ACM模板

1. 常见基础错误
2. 各种智障写法：
3. 没开long long
4. 精度不够(Java大法好)
5. 没有初始化
6. 浮点数判等
7. 读错题
8. 没有注意数据范围(数组开小或超时算法)
9. 没有特判某种情况 (边界, 0, -1, 特殊值等。每次交题之前一定要验证边 界值有没有溢出!!!)
10. 使用了cin/cout造成超时(尽量使用scanf)
11. 当a\*b/c时a\*b造成了溢出,应该改成a/c\*b

(二)各种编译器的提交问题,包括 ll ,I64d,f,lf 等。

1. 语言相关

(一)STL 的用法 .................................................... 8

1. Vector
2. Stack
3. Queue
4. priority\_queue
5. deque
6. Set
7. Map
8. String
9. Permutation
10. Bitset

(二)Java 大数 ................................................. 15

1. 基础知识
2. 枚举
3. 贪心
4. 模拟
5. 排序 ................................................... 17
6. 分治 ................................................... 20
7. 最近点对问题
8. 点分治
9. BFS
10. DFS
11. 二分
12. 三分 ................................................... 25
13. 双指针扫描 + 尺取法 ............................ 26
14. 动态规划
15. DP 基础
16. 基础 DP 问题
17. 背包模型 .............................. 28
18. 最长公共子序列(LCS) ........................... 31
19. 最长公共子串 .................................... 32
20. 最长上升子序列(LIS) ............................... 32
21. 括号序列模型 ....................................... 36
22. 递推模型
23. 线段覆盖问题
24. 连续段划分模型
25. 树形 DP
26. 状压 DP .................................................. 37
27. DP 优化
28. 区间 DP .................................................. 39
29. 数据结构(包含 RMQ 问题)
30. 并查集 .................................................... 43
31. 树状数组 .................................................... 44

1、前缀和

1. 线段树(Segtree)

1.基本更新和查询操作 .................................. 49

2.矩形面积交、面积并 .................................. 51

1. 字典树 ...................................................... 56
2. ST 表 ....................................................... 58
3. Tarjan(离线算法)
4. 链式前向星
5. 图论
6. 最短路 ........................................................ 59
7. Dijkstra
8. Dijkstra + heap
9. SPFA
10. Floyed(多源最短路)
11. 差分约束
12. 次短路与第 K 短路 ............................................ 66
13. 搜索

1、A\*搜索

2、ID A\*搜索

3、记忆化搜索

1. 最近公共祖先(LCA) ............................................ 68
2. 最小生成树(MST) + 最小树形图 ......................... 70
3. Prim(没什么卵用)
4. Kruskal

3. 朱刘算法

1. 树上差分
2. 树状倍增
3. 二分图 ........................................................... 74
4. 二分图最大匹配(匈牙利算法)
5. 二分图最小顶点覆盖数
6. 有向无环图(DAG)的最小路径覆盖数
7. 二分图的最大独立集数

5.判断一个图是否是二分图(交叉染色法)

1. 网络流
2. Dinic 算法
3. 最小费用最大流
4. 最小割

(十)树链剖分 ....................................................... 78

(十一)拓扑排序 .................................................... 82

1. 数论
2. 中国剩余定理 ................................................. 84
3. 二次剩余
4. 欧几里得 ...................................................... 86
5. 费马小定理
6. 扩展欧几里得
7. 最大公约数和最小公倍数 .................................. 88
8. 素数筛 ........................................................... 89
9. 埃氏筛法
10. 欧拉筛
11. 区间筛法
12. 米勒罗宾素性测试(miller-rabin)
13. 欧拉函数 ....................................................... 91
14. 打表法
15. 直接求法
16. 原根
17. 欧拉函数降幂 ............................................... 92
18. 分解质因数
19. 枚举因子
20. 素数筛
21. Pollard-rho 分解质因数
22. 离散对数
23. Baby Step Giant Step
24. Ex-baby step giant step
25. 原根
26. 求原根
27. 原根应用
28. K 次方根
29. 快速数论变换(NTT)
30. 快速幂
31. 快速幂 ........................................................ 94
32. 快速乘 ........................................................ 94
33. 矩阵快速幂 ................................................... 94
34. 康托展开................................................... 95
35. 快速傅氏变换(FFT)
36. 高斯消元
37. 普通高斯消元
38. 二进制高斯消元(xor 高斯消元)
39. 生成树计数
40. 度数矩阵
41. 基尔霍夫矩阵(拉普拉斯矩阵)
42. 基尔霍夫定理(Matrix Tree)定理

(十五)矩阵行列式

(十六)组合数 .................................................. 97

1. 打表法
2. Lucas大组合数
3. 预处理阶乘逆元法

(十七)全错排公式

1. 博弈论
2. 巴什博弈 ........................................................ 100
3. 威佐夫博弈 ...................................................... 100
4. Nim 博弈 ........................................................ 101
5. SG 函数
6. 敌对搜索 .......................................................... 102
7. 0-1 矩阵
8. 0-1 树
9. 字符串
10. KMP ................................................................... 103
11. AC自动机 .......................................................... 105
12. 后缀数组
13. 滚动哈希(Rabin-Karp算法) ............................... 108
14. 其他(杂七杂八的结论和函数).........................................109
15. 字符串和数字互转
16. 字符串和字符数组互转
17. 整除的性质
18. 各种距离区别
19. 位运算的应用与技巧
20. 二分查找及其变种

Ubuntu Vim 基本配置 ................................................. 116

热身赛前测试

一、语言相关

(一)STL 的用法

1. vector

(1) 头文件#include<vector>.

(2) 创建 vector 对象,vector<int> vec;

声明一个初始大小为n的int向量 vector<int> vec(n);

声明一个初始大小为n且值都是m的向量vector<int> vec(n, m);

声明并用tmp向量初始化vec向量vector<int> vec(tmp);

用向量vec的前n个值初始化tmp vector<int> tmp(vec.begin(), vec.begin() + n);

将arr数组的元素用于初始化vec向量int arr[5] = {1, 2, 3, 4, 5}; vector<int> vec(arr, arr + 5); 也可以写为匿名向量vector<int>{1,2,3,4,5}

(3)尾部插入数字:vec.push\_back(a);

(4)尾部删除数字:vec.pop\_back();

(5)使用下标访问元素,cout<<vec[0]<<endl;记住下标是从0开始的。

(6)使用迭代器访问元素。

vector<int>::iterator it;for(it=vec.begin();it!=vec.end();it++)

cout<<\*it<<endl;

(7)插入元素: vec.insert(vec.begin()+i,a);在第 i+1 个元素前面插入 a;

(8)删除元素: vec.erase(vec.begin()+2);删除第 3 个元素

vec.erase(vec.begin()+i,vec.end()+j);删除区间[i,j-1];区间从0 开始

(9)向量大小:vec.size();

(10)清空:vec.clear();

(11)排序:sort(vec.begin(),vec.end())

(12)逆序:reverse(vec.begin(),vec.end())

(13)查询值:find(vec.begin(),vec.end(),x)

**可以用下面的方式删除vector中所有为n的数**

v.erase(remove(v.begin(),v.end(),n),v.end());

1. stack

1.声明一个stack

stack<int> s1;

stack<string> s2;

2.stack中的操作

stack<int> s;

s.push(x) 无返回值，将元素x压栈

s.pop(); 退栈，无返回值

s.top(); 取栈顶元素，返回栈顶元素

s.empty(); 判断栈是否为空，如果是空，返回1，否则返回0

s.size(); 返回栈中元素的个数

在栈中没有提供清空操作的函数，但是可以间接地实现清空栈，

while(!s.empty()){

s.pop();

}

1. queue

push(x) 将x压入队列的末端

pop() 弹出队列的第一个元素(队顶元素)，注意此函数并不返回任何值

front() 返回第一个元素(队顶元素)

back() 返回最后被压入的元素(队尾元素)

empty() 当队列为空时，返回true

size() 返回队列的长度

1. priority\_queue

优先队列容器与队列一样，只能从队尾插入元素，从队首删除元素。但是它有一个特性，就是队列中最大的元素总是位于队首，所以出队时，并非按照先进先出的原则进行，而是将当前队列中最大的元素出队。这点类似于给队列里的元素进行了由大互小的顺序排序。元素的比较规则默认按元素值由大到小排序，可以重载“<”操作符来重新定义比较规则。

优先队列使用:

1、头文件是queue

2、priority\_queue<T>默认为大根堆,除非对运算符 < 进行重载

如：

struct Node{

int x, y;

}node;

bool operator<( Node a, Node b){

if(a.x==b.x) return a.y>b.y;

return a.x>b.x;

}

3、priority\_queue<T,vector<T>,greater<T>>升序排列(小根堆),less<T>降序排列(大根堆)

4、priority\_queue<T,vector<T>,cmp> 自定义排序(set也可以写成这种形式,set<T,cmp>)

如：

struct Node{

int x, y;

}node;

struct cmp{

bool operator()(Node a,Node b){

if(a.x==b.x) return a.y>b.y;

return a.x>b.x;}

};

priority\_queue<Node,vector<Node>,cmp>q;

1. deque

(1)构造函数

deque():创建一个空deque

deque(int nSize):创建一个deque,元素个数为nSize

deque(int nSize,const T& t):创建一个deque,元素个数为nSize,且值均为t

deque(const deque &):复制构造函数

(2)增加函数

void push\_front(const T& x):双端队列头部增加一个元素X

void push\_back(const T& x):双端队列尾部增加一个元素x

iterator insert(iterator it,const T& x):双端队列中某一元素前增加一个元素x

void insert(iterator it,int n,const T& x):双端队列中某一元素前增加n个相同的元素x

void insert(iterator it,const\_iterator first,const\_iteratorlast):双端队列中某一元素前插入另一个相同类型向量的[forst,last)间的数据

(3)删除函数

Iterator erase(iterator it):删除双端队列中的某一个元素

Iterator erase(iterator first,iterator last):删除双端队列中[first,last）中的元素

void pop\_front():删除双端队列中最前一个元素

void pop\_back():删除双端队列中最后一个元素

void clear():清空双端队列中最后一个元素

(4)遍历函数

reference at(int pos):返回pos位置元素的引用

reference front():返回首元素的引用

reference back():返回尾元素的引用

iterator begin():返回向量头指针，指向第一个元素

iterator end():返回指向向量中最后一个元素下一个元素的指针（不包含在向量中）

reverse\_iterator rbegin():反向迭代器，指向最后一个元素

reverse\_iterator rend():反向迭代器，指向第一个元素的前一个元素

(5)判断函数

bool empty() const:向量是否为空，若true,则向量中无元素

(6)大小函数

Int size() const:返回向量中元素的个数

int max\_size() const:返回最大可允许的双端对了元素数量值

(7)其他函数

void swap(deque&):交换两个同类型向量的数据

void assign(int n,const T& x):向量中第n个元素的值设置为x

1. set

begin() 　 返回set容器的第一个元素

end() 　　　　 返回set容器的最后一个元素

clear() 　　 删除set容器中的所有的元素

empty() 　　　 判断set容器是否为空

max\_size() 　 返回set容器可能包含的元素最大个数

size() 　　　　 返回当前set容器中的元素个数

count() 用来查找set中某个某个键值出现的次数,0表示未出现过,

1表示出现过

erase(iterator) 删除定位器iterator指向的值

erase(first,second) 删除定位器first和second之间的值

erase(key\_value) 删除键值key\_value的值

find() 返回给定值值得定位器，如果没找到则返回end()。

insert(key\_value) 将key\_value插入到set中

inset(first,second) 将定位器first到second之间的元素插入到set中，返回值是void.

1. Map

map<T1,T2> mp; 构造函数

三种插入键值对的方法

mp.insert(pair<T1,T2>(a,b));

mp.insert(pair<T1,T2>::value\_type(a,b));

mp[a] = b;

Iter -> first 取key

Iter -> second 取value

count() 用来查找map中某个某个键值出现的次数,0表示未出现过,

1表示出现过

1. string

string s1; 默认构造函数，s1为空串

string s2(s1); 将s2初始化为s1的一个副本

string s3("valuee"); 将s3初始化一个字符串面值副本

string s4(n,'c'); 将s4 初始化为字符'c'的n个副本

cin>>s5; 读取有效字符到遇到空格

getline(cin,s6); 读取字符到遇到换行，空格可读入，知道‘\n’结束getline(cin,s7,'a'); 一个直到‘a’结束，其中任何字符包括'\n'都能够读入

s.empty() 判断是否为空，bool型

s.size() 或 s.length() 返回字符的个数

s[n] 返回位置为n的字符，从0开始计数

s1+s2 连接，看下面例子：

可用此方法给字符串后面添加字符如：s=s+'a';

a: string s2=s1+", "; //对，把一个string对象和一个字符面值连接起来是允许的

b: string s4="hello "+", "; //错，不能将两个字符串面值相加

c: string s5=s1+", "+"world"; //对，前面两个相加相当于一个string对象；

d: string s6="hello" + ", " + s2; //错

（注：字符串尾部追加还可用s.append("abc")函数添加）

s1=s2 替换

s1==s2 相等，返回true或false

!=,<,<=,>,>= 字符串比较，两个字符串短的与长的前面匹配，短的小于长的

/\*-------------------------插入函数----------------------------------包括迭代器操作和下标操作，下标操作更灵活\*/

s.insert( it , p ); 把字符串p插入到it的位置

s.insert(p,n,t)； 迭代器p元素之前插入n个t的副本

s.insert(p,b,e); 迭代器p元素之前插入迭代器b到e之间的所有元素

s.insert(p,s2,poe2,len); 在下标p之前插入s2下标从poe2开始长度为len的元素

s.insert(pos,cp,len); 下标pos之前插入cp数组的前len个元素。

/\*-----------------------替换函数-------------------------------\*/

s.assign(b,e); 用迭代器b到e范围内的元素替换s

s.assign(n,t)； 用n个t的副本替换s

a.assign(s1,pos2,len);从s1的下标pos2开始连续替换len个。

s.replace ( 3 , 3 , " good " ) ; 从第三个起连续三个替换为good

s.substr(i,j) 截取从i到j的子串，包括i不包括j //string::npos 判断字符串是否结束

/\*-----------------------删除函数-----------------------------\*/

s.erase( 3 )||s.erase ( 0 , 4 ) ; 删除第四个元素或第一到第五个元素

/\*----------------------其他函数-----------------------------\*/

s.find ( "cat" ) ; 查找第一个出现的字符串”cat“，返回其下标值，查不到返回string::npos，也可查找字符；

s.find ( "cat",pos ) ; 从pos开始查找第一个出现的字符串”cat“，返回其下标值，查不到返回string::npos，也可查找字符

s.append(args); 将args接到s的后面

s.compare ( " good " ) ; s与”good“比较相等返回0，比"good"大返回1，小则返回-1；

reverse ( s.begin(), s.end () ); 反向排序函数，即字符串反转函数

count(s.begin(),s.end(),ch); 在s中查找ch出现的次数

1. permutation

#include<iostream>

#include<algorithm>//next\_permutation&&prev\_permutation

#include<string>

using namespace std;

int main()

{

int a[3]={1,2,3};

int b[3]={3,2,1};

string line="ABCD";

while(next\_permutation(a,a+3))//生成大于当前序列的排列,若排列存在返回true,否则返回false

{

for(int i=0;i<3;i++)

cout<<a[i];

cout<<endl;

}

cout<<endl;

while(prev\_permutation(b,b+3))//生成小于当前序列的排列,若排列存在返回true,否则返回false

{

for(int i=0;i<3;i++)

cout<<b[i];

cout<<endl;

}

cout<<endl;

while(next\_permutation(line.begin(),line.end()))

{

cout<<line<<endl;

}

}

10.bitset

头文件:#include<bitset>

构造方法：

bitset<length> b b为bitset对象,它能容纳length个bit位,每个元素初值为0

bitset<length> b(unsigned long u) b有length位,并用u赋值;如果u超过n位,则顶端被截除

bitset<length> b(string s) 以字符串 s 初始化长度为 length 的 b, s 必须仅包含01

bitset<length> b(string s, pos) b是 s 中从位置 pos 开始位的副本,前面的多余位自动填充0

bitset<length> b(s, pos, num) b是s中从位置pos开始的num个位的副本,如果num<n,则前面的空位自动填充0

cin >> b 如果输入的不是0或1的字符,只取该字符前面的二进制位.

