===================================================

XLB的ACM模板

(2019 5.17) 注：部分简单算法未放入模板

===================================================

【头文件】

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <string.h>

#include <string>

#include <limit.h>

#include <algorithm>

#include <iostream>

#include <ctype.h>

#include <set>

#include <map>

#include <vector>

#include <stack>

#include <queue>

#include <random>

#include <chrono>

#include <complex>

#define left (now<<1)

#define right ((now<<1)+1)

#define mid ((l+r)>>1)

#define midmid ((r + mid) / 2.0)

#define fst first

#define snd second

#define LONG\_LONG\_MAX 9223372036854775807ll

#define LONG\_LONG\_MIN -9223372036854775808ll

**using** **namespace** std;

**typedef** **long** **long** ll;

【读入相关】

%a,%A 读入一个浮点值(仅C99有效)

%c 读入一个字符

%d 读入十进制整数

%i 读入十进制，八进制，十六进制整数

%o 读入八进制整数

%x,%X 读入十六进制整数

%s 读入一个字符串，遇空格、制表符或换行符结束。

%f,%F,%e,%E,%g,%G 用来输入实数，可以用小数形式或指数形式输入。

%p 读入一个指针

%u 读入一个无符号十进制整数

%n 至此已读入值的等价字符数

%[] 扫描字符集合

%% 读%符号

【输入挂】

**int** read(){

**int** v = 0,f = 1;

**char** c = getchar();

**while**(c < 48 || 57 < c) {**if**(c == '-') f = -1; c = getchar();}

**while**(48 <= c && c <= 57) v = (v<<3) + v + v + c - 48,c = getchar();

**return** v \* f;

}

【分块+莫队】

BLOCK = (int)(sqrt(n))

be[i]表示第i个数字在第几块

莫队按第一维所在块排序，第二维大小为第二关键字，如果有三继续以3

【关闭同步流】

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

【随机函数】

使用mt19937而不是rand()

#include <chrono>

#include <random>

mt19937 rng(chrono::steady\_clock::now().time\_since\_epoch().count());

ll ans = rng();

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

范围为0 - 4294967295（2的32次方 - 1）

【随机生成整数】

**int** randInt(**int** l,**int** r){    //生成l到r的整数,l <= r

**return** (rng() % (r - l + 1)) + l;

}

【随机生成实数】

**double** randDouble(**double** l,**double** r,**int** d){  //生成l到r的实数 ,d为小数位数

    l = l \* pow(10,d); r = r \* pow(10,d);

**return** randInt(l,r) / (1.0 \* pow(10,d));

}

【对序列随机排列】

random\_shuffle(p.begin(),p.end());

【随机数种子】

srand(time(NULL));

【声明复数】

complex<double> z1 {3,4}; cin,cout输入输出

.image() //取虚部或修改虚部

.real() //取实部或修改实部

注意精度问题

目录

[一、 图论 7](#_Toc16429650)

[**1.1** **最短路** 7](#_Toc16429651)

[1.2.1 【Floyd算法】 7](#_Toc16429652)

[1.2.2 【dijkstra算法 + 堆优化】 8](#_Toc16429653)

[1.2.3 【spfa算法】 8](#_Toc16429654)

[1.2.4 【环类问题】 9](#_Toc16429655)

[1.2.5 【差分约束】 10](#_Toc16429656)

[1.2.6 【各种变形问题】 11](#_Toc16429657)

[**1.2** **网络流** 12](#_Toc16429658)

[1.2.1 【最大流（dinic算法）】 12](#_Toc16429659)

[1.2.2 【最小费用最大流（模板 洛谷P3381）】 13](#_Toc16429660)

[1.2.3 【各种变形问题】 14](#_Toc16429661)

[**1.3** **二分图以及匹配类问题** 15](#_Toc16429662)

[1.3.1 【二分图最大匹配（匈牙利算法）】 15](#_Toc16429663)

[1.3.2 【二分图最大匹配（HK算法）】 16](#_Toc16429664)

[1.3.3 【二分图最大权匹配（KM算法）】 17](#_Toc16429665)

[1.3.4 【一般图最大匹配（带花树算法）（模板：uoj 79）】 18](#_Toc16429666)

[1.3.5 【稳定婚姻匹配】 20](#_Toc16429667)

[1.3.6 【各种变形问题】 22](#_Toc16429668)

[**1.4** **欧拉路和欧拉回路（模板 uoj117）** 23](#_Toc16429669)

[**1.5** **最大团（极大团）和最大独立集** 24](#_Toc16429670)

[1.5.1 【最大团】（模板HDU 1530 必须运行低于3000MS以内可用） 24](#_Toc16429671)

[1.5.2 【极大团（模板POJ2989）】 25](#_Toc16429672)

[**1.6** **图上染色** 26](#_Toc16429673)

[1.6.1 【无向图色数（模板HDU1373）】 26](#_Toc16429674)

[二、 数据结构 27](#_Toc16429675)

[**2.1** **树上性质和各种问题** 27](#_Toc16429676)

[2.1.1 【树的直径】 27](#_Toc16429677)

[2.1.2 【树的重心】 27](#_Toc16429678)

[2.1.3 【树上差分】（模板 NOIP2015 运输计划） 28](#_Toc16429679)

[2.1.4 【离线Tarjan算法求LCA】 28](#_Toc16429680)

[2.1.5 【在线RMQ加时间戳算法求LCA】（模板 洛谷P3379） 29](#_Toc16429681)

[2.1.6 【图内生成树个数】 30](#_Toc16429682)

[2.1.7 【最小生成树的变种】 31](#_Toc16429683)

[2.1.8 【树上启发式合并】 31](#_Toc16429684)

[**2.2** **栈(队列)** 31](#_Toc16429685)

[【单调栈和单调队列】 31](#_Toc16429686)

[【中缀表达式转后缀表达式（逆波兰式）】 32](#_Toc16429687)

[【栈对后缀表达式求值】 33](#_Toc16429688)

[**2.3** **并查集（查询优化）** 33](#_Toc16429689)

[**2.4** **线段树（模板HDU1166）** 34](#_Toc16429690)

[【二维线段树（四叉树） （模板HDU1892）】 35](#_Toc16429691)

[【可持久化线段树（主席树）】（洛谷P3834 HDU 2665） 36](#_Toc16429692)

[**2.5** **树状数组（模板HDU1166）** 37](#_Toc16429693)

[【单点查询区间修改】 38](#_Toc16429694)

[【求逆序对数】 38](#_Toc16429695)

[【二维树状数组 （模板HDU1892）】 38](#_Toc16429696)

[**2.6** **字典树** 38](#_Toc16429697)

[【各种变形问题】 39](#_Toc16429698)

[**2.7** **KD树** 39](#_Toc16429699)

[**2.8** **树链剖分（模板 洛谷P3384）** 42](#_Toc16429700)

[**2.9** **树分治（模板poj 1741）** 43](#_Toc16429701)

[**2.10** **线性基（模板 洛谷P 3812）** 45](#_Toc16429702)

[三、 非几何数学 46](#_Toc16429703)

[**3.1** **各种公式** 46](#_Toc16429704)

[**3.2** **基础结构** 48](#_Toc16429705)

[**3.3** **基础函数** 49](#_Toc16429706)

[3.3.1 【快速幂和快速乘法】 49](#_Toc16429707)

[3.3.2 【排列数和组合数】 49](#_Toc16429708)

[3.3.3 【最小公约数和公倍数】 49](#_Toc16429709)

[3.3.4 【矩阵乘法和矩阵快速幂】 50](#_Toc16429710)

[3.3.5 【求1-n有多少数字和k不互素】 51](#_Toc16429711)

[3.3.6 【基姆拉尔森公式】 51](#_Toc16429712)

[3.3.7 【求单个欧拉函数】 52](#_Toc16429713)

[3.3.8 【素数，欧拉函数，莫比乌斯函数】 52](#_Toc16429714)

[**3.4** **较复杂算法** 53](#_Toc16429715)

[3.4.1 【Berlekamp\_Massey】 53](#_Toc16429716)

[3.4.2 【扩展欧几里得算法】 55](#_Toc16429717)

[3.4.3 【FFT快速傅里叶变换 （模板 洛谷P3803）】 55](#_Toc16429718)

[3.4.4 【NTT快速数论变换 】 57](#_Toc16429719)

[3.4.5 【FWT快速沃尔什变换（模板 洛谷P4717）】 57](#_Toc16429720)

[**3.5** **博弈问题** 58](#_Toc16429721)

[3.5.1 【巴什博弈(Bash Game)】 58](#_Toc16429722)

[3.5.2 【尼姆博弈(Nimm Game)】 58](#_Toc16429723)

[3.5.3 【威佐夫博奕（Wythoff Game）】 59](#_Toc16429724)

[3.5.4 【Sprague-Grundy定理（SG定理）】 59](#_Toc16429725)

[**3.6** **组合数学** 60](#_Toc16429726)

[3.6.1 【经典数列或者公式】 60](#_Toc16429727)

[3.6.2 【生成函数】 62](#_Toc16429728)

[**3.7** **其他问题** 62](#_Toc16429729)

[3.7.1 【n的阶乘里有几个因子2】 63](#_Toc16429730)

[3.7.2 【因数的个数是偶数还是奇数？】 63](#_Toc16429731)

[3.7.3 【输油管道问题和糖果传递问题】 63](#_Toc16429732)

[3.7.4 【平面上的欧拉公式】 63](#_Toc16429733)

[四、 字符串 63](#_Toc16429734)

[**4.1** **基础函数** 63](#_Toc16429735)

[**4.2** **字符串匹配** 65](#_Toc16429736)

[4.2.1 【kmp算法】 65](#_Toc16429737)

[4.2.2 【扩展KMP算法】 66](#_Toc16429738)

[4.2.3 【Shift-And / Shift-Or算法】 67](#_Toc16429739)

[**4.3** **回文串** 67](#_Toc16429740)

[4.3.1 【manacher算法】 67](#_Toc16429741)

[五、 计算几何 68](#_Toc16429742)

[**5.1** **常用参数和公式** 68](#_Toc16429743)

[**5.2** **基本结构体** 69](#_Toc16429744)

[**5.3** **结构体的基本运算** 70](#_Toc16429745)

[**5.4** **简单功能函数** 72](#_Toc16429746)

[**5.5** **复杂功能函数** 74](#_Toc16429747)

[5.5.1 【平面最远欧几里得距离点对】 74](#_Toc16429748)

[5.5.2 【平面最远曼哈顿距离点对】（模板HDU6435） 74](#_Toc16429749)

[5.5.3 【平面最近点对】（模板HDU1007） 74](#_Toc16429750)

[六、 其他 75](#_Toc16429751)

[**6.1** **搜索类** 75](#_Toc16429752)

[6.1.1 【A\*搜索】 75](#_Toc16429753)

[6.1.2 【爬山算法】 75](#_Toc16429754)

[6.1.3 【模拟退火算法】（模板HDU1109） 75](#_Toc16429755)

[6.1.4 【二分查找】 76](#_Toc16429756)

[6.1.5 【三分查找】（模板题HDU4355） 76](#_Toc16429757)

[七、 题库 76](#_Toc16429758)

[**7.1** **图类问题** 76](#_Toc16429759)

[**7.2** **树类问题** 76](#_Toc16429760)

[【思维】 77](#_Toc16429761)

[【树上DP】 77](#_Toc16429762)

[**7.3** **二维线段（区间）类问题** 77](#_Toc16429763)

[【最大不重叠子区间】 77](#_Toc16429764)

[【思维】 78](#_Toc16429765)

[**7.4** **动态规划类问题** 78](#_Toc16429766)

[**7.5** **博弈类问题** 78](#_Toc16429767)

[**7.6** **数学类问题** 78](#_Toc16429768)

[7.6.1 【计数：求n行三角形中等边三角形个数（斜着的也算）】 78](#_Toc16429769)

[7.6.2 【计数：求n行m列网格中正方形个数（斜着的算）】 78](#_Toc16429770)

[7.6.3 【计数：求n行m列网格中正方形个数（斜着的不算）】 79](#_Toc16429771)

[7.6.4 【计数：求n行m列网格中矩形个数（斜着的不算）】 79](#_Toc16429772)

[**7.7** **字符串类问题** 79](#_Toc16429773)

[八、 STL库 79](#_Toc16429774)

[**8.1** **vector（向量，动态数组）** 79](#_Toc16429775)

[**8.2** **map（映射容器）** 80](#_Toc16429776)

[**8.3** **set（集合容器）** 80](#_Toc16429777)

[**8.4** **multiset（可重集合）** 81](#_Toc16429778)

[**8.5** **queue（队列）** 82](#_Toc16429779)

[**8.6** **priority\_queue（优先队列）** 82](#_Toc16429780)

[**8.7** **string（字符串）** 82](#_Toc16429781)

[**8.8** **stack（栈）** 82](#_Toc16429782)

[**8.9** **bitset（状态压缩）** 83](#_Toc16429783)

[**8.10** **algorithm库** 83](#_Toc16429784)

[九、 其他 91](#_Toc16429785)

[9.1 【java大整数和大实数】 91](#_Toc16429786)

1. **图论**
2. **最短路**
3. 【Floyd算法】

O（）

思想：暴力跑一边动态规划即可。

有向图写法（无向图可以优化）

前置数据：

int g[MAXN][MAXN]; //邻接矩阵

**void** floyd(){

**for**(**int** k = 1; k <= n; ++k){

**for**(**int** i = 1; i <= n; ++i){

**for**(**int** j = 1; j <= n; ++j){

**if**(g[i][k] + g[k][j] < g[i][j]){

                    g[i][j] = g[i][k] + g[k][j];

                }

            }

        }

    }

}

1. 【dijkstra算法 + 堆优化】

O（）

思路：以当前最短距离最短，且vis为false的点贪心入堆，每次更改最短路，就新增一个qq项入队，所以记得对于已经得到最短路的点,即vis为true的点要continue掉。

注意：不可处理负边权，不可处理负环。

前置数据：链式前向星

int dis[MAXN]; //保存距离

bool vis[MAXN]; //标记访问

**struct** qq{

**int** u,w;

};

priority\_queue<qq,vector<qq>,cmp> q;

**void** dijkstra(**int** st){

**for**(**int** i = 1; i <= n; ++i){ dis[i] = 2147483647;} dis[st] = 0;

    memset(vis,**false**,**sizeof**(vis)); qq x; x.u = st; x.w = 0;

    q.push(x);

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

**int** now = q.top().u; q.pop();

**if**(vis[now]){ **continue**;}

vis[now] = **true**;

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(dis[now] + g[i].w < dis[g[i].to]){

                dis[g[i].to] = dis[now] + g[i].w;

                qq x; x.u = g[i].to; x.w = dis[g[i].to];

                q.push(x);

            }

        }

    }

}

1. 【spfa算法】

O（）

思想：bellman-ford算法的队列优化，效率比dijkstra要低不少，如果不是非要用spfa，尽量用dijkstra。就是弄个队列，然后扩展，如果有更短的路可以松弛，就松弛完，如果被松弛的点不在队列里，入队，直到队列为空。

注意：可以求负环，加个num数组计数，超过n次入队就return true，可以处理负边权。

前置数据：链式前向星

int dis[MAXN];

bool have[MAXN];

queue<int> q;

**bool** spfa(**int** st){    //不处理负环时，无返回值

fill(dis,dis + 1 + n,INT\_MAX);

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

//memset(num,0,sizeof(num));    //判负环

**while**(!q.empty()){ q.pop();} q.push(st); dis[st] = 0; have[st] = **true**;

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

**int** now = q.front(); q.pop();

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(dis[now] + g[i].w < dis[g[i].to]){

                dis[g[i].to] = dis[now] + g[i].w;

**if**(!have[g[i].to]){

                    //++num[g[i].to]; if(num[g[i].to] > n){ return true;} //判负环

                    have[g[i].to] = **true**; q.push(g[i].to);

                }

            }

        }

        have[now] = **false**;

}

// return false; //判负环

}

1. 【环类问题】

【判是否存在环】

有向图判是否存在环，直接dfs一下就完事了。

前置数据：vis[MAXN];

**bool** pdcir(**int** now){    //有向图判环

**if**(vis[now]==1){**return** **true**;}

**if**(vis[now]==2){**return** **false**;}

    vis[now] = 1;

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(pdcir(g[i].to)){

**return** **true**;

        }

    }

    vis[now] = 2;

**return** **false**;

}

【求最小环】

判最小环需要用到Floyd，用dijkstra的话，复杂度会过高的。

前置数据：

int g[MAXN][MAXN]; //邻接矩阵

**int** minc(){    //求无向图中最小环

**int** i,j,k,re;

    re=maxn;

**for**(k=1;k<=n;k++){ //枚举所有中间点

**for**(i=1;i<k;i++) //不能超过k

**for**(j=1;j<i;j++) //无向图中起点和终点一样

                re=min(re,dis[i][j]+g[j][k]+g[k][i]); //找环

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

**for**(j=1;j<i;j++)

              dis[i][j]=dis[j][i]=min(dis[i][j],dis[i][k]+dis[k][j]);

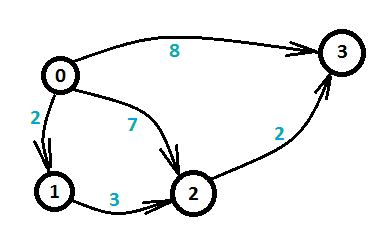
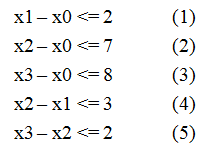
     } //正常的floyd算法。

**return** re;

}

1. 【差分约束】

求不等式组的最值可以转化为求最短或者最长路。（基本都用SPFA）



求（x3 – x0）的最大值，图中应为7

特殊情况和注意点

如果2点间不连通，则说明是可以是任意近和任意远。

如果存在负环，则说明无解。

具体输出按题目要求灵活变化，若没有以上2个情况，可以使用dijkstra降低复杂度，一般都是使用spfa。

不等式标准化

      如果给出的不等式有"<="也有">="，又该如何解决呢？很明显，首先需要关注最后的问题是什么，如果需要求的是两个变量差的最大值，那么需要将所有不等式转变成"<="的形式，建图后求最短路；相反，如果需要求的是两个变量差的最小值，那么需要将所有不等式转化成">="，建图后求最长路。

      如果有形如：A - B = c 这样的等式呢？我们可以将它转化成以下两个不等式：

**A - B >= c      (1)**

**A - B <= c      (2)**

       再通过上面的方法将其中一种不等号反向，建图即可。

       最后，如果这些变量都是整数域上的，那么遇到A - B < c这样的不带等号的不等式，我们需要将它转化成"<="或者">="的形式，即 A - B <= c - 1。

参考网页：<http://www.cppblog.com/menjitianya/archive/2015/11/19/212292.html>

1. 【各种变形问题】

【最短路条数计数】PAT上做到的题，呃，就是DP一下就完事，但是由于spfa能一直松弛，所以spfa求不了最短路究竟有几条，必须使用dijkstra，新增个dp数组，每次从队列取出一个新点时，看看周围有几个点的最短路是已经找到了的，对他们的dp求个和就好了。

【最短路路径输出】如果只要一条就很容易，spfa和dijkstra都可以，加个pre数组记录下前缀，最后递归一下即可。

【求最长（短）边最小（大）通路（poj2253，poj1797）】考虑dijkstra，dis改为maxn，起点为无限大，松弛改为max(g[now][i].w,dis[now]) < dis[g[now][i].to]这样的类似DP条件即可。

【A到B后还得回来，或者求其他点到起点的最短路（poj3268，poj1511）】前者就来回求一遍呗，后者的话，如果是有向图的话要搞个反图，然后起点单源最短路

【汇率交换看能不能赚到钱（poj1860，poj2240）】建图，然后跑spfa，如果存在负环，或者自己松弛到自己了（自己到自己的最短路变负）就能赚到钱

【求负环（poj3259）】spfa板子题。

【传递闭包（poj3660）】A能打败B，B能打败C，因此A就能打败C，求有多少人排名被确定。如果能打败就连有向边u到v为1，跑一边Floyd，然后暴力判断任意两头牛直接是否能从A走到B，或者B走到A（就是能确定胜负关系），如果可以就++ans，最后输出。

