**抽象数据类型：**

**快速排序：**

基本原理：

1. 暴力做法：

首先建立两个数组a和b，然后对存有数据的数组q进行扫描，将小于x的值放入a中，大于x的值放入b中，最后将a和b中的元素从新放入q数组中，一轮结束。

1. 简洁做法：

一个起点为l终点为r的数组q，取中间值x(（l+r）<<1即（l+r）/2)，i和j分别从l和r处开始，但是i=r先开始扫描，直到遇到q[i]>=x为止，然后j=r向左扫描，直到遇到q[j]<=x为止，q[i]与q[j]进行交换，i和j都向前移动一格，然后继续上述过程，直到i=j为止。

//（x可取左边界、右边界或中间值）（建议中间值）

注：x取值可能影响算法，出现死循环或者边界问题。

举例：

对1,2进行快排

如果在下面的代码中将x取值为q[r]，

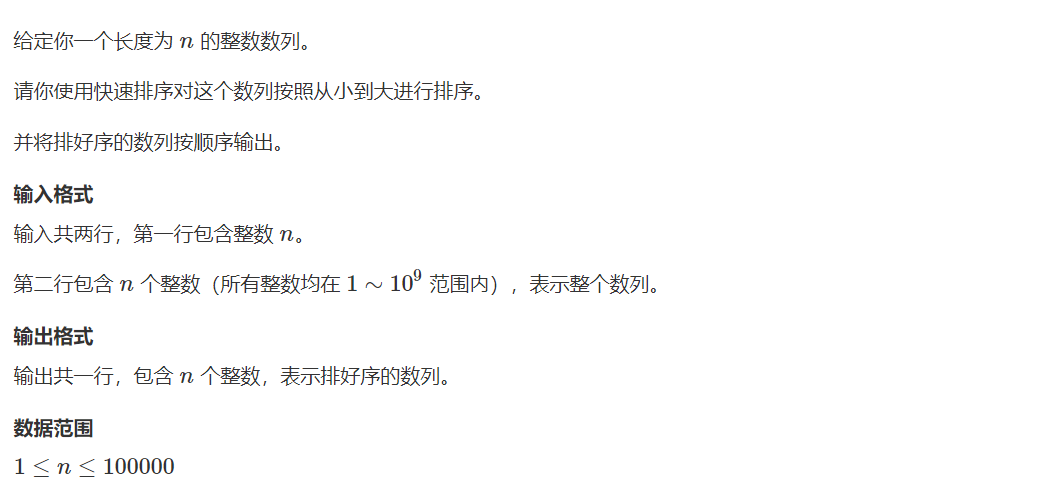
那么进过一轮排序后i=j=1；

下轮排序还是从[0,1]

陷入到了死循环中。

附：快速排序中只是将数组分成两块，左边满足小于等于x，右边大于等于x，与x位置无关。并且分界点是在i==j时。

样例：



#include<iostream>

using namespace std;

const int N=1e6+10;

int q[N];

int n;

void quick\_sort(int q[],int l,int r)

{

if(l>=r)return ;

int x=q[(l+r)/2],i=l-1,j=r+1; //保证是从l和r处开始的

while(i<j)

{

do i++;while(q[i]<x);

do j--;while(q[j]>x);

if(i<j)swap(q[i],q[j]); //交换两个值

}

quick\_sort(q,j+1,r);

quick\_sort(q,l,j); //分成两部分继续排序

}

int main()

{

scanf("%d",&n);

for(int i=0;i<n;i++)

scanf("%d",&q[i]);

quick\_sort(q,0,n-1);

for(int i=0;i<n;i++)

printf("%d ",q[i]);

return 0;

}

附：若想让快速排序具有稳定性，则可将数组的每一个元素变成二元组，使其每个值都不同。

**归并排序：**

基本原理：将一个数组中的数无限拆分更小的几个数组至无法拆分为止，然后将小数组中的有序元素合并至大的数组中，层层合并到原来的数组大小。（具有稳定性）

示例：

5 4 3 2 1

1.先对数组进行拆分：（1）a: 5 4 3 b: 2 1

（2）a: 5 4 c:3 d:2 e:1

(3) a:5 b:4 c:3 d:2 e:1

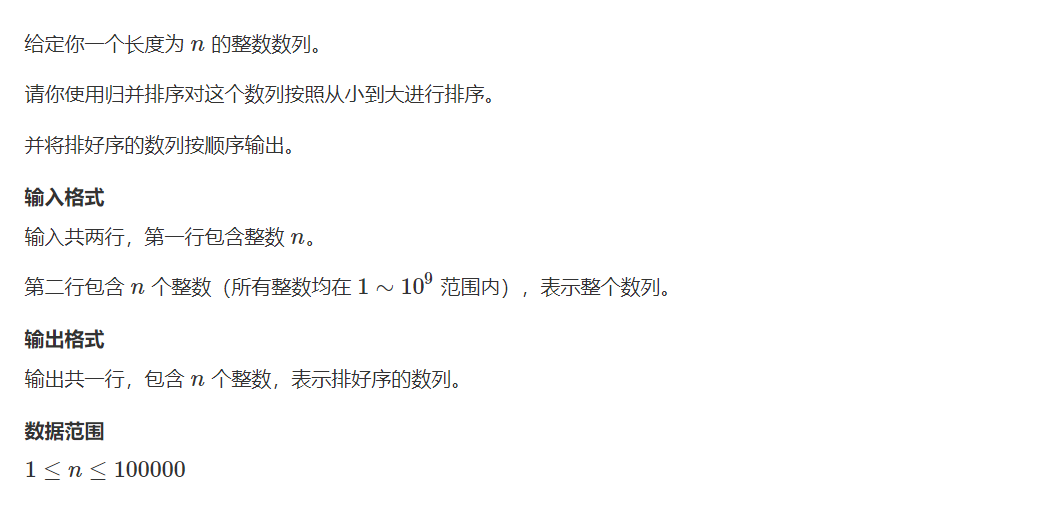
2.合并过程：(1) a:4 5 b:3 c:2 d:1

(2) a:3 4 5 b:1 2

(3) a: 1 2 3 4 5

合并原理：两个小数组中分别有两个指针i和j，都等于各自数组的下限，先开始比较，q[i]<q[j]则将q[i]放入和数组中，否则q[j]放入，然后对应指针加一，继续此过程，直到i或j一方到达自己的上限，然后另一方的剩下的所有元素按照顺序放入和数组中。

样例：



#include<iostream>

using namespace std;

const int N=100010;

int q[N],tmp[N];

int n;

void merge\_sort(int q[],int l,int r)

{

if(l>=r)return; //如果进入的为空或只有一个元素则直接返回上一层

int mid=(l+r)/2; //取中点

merge\_sort(q,l,mid),merge\_sort(q,mid+1,r); //继续递归拆分数组

int k=0,i=l,j=mid+1;

while(i<=mid&&j<=r)

if(q[i]<=q[j]) tmp[k++] = q[i++]; //q[i]<=q[j]的这个条件使其具有稳定性

else tmp [k++] = q [j ++];

while(i<=mid) tmp[k++]=q[i++];

while(j<=r) tmp[k++]=q[j++]; //合并数组

for (int i=l,j=0;i<=r; i++, j++ )

q[i] = tmp[j]; //将和数组中已经排好序的元素导入到原数组中

}

int main()

{

scanf("%d",&n);

for(int i=0;i<n;i++) scanf ("%d",&q[i]);

merge\_sort(q,0,n-1);

for(int i=0;i<n;i++) printf ("%d ",q[i]);

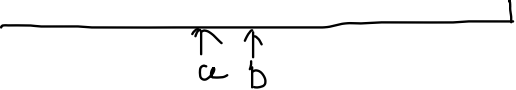
return 0;

}

**整数二分：**



基本原理：



如图所示为一个线性区域，1区域为满足<=x的区域，2区域为满足>=x的区域，而b和a分别为1和2区域的边界，我们要求得这两个边界：  
（1）寻找a边界：

L和r分别为其下限和上限

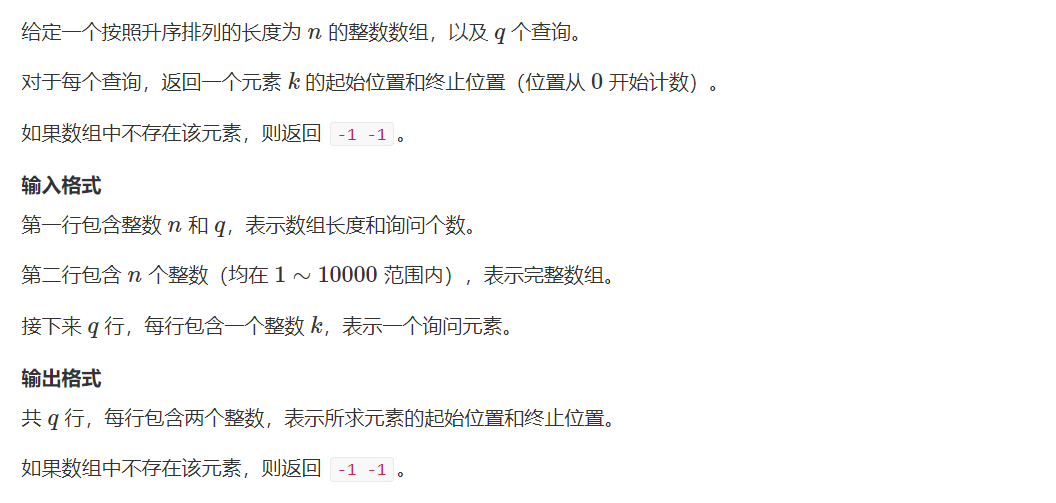
取一个中间值mid=(l+r)/2，判断其是否满足>=x,若满足则让r=mid，否则l=mid+1，接下来重复上过程，直到l=r，得到a=l。

（2）寻找b边界：

取中间值mid=(l+r+1)/2,判断其是否满足<=x,若满足则l=mid，否则r=mid-1,重复上述过程，最后得到l=r=b。

注：为什么第二个mid取(l+r+1)/2，当l=r-1，mid=l，如果判断后为ture，则区间为[mid,r]=[l,r]会陷入死循环中。

样例：



#include<iostream>

using namespace std;

const int N = 100010;

int q[N];

int k,n,m;

int main()

{

scanf("%d%d",&n,&m);

for(int i=0;i<n;i++)

scanf("%d",&q[i]);

while(m--)

{

int x;

scanf ("%d", &x);

int l=0,r=n-1;

while(l<r)

{

int mid = l + r >> 1;

if (q [ mid ] > =x) r = mid;

else l=mid+1;

} //寻求下界a

if(q[l]!=x) printf("-1 -1\n");

else

{

printf("%d ",l);

l=0,r=n-1;

while(l<r)

{

int mid=l+r+1>>1; //(l+r+1)>>1=(l+r+1)/2

if(q[mid]<=x) l=mid;

else r=mid-1;

} //寻求下界b

printf("%d\n",l);

}

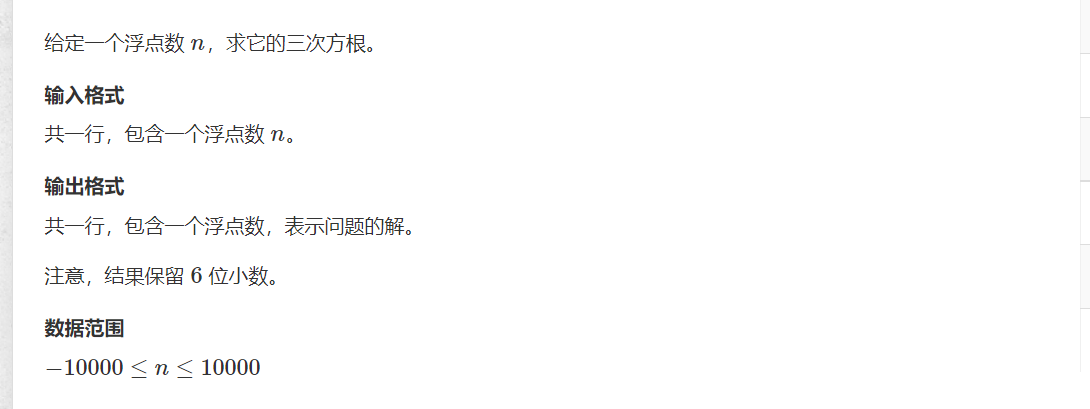
}

return 0;

}

**浮点数二分：**

类似于整数二分，同样是寻找边界的过程，但是区间是可以无限拆分的，所以我们需要设置一个精度来停止循环（或者次数也可以）示例：



#include<iostream>

using namespace std;

int main()