常用操作：

b.any( ) b 中是否存在值为 1 的二进制位

b.none( ) b 中是否不存在值为 1 的二进制位

b.set() 对 b 中全部元素设置为 1

b.reset() 对 b 中全部元素设置为 0

b.set(pos) 即 b[pos] = 1

b.set(pos, value) 即 b[pos] = value

b.reset(pos) 即 b[pos] = 0

b.to\_string() 返回 b 的 string 表示法

b.to\_ulong 返回 b 的 long 型表示法

b.count() 返回二进制为 1 的个数

b.size(） 二进制位的个数

b.flip() 所有二进制位按位取反

b.flip(pos) 处于 pos 位置的数取反

b.test(pos) 在pos处的二进制位是否为1

常见运算：

b1 = b2 & b3; 按位与

b1 = b2 | b3; 按位或

b1 = b2 ^ b3; 按位异或

b1 = ~b2; 按位补

b1 = b2 << 7; 移位

1. Java 大数
2. BigInteger

Ⅰ基本函数(加减乘除不会修改BigInteger本身,是不可变类)：

1.valueOf(parament);//将参数转换为制定的类型

2.add(); //大整数相加

3.subtract(); //相减

4.multiply(); //相乘

5.divide(); //相除取整

6.remainder(); //取余

7.pow(); //a.pow(b)=a^b

8.gcd(); //最大公约数

9.abs(); //绝对值

10.negate(); //取反数

11.mod(); //a.mod(b)=a%b=a.remainder(b);

12.max(); min(); //返回较大者/较小者

13.public int comareTo(); //大数比较

14.boolean equals(); //是否相等

15.BigInteger构造函数：

BigInteger(String val); //将指定字符串转换为十进制表示形式；

BigInteger(String val,int radix);//将指定基数的 BigInteger 的字符串表示形式转换为 BigInteger

Ⅱ.基本常量：

A=BigInteger.ONE 1

B=BigInteger.TEN 10

C=BigInteger.ZERO 0

1. BigDecimal

其他操作基本相同,但divide时要指定舍入方式,否则可能除不净抛出异常。

舍入方式：

ROUND\_UP 向远离0的方向舍入

ROUND\_DOWN 向0方向舍入

ROUND\_CEILING 向正无穷方向舍入

ROUND\_FLOOR 向负无穷方向舍入

ROUND\_HALF\_DOWN 向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，向下舍入, 例如1.55 保留一位小数结果为1.5

ROUND\_HALF\_UP 向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，向上舍入, 1.55保留一位小数结果为1.6

ROUND\_HALF\_EVEN 向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，如果保留位数是奇数，使用ROUND\_HALF\_UP ，如果是偶数，使用ROUND\_HALF\_DOWN

ROUND\_UNNECESSARY 计算结果是精确的，不需要舍入模式

改变BigDecimal保留位数方法

setScale(保留位数,舍入方式)

二、基础知识

1. 枚举

就是暴力算法，包括循环遍历和DFS暴搜,用于数据量不大的情况。过不了就想简便算法，DFS考虑剪枝或转dp，还可以二分枚举。

1. 贪心
2. 模拟
3. 排序

1.qsort

void qsort( void \*base, size\_t num, size\_t width, cmp )

int compare (const void \*elem1, const void \*elem2 ) );

参数意义如下:

第一个参数 base 是 需要排序的目标数组名（或者也可以理解成开始排序的地址，因为可以写&s[i]这样的表达式）

第二个参数 num 是 参与排序的目标数组元素个数

第三个参数 width 是单个元素的大小（或者目标数组中每一个元素长度），推荐使用sizeof(s[0]）这样的表达式

第四个参数 cmp是自定义排序规则

cmp的六种写法

1)对int类型数组排序

int num[100];

Sample:

int cmp ( const void \*a , const void \*b )

{

return \*(int \*)a - \*(int \*)b;

}

qsort(num,100,sizeof(num[0]),cmp);

2)对char类型数组排序（同int类型）

char word[100];

Sample:

int cmp( const void \*a , const void \*b )

{

return \*(char \*)a - \*(int \*)b;

}

qsort(word,100,sizeof(word[0]),cmp);

3)对double类型数组排序（特别要注意）

double in[100];

int cmp( const void \*a , const void \*b )

{

return \*(double \*)a > \*(double \*)b ? 1 : -1;

}

qsort(in,100,sizeof(in[0]),cmp)；

4)对结构体一级排序

struct In

{

double data;

int other;

}s[100]

//按照data的值从小到大将结构体排序,关于结构体内的排序关键数据data的类型可以很多种，参考上面的例子写

int cmp( const void \*a ,const void \*B)

{

return (\*(In \*)a)->data > (\*(In \*)B)->data ? 1 : -1;

}

qsort(s,100,sizeof(s[0]),cmp);

5)对结构体二级排序

struct In

{

int x;

int y;

}s[100];

//按照x从小到大排序，当x相等时按照y从大到小排序

int cmp( const void \*a , const void \*b )

{

struct In \*c = (In \*)a;

struct In \*d = (In \*)b;

if(c->x != d->x) return c->x - d->x;

else return d->y - c->y;

}

qsort(s,100,sizeof(s[0]),cmp);

6)对字符串进行排序

struct In

{

int data;

char str[100];

}s[100];

//按照结构体中字符串str的字典顺序排序

int cmp ( const void \*a , const void \*b )

{

return strcmp( (\*(In \*)a)->str , (\*(In \*)B)->str );

}

qsort(s,100,sizeof(s[0]),cmp);

1. sort

sort(begin,begin + n,cmp)

参数意义如下:

第一个参数是需要排序的目标数组的起始位置

第二个参数是起始位置加上要排序的元素个数

第三个参数是定义的cmp规则

cmp函数写法

struct Node{

int x;

double y;

}

bool cmp(Node a,Node b){  
 if(a.x != b.x)

return a.x < b.x;

return a.y > b.y;

}

写cmp的技巧是假设最后排序好的结果是a在b的前面,再考虑a和b对应量的大小关系

1. 分治
2. 最近点对问题

/\*0：把所有的点按照横坐标排序

1：用一条竖直的线L将所有的点分成两等份

2：递归算出左半部分的最近两点距离d1，右半部分的最近两点距离d2，取d=min(d1,d2)

3：算出“一个在左半部分，另一个在右半部分”这样的点对的最短距离d3。

4：结果＝min(d1,d2,d3)

3.1 删除所有到L的距离大于d的点。O(n)

3.2 把右半平面的点按照纵坐标y排序。O(nlogn)

3.3 对于左半平面内的每个点P1，找出右半平面内纵坐标与P1的纵坐标的差在d以内的点P2，

计算距离取最小值，算出d3。O(n\*6) = O(n)因为3.2的排序需要O(nlogn),所以整个算法的复杂度就是O(n((logn)^2))。\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cmath>

#include <algorithm>

using namespace std;

const double INF = 1e20;

const int N = 100005;

struct Point

{

double x;

double y;

}point[N];

int n;

int tmpt[N];

bool cmpxy(Point a,Point b)

{

if(a.x != b.x)

return a.x < b.x;

return a.y < b.y;

}

bool cmpy(int a,int b)

{

return point[a].y < point[b].y;

}

double dis(int i, int j)

{

return sqrt((point[i].x-point[j].x)\*(point[i].x-point[j].x)+ (point[i].y-point[j].y)\*(point[i].y-point[j].y));

}

double solve(int left, int right)

{

double d = INF;

if(left==right)

return d;

if(left + 1 == right)

return dis(left, right);

int mid = (left+right)>>1;

double d1 = solve(left,mid);

double d2 = solve(mid+1,right);

d = min(d1,d2);

int i,j,k=0;

//分离出宽度为d的区间

for(i = left; i <= right; i++)

{

if(fabs(point[mid].x-point[i].x) <= d)

tmpt[k++] = i;

}

sort(tmpt,tmpt+k,cmpy);

//线性扫描

for(i = 0; i < k; i++){

for(j= i+1; j < k && point[tmpt[j]].y-point[tmpt[i]].y<d; j++){

double d3 = dis(tmpt[i],tmpt[j]);

if(d3 < d)

d = d3;

}

}

return d;

}

int main()

{

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

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

scanf("%lf%lf",&point[i].x,&point[i].y);

sort(point,point+n,cmpxy);

printf("%.2lf\n",solve(0,n-1));

}

return 0;

}

1. 点分治

（求第K小的边，二分转换为下面的问题）

给一棵边带权树，问两点之间的距离小于等于K的点对有多少个

#include<stdio.h>

#include<vector>

#include<algorithm>

using namespace std;

#define N 10005

struct node{

int x, y;

node(int xx, int yy){

x = xx, y = yy;

}

};

vector<node> lin[N];

vector<int> d;

int sz[N], f[N];

int n, k, rt, vis[N], dis[N];

int ans, size;

//求树的重心

//sz[x]-->x的树大小，f[x]-->x最大子树的结点树

void getrt(int x, int fa){//利用\*sz,\*f求重心

sz[x] = 1;

f[x] = 0;

for(int i = 0; i < lin[x].size(); i++){

int u = lin[x][i].x;

if(vis[u] || u == fa) continue;

getrt(u, x);

sz[x] += sz[u];

f[x] = max(f[x], sz[u]);

}

f[x] = max(f[x], size-sz[x]);//x最大子树的结点数=max(与此子树大小-f[x],f[x])

if(f[x] < f[rt]) rt = x;

}

//cal中的dfs

void getdis(int x, int fa){//一遍dfs求值+求重心的一点预处理sz[x],求子树大小size

sz[x] = 1;

d.push\_back(dis[x]);

for(int i = 0; i < lin[x].size(); i++){

int u = lin[x][i].x;

if(vis[u] || u == fa) continue;

dis[u] = dis[x] + lin[x][i].y;

getdis(u, x);

sz[x] += sz[u];

}

}

//cal函数

/\*一般两个辅助数组

pre[x]代表到当前根的子树时，当前子树之前路径上值为x的方案数

now[x]代表到当前根的子树时，当前子树上路径上值为x的方案数\*/

int cal(int x, int y){

int res = 0;

d.clear();

dis[x] = y;

getdis(x, 0);

sort(d.begin(), d.end());

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

while(l < r){

while(d[l] + d[r] > k && l < r)

r--;

res += r - l;

l++;

}

return res;

}

void solve(int x){

ans += cal(x, 0);

vis[x] = 1;

for(int i = 0; i < lin[x].size(); i++){

int u = lin[x][i].x;

if(vis[u]) continue;

ans -= cal(u, lin[x][i].y);

f[0] = size = sz[u];

getrt(u, rt = 0);

solve(rt);

}

}

int main(){

while(~scanf("%d%d", &n, &k) && (n + k)){

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

vis[i] = 0;

lin[i].clear();

}

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

int u, v, l;

scanf("%d%d%d", &u, &v, &l);

lin[u].push\_back(node(v, l));

lin[v].push\_back(node(u, l));

}

ans = 0;

f[0] = size = n;

getrt(1, rt = 0);

solve(rt);

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

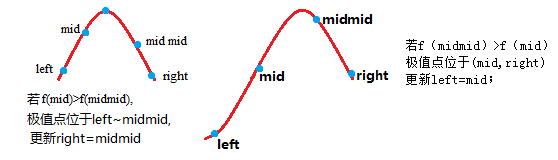
}

return 0;

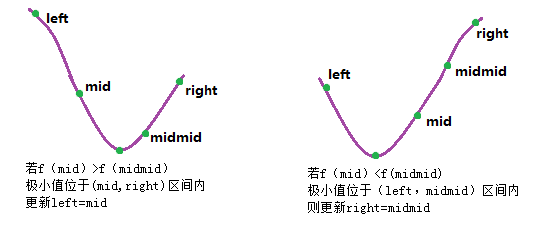
}

1. BFS
2. DFS
3. 二分
4. 三分

凸函数：



凹函数：



void Solve()

{

double left, right, m1, m2, m1\_value, m2\_value;

left = MIN;

right = MAX;

while (left + EPS < right)

{

m1 = left + (right - left)/3;

m2 = right - (right - left)/3;

m1\_value = f(m1);

m2\_value = f(m2);

//假设求解极大值

if (m1\_value >= m2\_value)

right = m2;

else

left = m1;

}

}

1. 双指针扫描 + 尺取法

/\*尺取法通常适用于选取区间有一定规律，或者说所选取的区间有一定

的变化趋势的情况，通俗地说，在对所选取区间进行判断之后，我们可

以明确如何进一步有方向地推进区间端点以求解满足条件的区间，如果

已经判断了目前所选取的区间，但却无法确定所要求解的区间如何进一步

得到根据其端点得到，那么尺取法便是不可行的。首先，明确题目所需要

求解的量之后，区间左右端点一般从最整个数组的起点开始，之后判断区

间是否符合条件在根据实际情况变化区间的端点求解答案。\*/

/\*　整个过程分为4步：

　　　　1.初始化左右端点

　　　　2.不断扩大右端点，直到满足条件

　　　　3.如果第二步中无法满足条件，则终止，否则更新结果

　　　　4.将左端点扩大1，然后回到第二步\*/

题意：给定长度为n的整数数列以及整数S，求出总和不小于S的连续子序列的长度的最小值，如果解 不存在，输出0.

#include <cstdio>

#include <algorithm>

#include <cstring>

#define MAX 100005

#define LL long long

#define INF 0x3f3f3f3f

using namespace std;

LL a[100010];

int n, t, ans = INF;

LL sum, s;

int main()

{

scanf("%d", &t);

while (t--){

scanf("%d %I64d", &n, &s);

for (int i = 0; i < n; i++) scanf("%I64d", a+i);

int st = 0, end = 0;

ans = INF; sum = 0;

while (1){

while (end<n && sum<s) sum += a[end++];

if (sum < s) break;

ans = min(ans, end-st);

sum -= a[st++];

}

if (ans == INF) ans = 0;

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

}

return 0;

}

三、动态规划

1. DP 基础
2. 基础 DP 问题
3. 背包模型

1) 01背包

#include <iostream>//每种物品仅有一件，可以选择放或不放。

#include <cstdio>

#include <cstring>

using namespace std;

int c[1005],w[1005];

int F[1005];

int main()

{

int t,n,V,i,v;

scanf("%d",&t);/\*有 N 件物品和一个容量为 V 的背包。放入第 i 件物品耗费的费用是 Ci

得到的价值是 Wi。求解将哪些物品装入背包可使价值总和最大。转移方程：F[i, v] = max{F[i − 1, v],

F[i − 1, v − Ci] + Wi}\*/

while(t--)

{

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

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

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

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

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

memset(F,0,sizeof(F));

/\*如果要求恰好装满背包，那么在初始化时除了 F[0] 为 0，其它

F[1..V ] 均设为 −∞，这样就可以保证最终得到的 F[V ] 是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 F[0..V ]

全部设为 0。\*/

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

for(v=V;v>=c[i];v--)//当V比较大的时候要进行常数优化,循环下界改成max(F[V - Σc[i->n]], c[i])

{

F[v]=max(F[v],F[v-c[i]]+w[i]);

}

printf("%d\n",F[V]);

}

return 0;

}

1. 完全背包

/\*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。放入第 i 种物品的费用是 Ci，价值是 Wi。求解：将哪些物品装入背包，可使这些物品的耗费的费用总和不超过背包容量，且价值总和最大。\*/

#include <iostream>

#include <stdio.h>

#include <algorithm>

#include <cstring>

#define maxn 10000+10

using namespace std;

int w[maxn],c[maxn];

int F[maxn];

int main()

{

int t,n,V;

cin>>t;

while(t--)

{

cin>>n>>V;

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

{

cin>>w[i]>>c[i];

}

memset(F,0,sizeof(F));//如果恰好装满,F[0] = 0,F[1...n] = -INF

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

{

for(int v=c[i];v<=V;v++)

{

F[v]=max(F[v],F[v-c[i]]+w[i]);

}

}

printf("%d\n",F[V]);

}

return 0;

}

1. 多重背包

/\*有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 M i 件可用,每件耗费的空间是 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耗费的空间总和不超过背包容量,且价值总和最大。\*/

#include <iostream>

#include <stdio.h>

using namespace std;

int F[105],v,n[105],w[105],c[105];

void ZeroOnePack(int cost,int value)

{

for(int i=v; i>=cost; i--)

F[i]=max(F[i],F[i-cost]+value);

}

void CompletePack(int cost,int value)

{

for(int i=cost; i<=v; i++)

F[i]=max(F[i],F[i-cost]+value);

}

void MultiPack(int cost,int value,int num)

{

int k;

if(num\*cost>=v)

CompletePack(cost,value);

else

{

k=1;

while(num>k)

{

ZeroOnePack(k\*cost,k\*value);

num-=k;

k\*=2;

}

ZeroOnePack(num\*cost,num\*value);

}

}

int main()

{

int c,m,i;

scanf("%d",&c);

while( c--)

{

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

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

scanf("%d%d%d",&w[i],&c[i],&n[i]);

memset(F,0,sizeof(F));