【思路建图】poj2502，poj1062，poj1847，HDU4725，HDU4370，BZOJ2118

【求所有不相交的最短路方案数（HDU3416）】有点难，先正反求最短路，所以要存个反图，然后枚举每条边，看看是否起点到这个边的左边，终点到这个边的右边，再加上这个边，刚好就等于起点到终点的最短路，如果是，就表示这个点是在起点到终点的最短路上，就记true，然后把这些边的容量设为1，直接来一个dinic求最大流，此时的最大流就是ans。

【可以把某几条路费用变0（BZOJ2763）】分层建k张图，类似DP跑最短路。

【保留K条边使得尽可能多的点的单源最短路长度不变（codeforces1076D）】考虑dijkstra生成树，从起点向外尽可能保留即可。

1. **网络流**
2. 【最大流（dinic算法）】

O（）

Dinic（实质是首先利用BFS求一遍地图的最短路（无边权）然后标记分层。再利用DFS，每次沿着标记1-2-3-4....找到最短的增广路来进行多路增广（增广后重新标记）。总复杂度上界为O(V^2E).但是在求二分图最大匹配的时候复杂度跟HK相同

int head[MAXN-node],len,dis[MAXN-node],st,ed,last[MAXN-node];

//last为辅助数组,其他含义为变量表面意思,len必须初始化为-1

star g[2\*MAXN-egde]; //链式前向星

**void** dinicBfs(**int** st){

    memset(dis,-1,**sizeof**(dis)); dis[st] = 0; q.push(st);

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

**int** now = q.front(); q.pop();

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(dis[g[i].to] == -1 && g[i].v > 0){

                dis[g[i].to] = dis[now] + 1;

                q.push(g[i].to);

            }

        }

    }

}

ll dinic(**int** now,ll mlow){

    ll re = 0;

**if**(now == ed || mlow == 0){

**return** mlow;

    }

**for**(**int** i = last[now]; i != -1; i = g[i].next){

**if**(dis[now] + 1 == dis[g[i].to] && g[i].v > 0){

            ll flow = dinic(g[i].to,min(mlow,g[i].v));

            mlow -= flow; g[i].v -= flow;

            re += flow; g[i ^ 1].v += flow;

**if**(mlow == 0){

**break**;

            }

        }

        last[now] = g[i].next;

    }

**return** re;

}

ll maxFlow(**int** st,**int** ed){

    ll re = 0; dinicBfs(st);

**while**(dis[ed] != -1){

        memcpy(last,head,**sizeof**(head));

        re += dinic(st,INF);

        dinicBfs(st);

    }

**return** re;

}

//在无向图中，两边都加上相同容量就好了

1. 【最小费用最大流（模板 洛谷P3381）】

在最大流的基础上，边上每单位流量有费用存在，需要总费用最少的最大流。

【spfa流】

在稠密图中效率可能有点慢,可能会被卡,被卡就用ZKW费用流

做法仍然是类似dinic找增广路，只不过找增广路不能用BFS了，而是用spfa找最短路（因为存在负数，不能用dijkstra）找到最短路以后，沿着最短路增广即可。

注意建图的时候，反边容量为0跟最大流一样，费用取相反数

spfa找增广路

前置数据：链式前向星存图

ll dis[MAXN]; //spfa求最短路

int pre[MAXN]; //标记最短路上那条边走向点i，倒着增广

queue<int> q; //spfa辅助队列

bool have[MAXN]; //spfa辅助数组

**void** dinic\_spfa(**int** st){

**for**(**int** i = 1; i <= n; ++i){

        dis[i] = LONG\_LONG\_MAX; have[i] = **false**; pre[i] = -1;

    }

    dis[st] = 0; have[st] = **true**; q.push(st);

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

**int** now = q.front(); q.pop();

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].w > 0 && dis[now] + g[i].f < dis[g[i].to]){

                dis[g[i].to] = g[i].f + dis[now];

                pre[g[i].to] = i;

**if**(!have[g[i].to]){

                    have[g[i].to] = **true**; q.push(g[i].to);

                }

            }

        }

        have[now] = **false**;

    }

}

pair<ll,ll> maxCostFlow(**int** st,**int** ed){ //返回值为<最大流,最小费用>

    ll re1 = 0,re2 = 0; dinic\_spfa(st);

**while**(dis[ed] != LONG\_LONG\_MAX){

**int** now = pre[ed]; ll minw = LONG\_LONG\_MAX;

**while**(now != -1){

            minw = min(minw,g[now].w);

            now = pre[g[now ^ 1].to];

        }

        now = pre[ed]; re1 += minw; re2 = re2 + dis[ed] \* minw;

**while**(now != -1){

            g[now].w -= minw; g[now ^ 1].w += minw;

            now = pre[g[now ^ 1].to];

        }

        dinic\_spfa(st);

    }

**return** make\_pair(re1,re2);

}

【ZKW流】

1. 【各种变形问题】

【二分图内多重匹配（POJ2289）】建个超级源点和超级汇点，源点容量都是1，右边匹配点到汇点的容量按你要能匹配的个数来确定就好。

【最小割】删掉图的一些边，边有权值，删掉总权值最小的边使得图不连通。可以证明答案等于最大流，所有满流的边就是要删除的边。

【最大权闭合子图】在有向图内，顶点有权值，选择一个子图，使得子图内所有点的后继都在子图内且权值最大，解法是源点接正权点容量为权，负权点接汇点，容量为权的相反数，然后点与点容量直接无限大，求(正权值之和-最小割)就是答案。

割掉s与i的边，表示不选择i点作为子图的点；   
割掉i与t的边，表示选择i点为子图的点。如果s与i有边，表示i存在子图中；   
如果i与t有边，表示i不存在于子图中。

【最大导出子图】（最大权闭合子图变形）选择一些边构成子图，边正权，顶点负权，使得子图权和最大。把边也看作点，边点和它的两端的点连接一下，因为边选了，点必须要选，因此有闭合子图的前后继关系，即边点的边指向2个端点，这样子按最大权闭合子图来建图就好了。

【二分图顶点最大权独立集】转为求二分图最小权匹配，拿KM，或者dinic做都可以，推荐dinic，源点连左边，流量为左边的点权，右边连汇点，左右连，流量无限大。然后跑出最大流，总点权减最大流即可。

【思路建图】poj3436，poj3281

1. **二分图以及匹配类问题**

一般二分图问题都可以变成网络流去求解，但是效率肯定差很多，还有一些其他的匹配问题也在这里给出。

二分图最大匹配时，二分图一般就有向图就好了，只要让左边点指向右边点就行了，因为rnum数组会帮忙把dfs从右边移回左边找增广路的。建有向图的话，就可以让2边的点的编号都从1开始了。

注意：如果二分图是木有分好的，但是题目告诉你一定是个二分图，或者已经染色过了，是可以跑匈牙利的。那么不需要强制把点分2个集合，直接开个match数组，然后建个**无向图**，就这么直接在普通图上跑匈牙利就行了，但是HK要先手动染个色，就算用dinic，也要先分好图染下色。

注意：树可以按层分二分图，棋盘可以按行列，或者是黑白划分二分图。

前置数据：链式前向星

bool vis[]（标记访问）

int r-num[]（左顶点匹配点）,lnum[]（右顶点匹配点）;

1. 【二分图最大匹配（匈牙利算法）】

O（）

**bool** path(**int** now,**int** m){    //now为当前点,m为右顶点

    vis[now] = **true**; //注意如果是没分好二分图的要加这句话

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(!vis[g[i].to]){

            vis[g[i].to] = **true**;

**if**(rnum[g[i].to] == -1 || path(rnum[g[i].to],m)){

                lnum[now] = g[i].to; rnum[g[i].to] = now; **return** **true**;

            }

        }

    }

**return** **false**;

}

**int** maxMatch(**int** n,**int** m){   //n为左顶点,m为右顶点

**int** re = 0;

    memset(lnum,-1,**sizeof**(lnum)); memset(rnum,-1,**sizeof**(rnum));

**for**(**int** i = 1; i <= n; ++i){

**if**(lnum[i] == -1){

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

**if**(path(i,n)){

                ++re;

            }

        }

    }

**return** re;

}

1. 【二分图最大匹配（HK算法）】

O（）

如果没分好点，还是尽量别用这个算法了。

跟普通匈牙利的区别在于，HK算法类似dinic那样，先进过BFS，找到最短的增广路，然后一口气将最短的增光路全部增光来降低复杂度。

BFS注意保存当前limit,在DFS增广最短路的时候，一旦右端点距离长度到达limit，除非该点是未匹配点，否则直接continue不要继续递归。即path函数中注释的这一段一定要记得加。

注意：对于求最大匹配，HK算法和dinic算法时间复杂度相同。

前置数据：链式前向星

bool vis[]（标记访问）

int r-num[]（左顶点匹配点）,lnum[]（右顶点匹配点）;

int dx[MAXN],dy[MAXN]；//BFS标记层数

queue<int> q1; //BFS用队列

**bool** path(**int** now,**int** limit){

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(!vis[g[i].to] && dx[now] + 1 == dy[g[i].to]){

            vis[g[i].to] = **true**;

**if**(dy[g[i].to] == limit && rnum[g[i].to] != -1){ **continue**;}  //防止找过长的增广路

**if**(rnum[g[i].to] == -1 || path(rnum[g[i].to],limit)){

                lnum[now] = g[i].to; rnum[g[i].to] = now; **return** **true**;

            }

        }

    }

**return** **false**;

}

**int** HK\_bfs(**int** n,**int** m){

**int** limit = INT\_MAX; memset(dx,-1,**sizeof**(dx)); memset(dy,-1,**sizeof**(dy));

**for**(**int** i = 1; i <= n; ++i){

**if**(lnum[i] == -1){

            q1.push(i); dx[i] = 0;

        }

    }

**while**(!q1.empty()){

**int** now = q1.front(); q1.pop();

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(dy[g[i].to] == -1){

                dy[g[i].to] = dx[now] + 1;

**if**(rnum[g[i].to] == -1){

                    limit = dy[g[i].to];

                }**else**{

**if**(dy[g[i].to] < limit && dx[rnum[g[i].to]] == -1){

                        dx[rnum[g[i].to]] = dy[g[i].to] + 1;

                        q1.push(rnum[g[i].to]);

                    }

                }

            }

        }

    }

**return** limit == INT\_MAX ? 0 : limit;

}

**int** maxMatch(**int** n,**int** m){

**int** re = 0,limit;

    memset(lnum,-1,**sizeof**(lnum)); memset(rnum,-1,**sizeof**(rnum));

**while**(limit = HK\_bfs(n,m)){

**for**(**int** i = 1; i <= n; ++i){

**if**(lnum[i] == -1){

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

**if**(path(i,limit)){

                    ++re;

                }

            }

        }

    }

**return** re;

}

1. 【二分图最大权匹配（KM算法）】

O（）

KM算法用来求解一个二分图中的最大权值完备匹配，也不一定是最大权，比如最小权也可以，具体看如何建模，建图必须要是邻接矩阵才行

在不要求必须是完备匹配的前提下，可以将不存在的边设为0，这样就能等价于求该问题的答案。

如果需要求取最小权，可以将权值变负，求取最大权值，然后取答案相反数即可。

其实思路很简单，就是很特别的贪心，通过顶标，一开始只保留最大权的边，然后匹配失败时候，再通过维护顶标来加边，再匹配，直到得到最大匹配，此时的最大匹配一定也是最大权的。因此KM必须要最大匹配，但是如果我们不要求最大匹配，也可以转化成最大的，方法上面有。

前置数据：

int g[MAXN][MAXN]; //邻接矩阵

int cx[MAXN],cy[MAXN],slack[MAXN],lnum[MAXN],rnum[MAXN];

//左顶标,右顶标，优化用数组，左匹配点，右匹配点

bool usex[MAXN],usey[MAXN];

//标记某次寻找增广路中，该点是否在增广路径中

**bool** path(**int** now,**int** m){

    usex[now] = **true**;

**for**(**int** i = 1; i <= m; ++i){

**if**(usey[i]){ **continue**;}

**if**(cx[now] + cy[i] - g[now][i] == 0){

            usey[i] = **true**;

**if**(rnum[i] == -1 || path(rnum[i],m)){

                lnum[now] = i; rnum[i] = now; **return** **true**;

            }

        }**else**{

            slack[i] = min(slack[i],cx[now] + cy[i] - g[now][i]);

        }

    }

**return** **false**;

}

**int** KMMaxMatch(**int** n,**int** m){    //左右顶点数，输出-1无解（没有最大匹配）

**int** re = 0,d; memset(lnum,-1,**sizeof**(lnum)); memset(rnum,-1,**sizeof**(rnum));

    memset(cy,0,**sizeof**(cy));

**for**(**int** i = 1; i <= n; ++i){

        cx[i] = 0;

**for**(**int** j = 1; j <= m; ++j){

            cx[i] = max(cx[i],g[i][j]);

        }

    }

**for**(**int** i = 1; i <= n; ++i){

        memset(usex,**false**,**sizeof**(usex)); memset(usey,**false**,**sizeof**(usey));

        fill(slack,slack + 1 + m,INT\_MAX);

**while**(!path(i,m)){

            d = INT\_MAX;

**for**(**int** j = 1; j <= m; ++j){

**if**(!usey[j]){

                    d = min(d,slack[j]);

                }

            }

**if**(d == INT\_MAX){ **return** -1;} //无法找到答案

**for**(**int** j = 1; j <= n; ++j){ **if**(usex[j]){ cx[j] -= d;}}

**for**(**int** j = 1; j <= m; ++j){ usey[j] ? cy[j] += d : slack[j] -= d;}

             //似乎可以改成下面这个，不敢确信，但是测了是AC的

            //for(int j = 1; j <= m; ++j){ if(usey[j]){ cy[j] += d;}}

            memset(usex,**false**,**sizeof**(usex)); memset(usey,**false**,**sizeof**(usey));

        }

    }

**for**(**int** i = 1; i <= n; ++i){

        re += g[i][lnum[i]];

    }

**return** re;

}

1. 【一般图最大匹配（带花树算法）（模板：uoj 79）】

O（）

对于一般图匹配问题，问题在于存在奇环，二分图可以证明是没有奇环的。带花树算法利用了一个奇环缩点后不影响答案这个结论，让环在搜索树上跟一朵朵花一样缩成一个点，然后找到最大匹配。

同样是对每个点找增广路，把起始点作为A类点，然后BFS分别把点标记为A类点和B类点，A类点搜索周围所有点，如果是B类点，就找与该B类点匹配的A类点入队列继续BFS（参考增广路中的走匹配边），如果发现未标记点，当然是发现增广路，沿着pre数组修改匹配即可，如果发现A类点，就是发现了奇环，这里通过修改pre数组巧妙的处理了奇环，使得能够正常匹配，通过暴力LCA（沿着pre）找到关键点，然后利用并查集缩点。边搜索边缩点，一直缩，花里可以有花….直到找到增广路。

当找到并修改匹配边和寻找下一个增广路时，把花全部展开就行。因为是在一颗搜索树上缩点（缩花）因此叫带花树算法，也叫Edmonds开花算法。

前置数据：链式前向星存图;

int match[MAXN],pre[MAXN],type[MAXN],fa[MAXN],tm,vis[MAXN];

//匹配,搜索前驱,点类型,并查集fa,lca辅助变量,暴力找lca辅助数组

queue<int> q;

**int** lca(**int** x,**int** y){ //暴力找LCA

    ++**tm**;

**while**(1){

**if**(x != -1){

            x = getFather(x); **if**(vis[x] == **tm**){ **return** x;}

            vis[x] = **tm**;

            match[x] == -1 ? x = -1 : x = pre[match[x]];

        }

        swap(x,y);

    }

}

**void** shrink(**int** x,**int** y,**int** root){    //缩花

**while**(getFather(x) != root){

        pre[x] = y; y = match[x];

**if**(type[y] == 2){ type[y] = 1; q.push(y);}

**if**(getFather(x) == x){ fa[x] = root;}

**if**(getFather(y) == y){ fa[y] = root;}

        x = pre[y];

    }

}

**bool** path(**int** st,**int** n){   //BFS

    memset(pre,-1,**sizeof**(pre)); memset(type,0,**sizeof**(type));

**for**(**int** i = 1; i <= n; ++i){ fa[i] = i;}

    type[st] = 1; qclear(q); q.push(st);

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

**int** now = q.front(); q.pop();

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(getFather(now) == getFather(g[i].to) || type[g[i].to] == 2){ **continue**;}

**if**(type[g[i].to] == 0){

                type[g[i].to] = 2; pre[g[i].to] = now;

**if**(match[g[i].to] == -1){

**int** cur,last; cur = g[i].to;

**while**(cur != -1){

                        last = match[pre[cur]];

                        match[cur] = pre[cur]; match[pre[cur]] = cur;

                        cur = last;

                    }

**return** **true**;

                }

                type[match[g[i].to]] = 1; q.push(match[g[i].to]);

            }**else**{

**int** root = lca(now,g[i].to);

                shrink(now,g[i].to,root);

                shrink(g[i].to,now,root);

            }

        }

    }

**return** **false**;

}

**int** maxMatch(**int** n){    //寻找最大匹配

**int** re = 0;

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

**for**(**int** i = 1; i <= n; ++i){

**if**(match[i] == -1){

            memset(vis,0,**sizeof**(vis)); time = 0;  //删掉这行可以加效率,有概率wa

**if**(path(i,n)){

                ++re;

            }

        }

    }

**return** re;

}

【判断是否为二分图】

O（n）

直接跑个BFS染个色呗，O（n）的复杂度

前置数据：queue<int> q;

bool vis[MAXN]; //标记是否访问，for循环扫全部连通块

int color[MAXN]; //染色数组

**bool** isBGraph(**int** st){

**while**(!q.empty()){ q.pop();} q.push(st); color[st] = 1;

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

**int** now = q.front(); q.pop(); vis[now] = **true**;

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(color[g[i].to] == color[now]){ **return** **false**;}

**if**(color[g[i].to] == -1){

                color[g[i].to] = color[now] == 1 ? 2 : 1;

                q.push(g[i].to);

            }

        }

    }

**return** **true**;

}

1. 【稳定婚姻匹配】

问题简单可描述成 有N男N女，每个人都按照他对异性的喜欢程度排名。现在需要写出一个算法安排这N个男的、N个女的结婚，要求两个人的婚姻应该是稳定的。

稳定：假如A与B匹配，C和D匹配，A相对于B，更喜欢D，而D相对于C更喜欢与A一起，这样A和D就很有可能会发生“”私奔“”,因为他们都对目前的对象不感到满意,如果其中一方，如A比起D确实跟喜欢B，那么D只能认命，这样的婚姻是稳定的。

算法中采用了男生主动追求女孩的形式。

    算法步骤描述：

        第一轮，每个男人都选择自己名单上排在首位的女人，并向她表白。这种时候会出现两种情况：（1）该女士还没有被男生追求过，则该女士接受该男生的请求。（2）若该女生已经接受过其他男生的追求，那么该女生会将该男士与她的现任男友进行比较，若更喜欢她的男友，那么拒绝这个人的追求，否则，抛弃其男友

      第一轮结束后，有些男人已经有女朋友了，有些男人仍然是单身。

       在第二轮追女行动中，每个单身男都从所有还没拒绝过他的女孩中选出自己最中意的那一个，并向她表白，不管她现在是否是单身。这种时候还是会遇到上面所说的两种情况，还是同样的解决方案。直到所有人都不在是单身。

前置数据：

int boyRankList[MAXN][MAXN],girlRankList[MAXN][MAXN];

//编号为i男(女)优先级第j高的人是谁

int boyMatch[MAXN],girlMatch[MAXN];

//男女生的匹配答案

int relation[MAXN][MAXN],manpos[MAXN],cnt;

//relation表示女i对男j的优先度

//manpos记录男生已经对自己排行榜上前几个人告白过

//cnt辅助计算relation数组

queue<int> q;

//用于模拟算法告白流程

**void** findMatch(){   //建立匹配（必有解）

**while**(!q.empty()){ q.pop();}

**for**(**int** i = 1; i <= n; ++i){

        q.push(i); boyMatch[i] = girlMatch[i] = -1; manpos[i] = 0;

    }

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

**int** s = q.size();

**for**(**int** i = 1; i <= s; ++i){

**int** now = q.front();

            ++manpos[now];

            q.pop();