{

double n;

cin>>n;

double l=-10000,r=10000; //r不能取n，当n在[-1,1]时，答案在区域外

while(r-l>10e-8)

{

double mid=(l+r)/2;

if(mid\*mid\*mid>=n) r=mid;

else l=mid;

}

printf("%lf",l);

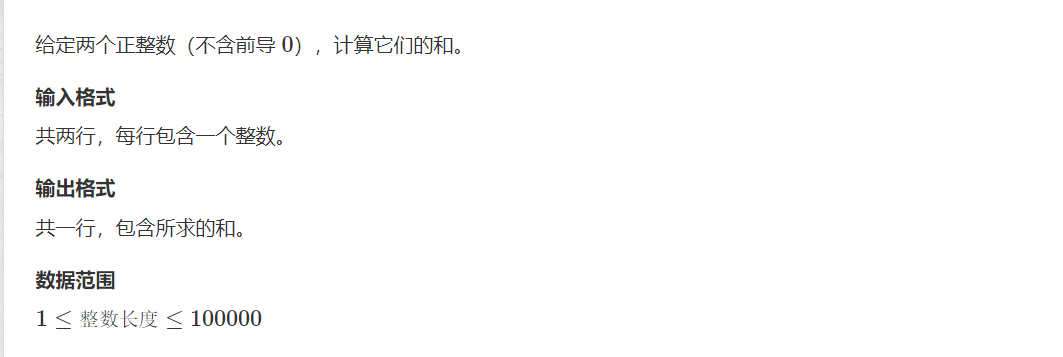
return 0;

}

**高精度：**

1. **加法**：
2. 存储：用数组来存储大整数的每一位，并且存储从个位开始，因为将高位存储在尾部可以方便进位。
3. 函数：使用vector库函数，可以使用size函数计算数组长度len，还可以用push\_back函数在数组的尾部直接增加一个元素，不用考虑数组大小。

示例：



#include<iostream>

#include<vector>

using namespace std;

vector<int> add(vector<int> A,vector<int> B)

{

vector<int> C;

int t=0;

for(int i=0;i<A.size()||i<B.size();i++)

{

if(i<A.size()) t+=A[i];

if(i<B.size()) t+=B[i];

C.push\_back(t%10); //t%10正好可以得到相应位上的数

t/=10; //t/10>0正好与下次循环的数相加相当于进位

}

if(t) C.push\_back(1); //最后一次循环中t>0则在尾部加一

return C;

}

int main()

{

string a,b;

vector<int> A,B;

cin>>a>>b; //用字符串一次性录入

for(int i=a.size()-1;i>=0;i--) A.push\_back(a[i]-'0'); //向尾部增加一个元素

for(int i=b.size()-1;i>=0;i--) B.push\_back(b[i]-'0');

auto C=add(A,B);

for(int i=C.size()-1;i>=0;i--)

printf("%d",C[i]);

}

1. **减法：**
2. 与加法类似，减法会有向前借位的问题，和得出答案为负的问题。所以要分两种情况：

A-B 1.A>B 2.A<B

1情况就用A-B即可，2情况则用B-A最后在答案上添上负号。

1. 需设置t变量用来存储低位存在的借位问题：

则有：

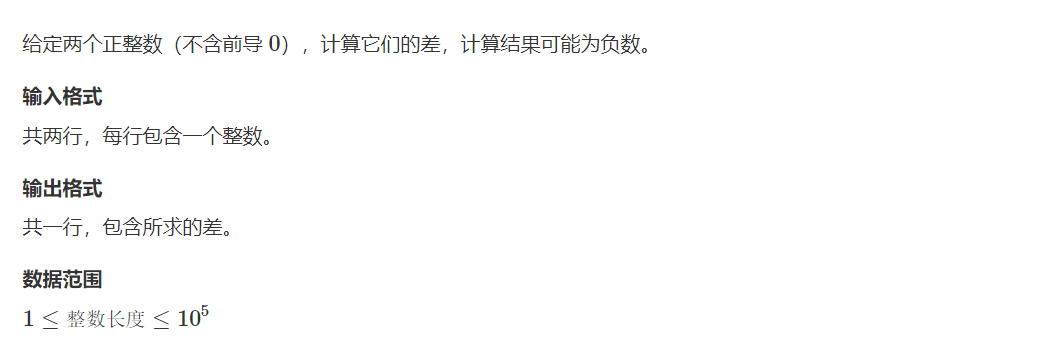
>=0, A[i]-B[i]-t

A[i]-B[i]-t



<0, A[i]-B[i]+10-t

样例：



#include<iostream>

#include<vector>

\

using namespace std;

int cmp(vector<int> &A,vector<int> &B) //判断A是否>=B

{

if(A.size()!=B.size())

return A.size()>B.size();

else

{

for(int i=A.size()-1;i>=0;i--)

if(A[i]!=B[i])

return A[i]>B[i];

}

return 1;

}

vector<int> sub(vector<int> &A,vector<int> &B)

//&A,&B直接调用内存，减少复制带来的额外时间

{

vector<int> C;

for(int i=0,t=0;i<A.size();i++)

{

t=A[i]-t;

if(i<B.size()) t-=B[i];

C.push\_back((t+10)%10); //统一A[i]-B[i]-t的问题

if(t<0)t=1;

else t=0;

}

while(C.size()>1&&C.back()==0) C.pop\_back(); //去除前端的0

return C;

}

int main()

{

string a,b;

cin>>a>>b;

vector<int> A,B,C;

for(int i=a.size()-1;i>=0;i--) A.push\_back(a[i]-'0');

for(int i=b.size()-1;i>=0;i--) B.push\_back(b[i]-'0');

if(cmp(A,B))

{

C=sub(A,B);

for(int i=C.size()-1;i>=0;i--)

printf("%d",C[i]);

}

else

{

C=sub(B,A);

printf("-");

for(int i=C.size()-1;i>=0;i--)

printf("%d",C[i]);

}

return 0;

}

1. **乘法：（一个大整数与小整数相乘）**
2. 一个大整数由数组A存储，小整数用int b存储，用t来存储需要进位的大小

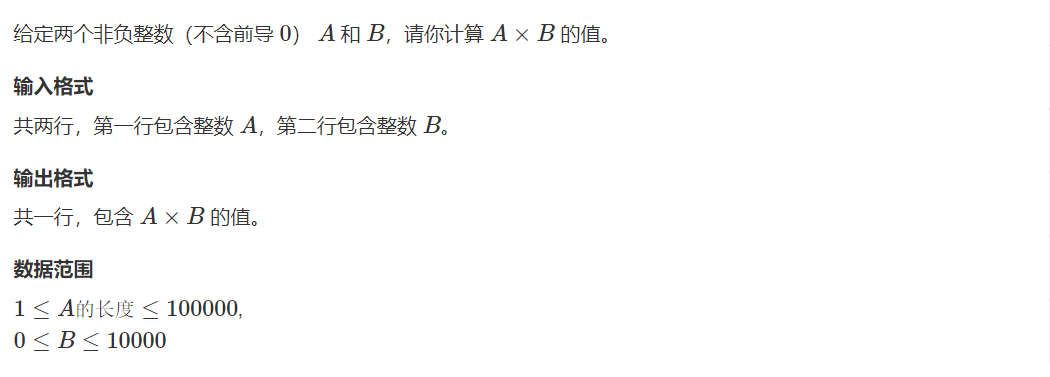
A[5] A[4] A[3] A[2] A[1] A[0]

X a

如图所示第一位相乘后就有，C[0]=(A[0] \* a) % 10,需要进位的t=(A[0]\*a)/10.

类似的C[1]=(A[1]\*a + t ) % 10,需要进位的部分t=(A[1]\*a + t )/10，剩下以此类推。

样例：



#include<iostream>

#include<vector>

using namespace std;

vector<int> mul(vector<int> &A,int b)

{

vector<int> C;

for(int i=0,t=0;i<A.size()||t;i++)

//判断条件中多出判断t在A的最后一位乘完后是否为0，否则进位

{

if(i<A.size()) t+=A[i]\*b;

C.push\_back(t%10);

t/=10;

}

while(C.back()==0&&C.size()>1) C.pop\_back(); //去除高位上的0

return C;

}

int main()

{

string a;

int b;

cin>>a>>b;

vector<int> A,C;

for(int i=a.size()-1;i>=0;i--) A.push\_back(a[i]-'0');

C=mul(A,b);

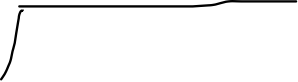
for(int i=C.size()-1;i>=0;i--) printf("%d",C[i]);

return 0;

}

1. **除法：**
2. 与前面不同，除法要先从高位开始，依次进行
3. 图示

C1 C2 C3 C4 C5

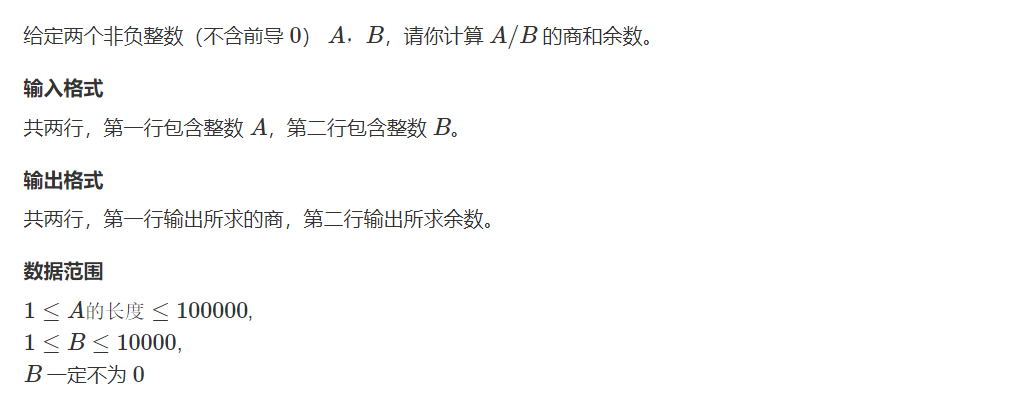


b A5 A4 A3 A2 A1

由图所示，C1=A5/b, t=A5 % b ,C2=(A4+10\*t)/b ,t=(10\*t+A4)%b，依次类推

T是用来存储上一位的余数，直到最后一位，t即为整个数除以b的余数。

示例：



#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

vector<int> div(vector<int> &A,int b,int &x) //x用来存储余数

{

vector<int> C;

int t=0;

for(int i=A.size()-1;i>=0;i--)

{

C.push\_back((10\*t+A[i])/b);

t=(10\*t+A[i])%b;

}

x=t;

reverse(C.begin(),C.end()); //将数组进行颠倒方便去高位0

while(C.back()==0&&C.size()>1) C.pop\_back();

return C;

}

int main()

{

string a;

int b,x;

cin>>a>>b;

vector<int> A,C;

for(int i=a.size()-1;i>=0;i--) A.push\_back(a[i]-'0');

C=div(A,b,x);

for(int i=C.size()-1;i>=0;i--) printf("%d",C[i]);

printf("\n");

printf("%d",x);

return 0;

}

**前缀和与差分：**

1. **前缀和：**

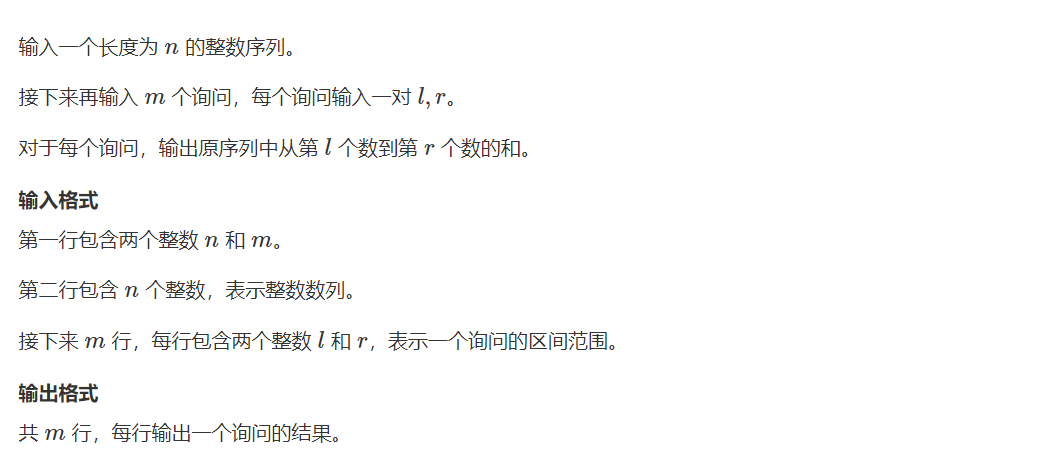
一组数列a[n],s[n]为为其前缀和，在解决区间和问题时，较为简便。

前缀和s[n]=s[1]+s[2]+…+s[n].