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

MultiPack(w[i],c[i],n[i]);

printf("%d\n",F[v]);

}

}

1. 最长公共子序列(LCS)

#include<cstdio>

#include<cstring>

#include<string>

#include<iostream>

#include<algorithm>

using namespace std;

int dp[1005][1005];

int main()

{

string A,B;//不必连续

while(cin>>A>>B)

{

memset(dp,0,sizeof(dp));

for(int i=1;i<=A.length();i++)

for(int j=1;j<=B.length();j++)

{

if(A[i-1]==B[j-1])//注意第i个和第j个字符下标为i-1和j-1

dp[i][j]=dp[i-1][j-1]+1;

else

dp[i][j]=max(dp[i-1][j],dp[i][j-1]);

}

cout<<dp[A.length()][B.length()]<<endl;

}

}

1. 最长公共子串

#include<cstdio>

#include<cstring>

#include<string>

#include<iostream>

#include<algorithm>

using namespace std;

int dp[1005][1005];

int main()

{

string A,B;

while(cin>>A>>B)//要连续

{

int result=0;

memset(dp,0,sizeof(dp));

for(int i=1;i<=A.length();i++)

for(int j=1;j<=B.length();j++)

{

if(A[i-1]==B[j-1])//注意第i个和第j个字符下标为i-1 和j-1

{

dp[i][j]=dp[i-1][j-1]+1;

result=max(dp[i][j],result);

}

else

{

dp[i][j]=0;

}

}

cout<<result<<endl;

}

}

1. 最长上升子序列(LIS)

#include<cstdio>//O(n^2)的算法

#include<iostream>

#include<string>

#include<algorithm>

using namespace std;

int dp[105];

string s;

int main()

{

int Max=0;

cin>>s;

int len=s.size();

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

{

dp[i]=1;

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

{

if(s[j]<s[i]&&dp[i]<dp[j]+1)

dp[i]=dp[j]+1;

}

Max=max(dp[i],Max);

}

cout<<Max<<endl;

}

//最长不下降子序列——动规优化算法O(nlogn)

#include<cstdlib>

#include<iostream>

#define N 100001

using namespace std;

int a[N];

int Link[N]; //Link[x] 代表链接a[x]的数的序号

int Long[N]; //Long[k] = min{a[t]} (F[t] = k, F[]为深度)

int n, Ma;

int Binary\_search(int key){ //二分查找

int l = 1, r = Ma, mid;

while(l <= r){

mid = (l + r) >> 1;

if(a[Long[mid]] < key)

l = mid + 1;

else

r = mid - 1;

}

return l - 1;

}

int main(){

cin >> n;

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

cin >> a[i];

Ma = 1; //初始最长序号数为1

Long[Ma] = 1; //初始为第一个元素

Link[1] = Long[Ma - 1]; //初始链为第一个元素

for(int i = 2; i <= n; i++){

if(a[i] >= a[Long[Ma]]){//接在既定序列之后

Ma++;

Long[Ma] = i;

Link[i] = Long[Ma - 1];

}

else{ //在既定序列中查找位置并插入

int x = Binary\_search(a[i]);

Long[x + 1] = i;

Link[i] = Long[x];

}

}

cout << Ma << endl;

int k = Ma;

int i = Long[Ma];

while(Ma > 0){

Long[Ma] = a[i]; //改为Long[Ma] = i可输出序号

i = Link[i];

Ma--;

}

//for(int i = 1; i <= k; i++)

// cout << Long[i] << " ";

return 0;

}

//最长不下降子序列——非动规 O(nlogn)

开辟一个堆栈数组Stack[],每次取栈顶元素top和读到的元素tmp做比较，如果tmp > top则将tmp入栈；如果tmp < top则二分查找栈中的比tmp大的第1个数，并用tmp替换它。最后其最长不下降子序列长度即为栈的大小。

#include<iostream>

#include<cstdlib>

#define N 1001

using namespace std;

int main(){

int top = 0, tmp, n;

int stack[N];

cin >> n;

stack[0] = -1; //第一个元素可能为-1

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

cin >> tmp;

if(tmp > stack[top]) //比栈顶元素大的数就入栈

stack[++top] = tmp;

else{

int l = 1, h = top;

int mid;

while(l <= h){ //二分检索栈中比tmp大的第一个数

mid = (l + h) >> 1;

if(tmp > stack[mid])

l = mid + 1;

else

h = mid - 1;

}

stack[l] = tmp; //用tmp替换

}

}

cout << top << endl; //最长序列数就是栈的大小

return 0;

}

上述代码也可以写成

void solve(){

fill(dp,dp+n,INF);

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

\*lower\_bound(dp,dp+n,a[i]) = a[i];

}

printf("%d\n",lower\_bound(dp,dp+n,INF) - dp);

}

1. 括号序列模型

#include <iostream>

#include <stack>

#include <algorithm>

using namespace std;

stack<char> st;

int Check(char \*p)

{

int len=strlen(p);

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

{

if(p[i]=='('||p[i]=='['|| p[i]=='{')//是左括号的入栈

{

st.push(p[i]);

}

if(p[i]==')')//是右括号 判断

{

if( !st.empty() && st.top()=='(')//匹配，出栈。注意：top(),pop()时要不为空，否则出错

{

st.pop();

}

else

return 0;//不匹配，则直接返回

}

if(p[i]==']')

{

if( !st.empty() && st.top()=='[')

{

st.pop();

}

else

return 0;

}

if(p[i]=='}')

{

if( !st.empty() && st.top()=='{')

{

st.pop();

}

else

return 0;

}

}

if(st.empty()==true)//栈为空了，说明匹配；否则，不匹配

{

return 1;

}

else

return 0;

}

void main()

{

char p[100];

cin>>p;

int r=Check(p);

if(r==1)

cout<<"正确"<<endl;

else

cout<<"错误"<<endl;

system("pause");

}

1. 递推模型
2. 线段覆盖问题
3. 连续段划分模型

(三)树形 DP

(四)状压 DP

TSP问题

题意：给出一幅图，图中有一些点，然后从第1个点出发，然后途径所有有石头的点，最后回到原点，然后求最小距离。

#include<bits/stdc++.h>

#define inf 0x3f3f3f3f

using namespace std;

int a[55][55];

int dis[55][55];

int dp[15][(1<<11)+10];

/\*第一维表示在哪个结点,第二维表示已走过状态的01序列(不包括起点),

0表示未走过,1表示走过,存储的是到目前状态的花费\*/

int n,m,cnt;

struct node{//记录特殊点

int x;

int y;

}p[55];

void solve(){

memset(dp,inf,sizeof(dp));//初始化为正无穷

dp[0][0]=0;//初始状态(起点为0号点)

for(int state=0;state<(1<<cnt);state++){/\*先枚举状态,后枚举所在点,

因为每个点只经过一次,dp是关于state层层递进的拓扑序列，先要枚举完同一个

状态下所有点的情况\*/

for(int i=0;i<cnt;i++){

if(dp[i][state]!=inf){//要保证这一状态已经到达过

for(int j=0;j<cnt;j++){//枚举下一次的石块位置

if((1<<j)&state>0) continue;//已经挖过的石块不重复挖

if(dis[i][j]==inf) continue;//如果不能到达也不行

dp[j][state|(1<<j)]=min(dp[j][state|(1<<j)],dp[i][state]+dis[i][j]);//转移到下一状态

}

}

}

}

}

int main(){

while(cin>>n>>m){

cnt=0;

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

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

cin>>a[i][j];

if(i==0&&j==0){//储存起点和有石头的点为关键点

p[cnt].x=i;

p[cnt++].y=j;

}

else if(a[i][j]>0){

p[cnt].x=i;

p[cnt++].y=j;

}

}

}

for(int i=0;i<cnt;i++){

/\*没有说每个点可以经过多次,而且考虑可知每个点经过一次的花费一定最小，

所以距离直接求曼哈顿距离就可以,如果可经过多次要考虑从每个关键点出发做

BFS得到dis数组。如果不是单位距离为1的方格图要用Floyd处理两点间的最短路\*/

dis[i][i]=0;

for(int j=i+1;j<cnt;j++){

dis[i][j]=dis[j][i]=abs(p[i].x-p[j].x)+abs(p[i].y-p[j].y);

}

}

solve();

int ans=inf;

for(int i=0;i<cnt;i++){

ans=min(ans,dp[i][(1<<cnt)-1]+dis[0][i]);//枚举终点再加上终点到起点的距离即可

}

cout<<ans<<endl;

}

}

1. DP 优化
2. 区间DP
3. 石子合并

/\*分析：要求n个石子归并，我们根据dp的思想划分成子问题，先求出每两个合并的最小代价，然后每三个的最小代价，依次知道n个。

定义状态dp [ i ] [ j ]为从第i个石子到第j个石子的合并最小代价。

那么dp [ i ] [ j ] = min(dp [ i ] [ k ] + dp [ k+1 ] [ j ])

那么我们就可以从小到大依次枚举让石子合并，直到所有的石子都合并。

这个问题可以用到平行四边形优化，用一个s【i】【j】=k 表示区间 i---j 从k点分开才是最优的，这样的话我们就可以优化掉一层复杂度，变为O（n^2）.\*/

#include <cstdio>

#include <cstring>

#include <algorithm>

#define N 210

int dp[N][N],sum[N],s[N][N];

int main()

{

int n;

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

{

int a[N];sum[0]=0;

memset(s,0,sizeof(s));

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

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

s[i][i]=i;

sum[i]=sum[i-1]+a[i];//因为要求解区间和，先维护前缀和

}

memset(dp,0,sizeof(dp));

int i,j,l,k;

for(l = 2; l <= n; ++l)//枚举区间长度

{

for(i = 1; i <= n - l + 1; ++i) //枚举区间左端点

{

j = i + l - 1;//根据左端点和区间长度求区间右端点

dp[i][j] = 0x3f3f3f3f;

for(k = s[i][j-1]; k <= s[i+1][j]; ++k)//四边形优化

{

if(dp[i][j]>dp[i][k] + dp[k + 1][j] + sum[j] - sum[i-1])

{

dp[i][j]=dp[i][k] + dp[k + 1][j] + sum[j] - sum[i-1];

s[i][j]=k;

}

}

}

}

printf("%d\n", dp[1][n]);

}

return 0;

}

1. 括号匹配

/\*定义dp [ i ] [ j ] 为串中第 i 个到第 j 个括号的最大匹配数目

1.如果第 i 个和第 j 个匹配,则dp [ i ] [ j ] = dp [ i+1 ] [ j-1 ] + 2 ;

2.如果第 i 个和第 j 个不匹配，枚举中间分割点k(i <= k < j)

dp[ i ] [ j ] = max ( dp [ i ] [ j ] , dp [ i ] [ k ] + dp [ k+1 ] [ j ] ) ;\*/

#include <iostream>

#include <cstring>

#include <algorithm>

#include <string>

using namespace std;

const int N = 120;

int dp[N][N];

int main()

{

string s;

while(cin>>s)

{

if(s=="end")

break;

memset(dp,0,sizeof(dp));

int n = s.size();

for(int len = 2;len <= n;len++)//枚举区间长度

{

for(int i = 0;i <= n - len; i++)//枚举区间左端点

{

int j = i + len - 1;//确定区间右端点

if(s[i]=='('&&s[j]==')' || s[i]=='['&&s[j]==']')

dp[i][j]=dp[i+1][j-1]+2;

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

dp[i][j]=max(dp[i][j],dp[i][k]+dp[k+1][j]);//枚举中间位置,注意j不取等号

}

}

cout<<dp[0][n-1]<<endl;

}

return 0;

}

3.整数划分

/\*区间dp，设dp[i][j] 表示在区间[0, i]之中，插入j个乘号可以得到的最大数

设a[i][j]为区间[i,j]所形成的数

所以 dp[i][j] = max(dp[k][j-1] \* a[k + 1][i])\*/

#include <cmath>

#include <cstdio>

#include <cstring>

#include <iostream>

#include <algorithm>

using namespace std;

long long dp[25][25];

long long a[25][25];

char str[25];

int main()

{

int len, t, m;

scanf("%d", &t);

while (t--)

{

scanf("%s%d", str, &m);

len = strlen(str);

m--;

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

memset (dp, 0, sizeof(dp));

for (int i = 0; i < len; i++) //先对a进行预处理，减少复杂度，a[i][j]表示第i段到第j段的数值

{

a[i][i] = str[i] - '0';

for (int j = i + 1; j < len; j++)

{

a[i][j] = a[i][j - 1] \* 10 + str[j] - '0';

}

}

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

{

dp[i][0] = a[0][i];

}

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

{

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

{

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

{

dp[i][j] = max(dp[i][j], dp[k][j - 1] \* a[k + 1][i]);

}

}

}

printf("%lld\n", dp[len - 1][m]);

}

return 0;

}

四、数据结构(包含 RMQ 问题)

1. 并查集

/\*

重点步骤:

1、Find函数(加路径压缩)

2、Union函数(找根结点后联合)

3、初始父结点要设为自己

4、判断是否为同一集合,只需判断根结点是否相同\*/

//复杂度为 O(α(n))，α(n)是阿克曼函数的反函数，比O(logn)要快

int f[maxn];//父亲数组

int rank[maxn];//高度数组

void init(int n){//初始化n个元素

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

f[i] = i;

rank[i] = 0;

}

}

int Find(int x){//查询树的根

return f[x]==x?x:f[x]=Find(f[x]);//路径压缩优化

}

/\*寻找祖先时，我们一般采用递归查找，但是当元素很多亦或是整棵树变为一条链时，

每次Find(x)都是O(n)的复杂度。为了避免这种情况，我们需对路径进行压缩，

即当我们经过”递推”找到祖先节点后，”回溯”的时候顺便将它的子孙节点都直接指向祖先，

这样以后再次Find(x)时复杂度就变成O(1)了，如下图所示。可见，路径压缩方便了以后的查找。\*/

void Union(int x,int y){//合并x和y所属的集合

x = Find(x);

y = Find(y);

if(x == y) return;

if(rank[x] < rank[y])

f[x] = y;

else{

f[y] = x;

if(rank[x] == rank[y])//rank优化，每次从rank小的向rank大的连边，使树的高度尽量小

rank[x]++;

}

}

bool same(int x,int y){//判断x和y是否属于同一个集合

return Find(x) == Find(y);

/\*判断是否同属于一个集合或者得到根结点要用Find函数，不能用f[x]==f[y]来判断，因为有时候路径压缩还没有完成，x y虽然在一个集合中，

但f[x] != f[y]\*/

}

//并查集求连通分支数，不要遍历判断Find(i) == i，而是Union一次，如果根不同,就把分支数减一

//种类并查集：用于判断几个种类的归属和矛盾问题。经典例题：食物链、Find them,catch them、A Bug's Life

/\*带权并查集：即在并查集元素中附加了信息，比如并查集的元素数量，和根结点的距离等，常常结合向量考虑。经典例题：Dragon Balls、Zjnu Stadium

带权并查集常常会遇到需要修改一个集合中所有元素某信息的情况，此时就考虑在Union中只修改根结点信息，在Find时路径压缩对其他结点进行修改\*/

1. 树状数组

#include<bits/stdc++.h>

//a下标从1开始

using namespace std;

#define mem(a, b) memset(a, b, sizeof(a))

#define inf 0x3f3f3f3f

#define ll long long

#define maxn 500005

/\*一个一维数组c[]

其中c[i]表示

[i-2^k+1,i]这个区间内所有元素的信息

c[i]有2^k项,k为i表示成二进制形式i末尾0的个数

如果需要求和：

那么c[i]表示[i-2^k+1,i]这个区间内所有元素的和

如果需要求最大值：

那么c[i]表示[i-2^k+1,i]这个区间内所有元素中最大的那一个

\*/

struct Num {

int val, pos;

}a[maxn];

ll c[maxn];

int n;

int lowbit(int x) {//将x表示成二进制,末尾有k个0,x&(-x)=2^k

return x&(-x);

}

void add(int pos, int num) {//修改a[pos],自底向上更新C数组,num为变化量,n为数组a中元素个数

while(pos <= n) {

c[pos] += num;

pos += lowbit(pos);//pos的父结点为 pos+lowbit(pos),从pos开始自底向上更新其所有祖先

}

}//操作之后只是修改了C数组的信息,a数组本身没有变化

/\*若要修改a[pos]值val1为val2,只要：

add(pos,-val1);

add(pos,val2);

a[x]=val2;

\*/

int query(int pos) {//查询前缀和,查询区间只要看作前缀和之差

int ans = 0;

while(pos > 0) {

ans += c[pos];

pos -= lowbit(pos);

}

return ans;

