

ACM模板



目录

[插入排序 3](#_Toc482472474)

[归并排序 3](#_Toc482472475)

[逆序对 4](#_Toc482472476)

[二分 6](#_Toc482472477)

[最大子数组 6](#_Toc482472478)

[1000亿以内的素数筛选 8](#_Toc482472479)

[DFS与BFS 11](#_Toc482472480)

[DFS 12](#_Toc482472481)

[BFS 15](#_Toc482472482)

[字符串 20](#_Toc482472483)

[字符串匹配（KMP算法） 20](#_Toc482472484)

[字典树（也称前缀树，Trie） 22](#_Toc482472485)

[01字典树 26](#_Toc482472486)

[LCA（Least Common Ancestors，最近公共祖先） 29](#_Toc482472487)

[Tarjan算法（离线算法） 29](#_Toc482472488)

[树的直径 33](#_Toc482472489)

[二叉树 35](#_Toc482472490)

[二叉树的层次遍历 35](#_Toc482472491)

[二叉树的递归遍历 38](#_Toc482472492)

[线段树 39](#_Toc482472493)

[单点更新 40](#_Toc482472494)

[区间更新 42](#_Toc482472495)

[树状数组 51](#_Toc482472496)

[一维 单点更新 区间查询 51](#_Toc482472497)

[一维 区间更新 单点查询 53](#_Toc482472498)

[一维 区间更新 区间查询 55](#_Toc482472499)

[二维 单点更新 区间查询 57](#_Toc482472500)

[二维 区间更新 单点查询 60](#_Toc482472501)

[二维 区间更新 区间查询 62](#_Toc482472502)

[博弈论 62](#_Toc482472503)

[巴什博弈（Bash Game） 62](#_Toc482472504)

[威佐夫博奕（Wythoff Game） 63](#_Toc482472505)

[尼姆博弈（Nimm Game） 64](#_Toc482472506)

[Fibonacci（斐波那契数列）博弈 67](#_Toc482472507)

[公平组合博弈（Impartial Combinatori Games） 68](#_Toc482472508)

[数论 72](#_Toc482472509)

[唯一分解定理 72](#_Toc482472510)

[斐波那契数列性质 72](#_Toc482472511)

[欧几里德算法（辗转相除法） 73](#_Toc482472512)

[Eratosthenes筛法 73](#_Toc482472513)

[扩展欧几里德算法 74](#_Toc482472514)

[大整数取模 76](#_Toc482472515)

[快速幂取模 76](#_Toc482472516)

[矩阵快速幂 76](#_Toc482472517)

[逆元 80](#_Toc482472518)

[图论 82](#_Toc482472519)

[最短路 82](#_Toc482472520)

[Dijkstra算法 82](#_Toc482472521)

[Floyd算法 83](#_Toc482472522)

[最小生成树 84](#_Toc482472523)

[Prim算法 84](#_Toc482472524)

[Kruskal算法 86](#_Toc482472525)

[动态规划(DP) 88](#_Toc482472526)

[资源型DP(背包) 88](#_Toc482472527)

[01背包 88](#_Toc482472528)

[完全背包 89](#_Toc482472529)

[多重背包 89](#_Toc482472530)

[分组背包 89](#_Toc482472531)

[线性DP 90](#_Toc482472532)

# 插入排序

PS: 两个值相同的元素在排序前后是否有位置变化,如果前后位置变化,则排序算法是不稳定的,否则是稳定的。插入排序是稳定的。STL里面的sort快速排序是不稳定的，但有稳定的stable\_sort。

INSERTION-SORT

升序模板：

for j = 2 to A.length

key = A[j]

i = j-1;

while i>0 and A[i]>key

A [i+1] = A[i]

i = i-1

A[i+1] = key

降序模板：

for j = 2 to A.length

key = A[j]

i = j-1

while i>0 and A[i]<key

A[i+1] = A[i]

i = i-1

A[i+1] = key

# 归并排序

方法：分治法

分解：分解待排序的n个元素的序列成各具n/2个元素的两个子序列。

解决：使用归并排序递归的排序两个子序列。

合并：合并两个已排序的子序列以产生已排序的答案。

伪代码：

MERGE(A,p,q,r)

n1=q-p+1

n2=r-q

let L[1...n1+1] and R[1...n2+1] be new arrays

for i = 1 to n1

L[i] = A[p+i-1]

for j = 1 to n2

R[j] = A[q+j]

L[n1+1] = 无穷大

R[n2+1] = 无穷大

i = 1

j = 1

for k = p to r

if L[i]<=R[j]

A[k]=L[i]

i++

else

A[k]=R[j]

j++

MERGE\_SORT(A,p,r)

if p < r

q = (p+r)/2

MERGE\_SORT(A,p,q)

MERGE\_SORT(A,q+1,r)

MERGE(A,p,q,r)

## 逆序对

可与归并排序挂钩！！

假设A[1..n]是一个有n个不同数的数组。若i < j 且 A[i] > A[j] ，则对偶(i,j)称为A的一个逆序对。

以hdu3743为例（求逆序对的个数）：

代码：

#include <iostream>

#include <algorithm>

#include <cstring>

#include <cstdlib>

#include <typeinfo>

#include <vector>

#include <iomanip>

#include <map>

#include <queue>

#include <set>

#include <stack>

using namespace std;

long long ans,sz[1000005];

void MERGE(int left,int mid,int right)

{

int n1=mid-left+1,n2=right-mid;

int L[n1+1],R[n2+1];

for(int i=0;i<n1;i++) L[i]=sz[left+i];

for(int i=0;i<n2;i++) R[i]=sz[mid+i+1];

L[n1]=1000000000;

R[n2]=1000000000;

int i=0,j=0;

for(int k=left;k<=right;k++)

{

if(L[i]<=R[j])

{

sz[k]=L[i];

i++;

}

else

{

sz[k]=R[j];

j++;

ans+=(n1-i);

}

}

}

void merge\_sort(int left,int right)

{

if(left<right)

{

int mid=(left+right)/2;

merge\_sort(left,mid);

merge\_sort(mid+1,right);

MERGE(left,mid,right);

}

}

int main()

{

int n;

while(scanf("%d",&n)!=EOF)

{

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

ans=0;

merge\_sort(0,n-1);

printf("%lld\n",ans);

}

return 0;

}

# 二分

模板：

int solve(int l,int r)

{

int mid;

while(l<r)

{

mid = (l+r+1)/2; //保证mid是一个比l大的数

if(judge(mid)) l = mid; //最后大的数给l，保证l==r，跳出

else r = mid-1; //否则mid-1给r，保证r==l，跳出

}

return l; //返回l或r均可，因为跳出时l==r

}

# 最大子数组

引进问题：知道股票在一段时间的涨跌情况，求在什么时候买进卖出得到最大收益。

以hdu1231为例题：

代码：

#include <cstdio>

#include <cmath>

#include <ctype.h>

#include <iostream>

#include <algorithm>

#include <cstring>

#include <cstdlib>

#include <typeinfo>

#include <vector>

#include <iomanip>

#include <map>

#include <queue>

#include <set>

#include <stack>

using namespace std;

int sz[10005];

long long ans;

int main()

{

int n,left,right;

while(scanf("%d",&n)&&n)

{

for(int i=1;i<=n;i++) scanf("%d",&sz[i]);

ans=sz[1];

left=right=sz[1];

long long sum=0;

int flag=sz[1];

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

{

if(sum<0)

{

sum=sz[i];

flag=sz[i];

}

else sum+=sz[i];

if(sum>ans)

{

ans=sum;

left=flag;

right=sz[i];

}

}

if(ans<0) printf("0 %d %d\n",sz[1],sz[n]);

else printf("%lld %d %d\n",ans,left,right);

}

return 0;

}

# 1000亿以内的素数筛选

以hdu5901为例：

题意：给一个数n，1<=n<=1e11；求n以内的素数的个数

给出两种代码:

　　　第一种方法Meisell-Lehmer算法只需265ms，空间消耗比后一种方法多。

　　　第二种方法不能运行但是能AC，只需35行，时间比第一种消耗多。

第一种代码:

#include<cstdio>

#include<cmath>

using namespace std;

#define LL long long

const int N = 5e6 + 2;

bool np[N];

int prime[N], pi[N];

int getprime()

{

int cnt = 0;

np[0] = np[1] = true;

pi[0] = pi[1] = 0;

for(int i = 2; i < N; ++i)

{

if(!np[i]) prime[++cnt] = i;

pi[i] = cnt;

for(int j = 1; j <= cnt && i \* prime[j] < N; ++j)

{

np[i \* prime[j]] = true;

if(i % prime[j] == 0) break;

}

}

return cnt;

}

const int M = 7;

const int PM = 2 \* 3 \* 5 \* 7 \* 11 \* 13 \* 17;

int phi[PM + 1][M + 1], sz[M + 1];

void init()

{

getprime();

sz[0] = 1;

for(int i = 0; i <= PM; ++i) phi[i][0] = i;

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

{

sz[i] = prime[i] \* sz[i - 1];

for(int j = 1; j <= PM; ++j) phi[j][i] = phi[j][i - 1] - phi[j / prime[i]][i - 1];

}

}

int sqrt2(LL x)

{

LL r = (LL)sqrt(x - 0.1);

while(r \* r <= x) ++r;

return int(r - 1);

}

int sqrt3(LL x)

{

LL r = (LL)cbrt(x - 0.1);

while(r \* r \* r <= x) ++r;

return int(r - 1);

}

LL getphi(LL x, int s)

{

if(s == 0) return x;

if(s <= M) return phi[x % sz[s]][s] + (x / sz[s]) \* phi[sz[s]][s];

if(x <= prime[s]\*prime[s]) return pi[x] - s + 1;

if(x <= prime[s]\*prime[s]\*prime[s] && x < N)

{

int s2x = pi[sqrt2(x)];

LL ans = pi[x] - (s2x + s - 2) \* (s2x - s + 1) / 2;

for(int i = s + 1; i <= s2x; ++i) ans += pi[x / prime[i]];

return ans;

}

return getphi(x, s - 1) - getphi(x / prime[s], s - 1);

}

LL getpi(LL x)

{

if(x < N) return pi[x];

LL ans = getphi(x, pi[sqrt3(x)]) + pi[sqrt3(x)] - 1;

for(int i = pi[sqrt3(x)] + 1, ed = pi[sqrt2(x)]; i <= ed; ++i) ans -= getpi(x / prime[i]) - i + 1;

return ans;

}

LL lehmer\_pi(LL x)

{

if(x < N) return pi[x];

int a = (int)lehmer\_pi(sqrt2(sqrt2(x)));

int b = (int)lehmer\_pi(sqrt2(x));

int c = (int)lehmer\_pi(sqrt3(x));

LL sum = getphi(x, a) +(LL)(b + a - 2) \* (b - a + 1) / 2;

for (int i = a + 1; i <= b; i++)

{

LL w = x / prime[i];

sum -= lehmer\_pi(w);

if (i > c) continue;

LL lim = lehmer\_pi(sqrt2(w));

for (int j = i; j <= lim; j++) sum -= lehmer\_pi(w / prime[j]) - (j - 1);

}

return sum;

}

int main()

{

init();

LL n;

while(~scanf("%lld",&n))

{

printf("%lld\n",lehmer\_pi(n));

}

return 0;

}

第二种代码：

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const ll maxn=1e11;

const ll maxp=sqrt(maxn)+10;

ll f[maxp],g[maxp];

ll solve(ll n)

{

ll i,j,m;

for(m=1;m\*m<=n;m++)

f[m]=n/m-1;

for(i=1;i<=m;i++)

g[i]=i-1;

for(i=2;i<=m;i++)

{

if(g[i]==g[i-1]) continue;

for(j=1;j<=min(m-1,n/i/i);j++)

{

if(i\*j<m)

f[j]-=f[i\*j]-g[i-1];

else

f[j]-=g[n/i/j]-g[i-1];

}

for(j=m;j>=i\*i;j--)

g[j]-=g[j/i]-g[i-1];

}

return f[1];

}

int main()

{

ll n;

while(scanf("%lld",&n)!=EOF)

printf("%lld\n",solve(n));

return 0;

}

# 搜索

## DFS与BFS

BFS：这是一种基于队列这种数据结构的搜索方式，它的特点是由每一个状态可以扩展出许多状态，然后再以此扩展，直到找到目标状态或者队列中头尾指针相遇，即队列中所有状态都已处理完毕。

DFS：基于递归的搜索方式，它的特点是由一个状态拓展一个状态，然后不停拓展，直到找到目标或者无法继续拓展结束一个状态的递归。

优缺点：

BFS:对于解决最短或最少问题特别有效，而且寻找深度小，但缺点是内存耗费量大（需要开大量的数组单元用来存储状态）。

DFS：对于解决遍历和求所有问题有效，对于问题搜索深度小的时候处理速度迅速，然而在深度很大的情况下效率不高

总结：不管是BFS还是DFS，它们虽然好用，但由于时间和空间的局限性，以至于它们只能解决数据量小的问题。

各种搜索题目归类：

坐标类型搜索 ：这种类型的搜索题目通常来说简单的比较简单，复杂的通常在边界的处理和情况的讨论方面会比较复杂，分析这类问题，我们首先要抓住题目的意思，看具体是怎么建立坐标系（特别重要），然后仔细分析到搜索的每一个阶段是如何通过条件转移到下一个阶段的。确定每一次递归（对于DFS）的回溯和深入条件，对于BFS，要注意每一次入队的条件同时注意判重。要牢牢把握目标状态是一个什么状态，在什么时候结束搜索。还有，DFS过程的参数如何设定，是带参数还是不带参数，带的话各个参数一定要保证能完全的表示一个状态，不会出现一个状态对应多个参数，而这一点对于BFS来说就稍简单些，只需要多设置些变量就可以了。

经典题目：细胞（NDK1435）、Tyvj：乳草的入侵、武士风度的牛

数值类型搜索：（虽然我也不知道该怎么叫，就起这个名字吧），这种类型的搜索就需要仔细分析分析了，一般来说采用DFS，而且它的终止条件一般都是很明显的，难就难在对于过程的把握，过程的把握类似于坐标类型的搜索（判重、深入、枚举），注意这种类型的搜索通常还要用到剪枝优化，对于那些明显不符合要求的特殊状态我们一定要在之前就去掉它，否则它会像滚雪球一样越滚越大，浪费我们的时间。

经典题目：Tyvj：派对；售货员的难题，以及各种有关题目

### DFS

坐标搜索：

以hdu1241为例：

代码：

#include <cstdio>

#include <cstdlib>

#include <iostream>

#include <cstring>

using namespace std;

int visit[105][105];

char str[105][105];

int n,m;

void dfs(int x,int y)

{

if(x<0||x>=n||y<0||y>=m) return;

if(visit[x][y]==1||str[x][y]!='@') return;

visit[x][y]=1;

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

{

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

{

if(i!=0||j!=0) dfs(x+i,y+j);

}

}

}

int main()

{

int ans;

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

{

ans=0;

for(int i=0;i<n;i++) scanf("%s",str[i]);

memset(visit,0,sizeof(visit));

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

{

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

{

if(str[i][j]=='@'&&visit[i][j]==0)

{

dfs(i,j);

ans++;

}

}

}

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

}

return 0;

}

数值搜索：

以hdu1175为例：

代码：

#include <cstdio>

#include <cstdlib>

#include <iostream>

#include <cstring>

using namespace std;

int sz[1005][1005],visit[1005][1005],x1,x2,y1,y2,n,m;

bool flag;

void dfs(int x,int y,int num,int dir)

{

if(x<1||x>n||y<1||y>m) return;

if(num>2) return;

if(x==x2&&y==y2){flag=true;return;}

if(flag) return;

if((x!=x1||y!=y1)&&sz[x][y]!=0) return;

if(num==2&&!(dir==1&&x2<x&&y2==y||dir==2&&x2>x&&y2==y||dir==3&&x2==x&&y2<y||dir==4&&x2==x&&y2>y)) return; //剪枝优化

visit[x][y]=1;

if(dir==1&&!visit[x-1][y])

{

dfs(x-1,y,num,1);

dfs(x-1,y,num+1,3);

dfs(x-1,y,num+1,4);

}

if(dir==2&&!visit[x+1][y])

{

dfs(x+1,y,num,2);

dfs(x+1,y,num+1,3);

dfs(x+1,y,num+1,4);

}

if(dir==3&&!visit[x][y-1])

{

dfs(x,y-1,num+1,1);

dfs(x,y-1,num+1,2);

dfs(x,y-1,num,3);

}

if(dir==4&&!visit[x][y+1])

{

dfs(x,y+1,num+1,1);

dfs(x,y+1,num+1,2);

dfs(x,y+1,num,4);

}

visit[x][y]=0;

}

int main()

{

int t;

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

{

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

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

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

scanf("%d",&t);

while(t--)

{

memset(visit,0,sizeof(visit));

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

if(sz[x1][y1]!=sz[x2][y2]||sz[x2][y2]==0) flag=false;

else

{

flag=false;

dfs(x1,y1,0,1);

dfs(x1,y1,0,2);

dfs(x1,y1,0,3);

dfs(x1,y1,0,4);

}

if(flag==true) printf("YES\n");

else printf("NO\n");

}

}

return 0;

}

### BFS

坐标搜索：

以hdu1372为例：

代码：

#include <cstdio>

#include <cstdlib>

#include <iostream>

#include <cstring>

#include <queue>

using namespace std;

struct point{

int x,y,num;

};

int visit[10][10],x1,x2,y1,y2;

const int dir[8][2]={1,2,1,-2,-1,2,-1,-2,2,1,2,-1,-2,1,-2,-1};

int bfs()

{

point p,tmp;

queue<point>q;

p.x=x1;

p.y=y1;

p.num=0;

memset(visit,0,sizeof(visit));

visit[p.x][p.y]=1;

q.push(p);

while(!q.empty())

{

p = q.front(); q.pop();

if(p.x==x2&&p.y==y2) return p.num;

for(int i=0;i<8;i++)

{

tmp.x=p.x+dir[i][0];

tmp.y=p.y+dir[i][1];

if(tmp.x<1||tmp.x>8||tmp.y<1||tmp.y>8) continue;

if(visit[tmp.x][tmp.y]==1) continue;

tmp.num = p.num+1;

q.push(tmp);

}

}

}

int main()

{

string a,b;

while(cin>>a>>b)

{

x1=a[1]-'0';

y1=a[0]-'a'+1;

x2=b[1]-'0';

y2=b[0]-'a'+1;

cout<<"To get from "<<a<<" to "<<b<<" takes "<<bfs()<<" knight moves.\n";

}

return 0;

}

数值搜索：

以hdu1495为例：

代码：

#include <cstdio>

#include <cstring>

#include <queue>

#include <iostream>

using namespace std;

struct Node{

int x,y,all,num;

};

int vis[105][105],n,m,s;

void bfs()

{

Node p,tmp;

queue<Node>q;

p.x=0,p.y=0,p.all=s,p.num=0;

memset(vis,0,sizeof(vis));

vis[p.x][p.y]=1;

q.push(p);

while(!q.empty())

{

p=q.front(),q.pop();

if(p.y==p.all&&p.y==s/2)