**if**(girlMatch[boyRankList[now][manpos[now]]] == -1){  //没男票

                girlMatch[boyRankList[now][manpos[now]]] = now;  //交男女朋友

                boyMatch[now] = boyRankList[now][manpos[now]];

            }**else**{

**if**(relation[boyRankList[now][manpos[now]]][now] < relation[boyRankList[now][manpos[now]]][girlMatch[boyRankList[now][manpos[now]]]]){   //比当前男友优先级更高

                    boyMatch[girlMatch[boyRankList[now][manpos[now]]]] = -1;  //前男友变成单身,重新加入队列

                    q.push(girlMatch[boyRankList[now][manpos[now]]]);

                    girlMatch[boyRankList[now][manpos[now]]] = now;  //抛弃前男友,和当前男生在一起

                    boyMatch[now] = boyRankList[now][manpos[now]];

                }**else**{

                    q.push(now);

                }

            }

        }

    }

}

1. 【各种变形问题】

【棋盘匹配（HDU1045）】棋盘上给障碍物，问可以放多少车可以不互相攻击。对棋盘先预处理，每一行连通的相连棋盘的看成一个点，每一列也是，编号，如果某一行和某一列重叠了，就连边。可以知道，编号完的每一行和每一列，只可能有一个相交点。匹配的意义为，将棋子放在行和列相交的点上。很显然答案就是最大匹配了！

【求哪几个匹配是所有最大匹配都有的（HDU1281）】直接求个最大匹配，枚举每对匹配，然后把那条边删掉，看最大匹配会不会变小就好了，O（n^3）

【棋盘上下左右匹配（HDU4185）】很多办法，其中之一是棋盘黑白染色分点后随便搞搞。

【最小顶点覆盖（HDU1054）】二分图中选最少的节点集合，使得所有边都跟集合内至少一个点有关联。最少点数 = 最大匹配数

【有向图最小不相交路径覆盖（HDU1151）】有向图里选择最少的不相交路径，使得所有路径都被走过，解法是把点拆成i和j，就是出点和入点，类似网络流，把出点和入点作为2个集合，如果i能到j，就左边的i连右边j，答案为总顶点数 – 最大匹配数。

【有向图最小相交路径覆盖（POJ2594）】先用floyd求传递闭包，对于所有能走到的点作拆点后连边，就变成了有向图最小不相交路径覆盖问题了。

【二分图最大独立集（HDU3829）】答案为总顶点数 – 最大匹配数。

【二分图多重匹配（POJ2289）】一个点能匹配多次，二分图的话，加个num表示能匹配的次数，当然也可以搞个网络流，建个超级源点和超级汇点，源点容量都是1，右边匹配点到汇点的容量按你要能匹配的个数来确定就好。

【二分图最大团】最大团 = 补图的最大独立集

【二分图最大权完美匹配（HDU2255）】KM的板子题，也可以用费用流解决

1. **欧拉路和欧拉回路（模板 uoj117）**

【模板题题意说明】

hdu6311 求多连通块无向图可以最少分成几条路径使得每条边都经过且只经过一次，并输出每个的路径

uoj117 有向图和无向图回路板子题，题目有点坑，点不是1 - n的，且有可能多连通块

经过所有边一次的路径（回路）

解法有Fluery算法和Hierholzer算法，这里用Hierholzer算法。

Fluery算法 时间复杂度为O（m），需要预先知道哪些边是桥，每次除非没办法，否则不走桥。

Hierholzer算法 优化后时间复杂度为O（m），直接DFS，如果这个点没有可以移动的边，输出这个点，并删边（这个是优化，如果还是暴力判断边是否被走过复杂度会到平房级），输出是倒序的。

**【有向图】**

欧拉回路：图连通，所有节点的出度等于入度，任意选起点

欧拉路：图连通，所有节点的出度等于入度；或者除两个节点以外的其余节点的入度和出度都相等，且这两个节点一个满足出度-入度等于1，另一个满足入度--出度等于1。起点为出度-入度=1的点，终点为入度-出度=1的点

**【无向图】**

欧拉回路：图连通，无奇度节点，任意选起点（HDU 1878）

欧拉路：图连通，只有两个奇度节点或者无奇度节点，如果有奇数节点，这两个奇数节点必定是终点和起点

注意点：注意多连通块无解，除去只有1个点的连通块。

前置数据：判断是否存在欧拉路

vector<int> path; //存欧拉路（路径按边序号排列,负数代表走反路，如果要按点序号排列,直接pushback（now）应该就好了）

int last[MAXN]; //优化，同dinic

**void** findpath(**int** now){

**for**(**int** i = last[now]; i != -1; i = last[now]){

        last[now] = g[i].next;

**if**(t == 1){ //无向图

**if**(g[i].ex){

                g[i].ex = **false**; g[i ^ 1].ex = **false**;

                findpath(g[i].to);

**if**(i % 2 == 0){

                    path.push\_back(i / 2 + 1);

                }**else**{

                    path.push\_back(-(i / 2 + 1));

                }

            }

        }**else**{ //有向图

            findpath(g[i].to);

            path.push\_back(i + 1);

        }

    }

}

**void** getpath(**int** st){

    path.clear(); for(int i = 1; i <= n; ++i){ last[i] = head[i];}

    findpath(st);

    reverse(path.begin(),path.end());

}

【无向图多笔画问题】

设，奇数度点有k个，需要max{k/2,1}  笔就可以画完。

1. **最大团（极大团）和最大独立集**

团：图中的一个完全子图

极大团：非任何团的子团

最大团：顶点数最多的极大团

最大独立集：反图的最大团

1. 【最大团】（模板HDU 1530 必须运行低于3000MS以内可用）

直接暴力DFS，加剪枝；

基础： 我们令cnum[i]表示点i到n中的最大团大小，则有cnum[i] = cnum[i + 1] 或cnum[i + 1] + 1,也就是说具有单调性（类似于DP），这样我们可以利用这个性质进行剪枝，有点类似于二分图增广路，倒着每次添加一个点i，看看能否形成更优的答案，如果形成了，那么直接return就好了。另外选点的时候，顺着按下标从小到大选，这样可以避免枚举一些重复情况。

剪枝1： 如果当前选取的点数加上剩下还可能选的点数都不能比之前的最优解优秀，那么return；

剪枝2：对于即将要选择的点x，如果cnum[x] 加上当前答案都不能比之前的最优解优秀，那么return；因为这说明就算x - n里面的最大团完全可以被添加进来，也不能得到更优的解

剪枝3：如基础所说，如果答案发生更新，直接return；

前置数据：int g[MAXN][MAXN],cnum[MAXN],cans;

//分别图的邻接矩阵，cnum为优化用数组，cans存最大团数量。

**bool** dfs(**int** step,**int** \*P,**int** Plen){

**int** P2[MAXN],P2len;

**if**(step > cans){ cans = step; **return** **true**;} //剪枝3

**for**(**int** i = 1; i <= Plen; ++i){

**if**(step + Plen - i + 1 <= cans){ **break**;} //剪枝1

**if**(step + cnum[P[i]] <= cans){ **break**;} //剪枝2

        P2len = 0;

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

**if**(g[P[i]][P[j]]){

                P2[++P2len] = P[j];

            }

        }

**if**(dfs(step + 1,P2,P2len)){ **return** **true**;}

    }

**return** **false**;

}

**void** maxClique(){

**int** P[MAXN],Plen = 0;

    cans = 1; cnum[n] = 1;

**for**(**int** i = n - 1; i >= 1; --i){

        Plen = 0;

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

**if**(g[i][j]){ P[++Plen] = j;}

        }

        dfs(1,P,Plen);

        cnum[i] = cans;

    }

}

1. 【极大团（模板POJ2989）】

剪枝1：若当前Not集合中存在一个点，与Can中所有点都相连，则在未来的搜索中它永远不会离开Not集合，故剪枝。

剪枝2：这个剪枝为了优化剪枝1的效果，并不是一个新的剪枝。原本的搜索我们总是从Can集合中随意选择一个继续dfs 但是我们可以挑选一个特殊的节点，来增强剪枝1的效果。设每个在Not集合中的节点有一个cnt值，为Can集合中与它不相连的节点个数。 我们于是选择Can集合中与cnt最小的节点不相连的节点进行下一步dfs。

前置数据：int g[MAXN][MAXN],cnum;

//图的邻接矩阵,cnum为极大团个数

**void** dfs(**int** \*P,**int** \*X,**int** Plen,**int** Xlen){

**if**(Plen == 0){

**if**(Xlen == 0){ ++cnum;}

**return**;

    }

**int** P2[MAXN],P2len,X2[MAXN],X2len,cnt,id,maxcnt;

    id = 0;

**for**(**int** i = 1; i <= Xlen; ++i){

        cnt = 0;

**for**(**int** j = 1; j <= Plen; ++j){

**if**(g[X[i]][P[j]] == 0){ ++cnt;}

        }

**if**(id == 0 || cnt < maxcnt){ maxcnt = cnt; id = i;}

    }

**if**(id != 0 && maxcnt == 0){ **return**;} //剪枝1

**for**(**int** i = 1; i <= Plen; ++i){

**if**(id != 0){ //剪枝2

**int** j;

**for**(j = i; j <= Plen; ++j){

**if**(g[P[j]][X[id]] == 0){ **break**;}

            }

            swap(P[i],P[j]);

        }

        P2len = X2len = 0;

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

**if**(g[P[i]][P[j]] == 1){ P2[++P2len] = P[j];}

        }

**for**(**int** j = 1; j <= Xlen; ++j){

**if**(g[P[i]][X[j]] == 1){ X2[++X2len] = X[j];}

        }

        dfs(P2,X2,P2len,X2len);

**if**(cnum > 1000){ **return**;}

        X[++Xlen] = P[i]; cnt = 0; --maxcnt;

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

**if**(g[P[i]][P[j]] == 0){ ++cnt;}

        }

**if**(id == 0 || cnt < maxcnt){ maxcnt = cnt; id = Xlen;};

**if**(id != 0 && maxcnt == 0){ **return**;} //剪枝1

    }

}

**void** getClique(){

**int** P[MAXN],Plen,X[MAXN],Xlen;

    cnum = 0; Xlen = 0; Plen = 0;

**for**(**int** i = 1; i <= n; ++i){ P[++Plen] = i;}

    dfs(P,X,Plen,Xlen);

}

1. **图上染色**
2. 【无向图色数（模板HDU1373）】

对无向图染色直接暴力染色就行，注意处理多连通块的情况。

前置数据：int color[MAXN],use[MAXN];

//染色情况，use辅助用数组，判断什么颜色没用过

queue<int> q;

**int** dye(**int** st){  //返回该连通块最大色数,注意main里对color置-1

**int** re = 0;

    q.push(st);

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

**int** now = q.front(); q.pop(); **int** nowcolor = 1;

**for**(**int** i = 1; i <= n; ++i){ use[i] = **false**;}

**for**(**int** i = 1; i <= n; ++i){

**if**(g[now][i] == 1 && color[i] != -1){

                use[color[i]] = **true**;

            }

**if**(g[now][i] == 1 && color[i] == -1){

                q.push(i);

            }

        }

**while**(use[nowcolor]){

            ++nowcolor;

        }

        color[now] = nowcolor;

        re = max(re,nowcolor);

    }

**return** re;

}

1. **数据结构**
2. **树上性质和各种问题**
3. 【树的直径】

直径是树的最长路径，求法是选任意一个点BFS，然后返回最远的点，然后从那个点再BFS一次，再返回另一个最远的点，这个2个点之间的路径为直径

1. 【树的重心】

重心是从树里找到一个点作为根，其所有的子树中最大的子树节点数尽可能少，每个节点到重心的距离和是最小的（不超过n / 2）。重心的求法是树形DP，对于一个节点来说，如果以它作为根，它的子树全部节点数不超过总节点数一半，这个点就是树的重心。

前置数据：链式前向星

vector<int> gp; //存放所有的重心

int ts[MAXN]; //辅助数组，存子树大小的

**int** gpdfs(**int** n,**int** fa,**int** now){

**int** maxs = 0; ts[now] = 0;

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].to != fa){

            ts[now] += gpdfs(n,now,g[i].to);

            maxs = max(maxs,ts[g[i].to]);

        }

    }

    ts[now]++;

    maxs = max(maxs,n - ts[now]);

**if**(maxs <= n / 2){

        gp.push\_back(now);

    }

**return** ts[now];

}

**void** getgp(**int** n,**int** st){

    gp.clear(); gpdfs(n,-1,st);

}

1. 【树上差分】（模板 NOIP2015 运输计划）

对树上某段路径上的所有边或者点做一些操作（比如加1）O（1），询问这段路径上的权值和O（n）

边差分：

tdf[i]表示点i到它父亲的这条边经过的次数。

例如u -> v

tdf[u]++; tdf[v]++; tdf[a] -= 2; a为u和v的lca

前缀和的话，tdf[x]等于tdf[x] + 所有儿子的tdf和

点差分:

tdf[i]表示点i经过的次数，tdf[a]--即可，其他部分和边差分相同。

1. 【离线Tarjan算法求LCA】

O(n + q)

在树内求2个点的最近公共祖先，或者是树上任意两点间最短路。

做法是DFS一遍这颗树，当回溯的时候，标记点被访问完全，然后枚举所有包含该节点的询问，如果存在询问中另一个节点已经被访问，那么那个节点缩入的节点（也就是并查集中的根节点）就是2个的LCA，然后距离也能算，枚举完循环后，把当前前缩入其父节点，也就是并查集并入父节点，因此并的时候需要注意方向。

前置数据结构【并查集】

前置结构：

**struct** star{   //存树（链式前向星）

**int** to,next,w;

};

**struct** query{   //存问题集,离线回答（链式前向星）

**int** v,next,id;

};

**struct** ansSet{   //存答案集合

**int** dis,anc;   //dis为2点间最短距离，anc为公共祖先

};

前置数据（省略2个链式前向星的数据和并查集相关数据）

qlen,qhead,head,len,fa.

ansSet ans[MAXN]; //存储最后答案

bool vis[MAXN]; //判断是否访问过,计算时用

int dis[MAXN]; //根节点到任意节点的距离,需要计算任意2点间距离的时候加入

**void** bfs(**int** st){   //预处理距离（如果需要计算距离的话）

    queue<**int**> q;

    memset(dis,-1,**sizeof**(dis));

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

        q.pop();

    }

    q.push(st); dis[st] = 0;

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

**int** now = q.front();

        q.pop();

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(dis[g[i].to] == -1){

                dis[g[i].to] = dis[now] + g[i].w;

                q.push(g[i].to);

            }

        }

    }

**return**;

}

**void** lcaInit(**int** n){   //进行lca前预处理,使用主算法前必须先调用

**for**(**int** i = 1; i <= n; ++i){

        fa[i] = i;

    }

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

}

**void** lcaTarjan(**int** last,**int** now){   //主算法,last为父节点,now为目前节点

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(last != g[i].to){

            lcaTarjan(now,g[i].to);

        }

    }

    vis[now] = **true**;

**for**(**int** i = qhead[now]; i != -1; i = q[i].next){

**if**(vis[q[i].v]){

**int** common = getfather(q[i].v);

**int** sum = dis[now] + dis[q[i].v] - 2 \* dis[common];

            ans[q[i].id].anc = common;

            ans[q[i].id].dis = sum;

        }

    }

    add(last,now); //顺序不能颠倒

**return**;

}

1. 【在线RMQ加时间戳算法求LCA】（模板 洛谷P3379）

O（nlogn + n+q）

没算距离，如果需要距离需要增加dis数组，并且在dfs时预处理出距离。

RMQ的dp数组大小按需要开！

前置数据：

int first[MAXN] //节点在序列第一次出现的位置

int dep[MAXN] //深度数组

int stamp [2\*MAXN] //dfs序列，返回根节点的时候也要记录

int dlen; 记录dfs序列长度

int dp[MAXN \* 2][20]; //RMQ辅助数组

要找u和v的LCA，取2个点第一次出现的位置，然后丢进RMQ即可

**void** RMQInit(**int** n){   //预处理

**for**(**int** i = 1; i <= n; ++i){

        dp[i][0] = stamp[i]; dp[i][0] = stamp[i];

    }

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

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

**if**(dep[dp[i][j - 1]] < dep[dp[i + (1 << (j - 1))][j - 1]]){

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

            }**else**{

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

            }

        }

    }

}

**int** RMQ(**int** l, **int** r){   //查找所需的元素

**if**(l > r){ swap(l,r);}

**int** k = 0;

**while**((1 << (k + 1)) <= r - l + 1){

        ++k;

    }

**return** dep[dp[l][k]] < dep[dp[r - (1 << k) + 1][k]] ? dp[l][k] : dp[r - (1 << k) + 1][k];

}

**void** dfs(**int** fa,**int** now){

    stamp[++dlen] = now; dep[now] = dep[fa] + 1; first[now] = dlen;

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].to == fa){ **continue**;}

        dfs(now,g[i].to);

        stamp[++dlen] = now;

    }

}

//主函数调用方法

dfs(s,s); RMQInit(dlen);

RMQ(first[u],first[v])

1. 【图内生成树个数】

度数矩阵 a[i][i]为点i的度 其他位置为0   
邻接矩阵 a[i][i]=0; a[i][j]=[i与j相连]   
基尔霍夫矩阵 = 度数矩阵 - 邻接矩阵   
Matrix-Tree 定理：   
一个n个点m条边的无向图的生成树总数为其对应的基尔霍夫矩阵的n-1阶余子式(行列式)

行列式某元素的余子式：行列式划去该元素所在的行与列的各元素,剩下的元素按原样排列,得到的新行列式.（就是不用考虑正负的那个）  
行列式某元素的代数余子式：行列式某元素的余子式与该元素对应的正负符号的乘积.

1. 【最小生成树的变种】

**1.增量最小生成树**（从包含n个点的空图开始，依次加入一些边，求当前状态下的最小生成树，如果不连通则无解）

解决办法：根据理论，一个若有一个环E，其中权值最大的边一点不在生成树内，因此每次加入一条边如果形成了环，可以用DFS或者BFS扫描一遍环，永久去除这条最大的边即可。因为只有一条固定的路，所以DFS和BFS效率并不会很低，由于每次输出都去除了之前的无用边，因此效率大大提高，时间复杂度为O（nmlogn）。

**2.最小瓶颈生成树**（）

1. 【树上启发式合并】

最典型的就是并查集按秩合并，有些要求重儿子（定义同树剖里的）

【HDU6430】对于树上点i，求LCA是i的所有点对的权值的gcd的最大值，作因数集合，所有小的集合往大的上去合并。

1. **栈(队列)**

【单调栈和单调队列】

顾名思义,维护一个严格单调的栈,可用来求取一个序列中每个值作为最大值或者最小值的最大连续区间，**单调队列**可以O（n）求滑窗最值。

例如 2 1 4 2 5 以4为最大值的连续区间,向左能扩展2格，向右能扩展1格子,即为[1,4]同理可得最小值的连续区间和其他值的最大最小连续区间.