}

/\*求an的前n项和,分割成cn计算：

如计算an的前13项 13=1101=2^0+2^2+2^3

a1+a2+…+a13=c13(a13)+c12(a12+a11+a10+a9)+c8(a8+……+a1)(因为c13有2^0项,c12有2^2项,c8有2^3项),

12=13-lowbit(13);8=12-lowbit(12)\*/

void update(int x,int y,int n){//区间更新,此时query(i)查询到的前缀和正好等于a[i],做到了单点查询

add(x,n);

add(y+1,-n);

}

1. 前缀和
2. 逆序数

/\*因为只能交换相邻两个数,也就是交换一次只能消除一个逆序数,

所以交换次数等于所有数字的逆序数之和,问题转换为求逆序数之和\*/

#include <cstdio>

#include <cstring>

#include <iostream>

#include <algorithm>

using namespace std;

const int maxn=500017;

int n;

int aa[maxn]; //离散化后的数组

int c[maxn]; //树状数组

struct Node

{

int v;

int order;

}in[maxn];

int Lowbit(int x) //2^k

{

return x&(-x);

}

void update(int i, int x)//i点增量为x

{

while(i <= n)

{

c[i] += x;

i += Lowbit(i);

}

}

int sum(int x)//区间求和 [1,x]

{

int sum=0;

while(x>0)

{

sum+=c[x];

x-=Lowbit(x);

}

return sum;

/\*当原数组的取值范围超出c数组的下标范围(如出现负数或者大数,因为树状数组是更新C数组的pos处,pos的范围就是原数组的范围),必须进行离散化.本题数字范围过大,所以离散化.具体操作是把原数组按照value排序,然后按照order映射到小的数,因为是求逆序数,所以对结果没有影响.如把-100,10000000000000,13映射到1,3,2不会影响结果\*/

}

bool cmp(Node a ,Node b)

{

return a.v < b.v;

}

int main()

{

int i,j;

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

{

//离散化

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

{

scanf("%d",&in[i].v);

in[i].order=i;

}

//按大小排序,按order放置。

sort(in+1,in+n+1,cmp);

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

aa[in[i].order] = i;

//树状数组求逆序

memset(c,0,sizeof(c));

\_\_int64 ans=0;

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

{

update(aa[i],1);

ans += i-sum(aa[i]);

//aa[i]逆序数个数,因为已经放置了i个数,其中有sum(aa[i])个数 <=aa[i],相减就是逆序数

}

printf("%I64d\n",ans);

}

return 0;

}

1. 线段树(Segtree)
2. // kuangbin 模板

struct Node{

int l;

int r;

int sum;

int lazy;

// other data

}segTree[MAXN << 2];

void build(int i, int l, int r){

segTree[i].l = l;

segTree[i].r = r;

segTree[i].sum = 0;

// other operation

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

if(l == r) return;

build(i << 1, l, mid);

build((i << 1) | 1, mid + 1, r);

}

void push\_up(int i){

if(segTree[i].l == segTree[i].r) return;

// segTree[i].sum = segTree[i << 1].sum + segTree[(i << 1) | 1].sum;

// operation

}

void push\_down(int i ){

if(segTree[i].l == segTree[i].r) return;

if(segTree[i].lazy){

// ...

}

}

void update(int i, int l, int r){

if(segTree[i].l == l && segTree[i].r == r){

// ...

}

push\_down(i);

int mid = (segTree[i].l + segTree[i].r) << 1;

if(r <= mid) update(i << 1, l, r);

else if(l > mid) update((i << 1) | 1, l, r);

else{

update(i << 1, l, mid);

update((i << 1) | 1, mid + 1, r);

}

}

int query(int i, int l, int r){

if(segTree[i].l == l && segTree[i].r == r){

// ...

}

push\_down(i);

int mid = (segTree[i].l + segTree[i].r) << 1;

if(r <= mid) return query(i << 1, l, r);

else if(l > mid) return query((i << 1) | 1, l , r);

else{

return query(i << 1, l, mid) + query((i << 1) | 1, mid + 1, r);

}

}

1.基本更新和查询操作

/\*建树和更新都要pushUp,区间更新和查询都要pushDown\*/

/\*node：区间结点号 begin：该node的区间左边界 end：该node的区间右边界

left：查询区间的左边界 right：查询区间的右边界 pos：查询区间的点\*/

#define lson (node<<1)

#define rson ((node<<1)|1)

#define mid ((begin+end)>>1) //最外层的括号不能丢!!!

int segTree[maxn\*4];

int lazy[maxn\*4];

void pushUp(int node){//pushUp自底向上更新区间和与最值

segTree[node]=segTree[lson]+segTree[rson];//segTree[node]=max(segTree[lson],segTree[rson])

}

void pushDown(int node,int begin,int end){

if(!lazy[node]) return;

segTree[lson]+=(mid-begin+1)\*lazy[node];

segTree[rson]+=(end-mid)\*lazy[node];

lazy[lson]+=lazy[node];

lazy[rson]+=lazy[node];

lazy[node]=0;

}

void build(int node,int begin,int end){//建树

lazy[node]=0;

if(begin==end){//begin==end表示管理的是结点

scanf("%d",&segTree[node]);//顺序输入结点

//segTree[node]=a[begin] 用于任意顺序输入,先将输入存入a数组

return;

}

build(lson,begin,mid);

build(rson,mid+1,end);

pushUp(node);

}

void update(int node,int begin,int end,int pos,int k){//单点更新

if(pos<begin||pos>end) return;//管理的区间不包含pos,直接return

if(begin==end){

segTree[node]+=k;

return;

}

update(lson,begin,mid,pos,k);

update(rson,mid+1,end,pos,k);

pushUp(node);

}

int query(int node,int begin,int end,int left,int right){//区间查询

if(left>end||right<begin) return 0;//查询结点和区间没有公共点

if(left<=begin&&right>=end) return segTree[node];//查询区间包含查询结点

pushDown(node,begin,end);

int sum=0;//int maxx=-1

sum+=query(lson,begin,mid,left,right);//maxx=max(maxx,query(lson,begin,mid,left,right))

sum+=query(rson,mid+1,end,left,right);//maxx=max(maxx,query(rson,mid+1,end,left,right))

return sum;

}

void update(int node,int begin,int end,int left,int right,int k){//区间更新

if(left>end||right<begin) return;

if(left<=begin&&right>=end){

segTree[node]+=(end-begin+1)\*k;

lazy[node]+=k;

return;

}

pushDown(node,begin,end);

update(lson,begin,mid,left,right,k);

update(rson,mid+1,end,left,right,k);

pushUp(node);

}

//离散化(1、长序列取一部分 2、面积并或面积交)

1. 矩形面积交、面积并

矩形面积并

#include<bits/stdc++.h>

#define lson (rt<<1)

#define rson ((rt<<1)|1)

#define mid ((l+r)>>1)

#define maxn 2005

using namespace std;

int n;

double y[maxn];

//沿x轴扫描,沿y轴建树,线段树的结点是纵向的线段,最下面一层结点以排序后相邻的y1,y2为边界

struct LINE // 存储线段信息；

{

double x; // 该线段的x坐标；

double y\_up,y\_down; // 竖向线段的上下端点；

int flag;//矩形的左边界为1,右边界为-1

}line[maxn];

struct node//线段树的结点,不再是单个点,是一个区间

{

double l,r; // 区间的左右边界,即某段扫描线的上下端点

double x; // 记录上一个横坐标位置，用于求面积；

int cover; // 记录覆盖的线段数;即同一方向的线段数;由flag累加

bool flag; // 标记只有一个区间的节点,即在线段树最底层的结点,我们将一个一个连续的区间离散化成一个节点；

}node[maxn<<2];

bool cmp(LINE a,LINE b)

{

return a.x<b.x;

}

void build(int rt,int l,int r) // 建树；

{

node[rt].l=y[l]; // 维护区间；

node[rt].r=y[r];

node[rt].x=-1;

node[rt].flag=false;

node[rt].cover=0;

if(l+1==r){ // 区间是连续的;

node[rt].flag=true; // 标记为结点;

return;

}

build(lson,l,mid);

build(rson,mid,r); // 因为将一个个连续区间离散成点，所以此处mid不需要+1；

}

double Insert\_query(int rt,double x,double l,double r,int flag)

/\*查询+更新x处（l,r）区间面积，此处l和r代表的是区间不是结点边界；\*/

{

if(l>=node[rt].r||r<=node[rt].l) return 0; // 该方向节点不包含所要查询的区间；

if(node[rt].flag){ // 找到只有一个区间的叶子节点；

if(node[rt].cover>0){

double pre=node[rt].x;

double ans=(x-pre)\*(node[rt].r-node[rt].l); // 计算面积；

node[rt].x=x; // 更新定位上一下x位置，便于下次计算面积；

node[rt].cover+=flag; // 更新覆盖的线段数；

return ans;

}

else{

node[rt].x=x;

node[rt].cover+=flag;

return 0;//没有产生面积并也要return 0

}

}

double ans1,ans2;

ans1=Insert\_query(lson,x,l,r,flag);

ans2=Insert\_query(rson,x,l,r,flag);

return ans1+ans2;

}

int main()

{

int Case=0;

double x1,x2,y1,y2;

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

int cnt=0;

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

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

y[cnt]=y1;

line[cnt].x=x1;

line[cnt].y\_down=y1;

line[cnt].y\_up=y2;

line[cnt++].flag=1; // 表示左边线段；

y[cnt]=y2;

line[cnt].x=x2;

line[cnt].y\_down=y1;

line[cnt].y\_up=y2;

line[cnt++].flag=-1; // 表示右边线段；

}

sort(y,y+cnt); // 将所有高度由小到大排序，将区间建树表示；

sort(line,line+cnt,cmp); // 排序，返回坐标x靠左的点；

build(1,0,cnt-1);

double area=0;

for(int i=0;i<cnt;i++){

area+=Insert\_query(1,line[i].x,line[i].y\_down,line[i].y\_up,line[i].flag);

}

printf("Test case #%d\nTotal explored area: %.2lf\n\n",++Case,area);

}

return 0;

}

矩形面积交

#include<bits/stdc++.h>

#define lson (rt<<1)

#define rson ((rt<<1)|1)

#define mid ((l+r)>>1)

#define maxn 2005

using namespace std;

int T,N;

double y[maxn];

struct LINE{

double x;

double y\_up,y\_down;

int flag;

}line[maxn];

struct node{

double l,r;

double x;

int cover;

bool flag;

}node[maxn<<2];

bool cmp(LINE a,LINE b){

return a.x<b.x;

}

void build(int rt,int l,int r){

node[rt].l=y[l];

node[rt].r=y[r];

node[rt].x=-1;

node[rt].flag=false;

node[rt].cover=0;

if(l+1==r){

node[rt].flag=true;

return;

}

build(lson,l,mid);

build(rson,mid,r);

}

double Insert\_query(int rt,double x,double l,double r,int flag){

if(l>=node[rt].r||r<=node[rt].l) return 0;

if(node[rt].flag){

if(node[rt].cover>1){

double pre=node[rt].x;

double ans=(x-pre)\*(node[rt].r-node[rt].l);

node[rt].x=x;

node[rt].cover+=flag;

return ans;

}

else{

node[rt].x=x;

node[rt].cover+=flag;

return 0;

}

}

double ans1,ans2;

ans1=Insert\_query(lson,x,l,r,flag);

ans2=Insert\_query(rson,x,l,r,flag);

return ans1+ans2;

}

int main(){

scanf("%d",&T);

while(T--){

int cnt=0;

scanf("%d",&N);

while(N--){

double x1,y1,x2,y2;

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

y[cnt]=y1;

line[cnt].x=x1;

line[cnt].y\_down=y1;

line[cnt].y\_up=y2;

line[cnt++].flag=1;

y[cnt]=y2;

line[cnt].x=x2;

line[cnt].y\_down=y1;

line[cnt].y\_up=y2;

line[cnt++].flag=-1;

}

sort(y,y+cnt);

sort(line,line+cnt,cmp);

build(1,0,cnt-1);

double area=0;

for(int i=0;i<cnt;i++){

area+=Insert\_query(1,line[i].x,line[i].y\_down,line[i].y\_up,line[i].flag);

}

printf("%.2lf\n",area);

}

}

1. 字典树

/\*又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变种。

典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），

(可以认为是一种用于存储字符串的数据结构)所以经常被搜索引擎系统用于

文本词频统计。它的优点是：利用字符串的公共前缀来节约存储空间，最大

限度地减少无谓的字符串比较，查询效率比哈希表高。字典树与字典很相似,

当你要查一个单词是不是在字典树中,首先看单词的第一个字母是不是在字典

的第一层,如果不在,说明字典树里没有该单词,如果在就在该字母的孩子节点

里找是不是有单词的第二个字母,没有说明没有该单词,有的话用同样的方法继

续查找.字典树不仅可以用来储存字母,也可以储存数字等其它数据。\*/

#define MAX 26

/\*MAX是表示每层有多少种类的数，如果只是小写字母，则26即可，

若改为大小写字母，则是52，若再加上数字，则是62了，这里根据题意来确定。

v可以表示一个字典树到此有多少相同前缀的数目，这里根据需要应当学会自由变化。\*/

struct Trie //字典树的数据结构,每个结点存有指向下一层结点的指针数组

{

Trie \*Next[MAX]; //结构体指针数组

int v; //根据需要变化

Trie(){

v = 1;

for(int i = 0;i < MAX;i++){

Next[i] = NULL;

}

}

};

Trie\* root;//记得最初要给root分配空间

/\*Trie的建立\*/

void createTrie(string str)

{

int len = str.size();

Trie \*p = root,\*q;//p指向当前查找结点,初始为根节点

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

{

int id = str[i]-'a';//此处以存放小写字母为例,若存放数字减去'0',大写字母减去'A'

if(p->Next[id] == NULL)//如果该层还没有该字母,开辟空间存放该字母

{

q = new Trie();

p->Next[id] = q;

p = p->Next[id];

}

else

{

p = p->Next[id];

p->v++;

}

}

}

/\*Trie的查找（最主要的操作）：

(1) 每次从根结点开始一次搜索；

(2) 取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索；

(3) 在相应的子树上，取得要查找关键词的第二个字母,并进一步选择对应的子树进行检索。

(4) 迭代过程……

(5) 在某个结点处，关键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找。\*/

int findTrie(string str)

{

int len = str.size();

Trie \*p = root;

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

{

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

p = p->Next[id];

if(p == NULL) //若为空集，表示不存以此为前缀的串

return 0;

}

return p->v; //返回到此有多少相同前缀的数目

}

void dealTrie(Trie\* T)//多组数据时可能会MLE,需要释放空间

{

int i;

if(T==NULL)

return;

for(i=0;i<MAX;i++)

{

dealTrie(T->Next[i]);

}

delete(T); //先释放所有后继结点,再释放当前结点，就像拓扑排序一样。否则后继结点将无法被访问到，但空间还未释放，造成了内存泄漏

}

1. ST 表

//dp1求区间最小值,dp2求区间最大值

int dp1[maxn][20],dp2[maxn][20];

int a[maxn];

void Init()

{

for(int i = 1; i <= n; i++) //注意a的下标要从1开始

{

dp1[i][0] = dp2[i][0] = a[i];

}

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

{

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

{

dp2[j][i] = max(dp2[j][i-1],dp2[j+(1<<(i-1))][i-1]);

dp1[j][i] = min(dp1[j][i-1],dp1[j+(1<<(i-1))][i-1]);

}

}

}

int Query\_max(int l,int r)

{

int k = (int)(log(double(r-l+1))/log((double)2));

return max(dp2[l][k], dp2[r-(1<<k)+1][k]);

}

int Query\_min(int l,int r)

{

int k = (int)(log(double(r-l+1))/log((double)2));

return min(dp1[l][k], dp1[r-(1<<k)+1][k]);

}

1. Tarjan(离线算法)
2. 链式前向星

六、图论

1. 最短路

//正权图的最长路，如果有正权圈则是正无穷，如无圈，可以把边权取反用最短路算法求解

1. Dijkstra

int cost[MAX\_V][MAX\_V]; //邻接矩阵

int d[MAX\_V]; // 最短距离

bool vis[MAX\_V]; //访问标志

int V;//顶点数

int prev[MAX\_V];//最短路上的前趋顶点

//求从起点s出发到各个顶点的最短距离

void dijkstra(int s) {

memset(d,INF,sizeof(d));

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

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

d[s] = 0;

while(1){

int v = -1;

//从尚未使用过的顶点中选择一个距离最小的顶点(使用过的顶点均已经确定了最短路)

for(int u = 0;u < V;u++){

if(!vis[u] && (v == -1 || d[u] < d[v])) v = u;