{

printf("%d\n",p.num);

return;

}

if(p.all+p.x>m)

{

tmp.x=m,tmp.y=p.y,tmp.all=p.all+p.x-m,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

else

{

tmp.x=p.x+p.all,tmp.y=p.y,tmp.all=0,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

if(p.all+p.y>n)

{

tmp.x=p.x,tmp.y=n,tmp.all=p.all+p.y-n,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

else

{

tmp.x=p.x,tmp.y=p.y+p.all,tmp.all=0,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

if(p.x+p.y>m)

{

tmp.x=m,tmp.y=p.x+p.y-m,tmp.all=p.all,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

else

{

tmp.x=p.x+p.y,tmp.y=0,tmp.all=p.all,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

tmp.x=p.x,tmp.y=0,tmp.all=p.y+p.all,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

if(p.x+p.y>n)

{

tmp.x=p.x+p.y-n,tmp.y=n,tmp.all=p.all,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

else

{

tmp.x=0,tmp.y=p.x+p.y,tmp.all=p.all,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

tmp.x=0,tmp.y=p.y,tmp.all=p.x+p.all,tmp.num=p.num+1;

if(!vis[tmp.x][tmp.y])

{

vis[tmp.x][tmp.y]=1;

q.push(tmp);

}

}

printf("NO\n");

}

int main()

{

while(scanf("%d%d%d",&s,&n,&m)&&(s||n||m))

{

if(s&1)

{

printf("NO\n");

continue;

}

if(m>n) swap(n,m);

bfs();

}

return 0;

}

## BFS+状态压缩

所谓状态压缩是指状态用一种压缩的形式存储。一般采用一个二进制数表示状态，有时也用三进制或四进制数等。用二进制数表示状态最大的好处就是在判断状态时可以采用位运算，这样能极大提高算法效率。

以hust1579为例

题意：给你T组输入，每组输入第一行是n，m，k。（1 <= n, m <= 30，0<=k<=10）代表n行m列的地图和k个隧道。（每个隧道有个对应的钥匙）接下来是地图，之后一行代表初始的钥匙个数及其钥匙的种类。接下来k行代表从A开始的每个隧道中钥匙的个数及其种类。地图从X出发，求到达Y的最少步数，不能到达就输出Unreachable。

题解：因为有最多有10把钥匙，所以我们可以用10个二进制位来保存当前位置各个钥匙的状态，有为1，没有0。把10把钥匙的状态压缩成一个2进制数。就是BFS+状态压缩题，具体看代码注释。与hdu1429不同的是加入隧道，可以从隧道的一头到达另一头。这题有点坑的是（和之前HZAU的现场赛的最后一道签到题类似，都是windows和Linux换行的问题）地图那里的输入不能用%c，必须%s，不然会RE。

代码：

#include <bits/stdc++.h>

**using** **namespace** std;

char mp[35][35];

//标记数组包括位置和钥匙的状态

bool vis[35][35][1025]; //把10个钥匙的状态压缩成一个数，最大2^10-1。

int tunnel[10][2][2]; //记录隧道的两个坐标

int cave[10]; //记录每个隧道的钥匙状态

int dis[4][2]={-1,0,0,-1,1,0,0,1};

int n,m,k;

struct node{

int x,y,key,step; //key表示钥匙状态，step表示步数

}st;

bool check(node tmp) //判断是否能走

{

**if**(tmp.x<0 || tmp.x>=n || tmp.y<0 || tmp.y>=m) **return** **false**;

**else** **if**(mp[tmp.x][tmp.y]=='#' || vis[tmp.x][tmp.y][tmp.key]) **return** **false**;

**else** **return** **true**;

}

void bfs()

{

memset(vis,**false**,**sizeof**(vis));

node tmp,next;

queue<node>q;

q.push(st);

vis[st.x][st.y][st.key]=**true**;

**while**(!q.empty())

{

tmp=q.front(),q.pop();

**for**(int i=0;i<4;i++)

{

next=tmp;

next.x += dis[i][0];

next.y += dis[i][1];

next.step++;

**if**(check(next))

{

**if**(mp[next.x][next.y]=='Y')

{

printf("%d\n",next.step);

**return**;

}

**else**

{

vis[next.x][next.y][next.key]=**true**;

q.push(next);

}

}

}

int id = mp[tmp.x][tmp.y]-'A';

**if**(id>=0 && id<k) //判断当前点是否为隧道

{

int flag=(tmp.key>>id)&1; //是否有当前隧道的钥匙

**if**(flag)

{

next=tmp;

next.step++;

next.key = next.key|cave[id]; //根据隧道中的钥匙更新钥匙状态

//到隧道另一头

**if**(next.x==tunnel[id][0][0]&&next.y == tunnel[id][0][1])

{

next.x=tunnel[id][1][0];

next.y=tunnel[id][1][1];

}

**else**

{

next.x=tunnel[id][0][0];

next.y=tunnel[id][0][1];

}

**if**(!vis[next.x][next.y][next.key])

{

vis[next.x][next.y][next.key]=**true**;

q.push(next);

}

}

}

}

printf("Unreachable\n");

}

int main()

{

int T;

scanf("%d",&T);

**for**(int ca=1;ca<=T;ca++)

{

memset(tunnel,-1,**sizeof**(tunnel));

memset(cave,0,**sizeof**(cave));

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

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

{

scanf("%s",mp[i]);

**for**(int j=0;j<m;j++)

{

**if**(mp[i][j]=='X') //初始st

{

st.x=i;

st.y=j;

st.key=st.step=0;

mp[i][j]='.';

}

**else** **if**(mp[i][j]-'A'>=0 && mp[i][j]-'A'<k) //记录隧道两头坐标

{

int id=mp[i][j]-'A';

**if**(tunnel[id][0][0]==-1)

{

tunnel[id][0][0]=i;

tunnel[id][0][1]=j;

}

**else**

{

tunnel[id][1][0]=i;

tunnel[id][1][1]=j;

}

}

}

}

int num;

char tmp;

scanf("%d",&num);

**while**(num--)

{

getchar();

scanf("%c",&tmp);

st.key = st.key|(1<<(tmp-'A')); //更新初始钥匙状态

}

**for**(int i=0;i<k;i++)

{

scanf("%d",&num);

**while**(num--)

{

getchar();

scanf("%c",&tmp);

cave[i] = cave[i]|(1<<(tmp-'A')); //每个隧道中钥匙状态

}

}

printf("Case #%d: ",ca);

bfs();

}

**return** 0;

}

## 迭代加深搜索（ID-DFS）

迭代加深搜索，实质上是限定下界的深度优先搜索。即首先允许深度优先搜索K层，若没有发现可行解，再将K+1后重复以上步骤搜索（即深度优先搜索k+1层），直到搜索到可行解。

在迭代加深搜索的算法中，连续的深度优先搜索被引入，每一个深度约束逐次加1，直到搜索到目标为止。可以看出，很多情况会重复搜索。但是它的好处在于：

1. 空间开销小。每个深度下实际上是一个有深度限制的深度优先搜索。而DFS的空间消耗小是众所周知的。
2. 利于深度剪枝。
3. 时间效率不低。虽然重复搜索，但是整体并不比广度有限搜索慢很多。

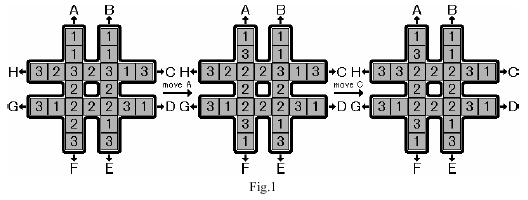
我们可以看出，迭代加深搜索算法就是仿广度优先搜索的深度优先搜索。既能满足深度优先搜索的线性存储要求，又能保证发现一个最小深度的目标结点。

从实际应用来看，迭代加深搜索的效果比较好，并不比广度优先搜索慢很多，但是空间复杂度却与深度优先搜索相同，比广度优先搜索小很多。

使用搜索算法的时候，选择正确的搜索方式很重要。当有一类问题需要做广度优先搜索，但却没有足够的空间，而时间却很充裕，碰到这类问题，我们可以选择迭代加深搜索算法。

一般来说，如果目标结点离根结点远，需要遍历整棵树，可以考虑使用深度优先搜索；如果目标离根结点近，或求最小步数，则考虑广度优先搜索或迭代加深搜索；若广度优先搜索存在空间不够的问题，则考虑使用迭代加深搜索。

以hdu1667为例：

、

代码：

#include <bits/stdc++.h>

**using** **namespace** std;

const int center[] = {6,7,8,11,12,15,16,17}; //中心8个点坐标

const int reverseop[] = {5,4,7,6,1,0,3,2,-1}; //每种操作的逆操作

const int index[8][7] = { //从A-H操作变动的下标

{ 0,2,6,11,15,20,22 }, //A

{ 1,3,8,12,17,21,23 }, //B

{ 10,9,8,7,6,5,4 }, //C

{ 19,18,17,16,15,14,13 }, //D

{ 23,21,17,12,8,3,1 }, //E

{ 22,20,15,11,6,2,0 }, //F

{ 13,14,15,16,17,18,19 }, //G

{ 4,5,6,7,8,9,10 }, //H

};

int mp[24];

char finop[105]; //搜到目标的操作序列

bool flag; //是否搜到目标

int get\_h()

{

int cnt[3]={0};

int num=-1;

**for**(int i=0;i<8;i++)

{

cnt[mp[center[i]]-1]++;

num = max(num,cnt[mp[center[i]]-1]);

}

**return** 8-num;

}

void option(int op)

{

int tmp=mp[index[op][0]];

**for**(int i=0;i<6;i++)

{

mp[index[op][i]] = mp[index[op][i+1]];

}

mp[index[op][6]]=tmp;

}

void dfs(int depth,int lastop,int maxdepth) //depth当前深度 lastop上一次的操作 maxdepth每次迭代加深搜索的最大深度

{

**if**(flag) **return**;

**if**(depth>maxdepth || depth+get\_h()>maxdepth) **return**; //可行性剪枝

**if**(get\_h()==0) //中心相同了

{

flag=**true**;

finop[depth]='\0';

printf("%s\n%d\n",finop,mp[center[0]]);

**return**;

}

**for**(int nextop=0;nextop<8;nextop++) //从A开始搜索

{

**if**(nextop!=reverseop[lastop]) //与上一次操作不互逆

{

option(nextop);

finop[depth]=nextop+'A';

dfs(depth+1,nextop,maxdepth);

option(reverseop[nextop]); //回溯还原

}

}

}

int main()

{

**while**(1)

{

**for**(int i=0;i<24;i++)

{

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

**if**(mp[i]==0) **return** 0;

}

**if**(get\_h()==0)

{

printf("No moves needed\n%d\n", mp[center[0]]);

**continue**;

}

flag=**false**;

**for**(int depth=1;!flag;depth++) //迭代加深

{

dfs(0,8,depth);

}

}

**return** 0;

}

# 字符串

## 字符串匹配（KMP算法）

KMP算法又称“看毛片”算法，是一种效率非常高的字符串匹配算法。

kmp算法完成的任务是：给定两个字符串O和f，长度分别为n和m，判断f是否在O中出现，如果出现则返回出现的位置。常规方法是遍历a的每一个位置，然后从该位置开始和b进行匹配，但是这种方法的复杂度是O(nm)。kmp算法通过一个O(m)的预处理，使匹配的复杂度降为O(n+m)。

思想：我们首先用一个图来描述kmp算法的思想。在字符串O中寻找f，当匹配到位置i时两个字符串不相等，这时我们需要将字符串f向前移动。常规方法是每次向前移动一位，但是它没有考虑前i-1位已经比较过这个事实，所以效率不高。事实上，如果我们提前计算某些信息，就有可能一次前移多位。假设我们根据已经获得的信息知道可以前移k位，我们分析移位前后的f有什么特点。我们可以得到如下的结论：

* A段字符串是f的一个前缀。
* B段字符串是f的一个后缀。
* A段字符串和B段字符串相等。

所以前移k位之后，可以继续比较位置i的前提是f的前i-1个位置满足：**长度为i-k-1的前缀A和后缀B相同**。只有这样，我们才可以前移k位后从新的位置继续比较。



所以kmp算法的核心即是计算字符串f每一个位置之前的字符串的前缀和后缀公共部分的最大长度（不包括字符串本身，否则最大长度始终是字符串本身）。获得f每一个位置的最大公共长度之后，就可以利用该最大公共长度快速和字符串O比较。当每次比较到两个字符串的字符不同时，我们就可以根据最大公共长度将字符串f向前移动(已匹配长度-最大公共长度)位，接着继续比较下一个位置。事实上，字符串f的前移只是概念上的前移，只要我们在比较的时候从最大公共长度之后比较f和O即可达到字符串f前移的目的。



以hdu1711为例：

题意：给出两个数组，求出第二个数组在第一个数组中最先匹配的位置。

代码：

#include <cstdio>

using namespace std;

const int maxm=1e4+5,maxn=1e6+5;

int n,m,sn[maxn],sm[maxm],next[maxm];

void set\_next()

{

next[0]=next[1]=0;

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

{

int tmp=next[i]; //已知next[i]求next[i+1]

while(tmp && sm[i]!=sm[tmp]) tmp = next[tmp];

if(sm[i]==sm[tmp]) tmp++;

next[i+1]=tmp;

}

}

int main()

{

int t,num;

bool flag;

scanf("%d",&t);

while(t--)

{

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

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

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

set\_next();

flag=false;

num=0;

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

{

while(num && sn[i]!=sm[num]) num=next[num];

if(sn[i]==sm[num]) num++;

if(num==m)

{

flag=true;

printf("%d\n",i-m+2);

break;

}

}

if(!flag) printf("-1\n");

}

return 0;

}

## 字典树（也称前缀树，Trie）

Trie树，又称单词查找树、字典树，是一种树形结构，是一种哈希树的变种，是一种用于快速检索的多叉树结构。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。

Trie高效的核心思想是空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提率的目的。

它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。

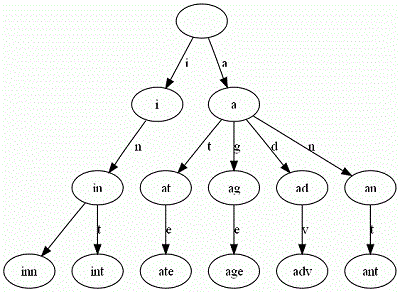
Trie树也有它的缺点,Trie树的内存消耗非常大。

三个基本特性：

1）根节点不包含字符，除根节点外每一个节点都只包含一个字符。

2）从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。

3）每个节点的所有子节点包含的字符都不相同。



应用：

1. 字符串检索，词频统计，搜索引擎的热门查询
2. 字符串最长公共前缀

Trie树利用多个字符串的公共前缀来节省存储空间，反之，当我们把大量字符串存储到一棵trie树上时，我们可以快速得到某些字符串的公共前缀。举例：给出N 个小写英文字母串，以及Q 个询问，即询问某两个串的最长公共前缀的长度是多少.

解决方案：

首先对所有的串建立其对应的字母树。此时发现，对于两个串的最长公共前缀的长度即它们所在结点的公共祖先个数，于是，问题就转化为了离线 （Offline）的最近公共祖先（Least Common Ancestor，简称LCA）问题。

1. 排序：Trie树是一棵多叉树，只要先序遍历整棵树，输出相应的字符串便是按字典序排序的结果。

以hdu1251为例：

题意：老师交给他很多单词(只有小写字母组成,不会有重复的单词出现),现在老师要他统计出以某个字符串为前缀的单词数量(单词本身也是自己的前缀)。

如何判断空行：用gets()读入。读入的回车符会自动转换为NULL。所以循环读入，每次检测读入进来的字符串的第一个字符是否为NULL即可。

代码有两种，一种用链表实现，一种用数组实现，数组做法在时间和空间上都优于链表。

代码（链表）：（好像过不了，内存超了，网上代码说能过）（把编译器从g++改为c++，头文件改一下能过）

#include <bits/stdc++.h>

using namespace std;

struct Trie{

Trie \*next[26];

int num;

Trie() //初始化

{

for(int i=0;i<26;i++) next[i]=NULL;

num=0;

}

}root;

int idx(char s)

{

return s-'a';

}

void Trie\_insert(char word[])

{

Trie \*p=&root;

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

{

if(p->next[idx(word[i])]==NULL) p->next[idx(word[i])] = new Trie;

p = p->next[idx(word[i])];

p->num++;

}

}

int Trie\_search(char word[])

{

Trie \*p=&root;

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

{

if(p->next[idx(word[i])]==NULL) return 0;

p = p->next[idx(word[i])];

}

return p->num;

}

int main()

{

char word[11];

while(cin.getline(word,11))

{

if(strlen(word)==0) break;

Trie\_insert(word);

}

while(scanf("%s",word)!=EOF)

{

printf("%d\n",Trie\_search(word));

}

return 0;

}

代码（数组）：

#include <bits/stdc++.h>

using namespace std;

const int maxn=1e6+5;

int trie[maxn][26],num[maxn],cnt=1;

void Trie\_insert(char word[])

{

int n=0;

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

{

if(trie[n][word[i]-'a']==0) trie[n][word[i]-'a']=cnt++;

n = trie[n][word[i]-'a'];

num[n]++;

}

}

int Trie\_search(char word[])

{

int n=0;

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

{

if(trie[n][word[i]-'a']==0) return 0;

n = trie[n][word[i]-'a'];

}

return num[n];

}

int main()

{

char word[11];

while(gets(word))

{

if(word[0]==NULL) break;

Trie\_insert(word);

}

while(scanf("%s",word)!=EOF)

{

printf("%d\n",Trie\_search(word));

}

return 0;

}

## 01字典树

其实01字典树就可以看成是把一个数的二进制字符化后插入到一颗一般的字典树中。

一般01字典树用来解决区间异或和之类的问题。

异或的性质：

1. 交换律

2. 结合律，即(a^b)^c = a^(b^c)）

3. 自反性，即x^x=0

4. x^0=x

其中运用最多的就是自反性。

有上述性质，对于区间异或和要知道如下性质：

XOR[l,r] = XOR[1,l-1] ^ XOR[1,r]

所以对于区间异或和之类的题目，比如求区间异或和的最大值，我们可以将r前的所有前缀异或和加入到一个01字典树中，然后查询一下[1,r]就可以得到以r为右边界的最大异或和区间。

在查询最大异或值时我们用贪心的策略，比如我们在字典树中查询10101的最大异或值。

我们从最高位即第5位开始查（我省略掉前面的0位），由于第5位是1（对于其它的任意数，我们设为idx），之后看字典树中有没有第5为是1^1（idx^1）的数，如果有就进入0（idx^1）的节点（贪心思想，即首先保证该位异或后值为1，使异或值尽可能大），没有就进入1（idx）节点，然后从高位到低位依次这样即可。

以hzau1206为例：

题意：有T组样例，每组样例给n个数，a1…an（n<=1000000）。求这n个数中最大异或和值的区间。有多个答案区间按字典序输出。

题解：把1-n的所有前缀异或和插入01字典树，然后按我上面说的区间异或的性质扫一遍就可以了。这题就还要注意一下区间按字典序输出，之前没看到这个，一直WA。具体看代码注释。