前置结构 stack<int>

int lmin[i] //第i个值作为最小值向左能扩展的最大量

int rmin[i] //第i个值作为最小值向右能扩展的最大量

**void** getMin(**int** n){   //求以该值为最小值的最大扩展,n为序列长度

**while**(!stk.empty()){

     stk.pop();

    }

    stk.push(1); num[n+1] = -1;   //将一个最小值插入队尾保证最后栈内所有值弹出

**for**(**int** i = 2; i <= n+1; ++i){

**if**(num[i] > num[stk.top()]){

            stk.push(i);

            lmin[i] = 0;

        }**else**{

**while**(!stk.empty() && num[i] <= num[stk.top()]){

                rmin[stk.top()] = i - stk.top() - 1;

                lmin[i] = lmin[stk.top()] + i - stk.top();

                stk.pop();

            }

            stk.push(i);

        }

    }

}

【中缀表达式转后缀表达式（逆波兰式）】

思路，遇到数字直接放入结果数组，遇到操作符，压人栈，压入前如果栈顶存在优先级大于等于的，就弹出，注意逻辑运算的！！，对于这种情况，需要先预处理掉，因为2个取反相当于空操作，下面这个目前只能逻辑运算。

逻辑运算的T和F要预先处理成0和1

前置数据：

map<char,int> rk; //设置优先级

stack<char> ope; //辅助栈

char a[MAXN],ex[MAXN],ex2[MAXN]; //原表达式，预处理后中缀，后缀结果

int len1,len2; //中缀和后缀的长度

**void** setrk(){

    rk['|'] = 1; rk['^'] = 2; rk['&'] = 3; rk['!'] = 9;

}

**void** changeExp(){

    len2 = 0;

**for**(**int** i = 0; i < len1; ++i){

**if**(ex[i] == '0' || ex[i] == '1'){

            ex2[len2++] = ex[i];

        }**else** **if**(ex[i] == '('){

            ope.push(ex[i]);

        }**else** **if**(ex[i] == ')'){

**while**(ope.top() != '('){

                ex2[len2++] = ope.top(); ope.pop();

            }

            ope.pop();

        }**else**{

**while**(!ope.empty() && ope.top() != '(' && rk[ope.top()] >= rk[ex[i]]){

                ex2[len2++] = ope.top(); ope.pop();

            }

            ope.push(ex[i]);

        }

    }

**while**(!ope.empty()){

        ex2[len2++] = ope.top(); ope.pop();

    }

}

【栈对后缀表达式求值】

**int** solveExp(){

**for**(**int** i = 0; i < len2; ++i){

**if**(ex2[i] == '0' || ex2[i] == '1'){

            number.push(ex2[i] - '0');

        }**else** **if**(ex2[i] == '!'){

**int** now = number.top(); number.pop(); now = now ^ 1; number.push(now);

        }**else**{

**int** now1 = number.top(); number.pop();

**int** now2 = number.top(); number.pop();

**switch**(ex2[i]){

**case** '|': number.push((now1 | now2)); **break**;

**case** '&': number.push((now1 & now2)); **break**;

            }

        }

    }

**int** re = number.top(); number.pop();

**return** re;

}

1. **并查集（查询优化）**

前置数据

int fa[] //节点数组,大小等于节点数

//初始化

**for**(**int** i = 1; i <= n; ++i){

    fa[i] = i;

}

**int** getfather(**int** x){ //寻找根节点，即查操作

**return** fa[x]==x?x:fa[x]=getfather(fa[x]);

}

**void** add(**int** x,**int** y){ //连接2个节点，即并操作

    x = getfather(x); y = getfather(y);

    fa[y] = x;

**return**;

}

1. **线段树（模板HDU1166）**

O（nlogn）建立

O（logn）查询

O（logn）修改

只给出区间查询区间修改的线段树，灵活乱用就好。

//前置宏定义

#define left now<<1   //左孩子

#define right (now<<1)+1   //右孩子

#define mid ((l+r)>>1)   //分段中点

**struct** segmentTree{   //线段树节点

**int** sum;

};

**const** **int** MAXN = 1e1;   //数据长度

segmentTree tree[MAXN\*4];   //数据长度的4倍

**void** builtTree(**int** now,**int** l,**int** r){   //建树

**if**(l == r){

        tree[now].add = 0;

        tree[now].sum = data[l]; **return**;   //叶子节点赋值

    }

    builtTree(left,l,mid); builtTree(right,mid+1,r);

    tree[now].sum = tree[left].sum + tree[right].sum;   //内节点处理

    tree[now].add = 0;

**return**;

}

**void** updata(ll now,ll l,ll r,ll setl,ll setr,ll num){   //区间更新

**if**(setl <= l && setr >= r){

        tree[now].add += num;

**return**;

    }

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

    tree[left].add += tree[now].add;

    tree[right].add += tree[now].add;

    tree[now].add = 0;

    tree[now].sum += (setr - setl + 1) \* num;

**if**(setl <= mid){

        updata(left,l,mid,setl,min(mid,setr),num);

    }

**if**(setr > mid){

        updata(right,mid+1,r,max(setl,mid+1),setr,num);

    }

**return**;

}

ll getSum(ll now,ll l,ll r,ll ql,ll qr){   //区间更新的区间查询

**if**(ql <= l && qr >= r){

**return** tree[now].sum + (tree[now].add\*(r-l+1));

    }

    ll ans = 0;

    ans += tree[now].add \* (qr - ql + 1);

**if**(ql <= mid){ ans += getSum(left,l,mid,ql,min(mid,qr));}

**if**(qr > mid){ ans += getSum(right,mid+1,r,max(ql,mid+1),qr);}

**return** ans;

}

【二维线段树（四叉树） （模板HDU1892）】

前置宏定义：

#define midx ((x + xx) >> 1)

#define midy ((y + yy) >> 1)

#define tl (now << 2)

#define tr ((now << 2) + 1)

#define bl ((now << 2) + 2)

#define br ((now << 2) + 3)

struct segmentTree2{

int num;

};

segmentTree2 tree[MAXN \* MAXN \* 6];

区间修改，单点查询，数字开6倍

**void** builtTree(**int** now,**int** x,**int** y,**int** xx,**int** yy){ //左上右下,主对角线

**if**(x == xx && y == yy){

        tree[now].num = 1; **return**;

    }

    builtTree(tl,x,y,midx,midy); tree[now].num = tree[tl].num; //左上

**if**(y != yy){ //右上

         builtTree(tr,x,midy + 1,midx,yy); tree[now].num += tree[tr].num;

    }

**if**(x != xx){ //左下

         builtTree(bl,midx + 1,y,xx,midy); tree[now].num += tree[bl].num;

    }

**if**(y != yy && x != xx){ //右下

        builtTree(br,midx + 1,midy + 1,xx,yy); tree[now].num += tree[br].num;

    }

}

**int** updateTree(**int** now,**int** x,**int** y,**int** xx,**int** yy,**int** setx,**int** sety,**int** num){

**int** p;

**if**(x == xx && y == yy){

**if**(num >= 0){

            p = num;

        }**else**{

            p = abs(num) <= tree[now].num ? num : -tree[now].num;

        }

        tree[now].num += p;

**return** p;

    }

**if**(setx <= midx && sety <= midy){ //左上

        p = updateTree(tl,x,y,midx,midy,setx,sety,num);

    }

**if**(y != yy && setx <= midx && sety > midy){ //右上

        p = updateTree(tr,x,midy + 1,midx,yy,setx,sety,num);

    }

**if**(x != xx && setx > midx && sety <= midy){ //左下

        p = updateTree(bl,midx + 1,y,xx,midy,setx,sety,num);

    }

**if**(y != yy && x != xx && setx > midx && sety > midy){ //右下

        p = updateTree(br,midx + 1,midy + 1,xx,yy,setx,sety,num);

    }

    tree[now].num += p;

**return** p;

}

**int** getsum(**int** now,**int** x,**int** y,**int** xx,**int** yy,**int** qx,**int** qy,**int** qxx,**int** qyy){

**if**(qx <= x && qy <= y && qxx >= xx && qyy >= yy){

**return** tree[now].num;

    }

**int** re = 0;

**if**(qx <= midx && qy <= midy){  //左上

        re += getsum(tl,x,y,midx,midy,qx,qy,min(qxx,midx),min(qyy,midy));

    }

**if**(y != yy && qx <= midx && qyy > midy){ //右上

        re += getsum(tr,x,midy + 1,midx,yy,qx,max(qy,midy + 1),min(qxx,midx),qyy);

    }

**if**(x != xx && qxx > midx && qy <= midy){  //坐下

        re += getsum(bl,midx + 1,y,xx,midy,max(qx,midx + 1),qy,qxx,min(qyy,midy));

    }

**if**(y != yy && x != xx && qxx > midx && qyy > midy){ //右下

        re += getsum(br,midx + 1,midy + 1,xx,yy,max(qx,midx + 1),max(qy,midy + 1),qxx,qyy);

    }

**return** re;

}

【可持久化线段树（主席树）】（洛谷P3834 HDU 2665）

可以保存历史数据的线段树。主要思路就是线段树修改一般只涉及部分区域，那么对涉及到的区域动态加节点维护就好了。

模板：求第k小，需要先离散化

前置数据int tcnt,rt[MAXN],rs[MAXN \* 72],ls[MAXN \* 72]; //动态开点，第i次修改的根位置，左右子树位置

segmentTree tree[MAXN \* 72]; //线段树

//建树，正常的来就行

**void** builtTree(**int** now,**int** l,**int** r){

    tcnt = max(tcnt,now);

**if**(l == r){

        tree[now].num = 0; **return**;

    }

    ls[now] = left;

    rs[now] = right;

    builtTree(left,l,mid); builtTree(right,mid + 1,r);

}

//添加修改，第一个参数是新根位，在main里要++cnt，然后lastid为在第id个历史版本上的线段树做修改

//pos为修改的位置，这里是单点修改

**void** addTree(**int** now,**int** l,**int** r,**int** lastid,**int** pos){

    tree[now].num = tree[lastid].num + 1;

**if**(l == r){ **return**;}

    ls[now] = ls[lastid];

    rs[now] = rs[lastid];

**if**(pos <= mid){

        ls[now] = ++tcnt;

        addTree(ls[now],l,mid,ls[lastid],pos);

    }**else**{

        rs[now] = ++tcnt;

        addTree(rs[now],mid + 1,r,rs[lastid],pos);

    }

}

//查询回答，区间查询的话，把2个线段树差分就好了

**int** getans(**int** now,**int** now2,**int** l,**int** r,**int** qk){

**if**(l == r){

**return** l;

    }

**int** lv = tree[ls[now]].num - tree[ls[now2]].num;

**if**(lv >= qk){

**return** getans(ls[now],ls[now2],l,mid,qk);

    }**else**{

**return** getans(rs[now],rs[now2],mid + 1,r,qk - lv);

    }

}

1. **树状数组（模板HDU1166）**

比较容易写单点查,区间增删，或者是区间查，单点增删，实质就是线段树割开了一段下来，快速动态维护前缀和

O（logn）修改

O（logn）查询

前置数据

int bit[] //节点数，从1开始才行

**int** lowbit(**int** x){

**return** x & (-x);

}

**int** getSum(**int** x){   //求1-x的区间和

**int** sum = 0;

**while**(x > 0){

        sum += bit[x]; x -= lowbit(x);

    }

**return** sum;

}

**void** bitAdd(**int** x, **int** num){   //bit[x]加上num

**while**(x <= MAXN){

        bit[x] += num; x += lowbit(x);

    }

**return**;

}

【单点查询区间修改】

比如要修改的区间为【A,B】操作为bitAdd(A,x); bitAdd(B,-x);

但是此时getSum(i)表示i该点的值了。

【求逆序对数】

记住用stable\_sort,否则会wa.

【二维树状数组 （模板HDU1892）】

**void** add(**int** x,**int** y,**int** num){

**while**(x <= 1001){

**int** nowy = y;

**while**(nowy <= 1001){

            bit[x][nowy] += num; nowy += lowbit(nowy);

        }

        x += lowbit(x);

    }

}

**int** getsum(**int** x,**int** y){

**int** re = 0;

**while**(x > 0){

**int** nowy = y;

**while**(nowy > 0){

            re += bit[x][nowy]; nowy -= lowbit(nowy);

        }

        x -= lowbit(x);

    }

**return** re;

}

**int** getans(**int** x,**int** y,**int** xx,**int** yy){

**return** getsum(xx,yy) - getsum(x - 1,yy) - getsum(xx,y - 1) + getsum(x - 1,y - 1);

}

1. **字典树**

（静态实现）

**struct** node{

**int** cnt;

**int** next[26];

};

node tree[200010];  //申请一颗长度为N的字典树

**int** len=1;  //tree[1] 为根节点,0不能使用。

//基本操作

**void** insertTree(**int** now,**char** \*str,**int** length){   //添加一个字符串

**for**(**int** i = 0; i < length; ++i){

**int** v = str[i]-'a';   //如果存的是数字修改这里

**if**(tree[now].next[v] == 0){

            tree[now].next[v] = ++len;

            tree[len].cnt = 0;

        }

        now = tree[now].next[v];

        ++tree[now].cnt;

    }

}

**int** searchTree(**int** now,**char** \*str,**int** length){   //查询一个前缀

**for**(**int** i = 0; i < length; ++i){

**int** v = str[i] - 'a';

**if**(tree[now].next[v] != 0){

            now = tree[now].next[v];

        }**else**{

**return** 0;

        }

    }

**return** tree[now].cnt;

}

【各种变形问题】

【求字符串b是多少个字符串的子串(一个字符串内算1个b)】查询函数不变，在添加字符串的时候把这个字符串的所有后缀全部存进去，如abcd你必须存abcd,bcd,cd,d.为字典树添加一个id属性，表明这些是第几个字符串加入的，因为同一个字符串只记录一个子串。只有id不相同的时候，cnt才加一。

【求N个字符串内一共存在多少个子串是字符串b(一个字符串内可能有多个b)】构造方法同上变形，唯一改变的是不加入id限制条件。

【对于某些异或的问题可以用01字典树】针对异或操作的黑科技，将一个非负正整数以2进制形式存入字典树,由于异或操作的特殊性，可以很快的根据题目需要得到一个贪心策略,只要按贪心策略向下查找即可.查询复杂度为树深。

1. **KD树**

kd树就是多维空间的区间划分树，可以求一个点跟其他点最近的那几个点。

建树复杂度是nlogn，利用nth\_element可以高效率的建立kd树。

默认情况，每一层按一维划分，如果需要有更好的效率，应该使用方差划分。

查询最近点，按这个点所处的位置直接递归，记录最优解，再回溯的过程中以最优解为半径画圆，如果和区间分解交割，那么就进入另一边子树内搜索，因此最坏复杂度仍然会变成暴力。

【静态KD树】（不支持删插）

前置数据:

int dv[4 \* MAXN],nowdv; //dv数组记录第i层以哪维划分，nowdv为辅助变量

kdtree tree[MAXN],kdans; //tree为kd树数组，ans为查询用

**struct** kdTree{    //KD树基本结构

**int** p[5];    //维度

**int** id;    //id, 辅助变量，可以不要

**double** dis;    //辅助变量，可以不要

};

**bool** kdCmp(kdTree a,kdTree b){    //kd树比较函数，用于建树

**return** a.p[nowdv] < b.p[nowdv];

}

**double** getDis(kdTree a,kdTree b){    //kd树计算距离

**double** re = 0;

**for**(**int** i = 0; i < m; ++i){

        re += (a.p[i] - b.p[i]) \* (a.p[i] - b.p[i]);

    }

    //re = sqrt(re);    //注意效率和精度问题

**return** re;

}

**int** getD(**int** l,**int** r){    //方差优化

**int** d = 0;

**double** now = -1;

**for**(**int** i = 0; i < m; ++i){    //修改维数大小

**double** ave = 0,sum = 0;

**for**(**int** j = l; j <= r; ++j){ ave += tree[j].p[i];}

        ave /= (r - l + 1) \* 1.0;

**for**(**int** j = l; j <= r; ++j){ sum += (tree[j].p[i] - ave) \* (tree[j].p[i] - ave);}

        sum /= (r - l + 1) \* 1.0;

**if**(sum > now){ now = sum; d = i;}

    }

**return** d;

}

**void** builtTree(**int** l,**int** r,**int** d){    //区间和按哪维划分（如果使用方差函数，d随意）

**if**(l > r){ **return**;}

    //d = getD(l,r);    //方差优化

    nowdv = d; dv[mid] = d;

    nth\_element(tree + l, tree + mid, tree + r + 1, kdCmp);

    builtTree(l,mid - 1,(d + 1) % m);

    builtTree(mid + 1,r,(d + 1) % m);

}

**void** searchTree(**int** l,**int** r,kdTree q){     //q为需要查询的坐标，下面代码需要改

**if**(l > r){ **return**;}

**double** nowd = getDis(tree[mid],q); tree[mid].dis = nowd;

**if**(nowd < kdans.dis && tree[mid].id != q.id){

        kdans.dis = nowd; kdans.id = tree[mid].id;

    }

**double** t = q.p[dv[mid]] - tree[mid].p[dv[mid]];

**if**(t < 0){

        searchTree(l,mid - 1,q);

//看看以该点作圆心是不是跟分界线相交，相交要到另一边,这里是因为不开根号所以t\*t

**if**(kdans.dis > t \* t){  //看看以该点作圆心是不是跟分界线相交，相交要到另一边

            searchTree(mid + 1,r,q);

        }

    }**else**{

        searchTree(mid + 1,r,q);

**if**(kdans.dis > t \* t){

            searchTree(l,mid - 1,q);

        }

    }

}

【动态KD树】（支持删插的KD树，数组要开大4倍）

**struct** kdTree{

**int** p[2],dv,id;

    ll dis;

**bool** leaf;

};

**void** builtTree(**int** now,**int** l,**int** r){

**if**(l > r){ tree[now].leaf = **true**; **return**;}

**int** d = getD(l,r); nowdv = d;

    nth\_element(a + l,a + mid,a + r + 1,kdcmp);

    tree[now] = a[mid]; tree[now].dv = d; tree[now].leaf = **false**;

    builtTree(left,l,mid - 1); builtTree(right,mid + 1,r);

}

**void** searchTree(**int** now,kdTree q){

**if**(tree[now].leaf){ **return**;}

    ll nowd = getDis(tree[now],q); tree[now].dis = nowd;

**if**(nowd < kdans.dis && tree[now].id != q.id){

        kdans.dis = nowd;

    }

    ll t = q.p[tree[now].dv] - tree[now].p[tree[now].dv];

**if**(t < 0){

        searchTree(left,q);

**if**(kdans.dis > t \* t){

            searchTree(right,q);

        }

    }**else**{

        searchTree(right,q);

**if**(kdans.dis > t \* t){

            searchTree(left,q);

        }

    }

}

【插入删除】

插入直接查这颗树到叶子，然后补上一维即可，至于按哪维划分随意。

但是不管你按哪维划分都无法使得树查询效率跟原来那么好，因此当插入大于一个阀值后，直接把数拍扁，暴力重建

1. **树链剖分（模板 洛谷P3384）**

动态的在树上对某个路径做一些操作（加减），并询问，或者是对整棵子树加减，可以使用树链剖分。

在序列上做这个操作很容易，但是在树上没办法做，实质上就是把树给分成了一条条连续的序列（链）然后简化成原来序列的问题了。

利用到了对于树的dfs序，一个节点的子树的dfs时间戳必定是一段连续的区间这个性质，来构造链。然而暴力dfs序来分链会有很多极端情况（类似于平衡二叉树要想办法平衡，KD树要想办法平均的划分）所以需要利用一些启发的信息来进行剖分，使得时间复杂度能不那么糟糕。所以树链剖分说白了就是一种特殊的dfs访问方法而已。

预处理有2步dfs，第一步求：

int fa[MAXN],sz[MAXN],hson[MAXN],dep[MAXN];

//节点父亲，子树大小，重儿子节点编号，当前节点深度

**void** tcpdfs1(**int** now){

**int** maxn = 0,maxnid = 0;

    dep[now] = dep[fa[now]] + 1;

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].to != fa[now]){

            fa[g[i].to] = now;

            tcpdfs1(g[i].to);

            sz[now] += sz[g[i].to];

**if**(sz[g[i].to] > maxn){

                maxn = sz[g[i].to]; maxnid = g[i].to;

            }

        }

    }

    ++sz[now]; hson[now] = maxnid;

}

第二步求：

int top[MAXN],id[MAXN],cid[MAXN],dlen;

//该链的链首是哪个节点，第i个点的dfs序（就是用dfs序重新编号节点），重新编号为i的节点是哪个节点，与id数组互逆，最后一个记录是dfs序号的辅助变量

**void** tcpdfs2(**int** now,**int** nowtop){

    id[now] = ++dlen; cid[dlen] = now; top[now] = nowtop;

**if**(hson[now] != 0){ tcpdfs2(hson[now],nowtop);}

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].to != fa[now] && g[i].to != hson[now]){

            tcpdfs2(g[i].to,g[i].to);

        }

    }

}

然后利用线段树（或者平衡树之类，维护重新编号的点就好了）

【基本操作：路径加减，路径和询问】

利用top数组，边向上爬，找lca，边对每条链在相应的线段树上区间加减区间求和就好了。

**void** tcpadd(**int** u,**int** v,**int** w){

    w %= p;

**while**(top[u] != top[v]){

**if**(dep[top[u]] < dep[top[v]]){ swap(u,v);}

        update(1,1,n,id[top[u]],id[u],w);

        u = fa[top[u]];

    }

**if**(dep[u] < dep[v]){ swap(u,v);}

    update(1,1,n,id[v],id[u],w);