}

if(v == -1) break; //如果没有未使用的顶点，即所有顶点的最短路都已经确定，跳出循环

vis[v] = true;

for(int u = 0;u < V;u++){ //用这个最小顶点更新邻居结点

if(d[u] > d[v] + cost[v][u]){

d[u] = d[v] + cost[v][u];

prev[u] = v;

}

}

}

}//不可计算负权图

// 到顶点t的最短路(其中一条，如果要求出所有最短路，先求最短路后dfs)

vector<int> get\_path(int t){

vector<int> path;

for( ; t != -1; t = prev[t])

path.push\_back(t); //不断沿着prev[t]直到走到t = s

//这样得到的是按照t到s的顺序，所以翻转之

reverse(path.begin(),path.end());

return path;

}

1. Dijkstra + heap

struct edge {

int to;

int cost;

};

typedef pair<int,int> P; //first是最短距离,second是顶点编号

int V;

vector<edge> G[MAX\_V];

int d[MAX\_V];

priority\_queue< P , vector<P> , greater<P> > que;

/\*优先队列,定义了一个由二元组组成的优先队列,通过指定greater<P>参数，

按照从小到大排序,先比较pair的第一维所以pair要按照<d[i],i>的顺序组

合而不是<i,d[i]>的顺序\*/

void dijkstra(int s) {

memset(d,INF,sizeof(d));

d[s] = 0;

que.push(P(0,s));

while(!que.empty()) {

P p = que.top();

que.pop();

int v = p.second;

if(d[v] < p.first) continue;//如果不是最短距离，丢弃

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

edge e = G[v][i];

if(d[e.to] > d[v] + e.cost){

d[e.to] = d[v] + e.cost;

que.push(P(d[e.to],e.to));

}

}

}

}

1. SPFA

/\*设立一个队列用来保存待优化的顶点，优化时每次取出队首顶点 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。而其检测负权回路的方法也很简单，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。\*/

#include<bits/stdc++.h>

#define maxn 105

using namespace std;

bool vis[maxn];

int dis[maxn];

struct Node{

int to;

int w;

Node(int too,int ww){

to=too;

w=ww;

}

};

vector<Node> cross[maxn];

int N,M,A,B,C;

void spfa(int st){

queue<int> Q;

Q.push(st);

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

memset(dis,0x3f3f3f3f,sizeof(dis));

vis[st]=true;

dis[st]=0;

while(!Q.empty()){

int cur=Q.front();

Q.pop();

vis[cur]=false;//出队把vis重新置为false,因为有可能再次访问

for(int i=0;i<cross[cur].size();i++){

if(dis[cross[cur][i].to]>dis[cur]+cross[cur][i].w){

dis[cross[cur][i].to]=dis[cur]+cross[cur][i].w;//松弛操作

if(!vis[cross[cur][i].to]){

Q.push(cross[cur][i].to);

vis[cross[cur][i].to]=true;

}//没有访问过的邻接点入队

}

}

}

}

int main(){

ios::sync\_with\_stdio(false);

while(cin>>N>>M){

if(N==0&&M==0) break;

for(int i=1;i<=N;i++){//每次记得清空邻接表

cross[i].clear();

}

while(M--){

cin>>A>>B>>C;

cross[A].push\_back(Node(B,C));

cross[B].push\_back(Node(A,C));

}

spfa(1);

cout<<dis[N]<<endl;

}

}

1. Floyed(多源最短路)

for(k=1;k<=n;k++) //n为点数

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

for(j=1;j<=n;j++)

if(e[i][k]<inf && e[k][j]<inf && e[i][k]+e[k][j]<e[i][j])

e[i][j]=e[i][k]+e[k][j]; //即对于一对i和j,能在中间找到一个中转点k使路径变短

/\*Floyd算法还可以用于求最短路径数量,就是在Floyd基础上结合乘法原理和加法原理如下:\*/

for(k=1;k<=n;k++)

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

for(j=1;j<=n;j++)

if(e[i][k]<inf && e[k][j]<inf && e[i][k]+e[k][j]==e[i][j]){//如果有相等的最短路径,增加count[i][j]

count[i][j] +=count[i][k]\*count[k][j];

}

If(e[i][k]<inf && e[k][j]<inf && e[i][k]+e[k][j]<e[i][j]){

e[i][j]=e[i][k]+e[k][j]; //即对于一对i和j,能在中间找到一个中转点k使路径变短

count[i][j] =count[i][k]\*count[k][j];/\*如果找到了更短的路径,原本count[i][j]

记录的就不再是i到j最短路的条数了,直接给count[i][j]赋新值\*/

}

/\*spfa复杂度为O(kE),是Bell-Ford算法的优化,性能不稳定,优先使用Dijstra算法，Dijstra算法复杂度是O(V^2),本质是贪心(在稠密图时选取);Dijstra加堆(优先队列)，复杂度是O((V+E)logV)(在稀疏图时选取);Floyd算法复杂度为O(V^3),本质是DP\*/

1. 差分约束

/\*差分约束系统：如果一个系统由n个变量和m个约束条件组成，形成m个形如ai - aj <= k

的不等式(i,j)∈[1,n],k为常数),则称其为差分约束系统。亦即,差分约束系统是求解关于一组

变量的特殊不等式组的方法\*/

/\*记第i号牛的位置是d[i]，首先牛是按照编号顺序排列的,所以d[i] <= d[i+1]成立。

其次，对于关系好的牛之间的最大距离限制,都有d[AL]+DL>=d[BL]成立。同样，对于

每对关系不好的牛,都有d[AD]+DD<=d[BD]成立。因此，原问题就转化为在满足这三类不等式

的情况下，求解d的d[N]-d[1]的最大值的问题。

这些不等式的特点是所有的式子两边都只出现了一个变量，实际上，图的最短路问题也可以用这样的

形式表达出来。记从起点s出发,d[v]代表s到v的最短距离,那么对于每条权值为w的边(u,v)，

都有d[v] + w >= d[u],只有当w在s到u的最短路上时取到等号，所以d[v]-d[s]的最大值就是

s到v的最短距离。

可以把此问题的每一个约束不等式对应成图的一条边来构图，然后通过解决最短路问题来解决原问题。

首先把顶点编号从1～N，d[i]<=d[i+1]变形为d[i+1]+0 >= d[i]，因此从顶点i+1向顶点i连

一条权值为0的边.同样d[AL]+DL>=d[BL]对应从顶点AL向BL连一条权值为DL的边,d[AD]+DD <= d[BD]

对应顶点BD向顶点AD连一条权值为-DD的边，所求问题就是从顶点1到N的最短路。由于图存在负权边，

因此使用SPFA算法而不是Dijsktra算法\*/

#include<iostream>

#include<queue>

#include<cstring>

#define MAX\_V 1005

#define INF 0x3f3f3f3f

using namespace std;

int N,ML,MD,A,B,D;

struct edge{

int to;

int cost;

edge(int t,int c){

to = t;

cost = c;

}

};

vector<edge> G[MAX\_V];

int dis[MAX\_V];

int cnt[MAX\_V];

bool vis[MAX\_V];

bool spfa(){

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

memset(dis,INF,sizeof(dis));

memset(cnt,0,sizeof(cnt));

queue<int> q;

dis[1] = 0;

q.push(1);

vis[1] = true;

while(!q.empty()){

int cur = q.front();

q.pop();

vis[cur] = false;

for(int i = 0;i < G[cur].size();i++){

if(dis[G[cur][i].to]>dis[cur]+G[cur][i].cost){

dis[G[cur][i].to]=dis[cur]+G[cur][i].cost;

if(!vis[G[cur][i].to]){

q.push(G[cur][i].to);

vis[G[cur][i].to]=true;

if(++cnt[G[cur][i].to] > N)

return true; //入队次数超过N，存在负环,即无解

}

}

}

}

return false;

}

int main(){

ios::sync\_with\_stdio(0);

while(cin>>N>>ML>>MD){

for(int i = 0;i < MAX\_V;i++){

G[i].clear();

}

for(int i = 1;i < N;i++){

G[i+1].push\_back(edge(i,0));

}

for(int i = 0;i < ML;i++){

cin>>A>>B>>D;

G[A].push\_back(edge(B,D));

}

for(int i = 0;i < MD;i++){

cin>>A>>B>>D;

G[B].push\_back(edge(A,-D));

}

if(!spfa()){

int ans = dis[N] - dis[1];

if(ans == INF)

cout<<-2<<endl;

else

cout<<dis[N] - dis[1]<<endl;

}

else

cout<<-1<<endl;

}

}

1. 次短路与第 K 短路

//求顶点1到N的次短路

#include<cstdio>

#include<vector>

#include<queue>

#include<utility>

#include<algorithm>

#define MAX\_N 5005

#define INF 0x3f3f3f3f

using namespace std;

typedef pair<int,int> P;

int N,R;

//利用Dijkstra加堆算法，对于每一个顶点不仅记录最短距离，还记录次短距离

struct edge{

int to;

int cost;

edge(int t,int c){

to = t;

cost = c;

}

};

vector<edge> G[MAX\_N];

int dist[MAX\_N]; //最短路

int dist2[MAX\_N]; //次短路

void solve(){

priority\_queue<P,vector<P>,greater<P> > que;

fill(dist,dist + N,INF);

fill(dist2,dist2 + N,INF);

dist[0] = 0;

que.push(P(0,0));

while(!que.empty()){

P p = que.top();

que.pop();

int v = p.second,d = p.first;

if(dist2[v] < d) continue;

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

edge &e = G[v][i];

int d2 = d + e.cost;

if(dist[e.to] > d2){

swap(dist[e.to],d2);

que.push(P(dist[e.to],e.to));

}

if(dist2[e.to] > d2 && dist[e.to] < d2) {

dist2[e.to] = d2;

que.push(P(dist2[e.to],e.to));

}

}

}

printf("%d\n",dist2[N-1]);

}

int main(){

while(~scanf("%d%d",&N,&R)){

int A,B,D;

while(R--){

scanf("%d%d%d",&A,&B,&D);

G[A-1].push\_back(edge(B-1,D));

G[B-1].push\_back(edge(A-1,D));

}

solve();

}

return 0;

}

(三)搜索

1、A\*搜索

2、ID A\*搜索

3、记忆化搜索

1. 最近公共祖先(LCA)

4

5 2

3 1 6

7 8 9

index 1 2 3 4 5 6 7 8 9 10 ......//遍历步数,也就是数组下标

depth 1 2 3 4 3 2 3 4 3 2 ......//深搜遍历时每一步的深度

occur 4 5 3 7 3 5 1 8 1 5 ......//每一步访问的结点编号

first 7 3 1 2 4 8 ......//每一个结点第一次被遍历到的步数

/\*先DFS得到depth、occur和first数组,要查询i,j的LCA,就先在first中查询i,j第一次被访问的步数;

在depth数组的[first[i],first[j]]区间用RMQ找到最小深度,对应位置的occur数组的值就是LCA\*/

#include <iostream>

#include <cstring>

#include <cmath>

#include <cstdio>

#include <algorithm>

#include <vector>

#define mem(a, b) memset(a, b, sizeof(a))

using namespace std;

#define maxn 10005

vector<int> edge[maxn];//邻接表

int first[maxn];//存放每个结点第一次被访问到的步数

int depth[maxn << 1];//存放每一步访问的深度

int occur[maxn << 1];//存放每一步访问的结点下标,每个结点可能遍历两遍,所以depth和occur数组开maxn<<1

int index;

void dfs(int num, int deep) {//num是该步访问的结点编号

occur[++index] = num;//index是数组下标也就是步数

depth[index] = deep;

if(!first[num])

first[num] = index;

for(int i = 0; i < edge[num].size(); i++) {

int v = edge[num][i];

dfs(v, deep + 1);

occur[++index] = num;//回溯也要标记occur和depth

depth[index] = deep;

}

}

int dp[maxn << 1][20];

void rmq() {

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

dp[i][0] = i;

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

for(int i = 1; i + (1 << j) - 1 <= index; i++) {

dp[i][j] = depth[dp[i][j - 1]] < depth[dp[i + (1 << (j - 1))][j - 1]] ? dp[i][j - 1] : dp[i + (1 << (j - 1))][j - 1];

}//比较的是深度depth,dp里存放的是对应的结点编号

}

int query(int x, int y) {

int l = first[x], r = first[y];

if(l > r) swap(l, r);

int k = (int)(log((double)(r - l + 1)) / log(2.0));

int mmin = depth[dp[l][k]] < depth[dp[r - (1 << k) + 1][k]] ? dp[l][k] : dp[r - (1 << k) + 1][k];

/\*比较的是深度,返回的是结点\*/

return occur[mmin];

}

bool in[maxn];

int main() {

int T;

scanf("%d", &T);

while(T--) {

int n;

scanf("%d", &n);

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

edge[i].clear();//注意要清空vector

mem(in, 0);//初始化数组

mem(depth, 0);

mem(dp, 0);

mem(first, 0);

for(int i = 1; i < n; i++) {//采用邻接表存储,将子结点存放到以父结点为表头结点的vector中

int num, v;

scanf("%d%d", &num, &v);

edge[num].push\_back(v);

in[v] = true;//把非根结点置为true

}

index = 0;//初始化index

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

if(!in[i]) {//in[i]为false说明是根节点,从此开始DFS

dfs(i, 0);

break;

}

rmq();

int l, r;

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

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

}

}

(五)最小生成树(MST) //最大生成树可以把边权取反后用最小生成树算法解

1.Prim(没什么卵用)

1. Kruskal

/\*Kruskal算法求MST(复杂度O(ElongE)),MST问题一般不用Prim算法(复杂度高)\*/

/\*MST性质:1、所有不同 MST的边权集合相同 2、MST的最大边是所有生成树中最小的\*/

#include<iostream>

#include<cstdio>

#include<string.h>

#include<algorithm>

using namespace std;

const int MAXN=505;//最大点数

const int MAXM=250050;//最大边数

int f[MAXN];//并查集使用

struct Edge

{

int from,to,w;

}edge[MAXM];//储存边的信息，包括起点/终点/权值

int cnt;//边数，加边前赋值为0

void addedge(int from,int to,int w)

{

edge[cnt].from=from;

edge[cnt].to=to;

edge[cnt++].w=w;

}

bool cmp(Edge a,Edge b)//排序函数，边按照权值从小到大排序

{

return a.w<b.w;

}

int Find(int x){

return f[x]==x?x:f[x]=Find(f[x]);

}

void Union(int x,int y){

x = Find(x);

y = Find(y);

if(x!=y)

f[y] = x;

}

int Kruskal(int n)//传入点数，返回最小生成树的权值，如果不连通返回-1

{

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

f[i]=i;

sort(edge,edge+cnt,cmp);

int tot=0;//计算加入的边数

int ans=0;

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

{

if(Find(edge[i].from)==Find(edge[i].to))

continue;

Union(edge[i].from,edge[i].to);

tot++;

ans+=edge[i].w;

if(tot==n-1)

break;

}

if(tot<n-1)

return -1;//不连通

else

return ans;

}

int main()

{

int T;

cin>>T;

int n;

int c;

while(T--)

{

cin>>n;

cnt=0;

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

{

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

{

cin>>c;

addedge(i,j,c);//建立无向图

}

}

cout<<Kruskal(n)<<endl;

}

return 0;

}

// 朱刘算法

// 最小树形图

#define inf 0x3f3f3f3f

#define N 100

int in[N];

int use[N];

int Hash[N];

int pre[N];

struct Edge{

int u;

int v;

int w;

}

int mini\_tree(int root, int n, int m){ // 树根, 结点数, 边数 start from 1

int ans = 0;

while(true){

memset(in, inf, sizeof in);

for(int i = 1; i <= m; ++i){ // 找最小入边

int u = edge[i].u;

int v = edge[i].v;

if(edge[i].w < in[v] && u != v){

in[v] = edge[i].w;

pre[v] = u;

}

}

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

if(i == root) continue;

ans += in[i]; // 求边权和

if(in[i] == inf) return -1; // 不存在最小树形图

}

memset(Hash, -1, sizeof Hash);

memset(use, -1, sizeof use);

int cnt = 0; // 环数量

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

int v = i;

while(v != root && use[v] != i && Hash[v] == -1){

// use 标记从i点出发能遍历到的点

// Hash 通过哈希映射缩点

use[v] = i;

v = pre[v];

}

if(v != root && Hash[v] == -1){ // 缩点

++cnt;

Hash[v] = cnt;

for(int u = pre[v]; u != v; u = pre[u]){

Hash[u] = cnt;

}

}

}

if(cnt == 0) // 完成

break;

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

if(Hash[i] == -1)

Hash[i] = ++cnt;

}

for(int i = 1; i <= m; ++i){ //建立新图

int u = edge[i].u;

int v = edge[i].v;

edge[i].u = Hash[u];

edge[i].v = Hash[v];

if(edge[i].u != edge[i].v){

edge[i].w -= in[v];