#include <bits/stdc++.h>

**using** **namespace** std;

const int maxn = 1e6+5; //数的最大个数

const int INF = 0x3f3f3f3f;

int ch[maxn\*32][2]; //int型是32位，所以32\*maxn

int idx[maxn\*32],cnt;

int ans,l,r; //ans保存最大异或和的值，l，r为其区间

void init() //01字典树的初始化

{

memset(ch,0,**sizeof**(ch));

memset(idx,INF,**sizeof**(idx));

cnt=1;

ans=0;

l=r=0;

}

void trie\_insert(int id,int x) //插入序号为id的前缀异或和x

{

int num=0;

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

{

int tmp = (x>>i)&1; //从第32位开始，以下和字典树类似

**if**(!ch[num][tmp]) ch[num][tmp] = cnt++;

num = ch[num][tmp];

}

//保证同一异或和的序号最小，为之后字典树输出区间准备

//比如插入1，5后再插入3，5最后idx[num]=1

**if**(id<idx[num]) idx[num] = id;

}

void trie\_query(int id,int x)

{

int num=0,m=0;

//m是以id为右边界，数x的异或的最大值

//num是与x异或后为m的值的节点序号

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

{

int tmp = (x>>i)&1;

**if**(ch[num][tmp^1])

{

num = ch[num][tmp^1];

m += (1<<i);

}

**else**

{

num = ch[num][tmp];

m += (0<<i);

}

}

**if**(m>ans) //如果m大于当前最大异或和的值，直接更新区间和最大异或和值

{

l = idx[num];

r = id;

ans = m;

}

**if**(m==ans) //如果等于，区间按字典序更新

{

**if**(idx[num]<l)

{

l = idx[num];

r = id;

}

**if**(idx[num]==l)

{

**if**(id<r)

{

l = idx[num];

r = id;

}

}

}

}

int main()

{

//freopen("std.in","r",stdin);

//freopen("out.txt","w",stdout);

int T,n,num,tmp;

scanf("%d",&T);

**for**(int ca=1;ca<=T;ca++)

{

init();

trie\_insert(0,0);

num=0;

scanf("%d",&n);

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

{

scanf("%d",&tmp);

{

num ^= tmp; //num为i为前缀的异或和

trie\_insert(i,num);

trie\_query(i,num); //查询以i为右边界的最大异或和值的区间

}

}

printf("Case #%d:\n%d %d\n",ca,l+1,r);

}

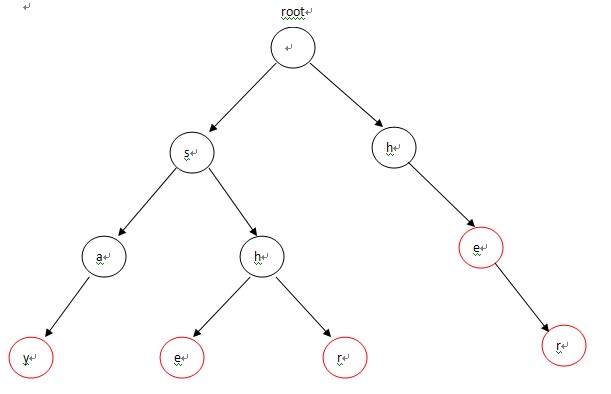
**return** 0;

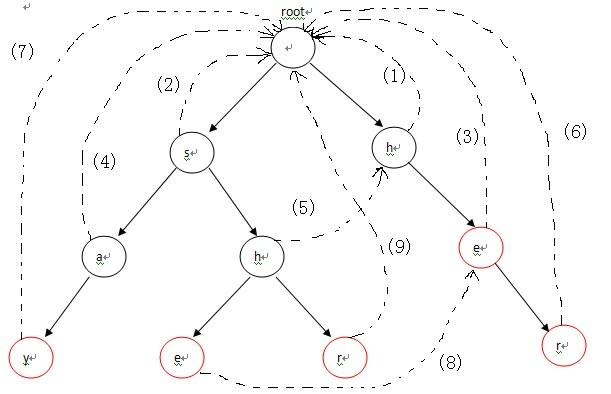
}

## AC自动机

AC自动机（Aho-Corasick automation），该算法在1975年产生于贝尔实验室，是著名的多模匹配算法之一。   
一个常见的例子就是给出n个单词，再给出一段包含m个字符的文章，让你找出有多少个单词在文章里出现过。

我们以5个单词say，she，shr，he，her和字符串yasherhs为例，问多少个单词在字符串中出现过。

构造Trie树如图   


然后我们详细讲解一下AC自动机最重要的部分，就是fail失败指针的求解。我们对着图看我的讲解。   


我们是用BFS来构造fail指针，并且运用的是KMP的思想。   
最开始先让root入队。队列｛root｝   
第1次循环时处理与root相连的字符即h和s。因为第一个字符不匹配需要重新匹配，所以h和s的fail指针都指向root（root是Trie入口，没有实际含义）即图中（1）和（2）。此时队列｛h，s｝   
第二次循环时处理与h相连的字符即e。当当前字符e失配时应该将其fail指针指向当前字符串（he）的最长后缀和Trie树中前缀的匹配（就是相当于指向其父节点h的fail指针指向的节点，看其有没有当前节点e的匹配，有的话就指向它，没有的话可以一直向上找直到fail指针指向root）。这里h的fail指针指向root，而root没有e的匹配，所以把e的fail指针指向root。此时队列｛s，e｝   
第三次循环时处理与s相连的字符a和h。其中a与第二次循环的e情况相同，将其fail指针指向root。h其父节点s的fail指针指向root，而root有h的匹配，所以把h的fail指针指向与root相连的h节点。   
之后情况依此类推。

以hdu2222为例

代码

#include <bits/stdc++.h>

**using** **namespace** std;

const int maxn = 1e6+5;

int t,n;

char word[55],str[maxn];

struct node{

node \*next[26];

node \*fail;

int num;

node()

{

**for**(int i=0;i<26;i++) next[i] = **NULL**;

fail = **NULL**;

num = 0;

}

};

node \*root;

void init()

{

root = **new** node;

}

void ac\_insert()

{

node \*p = root;

**for**(int i=0;word[i];i++)

{

int id = word[i]-'a';

**if**(p->next[id]==**NULL**) p->next[id] = **new** node;

p = p->next[id];

}

p->num++;

}

void ac\_setFail()

{

node \*p,\*tmp;

queue<node\*> q;

q.push(root);

**while**(!q.empty())

{

p = q.front(),q.pop();

**for**(int i=0;i<26;i++)

{

**if**(p->next[i]==**NULL**) **continue**;

**if**(p==root)

{

p->next[i]->fail = root;

q.push(p->next[i]);

}

**else**

{

tmp = p->fail;

**while**(tmp)

{

**if**(tmp->next[i])

{

p->next[i]->fail = tmp->next[i];

**break**;

}

tmp = tmp->fail;

}

**if**(tmp==**NULL**) p->next[i]->fail = root;

q.push(p->next[i]);

}

}

}

}

int ac\_automation()

{

int ans=0;

node \*p = root;

node \*tmp;

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

{

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

**while**(p->next[id]==**NULL** && p!=root) p = p->fail;

**if**(p->next[id]!=**NULL**) p = p->next[id];

tmp = p;

**while**(tmp!=root)

{

**if**(tmp->num>0)

{

ans += tmp->num;

tmp->num = 0;

}

**else** tmp = tmp->fail;

}

}

**return** ans;

}

int main()

{

scanf("%d",&t);

**while**(t--)

{

init();

scanf("%d",&n);

**while**(n--)

{

scanf("%s",word);

ac\_insert();

}

ac\_setFail();

scanf("%s",str);

printf("%d\n",ac\_automation());

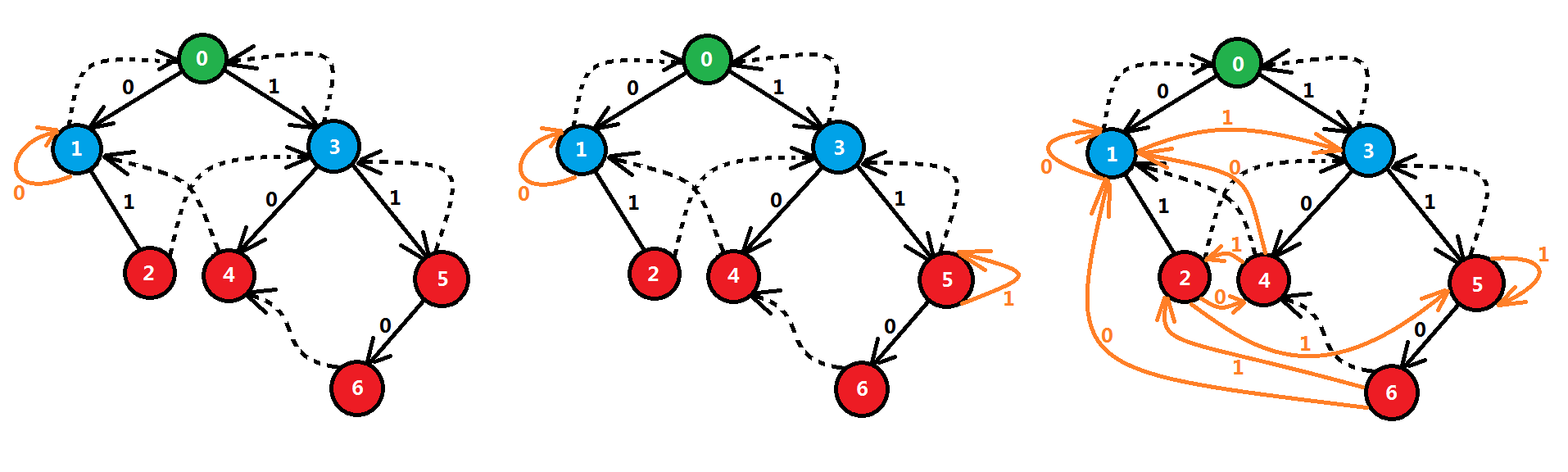
}

**return** 0;

}

## Trie图

Trie图相当于在AC自动机上优化了一下（感觉像是AC自动机的进阶版本），使匹配的速度更快。



（Trie图的构建同样是用BFS实现）   
为了方便描述，我们先把所有trie树上的结点进行编号，编号顺序为结点的插入顺序，根结点编号为0。

最开始根节点入队。   
第一次循环，从根节点开始访问所有字符集指针。因为根节点的所有字符集指针都存在，所以直接入队，所以1，3入队。此时队列为｛1，3｝   
第二次循环，从1号节点开始访问所有字符集指针。此时1号节点的’0’字符的指针为空即不存在，那么我们就将其指向1号节点的fail指针指向的节点（即根节点）的’0’字符指针指向的节点（就是1号节点）（有点绕，仔细体会一下）。1号节点的’1’字符指针存在，所以把2号指针入队，此时队列为｛3，2｝   
之后一直这样BFS即可。（上图1号节点的’1’字符指针的指向好像错了，应该就是指向2号节点的）

以hihoCoder1036为例

代码

#include <bits/stdc++.h>

**using** **namespace** std;

const int maxn = 1e6+5;

char word[maxn],str[maxn];

int n;

struct node{

node \*next[26];

node \*fail;

int num;

node()

{

**for**(int i=0;i<26;i++) next[i] = **NULL**;

fail = **NULL**;

num = 0;

}

}\*root;

void init()

{

root = **new** node();

}

void ac\_insert()

{

node \*p = root;

**for**(int i=0;word[i];i++)

{

int id = word[i]-'a';

**if**(p->next[id]==**NULL**) p->next[id] = **new** node;

p = p->next[id];

}

p->num++;

}

void ac\_setTrieGraph()

{

node \*p,\*tmp;

queue<node\*> q;

q.push(root);

**while**(!q.empty())

{

p = q.front(),q.pop();

**for**(int i=0;i<26;i++)

{

**if**(p->next[i]==**NULL**)

{

**if**(p==root) p->next[i] = root;

**else** p->next[i] = p->fail->next[i];

}

**else**

{

**if**(p==root)

{

p->next[i]->fail = root;

q.push(p->next[i]);

}

**else**

{

tmp = p->fail;

**while**(tmp)

{

**if**(tmp->next[i])

{

p->next[i]->fail = tmp->next[i];

**break**;

}

tmp = tmp->fail;

}

**if**(tmp==**NULL**) p->next[i]->fail = root;

q.push(p->next[i]);

}

}

}

}

}

bool ac\_automation()

{

node \*p = root;

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

{

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

p = p->next[id];

**if**(p->num>0) **return** **true**;

}

**return** **false**;

}

int main()

{

scanf("%d",&n);

init();

**while**(n--)

{

scanf("%s",word);

ac\_insert();

}

ac\_setTrieGraph();

scanf("%s",str);

**if**(ac\_automation()) printf("YES\n");

**else** printf("NO\n");

**return** 0;

}

# LCA（Least Common Ancestors，最近公共祖先）

## Tarjan算法（离线算法）

思想：从根节点开始形成一颗深搜树，从任意节点u出发，遍历其一枝树枝，这样在回溯时u的其中一个子树枝已经遍历，这时再把u放入合并集合中，这样u节点和所有之前遍历的u节点的子树枝节点的lca就是u；u和还未遍历的兄弟节点及其子树中的节点的lca就是u的父亲节点。

以hdu2586为例：

题意：给定一棵树，每条边都有一定的权值，m次询问，每次询问某两点间的距离。

题解：可以用LCA来解。首先找到u, v 两点的lca，然后计算一下距离值就可以了。这里的计算方法是，记下根结点到任意一点的距离dis[]，这样ans = dis[u] + dis[v] - 2 \* dis[lca(v, v)]。

两种代码风格：1.使用不定长数组vector 2.使用结构体

代码（1）：

#include<bits/stdc++.h>

using namespace std;

const int maxn=40005;

bool vis[maxn];

int dis[maxn],ans[205],root[maxn],n;

vector<int>v[maxn],w[maxn],num[maxn],quest[maxn];

int find\_root(int a)

{

return a==root[a] ? a : root[a]=find\_root(root[a]);

}

void munion(int a,int b)

{

a = find\_root(a);

b = find\_root(b);

if(a==b) return;

root[b]=a;

}

void init()

{

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

{

v[i].clear();

w[i].clear();

num[i].clear();

quest[i].clear();

vis[i]=false;

dis[i]=0;

root[i]=i;

}

}

void Tarjan(int cnt,int value)

{

vis[cnt]=true;

dis[cnt]=value;

for(int i=0;i<v[cnt].size();i++)

{

int tmp=v[cnt][i];

if(vis[tmp]) continue;

Tarjan(tmp,value+w[cnt][i]);

munion(cnt,tmp);

}

for(int i=0;i<quest[cnt].size();i++)

{

int tmp=quest[cnt][i];

if(!vis[tmp]) continue;

ans[num[cnt][i]]=dis[cnt]+dis[tmp]-2\*dis[find\_root(tmp)];

}

}

int main()

{

int T,a,b,m,k;

scanf("%d",&T);

while(T--)

{

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

init();

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

{

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

v[a].push\_back(b);

w[a].push\_back(k);

v[b].push\_back(a);

w[b].push\_back(k);

}

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

{

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

quest[a].push\_back(b);

quest[b].push\_back(a);

num[a].push\_back(i);

num[b].push\_back(i);

}

Tarjan(1,0);

for(int i=1;i<=m;i++) printf("%d\n",ans[i]);

}

return 0;

}

代码（2）：

#include <bits/stdc++.h>

using namespace std;

const int N=40005,M=205;

int vis[N],root[N],dis[N],ans[M][3],n,m,num1,num2;

struct node{

int v,d;

node \*next;

};

node \*link1[N],\*link2[N],edg1[N\*2],edg2[N\*2];

void add(int a,int b,int c,node \*link[],node edg[],int &num)

{

edg[num].v=a;

edg[num].d=c;

edg[num].next=link[b];

link[b]=edg+num++;

edg[num].v=b;

edg[num].d=c;

edg[num].next=link[a];

link[a]=edg+num++;

}

int find\_root(int a)

{

return a==root[a] ? a : root[a]=find\_root(root[a]);

}

void munion(int a,int b)

{

a = find\_root(a);

b = find\_root(b);

if(a==b) return;

root[b]=a;

}

void init()

{

num1=num2=0;

memset(vis,0,sizeof(vis));

memset(dis,0,sizeof(dis));

memset(link1,0,sizeof(link1));

memset(link2,0,sizeof(link2));

}

void Tarjan(int x)

{

vis[x]=1;

root[x]=x;

for(node \*p=link1[x];p;p = p->next)

{

if(!vis[p->v])

{

dis[p->v]=dis[x]+p->d;

Tarjan(p->v);

munion(x,p->v);

}

}

for(node \*p=link2[x];p;p=p->next)

{

if(vis[p->v])

{

ans[p->d][2]=find\_root(p->v);

}

}

}

int main()

{

int T,a,b,c;

scanf("%d",&T);

while(T--)

{

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

init();

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

{

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

add(a,b,c,link1,edg1,num1);

}

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

{

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

add(a,b,i,link2,edg2,num2);

ans[i][0]=a;ans[i][1]=b;

}

Tarjan(1);

for(int i=1;i<=m;i++) printf("%d\n",dis[ans[i][0]]+dis[ans[i][1]]-2\*dis[ans[i][2]]);

}

return 0;

}

# 树的直径

思想：树的直径的一个性质：距某个点最远的叶子节点一定是树的某一条直径的端点。这样就可以首先任取一个点，bfs求出离其最远的点，在用同样的方法求出离这个叶子节点最远的点，此时两点间的距离就是树的直径。

以poj1985为例：

题意：有n个农场，这n个农场有一些边连着，然后要你找出两个点，使得这一对点的路径长度最大，输出这个最大的长度。

代码：

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <iostream>

#include <queue>

using namespace std;

struct Edge{

int v,w,next;

}edge[100000];

int n,m,num,head[50000],diameter[50000];

void add(int u,int v,int w)

{

edge[num].v=u;

edge[num].w=w;

edge[num].next=head[v];

head[v]=num++;

edge[num].v=v;

edge[num].w=w;

edge[num].next=head[u];

head[u]=num++;

}

int bfs(int s,int flag)

{

int v,t,tmp=0,tmpi=s;

queue<int>q;

memset(diameter,-1,sizeof(diameter));

diameter[s]=0;

q.push(s);

while(!q.empty())

{

t=q.front();q.pop();

for(int i=head[t];i!=-1;i=edge[i].next)

{

v=edge[i].v;

if(diameter[v]==-1)

{

diameter[v]=diameter[t]+edge[i].w;

if(diameter[v]>tmp)

{

tmp=diameter[v];

tmpi=v;

}

q.push(v);

}

}

}

return flag ? tmpi : tmp;

}

int main()

{

int a,b,w,ans;

char c;

while(~scanf("%d%d",&n,&m))

{

memset(head,-1,sizeof(head));

num=0;

while(m--)

{

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

add(a,b,w);

}

ans=bfs(1,1);

ans=bfs(ans,0);

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

}

return 0;