有前缀和公式：s[i]=s[i-1]+a[i];

规定s[0]=0,以便解决s[1]=s[0]+a[1]的问题。

样例：



#include<iostream>

using namespace std;

const int N = 100010;

int n,m;

int q[N],s[N];

int main()

{

scanf("%d%d",&n,&m);

s[0]=0;

for(int i=1;i<=n;i++)

{

scanf("%d",&q[i]);

s[i]=s[i-1]+q[i];

}

while(m--)

{

int l,r;

scanf("%d%d",&l,&r);

printf("%d\n",s[r]-s[l-1]); //输出[l,r]的区间和

}

return 0;

}

1. **子矩阵的和**

基本原理：

1. 求子矩阵的和:

如图所示为一个二维矩阵，a[x][y]为其中的一个元素，若求其前缀和s[i][j]=s[i-1][j]+s[i][j-1]-s[i-1][j-1]+a[i][j]。

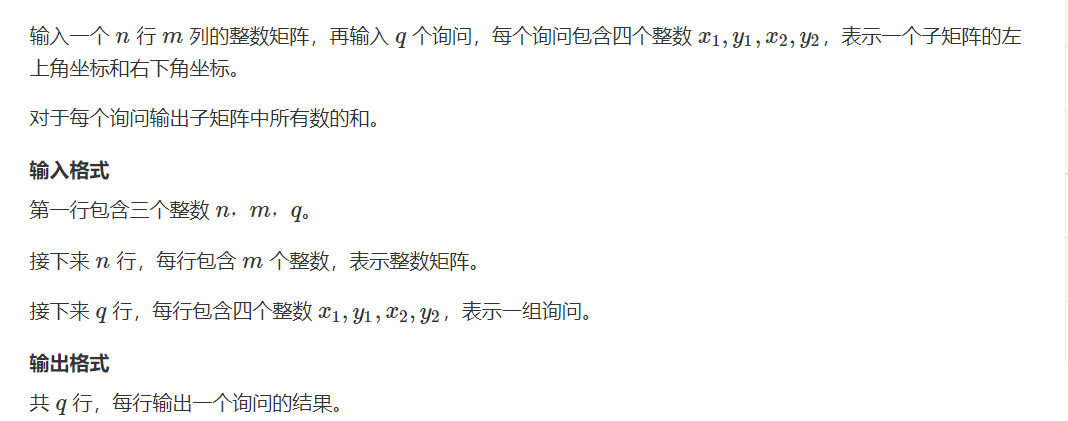
(i-1,j-1)

(I,j)

1. 求矩阵的区间和：

若求（x1,y1）为左上角(x2,y2)为右下角的区间矩阵的和值，s=s[ x2 ][ y2 ]-s[ x2 ][ y1-1 ]-s[ x1-1 ][ y2 ]+s[x1-1][y1-1].

样例：



#include<iostream>

using namespace std;

const int N = 1010;

int a[N][N],s[N][N];

int main()

{

int n,m,q;

scanf("%d%d%d",&n,&m,&q);

for(int i=1;i<=n;i++)

for(int j=1;j<=m;j++)

{

scanf("%d",&a[i][j]);

s[i][j]=s[i-1][j]+s[i][j-1]-s[i-1][j-1]+a[i][j]; //计算矩阵的前缀和

}

while(q--)

{

int x1,x2,y1,y2;

scanf("%d%d%d%d",&x1,&y1,&x2,&y2);

printf("%d\n",s[x2][y2]-s[x2][y1-1]-s[x1-1][y2]+s[x1-1][y1-1]); //计算区间矩阵和

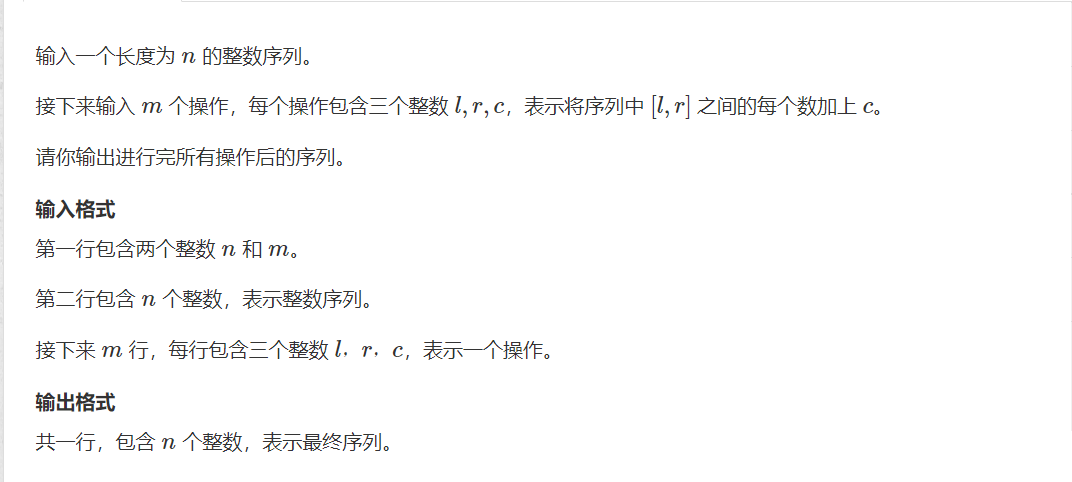
}

return 0;

}

1. **差分:**
2. 概念：差分是前缀和的逆运算 ，a[n]=b[1]+…+b[n],则称b为a的差分。
3. 基本原理：如果要在[l,r]区间上给a[i]都加上个c则只需要给b[l]+=c,b[r+1]-=c即可，同样的初始差分数组也同样可利用这个规律，即i=l=r区间上插入a[i]即可，即b[i]+=c,b[i+1]-=c.

样例：



#include<iostream>

using namespace std;

const int N = 100010;

int a[N],b[N];

int n,m;

void insert(int l,int r,int c) //插入函数

{

b[l]+=c;

b[r+1]-=c;

}

int main()

{

scanf("%d%d",&n,&m);

b[0]=0;

for(int i=1;i<=n;i++)

{

scanf("%d",&a[i]);

insert(i,i,a[i]); //用插入函数制造差分函数

}

while(m--)

{

int l,r,c;

scanf("%d%d%d",&l,&r,&c);

insert(l,r,c);

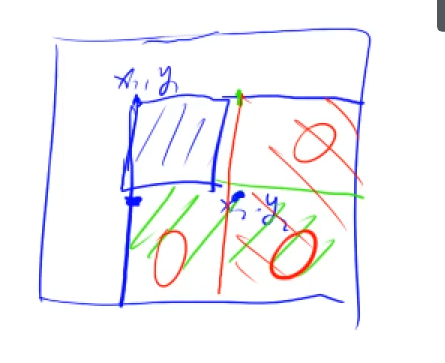
}

for(int i=1;i<=n;i++) b[i]+=b[i-1],printf("%d ",b[i]); //a[i]=b[0]+...+b[i]

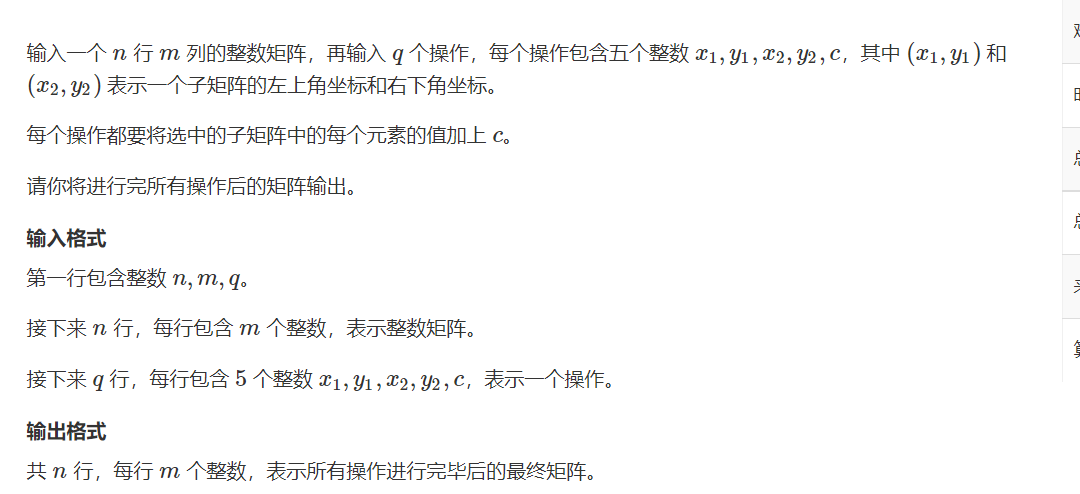
return 0;

}

1. **差分矩阵：**
2. 基本原理：为求区间矩阵和的逆运算，a为b数组的前缀和，b就为a的差分。
3. 如何给矩阵a的某个区间加上c：如图所示，给[x1,y1]到[x2,y2]区间加上个c，则首先有b[x1][y1]+=c，但为了解决只有[x1,y1]到[x2,y2]区间加c,则有b[x2+1][y1]-=c,b[x1][y2+1]-=c,最后还需要b[x2+1][y2+1]+=c，解决在重叠部分多减了c的问题。
4. 如何构造差分矩阵：类似给区间加c，构造相当于在左上角为[x,y],右下角为[x,y]的矩阵上加c,.



样例：



#include<iostream>

using namespace std;

const int N = 1010;

int a[N][N],b[N][N];

int m,n,q;

void insert(int x1,int y1,int x2,int y2,int c) //给[x1,y1]到[x2,y2]区间上加c操作

{

b[x1][y1]+=c;

b[x2+1][y1]-=c;

b[x1][y2+1]-=c;

b[x2+1][y2+1]+=c;

}

int main()

{

scanf("%d%d%d",&n,&m,&q);

for(int i=1;i<=n;i++)

for(int j=1;j<=m;j++)

{

scanf("%d",&a[i][j]);

insert(i,j,i,j,a[i][j]);

}

while(q--)

{

int x1,y1,x2,y2,c;

scanf("%d%d%d%d%d",&x1,&y1,&x2,&y2,&c);

insert(x1,y1,x2,y2,c);

}

for(int i=1;i<=n;i++)

{

for(int j=1;j<=m;j++)

{

b[i][j]+=b[i-1][j]+b[i][j-1]-b[i-1][j-1]; //区间矩阵前缀和

printf("%d ",b[i][j]);

}

printf("\n");

}

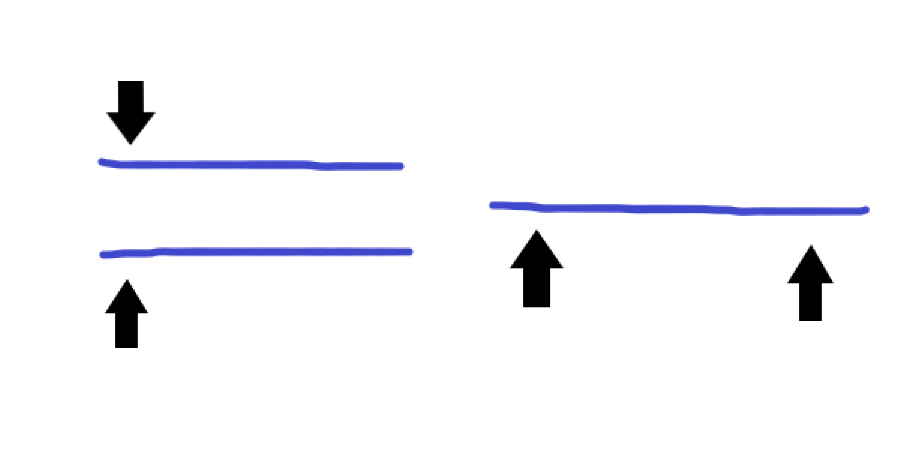
return 0;

}

**双指针算法：**

1. 基本原理：

两种：（1）类似归并 （2）类似快排



核心思想：

将 For(int i=0;i<n;i++)

For(int j=0;j<n;j++)