}

}

n = cnt; // 更新结点数

root = Hash[root];

}

return ans;

}

(六)树上差分

(七)树状倍增

(八)二分图

1. 二分图最大匹配(匈牙利算法)

/\*该算法的核心就是寻找增广路径，它是一种用增广路径求二分图最大匹配的算法。

明确两个概念：

交替路：从一个未匹配点出发，依次经过非匹配边、匹配边、非匹配边…形成的路径叫交替路。

增广路：从一个未匹配点出发，走交替路，如果途径另一个未匹配点（出发的点不算），则这条交替路称为增广路

匈牙利算法的原理：循环遍历二分图的两个点集的其中一个，每次从该点集的一个未匹配点出发寻找增广路，由于增广路里未匹配边比已匹配边多一条，

于是修改匹配图，把路径里所有匹配过的连线去掉匹配关系，把未匹配边改成匹配的，这样匹配数就比原来多了1个。不断执行上述操作，直到找不到增广路为止。

二分图的最大匹配就找到了。

\*/

bool vis[maxn];// vis数组保证在同一条增广路中不重复访问顶点，即不成环，每找一条增广路都要清空vis数组

vector<int> edge[maxn];

int matching[maxn]; //matching[i]存放i号女生匹配的男生

//建图时建有向图即可,匹配时只用算二分图的一边的匹配

bool Find(int x){//返回true表示x能找到匹配

for(int i = 0;i < edge[x].size();i++){

if(!vis[edge[x][i]]){

vis[edge[x][i]] = true;

if(matching[edge[x][i]] == -1 || Find(matching[edge[x][i]])){ /\*如果edge[x][i]号女生还没有匹配，就和x匹配;

或者和她匹配的男生能找到其他匹配的女生，也给x号男生腾出位置让edge[x][i]号女生和x号男生匹配，这就是一个寻找增广路的过程，

在递归回溯的过程中就对未匹配边和已匹配边进行了取反，于是匹配数加一\*/

matching[edge[x][i]] = x;

return true;

}

}

}

return false;

}

int match(){//求解最大匹配

int sum = 0;

for(int i = 1;i <= n;i++){ //遍历男生找增广路进行匹配

memset(vis,0,sizeof(vis)); //每次找一条增广路都要清空vis数组，因为同一条增广路中不能重复选择顶点，但不同增广路可以选择重复的顶点

if(Find(i))

sum++; //找到一条增广路就增加一个匹配

}

return sum;

}

/\*

最大匹配数：最大匹配的匹配边的数目

最小点覆盖数：选取最少的点，使任意一条边至少有一个端点被选择

最大独立集数：选取最多的点，使任意所选两点均不相连

最小路径覆盖数：对于一个 DAG（有向无环图），选取最少条路径，使得每个顶点属于且仅属于一条路径。路径长可以为 0（即单个点）。

定理1：最大匹配数 = 最小点覆盖数（这是 Konig 定理）

定理2：最大独立集数 = 顶点数 - 最大匹配数

定理3：最小路径覆盖数 = 顶点数 – 最大匹配数\*/

2.二分图最小顶点覆盖数

3.有向无环图(DAG)的最小路径覆盖数

1. 二分图的最大独立集数
2. 判断一个图是否是二分图

//方法一：dfs对图进行染色，保证相邻顶点染成不同颜色，如果最小着色数为2,则该图为二分图

//输入

vector<int> G[MAX\_V];//图

int V;//顶点数

int color[MAX\_V]; //顶点i的颜色,-1和1分别代表两种不同的颜色,0表示未染色

//把顶点染成-1或1

bool dfs(int v,int c) {

color[v] = c;

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

//如果相邻的顶点同色，则返回false

if(color[G[v][i]] == c) return false;

//如果相邻的顶点还没被染色，则染成-c

if(color[G[v][i]] == 0 && !dfs(G[v][i],-c))

return false;

}

//如果所有顶点都染过色且没有颜色冲突，则返回true

return true;

}

void solve(){

for(int i = 0;i < V;i++){

if(color[i] == 0){

//如果顶点i还没被染色，则染成1

if(!dfs(i,1)){

printf("No\n");

return;

}

}

}

printf("Yes\n");

}

/\*方法二：BFS染色判定\*/

#include<iostream>

#include<string.h>

#include<queue>

#include<vector>

using namespace std;

const int maxn = 1e4+5;

int color[maxn];

vector<int> d[maxn];

bool judge(int k){

queue<int> Q;

color[k] = 1;

Q.push(k);

while(!Q.empty()){

int now = Q.front();

Q.pop();

for(int i=0;i<d[now].size();i++){

int x = d[now][i];

if(color[x]==0){

color[x] = - color[now];

Q.push(x);

}

else if(color[x]==color[now]) return false;

}

}

return true;

}

int main(){

int t;

cin>>t;

while(t--){

int n,m;

cin>>n>>m;

memset(color,0,sizeof(color));

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

d[i].clear();

}

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

int a,b;

cin>>a>>b;

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

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

}

int i = 1;

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

if(color[i]==0){

if(!judge(i)) break;

}

}

if(i>n) cout<<"Correct"<<endl;

else cout<<"Wrong"<<endl;

}

}

(九)网络流

1.Dinic 算法

2.最小费用最大流

1. 最小割
2. 树链剖分

// 树链剖分

// SPOJ Query on a tree

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cmath>

#define MAXN 10005

using namespace std;

struct Node{

int to;

int next;

}edge[MAXN << 1];

int head[MAXN], edge\_num;

int top[MAXN]; // top[i] -> i 所在重链的顶端

int fa[MAXN];

int son[MAXN]; // 重儿子

int pos[MAXN]; // pos[i] -> i 与其父节点连边在线段树中的位置

int ffpos[MAXN];

int num[MAXN]; // num[i] -> i 节点子树的节点个数

int depth[MAXN];

int p;

void dfs\_1(int u, int pre, int dep){

depth[u] = dep;

fa[u] = pre;

num[u] = 1;

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

int v = edge[i].to;

if(v != pre){

dfs\_1(v, u, dep + 1);

num[u] += num[v];

if(son[u] == -1 || num[v] > num[son[u]])

son[u] = v;

}

}

}

void get\_pos(int u, int st){

top[u] = st;

if(son[u] != -1){

pos[u] = p++;

ffpos[pos[u]] = u;

get\_pos(son[u], st); // 保证重链标号连续性

}

else{

pos[u] = p++;

ffpos[pos[u]] = u;

return;

}

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

int v = edge[i].to;

if(v != son[u] && v != fa[u])

get\_pos(v, v); // 新链起点

}

}

void add\_edge(int u, int v){

edge[edge\_num].to = v;

edge[edge\_num].next = head[u];

head[u] = edge\_num;

++edge\_num;

}

// 线段树

struct segNode{

int l, r;

int MAX;

}segTree[MAXN << 2];

void build(int node, int l, int r){

segTree[node].l = l;

segTree[node].r = r;

segTree[node].MAX = 0;

if(l == r) return;

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

build((node << 1), l, mid);

build((node << 1) | 1, mid + 1, r);

}

void push\_up(int node){

segTree[node].MAX = max( segTree[node << 1].MAX,

segTree[(node << 1) | 1].MAX);

}

void update(int node, int k, int val){ // 单点更新

if(segTree[node].l == k && segTree[node].r == k){

segTree[node].MAX = val;

return;

}

int mid = (segTree[node].l + segTree[node].r) >> 1;

if(k <= mid)

update((node << 1), k, val);

else update((node << 1) | 1, k, val);

push\_up(node);

}

int query(int node, int l, int r){

if(segTree[node].l == l && segTree[node].r == r)

return segTree[node].MAX;

int mid = (segTree[node].l + segTree[node].r) >> 1;

if(r <= mid)

return query(node << 1, l, r);

else if(l > mid) return query((node << 1) | 1, l, r);

else

return max(query(node << 1, l, mid),

query((node << 1) | 1, mid + 1, r));

}

int solve(int u, int v){

int f1 = top[u];

int f2 = top[v];

int res = 0;

while(f1 != f2){

if(depth[f1] < depth[f2]){

swap(f1, f2);

swap(u, v);

}

res = max(res, query(1, pos[f1], pos[u]));

u = fa[f1];

f1 = top[u];

}

if(u == v) return res;

if(depth[u] > depth[v]) swap(u, v);

res = max(res, query(1, pos[son[u]], pos[v]));

return res;

}

int tmp[MAXN][3];

int main(){

int T, n;

int u, v;

char op[10];

scanf("%d", &T);

while(T--){

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

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

edge\_num = 0;

scanf("%d", &n);

for(int i = 0; i < n - 1; ++i){

scanf("%d%d%d", &tmp[i][0], &tmp[i][1], &tmp[i][2]);

add\_edge(tmp[i][0], tmp[i][1]);

add\_edge(tmp[i][1], tmp[i][0]);

}

p = 0;

dfs\_1(1, 0, 0);

get\_pos(1, 1);

build(1, 0, p - 1);

for(int i = 0; i < n - 1; ++i){

if(depth[tmp[i][0]] > depth[tmp[i][1]]){

swap(tmp[i][0], tmp[i][1]);

}

update(1, pos[tmp[i][1]], tmp[i][2]);

}

while(scanf("%s", &op)){

if(op[0] == 'D') break;

scanf("%d%d", &u, &v);

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

printf("%d\n", solve(u, v));

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

update(1, pos[tmp[u - 1][1]], v);

}

}

}

}

(十一)拓扑排序

/\*

拓扑排序的解法：

方法1：寻找入度为0的点

1、读入时记录每个点的入度。

2、遍历出所有入度为0的点，加入队列。

3、从队列里依次选择一个点，删除它的所有出边，将边连向的点入度减1。

4、执行3操作时判断入度减1后是否入度变为0，如果入度为0，加入队列。

5、重复执行3、4操作至所有点全部被处理。

\*/

//关键代码：

/\*如果最后输出的拓扑序的顶点数小于总顶点数，则表明存在有向环；

反之，则是DAG，如果任意时刻队列中元素个数大于1,表示有多个拓扑序

如果任意时刻队列中元素的个数始终为1,则有唯一的拓扑序\*/

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

if(in[i]==0)

q.push(i);

while(!q.empty()){

int u = q.front();

q.pop();

//printf("%d ",u);在这个时候输出要求的拓扑序

int len = G[u].size();

for(int i = 0;i < len;i++){

int to = G[u][i];

in[to]--;

if(in[to] == 0)

q.push(to);

}

}

/\*

方法2：dfs

其实就是在dfs的时候，对于一个点，在返回上一个调用处时，将其加入一个数组。

最后答案就是这个数组的逆序。考虑正确性，对于一个点v,它将指向许多点，当这些点

已经处理完后，v将返回上一个指向它的点的u,因为v指向的点必然在其后面，而u必然在它前面，

所以依次加入他们是符合拓扑序的，只不过是逆序的。

\*/

//关键代码：

void dfs(int u){

int len = G[u].size();

for(int i = 0;i < len;i++){

int to = G[u][i];

if(!vis[to]){

vis[to] = true;

dfs(to);

}

}

ans[++num] = u;//注意答案是这个数组的逆序

}

int main(){

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

if(!vis[i]){

vis[i]=true;

dfs(i);

}

}

七、数论

1. 中国剩余定理

/\*求解同余方程组:

x ≡a1(mod m1) 等价于 x= a1 + k1m1(1)

x ≡a2(mod m2) 等价于 x= a2 + k2m2(2)(其中m1和m2互质)

联立消去x,得 a1 + k1m1 = a2 + k2m2

其中 k1 和 k2 为未知数,即可看作求解

m1x-m2y=a2-a1 求得一组解记为k1',k2'

k1'代入(1)或k2'代入(2),得一个解记为x0.

又根据扩展欧几里得,k1通解 = k1'+m2t，k2通解 = k2'+m1t

k1通解代入(1)得x=x0+m1m2t,证明完毕

将两个方程推广到n个方程：

对于两两互质得m1,m2,...,mn

有可合并成一个同余方程x ≡x0(mod m1m2...mn)

即x=x0+m1m2..mnt\*/

/\*中国剩余定理(互质):

设正整数m1,m2,...,mk两两互素,则同余方程组

x ≡a1 (mod m1)

x ≡a2 (mod m2)

x ≡a3 (mod m3)

...............

x ≡ak (mod mk)

有整数解。并且在模 M = m1m2...mk下的解是唯一的

x=(a1M1M1^-1+a2M2M2^-1+...aKMKMK^-1)mod M

其中 Mi = M/mi,Mi^-1为Mi模mi的逆元\*/

/\*中国剩余定理(不互质):

设正整数m1,m2,...,mk,不一定互素,则同余方程组

x ≡a1 (mod m1)

x ≡a2 (mod m2)

x ≡a3 (mod m3)

...............

x ≡ak (mod mk)

有整数解

x≡x0(mod lcm(m1,m2,...,mk))

\*/

int CRT(int a[],int m[],int n)/\*计算 x = a[i] mod m[i], 一共n个方程,

m[]和 a[]的下标从 1 开始\*/

{

int M = 1;

int ans = 0;

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

M \*= m[i];

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

{

int x, y;

int w = M / m[i];

exgcd(w, m[i], x, y);/\*求解 w模m[i]的逆元赋值给x;因为wx ≡1(modm[i] )等价于求解

wx+m[i]y=1(因为w和m[i]必互质),gcd(w,m[i])=1\*/

ans += w\*x\*a[i];

ans %= M;

}

if(ans < 0) ans += M;

return ans;

}

ll CRT(ll a[], ll m[], ll n) {// 计算 x = a[i] mod m[i], 一共n个方程,m[]和 a[]的下标从 0开始

ll x0 = a[0], mod = m[0];

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

ll k1, k0, r = a[i] - x0;//每次循环合并两个方程

ll gcd = exgcd(m[i], mod, k1, k0);

if((a[i] - x0)%gcd != 0) return -1;

k0 = r/gcd\*k0;

ll res = m[i]/gcd;

k0 = (k0%res + res)%res;

x0 = x0 + k0\*mod;

mod = lcm(mod, m[i], gcd);

x0 %=mod;

}

return x0;

}

1. 二次剩余
2. 欧几里得

1.费马小定理

1. 扩展欧几里得

/\*扩展欧几里得算法,求解不定方程,模线性方程（线性同余方程）,模的逆元等\*/

/\*对于不定方程 a\*x+b\*y=c

1 判断是否有解 整数二元一次不定方程有解的充分必要是c|gcd(a,b)。如果不能整除则无解。

2 扩展欧几里德求特解 欧几里德给出了计算a\*x+b\*y=gcd(a,b)的解法

3 通解

a\*x+b\*y=gcd(a,b)在有解并求出特解(x,y)的情况下,通解可以表示为

对于一般式a\*x+b\*y=c,如果有解,只需把a\*x+b\*y=gcd(a,b)的特解乘上c/gcd(a,b)即可得到其特解,即通解公式里的x0,y0

x = x0 + (b/gcd)\*t

y = y0 –(a/gcd)\*t

求x的最小正整数解的方法

while(x<0)

x+=b,y-=a;

4 例子

求出a\*x+b\*y=gcd(a,b)的特解和一个通解

int main()

{

long long a,b,c;

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

long long x,y,d;

d=exgcd(a,b,x,y);

if (c%d!=0) printf("No solution!\n");

else

{

a/=d,b/=d,c/=d;

x\*=c,y\*=c;

printf("特解: %lld %lld\n",x,y);

printf("一个通解: %lld %lld\n",x+b,y-a);

}

return 0;

}\*/

/\*1.使用扩展欧几里德算法解决不定方程的办法:

a\*x+b\*y=gcd(a,b)在有解并求出特解(x,y)的情况下,通解可以表示为

对于一般式a\*x+b\*y=c,如果有解,只需把a\*x+b\*y=gcd(a,b)的特解乘上c/gcd(a,b)即可得到其特解,即通解公式里的x0,y0

x = x0 + (b/gcd)\*t

y = y0 –(a/gcd)\*t

设s= b/gcd (x两个相邻解的最小距离）则x的最小正整数解为xx=(x0%s+s)%s

一般求x的最小正整数解的方法

while(x<0)

x+=b,y-=a;\*/

/\*2.用扩展欧几里德算法求解模线性方程的方法：

同余方程 ax≡b (mod n)对于未知数 x 有解，

当且仅当 gcd(a,n) | b（|表示整除）。且方程有解时，方程有gcd(a,n)个解。

求解方程 ax≡b (mod n)相当于求解方程 ax=b+kn,等价于ax+ny=b, (x, y为整数)