}

# 二叉树

## 二叉树的层次遍历

用结构体加指针表示二叉树，用BFS遍历。

以UVA122为例（从上到下从左到右输出各节点值）：

代码：

#include<cstdio>

#include<cstdlib>

#include<cstring>

#include<vector>

#include<queue>

using namespace std;

const int maxn=256+5;

struct Node{

bool have\_value;

int v;

Node \*left,\*right;

Node():have\_value(false),left(NULL),right(NULL){}

};

Node \*root;

Node\* newnode() {return new Node();}

bool failed;

void addnode(int v,char \*s)

{

int n=strlen(s);

Node \*u = root;

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

{

if(s[i]=='L')

{

if(u->left==NULL) u->left=newnode();

u = u->left;

}

else

{

if(u->right==NULL) u->right=newnode();

u = u->right;

}

if(u->have\_value) failed=true;

u->v = v;

u->have\_value=true;

}

}

void remove\_tree(Node \*u)

{

if(u==NULL) return;

remove\_tree(u->left);

remove\_tree(u->right);

delete u;

}

char s[maxn];

bool read\_input()

{

failed=false;

remove\_tree(root);

root = newnode();

for(;;)

{

if(scanf("%s",s)!=1) return false;

if(!strcmp(s,"()")) break;

int v;

sscanf(&s[1],"%d",&v);

addnode(v,strchr(s,',')+1);

}

return true;

}

bool bfs(vector<int>& ans)

{

queue<Node\*> q;

ans.clear();

q.push(root);

while(!q.empty())

{

Node \*u=q.front();q.pop();

if(!u->have\_value) return false;

ans.push\_back(u->v);

if(u->left!=NULL) q.push(u->left);

if(u->right!=NULL) q.push(u->right);

}

return true;

}

int main()

{

vector<int> ans;

while(read\_input())

{

if(!bfs(ans)) failed=true;

if(failed) printf("not complete\n");

else

{

for(int i=0;i<ans.size();i++)

{

if(i==ans.size()-1) printf("%d\n",ans[i]);

else printf("%d ",ans[i]);

}

}

}

return 0;

}

## 二叉树的递归遍历

二叉树有三种DFS遍历：先序遍历(PreOrder)，中序遍历(InOrder)和后序遍历(PostOrder)。

以UVA548为例:

代码：

#include <bits/stdc++.h>

using namespace std;

const int maxn=10000+10;

int inorder[maxn],postorder[maxn],lch[maxn],rch[maxn];

int n;

bool read\_list(int \*a)

{

string line;

if(!getline(cin,line)) return false;

stringstream ss(line);

n=0;

int x;

while(ss >> x) a[n++]=x;

return n>0;

}

int build(int L1,int R1,int L2,int R2)

{

if(L1>R1) return 0;

int root=postorder[R2];

int p=L1;

while(inorder[p]!=root) p++;

int cnt=p-L1;

lch[root] = build(L1,p-1,L2,L2+cnt-1);

rch[root] = build(p+1,R1,L2+cnt,R2);

return root;

}

int best,best\_sum;

void dfs(int u,int sum)

{

sum+=u;

if(!lch[u]&&!rch[u])

{

if(sum<best\_sum || (sum==best\_sum&&u<best))

{

best = u;

best\_sum = sum;

}

}

if(lch[u]) dfs(lch[u],sum);

if(rch[u]) dfs(rch[u],sum);

}

int main()

{

while(read\_list(inorder))

{

read\_list(postorder);

build(0,n-1,0,n-1);

best\_sum=1000000000;

dfs(postorder[n-1],0);

cout<<best<<endl;

}

return 0;

}

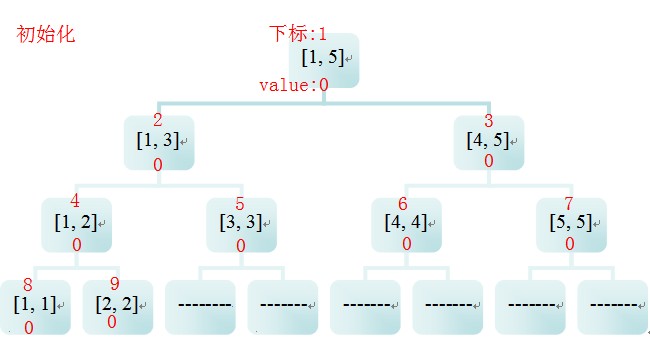
# 线段树

是一种数据结构，没有统一名称，线段树（segment tree）是最主要的叫法，其它包括区间树（interval tree）、范围树（range tree）。

## 单点更新

动态范围的最值问题。

假定根结点是长度为2^h的区间，第i层有2^i个结点，每个结点对应一个长度为2^(h-i)的区间。最大层编号为h，所以结点总数为1+2+4+……+2^h=2^(h+1)-1；所以一般线段树开结构体时需要两倍空间。当整个区间长度不是2的整数幂时，叶子不全在同一层，但树的最大层编号和结点总数仍满足上述结论。 对于线段树中的每一个下标为i非叶子节点[a,b]，它的左儿子下标为2\*i表示的区间为[a,(a+b)/2]，右儿子下标为2\*i+1表示的区间为[(a+b)/2+1,b]。因此线段树是平衡二叉树，最后的子节点数目为N，即整个线段区间的长度。

[](http://img.my.csdn.net/uploads/201208/06/1344267759_4413.jpg)

以hdu 1754 为例：

题意：

本题目包含多组测试，请处理到文件结束。  
在每个测试的第一行，有两个正整数 N 和 M ( 0<N<=200000,0<M<5000 )，分别代表学生的数目和操作的数目。  
学生ID编号分别从1编到N。  
第二行包含N个整数，代表这N个学生的初始成绩，其中第i个数代表ID为i的学生的成绩。  
接下来有M行。每一行有一个字符 C (只取'Q'或'U') ，和两个正整数A，B。  
当C为'Q'的时候，表示这是一条询问操作，它询问ID从A到B(包括A,B)的学生当中，成绩最高的是多少。  
当C为'U'的时候，表示这是一条更新操作，要求把ID为A的学生的成绩更改为B。

线段树单点更新裸题

代码：

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

const int MAX=1<<19; //200000大约2^18

const int maxn=200005;

struct segment\_tree{

int left,right,value;

}tree[MAX]; //区间[left,right]的最大值value

int vis[maxn],ans; //区间长度为1时vis[i]表示其结构体的下标值

void build\_segment\_tree(int i,int l,int r) //建树

{

tree[i].left=l;

tree[i].right=r;

tree[i].value=0;

if(l==r)

{

vis[l]=i;

return;

}

build\_segment\_tree(i\*2,l,(r+l)/2);

build\_segment\_tree(i\*2+1,(r+l)/2+1,r);

}

void update\_segment\_tree(int i) //单点更新

{

if(i==1) return;

int p=i/2;

tree[p].value=max(tree[p\*2].value,tree[p\*2+1].value);

update\_segment\_tree(p);

}

void query\_segment\_tree(int i,int l,int r) //询问

{

if(tree[i].left==l&&tree[i].right==r)

{

ans = max(tree[i].value,ans);

return;

}

if(l<=tree[i\*2].right)

{

if(r<=tree[i\*2].right) query\_segment\_tree(i\*2,l,r);

else

{

query\_segment\_tree(i\*2,l,tree[i\*2].right);

query\_segment\_tree(i\*2+1,tree[i\*2+1].left,r);

}

}

else query\_segment\_tree(i\*2+1,l,r);

}

int main()

{

int n,m,a,b;

char op;

build\_segment\_tree(1,1,200000);

while(~scanf("%d%d",&n,&m))

{

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

{

scanf("%d",&tree[vis[i]].value);

update\_segment\_tree(vis[i]);

}

while(m--)

{

getchar(); //去掉缓存区的换行符

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

switch(op)

{

case 'U':

tree[vis[a]].value=b;

update\_segment\_tree(vis[a]);

break;

case 'Q':

ans=-1;

query\_segment\_tree(1,a,b);

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

break;

}

}

}

return 0;

}

## 区间更新

区间更新是指更新某个区间内的叶子节点的值，因为涉及到的叶子节点不止一个，而叶子节点会影响其相应的非叶子节点，那么回溯需要更新的非叶子节点也会有很多，如果一次性更新完，操作的时间复杂度肯定不是O(lgn)，例如当我们要更新区间[0,3]内的叶子节点时，需要更新出了叶子节点3,9外的所有其他节点。为此引入了线段树中的延迟标记概念，这也是线段树的精华所在。

延迟标记：每个节点新增加一个标记，记录这个节点是否进行了某种修改(这种修改操作会影响其子节点)，对于任意区间的修改，我们先按照区间查询的方式将其划分成线段树中的节点，然后修改这些节点的信息，并给这些节点标记上代表这种修改操作的标记。在修改和查询的时候，如果我们到了一个节点p，并且决定考虑其子节点，那么我们就要看节点p是否被标记，如果有，就要按照标记修改其子节点的信息，并且给子节点都标上相同的标记，同时消掉节点p的标记。

模板以poj3468为例：

题意： 给你N个数，Q个操作，操作有两种，‘Q a b ’是询问a~b这段数的和，‘C a b c’是把a~b这段数都加上c。

代码：

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

const int MAX=1<<19;

const int maxn=1e5+5;

struct segment\_tree{

long long left,right,add,sum;

}tree[MAX];

long long value[maxn];

void build\_segment\_tree(long long i,long long l,long long r)

{

tree[i].left=l;

tree[i].right=r;

tree[i].add=0;

if(l==r)

{

tree[i].sum=value[l];

return;

}

build\_segment\_tree(i\*2,l,(l+r)/2);

build\_segment\_tree(i\*2+1,(l+r)/2+1,r);

tree[i].sum=tree[i\*2].sum+tree[i\*2+1].sum;

}

void update\_segment\_tree(long long i,long long l,long long r,long long x)

{

tree[i].sum+=(r-l+1)\*x;

if(tree[i].left==l&&tree[i].right==r)

{

tree[i].add+=x;

return;

}

if(tree[i].add)

{

tree[i\*2].add+=tree[i].add;

tree[i\*2+1].add+=tree[i].add;

tree[i\*2].sum+=(tree[i\*2].right-tree[i\*2].left+1)\*tree[i].add;

tree[i\*2+1].sum+=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*tree[i].add;

tree[i].add=0;

}

i<<=1;

if(l<=tree[i].right)

{

if(r<=tree[i].right) update\_segment\_tree(i,l,r,x);

else

{

update\_segment\_tree(i,l,tree[i].right,x);

update\_segment\_tree(i+1,tree[i+1].left,r,x);

}

}

else update\_segment\_tree(i+1,l,r,x);

}

long long query\_segment\_tree(long long i,long long l,long long r)

{

if(tree[i].left==l&&tree[i].right==r)

{

return tree[i].sum;

}

if(tree[i].add)

{

tree[i\*2].add+=tree[i].add;

tree[i\*2+1].add+=tree[i].add;

tree[i\*2].sum+=(tree[i\*2].right-tree[i\*2].left+1)\*tree[i].add;

tree[i\*2+1].sum+=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*tree[i].add;

tree[i].add=0;

}

i<<=1;

if(l<=tree[i].right)

{

if(r<=tree[i].right) return query\_segment\_tree(i,l,r);

else

{

return query\_segment\_tree(i,l,tree[i].right)+query\_segment\_tree(i+1,tree[i+1].left,r);

}

}

else return query\_segment\_tree(i+1,l,r);

}

int main()

{

long long n,q,a,b,c,i;

char op;

scanf("%lld%lld",&n,&q);

for(i=1;i<=n;i++) scanf("%lld",&value[i]);

build\_segment\_tree(1,1,n);

while(q--)

{

getchar();

scanf("%c%lld%lld",&op,&a,&b);

switch(op)

{

case 'C':

scanf("%lld",&c);

update\_segment\_tree(1,a,b,c);

break;

case 'Q':

printf("%lld\n",query\_segment\_tree(1,a,b));

break;

}

}

return 0;

}

另难题hdu4578

题意：n个数（初始为0）m个操作，操作类型有4中，操作1把区间的每个数+a，操作2把区间的每个数\*a.，操作3把区间的每个数=a，操作4，查询区间每个数p次方的和(1<=p<=3)。

题解：首先有三个延迟标记，add，mul和val。数的和，平方和，立方和以sum1，sum2，sum3表示。这里都用int型，long long型时间比int型慢一倍，会超时。这里把所有数看成（ax+b）的形式，当更新val时，把mul置1，add置0；当更新add时，add+=c即可；当更新mul时，同时更新add相当于（ax+b）\*c=acx+bc，这里ac等于新的a，bc等于新的b，所以mul\*=c，add\*=c。

然后对于 (ax+b)^3=a^3\*x^3+3\*a^2\*b\*x^2+3\*a\*b^2\*x+b^3，也就是root[t].sum3=a^3\*root[t].sum3+3\*a^2\*b\*root[t].sum2+3\*a\*b^2\*root[t].sum1+b^3\*(root[t].r-roo[t].l+1);

                          同理 root[t].sum2=root[t].sum2\*a^2+root[t].sum1\*b\*2+b^2\*(root[t].r-root[t].l+1);

                                     root[t].sum1=root[t].sum1\*a+b\*(root[t].r-root[t].l+1);

                                   注意：sum3,sum2,sum1在这个操作中的顺序不能变。

代码：

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

using namespace std;

const int MAX=1<<20;

const int maxn=1e5+5;

const int mod=10007;

struct segment\_tree{

int add,mul,val,sum1,sum2,sum3,left,right;

}tree[MAX];

void build\_segment\_tree(int i,int l,int r)

{

tree[i].left=l;tree[i].right=r;

tree[i].add=tree[i].val=tree[i].sum1=tree[i].sum2=tree[i].sum3=0;

tree[i].mul=1;

if(l==r) return;

build\_segment\_tree(i\*2,l,(l+r)/2);

build\_segment\_tree(i\*2+1,(l+r)/2+1,r);

}

void update\_segment\_tree(int i,int l,int r,int x,int judge)

{

//cout<<"update "<<i<<" "<<l<<" "<<r<<endl;

if(tree[i].left==l&&tree[i].right==r)

{

switch(judge)

{

case 1:

tree[i].add=(tree[i].add+x)%mod;

tree[i].sum3=(tree[i].sum3+3\*tree[i].sum2%mod\*x%mod+3\*x%mod\*x%mod\*tree[i].sum1%mod+x\*x%mod\*x%mod\*(r-l+1)%mod)%mod;

tree[i].sum2=(tree[i].sum2+2\*tree[i].sum1%mod\*x%mod+x\*x%mod\*(r-l+1)%mod)%mod;

tree[i].sum1=(tree[i].sum1+(r-l+1)\*x% mod)%mod;

break;

case 2:

tree[i].mul=(tree[i].mul\*x)%mod;

tree[i].add=(tree[i].add\*x)%mod;

tree[i].sum3=tree[i].sum3\*x%mod\*x%mod\*x%mod;

tree[i].sum2=tree[i].sum2\*x%mod\*x%mod;

tree[i].sum1=tree[i].sum1\*x%mod;

break;

case 3:

tree[i].val=x;

tree[i].add=0;

tree[i].mul=1;

tree[i].sum3=x\*x%mod\*x%mod\*(r-l+1)%mod;

tree[i].sum2=x\*x%mod\*(r-l+1)%mod;

tree[i].sum1=x\*(r-l+1)%mod;

break;

}

return;

}

int a=tree[i].add,b=tree[i].mul,c=tree[i].val;

if(tree[i].val)

{

tree[i\*2].mul=tree[i\*2+1].mul=b;

tree[i\*2].add=tree[i\*2+1].add=a;

tree[i\*2].val=tree[i\*2+1].val=c;

tree[i\*2].sum3=(tree[i\*2].right-tree[i\*2].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2].sum2=(tree[i\*2].right-tree[i\*2].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2].sum1=(tree[i\*2].right-tree[i\*2].left+1)\*(c\*b%mod+a)%mod;

tree[i\*2+1].sum3=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2+1].sum2=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2+1].sum1=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*(c\*b%mod+a)%mod;

tree[i].val=tree[i].add=0;

tree[i].mul=1;

}

else

{

tree[i\*2].add=(tree[i\*2].add\*b+a)%mod;

tree[i\*2].mul=tree[i\*2].mul\*b%mod;

tree[i\*2+1].add=(tree[i\*2+1].add\*b+a)%mod;

tree[i\*2+1].mul=tree[i\*2+1].mul\*b%mod;

tree[i\*2].sum3=(b\*b%mod\*b%mod\*tree[i\*2].sum3%mod+3\*b\*b%mod\*tree[i\*2].sum2%mod\*a%mod+3\*a\*a%mod\*b%mod\*tree[i\*2].sum1%mod+a\*a%mod\*a%mod\*(tree[i\*2].right-tree[i\*2].left+1)%mod)%mod;

tree[i\*2].sum2=(b\*b%mod\*tree[i\*2].sum2%mod+2\*b\*a%mod\*tree[i\*2].sum1%mod+a\*a%mod\*(tree[i\*2].right-tree[i\*2].left+1)%mod)%mod;

tree[i\*2].sum1=(b\*tree[i\*2].sum1%mod+a\*(tree[i\*2].right-tree[i\*2].left+1)%mod)%mod;

tree[i\*2+1].sum3=(b\*b%mod\*b%mod\*tree[i\*2+1].sum3%mod+3\*b\*b%mod\*tree[i\*2+1].sum2%mod\*a%mod+3\*a\*a%mod\*b%mod\*tree[i\*2+1].sum1%mod+a\*a%mod\*a%mod\*(tree[i\*2+1].right-tree[i\*2+1].left+1)%mod)%mod;

tree[i\*2+1].sum2=(b\*b%mod\*tree[i\*2+1].sum2%mod+2\*b\*a%mod\*tree[i\*2+1].sum1%mod+a\*a%mod\*(tree[i\*2+1].right-tree[i\*2+1].left+1)%mod)%mod;

tree[i\*2+1].sum1=(b\*tree[i\*2+1].sum1%mod+a\*(tree[i\*2+1].right-tree[i\*2+1].left+1)%mod)%mod;

tree[i].val=tree[i].add=0;

tree[i].mul=1;

}

if(l<=tree[i\*2].right)

{

if(r<=tree[i\*2].right) update\_segment\_tree(i\*2,l,r,x,judge);

else

{

update\_segment\_tree(i\*2,l,tree[i\*2].right,x,judge);

update\_segment\_tree(i\*2+1,tree[i\*2+1].left,r,x,judge);

}

}

else update\_segment\_tree(i\*2+1,l,r,x,judge);

tree[i].sum1=(tree[i\*2].sum1+tree[i\*2+1].sum1)%mod;

tree[i].sum2=(tree[i\*2].sum2+tree[i\*2+1].sum2)%mod;

tree[i].sum3=(tree[i\*2].sum3+tree[i\*2+1].sum3)%mod;

}

int query\_segment\_tree(int i,int l,int r,int x)

{

if(tree[i].right==r&&tree[i].left==l)

{

switch(x)

{

case 1: return tree[i].sum1;

case 2: return tree[i].sum2;

case 3: return tree[i].sum3;