这样的朴素算法（时间复杂度为O(n^2)）优化为时间复杂度为（O(n)）

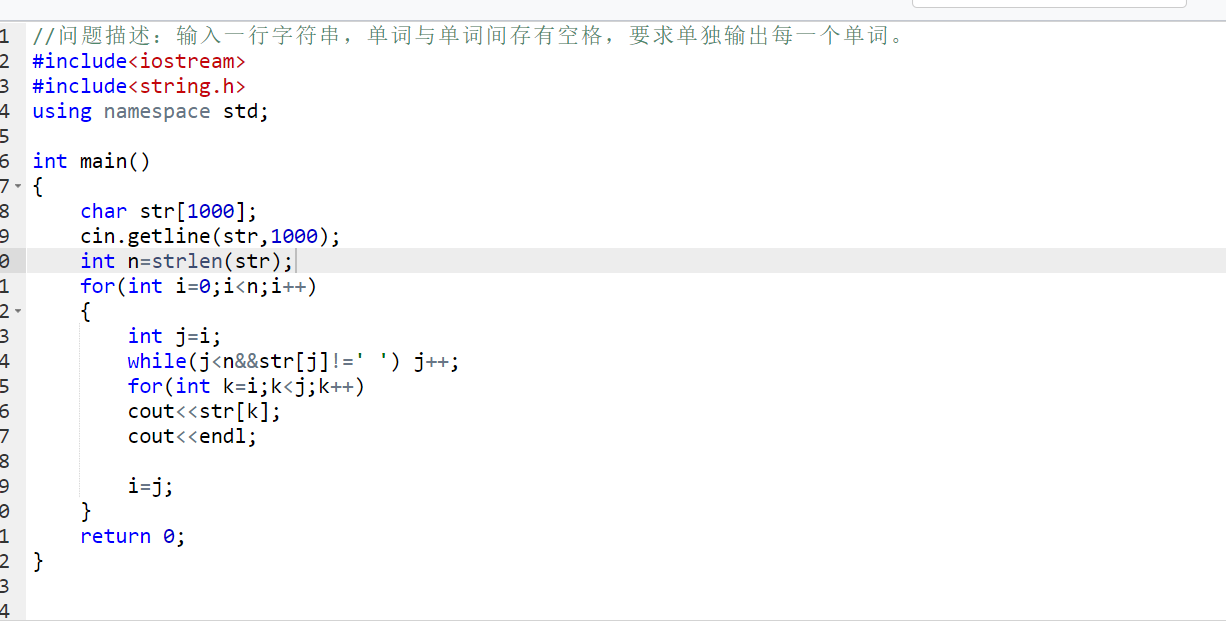
For(int i=0,i=0;i<n;i++)

{

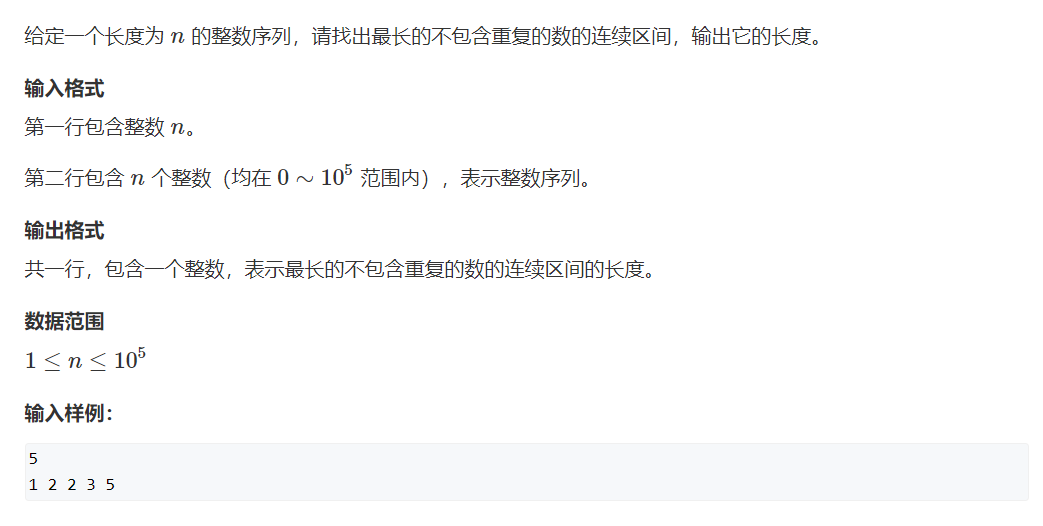
While(j<i&&check(I,j))

//基本逻辑

}



样例：



朴素做法（O(n^2)）:

for(int i=0;i<n;i++)

for(int j=0;j<=i;j++)

if(check(i,j))

res=max(res,i-j+1);

双指针做法（O(2\*n)）:

i++,向[j,i]中添加一个元素，然后检验i到j的最大长度i++的过程中，j++,时间复杂度就为O(2\*n)



#include<iostream>

using namespace std;

const int N = 1e5+10;

int a[N],s[N]={0};

int main()

{

int n;

scanf("%d",&n);

for(int i=0;i<n;i++) cin>>a[i];

int res=0;

for(int i=0,j=0;i<n;i++)

{

s[a[i]]++; //每当向[j,i]区间中添加一个元素，a[i]标记加1

while(s[a[i]]>1)

{

s[a[j]]--; //每当向[j,i]区间中减少一个元素，a[i]标记减1

j++;

}

res=max(res,i-j+1); //在增缩区间中找出最大值

}

cout<<res<<endl;

return 0;

}

// 每当添加区间内元素时，只用看被添加元素的标记值就可判断是否需要缩短区间，因为区间内的所有数都有其标记值等于1

**位运算：**

1. 基本原理：
2. 附加知识点：&1：取数的二进制表达式的第0位，lowbit ( x ):返回x的最后一位1的位置。
3. &1为什么可以取x的个位：(x<<k)&1可以取x的第k位

例如：x=1010 那么x&1=0000,x=1001 x&1=1，1=0001

1. Lowbit的运算：

x&-x=x&(~x+1):

x=1010…10000

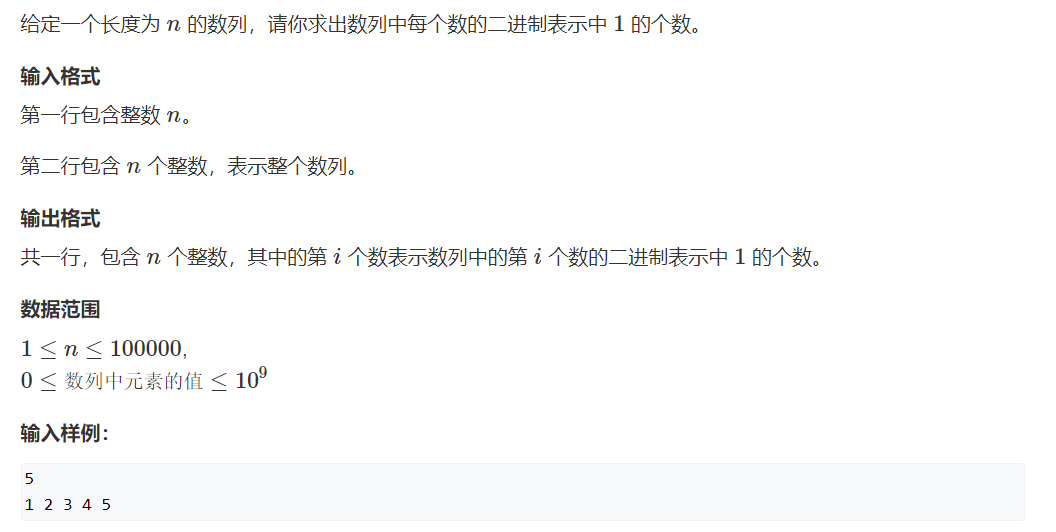
~x=0101…01111

~x+1=0101…10000,

X&(~x+1)=0000…10000

补码=反码+1；

样例：



#include<iostream>

using namespace std;

int lowbit(int x)

{

return x&-x;

}

int main()

{

int x,n;

cin>>n;

while(n--)

{

cin>>x;

int res = 0;

while(x) x-=lowbit(x),res++; // 每次减去x的最后一位

cout<<res<<" ";

}

return 0;

}

**离散化（整数、保序）：**

(1)基本原理：在面对大数据计算其区间值其中会出现许多无用值，及类似=0，这时需要对其进行离散化操作，类似于系数矩阵的三元组操作，即只留下有效值进行存储。

（2）基础知识:1. pair是C++中一种模板类型。每个pair对象可以存储两个值，这两个值可以是不同的数据类型。存储的值可以是基本数据类型也可以是自定义数据类型.

**声明命名空间：**

using namespace std;   
或  
using std::pair;  
pair<int, int> pdata;  
或使用全名  
std::pair<int, int> pdata;

pair有两个属性：first和second。

pair<int, int> p1(1, 2);

p1.first = 11; //修改第一个数值

p1.second = 22; //修改第二个数值

cout << p1.first << "," << p1.second << endl;

for(auto item:add) // 对add中的元素进行遍历

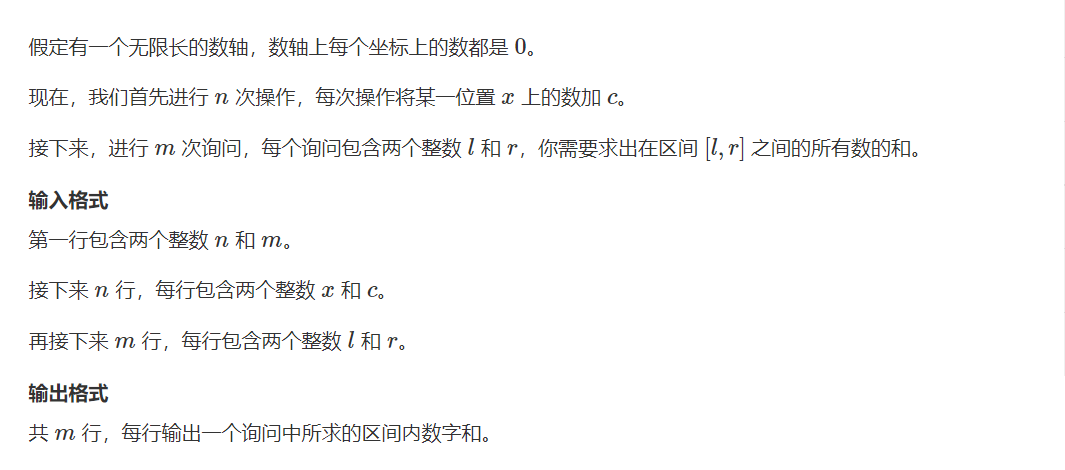
{

a[find(item.first)]+=item.second;

}



样例：



//离散化操作：

1. 面对本题一个无限长的轴上所有的坐标值都为0，会对其进行n次操作，和m次查询，查询中每次2个值，所以最坏情况需要2\*m+n长度的数组存储查询对应位置上的值.防止越界+30.
2. 其次要将数轴上离散的点集中在一个数组中，一个坐标对应数组的下标，对应其值。所以需要由其坐标查找其下标（即其相对位置）,用对分查找，区间值通过合并数组前缀和计算。

#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

typedef pair<int,int> pii; //相当于二元组

const int N = 300030; //最多需要n+l+r个元素，避免越界+10

int a[N],s[N]; //a记录元素值，s为前缀和

vector<int> alls; //记录所有的下标

vector<pii> num,add; //num记录l，r ，add记录操作中位置x和加上的c

int n,m;

int find(int x) //二分查找x位置在alls所处的位置

{

int l=0,r=alls.size()-1;

while(l<r)

{

int mid=(l+r)>>1;

if(alls[mid]>=x) r=mid;

else l=mid+1;

}

return l+1; //+1的原因是，求前缀和时默认s[0]=0，方便计算。

}

int main()

{

cin>>n>>m;

for(int i=0;i<n;i++)

{

int x,c;

scanf("%d%d",&x,&c);

add.push\_back({x,c});

alls.push\_back(x);

}

for(int i=0;i<m;i++)

{

int l,r;

scanf("%d%d",&l,&r);

num.push\_back({l,r});

alls.push\_back(l);

alls.push\_back(r);

}

sort(alls.begin(),alls.end()); //排序需要用到的所有位置

alls.erase(unique(alls.begin(),alls.end()),alls.end()); //删除重复的元素（unique函数为将独特的元素和重复的元素分开，返回的是重复的第一个元素的位置）

for(auto item:add) // 对add中的元素进行遍历

a[find(item.first)]+=item.second; //二分查找时+1使从1开始alls.size()结束

for(int i=1;i<=alls.size();i++) //前缀和从1开始（alls数组从0开始）所以<=alls.size()

s[i]=s[i-1]+a[i]; //求前缀和

for(auto item:num)

{

int l= find(item.first);

int r= find(item.second);

printf("%d\n",s[r]-s[l-1]);