设 d= gcd(a,n)，假如整数 x 和 y，满足 d= ax+ ny(用扩展欧几里德得出)。如果 d|b，则方程

a\* x0+ n\* y0= d， 方程两边乘以 b/ d，(因为 d|b，所以能够整除)，得到 a\* x0\* b/ d+ n\* y0\* b/ d= b。

所以 x= x0\* b/ d，

y= y0\* b/ d 为 ax+ ny= b 的一个解，所以 x= x0\* b/ d 为 ax=b (mod n ) 的解。

ax≡b (mod n)的一个解为 x0= x\*(b/d)mod n，且方程的 d 个解分别为 xi= (x0+i\*(n/d))mod n {i= 0... d-1}。

设s=n/d;

方程ax≡b (mod n)的最小正整数解为：(x%s+s)%s; 其中x为ax+ny=b的解\*/

/\*求解逆元

bx ≡1(mod p)等价于bx+py=1,x称为b模p的逆元\*/

//递归算法

int exgcd(int a,int b,int &x,int &y)

{

if(b==0)

{

x=1;

y=0;

return a;

}

int r=exgcd(b,a%b,x,y);

int t=x;

x=y;

y=t-a/b\*y;

return r; ///返回值为最大公约数,x,y是a\*x+b\*y=gcd(a,b)的一个特解,都要乘c/gcd(a,b)才是a\*x+b\*y=c的特解

}

//非递归算法

int exgcd(int m,int n,int &x,int &y)

{

int x1,y1,x0,y0;

x0=1; y0=0;

x1=0; y1=1;

x=0; y=1;

int r=m%n;

int q=(m-r)/n;

while(r)

{

x=x0-q\*x1; y=y0-q\*y1;

x0=x1; y0=y1;

x1=x; y1=y;

m=n; n=r; r=m%n;

q=(m-r)/n;

}

return n;

}

1. 最大公约数和最小公倍数(包括多个数的最大公约数和最小公倍数)

// gcd(a,b)等于gcd(b,a Mod b)

int gcd(int a, int b)

{

if (b==0)

return a;

else

return gcd(b, a%b);

}

// 先求a[0],a[1]的公约数，再求其与a[2]的公约数，依次类推

int ngcd(int \*a, int n)

{

if (n==1)

return \*a;

else

return gcd(a[n-1], ngcd(a, n-1));

}

// ab的最小公倍数=ab的乘积除以最大公约数

int lcm(int a, int b)

{

return a/gcd(a,b)\*b;//先除后乘,防溢出

}

// 与ngcd类似

int nlcm(int \*a, int n)

{

if (n==1)

return \*a;

else

return lcm(a[n-1], nlcm(a, n-1));

}

1. 素数筛
2. 埃氏筛法

int prime[maxn];

Int isprime[maxn];

void make\_prime(int N) { //一般线性筛法(埃氏筛法)

memset(isprime, true, sizeof(isprime));

isprime[0]=false;

isprime[1]=false;

int cnt=0;

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

if (isprime[i]) { //isprime[i]==1表示i没有被前面的素数筛过,即是素数

prime[cnt++]=i; //prime记录素数

for (int k=2\*i; k<N; k+=i)

isprime[k]=false;

}

}

1. 欧拉筛

void make\_prime(int N) { //快速线性筛法(欧拉筛)

memset(isprime, true, sizeof(isprime));

isprime[0]=false;

isprime[1]=false;

int cnt=0;

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

if (isprime[i]) {

prime[cnt++]=i; //prime记录素数

}

for(int j=0;j<cnt;j++){

if(i\*prime[j]>=N) break;

isprime[i\*prime[j]]=false;//用刚筛出的素数和之前筛出的素数相乘得一个合数,筛掉该合数

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

}

}

}

1. 区间筛法

typedef long long ll;

bool is\_prime[MAX\_L];

bool is\_prime\_small[MAX\_SQRT\_B];

//对区间[a,b)内的整数执行筛法。is\_prime[i-a] = true <=> i是素数

void segment\_sieve(ll a,ll b){

for(int i = 0;(ll)i \* i < b;i++) is\_prime\_small[i] = true;

for(int i = 0;i < b - a;i++) is\_prime[i] = true;

for(int i = 2;(ll) i \* i < b;i++){

if(is\_prime\_small[i]){

for(int j = 2 \* i;(ll)j \* j < b;j += i) is\_prime\_small[j] = false;

//筛[2,√b)

for(ll j = max(2LL,(a + i - 1) / i) \* i;j < b;j += i)

is\_prime[j - a] = false; //筛[a,b)

}

}

}

1. 米勒罗宾素性测试(miller-rabin)

(六)欧拉函数

1. 打表法

//筛法求欧拉函数

void euler(int n){

phi[1]=1;

for(int i=2;i<n;i++){

phi[i]=i;

}

for(int i=2;i<n;i++){

if(phi[i]==i){//phi[i]==i说明i没有被筛过,即i是质数

for(int j=i;j<n;j+=i) //筛掉以i为素因子的数

phi[j]=phi[j]/i\*(i-1);//先进行除法是为了防止中间数据的溢出

}

}

}

1. 直接求法

//直接求解欧拉函数

int euler(int n){ //返回euler(n)

int res=n,a=n;

for(int i=2;i\*i<=a;i++){ //合数 a 的最大质因数小于 根号 a

if(a%i==0){

res=res/i\*(i-1);//先进行除法是为了防止中间数据的溢出

while(a%i==0) a/=i;

}

}

if(a>1) res=res/a\*(a-1);

return res;

}

1. 原根
2. 欧拉函数降幂

IMG_256

例题：Super A^B mod C

#include <iostream>

#include <cstdio>

#include <cstring>

#include <algorithm>

using namespace std;

typedef long long ll;

ll phi(ll n) { //直接求解欧拉函数

int res = n, a = n;

int i;

for(i = 2; i \* i <= a; i++) {

if(a % i == 0) {

res -= res / i;

while(a % i == 0) a /= i;

}

}

if(a > 1) res -= res / a;

return res;

}

ll pw(ll a,ll b,ll mod) {//快速幂取模

a = a%mod;

ll ans = 1;

while(b) {

if(b&1 == 1) {

ans = ans\*a;

ans %= mod;

}

a = a\*a;

a %= mod;

b>>=1;

}

return ans;

}

int main() {

ll a, c;

string b;

while(cin>>a>>b>>c) {

ll phic = phi(c);

a %= c;

int i, len = b.size();

ll res = 0; //欧拉降幂,当指数特别大的时候要欧拉降幂

for(i = 0; i < len; i++) {

res = res \* 10 + b[i] - '0'; //根据模运算的性质可以把大数取模拆分成若干数取模的和

res %= phic;

}

cout<<pw(a, res + phic, c)<<endl;

}

return 0;

}

(七)分解质因数

1. 枚举因子
2. 素数筛
3. Pollard-rho 分解质因数

(八)离散对数

1. Baby Step Giant Step
2. Ex-baby step giant step

(九)原根

1. 求原根
2. 原根应用
3. K 次方根
4. 快速数论变换(NTT)

(十)快速幂

1. 快速幂

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

ll pw(ll a,ll b,ll mod) {

a = a%mod;

ll ans = 1;

while(b) {

if(b&1 == 1) {

ans = ans\*a;

ans %= mod;

}

a = a\*a;

a %= mod;

b>>=1;

}

return ans;

}

1. 快速乘

LL quick\_mul(LL x,LL y,LL MOD){//O(1)快速乘

x=x%MOD,y=y%MOD;

return((x\*y-(LL)(((long double)x\*y+0.5)/MOD)\*MOD)%MOD+MOD)%MOD;

}

1. 矩阵快速幂

typedef struct{

int a[2][2];

}mat;

mat matMultiply(mat x,mat y)

{

mat ans;

for(int i=0;i<2;i++){

for(int j=0;j<2;j++){

ans.a[i][j]=0;

for(int k=0;k<2;k++){

ans.a[i][j]+=x.a[i][k]\*y.a[k][j]%MOD;

}

ans.a[i][j]%=MOD;

}

}

return ans;

}

mat pow(int n,mat x){

mat res;

memset(res.a,0,sizeof(res.a));

for(int i=0;i<2;i++)

res.a[i][i]=1;

while(n)

{

if(n&1) res=matMultiply(res,x);

x=matMultiply(x,x);

n=n>>1;

}

return res;

}

1. 康托展开

/\*康托展开表示的是当前排列在n个不同元素的全排列中的名次。比如213在这3个数所有排列中排第3。

那么，对于n个数的排列，康托展开为：ans = a1 \* (n - 1)! + a2 \* (n - 2)! + ... + a(n-1) \* 1! + an \* 0!

其中ai表示第i个数在未出现过的数中排第几，举个简单的例子：

对于排列4213来说，4在4213中排第3，注意从0开始，2在213中排第1，1在13中排第0，3在3中排第0，即：

3 \* (4 - 1)! + 1 \* (3 - 1)! + 0 \* (2 - 1)! + 0 \* (1 - 1)! = 20，这样得到4213在所有排列中排第ans = 20\*/

const int FAC[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880}; // 阶乘

//代码实现：(从0开始计数)

//康托展开

int cantor(string str){

int len = str.size();

int ans = 0;

for(int i = 0; i < len; i++){

int tmp = 0;

for(int j = i + 1; j < len; j++)

if(str[j] < str[i])

tmp++;

ans += tmp \* FAC[len - i - 1];

}

return ans; //返回的是有几个排列小于当前排列，加1得当前是第几个排列

}

/\*康托展开的逆运算：就是根据某个排列的在总的排列中的名次来确定这个排列。

比如：求1234所有排列中排第20，就每次除以阶乘得到康托展开的系数，剩下的

取模继续运算。康托展开逆运算，n为排列长度，k为第几个排列\*/

string revCantor(int n, int k) {

k--; //把序号减1为康托展开的值，然后进行逆康托展开

vector<int> v; // 存放当前可选数

string str = ""; // 所求排列组合

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

v.push\_back(i);

for(int i = n;i >= 1;i--){

int now = k / FAC[i - 1];

int next = k % FAC[i - 1];

k = next;

str += (v[now] + '0');

v.erase(v.begin() + now);

}

return str;

}

(十二)快速傅氏变换(FFT)

(十三)高斯消元

1. 普通高斯消元
2. 二进制高斯消元(xor 高斯消元)

(十四)生成树计数

1. 度数矩阵
2. 基尔霍夫矩阵(拉普拉斯矩阵)
3. 基尔霍夫定理(Matrix Tree)定理

(十五)矩阵行列式

1. 组合数

1.打表法

/\*C(n, m) = C(n -1, m - 1) + C(n - 1, m)\*/

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

C[i][0] = 1 % P;

for (int j = 1; j <= i; ++j) {

C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % P;

}

}

2.Lucas大组合数

#include <cstdio>

#include <cstring>

#include <algorithm>

using namespace std;

typedef long long ll;

ll mod\_pow(ll x, ll n, ll p){

ll res = 1;

while(n){

if(n & 1) res =res \* x % p;

x = x \* x % p;

n >>= 1;

}

return res;

}

ll comb(ll n, ll m, ll p){

if(m > n) return 0;

ll ret = 1;

m = min(n - m, m);

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

ll a = (n + i - m) % p;

ll b = i % p;

ret = ret \* (a \* mod\_pow(b, p - 2, p) % p) % p;

}

return ret;

}

ll Lucas(ll n, ll m, ll p){

if(m == 0) return 1;

return comb(n % p, m % p, p) \* Lucas(n / p, m / p, p) % p;

}

int main(){

int T;

ll n, m, p;

scanf("%d", &T);

while(T--){

scanf("%I64d%I64d%I64d", &n, &m, &p);

printf("%I64d\n", Lucas(n, m, p));

}

return 0;

}

1. 预处理阶乘逆元法（如果数不是很大用此法，比Lucas快）

LL inv[MAXN];

LL fac[MAXN];

LL Com(int n,int m){

return fac[n]\*inv[m]%mod\*inv[n-m]%mod;

}

void init(){

inv[0]=fac[0]=1;

inv[1]=1;

for(int i=1;i<MAXN;i++){

fac[i]=fac[i-1]\*i%mod;

}

inv[1]=1;

for(int i=2;i<MAXN;i++){

inv[i]=(LL)(mod-mod/i)\*inv[mod%i]%mod;

}

inv[0]=1;

for(int i=1;i<MAXN;i++){

inv[i]=inv[i-1]\*inv[i]%mod;

}

}

1. 全错排公式

d[1]=0;   d[2]=1 d[n]= (n-1)\*( d[n-1] + d[n-2])

//d[i]表示i个元素的全错排种数

八、博弈论

1. 巴什博弈

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。

#include<stdio.h>

typedef long long ll;

int main()

{

int T;

scanf("%d",&T);

while(T--)

{

ll n,k;

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

if(n%(k+1)==0)

printf("B\n");

else printf("A\n");

}

return 0;

}

1. 威佐夫博弈

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。

#include <cstdio>

#include <cstdlib>

#include <cmath>

using namespace std;

int main(){

int a,b,k,a\_k;

while(scanf("%d%d",&a,&b)!=EOF){

k = abs(a-b);

a = a < b? a : b;

a\_k = floor(k\*(1.0 + sqrt(5.0))/2);

printf("%d\n",a!=a\_k);

//输出为0，说明该点为必败点，1为必胜点

}

return 0;

}

1. Nim 博弈

有若干堆石子，每堆石子的数量都是有限的，合法的移动是“选择一堆石子并拿走若干颗（不能不拿）”，如果轮到某个人时所有的石子堆都已经被拿空了，则判负

#include <iostream>

using namespace std;

int main()

{

int n;

int sum;

int i;

while(cin>>n)

{

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

{

int a;

cin>>a;

if(i==0)

{

sum=a;

}

else

{

sum=sum^a;

}

}

if(sum)

cout<<"Yes"<<endl;

else

cout<<"No"<<endl;

}

return 0;

}

1. SG 函数
2. 敌对搜索

定义一种新的黑白棋：

1. 棋盘大小为5\*5的格子；

2. 有些格子不能放棋子；

3. 同一个格子最多放一个棋子；

4. 先手执白棋，后手执黑棋；

5. 先手第一次可以把棋放在任意可以放的位置上；

6. 接下来两人轮流放棋子，这个棋子必须与上一个人放的棋子相邻

请问：两人都是最优策略，是先手赢，还是先手输？

#include<cstdio>

#include<iostream>

using namespace std;

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

int chess[10][10];

//敌对搜索的特点是每一层dfs的对象不同,是必胜态和必败态的交替

int dfs(int a,int b){//在(a,b)下棋,若当前为必胜态返回1,否则返回0

int flag=0;

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

int xx=a+dir[i][0];

int yy=b+dir[i][1];

if(xx>=1&&xx<=5&&yy>=1&&yy<=5&&chess[xx][yy]==0){

chess[xx][yy]=1;

if(!dfs(xx,yy)){/\*枚举周围4个相邻的格子,如果有一个进入必败态,当前就是必胜态;

如果四个各自都是必胜态,当前就是必败态 \*/

flag=1;

chess[xx][yy]=0;//break之前要回溯

break;

}

chess[xx][yy]=0;

}

}

return flag;

}

int main(){

int T;

scanf("%d",&T);

while(T--){

for(int i=1;i<=5;i++){

for(int j=1;j<=5;j++){

scanf("%1d",&chess[i][j]);//黑科技,可以每次读入一位整数;另外可以用scanf("%d\n",&T)吸收回车

}

}

int flag=0;

for(int i=1;i<=5;i++){

for(int j=1;j<=5;j++){

if(chess[i][j]==0){

chess[i][j]=1;

if(!dfs(i,j)){//先手枚举每一个可以开始下棋的位置开始下棋

flag=1;

}

chess[i][j]=0;//每一次要回溯起点

}

}

}

if(flag)

printf("win\n");

else

printf("lose\n");

}

}

1. 0-1 矩阵
2. 0-1 树
3. 字符串
4. KMP

/\*KMP算法是单模式匹配算法,即给定一个主串O和一个模式串f,长度分别为n和m，询问模式串f是否在主串O中出现过,

若出现过返回其位置。普通的BF朴素匹配算法，遍历主串的每一个位置，然后从该位置开始和

模式串进行匹配，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)。\*/

/\*KMP算法思想：KMP算法的核心是：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。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f的最大公共前缀之后的字符开始比较o和f即可

\*/

/\*预处理Next数组。其中Next[i]表示：Next[i]表示f[0:i]的前后缀最大公共长度(不包括f[0:i]本身),也表示模式串f的下标i失配时应该回溯到的比较位置