}

}

int a=tree[i].add,b=tree[i].mul,c=tree[i].val;

if(tree[i].val)

{

tree[i\*2].mul=tree[i\*2+1].mul=b;

tree[i\*2].add=tree[i\*2+1].add=a;

tree[i\*2].val=tree[i\*2+1].val=c;

tree[i\*2].sum3=(tree[i\*2].right-tree[i\*2].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2].sum2=(tree[i\*2].right-tree[i\*2].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2].sum1=(tree[i\*2].right-tree[i\*2].left+1)\*(c\*b%mod+a)%mod;

tree[i\*2+1].sum3=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2+1].sum2=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*(c\*b%mod+a)%mod\*(c\*b%mod+a)%mod;

tree[i\*2+1].sum1=(tree[i\*2+1].right-tree[i\*2+1].left+1)\*(c\*b%mod+a)%mod;

tree[i].val=tree[i].add=0;

tree[i].mul=1;

}

else

{

tree[i\*2].add=(tree[i\*2].add\*b%mod+a)%mod;

tree[i\*2].mul=tree[i\*2].mul\*b%mod;

tree[i\*2+1].add=(tree[i\*2+1].add\*b%mod+a)%mod;

tree[i\*2+1].mul=tree[i\*2+1].mul\*b%mod;

tree[i\*2].sum3=(b\*b%mod\*b%mod\*tree[i\*2].sum3%mod+3\*b\*b%mod\*tree[i\*2].sum2%mod\*a%mod+3\*a\*a%mod\*b%mod\*tree[i\*2].sum1%mod+a\*a%mod\*a%mod\*(tree[i\*2].right-tree[i\*2].left+1)%mod)%mod;

tree[i\*2].sum2=(b\*b%mod\*tree[i\*2].sum2%mod+2\*b\*a%mod\*tree[i\*2].sum1%mod+a\*a%mod\*(tree[i\*2].right-tree[i\*2].left+1)%mod)%mod;

tree[i\*2].sum1=(b\*tree[i\*2].sum1%mod+a\*(tree[i\*2].right-tree[i\*2].left+1)%mod)%mod;

tree[i\*2+1].sum3=(b\*b%mod\*b%mod\*tree[i\*2+1].sum3%mod+3\*b\*b%mod\*tree[i\*2+1].sum2%mod\*a%mod+3\*a\*a%mod\*b%mod\*tree[i\*2+1].sum1%mod+a\*a%mod\*a%mod\*(tree[i\*2+1].right-tree[i\*2+1].left+1)%mod)%mod;

tree[i\*2+1].sum2=(b\*b%mod\*tree[i\*2+1].sum2%mod+2\*b\*a%mod\*tree[i\*2+1].sum1%mod+a\*a%mod\*(tree[i\*2+1].right-tree[i\*2+1].left+1)%mod)%mod;

tree[i\*2+1].sum1=(b\*tree[i\*2+1].sum1%mod+a\*(tree[i\*2+1].right-tree[i\*2+1].left+1)%mod)%mod;

tree[i].val=tree[i].add=0;

tree[i].mul=1;

}

if(l<=tree[i\*2].right)

{

if(r<=tree[i\*2].right) return query\_segment\_tree(i\*2,l,r,x);

else return (query\_segment\_tree(i\*2,l,tree[i\*2].right,x)+query\_segment\_tree(i\*2+1,tree[i\*2+1].left,r,x))%mod;

}

else return query\_segment\_tree(i\*2+1,l,r,x);

}

int main()

{

int n,m,op,a,b,c;

while(~scanf("%d%d",&n,&m)&&n&&m)

{

build\_segment\_tree(1,1,n);

while(m--)

{

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

if(op==4)

{

printf("%d\n",query\_segment\_tree(1,a,b,c));

}

else

{

update\_segment\_tree(1,a,b,c,op);

}

}

}

return 0;

}

# 树状数组

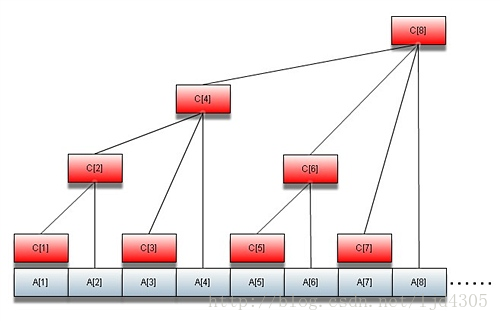
## 一维 单点更新 区间查询

树状数组这个数据结构是在设计压缩算法时被发现的。我觉得其中最重要的是运用了二进制的思想。（这个接下来会重点说）

树状数组的用途就是维护一个数组，重点不是这个数组，而是要维护的东西，最常用的求区间和问题，单点更新。但是某些大牛很6，完成部分线段树能完成的功能，比如区间更新，区间求最值问题。

树状数组是一个可以很高效的进行区间统计的数据结构。在思想上类似于线段树，比线段树节省空间，编程复杂度比线段树低，但适用范围比线段树小。树状数组和线段树很像，能用树状数组解决的问题，基本上都能用线段树解决，而线段树能解决的树状数组不一定能解决。相比较而言，树状数组效率要高很多。

树状数组(Binary Indexed Tree(B.I.T), Fenwick Tree)是一个查询和修改复杂度都为log(n)的数据结构。 平常我们遇到的一些对数组进行维护查询的操作，比较常见的如，修改某点的值、求某个区间的和，而这两种恰恰是树状数组的强项！当然，数据规模不大的时候，对于修改某点的值是非常容易的，复杂度是O(1)，但是对于求一个区间的和就要扫一遍了，复杂度是O(N)，如果实时的对数组进行M次修改或求和，最坏的情况下复杂度是O(M\*N)，当规模增大后这是划不来的！而树状数组干同样的事复杂度却是O(M\*log2N)。



树状数组的基础是一个被构造出来的式子：C[i]=A[i]+A[i-1]+….+A[i-2^k+1]，k代表i的二进制的最后连续0的个数 比如 对于1000和101000，k=3。即C[i]=A[i]+…有lowbit(i)个A数组相加。lowbit(i)代表2^k即i最低位的1所代表的数。

接下来则很容易发现 节点和子节点的是有关系的，这种关系就是 i=j+lowbit(j);，lowbit是j的最低位1所代表的数字 比如对于 1000(8的二进制) 1000=100+lowbit(100)=110+lowbit(110)=111+lowbit(111)。这个关系是树状数组的核心，有了这个关系，我们可以把子区间的变化以log2n的次数传递上去。

接下来说一下求 1-n的和，举个例子 求1-11000 则 Sum(11000)=C[11000]+C[10000]。 因为 根据C[i]的构造方法 C[i]是从A[i]开始的2^k个元素的和，则C[11000]求了A[11000] +A[10111]+ A[10110]+ A[10101] +A[10100] +A[10010]+ A[10001] 这2^k个数 然后接着C[10000]求出了剩下 10000个元素的和 到了这我们就大概了解了树状数组的发明者的天才的构造是从何而来的了。普通的求1-n的和储存的数据太多，而这位天才则想，我们能不能根据二进制的思想来储存这些值呢？任何一个数，都可以由若干个二进制数相加而成，如果我们在求Sum(n)之前就知道了 n对应的二进制数从最低位开始，每个1所代表的数字的前2^i个数的和，我们不就能在时间复杂度log2n内求出所有的值 比如 101110 我们如果知道 101110->101101，101100->101001，101000->100001，100000->1各自的和，就可以在空间复杂度和时间复杂度很小的情况下求出1-101110了。然后对于区间和的修改，又利用每个小区间向上转移，修改了大区间的和。

int lowbit(int x) //求x的最低位1

{

**return** x&(-x); //这个自己体会下，了解点位于位运算的应该还是能明白。

}

顺便说一下，用x&(-x)==x还能判断x是不是2^n数。

void update(int i,int value)

{

**while**(i<maxn)

{

c[i] += value;

i += lowbit(i);

}

}

int query(int i)

{

int ans=0;

**while**(i>0)

{

ans += c[i];

i -= lowbit(i);

}

**return** ans;

}

这里一定要注意：query函数的判断条件如果习惯写成while(x)的请注意一下，如果x为负数的话，会造成死循环。一定要将数据偏移到1下标开始，因为你是从1下标开始建的树状数组。

## 一维 区间更新 单点查询

以hdu1556为例

题意：n个气球编号1-n（n<=100000），n次对[a,b]区间内的气球进行涂色，输出1-n的气球最终被涂色的次数。

这题可以说是树状数组区间更新，单点查询的模板题。

对于树状数组的区间更新，单点查询，其实就是维护一个差分数组。以该题为例：

定义两个数组a[],b[]。a[]数组表示气球涂色次数，初始均为0。设b[n] = a[n] - a[n-1]（即b[]数组是a[]数组的一个差分数组），因为a[]数组初始均为0，所以b[]数组初始也为0，且b[1]+b[2]+…+b[n] = a[n] - a[0] = a[n]。

假设这里我们要对区间为[l,r]的气球图n次色，也就是对a[]数组的[l,r]区间都加n。即：a[1]′=a[1]，…，a[l−1]′=a[l−1]，a[l]′=a[l]+n，…，a[r]′=a[r]+n，a[r+1]′=a[r+1]，…那么对于b[1]，…，b[l−1]，b[l+1]，…，b[r]，b[r+2]，…不会变（区间外部不变导致b[1−l]和b[r+2−n]不变，区间内部增量相同导致b[l+1−r]不变），但是b[l]会+n，b[r+1]会−n。

这就转变成了对b[]数组的单点更新、区间查询。

代码：

#include <bits/stdc++.h>

**using** **namespace** std;

const int maxn = 1e5+5;

int sz[maxn],n;

int lowbit(int x)

{

**return** x&(-x);

}

int update(int x,int num)

{

**while**(x<=n)

{

sz[x] += num;

x += lowbit(x);

}

}

int query(int x)

{

int ans = 0;

**while**(x>0)

{

ans += sz[x];

x -= lowbit(x);

}

**return** ans;

}

int main()

{

int a,b;

**while**(~scanf("%d",&n)&&n!=0)

{

memset(sz,0,**sizeof**(sz));

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

{

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

update(a,1);

update(b+1,-1);

}

for(int i=1;i<=n;i++) printf("%d%c",query(i),(i==n ? '\n' : ' '));

}

return 0;

}

## 一维 区间更新 区间查询

其实树状数组本质就是单点更新、区间查询，其它用法都是通过转化成本质用法实现

一维的区间更新和单点查询是通过维护一个差分数组转化实现，而一维的区间更新和区间查询是通过维护一个差分数组和一个与差分数组有关的数组转化实现。

设原数组是a[n]，差分数组c[n]，c[i]=a[i]-a[i-1]，那么明显地a[n]=，如果想要修改a[i]到a[j] (比如+v)，只需令c[i]+=v,c[j+1]-=v即可，到这区间更新就实现了。

那么怎么实现区间查询呢？（比如查询a[1]-a[n]）

a[1] + a[2] + … + a[n]

= c[1] + (c[1]+c[2]) + … + (c[1]+c[2]+…+c[n])

= n\*c[1] + (n-1)\*c[2] + … + c[n]

= (n+1)\*(c[1]+…+c[n]) - (1\*c[1] + 2\*c[2] + … + n\*c[n])

所以我们在维护差分数组c[]的同时维护与之相关的数组c2[n] = n\*c[n]即可

以poj3468（线段树区间更新的模板题）为例：

代码：

//#include <bits/stdc++.h>

#include <cstdio>

#include <cstdlib>

#include <cstring>

**using** **namespace** std;

**typedef** long long ll;

const int maxn = 1e5+5;

ll a[maxn],c1[maxn],c2[maxn],n,q;

ll lowbit(ll x)

{

**return** x&(-x);

}

void update(ll x,ll value)

{

ll i=x;

**while**(x<=n)

{

c1[x] += value;

c2[x] += i\*value;

//更新c1[]的同时更新c2[]

x += lowbit(x);

}

}

ll query(ll x)

{

ll tmp1=0,tmp2=0,i=x;

**while**(x>0)

{

tmp1 += c1[x];

tmp2 += c2[x];

x -= lowbit(x);

}

**return** (i+1)\*tmp1-tmp2;

//tmp1 = c1[1]+...+c1[n],tmp2 = c2[1]+...+c2[n] = 1\*c1[1]+...+n\*c1[n]

//返回a[1]+...+a[x]

}

int main()

{

ll tmp;

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

memset(a,0,**sizeof**(a));

memset(c1,0,**sizeof**(c1));

memset(c2,0,**sizeof**(c2));

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

{

scanf("%lld",a+i);

//初始c1[]和c2[],单点更新

update(i,a[i]-a[i-1]);

}

int a,b;

char op;

**while**(q--)

{

getchar();

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

**if**(op=='C')

{

scanf("%lld",&tmp);

//区间更新a[],通过两次单点更新c1[]实现,更新c1[]的同时更新c2[]

update(a,tmp);

update(b+1,-tmp);

}

**else**

{

printf("%lld\n",query(b)-query(a-1));

//输出a[a]+...+a[b]

}

}

**return** 0;

}

## 二维 单点更新 区间查询

二维的树状数组与一维的没有本质区别，只是把一维扩展到了二维。

我们看看树状数组是怎么扩展到二维的。和一维一样，设a[][]，c[][]，我们看看这时候c[][]的组成。

设原始二维数组为：

A[][]={{a11,a12,a13,a14,a15,a16,a17,a18,a19},

{a21,a22,a23,a24,a25,a26,a27,a28,a29},

{a31,a32,a33,a34,a35,a36,a37,a38,a39},

{a41,a42,a43,a44,a45,a46,a47,a48,a49}};

记:

B[1]={a11,a11+a12,a13,a11+a12+a13+a14,a15,a15+a16,…} 这是第一行的一维树状数组

B[2]={a21,a21+a22,a23,a21+a22+a23+a24,a25,a25+a26,…} 这是第二行的一维树状数组

B[3]={a31,a31+a32,a33,a31+a32+a33+a34,a35,a35+a36,…} 这是第三行的一维树状数组

B[4]={a41,a41+a42,a43,a41+a42+a43+a44,a45,a45+a46,…} 这是第四行的一维树状数组

那么：

C[1][1]=a11,C[1][2]=a11+a12,C[1][3]=a13,C[1][4]=a11+a12+a13+a14,c[1][5]=a15,C[1][6]=a15+a16,… 这是A[][]第一行的一维树状数组

C[2][1]=a11+a21,C[2][2]=a11+a12+a21+a22,C[2][3]=a13+a23,C[2][4]=a11+a12+a13+a14+a21+a22+a23+a24,C[2][5]=a15+a25,C[2][6]=a15+a16+a25+a26,… 这是A[][]数组第一行与第二行相加后的树状数组

C[3][1]=a31,C[3][2]=a31+a32,C[3][3]=a33,C[3][4]=a31+a32+a33+a34,C[3][5]=a35,C[3][6]=a35+a36,…这是A[][]第三行的一维树状数组

C[4][1]=a11+a21+a31+a41,C[4][2]=a11+a12+a21+a22+a31+a32+a41+a42,C[4][3]=a13+a23+a33+a43,… 这是A[][]数组第一行+第二行+第三行+第四行后的树状数组

二维数组的规律就是,不管是横坐标还是纵坐标,将他们单独拿出来,他们都符合x += lowbit(x)，属于它的父亲节点，即都符合那个经典的图。比如C[4][2]：

单独看横坐标（就是把纵坐标去掉，重复的只算一个）：C[4] = C[2]+C[3]+a[4] （经典图得出）= a[1]+a[2]+a[3]+a[4] （从上面公式得出）

单独看纵坐标（就是把横坐标去掉，重复的只算一个）：C[2] = C[1]+a[2] （经典图得出）= a[1]+a[2] （上面公式得出）

以hdu2642为例：

题意：一个星空，二维的。上面有1000\*1000的格点,每个格点上有星星在闪烁。一开始时星星全部暗淡着，有M个操作：

B x y 点亮一盏星星

D x y 熄灭一盏星星

Q x1 x2 y1 y2 查询这个矩形里面亮着的星星的个数。

题解：这就是个二维树状数组单点更新、区间查询的模板题，直接写即可。不过有几个需要注意的地方。就像我在一维里说的，树状数组下标是从1开始维护的，所以我们要把数据偏移到下标从1开始，二维里也是一样。这里坐标可能为0，所以我们把每个坐标都++，然后就是点亮的星星不能重复点亮，暗淡的星星不能重复暗淡，设一个状态的数组即可。

代码：

#include <bits/stdc++.h>

**using** **namespace** std;

const int maxn = 1005;

int sz[maxn][maxn],status[maxn][maxn];

int lowbit(int x)

{

**return** x&(-x);

}

void update(int x,int y,int value)

{

**for**(int i=x;i<=maxn;i+=lowbit(i))

**for**(int j=y;j<=maxn;j+=lowbit(j))

sz[i][j] += value;

}

int query(int x,int y)

{

int ans = 0;

**for**(int i=x;i>0;i-=lowbit(i))

**for**(int j=y;j>0;j-=lowbit(j))

ans += sz[i][j];

**return** ans;

}

int main()

{

int T;

int x1,x2,y1,y2;

char op;

memset(sz,0,**sizeof**(sz));

memset(status,0,**sizeof**(status));

scanf("%d",&T);

**while**(T--)

{

getchar();

scanf("%c",&op);

**switch**(op)

{

**case** 'B':

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

x1++,y1++;

**if**(status[x1][y1]==0)

{

update(x1,y1,1);

status[x1][y1]=1;

}

**break**;

**case** 'D':

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

x1++,y1++;

**if**(status[x1][y1]>0)

{

update(x1,y1,-1);

status[x1][y1]=0;

}

**break**;

**case** 'Q':

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

x1++,x2++,y1++,y2++;

**if**(x1>x2) swap(x1,x2);

**if**(y1>y2) swap(y1,y2);

printf("%d\n",query(x2,y2)-query(x2,y1-1)-query(x1-1,y2)+query(x1-1,y1-1));

**break**;

}

}

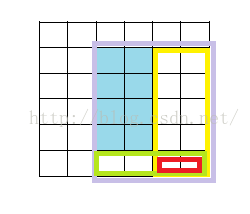
**return** 0;

}

## 二维 区间更新 单点查询

可以通过一维树状数组区间更新、单点查询类推得到。

区间更新：就像这个图，我们想对蓝色区间内修改一次，那么如果我们改变第一个蓝色的点会造成紫色区间内的修改，所以我们必须再通过修改黄、绿、红来抵消这些多余的影响。（容斥原理）



单点查询：直接查询即可。

以poj2155为例：

题意：给了一个N\*N的矩阵(1<=N<=1000)。初始时每个格子里都是0，而对一个格子进行操作是将其0变1或1变0。现在不断地对其一些子矩阵进行操作，并且不断询问某个格子是0还是1。

题解：就是二维树状数组区间更新、单点查询的裸题，直接写即可。这里可以用异或可以简化操作。具体看代码注释。

//#include <bits/stdc++.h>

#include <cstdio>

#include <cstdlib>

#include <cstring>

**using** **namespace** std;

const int maxn = 1005;

int sz[maxn][maxn];

int lowbit(int x)