}

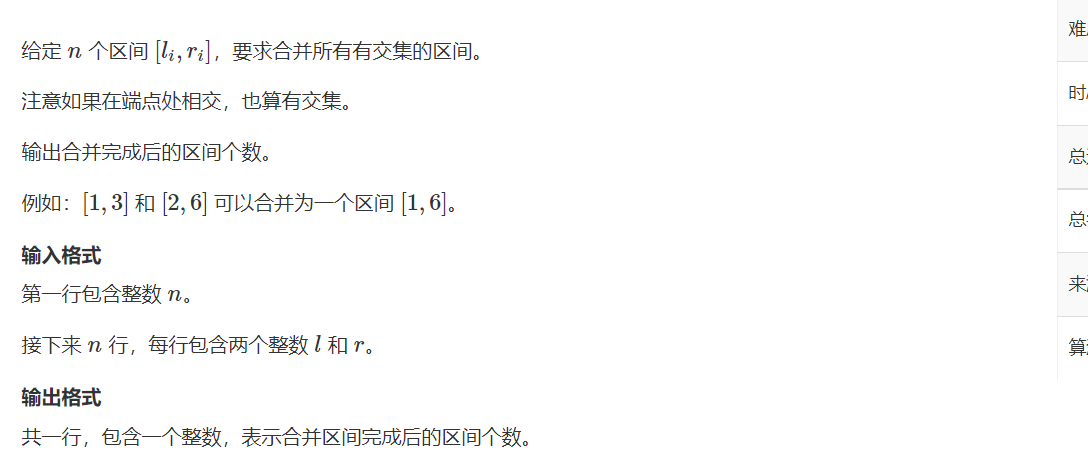
return 0;

}

**区间和：**

1. 基本原理：界定两个区间左边界和有边界之间的关系，来判断两个区间的关系：1.包含 2.有一部分交集 3.没有交集

样例：



#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;

typedef pair<int,int> pii;

vector<pii> segs;

void merage(vector<pii> &segs)

{

vector<pii> res;

sort(segs.begin(),segs.end()); //按照右端点优先排序

int st = -2e9,ed=-2e9;

for(auto seg:segs)

{

if(ed<seg.first) //有边界与下一个区间的左边界的大小，判断是否有交集

{

if(st!= -2e9) res.push\_back({st,ed});

st=seg.first,ed=seg.second; //如果不是第一次就更新st和ed

}

else ed=max(ed,seg.second);

}

if(st!=-2e9) res.push\_back({st,ed}); //将最后一个区间加入

segs=res;

}

int main()

{

int n;

cin>>n;

for(int i=0;i<n;i++)

{

int l,r;

scanf("%d%d",&l,&r);

segs.push\_back({l,r});

}

merage(segs);

cout<<segs.size()<<endl;

return 0;

**链表：**

1. **单链表：**

实现： 1.结构体+指针：Struct Node

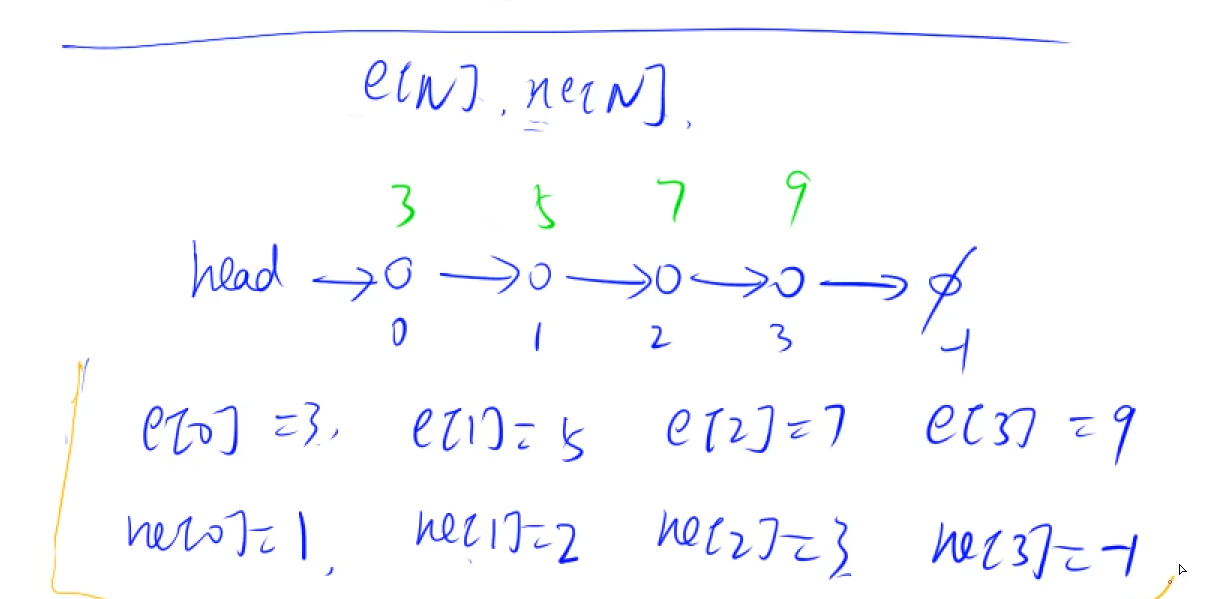
{

Int val;

Node \*next;

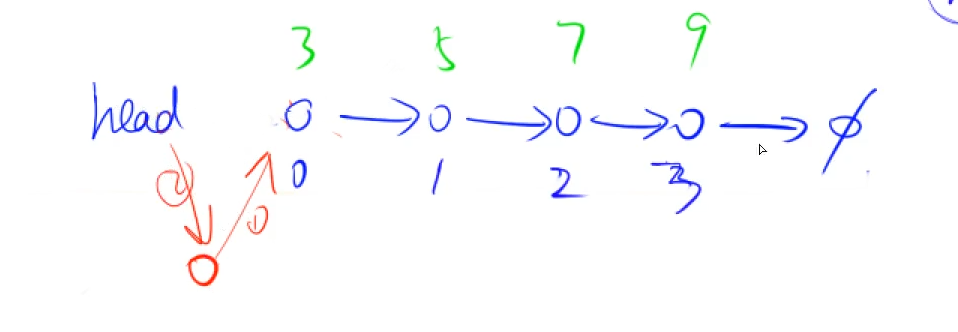
}；

1. 数组模拟：

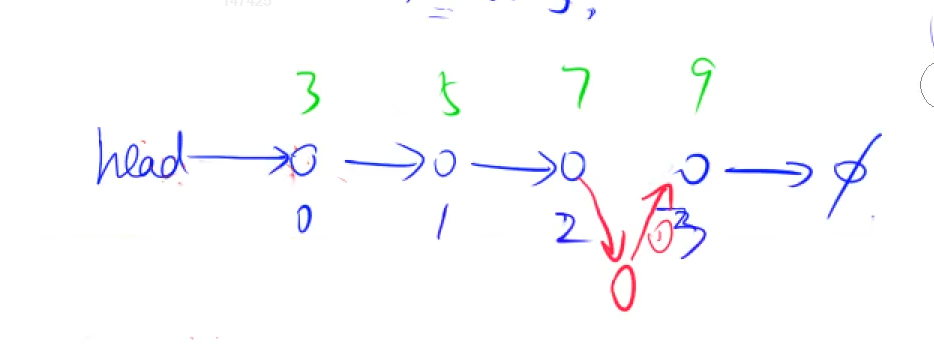
e[ n ] 相当于val存储值， ne[ n ]相当于next指针存储下一个节点的位置。

链表的操作：

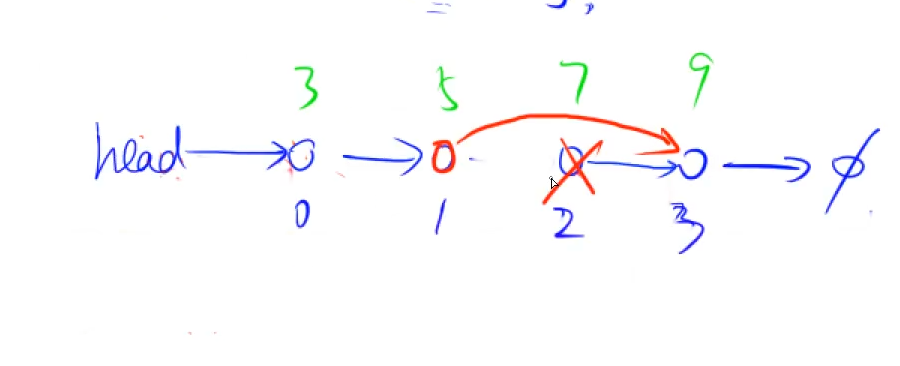
1. 插入（单链表中，在O（1）的时间内插入只能插入到k点之后，之前只能通过遍历）：1.插入到头节点之后：首先将插入点的next指针指向head的next指针所指向的位置，然后将head的next指针指向插入点的位置。



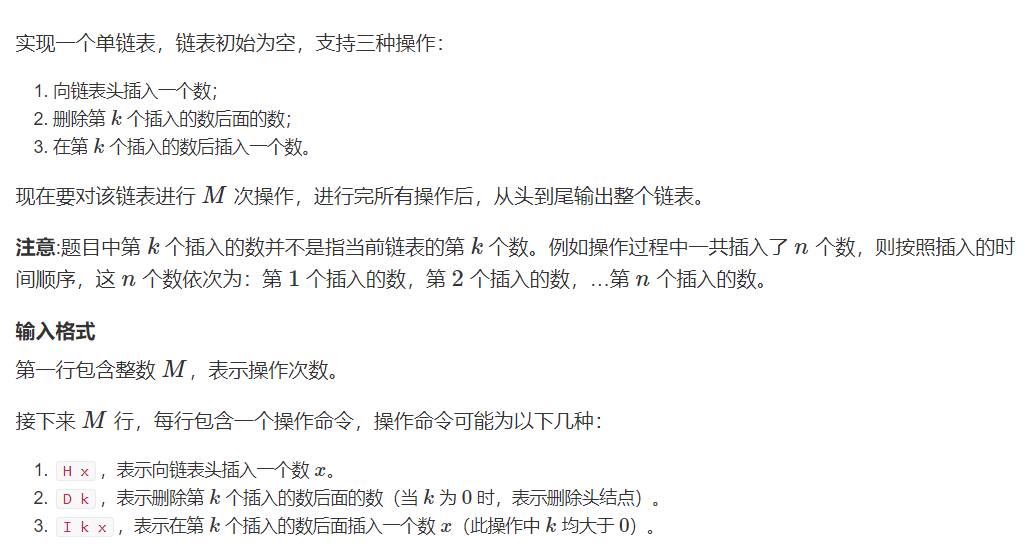
2.插入到第k个点：类似插入到头结点之后，即先将插入点的指针指向第k个点的next指针所指位置，然后将第k个点的指针指向插入点的位置



1. 删除：直接将删除点前一个节点的指针指向删除点指针指向的位置



样例：



#include<iostream>

using namespace std;

const int N = 100010;

//head指针是存储头节点的位置

//e数组是存储k位置处的节点值，en存储第k个节点的下一个节点的位置

//idx存储当前使用到的位置

int head,e[N],en[N],idx;

//初始化链表

void init()

{

head=-1,idx=0;

}

//插入节点到头结点

void add\_to\_head(int x)

{

e[idx]=x;

en[idx]=head;

head=idx++;

}

//插入到第k个节点之后

void add(int k,int x)

{

e[idx]=x;

en[idx]=en[k];

en[k]=idx++;

}

//删除第k点后面的数

void remove(int k)

{

en[k]=en[en[k]];

}

int main()

{

int m;

scanf("%d",&m);

init();

while(m--)

{

char op;

cin>>op;

int k,x;

if(op=='H')

{

scanf("%d",&x);

add\_to\_head(x);

}

else if(op=='D')

{

scanf("%d",&k);

if(k==0)

head=en[head]; //删除头结点；

else

remove(k-1);

}

else

{

scanf("%d%d",&k,&x);

add(k-1,x);

}

}

for(int i=head;i!=-1;i=en[i]) //依次输出链表上的值;

cout<<e[i]<<" ";

cout<<endl;

return 0;