}

**int** tcpans(**int** u,**int** v){

**int** re = 0;

**while**(top[u] != top[v]){

**if**(dep[top[u]] < dep[top[v]]){ swap(u,v);}

        re += getsum(1,1,n,id[top[u]],id[u]); re %= p;

        u = fa[top[u]];

    }

**if**(dep[u] < dep[v]){ swap(u,v);}

    re += getsum(1,1,n,id[v],id[u]); re %= p;

**return** re;

}

1. **树分治（模板poj 1741）**

大致思路就是把答案分成2部分，经过根和没经过根的，然后把经过根的用容斥什么的算出来。这个时候把树分成了好几部分（去掉根后，每个子树自己成了一棵树）每个子树的答案是独立的，而且跟子树的根是哪个点没有关系

这就使得我们可以让子树的重心当子树的根，继续递归，因为一直让重心当根的话，树就可以尽可能的平衡，是一种比较巧妙的暴力。

前置数据：链式前向星

vector<int> gp; //所有重心存在这里

int ts[MAXN]; //以i为根的子树大小

int td[MAXN],tdlen; //点分治辅助计算数组

bool tvis[MAXN]; //点分治记录已经算过答案的点,顺便分割这棵树

**int** gpdfs(**int** n,**int** fa,**int** now){ //求树的重心,n是子树大小

**int** maxs = 0; ts[now] = 0;

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].to != fa && !tvis[g[i].to]){

            ts[now] += gpdfs(n,now,g[i].to);

            maxs = max(maxs,ts[g[i].to]);

        }

    }

    ++ts[now];

    maxs = max(maxs,n - ts[now]);

**if**(maxs <= n / 2){

        gp.push\_back(now);

    }

**return** ts[now];

}

**void** getgp(**int** n,**int** st){ //求子树重心

    gp.clear(); gpdfs(n,-1,st);

}

**void** tdivdfs2(**int** fa,**int** now,**int** length){ //处理出所有经过根的路径

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].to != fa && !tvis[g[i].to]){

            tdivdfs2(now,g[i].to,length + g[i].w);

        }

    }

    td[++tdlen] = length;

}

**int** tdivdfs1(**int** fa,**int** now,**int** length){  //容斥原理

**int** re = 0;

**if**(fa == -1){

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(g[i].to != fa && !tvis[g[i].to]){

                re -= tdivdfs1(now,g[i].to,length + g[i].w);

            }

        }

    }

    tdlen = 0;

    tdivdfs2(fa,now,length);

    sort(td + 1,td + 1 + tdlen);

**for**(**int** i = 1; i <= tdlen; ++i){ //二分求答案,其实可以双指针优化到O(n)

**if**(td[i] >= k){ **break**;}

**int** pos = upper\_bound(td + i + 1,td + 1 + tdlen,k - td[i]) - td;

        re += pos - i - 1;

    }

**return** re;

}

**int** tdiv(**int** now){ //点分治

**int** re = 0;

    tvis[now] = **true**;

    re += tdivdfs1(-1,now,0);

**for**(**int** i = head[now]; i != -1; i = g[i].next){

**if**(!tvis[g[i].to]){

            getgp(ts[g[i].to],g[i].to); //第一遍dfs是找当前子树重心

**int** rt = gp[0];

            getgp(ts[g[i].to],rt); //这次dfs主要是处理出以重心为根后,这棵子树各个部分的大小(其实可以优化去掉这一句)

            re += tdiv(rt);

        }

    }

**return** re;

}

//主函数里先找一下最初的重心

**for**(**int** i = 1; i <= n; ++i){ tvis[i] = **false**;}

getgp(n,1); **int** rt = gp[0]; getgp(n,rt);

**int** ans = tdiv(rt);

1. **线性基（模板 洛谷P 3812）**

1.设线性基的异或集合中不存在0。

2.线性基的异或集合中每个元素的异或方案唯一，其实这个跟性质1是等价的。

3.线性基二进制最高位互不相同。

4.如果线性基是满的，它的异或集合为[1,2^(n−1)][1,2^(n−1)]。

5.线性基中元素互相异或，异或集合不变。

基础操作可以取若干个数异或最大值，最小值，第k大

第k大的话，先对线性基用性质3进行rebuild，如果j >i，且aj的第i位为1，那么aj异或上ai，然后从小到大把有值的线性基值给丢一个p数组里（p要0开始）。

然后根据k的二进制，如果k的那一位是1，那么就异或上那个p，最后答案就是了。

ll lb[70];

**void** lbinit(){

    memset(lb,0,**sizeof**(lb));

}

//插入一个x进入线性基

**bool** lbadd(ll x){

**for**(**int** i = 55; i >= 0; --i){

**if**((x & (1ll << i)) > 0){

**if**(lb[i] == 0){

                lb[i] = x;

**break**;

            }**else**{

                x = x ^ lb[i];

            }

        }

    }

**return** x > 0;

}

//取线性基最大值

//取最小值就是线性基最低位的值

ll getmax(){ //取线性基最大值

    ll re = 0;

**for**(**int** i = 55; i >= 0; --i){

**if**((lb[i] ^ re) > re){

            re = lb[i] ^ re;

        }

    }

**return** re;

}

1. **非几何数学**
2. **各种公式**

【对数换底公式】

https://gss2.bdstatic.com/9fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D100/sign=c4cd29ef3387e9504617f76c2039531b/77c6a7efce1b9d168f37bc4dfadeb48f8c54642d.jpg

【等比数列】

formula

formula

【判断闰年】

1、非整百年：能被4整除的为闰年。（如2004年就是闰年,2100年不是闰年）

2、整百年：能被400整除的是闰年。(如2000年是闰年，1900年不是闰年)

【取余和求模】

-7 Mod 4，求余是等于-3,求模是1，c/c++，java 中%为取余，而python则为取模

ll mod(ll num,ll k,ll &c){    //求模运算，c为商

**if**((num > 0 && k > 0) || (num < 0 && k < 0)){

        c = num / k; **return** num % k;

    }**else**{

        c = num / k - 1; **return** num % k + k;

    }

}

【求逆元和计算模意义下除法】

设：求a在模 p 意义下的逆元。

一、扩展欧几里得（要求gcd(a,p） = 1)

扩展欧几里得的x就是答案，但是要保证是最小正整数

return egcd(a,mod,x,y) == 1 ? ((x % mod) + mod) % mod : -1;

二、费马小定理 or 欧拉定理（要求p为素数）

逆元为a的p - 2次方，可以用快速幂求

三、通用公式（没要求，小心溢出）

ans = (a % (p \* b)) / b

四、线性递推（2 到 (p - 1）)

inv[1] = 1;

**for**(**int** i = 2; i <= n; ++i){

    inv[i] = (p - p / i) \* inv[p % i] % p;

}

【欧拉降幂】

如果指数过于庞大要使用欧拉降幂

利用扩展欧拉定理

ll phi = euler(m); b = 0;

**char** c; c = getchar(); **bool** flag = **false**;

**while**(c < '0' || c > '9'){ c = getchar();}

**while**(c >= '0' && c <= '9'){

    b \*= 10; b += c - '0';

**if**(b >= phi){ b %= phi; flag = **true**;}

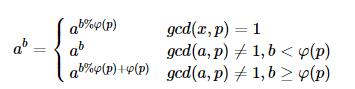
    c = getchar();

}

//printf("%c\n",c);

**if**(flag){ b += phi;}

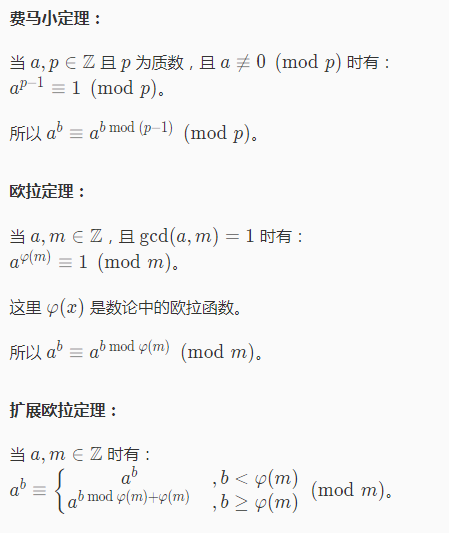
ll ans = quickpow(a,b);



【切比雪夫距离和曼哈顿距离转化】

将一个点(x,y)的坐标变为 \large (x+y ,x-y) 后,原坐标系中的曼哈顿距离 == 新坐标系中的切比雪夫距离

将一个点(x,y)的坐标变为  \large ( \frac{x+y}{2} ,\frac{x-y}{2})  后,原坐标系中的切比雪夫距离 == 新坐标系中的曼哈顿距离



【前缀和和差分（模板：HDU 6514）】

（一维前缀和）略

（一维差分）略

（二维前缀和）

pre[i][j] = pre[i][j - 1] + pre[i - 1][j] - pre[i - 1][j - 1] + a[i][j];

或：a[i][j] += a[i][j - 1] + a[i - 1][j] - a[i - 1][j - 1];

（二维差分）假设修改的范围左上角是（x,y）右下角是（xx,yy）

sub[x][y] += num; sub[x][yy + 1] -= num; sub[xx + 1][y] -= num; sub[xx + 1][yy + 1] +=num;

【约瑟夫环】

//a为出队顺序

**for**(**int** i = 1; i <= n; ++i){

**int** p = i \* m;

**while**(p > n){ p = p - n + (p - n - 1) / (m - 1);}

    a[p] = i;

**if**(i == 1){ pos = p;}

}

//最后一个谁出队

#include <stdio.h>

**int** main(**void**)

{

**int** n,m,i,s,k;

    //freopen("E://input.txt","r",stdin);

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

    {   //n人数,m间隔数,k起始点

        s=0;

**for** (i=2; i<=n-1; i++)

            s=(s+m)%i;

        printf ("%d/n", (s+k)%n+1);

    }

**return** 0 ;

}

1. **基础结构**

**struct** Matrix{   //矩阵类

**int** n,m;

**int** data[MAXN][MAXN];

};

1. **基础函数**
2. 【快速幂和快速乘法】

快速乘法好像网上又叫龟速乘？算了，叫啥没啥区别。

ll quickPow(ll n,ll m){  //快速幂

    ll re = 1;

**while**(m != 0){

**if**((m&1) == 1){re = re \* n;}

        n = n \* n; m = m >> 1;

    }

**return** re;

}

ll quickMul(ll a,ll b){   //快速乘法,主要可以用来防溢出

    ll re = 0;

**while**(b != 0){

**if**((b & 1) == 1){

            re += a;

        }

        a = a << 1;

        b = b >> 1;

    }

**return** re;

}

1. 【排列数和组合数】

ll A(ll n, ll m){   //排列数

**if**(m > n)**return** 0;

    ll re = 1,i;

**for**(i = 1; i <= m; ++i){

        re \*= n-i+1;

    }

**return** re;

}

ll C(ll n, ll m){   //组合数

**if**(m > n)**return** 0;

    ll re = 1, i;

**if**(m>(n-m))m = n - m;

**for**(i = 1; i <= m; ++i){

        re = re\*(i+(n-m))/i;

    }

**return** re;

}

1. 【最小公约数和公倍数】

**int** gcd(**int** n,**int** m){  //求n和m的最大公约数

**return** n%m?gcd(m,n%m):m;

}

前置函数【gcd】

**int** lcm(**int** n,**int** m){  //求n和m的最小公倍数

**return** n/gcd(n,m)\*m;

}

1. 【矩阵乘法和矩阵快速幂】

例子：f(n)=a\*f(n-1)+b\*f(n-2)+c；（a,b,c是常数） 前面矩阵记mb后面为ma

**https://img-blog.csdn.net/20160728212613431?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center**

下面模板是要把ma转置一下,然后ma矩乘mb的n次

前置结构【Matrix】

Matrix MatrixMul(Matrix a, Matrix b){  //矩阵乘法

    Matrix re;

**if**(a.m != b.n){

        printf("error\n");

**return** re;

    }

    memset(re.data,0,**sizeof**(re.data));

    re.n = a.n; re.m = b.m;

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

**for**(**int** j = 1; j <= a.m; ++j){

**if**(a.data[i][j] == 0){

**continue**;

            }

**for**(**int** k = 1; k <= b.m; ++k){

                re.data[i][k] += (a.data[i][j] % MOD \* b.data[j][k] % MOD) % MOD;

                re.data[i][k] %= MOD;

            }

        }

    }

**return** re;

}

前置结构【Matrix】

前置函数【matMul】

Matrix MatrixPow(Matrix a,**int** b){

    Matrix re;

**if**(a.n != a.m){

        printf("error2\n"); **return** re;

    }

    re.n = re.m = a.n; memset(re.data,0,**sizeof**(re.data));

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

        re.data[i][i] = 1;

    }

**while**(b != 0){

**if**((b & 1) == 1){ re = MatrixMul(re,a); }

        a = MatrixMul(a,a); b = b >> 1;

    }

**return** re;

}

//矩阵快速幂方便输入

**void** inputMat(**int** n,**int** m,Matrix &a,ll \*b){

    a.n = n; a.m = m;

**for**(**int** i = 1; i <= n; ++i){

**for**(**int** j = 1; j <= m; ++j){

            a.data[i][j] = \*(b + (i - 1) \* m + (j - 1));

        }

    }

}

**void** init(){

    ll pt[1][7] = {b,a,16,8,4,2,1};

    inputMat(1,7,ma,\*pt);

    ll pt2[7][7] = {1,1,0,0,0,0,0,

                    2,0,0,0,0,0,0,

                    1,0,1,0,0,0,0,

                    4,0,4,1,0,0,0,

                    6,0,6,3,1,0,0,

                    4,0,4,3,2,1,0,

                    1,0,1,1,1,1,1,};

    inputMat(7,7,mb,\*pt2);

}

1. 【求1-n有多少数字和k不互素】

（n – getNum(n)就是该范围内与k互素的数字个数）

前置数据：vector<ll> sushu; //存放k的所有素因子

ll getNum(ll n){    //求1-n有多少数字和k不互素（容斥原理）

    ll len = 1 << sushu.size();

    ll re = 0;

**if**(n == 0){ **return** 0;}

**for**(**int** i = 1; i < len; ++i){

        ll sum = 0;

        ll now = 1;

**for**(**int** j = 0; j < sushu.size(); ++j){

**if**((1<<(j))&i){

                ++sum; now \*= sushu[j];

            }

        }

        sum&1?re+=n/now:re-=n/now;

    }

**return** re;

}

1. 【基姆拉尔森公式】

/\*基姆拉尔森计算公式：W= (d+2\*m+3\*(m+1)/5+y+y/4-y/100+y/400)%7

在公式中d表示日期中的日数，m表示月份数，y表示年数。

注意：在公式中有个与其他公式不同的地方：

把一月和二月看成是上一年的十三月和十四月，例：如果是2004-1-10则换算成：2003-13-10来代入公式计算。

并且这个公式中的d表示日期+1，w就直接是周几（0 - 6）。\*/

**int** getDate(**int** y,**int** m,**int** d){

**if**(m == 1 || m == 2){

        m += 12; y--;

    }

**return** (d + 1 + 2 \* m + 3 \* (m + 1) / 5 + y + y / 4 - y / 100 + y / 400) % 7;

}

1. 【求单个欧拉函数】

**int** euler(**int** x){   //求一个数的欧拉函数值

**int** re,a,i;

    re = x; a = x; //初始化

**for**(i = 2; i \* i <= a; ++i){

**if**(a % i == 0){ //如果是a的一个因子

            re = re / i \* (i - 1); //根据定义更新答案

        }

**while**(a % i == 0){ //让a除去这个所有的i因子

            a /= i;

        }

    }

**if**(a > 1){ //如果剩下的仍然是这个因子

        re = re / a \* (a-1); //继续更新

    }

**return** re; //返回答案

}

1. 【素数，欧拉函数，莫比乌斯函数】

前置数据 vector<int> pnum; //存储所有素数

bool sushu[MAXN]; //标记第i个数是不是素数

phi[MAXN]; //标记第i个数的欧拉函数

**void** prime(**bool** sushu[],**int** n){

**for**(**int** i = 2; i <= n; ++i){

        //mu[i] = 0;

        sushu[i] = **true**;

    }

    sushu[1] = **false**; pnum.clear();

    //phi[1] = 1;

    //mu[1] = 1;

**for**(**int** i = 2; i <= n; ++i){

**if**(sushu[i]){

            pnum.push\_back(i);

            //phi[i] = i - 1;

            //mu[i] = -1;

        }

**for**(**int** j = 0; j < pnum.size(),pnum[j] \* i <= n; ++j){

**int** now = pnum[j] \* i;

            sushu[now] = **false**;

**if**(i % pnum[j] == 0){

                //phi[now] = phi[i] \* pnum[j];

**break**;

            }**else**{

                //phi[now] = phi[i] \* (pnum[j] - 1);

                //mu[now] = -mu[i];

            }

        }

    }

**return**;

}

欧拉公式的延伸：一个数的与其互质的数(<n)的总和是euler(n)\*n/2

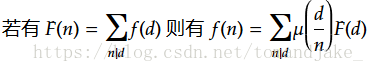
当n为奇数时，有φ（2\*n） = φ（n）。

当n为偶数时，有φ（2\*n） = 2φ（n）

莫比乌斯函数：https://img-blog.csdn.net/2018071823001066?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3RvbWFuZGpha2Vf/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

莫比乌斯反演（2种形式）：

F(n)=\sum_{d|n}f(d) f(n)=\sum_{d|n}\mu(d)F(\frac{n}{d})



poj3904：求存在几个四元组的gcd为1

F（n）为四元组gcd为n的倍数的个数，f（n）为四元组gcd为n的个数

1. **较复杂算法**
2. 【Berlekamp\_Massey】

const double EPS = 1e-8;

int bm[10] = {0,1,3,7,15,31,63};

int bmlen;

double bmAns[10],sup[10],last[10]; //sup和last为辅助数组

（有问题，尽量先别用，要用尽量给超过10项）

（不取模版本）

**void** Berlekamp\_Massey(**int** n){

**int** fail = 1,suplen,lastlen;

**double** mul,d,lastd;

    bmlen = 0; suplen = 0; lastlen = 0;

**for**(**int** i = 1; i <= n; ++i){

**if**(i == 1){

            bmAns[++bmlen] = 0; lastd = bm[i];

**continue**;

        }

**double** now = 0;

**for**(**int** j = 1; j <= bmlen; ++j){ now += bm[i - j] \* bmAns[j];}

        d = bm[i] - now;

**if**(fabs(d) < EPS){ **continue**;}

        mul = 1.0 \* d / lastd; suplen = i - fail + lastlen;

**for**(**int** j = 1; j <= i - fail - 1; ++j){ sup[j] = 0;}

        sup[i - fail] = mul;

**for**(**int** j = i - fail + 1; j <= suplen; ++j){ sup[j] = -mul \* last[j - i + fail];}

**for**(**int** j = 1; j <= bmlen; ++j){ last[j] = bmAns[j]; lastlen = bmlen;}

**for**(**int** j = 1; j <= suplen; ++j){ bmAns[j] += sup[j];} bmlen = suplen;

        fail = i; lastd = d;

    }

}

（对素数取模版本，不存在负数）

求可能存在的线性递推式，可以用来打表找规律之类的，复杂度O（n^2）

//对素数取模数版本

**const** **int** MOD = 1e9 + 7;

ll bm[10] = {0,1,3,7,15,31,63};

ll bmlen;

ll bmAns[10],sup[10],last[10];    //sup和last为辅助数组

//bmAns为递推式，bmlen为递推式长度，n为数列长度

ll quickPow(ll n,ll m){  //快速幂

    ll re = 1;

**while**(m != 0){

**if**((m & 1) == 1){re = re \* n; re %= MOD;}

        n = n \* n; m = m >> 1; n %= MOD;

    }

**return** re % MOD;

}

**void** Berlekamp\_Massey(**int** n){

    ll fail = 1,suplen,lastlen;

    ll mul,d,lastd;

    bmlen = 0; suplen = 0; lastlen = 0;

**for**(**int** i = 1; i <= n; ++i){

**if**(i == 1){

            bmAns[++bmlen] = 0; lastd = bm[i];

**continue**;

        }

        ll now = 0;

