tr[u].sum + (LL)(tr[u].r - tr[u].l + 1) * d))%mod;
        tr[u].mul = ((LL)tr[u].mul * c)%mod;
        tr[u].add = ((LL)tr[u].add*c + d)%mod;
    }else{
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if(l <= mid)  modify(u<<1,l,r,c,d);
        if(r > mid)   modify(u<<1|1,l,r,c,d);
        pushup(u);
    }
}
int main()
{
    scanf("%d %d",&n,&mod);
    for(int i = 1;i<=n;i++) scanf("%lld",&w[i]);
    build(1,1,n);
    scanf("%d",&m);
    int op,l,r,c;
    while(m--)
    {
        scanf("%d %d %d",&op,&l,&r);
        if(op == 1)
        {
            scanf("%d",&c);
            modify(1,l,r,c,0);
        }else if(op == 2)
        {
            scanf("%d",&c);
            modify(1,l,r,1,c);
        }else{
            printf("%lld\n",query(1,l,r));
        }
    }
    return 0;
}
```

其它算法

RMQ算法