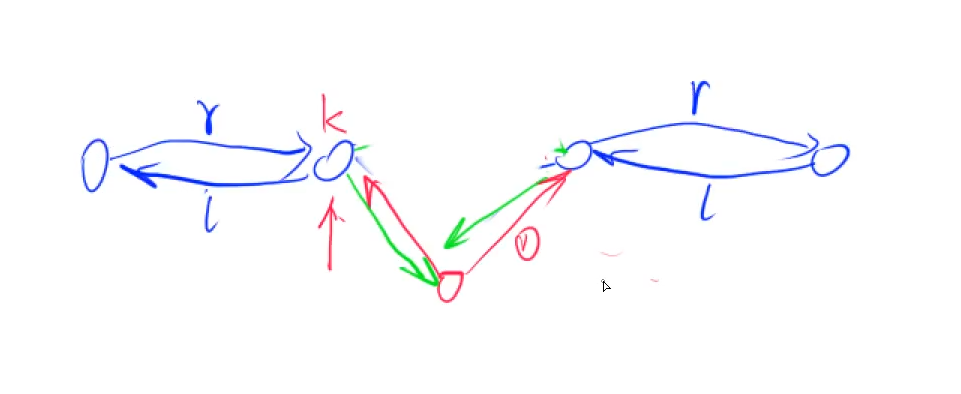
}

**2.双链表：**

(1)实现方法：数组实现：

类似于单链表，双链表e[N]存储节点值，l[N]存储节点前一个节点的位置，r[N]存储节点后一个节点的位置。(为了简便指针为0的节点为头指针，最后一个点为尾指针)

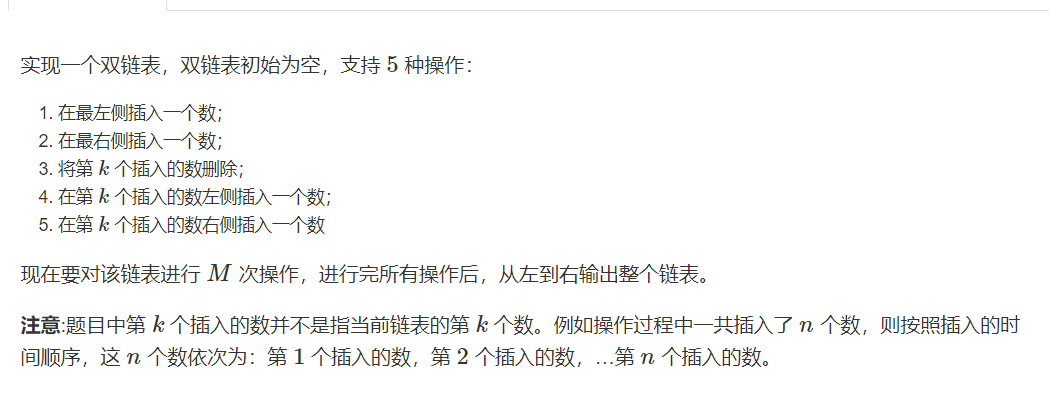
1. 操作：
2. 插入：（1）在k节点的右边插入：首先将插入节点的左指针指向k节点，右指针指向k节点的下一个节点，然后将k节点的右边一个节点的左指针指向插入节点，k节点的右指针指向插入节点。



（2）在k的左边插入：调用右边插入的函数即可，即插入l[k]节点的右边。

2.删除：删除节点的左边一个节点的右指针指向其右边一个节点，右边一个节点左指针指向其左边一个节点。

样例：



#include<iostream>

using namespace std;

const int N = 100010;

int e[N],l[N],r[N],idx;

//初始化

void init()

{

r[0]=1,l[1]=0;

idx=2; //0，1是头指针和尾指针,不带值,idx从2开始所以操作时k+1

}

//向第k个节点右边插入

void add\_to\_right(int k,int x)

{

e[idx]=x;

l[idx]=k;

r[idx]=r[k];

l[r[k]]=idx;

r[k]=idx++;

}

//删除第k个节点

void remove(int k)

{

r[l[k]]=r[k];

l[r[k]]=l[k];

}

int main()

{

int m;

scanf("%d",&m);

init();

while(m--)

{

string op;

cin>>op;

int k,x;

if(op=="IL")

{

scanf("%d%d",&k,&x);

add\_to\_right(l[k+1],x);

}

else if(op=="IR")

{

scanf("%d%d",&k,&x);

add\_to\_right(k+1,x);

}

else if(op=="D")

{

scanf("%d",&k);

remove(k+1);

}

else if(op=="L")

{

scanf("%d",&x);

add\_to\_right(0,x);

}

else

{

scanf("%d",&x);

add\_to\_right(l[1],x);

}

}

for(int i=r[0];i!=1;i=r[i])

cout<<e[i]<<" ";

cout<<endl;

return 0;

}

**栈和队列：**

**1.栈：**

**先进后出**

运用数组对其模拟：tt用来表示栈顶，初始时tt=0，通过tt--和str[ tt ++ ]=x对其完成入栈和出栈操作，if( tt >0) 判断栈中是否为空。

样例：



#include<iostream>

using namespace std;

const int N = 100010;

int str[N];

int main()

{

int m,x;

int top=-1; //栈顶指针

scanf("%d",&m);

while(m--)

{

string op;

cin>>op;

if(op=="push")

{

scanf("%d",&x);

str[++top]=x; //入栈

}

else if(op=="pop")

top--; //模拟栈顶弹出

else if(op=="empty")

{

if(top<0)

printf("YES\n"); //通过栈顶指针判断是否为空

else

printf("NO\n");

}

else

printf("%d\n",str[top]); //输出栈顶元素

}

return 0;

}

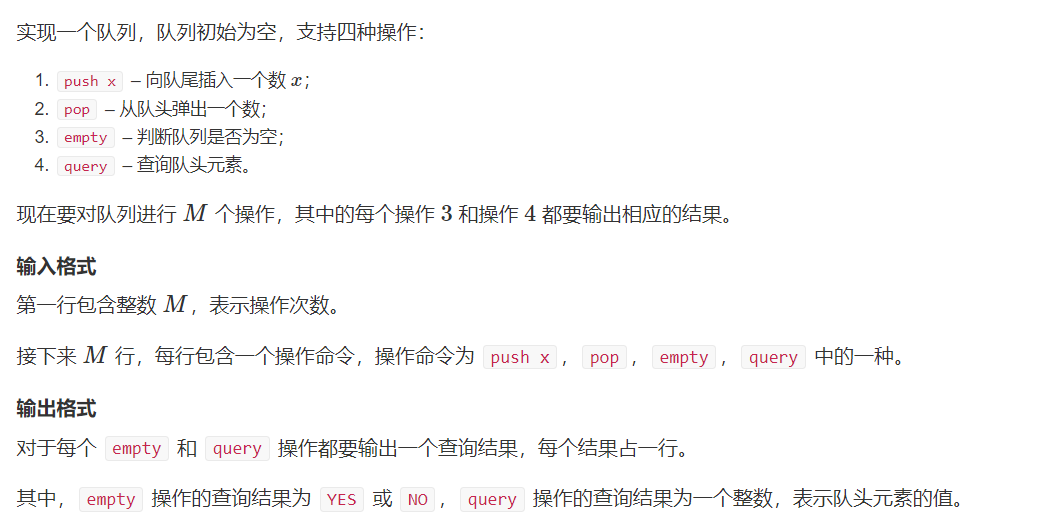
**2.队列：**

**先进先出**

hh(队头) tt（队尾）

运用数组对其模拟：在队尾插入：str[ ++tt]=x 弹出：hh++ if(tt>hh)判断是否为空。

样例：



#include<iostream>

using namespace std;

const int N = 100010;

int str[N];

int tt=-1,hh=0; //队尾和队头指针

int main()

{

int m,x;

scanf("%d",&m);

while(m--)

{

string op;

cin>>op;

if(op=="push") //在队尾插入元素

{

scanf("%d",&x);

str[++tt]=x;

}

else if(op=="pop") //通过队头指针的自增来弹出元素

hh++;

else if(op=="empty") //通过队头是否等于队尾判断是否为空

{

if(tt<hh)

printf("YES\n");

else

printf("NO\n");

}

else

printf("%d\n",str[hh]); //输出队头元素

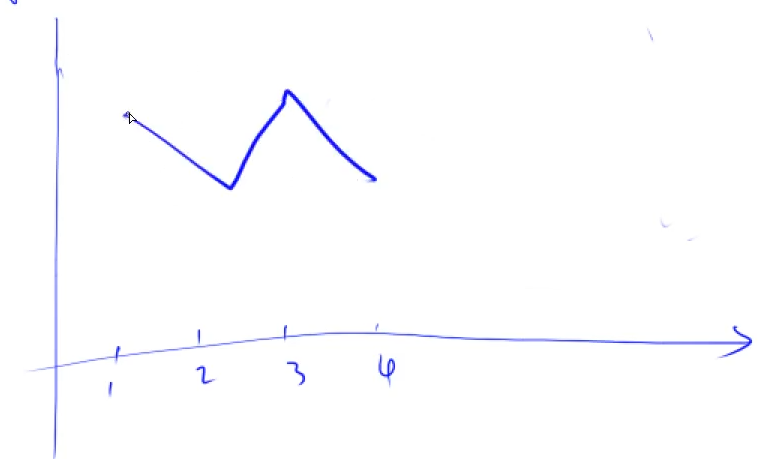
}

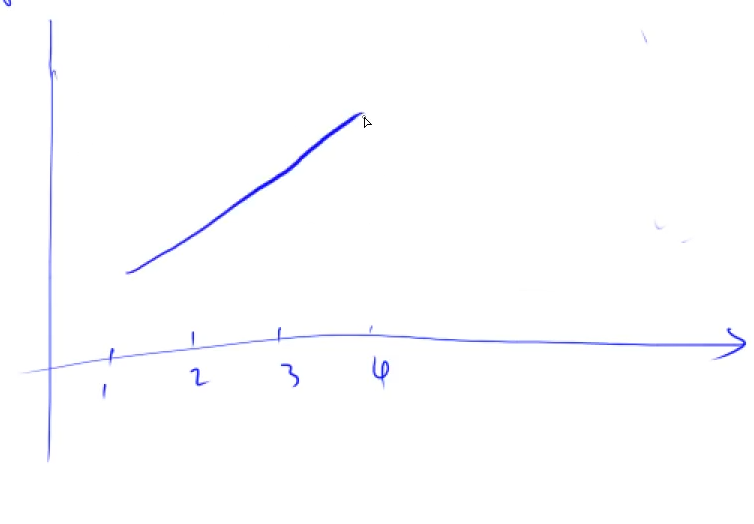
return 0;

}

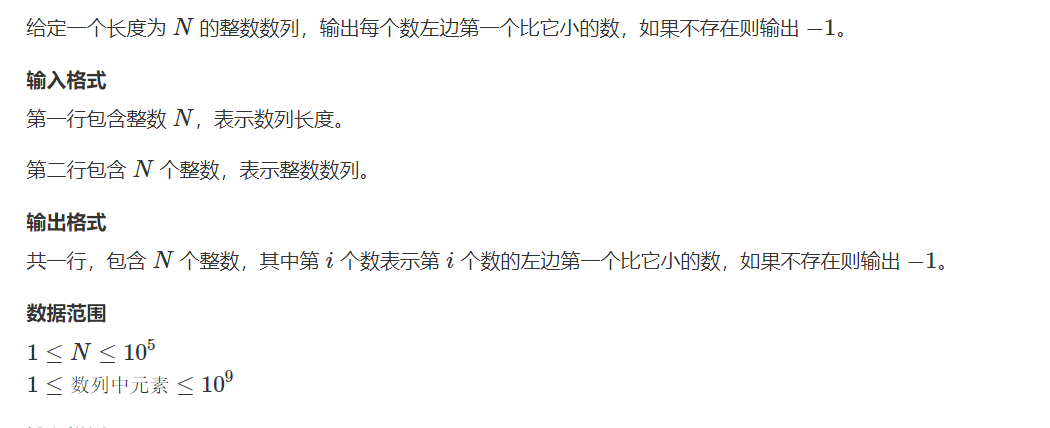
1. **单调栈：**
2. 基本原理：单调栈中是单调递增的数列，通过对一组无序数列的入栈出栈操作，删除某些元素，使其达到单调，来进行操作。

如下图所示：





样例：



如题意所示输出元素左边第一个比它小的数，没有输出-1，那么有以下规律：

如果i<j,a[i]>a[j],那么有在判断时a[i]就是一个无用元素，因为能选a[i]时a[j]必符合条件。

#include<iostream>

using namespace std;

const int N = 100010;

int str[N]={0};

int main()

{

int tt=0,n,x;

scanf("%d",&n);

for(int i=0;i<n;i++)

{

scanf("%d",&x);

while(tt&&str[tt]>=x) tt--; //消除无用元素

if(tt) printf("%d ",str[tt]);

else printf("-1 ");

str[++tt]=x;

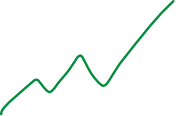
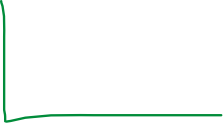
}

return 0;

}

**3.单调队列：**

（1）基本原理：类似单调栈，通过消除无用元素，排除逆序，达到单调的目的。



例题：



#include<iostream>

using namespace std;

const int N = 1000010;

int a[N],q[N];

int main()

{

int n,k;

int tt=-1,hh=0;

scanf("%d%d",&n,&k);

for(int i=0;i<n;i++)

{

scanf("%d",&a[i]);

if(i-k+1>q[hh]) hh++; //判断元素是否还在滑动窗口内

while(tt>=hh && a[q[tt]]>=a[i]) tt--; //除去无用元素，消除逆序的过程

q[++tt]=i;

if(i-k+1>=0) printf("%d ",a[q[hh]]); //输出滑动窗口左端最小值

}

printf("\n");

tt=-1,hh=0;

for(int i=0;i<n;i++)

{

scanf("%d",&a[i]);

if(i-k+1>q[hh]) hh++; //判断元素是否还在滑动窗口内

while(tt>=hh && a[q[tt]]<=a[i]) tt--; //除去无用元素，消除逆序的过程

q[++tt]=i;

if(i-k+1>=0) printf("%d ",a[q[hh]]); //输出滑动窗口左端最大值

}

return 0;

}

**KMP:**

（1）基本原理：字符串：s，模式串：p.