**for**(**int** j = 1; j <= bmlen; ++j){ now += (bm[i - j] \* bmAns[j]) % MOD; now %= MOD;}

        d = bm[i] - now; **if**(d < 0){ d += MOD;}

**if**(d == 0){ **continue**;}

        mul = (d \* quickPow(lastd,MOD - 2)) % MOD; suplen = i - fail + lastlen;

**for**(**int** j = 1; j <= i - fail - 1; ++j){ sup[j] = 0;}

        sup[i - fail] = mul;

**for**(**int** j = i - fail + 1; j <= suplen; ++j){

            sup[j] = (-mul \* last[j - i + fail]) % MOD;

**if**(sup[j] < 0){sup[j] += MOD;}    //如果不允许出现负数就用这个

        }

**for**(**int** j = 1; j <= bmlen; ++j){ last[j] = bmAns[j]; lastlen = bmlen;}

**for**(**int** j = 1; j <= suplen; ++j){ bmAns[j] = (bmAns[j] + sup[j]) % MOD;} bmlen = suplen;

        fail = i; lastd = d;

    }

}

1. 【扩展欧几里得算法】

可以用来求取 e \* d ≡ 1 （mod  p） 同余方程

上式等价与 ed + py = 1

此式中的y是我们不需要的.

x0和y0为方程ax + by = gcd(a,b)的一组特解,返回值为gcd(a,b);

通解为

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

y = y0 - (a / gcd) \* t

**int** egcd(**int** a,**int** b,**int** &x,**int** &y){  //扩展欧几里得算法，x和y为x0，y0

**if**(b == 0){

        x = 1; y = 0;

**return** a;

    }

**int** re = egcd(b,a%b,x,y);

**int** t = x; x = y; y = t - a/b\*y;

**return** re;

}

1. 【FFT快速傅里叶变换 （模板 洛谷P3803）】

快速求2个多项式的乘法。

多项式由系数表示法转为点值表示法的过程，就成为DFT；

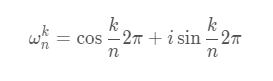
相对地，把一个多项式的点值表示法转化为系数表示法的过程，就是IDFT。

原理：利用分治O（logn）快速解一个多项式

\\G(x)= a_{0}+ a_{2}x +a_{4}x^2+ a_{6}x^3 \\ H(x) = a_{1}+ a_{3}x+ a_{5}x^2+ a_{7}x^3\\

G（x）为偶次，H（x）为奇数次

f(x) = G(x^2) + x*H(x^2)



代入k = 1就是单位复根

先求答案的长度小于的最近的2的幂次（长度包括0次项）

再把系数存a和b里（记得补长），然后分别做DFT。

然后c为a乘以b

最后对c做IDFT，实部为答案，注意对精度处理一下再用。

前置数据：complex<double> a[4 \* MAXN],b[4 \* MAXN],c[4 \* MAXN];

**void** fft\_change(complex<**double**> y[],**int** len){ //颠倒二进制

**int** i,j,k; j = len / 2;

**for**(**int** i = 1; i < len - 1; ++i){

**if**(i < j){ swap(y[i],y[j]);}

        k = len / 2;

**while**(j >= k){

            j -= k;

            k >>= 1;

        }

        j += k;

    }

}

//flag为1时是DFT,-1是IDFT

//len必须是2的幂次

**void** fft(complex<**double**> y[],**int** len,**int** flag){

    fft\_change(y,len);

**for**(**int** s = 2; s <= len; s \*= 2){ //合并长度

        complex<**double**> wn = {cos(flag \* 2 \* PI / s),sin(flag \* 2 \* PI / s)};

**for**(**int** j = 0; j < len; j += s){  //蝶形运算

            complex<**double**> w = {1,0};

**for**(**int** k = j; k < j + s / 2; ++k){

                complex<**double**> a = y[k];

                complex<**double**> b = w \* y[k + s / 2];

                y[k] = a + b;

                y[k + s / 2] = a - b;

                w \*= wn;

            }

        }

    }

**if**(flag == -1){

**for**(**int** i = 0; i < len; ++i){

            y[i].real(y[i].real() / (1.0 \* len));

        }

    }

}

**int** main(){

    //n,m为自多项式的长度,包括0次项,anslen为答案长度

    scanf("%d%d",&n,&m); anslen = 1;

    ++n; ++m;

**while**(anslen <= n + m - 1){ //求最小的2的幂次使得其比答案的最长长度要长（可改）

        anslen \*= 2;

    }

**for**(**int** i = 0; i < n; ++i){

**double** num; scanf("%lf",&num);

        a[i].real(num); a[i].imag(0);

    }

**for**(**int** i = n; i < anslen; ++i){ a[i].real(0),a[i].imag(0);} //多余位补0

**for**(**int** i = 0; i < m; ++i){

**double** num; scanf("%lf",&num);

        b[i].real(num); b[i].imag(0);

    }

**for**(**int** i = m; i < anslen; ++i){ b[i].real(0); b[i].imag(0);} //多余位补0

    fft(a,anslen,1); fft(b,anslen,1);

**for**(**int** i = 0; i < anslen; ++i){

        c[i] = a[i] \* b[i];

    }

    fft(c,anslen,-1);

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

        //系数非整数时记得改精度

**int** p = (**int**)(c[i].real() + 0.5);

        printf("%d ",p);

    }

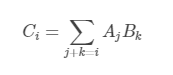
    printf("\n");

**return** 0;

}

1. 【NTT快速数论变换 】
2. 【FWT快速沃尔什变换（模板 洛谷P4717）】

多项式乘法是快速求这个（FFT解决这个问题）



但是如果现在指数不是相加，而是进行位运算，就要用FWT



应用是类似母函数问题上，比如集合A集合B，2个分别取一个异或的答案集合C，直接暴力枚举是平方级复杂度，可以套FWT降低到nlogn

原理类似FFT，只是分治的时候按前后分半（所以长度也要预处理成2的幂次）

不同运算的公式不同的，直接抄公式就好了。

前置数据:y[MAXN]; //系数数组

//快速沃尔什变换

//ope 1:and 2:or 3:xor

//flag:1是正变换，-1是负变换

**inline** **void** fwt(**int** y[],**int** len,**int** flag,**int** ope){

**for**(**int** s = 2; s <= len; s \*= 2){ //合并长度

**for**(**int** j = 0; j < len; j += s){  //蝶形运算

**for**(**int** k = j; k < j + s / 2; ++k){

**int** a = y[k];

**int** b = y[k + s / 2];

**if**(flag == 1){

**switch**(ope){

**case** 1: y[k] += b; md(y[k]); **break**;

**case** 2: y[k + s / 2] += a; md(y[k + s / 2]); **break**;

**case** 3: y[k] = a + b; y[k + s / 2] = a - b; md(y[k]); md(y[k + s / 2]); **break**;

                    }

                }**else**{

**switch**(ope){

**case** 1: y[k] -= b; md(y[k]); **break**;

**case** 2: y[k + s / 2] -= a; md(y[k + s / 2]); **break**;

**case** 3: y[k] = a + b; md(y[k]);

                                y[k] = (1ll \* y[k] \* INV) % MOD;

                                y[k + s / 2] = (a - b);  md(y[k + s / 2]);

                                y[k + s / 2] = (1ll \* y[k + s / 2] \* INV) % MOD; **break**;

                        //\*INV是2在模p意义下的逆元，如果不需要取余可以直接/2

                    }

                }

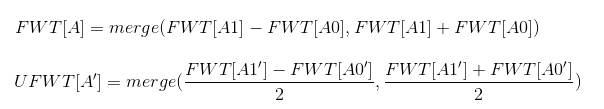
            }

        }

    }

}

同或运算的FWT：



1. **博弈问题**
2. 【巴什博弈(Bash Game)】

只有一堆n个物品，两个人从轮流中取出（1~m）个；最后取光者胜。

解：当前n = k \* (m + 1)时必败

//从n个里面取,每次可以取1-m个,先手必胜。

**bool** bash\_game(**int** n,**int** m){

**return** (n % (m + 1) != 0);

}

1. 【尼姆博弈(Nimm Game)】

有n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

解：如果把n堆抽象为n个非负整数,再将n个整数转化为二进制,然后对n个二进制数按位相加(不进位，就是异或),若每一位相加都为偶数,

那么称这个状态为偶状态 , 否则称它为奇状态.

偶状态必败。

前置数据：f[MAXN]; //每堆物品的数量

//从n堆物品里,每次取1 - 任意件,取完的胜利

**bool** nimm\_game(**int** n){

**int** flag = 0;

**for**(**int** i = 1; i <= n; ++i)

    flag ^= f[i];

**return** flag > 0;

}

1. 【威佐夫博奕（Wythoff Game）】

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。

**bool** wythoff\_game(**int** n){

**if**(a > b){ swap(a,b);}

**double** k = (sqrt(5.0) - 1.0) / 2.0;

**int** j = a \* k;

**if**(a != j \* (**int**)(k + 1)){

        j++;

**if**(a+j==b){

**return** **false**;

        }**else**{

**return** **true**;

        }

    }

}

1. 【Sprague-Grundy定理（SG定理）】

必胜点和必败点的概念：

P点：必败点，换而言之，就是谁处于此位置，则在双方操作正确的情况下必败。

N点：必胜点，处于此情况下，双方操作均正确的情况下必胜。

必胜点和必败点的性质：

1、所有终结点是 必败点 P 。（我们以此为基本前提进行推理，换句话说，我们以此为假设）

2、从任何必胜点N 操作，至少有一种方式可以进入必败点 P。

3、无论如何操作，必败点P 都只能进入 必胜点 N。

SG函数：

首先定义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。

对于任意状态 x ， 定义 SG(x) = mex(S),其中 S 是 x 后继状态的SG函数值的集合。如 x 有三个后继状态分别为 SG(a),SG(b),SG(c)，那么SG(x) = mex{SG(a),SG(b),SG(c)}。 这样 集合S 的终态必然是空集，所以SG函数的终态为 SG(x) = 0,当且仅当 x 为必败点P时。

**【实例】取石子问题**

有1堆n个的石子，每次只能取{ 1, 3, 4 }个石子，先取完石子者胜利，那么各个数的SG值为多少？

SG[0]=0，f[]={1,3,4},

x=1 时，可以取走1 - f{1}个石子，剩余{0}个，所以 SG[1] = mex{ SG[0] }= mex{0} = 1;

x=2 时，可以取走2 - f{1}个石子，剩余{1}个，所以 SG[2] = mex{ SG[1] }= mex{1} = 0;

x=3 时，可以取走3 - f{1,3}个石子，剩余{2,0}个，所以 SG[3] = mex{SG[2],SG[0]} = mex{0,0} =1;

x=4 时，可以取走4-  f{1,3,4}个石子，剩余{3,1,0}个，所以 SG[4] = mex{SG[3],SG[1],SG[0]} = mex{1,1,0} = 2;

x=5 时，可以取走5 - f{1,3,4}个石子，剩余{4,2,1}个，所以SG[5] = mex{SG[4],SG[2],SG[1]} =mex{2,0,1} = 3;

1. **组合数学**
2. 【经典数列或者公式】

【有无限重复元素的排列问题】

字母总数!/(各个相同字母的数量!)

11!/1! \* 4! \* 4! \* 2!

【有无限重复元素的组合问题】

前置函数：C

ll Cr(ll n, ll m){  //n个元素取m件组合(m可以大于N)

**return** C(n+m-1,m);

}

【环排列问题】

ll cp(ll n,ll m){   //n取m个进行环排列

**return** A(n,m)/m;

}

【环形染色问题】

前置函数：quickPow

ll cr(ll n,ll m){   //环形染色问题,n块,m色

**return** n%2==0?(quickPow((m-1),n) + (m-1)):(quickPow((m-1),n) - (m-1));

}

【全错位排序】

**void** cuopai(ll a[],ll n){   //全错位排列

    ll i;

    a[1] = 0; a[2] = 1; a[3] = 2;

**for**(i = 4; i <= n; ++i){

        a[i] = (i-1)\*(a[i-1]+a[i-2]);

    }

}

【卡特兰数列】

数列前n项：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 1 | 2 | 5 | 14 | 42 | 132 | 429 | 1430 | 4862 |

令h(0)=1,h(1)=1，catalan数满足递推式

h(n)= h(0)\*h(n-1)+h(1)\*h(n-2) + ... + h(n-1)h(0) (n>=2)

另类递推式：

h(n)=h(n-1)\*(4\*n-2)/(n+1);

ll catalan(ll a[],ll n){   //生成卡特兰数列

    a[0] = 1; a[1] = 1; a[2] = 2;

**for**(ll i = 3; i <= n; ++i){

        a[i] = a[i-1] \* (4\*i-2)/(i+1);

    }

}

【斐波那契数列】

略

【Bell数】

B(n)是包含n个元素的集合的划分方法的数目

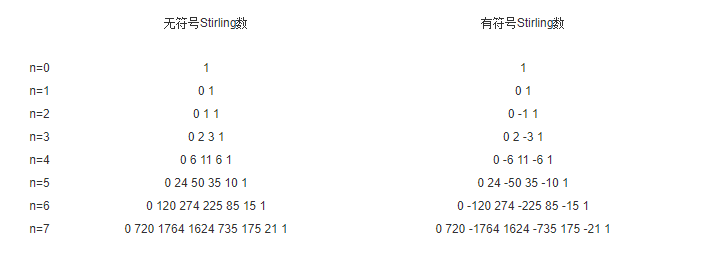
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | … |  |  |
| 1 | 1 | 2 | 5 | 15 | 52 | 203 | … |  |  |

递推公式为，  
B(0) = 1,  
B(n+1) = Sum(0,n) C(n,k)B(k). n = 1,2,...  
其中，Sum(0,n)表示对k从0到n求和，C(n,k) = n!/[k!(n-k)!]

【Stirling数】

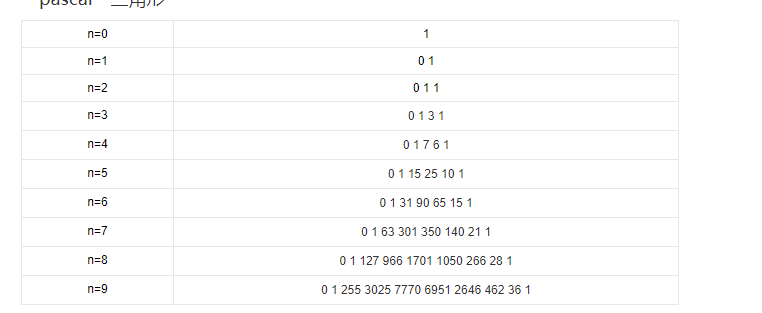
第一类Stirling数是有正负的，其绝对值是包含n个元素的集合分作k个环排列的方法数目。

递推公式为， S(n,0) = 0, S(1,1) = 1. S(n+1,k) = S(n,k-1) + nS(n,k)。



第二类Stirling数是把包含n个元素的集合划分为正好k个非空子集的方法的数目。

递推公式为， S(n,n) = S(n,1) = 1, S(n,k) = S(n-1,k-1) + kS(n-1,k).



【鸽笼原理】 n个物品放进n-1个盒子内,必有一个盒子内物品数目大于1.

【强化鸽笼原理】p1 p2 p3….pn个物品放进n个盒子内,必有盒子ki内物品大于pi

等价于【平均原理】

【ramsey定理】6个点,一定存在3个点互相连通或者3个点互相不连通（可推广）

1. 【生成函数】

可以求很多组合问题，可以利用FFT优化计算复杂度

【普通型函数】

https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D291/sign=ed15efd901b30f24319aeb0af994d192/b21c8701a18b87d6b1e3efa7090828381e30fde1.jpg

例：有重量为1,3,5（克）的砝码个两个，问：

(1)可以称出多少种不同重量的物品？

(2)若要称出重量为7克的物品，所使用的法码有多少种本质不同的情况？

第一问：

设（普通型）生成函数

G(x)=(1+x+x^2)(1+x^3+x^6)(1+x^5+x^{10})

其中，1 表示不用重量为 1克的砝码，  x表示使用 1个重量为 1克的砝码，....以此类推、

展开为

G(x)=1+x+x^2+x^3+x^4+2x^5+2x^6+2x^7+...+x^{18}

因此，可以称出19种不同重量的物品

第二问：

由第一问的展开式可知答案为 2

1. **其他问题**
2. 【n的阶乘里有几个因子2】

N!所含质因子2的个数等于N减去N的二进制表示中1的数目。

【九余数定理】：

一个数的各位数字之和相加后得到的<10的数字称为这个数的九余数（如果相加结果大于9，则继续各位相加） PS: 如果余0要改成9.

1. 【因数的个数是偶数还是奇数？】

如果一个数是完全平方数

则它的因数的个数是奇数

而如果他不是完全平方数

则它的因数的个数是偶数

（完全平方即用一个整数乘以自己例如1\*1，2\*2，3\*3等，依此类推）

1. 【输油管道问题和糖果传递问题】

坐标轴上有N个点，取一个点，使得该点与所有点连线的绝对值的总和最小。这个点应该取中位数。

糖果传递是纸牌均分的首尾相接循环版本。设ave为平均数，ai为每个小朋友拥有的糖果数量，求出ci，c1 = 0 && ci = c(i - 1) + ai - ave，然后ci就变成输油管道问题，取ci的中位数，然后其他所有值跟中位数求绝对值即可。

1. 【平面上的欧拉公式】

平面上的欧拉公式：V−E+R=C+1，其中 V(vertex) 表示交点数目，E(edge) 表示边数，R(region) 表示区域数，C(connection) 用来分割的线所构成的连通块个数

1. **字符串**
2. **基础函数**

1）字符操作  
tolower(ch)将字符ch转为小写

toupper(ch)将字符ch转为大学

2）字符串操作

strcpy(p, p1) 复制p1字符串到p

strncpy(p, p1, n) 复制p1指定长度字符串到p

strcat(p, p1) 将p1字符串接在p之后

strncat(p, p1, n) 将p1字符串指定长度字符串接在p之后

strlen(p) 取字符串p的长度

strcmp(p, p1) 比较字符串按ASCII码表

当p1<p2时，返回值<0

当p1=p2时，返回值=0

当p1>p2时，返回值>0

strcasecmp(p, p1)忽略大小写比较字符串

strncmp(p, p1, n) 比较指定长度字符串

strchr(p, c) 在字符串p中查找指定字符c

strrchr(p, c) 在字符串p中反向查找字符c

strstr(p, p1) 从字符串p中寻找p1第一次出现的位置

2）字符串到数值类型的转换

atoi(p) 字符串转换到 int 整型

atof(p) 字符串转换到 double 符点数

atol(p) 字符串转换到 long 整型

itoa(i ,num ,10 );

i ---- 需要转换成字符串的数字

num ---- 转换后保存字符串的变量

10 ---- 转换数字的基数（即进制）。10就是说按10进制转换数字。还可以是2，8，16等等你喜欢的进制类型

返回值：指向num这个字符串的指针

【itoa Linux下不支持】

3）字符检查

isalpha() 检查是否为字母字符

isupper() 检查是否为大写字母字符

islower() 检查是否为小写字母字符

isdigit() 检查是否为数字

isxdigit() 检查是否为十六进制数字表示的有效字符

isspace() 检查是否为空格类型字符

iscntrl() 检查是否为控制字符

ispunct() 检查是否为标点符号

isalnum() 检查是否为字母和数字

isprint() 检查是否是可打印字符

isgraph() 检查是否是图形字符，等效于 isalnum() | ispunct()

1. **字符串匹配**

匹配串：待匹配的母串

模式串：带从匹配串里查找的串

1. 【kmp算法】

O（n + m）

维护一个next数组，在匹配串里找第一个模式串

前置数据：kmpNext【MAXN】 //模板串预处理用数组

**void** kmpInit(**char** \*str,**int** len){   //对模板串进行预处理

    kmpNext[0] = -1; kmpNext[1] = 0;

**int** p = 0;

**for**(**int** i = 2; i < len; ++i){

**if**(str[p] == str[i - 1]){

            ++p;

        }**else**{

            kmpNext[p] == -1 ? p = 0 : p = kmpNext[p];

**if**(str[p] == str[i - 1]){

                ++p;

            }

        }

**if**(str[p] == str[i]){

            kmpNext[i] = kmpNext[p];

        }**else**{

            kmpNext[i] = p;