{

**return** x&(-x);

}

void update(int x,int y)

{

**for**(int i=x;i<maxn;i+=lowbit(i))

**for**(int j=y;j<maxn;j+=lowbit(j))

sz[i][j] ^= 1; //从0变1，1变0

}

int query(int x,int y)

{

int ans = 0;

**for**(int i=x;i>0;i-=lowbit(i))

**for**(int j=y;j>0;j-=lowbit(j))

ans ^= sz[i][j];

**return** ans; //奇数个1说明就是1，偶数个1为0

}

int main()

{

int T;

int n,q,x1,x2,y1,y2;

char op;

scanf("%d",&T);

**while**(T--)

{

memset(sz,0,**sizeof**(sz));

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

**while**(q--)

{

getchar();

scanf("%c",&op);

**if**(op=='C')

{

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

update(x1,y1);

update(x1,y2+1);

update(x2+1,y1);

update(x2+1,y2+1);

}

**else**

{

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

printf("%d\n",query(x1,y1));

}

}

printf("\n");

}

**return** 0;

}

## 二维 区间更新 区间查询

二维的区间更新上面已经讲了，现在来看区间查询。

设（x1,y1）和（x2,y2），（x1<=x2,y1<=y2）

sum(x1,y1,x2,y2)=sum(1,1,x2,y2)−sum(1,1,x2,y1−1)−sum(1,1,x1−1,y2)+sum(1,1,x1−1,y1−1)

那么我们只需知道如何查询类似区间sum（1，1，x1，y1）就可以了。

Sum(1,1,x1,y1)=

而

=

所以我们维护四个树状数组即可delta[i][j],delta[i][j]∗i,delta[i][j]∗j,delta[i][j]∗i∗j。

# 博弈论

## 巴什博弈（Bash Game）

1. 问题模型：只有一堆n个物品，两个人轮流从这堆物品中取物，规定每次至少取一个，最多取m个，最后取光者得胜。
2. 解决思路：当n=m+1时，由于一次最多只能取m个，所以无论先取者拿走多少个，后取者都能够一次拿走剩余的物品，后者取胜，所以当一方面对的局势是n%(m+1)=0时，其面临的是必败的局势。所以当n=（m+1)\*r+s，（r为任意自然数，s≤m)时,如果先取者要拿走s个物品，如果后取者拿走x（≤m)个，那么先取者再拿走m+1-k个，结果剩下（m+1）（r-1）个，以后保持这样的取法，那么先取者肯定获胜。总之，要保持给对手留下（m+1）的倍数，就能最后获胜。
3. 变形：条件不变，改为最后取光的人输或两个人轮流报数，每次至少报一个，最多报十个，谁能报到100者胜。

结论：n%(m+1)==0 后手必胜否则后手必败。

## 威佐夫博奕（Wythoff Game）

1. 问题模型：有两堆各若干个物品，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

2. 解决思路：设（ai,bi）（ai ≤bi  ,i=0，1，2，…,n)表示两堆物品的数量并称其为局势，如果甲面对（0，0），那么甲已经输了，这种局势我们称为奇异局势。前几个奇异局势是：（0，0）、（1，2）、（3，5）、（4，7）、（6，10）、（8，13）、（9，15）、（11，18）、（12，20）。

奇异局势有如下性质：

<1>. 任何自然数都包含在一个且仅有一个奇异局势中。

<2>. 任意操作都可将奇异局势变为非奇异局势。

<3>. 采用适当的方法，可以将非奇异局势变为奇异局势。

假设面对的局势是（a,b），若 b = a，则同时从两堆中取走 a 个物体，就变为了奇异局势（0，0）；如果a = ak ，b > bk 那么，取走b - bk个物体，即变为奇异局势；如果 a = ak ， b < bk 则同时从两堆中拿走a-a[b-a] 个物体变为奇异局势（ a[b-a], b-a+a[b-a]）；如果a > ak ，b= ak + k 则从第一堆中拿走多余的数量a - ak 即可；如果a < ak ，b= ak + k,分两种情况，第一种，a=aj （j < k）从第二堆里面拿走 b - bj 即可；第二种，a=bj （j < k）从第二堆里面拿走 b - aj 即可。

最后，任给一个局势（a，b），怎样判断它是不是奇异局势？

若a=[(b-a)\*（1+√5）/2] （方括号为取整函数）那么为奇异局势，否则不是。

1. 结论：面对非奇异局势，先拿者必胜；反之，则后拿者取胜。

以poj 1067为例：

代码：

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <cmath>

#include <iostream>

using namespace std;

typedef long long ll;

ll a,b;

int Wythoff()

{

ll tmp = floor((b-a)\*(sqrt(5)+1)/2.0);

if(tmp==a) return 0;

return 1;

}

int main()

{

while(~scanf("%lld%lld",&a,&b))

{

if(a>b) swap(a,b);

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

}

return 0;

}

## 尼姆博弈（Nimm Game）

1. 问题模型：有三堆各若干个物品，两个人轮流从某一堆取任意多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

2. 解决思路：这种情况最有意思，它与二进制有密切关系，我们用（a，b，c）表示某种局势，首先（0，0，0）显然是奇异局势，无论谁面对奇异局势，都必然失败。第二种奇异局势是（0，n，n），只要与对手拿走一样多的物品，最后都将导致（0，0，0）。仔细分析一下，（1，2，3）也是奇异局势，无论对手如何拿，接下来都可以变为（0，n，n）的情形。 计算机算法里面有一种叫做按位模2加，也叫做异或的运算。对于任何奇异局势(a,b,c)其a^b^c=0。如果我们面对的是一个非奇异局势（a，b，c），要如何变为奇异局势呢？假设 a < b< c,我们只要将 c 变为 a^b,即可,因为有如下的运算结果: a^b^(a^b)=(a^a)^(b^b)=0^0=0。要将c 变为a^b，只要从 c中减去 c-（a^b）即可。

3. 推广：当石子堆数为n堆时，则推广为当对每堆的数目进行异或之后值为零是必败态。

取火柴的游戏  
题目1：今有若干堆火柴，两人依次从中拿取，规定每次只能从一堆中取若干根，   
可将一堆全取走，但不可不取，最后取完者为胜，求必胜的方法。   
题目2：今有若干堆火柴，两人依次从中拿取，规定每次只能从一堆中取若干根，   
可将一堆全取走，但不可不取，最后取完者为负，求必胜的方法。  
  
先解决第一个问题吧。  
定义：若所有火柴数异或为0，则该状态被称为利他态，用字母T表示；否则，   
为利己态，用S表示。  
[定理1]：对于任何一个S态，总能从一堆火柴中取出若干个使之成为T态。  
  
[定理2]：T态，取任何一堆的若干根，都将成为S态。  
  
[定理 3]：S态，只要方法正确，必赢。   
 最终胜利即由S态转变为T态，任何一个S态，只要把它变为T态，（由定理1，可以把它变成T态。）对方只能把T态转变为S态(定理2)。这样，所有S态向T态的转变都可以有己方控制，对方只能被动地实现由T态转变为S态。故S态必赢。  
[定理4]：T态，只要对方法正确，必败。

接着来解决第二个问题。  
定义：若一堆中仅有1根火柴，则被称为孤单堆。若大于1根，则称为充裕堆。  
定义：T态中，若充裕堆的堆数大于等于2，则称为完全利他态，用T2表示；若充裕堆的堆数等于0，则称为部分利他态，用T0表示。  
   
孤单堆的根数异或只会影响二进制的最后一位，但充裕堆会影响高位（非最后一位）。一个充裕堆，高位必有一位不为0，则所有根数异或不为0。故不会是T态。  
[定理5]：S0态，即仅有奇数个孤单堆，必败。T0态必胜。   
证明：  
S0态，其实就是每次只能取一根。每次第奇数根都由己取，第偶数根都由对   
方取，所以最后一根必己取。败。同理,  T0态必胜#  
[定理6]：S1态，只要方法正确，必胜。   
证明：  
若此时孤单堆堆数为奇数，把充裕堆取完；否则，取成一根。这样，就变成奇数个孤单堆，由对方取。由定理5，对方必输。己必胜。  #   
[定理7]：S2态不可转一次变为T0态。   
证明：  
充裕堆数不可能一次由2变为0。得证。  #

[定理8]：S2态可一次转变为T2态。   
证明：  
由定理1，S态可转变为T态，态可一次转变为T态，又由定理6，S2态不可转一次变为T0态，所以转变的T态为T2态。  #   
[定理9]：T2态，只能转变为S2态或S1态。   
证明：  
由定理2，T态必然变为S态。由于充裕堆数不可能一次由2变为0，所以此时的S态不可能为S0态。命题得证。   
[定理10]：S2态，只要方法正确，必胜.   
证明：  
方法如下：   
      1）  S2态，就把它变为T2态。（由定理8）   
      2）  对方只能T2转变成S2态或S1态（定理9）  
    若转变为S2,  转向1）   
    若转变为S1,  这己必胜。（定理5）   
[定理11]：T2态必输。

综上所述，必输态有：  T2,S0   
          必胜态：    S2,S1,T0.   
两题比较：   
第一题的全过程其实如下：   
S2->T2->S2->T2->  ……  ->T2->S1->T0->S0->T0->……->S0->T0(全0)   
第二题的全过程其实如下：   
S2->T2->S2->T2->  ……  ->T2->S1->S0->T0->S0->……->S0->T0(全0)   
下划线表示胜利一方的取法。  是否发现了他们的惊人相似之处。   
我们不难发现(见加黑部分)，S1态可以转变为S0态（第二题做法），也可以转变为   
T0（第一题做法）。哪一方控制了S1态，他即可以有办法使自己得到最后一根（转变为   
T0）,也可以使对方得到最后一根（转变为S0）。   
  所以，抢夺S1是制胜的关键！   
  为此，始终把T2态让给对方，将使对方处于被动状态，他早晚将把状态变为S1.

取火柴游戏题目二，以hdu 1907为例：

代码：

#include <bits/stdc++.h>

using namespace std;

int main()

{

int T,n,ans,tmp,flag;

scanf("%d",&T);

while(T--)

{

flag=0;

ans=0;

scanf("%d",&n);

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

{

scanf("%d",&tmp);

ans ^= tmp;

if(tmp>1) flag=1;

}

if(flag)

{

if(ans==0) printf("Brother\n");

else printf("John\n");

}

else

{

if(n&1) printf("Brother\n");

else printf("John\n");

}

}

return 0;

}

## Fibonacci（斐波那契数列）博弈

1. 问题模型：有一堆个数为n的石子，游戏双方轮流取石子，满足：

（1）先手不能在第一次把所有的石子取完；

（2）之后每次可以取的石子数介于1到对手刚取的石子数的2倍之间（包含1和对手刚取的石子数的2倍）。 约定取走最后一个石子的人为赢家。

2. 解决思路：当n为Fibonacci数时，先手必败。即存在先手的必败态当且仅当石头个数为Fibonacci数。

    证明：根据“Zeckendorf定理”（齐肯多夫定理）：任何正整数可以表示为若干个不连续的Fibonacci数之和。如n=83 = 55+21+5+2，我们看看这个分解有什么指导意义：假如先手取2颗，那么后手无法取5颗或更多，而5是一个Fibonacci数，那么一定是先手取走这5颗石子中的最后一颗，同理，接下去先手取走接下来的后21颗中的最后一颗，再取走后55颗中的最后一颗，那么先手赢。

    反证：如果n是Fibonacci数，如n=89：记先手一开始所取的石子数为y

    （1）若y>=34颗（也就是89的向前两项），那么一定后手赢，因为89-34=55=34+21<2\*34。

    （2）y<34时剩下的石子数x介于55到89之间，它一定不是一个Fibonacci数，把x分解成Fibonacci数：x=55+f[i]+…+f[j]，若，如果f[j]<=2y，那么对B就是面临x局面的先手，所以根据之前的分析，后手只要先取f[j]个即可，以后再按之前的分析就可保证必胜。

3. 以hdu2516为例：（斐波那契数列大约在50项左右爆int）

代码：

#include <bits/stdc++.h>

using namespace std;

int sz[55];

void init()

{

sz[0]=0;sz[1]=1;

for(int i=2;i<55;i++) sz[i]=sz[i-1]+sz[i-2];

}

int main()

{

int n;

bool flag;

init();

while(~scanf("%d",&n)&&n!=0)

{

flag = false;

for(int i=1;i<55;i++)

{

if(n==sz[i])

{

printf("Second win\n");

flag=true;

}

}

if(!flag) printf("First win\n");

}

return 0;

}

## 公平组合博弈（Impartial Combinatori Games）

1. 定义：

（1）两人参与。

（2）游戏局面的状态集合是有限。

（3）对于同一个局面，两个游戏者的可操作集合完全相同

（4）游戏者轮流进行游戏。

（5）当无法进行操作时游戏结束，此时不能进行操作的一方算输。

（6）无论游戏如何进行，总可以在有限步数之内结束。

2. 模型：给定一个有向无环图和一个起始顶点上的一枚棋子，两名选手交替的将这枚棋子沿有向边进行移动，无法移动者判负。事实上，这个游戏可以认为是所有公平组合游戏（Impartial Combinatori Games）的抽象模型。其实，任何一个ICG都可以通过把每个局势看成一个顶点，对每个局势和它的子局势连一条有向边来抽象成这个“有向图游戏”。

3. Sprague-Grudy定理：

令N = {0, 1, 2, 3, ...} 为自然数的集合。Sprague-Grundy 函数给游戏中的每个状态分配了一个自然数。结点v的Grundy值等于没有在v的后继的Grundy值中出现的最小自然数.

形式上：给定一个有限子集 S ⊂ N,令mex S(最小排斥值)为没有出现在S中的最小自然数。定义mex(minimal excludant)运算，这是施加于一个集合的运算，表示最小的不属于这个集合的非负整数。例如mex{0,1,2,4}=3、mex{2,3,5}=0、mex{}=0。

对于一个给定的有向无环图，定义关于图的每个顶点的Sprague-Garundy函数g如下：g(x)=mex{ g(y) | y是x的后继 }。

4. SG函数的性质：

（1）所有的终结点所对应的顶点，其SG值为0，因为它的后继集合是空集——所有终结点是必败点（P点）。

（2）对于一个g(x)=0的顶点x，它的所有后继y都满足g(y)!=0——无论如何操作，从必败点（P点）都只能进入必胜点（N点）//对手走完又只能把N留给我们。

（3）对于一个g(x)!=0的顶点，必定存在一个后继点y满足g(y)=0——从任何必胜点（N点）操作，至少有一种方法可以进入必败点（P点）//就是那种我们要走的方法。

5. SG函数解决问题的思路：

样例问题：有n堆石子，每次可以从第一堆里面取1，2或3颗，可以从第二堆里面取奇数颗，可以从之后堆里面取任意颗。

思路：我们通过计算有向无环图的每个顶点的 SG 值，就可以对每种局面找到必胜策略了。但 SG 函数的用途远没有这样简单。如果将有向图游戏变复杂一点，比如说，有向图上并不是只有一枚棋子，而是有 n 枚棋子，每次可以任选一颗进行移动，这时，怎样找到必胜策略呢？

让我们再来考虑一下顶点的 SG 值的意义。当 g(x)=k 时，表明对于任意一个 0<=i<k，都存在 x 的一个后继 y 满足 g(y)=i。也就是说，当某枚棋子的 SG 值是 k 时，我们可以把它变成0、变成 1、„„、变成 k-1，但绝对不能保持 k 不变。不知道你能不能根据这个联想到 Nim游戏， Nim 游戏的规则就是： 每次选择一堆数量为 k 的石子， 可以把它变成 0、 变成 1、 „„、变成 k-1，但绝对不能保持 k 不变。这表明，如果将 n 枚棋子所在的顶点的 SG 值看作 n 堆

相应数量的石子， 那么这个 Nim 游戏的每个必胜策略都对应于原来这 n 枚棋子的必胜策略！

对于 n 个棋子，设它们对应的顶点的 SG 值分别为(a1,a2,„,an)，再设局面(a1,a2,„,an)时的Nim 游戏的一种必胜策略是把 ai 变成 k， 那么原游戏的一种必胜策略就是把第 i 枚棋子移动到一个 SG 值为 k 的顶点。这听上去有点过于神奇——怎么绕了一圈又回到 Nim 游戏上了。

其实我们还是只要证明这种多棋子的有向图游戏的局面是 P-position 当且仅当所有棋子所在的位置的 SG 函数的异或为 0。这个证明与上节的 Bouton’s Theorem 几乎是完全相同的，只需要适当的改几个名词就行了。

刚才，我为了使问题看上去更容易一些，认为 n 枚棋子是在一个有向图上移动。但如果不是在一个有向图上，而是每个棋子在一个有向图上，每次可以任选一个棋子（也就是任选一个有向图）进行移动，这样也不会给结论带来任何变化。

所以我们可以定义有向图游戏的和(Sum of Graph Games)：设 G1、G2、„„、Gn 是 n 个有向图游戏，定义游戏 G 是 G1、G2、„„、Gn 的和(Sum)，游戏 G 的移动规则是：任选一个子游戏Gi并移动上面的棋子。 Sprague-Grundy Theorem就是： g(G)=g(G1)^g(G2)^„^g(Gn)。也就是说，游戏的和的 SG 函数值是它的所有子游戏的 SG 函数值的异或。再考虑在本文一开头的一句话：任何一个 ICG 都可以抽象成一个有向图游戏。所以“SG 函数”和“游戏的和”的概念就不是局限于有向图游戏。我们给每个 ICG 的每个 position 定义SG 值，也可以定义 n 个 ICG 的和。所以说当我们面对由 n 个游戏组合成的一个游戏时，只需对于每个游戏找出求它的每个局面的 SG 值的方法， 就可以把这些 SG 值全部看成 Nim 的石子堆，然后依照找 Nim 的必胜策略的方法来找这个游戏的必胜策略了！

回到本文开头的问题。有 n 堆石子，每次可以从第 1 堆石子里取 1 颗、2 颗或 3 颗，可以从第 2 堆石子里取奇数颗， 可以从第 3 堆及以后石子里取任意颗„„我们可以把它看作 3 个子游戏，第 1 个子游戏只有一堆石子，每次可以取 1、2、3 颗，很容易看出 x 颗石子的局面的SG 值是 x%4。第 2 个子游戏也是只有一堆石子，每次可以取奇数颗，经过简单的画图可以知道这个游戏有 x 颗石子时的 SG 值是 x%2。 第 3 个游戏有 n-2 堆石子， 就是一个 Nim 游戏。

对于原游戏的每个局面，把三个子游戏的 SG 值异或一下就得到了整个游戏的 SG 值，然后就可以根据这个 SG 值判断是否有必胜策略以及做出决策了。 其实看作 3 个子游戏还是保守了些，干脆看作 n 个子游戏，其中第 1、2 个子游戏如上所述，第 3 个及以后的子游戏都是“1 堆石子，每次取几颗都可以” ，称为“任取石子游戏” ，这个超简单的游戏有 x 颗石子的SG 值显然就是 x。其实，n 堆石子的 Nim 游戏本身不就是 n 个“任取石子游戏”的和吗？