1.暴力做法：

i=0,j=0;

while(i<p.size()&&j<s.size())

{

if(p[i]==s[j])

{

i++;

j++

}

else

{

i=0;

j=j-i+1;

}

}

if(i==p.size()-1)

return i-j;

else

return -1;

2.KMP做法：

假设s：A B C A B C D

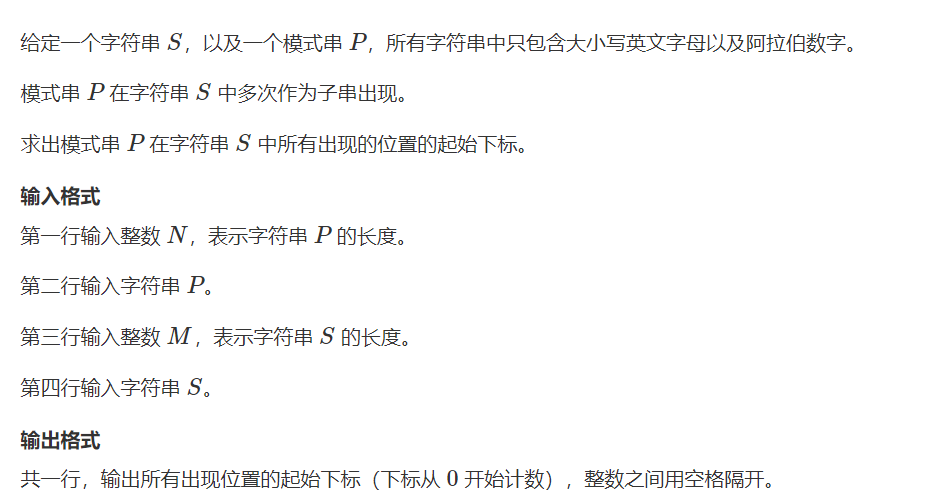
P：A B C A B B

（1）kmp匹配：开始从第一个位置匹配，当匹配第6个位置时，B!=C匹配失败，显然可知我们移动时应该将p字符串的第三个位置，与s字符串的第6个位置进行匹配，即s字符串指针不移动，p字符串的指针移项至第三个位置(因为存在性质p[0~k-1]==p[j-k-j-1]，j为size（）,才有向后移项对应多少位置)，因为匹配时前五项与p都相同，而第二个A出现的位置是4，应将p的1位置处与匹配时第4个位置对应，前面已匹配过相等，所以从p的第三个位置开始，s的第6个位置。

（2）Next数组：在kmp匹配时在第k个位置匹配失败时对应从p数组第几个开始的下标。

（3）如何获取next数组：对其本身进行kmp匹配，即p匹配p。(前缀与后缀相等的数目)

样例：



#include<iostream>

using namespace std;

const int N = 1e5+10,M=1e6+10;

char s[M],p[N];

int nex[N];

int main()

{

int n,m;

cin>> n >> p+1 >> m >> s+1; //下标从1开始

for(int i=2,j=0;i<=n;i++)

{

while(j&&p[i]!=p[j+1]) j=nex[j];

if(p[i]==p[j+1]) j++;

nex[i]=j; //对p自己进行kmp匹配即可得到next数组。

}

for(int i=1,j=0;i<=m;i++)

{

while(j&&s[i]!=p[j+1]) j=nex[j];

if(s[i]==p[j+1]) j++;

if(j==n)

{

printf("%d ",i-n); //输出匹配成功开始的位置

j=nex[j]; //重新开始匹配

}

}

return 0;

**Tire树:**

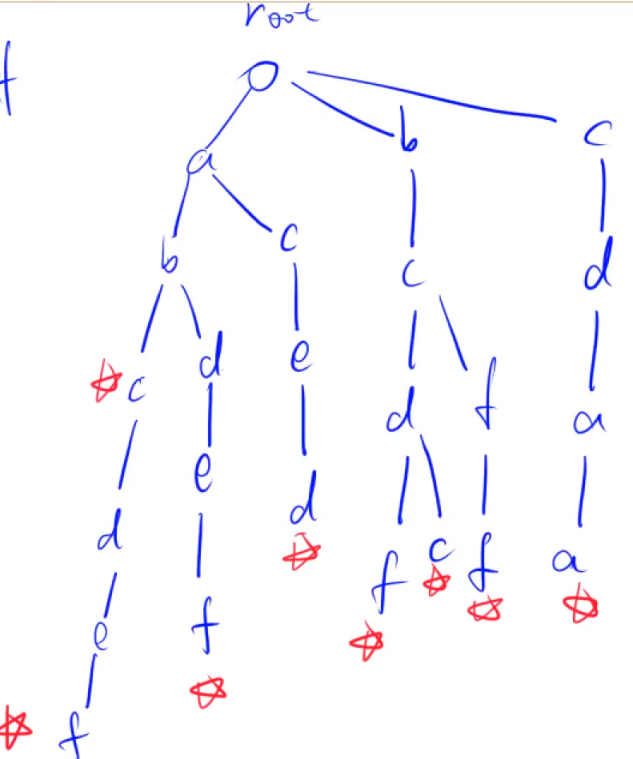
（1）高效地存储和查找字符串集合的数据结构

（2）操作：

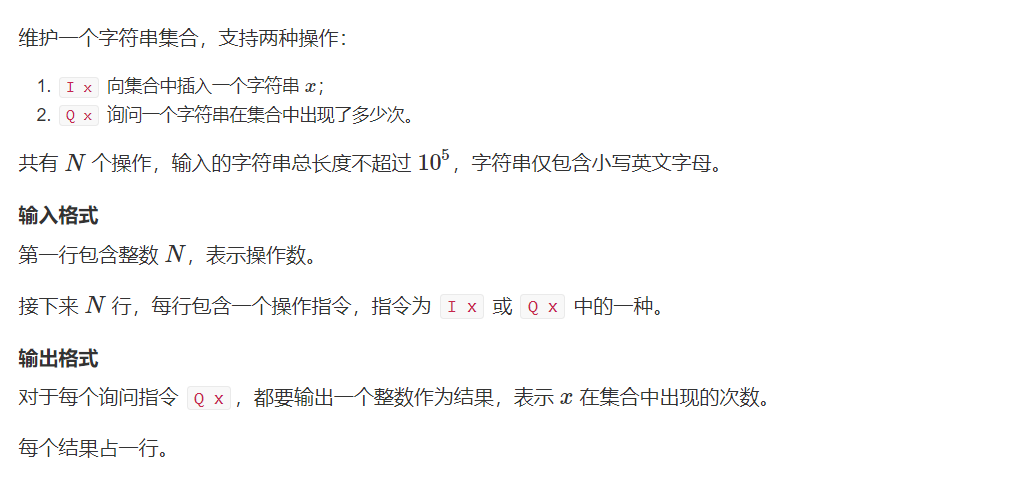
1.存储：

输入：abcdef abdef aced bcdf bcff cdaa abc

如图所示为tire树的存储方式，其中root为根节点，星号代表其中一个字符串的截止。（类似链表）



例题：



#include<iostream>

using namespace std;

const int N = 1e5+10;

int son[N][26],cnt[N],idx=0; //son[x][26]数组存储的是x节点下的所有子节点（son[x][0]表示x节点的第一个子节点），cnt数组存储的一个字符串的最后字母对应的idx（idx是惟一的，表示第几个节点）

void insert(char str[]) //插入一个新字符串

{

int p=0;

for(int i=0;str[i];i++) //p相当于自上而下的层数

{

int u=str[i]-'a';

if(!son[p][u]) son[p][u]=++idx;

p=son[p][u];

}

cnt[p]++; //相同字符串的数量

}

int query(char str[])

{

int p=0;

for(int i=0;str[i];i++)

{

int u=str[i]-'a';

if(!son[p][u]) return 0;

p=son[p][u];

}

return cnt[p];

}

int main()

{

int n;

char op[2],str[N];

scanf("%d",&n);

while(n--)

{

scanf("%s%s",op,str);

if(op[0]=='I')

insert(str);

else

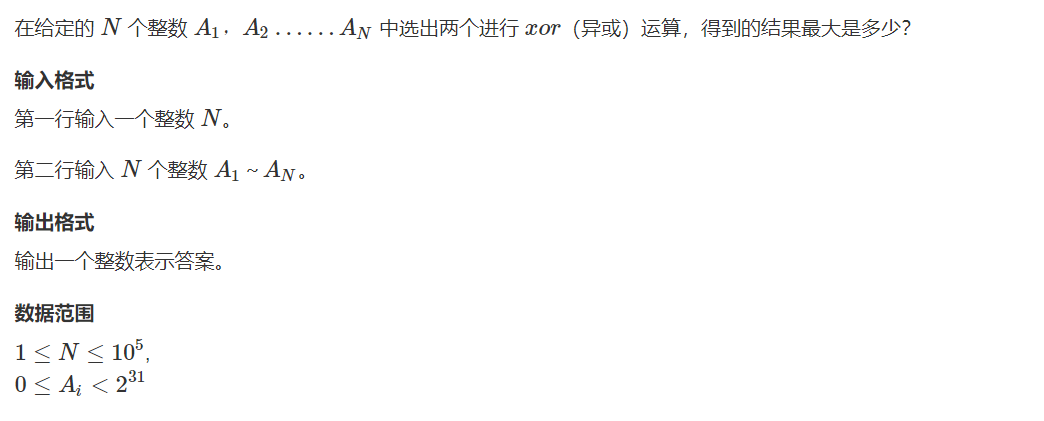
printf("%d\n",query(str));

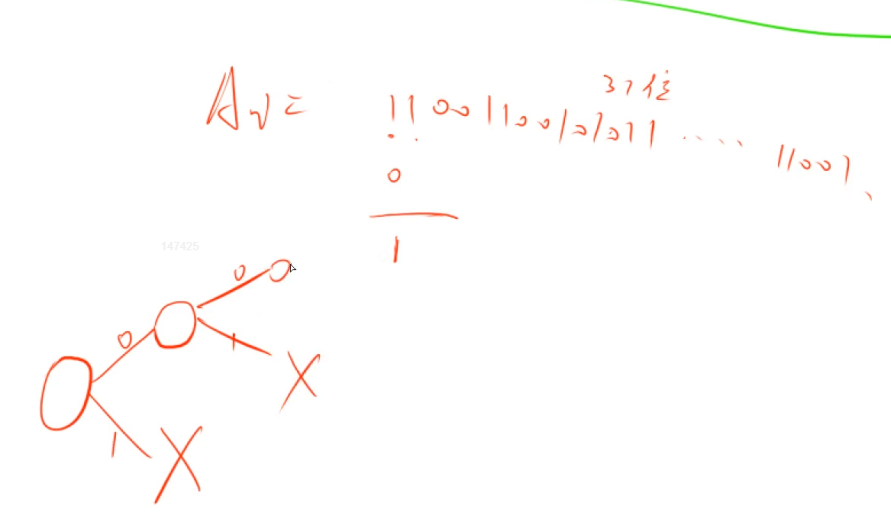
}

return 0;

}

变形：





如图取异或最大值，应从最高位开始排除，第一位为1则最好为0，没有最高位为0的点则只能选1，第二位为1则最好为0同上一直到最低位。

#include<iostream>

#include<algorithm>

using namespace std;

const int N = 1e5+10;

int son[31\*N][2],a[N],idx=0; //前面的31和下面>>所移最多有关系而且，必须要有>>0 的情况,所以i>=0.