        }

    }

**return**;

}

**int** kmp(**char** \*a,**char** \*b){  //kmp算法,a是匹配串,b是模式串,返回首符号位,-1表示未能匹配到.

**int** re = -1,now = 0,st = 0;

**int** lena = strlen(a);

**int** lenb = strlen(b);

**while**(lena - now >= lenb){

**bool** flag = **true**;

**for**(**int** i = st; i < lenb; ++i){

**if**(a[now+i] != b[i]){

                flag = **false**;

                now += i - kmpNext[i];

                kmpNext[i] == -1 ? st = 0 : st = kmpNext[i];

**break**;

            }

        }

**if**(flag){

**return** now;

        }

    }

**return** re;

}

1. 【扩展KMP算法】

O（n + m）

kmp算法的修改版本，多维护了一个数组，可以求模板串和匹配串的每个子串的最长公共前缀，把b丢进预处理里面，如果a和b相等就随便丢哪个都一样

前置数据：

int exKmpNext[MAXN],extend[MAXN];

//extend[i]为b+i 与 a的最长公共前缀长度

char a[MAXN],b[MAXN];

**void** exKmpInit(**char** \*str){   //扩展kmp算法预处理

**int** len = strlen(str); exKmpNext[0] = len;

**int** i = 1;

**while**(str[i - 1] == str[i] && i < len){ ++i;}

    exKmpNext[1] = i - 1;

**int** st = 1;

**int** ed = st + exKmpNext[st] - 1;

**for**(**int** i = 2; i < len; ++i){

**if**(exKmpNext[i - st] + i - 1 < ed){

            exKmpNext[i] = exKmpNext[i - st];

        }**else**{

**int** num = ed - i + 1;   //预先已知匹配数

**int** pos = ed;   //已结匹配的位数

**if**(i > ed){    //已经离开范围,只能

                num = 0; pos = i - 1;

            }

**while**(pos < len - 1 && str[num] == str[pos + 1]){    //暴力匹配

                ++pos; ++num;

            }

            exKmpNext[i] = num;

**if**(pos > ed){

                st = i; ed = pos;

            }

        }

    }

}

**void** exkmp(**char** \*a,**char** \*b){  //扩展KMP主算法,根据next数组求出extend数组,a为匹配串,b为模式串

**int** lena = strlen(a); **int** lenb = strlen(b);

**int** i = 0;

**while**(a[i] == b[i] && i < lenb){ ++i;}

    extend[0] = i;

**int** st = 0; **int** ed = i - 1;

**for**(**int** i = 1; i < lena; ++i){

**if**(exKmpNext[i - st] + i - 1 < ed){

            extend[i] = exKmpNext[i - st];

        }**else**{

**int** num = ed - i + 1; **int** pos = ed;

**if**(i > ed){ num = 0; pos = i - 1;}

**while**(pos < lena - 1 && b[num] == a[pos + 1]){    //暴力匹配

                ++pos; ++num;

            }

            extend[i] = num;

**if**(pos > ed){

                st = i; ed = pos;

            }

        }

    }

}

1. 【Shift-And / Shift-Or算法】

O（n \* m）应该是这样，但是一般不会到最差情况

如果匹配串是第i个位置可以是某某某字符（好几个），这样子的匹配，可以用这个算法。

前置数据：

bitset<MAXN> b[20],d;

//MAXN表示模式串长度，b数组大小为出现的字符的范围，如英文字母为26，d为辅助数组

（假设范围是数字0 - 9）

初始化：b[1] = 00000101 表示模式串第0位和第3位是数字1

D = 00000101 表示当前下，前1个字符和前3个字符，既是模式串前缀又是匹配串后缀

**void** shiftAnd(){

**for**(**int** i = 0; i < len; ++i){

        d <<= 1; d[0] = **true**; d &= b[shiftStr[i]];

    }

}

1. **回文串**
2. 【manacher算法】

O（n）

求出字符串内所有回文串以及最长回文串。

首先在字符串0位插一个防越界字符如$

然后在字符串每个字符两侧插入#，使得字符串变成奇数长只有奇数串，总长变为2倍。

前置数据：

char b[2 \* MAXN]; //存放处理后用于算法的字符串。

int p[2 \* MAXN]; //算法辅助数组，p[i]表示预处理后的字符串，以第i位（包括第i位，所以至少为1）为半径的最长对称区间。

**int** manacher(**char** \*a,**int** lena){    //a为字符串，lena为字符串长度，返回最长回文，为0表示为空字符串。

**int** re = 0,mx = 0,md = 0;

**char** last = a[lena]; a[lena] = '\0'; //防止输入的字符串末尾非'\0'而造成越界

**for**(**int** i = 1; i < lena; ++i){

        p[i] = 1;

**if**(i < mx){

            p[i] = min(mx - i,p[md - i + md]);

        }

**int** s = p[i];

**while**(a[i - s] == a[i + s]){

            ++s;

        }

        p[i] = s; md = i; mx = md + p[i] - 1;

        re = max(re,p[i] - 1);

    }

    a[lena] = last;

**return** re;

}

【回文串个数】

对所有的p[i] / 2求和即可。

1. **计算几何**
2. **常用参数和公式**

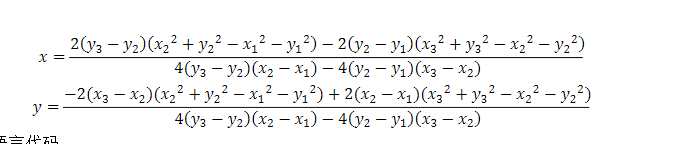
**const** **double** PI = acos(-1);  //圆周率

**const** **double** EPS = 0.000001; //精度

**const** **double** E = 2.718281828459;

**const** **double** ln(2) = 0.69314718055995;

【已知三点求三角形外心（外接圆圆心）】



【三角形】

https://gss3.bdstatic.com/-Po3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D217/sign=44dec7aa7e310a55c024d9f580444387/6f061d950a7b0208c81c62b865d9f2d3562cc896.jpg**正弦定理**

一个三角形中，各边和所对角的正弦之比相等，且该比值等于该三角形外接圆的直径（半径的2倍）长度。

https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D135/sign=9f63b465fb03738dda4a0821861ab073/908fa0ec08fa513d00f5a71b306d55fbb3fbd99c.jpg**余弦定理**

另外几种类似

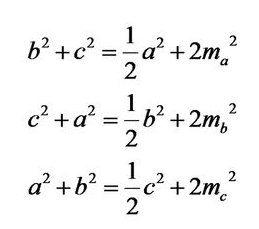
formulaformula**三角形面积**

后者为海伦公式，p为半周长

https://gss2.bdstatic.com/9fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D264/sign=a954e673b54543a9f11bfdca2a168a7b/f3d3572c11dfa9ec905e6e9060d0f703908fc1c0.jpg**三角函数**

https://gss3.bdstatic.com/-Po3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D264/sign=781793e60db30f24319aeb05fc94d192/4bed2e738bd4b31c849fac3c85d6277f9f2ff891.jpg

https://gss1.bdstatic.com/-vo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D201/sign=4d61146fbe3eb13540c7b0bb971fa8cb/80cb39dbb6fd52661d7d0493a918972bd50736e6.jpg

把加减全部取反就是减法。

**三角形中线**

ma表示角A的中线，mb,mc的把右边的a换成b和c就行了

https://gss2.bdstatic.com/9fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D151/sign=34a32a06afaf2eddd0f14decbc110102/574e9258d109b3dede28b068cfbf6c81800a4c1e.jpg**三角形角平分线**

其中p是半周长

**三角形的高** 随便求，先搞面积，或者正弦余弦定理。

**三角形内切圆半径**

S为面积，c为周长

**三角形外接圆半径** 见正弦定理

【圆】

1. **基本结构体**

**struct** Point{   //点

**double** x,y;

};

**struct** Vector{   //向量

**double** x,y;

};

前置结构：【Point】

**struct** Circle{   //圆

    Point center;

**double** r;

};

前置结构：【Point】

**struct** Rectangle{   //矩形

    Point left; //左下角

    Point right; //右上角

};

前置结构：【Point】

**struct** Segment{   //二维线段

    Point l,r;

};

1. **结构体的基本运算**

【Point】

Vector newVector(Point a,Point b){   //生成向量a -> b

    Vector re;

    re.x = b.x - a.x;

    re.y = b.y - a.y;

**return** re;

}

**bool** operator ==(Point a,Point b){   //点是否相等

**if**(a.x == b.x && a.y == b.y){

**return** **true**;

    }**else**{

**return** **false**;

    }

}

**double** dis(Point a,Point b){    //两点间距离

**return** sqrt(pow(a.x - b.x,2.0) + pow(a.y - b.y,2.0));

}

**double** slope(Point a,Point b){   //两点间斜率

**return** (b.y - a.y) / (b.x - a.x);

}

【Vector】

Vector operator +(Vector a,Vector b){   //向量加法

    Vector re;

    re.x = a.x + b.x;

    re.y = a.y + b.y;

**return** re;

}

Vector operator -(Vector a){   //向量取反

    Vector re;

    re.x = -a.x; re.y = -a.y;

**return** re;

}

Vector operator -(Vector a,Vector b){   //向量减法

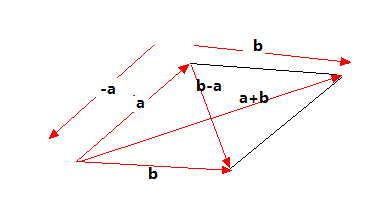
    Vector re;

    re.x = a.x - b.x;

    re.y = a.y - b.y;

**return** re;

}



**double** dj(Vector a, Vector b){    //向量点积

**return** a.x \* b.x + a.y \* b.y;

}

//几何意义，向量a在向量b方向上的投影与向量b的模的乘积

//a·b=（x1\*x2+y1\*y2）=|a||b|·cosθ

**double** cj(Vector a,Vector b){ //向量叉积

**return** a.x \* b.y - b.x \* a.y;

**}**

//几何意义，以a，b为边的平行四边形面积

//axb=x1y2-x2y1=|a||b|·sinθ

**double** vector\_length(Vector a){ //向量模

**return** sqrt(a.x \* a.x + a.y \* a.y);

}

Vector vectorRot(Vector a,**double** radio){   //向量逆时针旋转

    Vector re;

    re.x = a.x \* cos(radio) - a.y \* sin(radio);

    re.y = a.x \* sin(radio) + a.y \* cos(radio);

**return** re;

}

【Circle】

**double** circle\_area(**double** r){ //求圆面积

**return** r\*r\*PI;

}

**double** circle\_line(**double** r){ //求圆周长

**return** 2\*r\*PI;

}

1. **简单功能函数**

前置结构：【Point】

重心：三角形中线交点

Point t\_center\_of\_gravity(Point a,Point b,Point c){   //求平面三角形重心

    Point re;

    re.x = (a.x + b.x + c.x) / 3;

    re.y = (a.y + b.y + c.y) / 3;

**return** re;

}

前置结构：【Rectangle】【Point】

**bool** is\_in\_rectangle(Point a, Rectangle r){  //判断点是否在矩形内.

**if**(a.x >= r.left.x && a.x <= r.right.x){

**if**(a.y >= r.left.y && a.y <= r.right.y){

**return** **true**;

        }

    }

**return** **false**;

}

前置结构：【Rectangle】【Point】

**double** rectangle\_public\_area(Rectangle a, Rectangle b){   //求2个矩形相交面积

**if**(max(a.left.x,b.left.x) >= min(a.right.x,b.right.x) ||

       max(a.left.y,b.left.y) >= min(a.right.y,b.right.y)){

**return** 0;

    }**else**{

**return** (min(a.right.x,b.right.x) - max(a.left.x,b.left.x)) \*

               (min(a.right.y,b.right.y) - max(a.left.y,b.left.y));

    }

}

前置结构：【Point】【Circle】

前置函数：【dis】【circle\_area】

**double** circle\_public\_area(Circle a,Circle b){   //计算2圆相交面积

**double** cdis,sumdis,ans;

    cdis = dis(a.center,b.center);

    sumdis = a.r + b.r;

**if**(cdis >= sumdis){

**return** 0;

    }

**if**(cdis + min(a.r,b.r) <= max(a.r,b.r)){

**return** circle\_area(min(a.r,b.r));

    }

**double** angle1,angle2; //2个圆心角

    angle1 = acos((cdis\*cdis + a.r\*a.r - b.r\*b.r) / (2\*cdis\*a.r));

    angle2 = acos((cdis\*cdis + b.r\*b.r - a.r\*a.r) / (2\*cdis\*b.r));

**double** s1,s2; //2个扇形面积

    s1 = a.r \* a.r \* angle1;

    s2 = b.r \* b.r \* angle2;

**double** s3,s4; //2个三角形面积

    s3 = a.r \* a.r \* sin(angle1) \* cos(angle1);

    s4 = b.r \* b.r \* sin(angle2) \* cos(angle2);

**double** re = s1 + s2 - (s3 + s4);

**return** re;

}

前置结构：【Point】【Segment】【Vector】  
前置函数：【newVector】【cross】

**bool** segment\_intersection(Segment y1, Segment y2){//判断两条线段是否相交

    Vector v1,v2,v3,v4,v5,v6;

**double** d1,d2,d3,d4;

    v1 = newVector(y2.a, y2.b);

    v2 = newVector(y2.a, y1.a);

    v3 = newVector(y2.a, y1.b);

    v4 = newVector(y1.a, y1.b);

    v5 = newVector(y1.a, y2.a);

    v6 = newVector(y1.a, y2.b);

    d1 = cross (v2,v1);

    d2 = cross (v3,v1);

    d3 = cross (v5,v4);

    d4 = cross (v6,v4);

**if**(d1 \* d2 <= 0 && d3 \* d4 <= 0) **return** **true**;

**return** **false**;

}

前置结构【Point】【Vector】

前置函数【newVector】【cross】

**bool** convex(point a[],**int** n){   //判断点是否是凸多边形（点逆时针读入数组）

**int** i;

    Vector u,v;

**for**(i = 1; i <= n; ++i){

**if**(i == n-1){

            u = newVector(a[n-1],a[n]);

            v = newVector(a[n-1],a[1]);

        }**else** **if**(i == n){

            u = newVector(a[n],a[1]);

            v = newVector(a[n],a[2]);

        }**else**{

            u = newVector(a[i],a[i+1]);

            v = newVector(a[i],a[i+2]);

        }

**double** t = cross(u,v);

        //顺时针为double t = cross(v,u);

**if**(t < 0){

**return** **false**;

        }

    }

**return** **true**;

}

1. **复杂功能函数**
2. 【平面最远欧几里得距离点对】

利用旋转卡壳。

1. 【平面最远曼哈顿距离点对】（模板HDU6435）

以二维为例，把绝对值拆开，我们随便选四个的其中一种情况来说明。

(x1 - x2) + (y1 - y2) 移项可得 -> (x1 + y1) + (- x2 - y2)

可以知道有四种情况，可以表示成 (? x1 ? y1) + (-? x2 -? y2)

?表示可以取正负，因此枚举2 ^ k次方（k表示维度）状压就好了。

1. 【平面最近点对】（模板HDU1007）

分治递归或者套个方差优化的KD树

前置结构【Point】

前置函数【dis】

前置数据 Point a[MAXN]

**bool** cmpx(point a,point b){

**return** a.x < b.x;

}

**bool** cmpy(point a,point b){

**return** a.y < b.y;

}

**double** findClosestPair(**int** l,**int** r){

**if**(l + 1 == r) **return** dis(a[l],a[r]);

**if**(l == r) **return** 9999999; //无穷大

**int** mid = (l + r) >> 1;

**double** d1 = min(findClosestPair(l,mid),findClosestPair(mid + 1,r));

    lp.clear(); rp.clear();

**for**(**int** i = l;i <= mid; ++i)

**if**(fabs(a[i].x - a[mid].x) < d1)

            lp.push\_back(a[i]);

**for**(**int** i = mid + 1; i <= r; ++i)

**if**(fabs(a[i].x - a[mid].x) < d1)

            rp.push\_back(a[i]);

    sort(rp.begin(),rp.end(),cmpy);

    sort(lp.begin(),lp.end(),cmpy);

**int** pos = 0;

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

**for**(**int** j = pos; j < rp.size(); ++j){

**if**(lp[i].y - rp[j].y > d1) pos = j + 1;

**if**(fabs(lp[i].y - rp[j].y) < d1) d1 = min(d1,dis(lp[i],rp[j]));

**if**(lp[i].y - rp[j].y < -d1) **break**;

        }

**return** d1;

}

1. **其他**
2. **搜索类**
3. 【A\*搜索】

F(x) = g(x) + h(x) 优先队列按F(x) 排序进行BFS即可

1. 【爬山算法】

请参照下面的模拟退火。

1. 【模拟退火算法】（模板HDU1109）

相比于爬山算法，多了一个可以按概率接受更差解的部分（并不是很理解该怎么用，所以大多数情况下请用上下面的优化继续爬山）。

常用优化方法，多调用几次，起始点尽可能随机，温度要足够高，降温要尽量慢，但是又要保证时间足够，最后就是每个温度下要让状态充分运动（即多次运行）。

参数分别为初始值，初始温度，终温，和降温比例，终温一般为精度小2个数量级。

初始值：随机

初始温度：整个答案范围

终温：比精度小2个数量级

降温比例 0.75 - 0.98

Point SA(Point st,**double** SA\_TMAX,**double** SA\_TMIN,**double** SA\_ratio){

    Point re = st,now;

**double** T = SA\_TMAX,ans,angle;

    ans = INT\_MAX;

**for**(**int** i = 1; i <= n; ++i){ ans = min(ans,dis(re,a[i]));}

**while**(T > SA\_TMIN){

**for**(**int** i = 1; i <= 30; ++i){

            angle = randDouble(0, 2 \* 3.14159,5);

            now.x = re.x + cos(angle) \* T;

            now.y = re.y + sin(angle) \* T;

**if**(now.x < 0 || now.x > x || now.y < 0 || now.y > y){

**continue**;

            }

**double** nowans = INT\_MAX;

**for**(**int** j = 1; j <= n; ++j){

                nowans = min(nowans,dis(now,a[j]));

            }

**if**(nowans > ans){

                re = now; ans = nowans;

            }

            //模拟退火，以一定概率接受较差解，但是用不来，先注释着

//            if(nowans < ans);

//                double delta = nowans - ans;

//                if(exp(delta / T) < randDouble(0,1,4)){

//                    re = now; ans = nowans;

//                }

//            }

        }

        T \*= SA\_ratio;

    }

**return** re;

}

1. 【二分查找】

取(l + r) / 2，l = mid + 1,r = mid停止条件是l < r如果数列是XXOO，最后会停在第一个O这里，如果要找最后一个X，把答案减一就可以了，但是要注意，如果数列全是X，要注意把r多加1，或者处理一下，不然会漏掉答案。

1. 【三分查找】（模板题HDU4355）

要注意midmid取右边那个，由于整除的性质，不然会死循环，整数的话l + 1 < r

ans在r处，求最大值时，小于号变大于号，且ans在l处

**while**(l + EPS < r){

**if**(pd(midmid) < pd(mid)){

        l = mid;

    }**else**{

        r = midmid;

    }

}

1. **题库**
2. **图类问题**
3. **树类问题**

【思维】

O（n）（CF 1073F）：

**给出一棵树，找2个路径，满足2个路径起点终点两两不同，且有公共部分，在公共部分最长的前提下，要2个路径总和尽可能长。**

想到直径，对叶子开始，往上缩，缩到第一个大于等于3个度的点，对于缩入的点，记下最长和次长的分支，然后对缩完的树找直径，直径的端点最长+次长应该尽可能长，找到即可。

O（nlogn）（CF1141G）：

**给出一棵树，对边染色，要求每个点的所有度的颜色都是不一样的，并且可以选择不多于k个点不满足这个要求，求最少色数，和满足的方案。**

考虑没有k，显然直接O（n）BFS染色就好了，可以知道色数等于图内节点的最大度数，现在加上k的条件，明显按度数从大到小取就好了，因此最少色数直接求第k大即可。对于满足方案，对于每条边单独考虑，首先是连接2个都是选择点的边，显然随便什么颜色都可以，那就1号色吧，对于都非选择边的，这部分暴力染色即可，注意有多个连通块，对于最后一种情况的，只需要考虑非选择点用了哪些颜色就好了，对每个点判断一次，把没用过的颜色依次染在剩下这一类边即可。