所以，对于我们来说，SG 函数与“游戏的和”的概念不是让我们去组合、制造稀奇古怪的游戏， 而是把遇到的看上去有些复杂的游戏试图分成若干个子游戏， 对于每个比原游戏简化很多的子游戏找出它的 SG 函数，然后全部异或起来就得到了原游戏的 SG 函数，就可以解决原游戏了。

6. 计算：

（1）可选步数为1-m的连续整数，直接取模即可，SG(x) = x % (m+1);

（2）可选步数为任意步，SG(x) = x;

（3）可选步数为一系列不连续的数，用mex(计算每个节点的值)

以hdu1536为例：

题意：给出m，接下来m个数表示每堆能拿的数字，有t种情况，每种情况有n堆，接下来n个数表示每堆的个数。求先手输赢。

代码：

#include <bits/stdc++.h>

using namespace std;

const int maxm=105,maxn=10005;

int m,s[maxm],sg[maxn];

bool vis[maxn];

void set\_sg()

{

memset(sg,0,sizeof(sg));

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

{

memset(vis,false,sizeof(vis));

for(int j=0;j<m&&i>=s[j];j++) vis[sg[i-s[j]]]=true; //把节点i的每个后继节点的sg值的vis置1

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

{

if(!vis[j])

{

sg[i]=j;

break;

}

}

}

return;

}

int main()

{

int t,n,ans,tmp;

while(scanf("%d",&m)&&m)

{

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

sort(s,s+m); //一定要排序

set\_sg();

scanf("%d",&t);

while(t--)

{

ans=0;

scanf("%d",&n);

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

{

scanf("%d",&tmp);

ans ^= sg[tmp];

}

if(ans) printf("W");

else printf("L");

}

printf("\n");

}

return 0;

}

# 数学

## 组合数学

### 莫比乌斯反演

公式：（两种形式）

约数和：

倍数和：

其中d|n表示n%d=0即d为n的因子，μ()是莫比乌斯函数

应用：对于一些函数f(n)，如果我们很难直接求出它的值，而容易求出倍数和或约数和F(n)，那么我们可以通过莫比乌斯反演来求得f(n)的值。

莫比乌斯函数

定义：

设，其中为互异素数

即n的所有质因子次数均为1

性质：

1. 对于任意的正整数n

2. 莫比乌斯函数是积性函数：μ(a∗b)=μ(a)∗μ(b),gcd(a,b)=1

求莫比乌斯函数的模板：

const int maxn = 1e4+5 //所求莫比乌斯函数的范围，我以10000为例

int mu[maxn]; //莫比乌斯函数值

int vis[maxn]; //vis[i]=1表示i不是素数，为0表示i是素数

int prime[maxn]; //存放素数

void Moblus()

{

memset(vis,0,**sizeof**(vis);

int num=0; //素数的个数

mu[1]=1;

**for**(int i=2;i<maxn;i++)

{

**if**(!vis[i]) //i是素数

{

mu[i]=-1; //根据定义知单个素数其莫比乌斯函数值为(-1)^1=-1

prime[num++]=i; //存进prime[]中

}

**for**(int j=0;j<num && prime[j]\*i<maxn;j++)

{

vis[prime[j]\*i]=1; //素数的倍数不为素数

**if**(i%prime[j]==0)

{

mu[prime[j]\*i]=0; //质因子的次数不为1

**break**;

}

**else** mu[prime[j]\*i] = -mu[i]; //根据莫比乌斯函数的性质mu[prime[j]\*i]=mu[prime[j]]\*mu[i]=-mu[i]，因为prime[j]为单素数

}

}

}

以hust1720为例

题意：给你n\*m个数（1<=n,m<=100），每个数不大于10000，问你有多少组（a,b,c,d）满足gcd(a,b,c,d)==1。PS：（1，2，3，4）和（2，1，3，4）是同一组。

题解：可以说是一个裸的莫比乌斯反演。因为我们直接求gcd(a,b,c,d)==1的组数不好快速求解，而时限只有1s。但是当我们求其倍数即gcd(a,b,c,d)==1\*k的组数时更好求解（只需知道k的倍数num就可得出答案）。   
所以我们直接用莫比乌斯反演：设f(x)为gcd(a,b,c,d)==x的组数，F(x)为gcd(a,b,c,d)==k\*x的组数，设有num个数是x的倍数，所以F(x)=。   
根据莫比乌斯反演公式： 

f(1)即所求答案，设x=1，ma为给出数中最大值，那么 

代码:

#include <bits/stdc++.h>

**using** **namespace** std;

#define time\_ (printf("%.6f\n", double(clock())/CLOCKS\_PER\_SEC))

**typedef** long long ll;

const int INF = 0x3f3f3f3f;

const double pi = acos(-1.0);

const int maxn = 1e4+5;

int vis[maxn],prime[maxn],mu[maxn];

int sz[maxn],cnt[maxn]; //cnt[]为1-10000每个数的个数，sz[]为1-10000每个数的倍数的个数

void Moblus() //计算莫比乌斯函数的值

{

memset(vis,0,**sizeof**(vis));

int num=0;

mu[1]=1;

**for**(int i=2;i<maxn;i++)

{

**if**(!vis[i])

{

prime[num++]=i;

mu[i]=-1;

}

**for**(int j=0;j<num && i\*prime[j]<maxn;j++)

{

vis[prime[j]\*i]=1;

**if**(i%prime[j]==0)

{

mu[prime[j]\*i]=0;

**break**;

}

**else** mu[prime[j]\*i]=-mu[i];

}

}

}

ll solve(int x)

{

memset(sz,0,**sizeof**(sz));

**for**(int i=1;i<=x;i++) //求每个数倍数的个数

{

**for**(int j=i;j<=x;j+=i)

{

sz[i] += cnt[j];

}

}

ll ans=0;

**for**(int i=1;i<=x;i++)

{

**if**(sz[i]>=4) //如果倍数个数大于4，即可用排列组合计算F(i),否则组数为0

{

ans += (ll)mu[i]\*sz[i]\*(sz[i]-1)\*(sz[i]-2)\*(sz[i]-3)/24;

}

}

**return** ans;

}

int main()

{

int n,m,tmp,ma;

Moblus();

**while**(~scanf("%d%d",&n,&m))

{

memset(cnt,0,**sizeof**(cnt));

ma=0;

**for**(int i=0;i<n\*m;i++)

{

scanf("%d",&tmp);

++cnt[tmp];

ma = max(ma,tmp);

}

**if**(n\*m<4) printf("0\n");

**else** printf("%lld\n",solve(ma));

}

**return** 0;

}

## 数论

### 唯一分解定理

唯一分解定理：可以把一个数X唯一分解成若干素数相乘形式：

X = …

### 斐波那契数列性质

**性质1**  Fibonacci数列的通项公式:

（n≥1）

**性质2**  Fibonacci数列的前n项和：

**性质3**  Fibonacci数列的奇数项和：

**性质4** Fibonacci数列的前n项平方和: 

**性质5**  Fibonacci数列的相邻项乘积之和：

**性质6** 若连分数，

那么

**性质7 **

**性质8 **

**性质9** 

**性质10** 

**性质11**

**性质12** gcd() =

### 欧几里德算法（辗转相除法）

求最大公约数：

int gcd(int a,int b)

{

return b==0 ? a : gcd(b,a%b);

}

对于求最小公倍数有一公式：

gcd(a,b)\*lcm(a,b)=a\*b lcm(a,b)为a,b最大公倍数

所以lcm(a,b) = a / gcd(a,b) \* b (先除后乘，保证不会溢出)

### Eratosthenes筛法

int m = sqrt(n+0.5);

memset(vis,0,sizeof(vis));

for(int i=2;i<=m;i++) if(!vis[i])

for(int j=i\*i;j<=n;j+=i) vis[j]=1;

能快速得到以内所有的素数

### 扩展欧几里德算法

求：ax + by = gcd(a,b) 的解

void gcd(int a,int b,int& d,int& x,int & y)

{

if(!b) {d = a; x = 1; y = 0; }

else {gcd(b, a%b, d, y, x); y -= x\*(a/b); }

}

假设当前我们要处理的是求出 a 和 b的最大公约数，并求出 x 和 y 使得 a\*x + b\*y= gcd ，而我们已经求出了下一个状态：b 和 a%b 的最大公约数，并且求出了一组x1 和y1 使得： b\*x1 + (a%b)\*y1 = gcd ， 那么这两个相邻的状态之间是否存在一种关系呢？

    我们知道： a%b = a - (a/b)\*b（这里的 “/” 指的是整除，例如 5/2=2 , 1/3=0），那么，我们可以进一步得到：

        gcd = b\*x1 + (a-(a/b)\*b)\*y1

            = b\*x1 + a\*y1 – (a/b)\*b\*y1

            = a\*y1 + b\*(x1 – a/b\*y1)

    对比之前我们的状态：求一组 x 和 y 使得：a\*x + b\*y = gcd ，是否发现了什么？

    这里：

        x = y1

        y = x1 – a/b\*y1

对该方程任意整数解为：(x0+kb’,y0-ka’) a’ = a/gcd(a,b) b’ = b/gcd(a,b)

对于方程 ax+by=c 当c是gcd(a,b)倍数时有整数解，否则无整数解。

扩展欧几里德算法（ax+by=c）的最小正整数解：

先求出ax+by=gcd(a,b)的解x0,y0

令d=gcd(a,b) x = x0\*c/d g = b/d

最小正整数解为： x = (x%g+g)%g

以codeforces724C为例：

题意：n\*m的矩形内有k个点，四周有围墙围起来。从(0,0)45度发射小球，速度为√2每次遇到墙都正常反弹，直到射到顶点被吸收。问每个点第一次被经过的时刻。

思想：把矩形对称展开，最后小球在横纵坐标均为maxx=mn/gcd(m,n)处被吸收。   
原坐标为(x,y)的小球经过轴对称展开，其坐标为(2kn±x,2sm±y),k,s为整数.要使得在吸收前经过点，则坐标必须在线段(0, 0)到(maxx, mxx)之间。   
即要解方程2kn±x=2sm±y，求为正最小的2kn±x。利用扩展欧几里得解方程。

代码：

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

ll n,m,maxx;

void ex\_gcd(ll a,ll b,ll &d,ll &x,ll &y)

{

if(!b){d=a;x=1;y=0;}

else{ex\_gcd(b,a%b,d,y,x);y -= x\*(a/b);}

}

ll solve(ll x,ll y)

{

ll k,s,g;

ex\_gcd(2\*n,-2\*m,g,k,s);

if((y-x)%g) return maxx+1;

else

{

ll mod = (-2\*m)/g;

if(mod<0) mod = -mod;

k \*= (y-x)/g;

k = (k%mod+mod)%mod;

ll time=2\*k\*n+x;

if(time<0||time>maxx) return maxx+1;

return time;

}

}

ll answer(ll x,ll y)

{

ll g = \_\_gcd(n,m);

maxx = n / g \* m;

ll ans = maxx + 1;

ans = min(ans,solve(-x,-y));

ans = min(ans,solve(-x,y));

ans = min(ans,solve(x,-y));

ans = min(ans,solve(x,y));

if(ans==maxx+1) return -1;

else return ans;

}

int main()

{

ll x,y,k;

while(~scanf("%I64d%I64d%I64d",&n,&m,&k))

{

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

{

scanf("%I64d%I64d",&x,&y);

printf("%I64d\n",answer(x,y));

}

}

return 0;

}

### 大整数取模

n mod m n<=10^100 m<=10^9

代码：

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

int len = strlen(n);

int ans = 0;

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

ans = (int)(((long long)ans\*10+n[i]-'0')%m);

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

### 快速幂取模

求a^b%c

代码：

int ans = 1;

a %= c;

while(b>0)

{

if(b%2==1) ans = (ans \*a)%c;

b /= 2;

a = (a\*a)%c;

}

分治用递归也可实现

代码：

int pow\_mod(int a,int b,int c)

{

if(n==0) return 1;

int x = pow\_mod(a,b/2,c);

long long ans = (long long)x \* x % c;

if(b%2==1) ans = ans \* a % c;

return (int)ans;

}

### 矩阵快速幂

矩阵快速幂的原理和快速幂一样，只不过把数的乘法换成矩阵的乘法。

矩阵的乘法（代码）：

int ans[][N];

void multi(int a[][N],int b[][N],int n)

{

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

{

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

{

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

{

ans[i][j]=a[i][k]\*b[k][j];

}

}

}

}

这种可以在第二重for判断if(a[i][k]==0)continue;对于矩阵有较多0的有一定效果。

时间复杂度是O(n^3)。

矩阵快速幂模板：

int tmp[][N];

void multi(int a[][N],int b[][N],int n)

{

memset(tmp,0,sizeof(tmp));

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

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

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

{

tmp[i][j]+=a[i][k]\*b[k][j];

tmp[i][j] %= mod;

}

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

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

a[i][j]=tmp[i][j];

}

int ans[][N];

void pow\_mod(int a[][N],int b,int n)

{

memset(ans,0,sizeof(ans));

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

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

if(i==j) ans[i][j]=1;

while(b)

{

if(b&1) multi(ans,a,n);

b>>=1;

multi(a,a,n);

}

}

在算法竞赛中一般只会用到方阵，只要遇到“把一个向量v变成另一个向量v’，并且v’的每一个分量都是v各个分量的线性组合”的情况，可以考虑用矩阵。

这类题目难点主要在于寻找递推关系，构造矩阵求解。

例hdu1757：题意：求 f(k)%m k<2\*10^9 , m < 10^5

当x<10时 f(x)=x; 当x>=10时 f(x)=a0\*f(x-1)+……+a9\*f(x-10)

递推式已给出，该题难点主要在于构造矩阵，构造如下：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| fk | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

=

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | a8 | a9 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

\*

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| fk-1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-6 | 0\* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fk-10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

所以只需构造出上诉两个矩阵，求出a0那个矩阵的k-9次方再乘以f(0)那个矩阵即可，注意乘的过程中取余。

代码：

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

using namespace std;

typedef long long ll;

int n,mod;

ll tmp[10][10],ans[10][10],sz[10][10],cmp[10][10];

void matrix\_multi(ll a[][10],ll b[][10])

{

memset(cmp,0,sizeof(cmp));

for(int i=0;i<10;i++)

{

for(int k=0;k<10;k++)

{

if(a[i][k]==0) continue;

for(int j=0;j<10;j++)

{

cmp[i][j]+=a[i][k]\*b[k][j];

cmp[i][j]%=mod;

}

}

}

for(int i=0;i<10;i++)

for(int j=0;j<10;j++)

a[i][j]=cmp[i][j];

}

void pow\_mod(ll a[][10])

{

memset(ans,0,sizeof(ans));

for(int i=0;i<10;i++)

for(int j=0;j<10;j++)

if(i==j) ans[i][j]=1;

int b=n-9;

while(b)

{

if(b&1) matrix\_multi(ans,a);

b>>=1;

matrix\_multi(a,a);

}

}

int main()

{

while(~scanf("%d%d",&n,&mod))

{

memset(tmp,0,sizeof(tmp));

memset(sz,0,sizeof(sz));

for(int i=0;i<10;i++) scanf("%lld",&tmp[0][i]);

tmp[1][0]=tmp[2][1]=tmp[3][2]=tmp[4][3]=tmp[5][4]=tmp[6][5]=tmp[7][6]=tmp[8][7]=tmp[9][8]=1;

for(int i=0;i<10;i++) sz[i][0]=9-i;

pow\_mod(tmp);

matrix\_multi(ans,sz);

printf("%lld\n",ans[0][0]);

}

return 0;

}

### 逆元

求 %mod的方法：

1.乘法逆元：(a/b)%mod=a\*(b^(mod-2)) a,mod互为素数。

有一种不需mod为素数的：

(a/b)%mod=a % (mod\*b) / b 通用

2.当n和m比较大，mod是素数且比较小的时候（10^5左右），通过Lucas定理计算：

Lucas定理：A、B是非负整数，p是质数。A B写成p进制：A=a[n]a[n-1]…a[0]，B=b[n]b[n-1]…b[0]。  
则组合数C(A,B)与C(a[n],b[n])*C(a[n-1],b[n-1])*…\*C(a[0],b[0]) mod p同余  
即：Lucas(n,m,p)=C(n%p,m%p)\*Lucas(n/p,m/p,p)

以hdu5894为例：

题意：给你n个围成一个圆的桌子，m个人，相邻两个人之间相差至少K个桌子。问有多少种坐法。

题解：首先确定第一个人的座位，从n个座位中选择一个，然后确定出符合条件的k\*m个座位。最后剩下n-1-k\*m个座位，从中选出m-1个座位坐人。总数sum=n\*C(n-1-k\*m,m-1);

由于有m个重合，因此要sum=sum/m;例如：（2，4，7），（4，7，2），（7，2，4）是一样的。

计算组合数和sum/m时可以使用乘法逆元：(a/b)%mod=a\*(b^(mod-2)) mod为素数。

代码：

#include<cstdio>

using namespace std;

typedef long long ll;

const int mod=1e9+7;

ll qmod(ll n,ll p) //快速幂取模

{

ll result=1;

while(p>0)

{

if(p%2==1)

result=(result\*n)%mod;

p/=2;

n=(n\*n)%mod;

}

return result;

}

ll c(ll n,ll m)

{

if(n<m)

return 0;

ll ans=1;

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

ans=ans\*((n-m+i)\*qmod(i,mod-2)%mod)%mod; //利用乘法逆元

return ans;

}

main()

{

ll t,n,m,k;

scanf("%lld",&t);

while(t--)

{

scanf("%lld%lld%lld",&n,&m,&k);

if(m==1)

printf("%lld\n",n);

else

printf("%lld\n",(n\*c(n-k\*m-1,m-1)%mod)\*qmod(m,mod-2)%mod);

}

}

# 图论

## 最短路

### Dijkstra算法

Dijkstra算法适用于边权为正的情况，用于计算正权图上的单源最短路（Single-Source Shortest Paths,SSSP），即从单个源点出发，到所有结点的最短路。该算法同时适用于有向图和无向图。

时间复杂度为O(n^2)

算法思想：

首先初始化dist为INF=0x3f3f3f3f，dist[1]=0，然后循环n次，在所有未标号的结点中，选出dist值最小的结点x，给结点x标记，对从x结点出发的所有边（x,y）更新dist[y]。 该算法也能很方便的打印出结点1到所有结点的最短路本身，只需在更新dist[y]时用path数组维护从最短路到该结点的父结点的下标即可。

以hdu2544为例：

代码：

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

using namespace std;

const int INF=0x3f3f3f3f;

int n,m,dist[105],sz[105][105],vis[105];

int dijkstra()

{

int tmp,x;

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

{

tmp=INF;

for(int j=1;j<=n;j++) if(!vis[j] && dist[j]<tmp) tmp=dist[x=j];

vis[x]=1;

for(int j=1;j<=n;j++) dist[j]=min(dist[j],dist[x]+sz[x][j]);

}

return dist[n];

}

int main()

{

int a,b,c;

while(~scanf("%d%d",&n,&m)&&n&&m)