比如：

f: a b a a b b a b a a b

Next: 0 0 1 1 2 0 1 2 3 4 5

求Next[i]：Next[0] = 0(前后缀最大公共长度为0)，如果f[i]和位置Next[i-1]处的字符相同，则Next[i]等于Next[i - 1]加1，就这样一直寻找此大前缀，次次大前缀，直到字符串长度为0为止。注意不能等于0！因为Next[0] = 0，会死循环\*/

int Next[maxn]; //Next[i]表示：Next[i]表示f[0:i]的前后缀最大公共长度(不包括f[0:i]本身),也表示模式串f的下标i失配时应该回溯到的比较位置

int pos[maxn]; //存放匹配位置的起始下标

void getNext(string find){

int len = find.size();

Next[0] = 0;

for(int i = 1;i < len;i++){

int j = Next[i - 1]; //j在每次循环开始都初始化为Next[i-1]，表示需要和i处字符比较的位置

while(j > 0 && find[j] != find[i])

j = Next[j - 1]; //寻找最大前缀的最大前缀

if(find[j] == find[i])

j++;

Next[i] = j;

}

}

void KMPMatch(string original,string find) {

int cnt = 0;

int j = 0;

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

while (j > 0 && original[i] != find[j])

j = Next[j - 1];

if (original[i] == find[j])

j++;

if (j == find.size()) {

pos[cnt++] = i - j + 1; //pos数组记录匹配成功时子串在主串中的位置

j = Next[j - 1]; //匹配成功后模式串向后滑动，查找下一个匹配位置

}

}

}

具体见博客：http://www.cnblogs.com/SYCstudio/p/7194315.html

1. AC自动机

#include<cstdio>

#include<cstring>

#include<cstdlib>

#include<iostream>

#define MAX 26

const int MAXN = 10000000;

using namespace std;

struct Trie

{

int count;

Trie \*Next[MAX];

Trie \*fail; //失配指针

};

Trie \*q[MAXN]; //队列，采用bfs构造失配指针

char keyword[55];//模式串

char str[1000010];// 需要查找的主串

int head,tail;//队列 头尾指针

Trie \*root;

void insert(char \*word)

{

int index,len;

Trie \*p = root,\*newnode;

len = strlen(word);

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

{

index=word[i]-'a';

if(p->Next[index]==NULL)

{

newnode=(Trie \*)malloc(sizeof(Trie));

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

newnode->Next[j]=NULL;

newnode->count=0;

newnode->fail=NULL;

p->Next[index]=newnode;

}

p=p->Next[index];

}

p->count++;

}

void build\_ac\_automation()

{

head=0;

tail=1;

q[head]=root;

Trie \*temp,\*p;

while(head<tail) {

temp=q[head++];

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

{

if(temp->Next[i]){

if(temp==root)

temp->Next[i]->fail=root;

else {

p=temp->fail;

while(p){

if(p->Next[i]){

temp->Next[i]->fail=p->Next[i];

break;

}

p=p->fail;

}

if(!p)

temp->Next[i]->fail=root;

}

q[tail++]=temp->Next[i];

}

}

}

}

int query()

{

int i,cnt=0,index,len=strlen(str);

Trie \*p=root;

for(i=0; i < len ;i ++)

{

index=str[i]-'a';

while( !p->Next[index] && p != root)

p=p->fail;

p=p->Next[index];

if(!p)

p=root;

Trie \*temp=p;

while(temp != root ){

if(temp->count>=0){

cnt+=temp->count;

temp->count=-1;

}

else break;

temp=temp->fail;

}

}

return cnt;

}

int main()

{

int i,T,n,ans;

cin>>T;

while(T--)

{

root=(Trie\*)malloc(sizeof(Trie));

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

root->Next[j]=NULL;//记得初始化为空

root->fail=NULL;

root->count=0;

cin>>n;

getchar();

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

{

gets(keyword);//模式串建树

insert(keyword);

}

build\_ac\_automation();

gets(str);//输入主串查询匹配

ans=query();

cout<<ans<<endl;

}

return 0;

}

1. 后缀数组
2. 滚动哈希(Rabin-Karp算法)

/\*滚动哈希：O（n+m）时间内完成字符串匹配；

实现：选取两个合适的互素常数b和h（l<b<h），假设字符串C=c1c2c3...cm，定义哈希函数：

H(C)=（c1\*b^(m-1)+c2\*b^(m-2)+...+cm\*b^0）mod h其中b是基数，相当于把字符串看

作b进制数。这样，字符串S=s1s2s3...sn从位置k+1开始长度为m的字符串子串S[k+1...k+m]

的哈希值，就可以利用从位置k开始的字符串子串S[k...k+m-1]的哈希值，直接进行如下计算：

H(S[k+1...k+m])=（H(S[k...k+m-1]）\* b - sk\*b^m + s(k+m)） mod h

于是，只要不断这样计算开始位置右移一位后的字符串子串的哈希值，就可以在O（n）时间内得到

所有位置对应的哈希值，从而可以在O（n+m）时间内完成字符串匹配。在实现时，可以用64位无符

号整数计算哈希值，并取h等于2^64，通过自然溢出省去求模运算。

\*/

typedef unsigned long long ull;

const ull b=100000007;//哈希的基数；

//C是否在S中出现

bool contain(string C,string S)

{

int m=C.length(),n=S.length();

if(m>n) return false;

//计算b的m次方

ull t=1;

for(int i=0;i<m;i++) t\*=b;

//计算C和S长度为m的前缀对应的哈希值

ull Chash=0,Shash=0;

for(int i=0;i<m;i++) Chash=Chash\*b+C[i];

for(int i=0;i<m;i++) Shash=Shash\*b+S[i];

//对S不断右移一位，更新哈希值并判断

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

if(Chash==Shash) return true;//S从位置i开始长度为m的字符串子串等于C；

if(i+m<n) Shash=Shash\*b-S[i]\*t+S[i+m];

}

return false;

}

1. 其他
2. 字符串和数字互转

数字转字符串：

stringstream(方便但转换大量数据较慢)：

用法：

stringstream ss;

string s;

ss << i;

ss >> s;

sprintf函数：

用法：

int a;

string str;

char buff[10];

sprintf(buff,“%d”,a);

str = buff;

字符串转数字

stringstream(方便但转换大量数据较慢)：

用法：

stringstream ss;

int num;

string s;

ss << s;

ss >> num;

sscanf函数：

char str[] = “15.455”;

int i;

float fp;

sscanf(str,“%d”,&i); // i = 15

sscanf(str,”%f”,&fp); // fp = 15.455000

1. 字符串和字符数组互转

字符数组转字符串

直接把字符数组赋给字符串即可

字符串转字符数组

string s;

char str[] = s.c\_str();

1. cin读入一行

string str;

getline(cin,str);

1. 二维数组开不下可以考虑vector数组，string数组或者map数组

1. cout控制保留小数点位数

头文件iomanip

cout <<fixed<< setprecision(2) << a <<endl;

1. 去重

n = unique(a,a+n) - a;

实质是把重复元素放到数组末尾，没有真正删除，返回指向最后一个非重复元素的迭代器

3.整除的性质

（1）任何一个整数,都能被1整除.

（2）若一个整数的末位是0、2、4、6或8,则这个数能被2整除.

（3）若一个整数的数字和能被3整除,则这个整数能被3整除.

(4) 若一个整数的末尾两位数能被4整除,则这个数能被4整除.

（5）若一个整数的末位是0或5,则这个数能被5整除.

（6）若一个整数能被2和3整除,则这个数能被6整除.

（7）若一个整数的个位数字截去,再从余下的数中,减去个位数的2倍,如果差是7的倍数,则原数能被7整除.如果差太大或心算不易看出是否7的倍数,就需要继续上述「截尾、倍大、相减、验差」的过程,直到能清楚判断为止.例如,判断133是否7的倍数的过程如下：13－3×2＝7,所以133是7的倍数；又例如判断6139是否7的倍数的过程如下：613－9×2＝595 , 59－5×2＝49,所以6139是7的倍数,余类推.

（8）若一个整数的未尾三位数能被8整除,则这个数能被8整除.

（9）若一个整数的数字和能被9整除,则这个整数能被9整除.

（10）若一个整数的末位是0,则这个数能被10整除.

（11）若一个整数的奇位数字之和与偶位数字之和的差能被11整除,则这个数能被11整除.11的倍数检验法也可用上述检查7的「割尾法」处理!过程唯一不同的是：倍数不是2而是1!

（12）若一个整数能被3和4整除,则这个数能被12整除.

（13）若一个整数的个位数字截去,再从余下的数中,加上个位数的4倍,如果差是13的倍数,则原数能被13整除.如果差太大或心算不易看出是否13的倍数,就需要继续上述「截尾、倍大、相加、验差」的过程,直到能清楚判断为止.

（14）若一个整数的个位数字截去,再从余下的数中,减去个位数的5倍,如果差是17的倍数,则原数能被17整除.如果差太大或心算不易看出是否17的倍数,就需要继续上述「截尾、倍大、相减、验差」的过程,直到能清楚判断为止.

（15）若一个整数的个位数字截去,再从余下的数中,加上个位数的2倍,如果差是19的倍数,则原数能被19整除.如果差太大或心算不易看出是否19的倍数,就需要继续上述「截尾、倍大、相加、验差」的过程,直到能清楚判断为止.

（16）若一个整数的末三位与3倍的前面的隔出数的差能被17整除,则这个数能被17整除.

（17）若一个整数的末三位与7倍的前面的隔出数的差能被19整除,则这个数能被19整除.

（18）若一个整数的末四位与前面5倍的隔出数的差能被23(或29)整除,则这个数能被23整除

4.各种距离区别

1. 欧式距离（欧几里得距离或欧几里得度量） Euclidean Metric

即两个点之间的距离

IMG_256

1. 曼哈顿距离 Manhattan distance

两个点在标准坐标系上的绝对轴距总和

IMG_256

1. 切比雪夫距离 Chebyshev distance 或 Supremum distance

各坐标数值差的最大值

IMG_256

1. 明式距离（明可夫斯基距离） Minkowski distance

纬度等于1时候，公式等价于曼哈顿距离

等于2时候，公式等价于欧氏距离

当大于2 到无穷大时候，公式等价于切比雪夫距离

5.位运算的应用与技巧

1.枚举n个元素的所有子集

for(int i = 0; i < 1<<n; i++)

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

if(i & (1<<j))

printf("%d ", j);

2.消去二进制中最右侧的那个1

x & (x - 1)

应用一:用 O（1）时间检测整数n是否是2的幂次

(两个条件：n > 0， n的二进制表示中只有一个1)

return n > 0 && (n & (n - 1)) == 0;

应用二：计算在一个32位的整数的二进制表示中有多少个1

int t = 0;

while(n != 0){n = n & (n - 1); t++}

return t;

3.交换两个整数的值

int a, b;

a = a ^ b;

b = a ^ b;

a = a ^ b;

4.与2的x次方的运算

int a;

a <<= x;//乘2^x

a >= x;//除2^x

a & (1 << x) - 1 //取a%(2^x)的余数

a & 1//判断奇偶性，为0则a为偶数,否则为奇数

a & (a-1)//判断a是否为2的幂或者0，结果为0代表是，否则代表不是

5.其他常用技巧

int a, b;

a = ~a + 1//取相反数

a = (a ^ (a >> 31)) - (a >> 31) //取绝对值

(a & b) + ((a ^ b) >> 1) //取平均值

a ^ b//判断a、b符号是否相同，如果结果>0则相同，否则不同

常见的二进制位的变换操作:

去掉最后一位 x >> 1

在最后加一个0 x << 1

在最后加一个1 x << 1 + 1

把最后一位变成1 x | 1

把最后一位变成0 x | 1 - 1

最后一位取反 x ^ 1

把右数第k位变成1 x | (1 << (k-1))

把右数第k位变成0 x & ~(1 << (k-1))

右数第k位取反 x ^ (1 << (k-1))

取末k位 x & ((1 << k)-1)

取右数第k位 x >> (k-1) & 1

把末k位变成1 x | ((1 << k)-1)

末k位取反 x ^ ((1<< k)-1)

把右边连续的1变成0 x &(x+1)

把右起第一个0变成1 x | (x+1)

把右边连续的0变成1 x | (x-1)

取右边连续的1 (x ^ (x+1)) >> 1

去掉右起第一个1的左边 x & (x ^ (x-1))

1. 二分查找及其变种

**注意点：**

1. 二分查找的循环条件都为left <= right
2. 查找第一个返回left，查找最后一个返回right

3、left永远向mid + 1靠，right永远向mid - 1靠，查找第一个让right向左靠，查找最后一个让left向右靠

4、需要验证满足条件的数是否存在

/\*\*

\* 二分查找，找到该值在数组中的下标，否则为-1

\*/

static int binarySerach(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] == key) {

return mid;

}

else if (array[mid] < key) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

return -1;

}

变种1

// 查找第一个相等的元素

static int findFirstEqual(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] >= key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

if (left < array.length && array[left] == key) {

return left;

}

return -1;

}

变种2

// 查找最后一个相等的元素

static int findLastEqual(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] <= key) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

if (right >= 0 && array[right] == key) {

return right;

}

return -1;

}

变种3

// 查找最后一个等于或者小于key的元素

static int findLastEqualSmaller(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] > key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return right;

}

变种4

// 查找最后一个小于key的元素

static int findLastSmaller(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] >= key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return right;

}

变种5

// 查找第一个等于或者大于key的元素

static int findFirstEqualLarger(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] >= key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return left;

}

变种6

// 查找第一个大于key的元素

static int findFirstLarger(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] > key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return left;

}

Ubuntu Vim 基本配置

配置文件 /etc/vim/vimrc

set ai                          " 自动缩进，新行与前面的行保持—致的自动空格

set aw                        " 自动写，转入shell或使用：n编辑其他文件时，当前的缓冲区被写入

set flash                     " 在出错处闪烁但不呜叫(缺省)

set ic                          " 在查询及模式匹配时忽赂大小写

set nu

set number                " 屏幕左边显示行号

set showmatch          " 显示括号配对，当键入“]”“)”时，高亮度显示匹配的括号

set showmode           " 处于文本输入方式时加亮按钮条中的模式指示器

set showcmd             " 在状态栏显示目前所执行的指令，未完成的指令片段亦会显示出来

set warn/nowarn        " 对文本进行了新的修改后，离开shell时系统给出显示(缺省)

set ws/nows               " 在搜索时如到达文件尾则绕回文件头继续搜索

set wrap/nowrap        " 长行显示自动折行

colorscheme evening " 设定背景为夜间模式

filetype plugin on        " 自动识别文件类型，自动匹配对应的, “文件类型Plugin.vim”文件，使用缩进定义文件

set autoindent            " 设置自动缩进：即每行的缩进值与上一行相等；使用 noautoindent 取消设置

set cindent                 " 以C/C++的模式缩进

set noignorecase       " 默认区分大小写

set ruler                     " 打开状态栏标尺

set scrolloff=5            " 设定光标离窗口上下边界 5 行时窗口自动滚动

set shiftwidth=4          " 设定 << 和 >> 命令移动时的宽度为 4

set softtabstop=4       " 使得按退格键时可以一次删掉 4 个空格,不足 4 个时删掉所有剩下的空格）

set tabstop=4             " 设定 tab 长度为 4

set wrap                     " 自动换行显示

syntax enable

syntax on                    " 自动语法高亮

inoremap ( <c-r>=AutoPair('(', ')')<CR>

inoremap ) <c-r>=ClosePair(')')<CR>

inoremap { <c-r>=AutoPair('{', '}')<CR>

inoremap } <c-r>=ClosePair('}')<CR>

inoremap [ <c-r>=AutoPair('[', ']')<CR>

inoremap ] <c-r>=ClosePair(']')<CR>

inoremap " <c-r>=SamePair('"')<CR>

inoremap ' <c-r>=SamePair("'")<CR>

inoremap ` <c-r>=SamePair('`')<CR>

热身赛前测试

测试具体内容  
1、JAVA的大数，主类名，以及类似isPrime等常用方法，打印JAVA.math。  
2、提问是否有MLE，试验数组能开多大  
3、试验是否有PE  
4、RE的时候是返回WA还是RE  
5、memset是否要加memory.h，还是string.h  
6、stdio.h和cmath和cstring是否需要  
7、测算CPU运行速度。(因为主机和选手的机器一般是一样的) 10^8  
8、试验打印功能和与裁判交流的功能  
9、提交做好文件输出的C++和JAVA任务  
10、STL-vector,string,map,set提前尝试一下能不能用  
11、试一下机器是否会开机还原  
12、C++和JAVA都试一下，有没有自动补全  
13、代码长度是否有限制   
14、试一下NetBeans里面如果下标为负是不是就在编译器里就可以返回SF   
15、问一下和试一下栈空间大小