【树上DP】

**给一颗树，节点上权值，选一点作为根，使得其他点的权值乘以到这个点的距离的总和（记作价值）是最大的（CF1092F）**

树形DP，先1以为根，自低向上，dp[i]表示以i为子树的最大价值，num[i]表示以i为子树的所有节点和（包括i），求完以后，dp[1]当然就是1为根的价值，然后从1做BFS，通过对原dp和num处理一下，就可以求出新的dp[i]，此时的dp[i]就表示以i为根节点的价值，对所有dp[i]取max就好了。

**给一颗树，节点上有取max和取min操作，叶子上可以任意写1 - n（唯一）个数字，问最好写法下，到根以后最大值是多少（CF1153D）**

树形DP，dp[i] = k表示第i个节点的子树下，从叶子到第i个节点后，最大能产生第k大的值。边界条件是叶子为dp[i] = 1，转移条件是，如果当前节点非叶子，且为max，那么dp[now] = 所有叶子里的dp最大值，如果当前节点为min操作，则为所有叶子的dp和，当前节点的答案为：该子树的叶子数 - dp值 + 1

1. **二维线段（区间）类问题**

【最大不重叠子区间】

O（nlogn）模板（HDU 2037）：

贪心，按区间右端点从小到大作为第一关键字排序，左端点从小到大作为第二关键字，然后从左到右选，没跟前面重叠就选。

O（n^2）变形1 （CF 1141F2）：

**求区间内最多有多少不重叠的连续子区间的和相同 ？**

暴力+贪心，枚举所有子区间，和相同的分一组，然后对于每组来说，就是模板问题，取max即可。

【思维】

**给定一列数，每次可以选择相邻2个（相同高度的）同时加1，或者单个加2，问能否让数列最后完全相等？（CF1902D1）**

按奇数偶数分类，变成01序列，然后看连续区间（可以压缩一下），然后弄个栈，每次放进一段，如果都是奇数段和偶数段就合并。然后对栈做while判断，只要栈顶是偶数就弹出即可。

**给定一列数，每次可以选择相邻2个（相同高度的）同时加1，问能否让数列最后完全相等？（CF1902D2）**

同样利用栈，可以先连续相同的数字区间压缩一下，然后每次放入一个数字，while判断看看能不能跟栈顶合并，合并的条件是高度相同，或者是前面是个2的倍数的区间，且高度比当前低，这样判完以后，可能会出现2 1 1这种情况，再扫描一遍栈，如果合并完以后的区间存在多个奇数区间，就不行，或者是存在单个奇数区间，但是这个奇数区间的高度不是原数列里最高的，也不行。

1. **动态规划类问题**
2. **博弈类问题**
3. **数学类问题**
4. 【计数：求n行三角形中等边三角形个数（斜着的也算）】

数列前n项：1,5,15,35,70,126,210……

通项:C(n + 3,4)

1. 【计数：求n行m列网格中正方形个数（斜着的算）】

数列前几项(n >= m)

1;

  2,   6;

  3,  10,  20;

  4,  14,  30,  50;

  5,  18,  40,  70, 105;

  6,  22,  50,  90, 140, 196;

  7,  26,  60, 110, 175, 252, 336;

通项：T(n, m) = m\*(m+1)\*(m+2)\*(2\*n - m + 1)/12

1. 【计数：求n行m列网格中正方形个数（斜着的不算）】

数列前几项(n >= m)

1;

2, 5;

3, 8, 14;

4, 11, 20, 30;

5, 14, 26, 40, 55;

6, 17, 32, 50, 70, 91;

7, 20, 38, 60, 85, 112, 140;

通项：T(n, m) = (m+3\*m\*n+3\*m^2\*n-m^3)/6

1. 【计数：求n行m列网格中矩形个数（斜着的不算）】

(n >= m)

通项：T（n，m） = n\*m\*(n+1)\*(m+1)/4;

1. **字符串类问题**
2. **STL库**
3. **vector（向量，动态数组）**

【求交集,并集,差集】

要注意先对a和b排序，set也可以那么求

a.push\_back(1); a.push\_back(5); a.push\_back(8);

b.push\_back(3); b.push\_back(6); b.push\_back(8);

set\_union(a.begin(),a.end(),b.begin(),b.end(),back\_inserter(c));

输出1 3 5 6 8

注意要先对c进行clear，否则答案会异常。

set\_difference //差集

set\_intersection //交集

set\_symmetric\_difference //对称差集，不同时属于2个集合的元素

1. **map（映射容器）**

【构造】

map <数据类型1（键值）,数据类型2（值）> 容器名;

例：map<int,string> student\_list;

迭代器：map <int,string>::iterator it;

【插入数据】

a.insert(pair<int,string>(1,"xiaolongbao"));

a[1] = "xiaolongbao";

【访问数据】

直接访问下标 cout<<a[1]<<endl;

迭代器:

for(it = a.begin(); it != a.end(); ++it){

cout<<it->second<<endl;

}

注：it->first 为键值，it->second 为值。逆向迭代用rbegin(); rend();

【查找和删除元素】

查找：it = student\_list.find(键值);

删除：a.erase(键值);

1. **set（集合容器）**

【构造】

set <数据类型> 容器名;

例：set<int> student\_set;

迭代器：set <int>::iterator it;

【插入数据】

Student\_set.insert(对应类型的数据);

【遍历元素】 (迭代器返回指针);

for(it = a.begin(); it != a.end(); ++it){

printf("%d ",\*it);

}

【改变排序规则】

//和sort的cmp是相同的，这里是从大到小排

struct cmp{

bool operator()(int x,int y){

return x > y;

}

};

set<int,cmp> s;

【重载相等判断】

struct cmp2{

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

if(a.snd != b.snd){

return a.snd < b.snd;

}

return a.fst < b.fst;

}

};

永远让比较函数对相同元素先返回false,原理就是a == b的话,a < b = false,b < a = false;

判断大小与sort相同

然后排序顺序用<和>

1. **multiset（可重集合）**

【构造】

multiset <数据类型> 容器名;

例：multiset<int> data;

迭代器：multiset <int>::iterator it;

【插入数据】

data.insert(对应类型的数据);

【遍历元素】 (迭代器返回指针);

for(it = a.begin(); it != a.end(); ++it){

printf("%d ",\*it);

}

【删除元素】

it = data.find(p);

data.erase(it);

1. **queue（队列）**

略

1. **priority\_queue（优先队列）**

**struct** cmp{

**bool** operator () (**int** x, **int** y){

**return** x > y;   //小根堆

    }

};

priority\_queue<**int**,vector<**int**>,cmp> line;

1. **string（字符串）**

只能用cin,cout读入输出.

find(); //函数可以找子串，返回首个下标，没找到返回string.npos

substr(1,2); //返回某个子字符串,abcdef返回bcd

data(); //将内容以字符数组形式返回

erase(a,b); 从第a个位置删b个数字，注意不要用到begin

length() 等价于 size() 都是求长度。

1. **stack（栈）**

略

1. **bitset（状态压缩）**

进行一些与二进制相同的操作：

b<<1; //b整体左移

b|=10; //b或 1010

位数也是左边是低位，右边是高位。从0开始。可以直接访问为b[0]

【常用初始化bitset对象的方法】

bitset<n> b; b有n位，每位都为0

b = 8;

【bitset操作】

b.any() //b中是否存在置为1的二进制位？

b.none() //b中不存在置为1的二进制位吗？

b.count() //b中置为1的二进制位的个数

b.size() //b中二进制位的个数

b[pos] //访问b中在pos处的二进制位

b.test(pos) //b中在pos处的二进制位是否为1？

b.set() //把b中所有二进制位都置为1

b.set(pos) //把b中在pos处的二进制位置为1

b.reset() //把b中所有二进制位都置为0

b.reset(pos) //把b中在pos处的二进制位置为0

b.flip() //把b中所有二进制位逐位取反

b.flip(pos) //把b中在pos处的二进制位取反

b.to\_ulong() 返回它转换为unsigned long的结果，如果超出范围则报错  
b.to\_ullong() 返回它转换为unsigned long long的结果，如果超出范围则报错  
b.to\_string() 返回它转换为string的结果

1. **algorithm库**

【二分查找】

**int** main(){

**int** point[10] = {1,3,7,7,9};

**int** tmp = upper\_bound(point, point + 5, 7) - point;//按从小到大，7最多能插入数组point的哪个位置

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

tmp = lower\_bound(point, point + 5, 7) - point;////按从小到大，7最少能插入数组point的哪个位置

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

**return** 0;

}

输出结果

4

2

【删除】

multiset<**int**>::iterator it;

**for** (it = it.begin(); it != it.end(); ) {

**if** (\*it % 2 == 0)

    it.erase(it++);

**else**

    ++it;

}

**查找序列操作**

|  |  |
| --- | --- |
| adjacent\_find | 在一个数组中寻找两个相邻的元素。如果满足条件，就返回这两个相等元素第一个元素的迭代器，不等的，就返回a.end().  例：it = adjacent\_find(a.begin(),a.end(),cmp); |
| count | 返回值等价于给定值的元素的个数  例：int num = count(a.begin(),a.end(),6); |
| count\_if | 返回值满足给定条件的元素的个数  例：int num = count\_if(a.begin(),a.end(),cmp); |
| equal | 返回a是否和b的某一段相等（类似a是b的子串）  例：bool equal(a.begin(),a.end(),b.begin(),cmp) |
| find | 返回a中第一个值等价于给定值的元素的迭代器  例：it = find(a.begin(),a.end(),6); |
| find\_if | 返回a中第一个值满足给定条件的元素  例：it = find\_if(a.begin(),a.end(),cmp); |
| find\_end | 反向查找a中存在的第一个满足关系的子连续序列b（从a中找跟b相等的一段，使得a和b对应位置满足对应关系）  例：it = find\_end(a.begin(),a.end(),b.begin(),b.end(),cmp); |
| find\_first\_of | 查找a中第一个与b中任一元素等价的元素的位置（类似a中找第一个元素在集合b中）  例：it = find\_first\_of(a.begin(),a.end(),b.begin(),b.end()); |
| for\_each | 对a中的每个元素调用指定函数  例：for\_each(a.begin(),a.end(),func); |
| mismatch | 返回a和b第一个不对应满足关系的元素的位置对（返回一个迭代器pair）  例：pair<vector<int>::iterator,vector<int>::iterator> pp;  pp = mismatch(a.begin() ,a.end(),b.begin(),cmp);  pair(n,m) 此时为a的第n个元素和b的第m个不满足关系 |
| search | 正向查找a中存在的第一个满足关系的子连续序列b（正向的find\_end）  例：search(a.begin(),a.end(),b.begin(),b.end(),cmp); |
| search\_n | 返回a中第一处找到连续2个与7有cmp关系的迭代器  例：it = search\_n(a.begin(),a.end(),2,7,cmp); |

**修改内容的序列操作**

|  |  |
| --- | --- |
| copy | 将a中的元素拷贝到新的位置处（如果想拷贝到vector里，得先通过resize()  保证能够容下这些元素，或者back\_inserter（vector名）复制到容器尾部，数组只要够大就行）  例：it = copy(a.begin(),a.end(),c) |
| copy\_backward | 将一个范围中的元素按逆序拷贝到新的位置处（倒着赋值）  例：It = copy\_backward(a.begin(),a.end(),c + 1 + n); |
| fill | 将一个范围的元素赋值为给定值  例：fill(c+1,c+1+n,99) 或 fill(a.begin(),a.end(),99); （a的容量） |
| fill\_n | 将某个位置开始的 n 个元素赋值为给定值  例：fill\_n(c+1,n,99); |
| iter\_swap | 交换两个迭代器（Iterator）指向的元素 |
| remove | 将一个范围中值等价于给定值的元素删除（只是把要删除的数的后面的数都向前移动了一格，复杂度O（n）的那种，size不变，如果要变用earse）返回删完后的end迭代器或指针，减去开头就是长度。  例：it = remove(c+1,c+1+n,6) 或it = remove(a.begin(),a.end(),6);  a.erase(remove\_if(a.begin(),a.end(),cmp),a.end()); // 真的删除所有 |
| remove\_if | 将一个范围中值满足给定条件的元素删除  例：it = remove\_if(a.begin(),a.end(),cmp); |
| remove\_copy | 拷贝一个范围的元素，将其中值等价于给定值的元素删除（事先保证容量够）返回一个指针或者迭代器，表示最后一个元素位置+1（减去开头就是长度）  例：it = remove\_copy\_if(a.begin(),a.end(),back\_inserter(b),6); |
| remove\_copy\_if | 拷贝一个范围的元素，将其中值满足给定条件的元素删除（事先保证容量够）  例：It = remove\_copy\_if(a.begin(),a.end(),back\_inserter(b),cmp); |
| replace | 将一个范围中值等价于给定值的元素赋值为新的值  例：replace(a.begin(),a.end(),6,999); |
| replace\_if | 将一个范围中值满足给定条件的元素赋值为新的值  例：replace\_if(a.begin(),a.end(),cmp,999); |
| replace\_copy | 拷贝一个范围的元素，将一个范围中值等价于给定值的元素赋值为新的值（返回值是一个指针或者迭代器，减去开头就是长度）  例：p = replace\_copy(a.begin(),a.end(),c,6,999); |
| replace\_copy\_if | 拷贝一个范围的元素，将其中值满足给定条件的元素赋值为新的值  例：p = replace\_copy\_if(a.begin(),a.end(),c,cmp,999); |
| reverse | 反转排序指定范围中的元素  例：reverse(a.begin(),a.end()); |
| reverse\_copy | 拷贝指定范围的反转排序结果  例：p = reverse\_copy(a.begin(),a.end(),c); |
| rotate | 循环移动a中的元素且移动至a+2开头(O(n))  例：rotate(a.begin(),a.begin() + 2,a.end()); |
| rotate\_copy | 拷贝指定范围的循环移动结果  例：p = rotate\_copy(a.begin(),a.begin() + 2,a.end(),c); |
| swap | 交换两个对象的值  例：swap(a,b) |
| swap\_ranges | 交换两个范围的元素 |
| transform | 对指定范围中的每个元素调用某个函数以改变元素的值(2 – n 全加1)  int func(int a){  ++a; return a;  }  例：transform(a.begin() + 1,a.end(),a.begin() + 1,func); |
| unique | 例：unique(a.begin(),a.end());  unique函数功能是去除相邻的重复元素，注意是相邻，所以必须先使用sort函数。还有一个容易忽视的特性是它并不真正把重复的元素删除。之所以说比不真正把重复的元素删除，因为unique实际上并没有删除任何元素，而是将无重复的元素复制到序列的前段，从而覆盖相邻的重复元素。unique返回的迭代器指向超出无重复的元素范围末端的下一个位置。   sort(c.begin(), c.end());      T::iterator new\_end = unique(c.begin(), c.end());//"删除"相邻的重复元素      c.erase(new\_end, c.end());//删除(真正的删除)重复的元素 |
| unique\_copy | 拷贝指定范围的唯一化（参考上述的 unique）结果  例：p = unique\_copy(a.begin(),a.end(),c); |

**划分操作**

|  |  |
| --- | --- |
| partition | 将某个范围划分为两组（将满足条件的放在前部分，不满足放在后部分，返回第二部分第一个元素的位置指针或迭代器）  例：it = partition(a.begin(),a.end(),cmp); |
| stable\_partition | 稳定划分，两组元素各维持相对顺序 |

**排序操作**

|  |  |
| --- | --- |
| nth\_element | 部份排序指定范围中的元素，使得范围按给定位置处的元素划分(求第n大很好)  使得第3位的元素排在第3位，比它小的在前面，大的在后面，无序（期望O（n），最坏O（n^2））  例：nth\_element(a.begin(),a.begin() + 2,a.end()); |
| partial\_sort | 部分排序（将前6小的数排序并放好位，后面的无序O（nlogn）主要是找出前n个最值，并对其进行排序）  例：partial\_sort(a.begin(),a.begin()+6,a.end()); |
| partial\_sort\_copy | 拷贝部分排序的结果 |
| sort | 排序 |
| stable\_sort | 稳定排序 |

**二分法查找操作**

|  |  |
| --- | --- |
| binary\_search | 判断范围中是否存在值等价于给定值的元素  例：bool f = binary\_search(a.begin(),a.end(),6) |
| equal\_range | 首先，这个函数只能做用于已经排序的容器，必须按照从小到大进行排序  该函数返回的是一对迭代器，第一个迭代器指向所查找元素的第一次出现的位置，第二个迭代器指向所查找元素最后一次出现位置的后一个位置（lower的位置+upper的位置） |
| lower\_bound | 返回指向范围中第一个值大于或等于给定值的元素的迭代器（第一个该数）  重载运算符和cmp一样  cmp右边是待查找变量,找到第一个false的 |
| upper\_bound | 返回指向范围中第一个值大于给定值的元素的迭代器（-1是最后一个该数）  cmp 左边是待查找变量,找到第一个true的 |

**集合操作**

|  |  |
| --- | --- |
| includes | 测试a中是否包含b的全部元素（有重集合定义下）  例如：1 1 2包含 1 2 不包含 1 2 2  例：bool f = includes(a.begin(),a.end(),b.begin(),b.end()) |
| inplace\_merge | 将2部分有序的合并成一个有序的（实质就是归并排序）  0 – 2 和 2 – n 归并排序（可能不稳定）  例：inplace\_merge(a.begin(),a.begin() + 3,a.end()); |
| merge | 2个序列合并（直接接上去），返回迭代器或指针，表示长度  例：p = merge(a.begin(),a.end(),b.begin(),b.end(),c); |
| set\_difference | 获得两个集合的差集 |
| set\_intersection | 获得两个集合的交集 |
| set\_symmetric\_difference | 获得两个集合的对称差（并集 – 交集） |
| set\_union | 获得两个集合的并集（应事先有序（用set），稳定，O(m+n)）返回迭代器或指针，上面几个方法类似  p = set\_union(o.begin(),o.end(),t.begin(),t.end(),c); |

**堆操作**

|  |  |
| --- | --- |
| make\_heap | 用给定范围构造出一个堆（cmp是和sort反的）  例：make\_heap(a.begin(),a.end(),cmp); |
| pop\_heap | 从一个堆中删除最大的元素，把它放在结尾（cmp是和sort反的）  例：pop\_heap(a.begin(),a.end()); |
| push\_heap | 向堆中增加一个元素（cmp是和sort反的）  例：scanf("%d",&num); a.push\_back(num); push\_heap(a.begin(),a.end()，cmp); |
| sort\_heap | 将满足堆结构的范围排序（必须先make\_heap）  sort\_heap(a.begin(),a.end(),cmp); |
| pop\_back | 删除末尾元素（真删除），pop后配合该函数可以做到真删除堆顶  例：a.pop\_back(); |

**最大/最小操作**

|  |  |
| --- | --- |
| lexicographical\_compare | 比较两个序列的字典序  if(lexicographical\_compare(a.begin(),a.end(),b.begin(),b.end())){  printf("a小于b");  }else{  printf("a大于b");  } |
| max | 返回两个元素中值最大的元素 |
| max\_element | 返回给定范围中值最大的元素（返回指向最大值的指针或者迭代器）（cmp是反的）  return a < b //找最大值  例：p = max\_element(c + 1,c + 1 + n,cmp); |
| min | 返回两个元素中值最小的元素 |
| min\_element | 返回给定范围中值最小的元素（cmp是正的） |
| next\_permutation | 返回给定范围中的元素组成的下一个按字典序的排列（复杂度O(n)）  以运算符 < 比较，可以加cmp  bool cmp(int a,int b){ //表示第一个序列所有相邻数字满足前者大于后者  return a > b;  }  4 3 2 1 –> 1 2 3 4  例：next\_permutation(a.begin(),a.end(),cmp) |
| prev\_permutation | 返回给定范围中的元素组成的上一个按字典序的排列 |

1. **其他**
2. 【java大整数和大实数】

**import** java.math.BigInteger;

**import** java.math.BigDecimal;

**bitLength**，求大整数二进制长度

大实数保留小数规则

a = a.setScale(3, RoundingMode.HALF\_UP);    //保留3位小数，且四舍五入

ROUND\_CEILING    //向正无穷方向舍入

ROUND\_DOWN    //向零方向舍入

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\_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\_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\_UNNECESSARY    //计算结果是精确的，不需要舍入模式

ROUND\_UP    //向远离0的方向舍入