{

memset(sz,INF,sizeof(sz));

memset(dist,INF,sizeof(dist));

memset(vis,0,sizeof(vis));

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

{

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

sz[a][b]=sz[b][a]=c;

}

dist[1]=0;

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

}

return 0;

}

优先队列优化

以hdu2066为例

代码

#include <bits/stdc++.h>

**using** **namespace** std;

const int INF = 0x3f3f3f3f;

const int maxn = 1e3+5;

int t,s,d;

struct Edge{

int to,len; //存储边信息，to是到的点，len是时间

};

vector<Edge> edge[maxn]; //edge[i]记录与i点相连的边

//把i点和dist[i]打包放入优先队列。

//这里要注意pair是按照第一个元素的大小排序，如果相同才按照第二个，所以我们要把dist[i]包装在第一个元素上。

**typedef** pair<int,int> P;

int dist[maxn]; //dist[i]记录从源点到i点的最短时间

void init() //初始化

{

**for**(int i=0;i<maxn;i++) edge[i].clear();

memset(dist,INF,**sizeof**(dist));

}

void Dijkstra()

{

priority\_queue<P,vector<P>,greater<P> >pq;

P p;

Edge tmp;

dist[0]=0; //源点dist置0

pq.push(P(0,0)); //源点加入队列

**while**(!pq.empty())

{

p = pq.top(),pq.pop();

int u = p.second; //得到当前节点

**for**(int i=0;i<edge[u].size();i++) //遍历与当前节点相连的节点

{

tmp = edge[u][i];

**if**(dist[tmp.to]>dist[u]+tmp.len)

{

dist[tmp.to] = dist[u]+tmp.len; //更新

pq.push(P(dist[tmp.to],tmp.to));

}

}

}

}

int main()

{

**while**(~scanf("%d%d%d",&t,&s,&d))

{

init();

int a,b,c;

Edge tmp;

**while**(t--)

{

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

//建边

tmp.to = b,tmp.len = c;

edge[a].push\_back(tmp);

tmp.to = a,tmp.len = c;

edge[b].push\_back(tmp);

}

**while**(s--)

{

scanf("%d",&a);

//源点（家）和车站相连

tmp.to = a,tmp.len = 0;

edge[0].push\_back(tmp);

tmp.to = 0,tmp.len = 0;

edge[a].push\_back(tmp);

}

Dijkstra();

int ans = INF;

**while**(d--)

{

scanf("%d",&a);

**if**(dist[a]<ans) ans = dist[a]; //得到所有终点的最短时间

}

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

}

**return** 0;

}

### SPFA

SPFA（Shortest Path Faster Algorithm），是西南交通大学段凡丁于 1994 年发表的，其在Bellman-ford算法的基础上加上一个队列优化，减少了冗余的松弛操作，是一种高效的最短路算法。

SPFA和Bellman-ford一般都是用来处理带负权的最短路问题（Dijkstra不能处理带负权回路的最短路），而SPFA是在Bellman-ford的基础上优化的，所以我先介绍一下Bellman-ford算法的思想。

Bellman-ford思想   
设顶点数为 n，边数为 m，源点为 source，数组dist[i]记录从源点 source 到顶点 i 的最短路径，除了dist[source]初始化为 0 外，其它dist[]皆初始化为 MAX。   
然后循环 n-1 次，每次循环遍历所有的边e（u,v），然后进行松弛操作（就是dist[u] + w(u, v) < dist[v]，则使 dist[v] = dist[u] + w(u, v)，其中 w(u, v) 为边 e(u, v) 的权值），n-1次循环后从源点到其它所有点的最短路径就确定了。   
那么怎么判断负权？想一想，本来经过n-1次循环，所有点的最短路径应该已经确定。但是如果存在负权回路，那么可以不断在负权回路那里循环，不断减小（其实就是最短路可以无限小）。所以我们进行第n次循环，如果还能进行松弛操作，说明带负权回路，不能说明不带。

SPFA思想   
SPFA就是用一个队列来优化Bellman-ford。   
设立一个队列用来保存待优化的节点，优化时每次取出队首节点 u，并且用 u 点当前的最短路径估计值dist[u]对与 u 点邻接的顶点 v 进行松弛操作，如果 v 点的最短路径估计值dist[v]可以更小，且 v 点不在当前的队列中，就将 v 点放入队尾。这样不断从队列中取出顶点来进行松弛操作，直至队列空为止。   
SPFA怎么判断负权回路？如果某个点进入队列的次数大于等于 n，则存在负权回路，其中 n 为图的顶点数。因为即使一个点与其它所有点都存在路径，每次都能放入队列，最多也就是n-1次，多的话肯定是存在负权回路使它能不断进入负权回路。

对于最短路问题，个人觉得如果不带负权的话最好用邻接表+最小堆优化的Dijkstra（边数少也可考虑用SPFA），复杂度适宜，O（（n+m）logn），也稳定。如果带负权回路就用SPFA，时间复杂度（最好O(m)，最差O(n\*m)），不稳定。

以hdu4076为例

代码

#include <bits/stdc++.h>

**using** **namespace** std;

#define INF 0x3f3f3f3f

int w,h;

int dis[4][2]={-1,0,0,-1,1,0,0,1};//上下左右走

struct node{

int x,y;

};

int matrix[35][35],dist[35][35],vis[35][35],nx[35][35],ny[35][35];

//matrix保存地图状态，0是草，-1是墓碑，1是洞

//dist[i][j]保存到i，j的最短时间，vis保存访问状态

//nx，ny保存洞的终点坐标

int t[35][35],inqueue\_num[35][35];

//t保存洞的用时，inqueue\_num保存入队次数

bool SPFA()

{

bool flag=**true**;

node p,next;

queue<node>q;

//起点入队

p.x=p.y=0;

dist[0][0]=0;

vis[0][0]=1;

inqueue\_num[0][0]++;

q.push(p);

**while**(!q.empty())

{

p = q.front(),q.pop();

vis[p.x][p.y]=0;

**if**(p.x==w-1&&p.y==h-1) **continue**; //到达终点之后就不用再入队松弛了

**if**(matrix[p.x][p.y]) //有洞

{

next.x = nx[p.x][p.y];

next.y = ny[p.x][p.y];

**if**(dist[next.x][next.y]>dist[p.x][p.y]+t[p.x][p.y]) //如果能更新

{

dist[next.x][next.y]=dist[p.x][p.y]+t[p.x][p.y]; //松弛

**if**(!vis[next.x][next.y])

{

q.push(next);

inqueue\_num[next.x][next.y]++;

**if**(inqueue\_num[next.x][next.y]>=w\*h) **return** flag=**false**; //存在负权回路

vis[next.x][next.y]=1;

}

}

**continue**; //有洞必须到洞的终点

}

**for**(int i=0;i<4;i++) //上下左右遍历

{

next.x = p.x+dis[i][0];

next.y = p.y+dis[i][1];

//超出地图范围或有墓碑

**if**(next.x<0 || next.x>w-1 || next.y<0 || next.y>h-1 || matrix[next.x][next.y]==-1) **continue**;

**if**(dist[next.x][next.y]>dist[p.x][p.y]+1)

{

dist[next.x][next.y]=dist[p.x][p.y]+1;

**if**(!vis[next.x][next.y])

{

q.push(next);

inqueue\_num[next.x][next.y]++;

**if**(inqueue\_num[next.x][next.y]>=w\*h) **return** flag=**false**;

vis[next.x][next.y]=1;

}

}

}

}

**return** flag;

}

int main()

{

**while**(~scanf("%d%d",&w,&h)&&w!=0&&h!=0)

{

int num,x1,x2,y1,y2,tmp;

memset(matrix,0,**sizeof**(matrix));

memset(dist,INF,**sizeof**(dist));

memset(vis,0,**sizeof**(vis));

memset(inqueue\_num,0,**sizeof**(inqueue\_num));

scanf("%d",&num);

**while**(num--)

{

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

matrix[x1][y1] = -1;

}

scanf("%d",&num);

**while**(num--)

{

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

matrix[x1][y1]=1;

nx[x1][y1]=x2,ny[x1][y1]=y2;

t[x1][y1]=tmp;

}

**if**(SPFA())

{

**if**(dist[w-1][h-1]==INF) printf("Impossible\n"); //不能到达终点

**else** printf("%d\n",dist[w-1][h-1]);

}

**else** printf("Never\n"); //有负权回路

}

**return** 0;

}

### Floyd算法

如果需要求出没两点之间的最短路，不必调用n次Dijkstra（边权均为正）或者Bellman-ford（有负权），用Floyd即可。

算法思想：

在i，j结点之间加入结点V1，比较（i,V1,j）和（i,j）的路劲长度，更新i，j的最短路作为i到j的且中间结点数不大于1的最短路径。就这样更新n次即可得到i到j的最短路径。

调用之前先初始化dist[i][i]=0，其它dist值为INF。

模板：

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

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

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

dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]);

在有向图中，有时不关心路径长度，而只关心每两点之间是否有通路，用1，0表示连通，不连通。这样只需将dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])改为dist[i][j]=dist[i][j] || (dist[i][k] && dist[k][j])即可。

依旧以hdu2544为例：

代码：

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

using namespace std;

const int INF=0x3f3f3f3f;

int n,m,dist[105][105];

int floyd()

{

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

{

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

{

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

{

dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]);

}

}

}

return dist[1][n];

}

int main()

{

int a,b,c;

while(~scanf("%d%d",&n,&m)&&n&&m)

{

memset(dist,INF,sizeof(dist));

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

{

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

dist[a][b]=dist[b][a]=c;

}

for(int i=1;i<=n;i++) dist[i][i]=0;

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

}

return 0;

}

## 最小生成树

### Prim算法

时间复杂度为O(n^2,n为点数)。

算法思想：

先任意找一个点标记，然后每次找一条最短的两端分别为标记和未标记的边加进来，再把未标记的点标记上。即每次加入一条合法的最短的边，每次扩展一个点由未标记为已标记，直至扩展为N个点。

以hdu1233为例：

题意：

N个村庄，n\*(n-1)/2条路，使村庄全部连通并使路的长度最短，求最短路长，n<100。

代码：

#include<stdio.h>

#include<string.h>

#define MAX 100000000

int map[101][101],visit[101]; /\*map[x][y]记录x点到y点的距离，visit[],记录点是否标记。\*/

long long prim(int n);

int main()

{

int n;

while(scanf("%d",&n)&&n)

{

int m = n\*(n-1)/2;

int i,j,a,b,c;

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

{

for(j=1;j<=n;j++)

{

if(i==j) map[i][j] = 0;

else map[i][j]=map[j][i]=MAX;

}

}

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

{

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

map[a][b]=map[b][a]=c; /\*初始完map[][]\*/

}

long long ans = prim(n);

printf("%lld\n",ans);

}

return 0;

}

long long prim(int n)

{

long long ans = 0;

int minimal[105],i,j,min,k;

memset(minimal,0,sizeof(minimal));

memset(visit,0,sizeof(visit));

visit[1] = 1; /\*从1开始，标记1\*/

for(i=1;i<=n;i++) minimal[i] = map[1][i]; /\*得到从开始点到其它个点的距离。\*/

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

{

min = MAX;

for(j=1;j<=n;j++)

{

if(minimal[j]<min&&visit[j]==0)

{

min = minimal[j]; /\*得到其它未标记点到标记点的最短距离。\*/

k = j; /\*得到最短距离的未标记点\*/

}

}

ans += min;

visit[k] = 1; /\*把未标记点标记\*/

for(j=1;j<=n;j++)

{

if(minimal[j]>map[k][j]&&visit[j]==0&&k!=j)

{

minimal[j] = map[k][j]; /\*重新初始其它未标记点到标记点的距离，若其它未标记点到新标记点k的距离更短则替换。\*/

}

}

}

return ans;

}

### Kruskal算法

时间复杂度为O(mlogm,m为边数)。

算法思想：

Kruskal是通过一个贪心的想法：每次取剩下的边权最小的边，如果加上这条边以后图中出现一个环，则破坏生成树的性质，就不选这条边。依次进行直到整张图出现一颗生成树为止。

依旧以hdu1233为例：

#include<stdio.h>

#define N 105

int set[N];

int comp(const void \*a,const void \*b);

long long as(int sz[][3],int n,int a);

int fine(int a);

int main()

{

int n;

while(scanf("%d",&n)&&n)

{

int m = n\*(n-1)/2;

int sz[m][3],i;

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

{

scanf("%d%d%d",&sz[i][0],&sz[i][1],&sz[i][2]);

}

for(i=0;i<N;i++) set[i] = i;

qsort(sz,m,sizeof(int)\*3,comp);

long long ans = as(sz,m,n);

printf("%lld\n",ans);

}

return 0;

}

int comp(const void \*a,const void \*b)

{

return ((int \*)a)[2]-((int \*)b)[2];

}

int fine(int a)

{

return a==set[a] ? a : (set[a]=fine(set[a]));

}

long long as(int sz[][3],int n,int a)

{

long long ans = 0;

int i;

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

{

if(fine(sz[i][0])!=fine(sz[i][1]))

{

set[fine(sz[i][1])] = fine(sz[i][0]);

ans += sz[i][2];

a--;

if(a==1) break;

}

else continue;

}

return ans;

}

不过hdu里的编译器好像不支持qsort二维数组排序，把qsort二维数组排序写成自定义的函数即可过。

void qsort(int sz[][3],int a) //按边排序

{

int i,j,x,exchange;

for(i=0;i<a-1;i++)

{

for(j=i+1;j<a;j++)

{

if(sz[i][2]>sz[j][2])

{

for(x=0;x<3;x++)

{

exchange = sz[i][x];

sz[i][x] = sz[j][x];

sz[j][x] = exchange;

}

}

}

}

}

# 动态规划(DP)

## 资源型DP(背包)

### 01背包

01背包（ZeroOnePack）: 有N件物品和一个容量为V的背包。（每种物品均只有一件）第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使价值总和最大。

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用dp[i][v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。

则其状态转移方程便是：

**dp[i][v]=max(dp[i-1][v],dp[i-1][v-c[i]]+w[i])**

我们通过i从1到n的循环来依次表示前i件物品存入的状态。即：for i=1..N

二重循环用逆序。即：for j=v..0

**for i=1..N**

**for v=V..c[i]**

**dp[v]=max{dp[v],dp[v-c[i]]+w[i]};**

### 完全背包

完全背包(CompletePack): 有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

状态转移方程：

**dp[i][v]=max(dp[i-1][v-k\*c[i]]+k\*w[i],dp[i-1][v])  (0<=k\*c[i]<=v)**

**for i=1..N**

**for v=0..V**

**dp[v]=max{dp[v],dp[v-c[i]]+w[i]}**

### 多重背包

多重背包(MultiplePack): 有N种物品和一个容量为V的背包。第i种物品最多有n[i]件可用，每件费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

状态转移方程：

**dp[i][v]=max(dp[i-1][v-k\*c[i]]+k\*w[i],dp[i-1][v]) (0<=k<=n[i])**

### 分组背包

分组背包(PacketPack): 有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

设dp[k][v]表示前k组物品花费费用v能取得的最大权值

状态转移方程：

**dp[k][v]=max{dp[k-1][v],dp[k-1][v-c[i]]+w[i]|物品i属于组k}**

**for 所有的组k  
    for v=V..0  
        for 所有的i属于组k  
            f[v]=max{f[v],f[v-c[i]]+w[i]}**

## 最长公共子序列（LCS）

子序列：

在数学中，某个序列的子序列是从最初序列通过去除某些元素但不破坏余下元素的相对位置（在前或在后）而形成的新序列。比如：ABCDEFGHI的一个子序列是ADFGI。

最长公共子序列：

理解了子序列就能简单的理解最长公共子序列。就是对于两个序列，有一个序列三满足既是序列1的子序列也是序列2的子序列而且是满足这个条件的子序列中最长的一个，它就是序列1，2的最长公共子序列。   
比如：BDCABA和ABCBDAB的最长公共子序列就是BCBA。

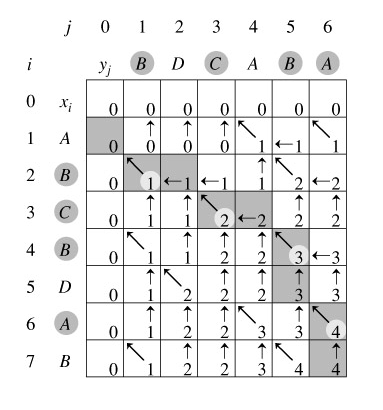
设：str1=，str2=，它们的LCS为str=

我们来看看怎么推出状态转移方程   
首先，不难得出如下结论：

1. 如果=，那么==且是和的最长公共子序列。（感觉运用了分治的思想）   
2. 如果 != ，那么是MAX（LCS（，），LCS（，））那一个的最长公共子序列。

由上述结论，我们设dp[i][j]为和的最长公共子序列的长度。所以：

以BDCABA和ABCBDAB为例，我们看看它的递推过程



代码模板

int LCS(char \*str1, char \*str2)

{

int len1=strlen(str1);

int len2=strlen(str2);

memset(dp,0,**sizeof**(dp);

**for**(int i=1;i<=len1;i++)

{

**for**(int j=1;j<=len2;j++)

{

**if**(str1[i-1]==str2[j-1]) dp[i][j] = dp[i-1][j-1]+1;

**else** dp[i][j] = max(dp[i-1][j],dp[i][j-1]);

}

}

**return** dp[len1][len2];

}

给出两个字符串，求出其最长公共子序列及其长度。

#include <bits/stdc++.h>

**using** **namespace** std;

#define time\_ (printf("%.6f\n", double(clock())/CLOCKS\_PER\_SEC))

**typedef** long long ll;

const int maxn = 1e3+5;

int dp[maxn][maxn];

//记录dp[i][j]是从哪递推过来的

//-1是dp[i-1][j-1],0是dp[i][j-1],1是dp[i-1][j]

int st[maxn][maxn];

char str1[maxn],str2[maxn];

void PrintLCS(int i,int j)

{

**if**(i==0 || j==0) **return**;

**if**(st[i][j]==-1)

{

PrintLCS(i-1,j-1);

printf("%c",str1[i-1]);

}

**else** **if**(st[i][j]==0)

{

PrintLCS(i,j-1);

}

**else**

{

PrintLCS(i-1,j);

}

}

void LCS()

{

int len1=strlen(str1);

int len2=strlen(str2);

memset(dp,0,**sizeof**(dp));

**for**(int i=1;i<=len1;i++)

{

**for**(int j=1;j<=len2;j++)

{

**if**(str1[i-1]==str2[j-1]) //这里是因为dp下标是从1开始，str1和str2的下标是从0开始

{

dp[i][j] = dp[i-1][j-1]+1;

st[i][j] = -1;

}

**else** **if**(dp[i][j-1]>dp[i-1][j])

{

dp[i][j] = dp[i][j-1];

st[i][j] = 0;

}

**else**

{

dp[i][j] = dp[i-1][j];

st[i][j] = 1;

}

}

}

printf("%d\n",dp[len1][len2]);

PrintLCS(len1,len2);

printf("\n");

}

int main()

{

gets(str1);

gets(str2);

LCS();

**return** 0;

}