void insert(int m )

{

int p=0;

for(int i=31;i>=0;i--)

{

int k=m>>i&1; //取其第i位

if(!son[p][k]) son[p][k]=++idx;

p=son[p][k];

}

}

int query(int m)

{

int p=0;

int res=0;

for(int i=31;i>=0;i--)

{

int k=m>>i&1;

if(son[p][!k])

{

p=son[p][!k];

res=2\*res+1; //做加和

}

else

{

p=son[p][k];

res=2\*res+0;

}

}

return res;

}

int main()

{

int n,res=-1;

scanf("%d",&n);

for(int i=0;i<n;i++)

{

scanf("%d",&a[i]);

insert(a[i]);

}

for(int i=0;i<n;i++)

res=max(res,query(a[i]));

printf("%d",res);

return 0;

}

**并查集:**

1. 操作（近乎O（1））：1.将两个集合合并2.询问两个元素是否在一个集合当中。
2. 基本原理：每个集合用一颗树来表示，树根的编号就是整个集合的编号，每个节点存储它的父节点，p[ x ]表示x的父节点。

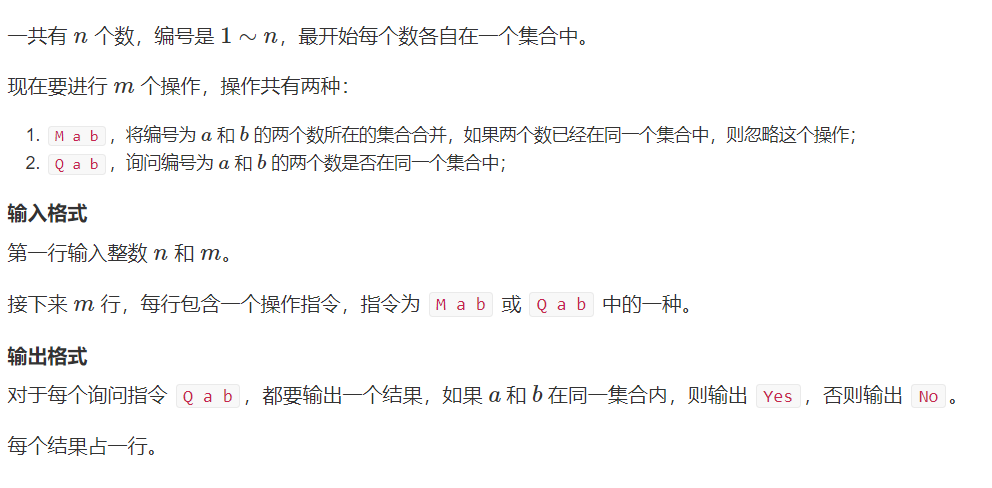
需要解决的问题：

* + - 1. 如何判断树根：if(p[x] == x)
      2. 如何求x的集合编号：while(x!= p[x]) x=p[x];
      3. 如何合并两个集合：px是x的集合编号，py是y的集合编号，p[x] = y.

即将x的根节点插入到y根节点的子节点中。

1. 优化（路径压缩）：在查询过某个节点的根节点后，会将其查找路径上所有点与根节点建立联系，之后再查询时可以直接查到根节点。（时间近乎O(1)）

例题：



#include<iostream>

using namespace std;

const int N = 1e5+10;

int p[N]; //p[x]用来存储x点的父节点

int find(int x) //返回根节点+路径压缩

{

if(p[x]!=x) p[x]=find(p[x]); //当p[x]=x时，即为根节点,并且在找到根节点的过程中，使其所有途经点的父节点的值改为了根节点

return p[x];

}

int main()

{

int n,m;

scanf("%d%d",&n,&m);

for(int i=1;i<=n;i++)

p[i]=i; //初始化最开始每一个数在各自的集合当中

while(m--)

{

char op[2];

int a,b;

scanf("%s%d%d",op,&a,&b);

if(op[0]=='M') p[find(a)] = find(b); //将a集合的根节点的父节点设为b所在集合中的父节点

else

if(find(a)==find(b))

printf("Yes\n");

else

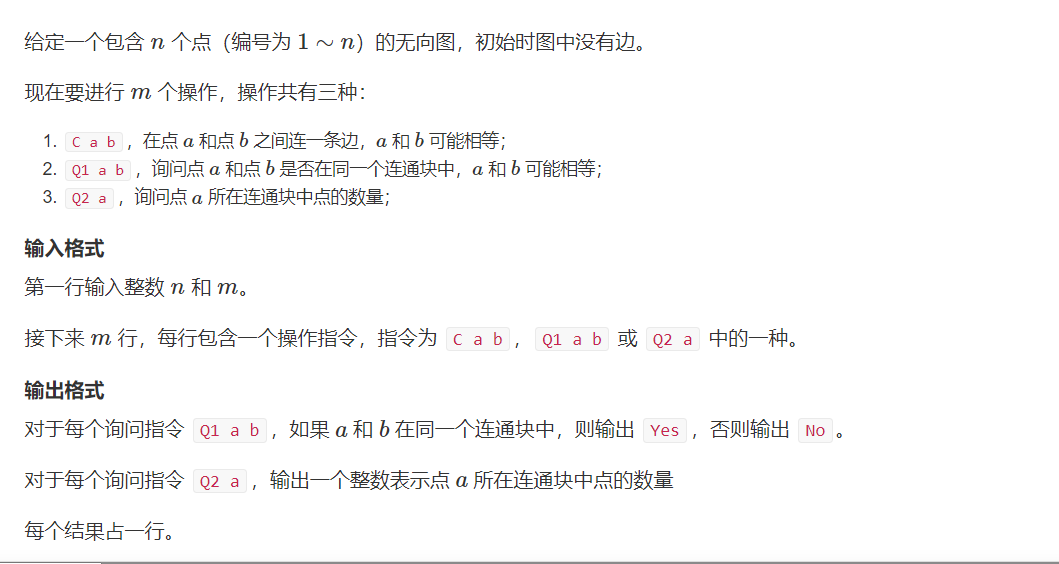
printf("No\n");

}

return 0;

}

变形：



若实现动态存储集合中的点的数目，则可以用一个数组来存储其根节点下所有点的数目，但是仅有根节点的数目数组有意义。

#include<iostream>

using namespace std;

const int N = 1e5+10;

int p[N],siz[N]; //p[x]用来存储x点的父节点,size[x]存储集合内拥有点的个数（仅有根节点有意义）

int find(int x) //返回根节点+路径压缩

{

if(p[x]!=x) p[x]=find(p[x]);

return p[x];

}

int main()

{

int n,m;

scanf("%d%d",&n,&m);

for(int i=1;i<=n;i++)

p[i]=i,siz[i]=1; //初始化最开始每一个数在各自的集合当中

while(m--)

{

char op[5];

int a,b;

scanf("%s",op);

if(op[0]=='C')

{

scanf("%d%d",&a,&b);

if(find(a)==find(b)) continue; //在同一个集合继续循环即可

siz[find(b)]+=siz[find(a)]; //合并后b集合点的数量=之前b集合的点数目+a集合的点数目

p[find(a)] = find(b); //将a集合的根节点的父节点设为b所在集合中的父节点

}

else if(op[1]=='1')

{

scanf("%d%d",&a,&b);

if(find(a)==find(b))

printf("Yes\n");

else

printf("No\n");

}

else

{

scanf("%d",&a);

printf("%d\n",siz[find(a)]);

}

}

return 0;

}

**堆：**

1. 基本原理：是一个完全二叉树；小根堆：根节点小于等于左右子节点(下标从1开始否则子节点和根节点计算公式会出错2\*x=0)。
2. 基本操作：

(1)存储：用一个一维数组存储，x的左儿子：2x，x的右儿子：2x+1.

(2)down(x):x的向下操作（相当于将堆顶换成一个元素x，x与子节点比较向下交换的过程。

(3)up(x):x的向上操作（与down（x）相反）

3.操作：

(1)插入一个数：heap[size++]=x(heap是当前的一维数组，size是数组大小)将插入数放在一维数组的尾部;up(size)对插入数进行向上操作。

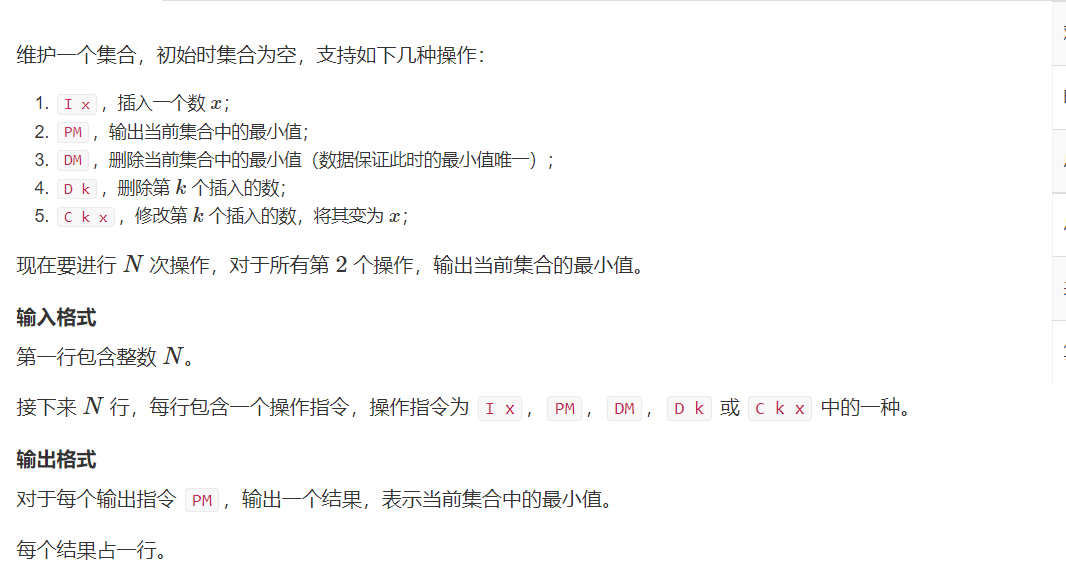
（2）输出堆的最小值：heap[1].

（3）删除最小值：heap[1]=heap[size];size--;down(1);让堆顶的值等于堆底的值（覆盖），然后size--删掉尾部的节点，再down（1）维护小根堆。

(4)删除任意元素：heap[k]=heap[size];size--;down(k);up(k);类似删除最小值，先用尾部节点覆盖掉第k个节点，然后删掉尾结点，再维护堆（覆盖后down和up操作只会执行一个要么比子节点大，要么比根节点小或相等两个都不执行(堆的两侧可能出现尾结点比某个点小)）。

(5)修改任意元素：heap[k]=x;down(k),up(k)。（类似删除只会执行一个）

例题：



#include<iostream>

#include<algorithm>

#include<cstring>

using namespace std;

const int N = 1e5+10;

int h[N],ph[N],hp[N],siz; //ph[k]=j存的是第k个插入的点的下标j，hp[j]=k存的是下标是j的点是第k个插入的

void heap\_swap(int a,int b)

{

swap(ph[hp[a]],ph[hp[b]]); //ph和hp相互映射方便交换,没有hp光凭无法根据下标得知是第几个插入的

swap(hp[a],hp[b]);

swap(h[a],h[b]);

}

void down(int u)

{

int t=u;

if(2\*u<=siz&&h[2\*u]<h[t]) t=u\*2;

if(2\*u+1<=siz&&h[2\*u+1]<h[t]) t=u\*2+1; //判断与子节点大小

if(t!=u)

{

heap\_swap(u,t);

down(t); //h[u]与h[t]交换了位置，现在h[t]存储的是h[u]原来存储的值

}

}

void up(int u)

{

while(u/2&&h[u/2]>h[u])

{

heap\_swap(u,u/2);

u/=2;

}

}

int main()

{

int n,m=0;

scanf("%d",&n);

while(n--)

{

char op[10];

int x,k;

scanf("%s",op);

if(!strcmp(op,"I"))

{

scanf("%d",&x);

siz++;

m++;

ph[m]=siz;

hp[siz]=m;

h[siz]= x;

up(siz);

}

else if(!strcmp(op,"PM")) printf("%d\n",h[1]);

else if(!strcmp(op,"DM"))

{

heap\_swap(1,siz);

siz--;

down(1); //删除操作

}

else if(!strcmp(op,"D"))

{

scanf("%d",&k);

k=ph[k];

heap\_swap(k,siz);

siz--;

down(k),up(k);

}

else

{

scanf("%d%d",&k,&x);

k=ph[k];

h[k]=x;

down(k),up(k);

}

}

return 0;

}